Брайан Гетц
Тим Пайерлс, Джошуа Блох,
Джозеф Боубер, Дэвид Холмс,
Даг Ли

JAVA CONCURRENCY НА ПРАКТИКЕ

Java Concurrency in Practice

Brian Goetz
with
Tim Peierls
Joshua Bloch
Joseph Bowbeer
David Holmes
and Doug Lea

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Брайан Гетц
Тим Пайерлс, Джошуа Блох,
Джозеф Боубер, Дэвид Холмс,
Даг Ли

JAVA CONCURRENCY НА ПРАКТИКЕ

ПИТЕР®
Санкт-Петербург · Москва · Екатеринбург · Воронеж
Нижний Новгород · Ростов-на-Дону
Самара · Минск
2020

ББК 32.973.2-018-02
УДК 004.4
Д40

Гетц Брайан, Пайерлс Тим, Блох Джошуа, Боубер Джозеф, Холмс Дэвид, Ли Даг

Д40 Java Concurrency на практике. — СПб.: Питер, 2020. — 464 с.: ил. — (Серия «Для профессионалов»).

ISBN 978-5-4461-1314-9

Потоки являются фундаментальной частью платформы Java. Многоядерные процессоры — это обыденная реальность, а эффективное использование параллелизма стало необходимым для создания любого высокопроизводительного приложения. Улучшенная виртуальная машина Java, поддержка высокопроизводительных классов и богатый набор строительных блоков для задач распараллеливания стали в свое время прорывом в разработке параллельных приложений. В «Java Concurrency на практике» сами создатели прорывной технологии объясняют не только принципы работы, но и рассказывают о паттернах проектирования. Легко создать конкурентную программу, которая вроде бы будет работать. Однако разработка, тестирование и отладка многопоточных программ доставляют много проблем. Код перестает работать именно тогда, как это важнее всего: при большой нагрузке. В «Java Concurrency на практике» вы найдете как теорию, так и конкретные методы создания надежных, масштабируемых и поддерживаемых параллельных приложений. Авторы не предлагают перечень API и механизмов параллелизма, они знакомят с правилами проектирования, паттернами и моделями, которые не зависят от версии Java и на протяжении многих лет остаются актуальными и эффективными.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.973.2-018-02
УДК 004.4

ISBN 978-0321349606 англ.
978-5-4461-1314-9

Оглавление

Отзывы

Мне действительно повезло, что я работал с фантастической командой над проектом и реализацией функционала конкурентности, добавленного в платформу Java, в Java 5.0 и Java 6. Теперь эта же команда представляет лучшее объяснение нового функционала и конкурентности в целом, причем не только для продвинутых пользователей, но и для всех программистов, работающих с Java.

Мартин Бухгольц,
JDK Concurrency Czar, Sun Microsystems

В течение последних тридцати лет производительность компьютера определялась законом Мура; отныне она будет определяться законом Амдала. Написание кода, эффективно использующего многочисленные процессоры, — сложная задача. Но книга «Java Concurrency на практике» предоставляет вам концепции и технические решения для создания безопасных и масштабируемых программ как сегодняшних, так и завтрашних систем.

Дорон Раджван,
научный сотрудник, Intel Corp

Эта книга поможет написать — а также спроектировать, отладить, сопроводить, проанализировать — многопоточные программы на Java. Если вам когда-либо приходилось синхронизировать метод, но вы не представляли зачем, то вам просто необходимо прочитать эту книгу.

Тед Ньюард,
автор книги Effective Enterprise Java
(«Эффективный язык Java для предприятий»)

Брайан подробно разъяснил фундаментальные проблемы и сложности конкурентности. Эта книга обязательна к прочтению всем, кто использует потоки и заинтересован в производительности.

Кирк Пеппердин,
директор по исследованиям и разработкам
JavaPerformanceTuning.com

Эта книга исчерпывает очень глубокую и тонкую тему в ясной и краткой форме, что делает ее идеальным руководством по конкурентности в Java. Каждая страница заполнена задачами (и их решениями!), с которыми программисты сталкиваются каждый день. Тема конкурентности сегодня очень актуальна, поскольку продолжать наращивать частоту работы одного ядра по закону Мура становится все труднее и набирают популярность многоядерные процессоры.

Доктор Клифф Клик,
старший инженер по программному обеспечению,
Azul Systems

Думаю, я тот человек, который написал больше взаимных блокировок потоков и допустил больше промашек в их синхронизации, чем любой другой программист. И считаю, что книга Брайана прекрасно справляется с этой сложной темой благодаря практическому подходу. Я рекомендую ее всем читателям «Бюллетеня для специалистов по Java», потому что она интересна, полезна и актуальна.

Доктор Хайнц Кабуц,
«Бюллетень для специалистов по Java»

Я сосредоточил свою карьеру на упрощении решения задач и вижу, насколько качественно в этой книге разобраны сложные вопросы конкурентности. Книга является революционной по своему подходу, плавной и легкой по стилю, и своевременной — ей суждено стать знаковой.

Брюс Тейт,
автор книги «Горький вкус Java»[1]

[1] *Тейт Б.* Горький вкус Java / Пер. с англ. Е. Матвеева. — СПб.: Питер, 2003. — 332 с.

Книга «Java Concurrency на практике» является бесценным сборником ноу-хау многопоточной обработки. Я нашел чтение этой книги интеллектуально захватывающим, отчасти потому, что она не только является отличным введением в многопоточное API в Java, но и тщательно и доступно фиксирует экспертные знания по многопоточной обработке, которые не так легко найти в других источниках.

Билл Веннерс,
автор книги Inside the Java Virtual Machine
(«Внутри виртуальной машины Java»)

Посвящается Джессике

Листинги

Предисловие

Настало время, когда многоядерные процессоры становятся достаточно недорогими для установки в настольных системах среднего уровня. И не случайно разработчики отмечают все больше сообщений об ошибках, связанных с многопоточностью в своих проектах. На сайте разработчиков *NetBeans* один сопроводитель описал, как на класс было наложено 14 патчей для исправления проблем многопоточной обработки. Дион Алмаер, бывший редактор *TheServerSide*, в своем блоге сказал (после болезненного сеанса отладки, который в итоге выявил ошибку многопоточности), что большинство программ Java изобилуют ошибками *конкурентности* (concurrent)[1], и порядок работы программ организуется «случайным образом».

Разработка, тестирование и отладка многопоточных программ бывает чрезвычайно сложной, поскольку ошибки конкурентности трудно спрогнозировать — часто они проявляются в неподходящий момент в условиях большой нагрузки.

Одной из сложностей создания конкурентных программ на Java является несовпадение между тем, что предлагает платформа, и потребностями разработчиков. Язык предоставляет низкоуровневые *механизмы* (mechanisms), такие как синхронизация и ожидание по условию, но они должны использоваться согласованно для реализации протоколов или *политик* (policies) уровня приложения. Без таких политик программы внешне работают, но остаются неисправными. Большинство книг по конкурентности малополезны, поскольку сосредоточены на низкоуровневых механизмах и API, а не на политиках и шаблонах уровня проекта.

---

[1] Если исходить из содержания книги, то ее следовало бы перевести как «Многопоточное программирование в Java на практике», но в русском языке уже появилось слово «конкурентность», которое означает одновременность процессов на логическом уровне (на физическом уровне за счет аппаратной поддержки есть термин «параллелизм»). — *Примеч. науч. ред.*

Язык предоставляет новые компоненты более высокого уровня и дополнительные низкоуровневые механизмы, которые облегчают построение конкурентных приложений как новичкам, так и экспертам. Авторы данной книги — члены экспертной группы JCP, создавшей эти механизмы, — представили описание нового функционала, базовые паттерны проектирования, включенные в платформенные библиотеки, и ожидаемые сценарии их использования.

Наша цель — дать читателям набор правил проектирования и ментальных моделей, которые сделают легче и интереснее создание правильных, производительных конкурентных классов и приложений на Java.

Мы надеемся, что книга вам понравится.

Брайан Гетц
Уиллистон, штат Вермонт,
Март 2006

Как пользоваться книгой

Мы представляем *упрощенный* набор правил для написания конкурентных программ. Эксперты могут посмотреть на эти правила и сказать: «Хм, это не совсем верно: класс C — потокобезопасный, несмотря на то что он нарушает правило R». Конечно, можно писать корректные программы, которые нарушают наши правила, что потребует глубокого понимания деталей модели памяти Java. Но мы хотим, чтобы разработчики смогли обойтись *без* освоения этих деталей.

Книга «Java Concurrency на практике» не является введением в конкурентность — для этого смотрите вводную главу в таких изданиях, как «Язык программирования Java»[1]. Также она не является энциклопедическим справочником по конкурентности — для этого читайте *Concurrent Programming in Java* («Конкурентное программирование на Java») Дага Ли. Она предлагает практические правила проектирования, которые помогут разработчикам в сложном процессе создания безопасных и производительных конкурентных классов. В необходимых случаях мы

[1] *Арнолд К., Гослинг Д., Холмс Д.* Язык программирования Java / *Пер. с англ.* — М: Вильямс, 2001. — 623 с.: ил.

ссылаемся на соответствующие разделы книг *The Java Programming Language* («Язык программирования Java»), *Concurrent Programming in Java* («Конкурентное программирование на Java») и *The Java Language Specification* («Спецификация языка Java») Джеймса Гослинга и «Java. Эффективное программирование»[1] с использованием условных обозначений [JPL n.m], [CPJ n.m], [JLS n.m] и [EJ пункт n].

После введения (главы 1) книга делится на четыре части:

Основы. Часть I (главы 2–5) сконцентрирована на основных понятиях конкурентности и потокобезопасности, а также посвящена составлению потокобезопасных классов из конкурентных строительных блоков, предоставляемых библиотекой классов. Часть завершается памяткой, обобщающей наиболее важные правила.

Главы 2 («Потокобезопасность») и 3 («Совместное использование объектов») образуют основу данной книги. Здесь приводятся почти все правила по предотвращению угроз конкурентности, созданию потокобезопасных классов и верификации потокобезопасности. Даже если вы предпочитаете теории практику, обязательно прочтите эти главы, прежде чем приступите к написанию конкурентного кода!

Глава 4 («Компоновка объектов») охватывает технические решения для формирования крупных потокобезопасных классов. В главе 5 («Строительные блоки») говорится о потокобезопасных коллекциях и синхронизаторах из платформенных библиотек.

Структурирование конкурентных приложений. В части II (главы 6–9) описывается эксплуатация потоков, направленная на повышение пропускной способности и отзывчивости конкурентных приложений. Глава 6 («Выполнение задач») посвящена идентификации параллелизуемых задач и их выполнению в рамках структуры выполнения задач. Глава 7 («Отмена и выключение») описывает технические решения, которые заставляют задачи и потоки завершаться по требованию, ведь способность конкурентного приложения справляться с отменой и выключением определяет его качество. В главе 8 («Применение пулов потоков») рассматривается более продвинутый функционал структуры выполнения

[1] *Блох Д.* Java. Эффективное программирование / *Пер. с англ.* И. Красикова. — 3-е изд. — М.; СПб.: Диалектика, 2018. — 457 с.

задач. Глава 9 («Приложения с GUI») посвящена техническим решениям для повышения отзывчивости в однопоточных подсистемах.

Жизнеспособность, производительность и тестирование. Часть III (главы 10–12) касается обеспечения потребностей разработчиков с приемлемой производительностью. Глава 10 («Предотвращение сбоев жизнеспособности») описывает, как избежать сбоев в жизнеспособности, которые могут помешать продвижению программ. Глава 11 («Производительность и масштабирование») посвящена техническим решениям, повышающим производительность и масштабируемость конкурентного кода. Глава 12 («Тестирование конкурентных программ») описывает способы тестирования правильности и производительности конкурентного кода.

Продвинутые темы. Часть IV (главы 13–16) охватывает темы, которые могут быть интересны только опытным разработчикам: явные замки, атомарные переменные, неблокирующие алгоритмы и разработку собственных синхронизаторов.

Примеры исходного кода

Общие понятия из книги применимы к версиям от Java 5.0 и более поздним версиям, и даже к средам, отличным от Java.

Примеры кода были сжаты для того, чтобы осветить его важные части. Полные версии примеров кода, а также дополнительные примеры и исправления представлены на сайте книги: http://www.javaconcurrencyinpractice.com.

Мы использовали три вида примеров кода: хороший, не очень хороший и плохой. Хорошие примеры иллюстрируют технические решения, которые следует воспроизводить. Решения из плохих примеров воспроизводить *не* следует, они отмечены значком «Мистер Юк»[1] (см. листинг 1). Не очень хорошие примеры показывают технические решения, которые не *обязательно* ошибочные, но рискованные, они украшены значком «Мистер мог быть и посчастливее», как в листинге 2.

[1] «Мистер Юк» является зарегистрированным товарным знаком детской больницы Питтсбурга и приводится в книге с разрешения.

Листинг 1. Плохой способ сортировки списка. *Так делать не следует*

```
public <T extends Comparable<? super T>> void sort(List<T> list) {
    // Никогда не возвращает правильный ответ!
    System.exit(0);
}
```

Плохие примеры необходимы для иллюстрации распространенных подводных камней и, что более важно, они демонстрируют нарушения в потокобезопасности для дальнейшего анализа работы программы.

Листинг 2. Не самый оптимальный способ сортировки списка

```
public <T extends Comparable<? super T>> void sort(List<T> list) {
    for (int i=0; i<1000000; i++)
        doNothing();
    Collections.sort(list);
}
```

Благодарности

Эта книга выросла из процесса разработки пакета java.util.concurrent, созданного Java Community Process JSR 166 для включения в Java 5.0. Вклад в JSR 166 внесли многие разработчики. Мы благодарим Мартина Букхольца за всю работу, связанную с получением кода в JDK, и всех читателей рассылки, заинтересованных в конкурентности, которые внесли свои предложения и отзывы по черновому варианту API.

Эта книга была улучшена предложениями небольшой группы рецензентов, советников, вдохновителей и диванных критиков. Мы хотели бы поблагодарить Диона Алмаера, Трейси Бялик, Синди Блох, Мартина Букхольца, Пола Кристмана, Клиффа Клика, Стюарта Холлоуэя, Дэвида Ховмейера, Джейсона Хантера, Майкла Хантера, Джереми Хилтона, Хайнца Кабуца,

Роберта Кухара, Рамниваса Ладдэда, Джареда Леви, Николь Льюис, Виктора Лучангко, Джереми Мэнсона, Пола Мартина, Берна Массингилла, Майкла Маурера, Теда Ньюарда, Кирка Пеппердина, Билла Пью, Сэма Пуллара, Расса Руфера, Билла Шерера, Джеффри Сигала, Брюса Тейта, Джил Тене, Пола Тайма и представителей Группы паттернов Кремниевой долины, которые в процессе многочисленных бесед внесли предложения, позволившие сделать книгу лучше.

Мы особенно благодарны Клиффу Биффлу, Бэрри Хэйсу, Дэвиду Курцинеку, Анжелике Лангер, Дорону Раджвану и Биллу Веннерсу, которые подробно проанализировали рукопись, нашли недочеты в примерах кода и предложили усовершенствования.

Мы благодарим Катрин Эйвери за отличную работу по редактированию, Розмари Симпсон, а также Ами Дьюар — за иллюстрации.

Спасибо всей команде издательства *Addison-Wesley*. Энн Селлерс запустила проект, Грег Доэнч довел его до завершения, а Элизабет Райан провела его через производственный процесс.

Мы хотели бы также поблагодарить тысячи инженеров, разработавших программное обеспечение, использованное для создания книги: TEX LTEX, Adobe Acrobat, pic, grap, Adobe Illustrator, Perl, Apache Ant, IntelliJ IDEA, GNU emacs, Subversion, TortoiseSVN, и конечно, платформу Java и библиотеки классов.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

1 Введение

Писать программы трудно, а конкурентные программы — еще труднее. Зачем тогда вообще разбираться с конкурентностью? Потоки выполнения являются необходимым функционалом языка Java, поскольку могут упростить разработку систем, превращая сложный асинхронный код в более простой и прямолинейный. Кроме того, потоки — это самый лучший способ задействовать сразу несколько вычислительных мощностей многопроцессорных систем. И по мере увеличения числа процессоров эффективное применение конкурентности будет становиться все важнее.

1.1. Кратчайшая история конкурентности

Первые компьютеры работали без операционных систем. Они выполняли единственную программу от начала до конца, которая имела прямой доступ ко всем ресурсам машины. Создание и выполнение таких программ требовало дорогостоящих и дефицитных ресурсов.

Со временем операционные системы эволюционировали, позволив выполнять несколько программ одновременно в рамках *процессов* (processes) — изолированных, независимо выполняемых программ, которые пользуются ресурсами операционной системы, такими как память, дескрипторы файлов и учетные данные системы безопасности. При необходимости процессы могут взаимодействовать друг с другом с помощью коммуникационных механизмов: сокетов, обработчиков сигналов, совместной памяти, семафоров и файлов.

Перемены произошли по нескольким причинам:

Задействованность ресурсов. Время ожидания внешних операций (ввода и вывода), когда программы не делают никакой полезной работы, удобно использовать для запуска других программ.

Справедливость. Разные пользователи и программы могут иметь одинаковые права на машинные ресурсы и должны использовать компьютер совместно, не дожидаясь полного завершения одной программы перед запуском другой.

Удобство. Зачастую проще написать несколько программ, каждая из которых выполняет одну задачу, и координировать их друг с другом, чем написать единственную программу, выполняющую все задачи.

В ранних системах с режимом разделения времени каждый процесс напоминал работу компьютера фон Неймана: он хранил в памяти инструкции и данные, которые последовательно выполнял в соответствии с семантикой машинного языка, взаимодействуя с внешним миром через набор примитивов ввода-вывода. Для каждой выполняемой инструкции была четко определена следующая, и управление соответствовало правилам набора инструкций. Почти все широко используемые сегодня языки программирования следуют этой модели, определяющей их спецификацию.

Последовательное программирование интуитивно понятно: человек решает одну задачу за один раз, выстраивает очередность действий. Встать с кровати, накинуть халат, заварить чай. Как и в программировании, любое из этих реальных действий (например заварить чай) является абстракцией для другой последовательности: открыть шкаф, выбрать чай, насыпать его в чайник, налить необходимое количество воды, поставить чайник на плиту, включить плиту, дождаться, когда закипит вода, и т. д. Некоторые шаги предусматривают *асинхронность* (asynchrony) — пока вода нагревается, вы можете, например, запустить тостер. Производители чайников и тостеров знают, что их изделия часто используются асинхронно, поэтому снабжают их звуковым сигналом завершения задачи. Нахождение баланса между последовательностью и асинхронностью является критерием эффективности как для человека, так и для программы.

Проблемы задействованности ресурсов, справедливости и удобства способствовали развитию не только процессов, но и *потоков выполнения* (threads), которые позволяют многочисленным программным потокам данных сосуществовать внутри процесса. Совместно используя

общепроцессные ресурсы, каждый поток выполнения имеет свой программный счетчик, стек и локальные переменные. Потоки обеспечивают естественную декомпозицию для задействования аппаратного параллелизма в многопроцессорных системах и внутри одной программы могут действовать одновременно в нескольких процессорах.

Потоки иногда называют *облегченными процессами* (lightweight processes), и в большинстве современных операционных систем именно они считаются основными единицами планирования. При отсутствии явной координации потоки выполняются одновременно и асинхронно по отношению друг к другу. Имея равный доступ к адресному пространству памяти процесса, все потоки используют одинаковые переменные и объекты из одной кучи, благодаря чему работают точнее межпроцессных механизмов. Но без явной синхронизации, обеспечивающей координацию доступа к совместным данным, потоки могут изменять переменные, которые уже используются другими потоками, с непредсказуемым результатом.

1.2. Преимущества потоков

При надлежащем использовании потоки позволяют снизить стоимость разработки и обслуживания, а также повысить производительность сложных приложений. Они превращают асинхронные потоки информации в преимущественно последовательные, а запутанный код — в прямолинейный.

В приложениях с GUI потоки повышают отзывчивость пользовательского интерфейса, а в серверных приложениях улучшают использование ресурсов и увеличивают пропускную способность. Они упрощают реализацию виртуальной машины Java, поскольку сборщик мусора, как и большинство приложений на Java, работает с потоками.

1.2.1. Задействование множества процессоров

Раньше многопроцессорные системы находились только в крупных центрах хранения и обработки данных и научно-вычислительных центрах. Сегодня они стали дешевле и доступнее: даже низкоуровневые серверные и среднеуровневые настольные системы часто имеют несколько процессоров. Эта тенденция будет только ускоряться, поскольку становится все труднее масштабировать тактовые частоты, и производители процессоров

начинают размещать больше процессорных ядер на один чип. Все крупные производители чипов переживают переходный период.

Программа с одним потоком может работать не более чем на одном процессоре один раз. Это означает, что в двухпроцессорной системе однопоточная программа отказывается от доступа к половине процессорных ресурсов, а в 100-процессорной системе — от доступа к 99 % ресурсов. Очевидно, что более эффективно использовать процессорные ресурсы смогут многопоточные программы.

Причем использование нескольких потоков позволяет повысить пропускную способность однопроцессорных систем. В многопоточной программе поток может продолжать работать в то время, как другой поток ожидает завершения внешней операции, что позволяет приложению продвигаться вперед. (Представьте чтение газеты во время ожидания закипания воды, а не после него.)

1.2.2. Простота моделирования

Часто легче управлять временем, когда нужно выполнить только один тип задачи (исправить вот эти двенадцать багов), а не несколько (исправить баги, провести собеседование с несколькими кандидатами на должность системного администратора, завершить оценку результативности команды и создать слайды для презентации). Имея перед собой один тип задачи, вы можете начать с верха стопки и продолжать работать до тех пор, пока стопка (или вы) не будет исчерпана. Вам не нужно тратить энергию, выясняя, над чем работать дальше. Согласитесь, управление приоритетами, предельными сроками и переход от задачи к задаче несут издержки.

Программа, которая последовательно обрабатывает один тип задач, проще в написании, менее подвержена ошибкам и легче тестируется, чем та, которая управляет сразу несколькими задачами разного типа. Поэтому сложную асинхронную работу мы разобьем на несколько более простых синхронных потоков данных, которые попадут в отдельные потоки выполнения, взаимодействуя друг с другом только в определенных точках синхронизации.

По такому принципу работают структуры, такие как сервлеты. Они обрабатывают сведения об управлении запросами, создают потоки и балансируют нагрузку, направляя части обработки запросов к соответствующим

компонентам приложения на соответствующих этапах процесса. Метод `service` сервлета может обрабатывать запрос как однопоточная программа, упрощая разработку компонентов структуры.

1.2.3. Упрощенная обработка асинхронных событий

Разработка серверного приложения, принимающего сокетные соединения от многочисленных удаленных клиентов, станет проще, если каждому соединению выделить собственный поток и разрешить использовать синхронный ввод-вывод.

Если приложение начинает читать из сокета при отсутствии данных, то операция `read` блокирует обработку до тех пор, пока некоторые данные не станут доступны. В однопоточном приложении останавливается обработка не только соответствующего запроса, но и всех запросов, пока один поток заблокирован. Для того чтобы избежать этой проблемы, однопоточные серверные приложения используют неблокирующий ввод-вывод, который намного сложнее и более подвержен ошибкам, чем синхронный.

Исторически сложилось так, что операционные системы устанавливали ограничения на количество потоков, которые мог создавать процесс, — допускалось всего несколько сотен (или даже меньше). В результате получили развитие механизмы мультиплексированного ввода-вывода, такие как системные вызовы Unix `select` и `poll`. В целях доступа к этим механизмам библиотеки классов Java приобрели набор пакетов java.nio. Однако сегодня операционные системы поддерживают увеличение числа потоков с помощью модели «один поток в расчете на клиента»[1].

1.2.4. Более отзывчивые пользовательские интерфейсы

Приложения с графическим пользовательским интерфейсом (GUI) раньше были однопоточными, и приходилось либо часто опрашивать весь код на наличие входных событий (запутанное и утомительное занятие), либо

[1] Пакет NPTL threads, теперь являющийся частью большинства дистрибутивов Linux, был разработан для поддержки сотен тысяч потоков. Неблокирующий ввод-вывод имеет свои преимущества, но более совершенная поддержка потоков операционной системой означает, что сокращается количество ситуаций, для которых это *необходимо*.

выполнять весь код приложения косвенно, через главный событийный цикл. Если код, вызываемый из главного событийного цикла, выполнялся слишком долго, пользовательский интерфейс «замораживался».

Современные структуры GUI, такие как инструментальный пакет AWT и платформа Swing, заменяют главный событийный цикл *потоком диспетчеризации событий* (event dispatch thread, EDT). Когда происходит событие пользовательского интерфейса, такое как нажатие кнопки, в потоке событий вызываются соответствующие обработчики. Большинство структур GUI являются однопоточными подсистемами, поэтому главный событийный цикл в них по-прежнему присутствует, но он выполняется в собственном потоке под управлением инструмента GUI, а не приложения.

Если в потоке событий выполняются только скоротечные задачи, то интерфейс остается отзывчивым. Однако во время выполнения длительной задачи любое другое действие будет обрабатываться в потоке событий с задержкой, пользовательский интерфейс перестанет откликаться и не предоставит возможность отменить виновную задачу, даже если отобразит кнопку отмены! Поэтому длительной задаче необходим отдельный поток выполнения.

1.3. Риски для потоков

Благодаря языковой и библиотечной поддержке и формальной кроссплатформенной модели памяти Java упрощается разработка *конкурентных* приложений, работающих по принципу «написано однажды, работает везде». Вместе с этим расширяются возможности разработчиков. Но востребованность многопоточных сред требует понимания вопросов потокобезопасности.

1.3.1. Угрозы безопасности

Вопреки ожиданиям потокобезопасность может быть едва различимой, поскольку при отсутствии достаточной синхронизации порядок операций в потоках бывает непредсказуемым. Класс `UnsafeSequence` в листинге 1.1, который должен генерировать последовательность уникальных целочисленных значений, иллюстрирует, как перемежение действий в многочисленных потоках может привести к нежелательным результатам. В однопоточной среде он ведет себя правильно, а в многопоточной — нет.

Листинг 1.1. Непотокобезопасный генератор последовательности

```
@NotThreadSafe
public class UnsafeSequence {
    private int value;

    /** Возвращает уникальное значение. */
    public int getNext() {
        return value++;
    }
}
```

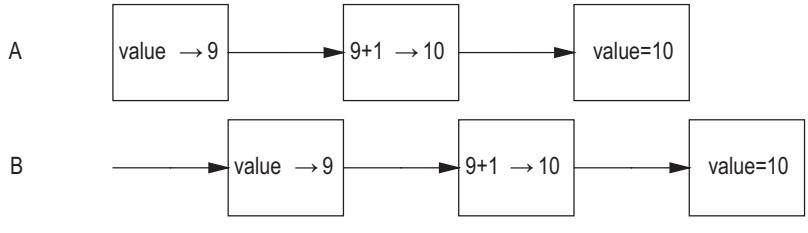

Рис. 1.1. Неудачное выполнение UnsafeSequence.nextValue

При неудачной временно́й координации два потока могут вызвать метод `getNext` и получить *неуникальные значения* (см. рис. 1.1). Операция приращения `nextvalue++` состоит из трех отдельных операций: чтения значения, добавления в него единицы и записи нового значения. Поскольку операции в потоках произвольно перемежаются, два потока могут прочитать значение в одно и то же время, добавить в него единицу и вернуть один и тот же порядковый номер.

На рис. 1.1 время течет слева направо и каждый ряд представляет действия другого потока. Подобные схемы перемежения обычно изображают худший вариант[1] и предназначены для того, чтобы показывать, как опасно надеяться на четкий порядок.

[1] На самом деле, как мы увидим в главе 3, худший вариант — не самый худший. Виной тому возможность переупорядочивания.

В классе `UnsafeSequence` используется нестандартная аннотация: `@NotThreadSafe`. Это одна из нескольких настраиваемых аннотаций, используемых в этой книге для документирования свойств конкурентности у классов и членов классов. (Другими используемыми таким образом аннотациями уровня классов будут `@ThreadSafe` и `@Immutable`; подробности см. в приложении А.) Если класс аннотирован `@ThreadSafe`, то его можно использовать в многопоточной среде: сопроводители получат уведомление о том, что он обеспечивает и сохраняет потокобезопасность, а инструменты анализа программного обеспечения идентифицируют возможные ошибки кодирования.

Класс `UnsafeSequence` иллюстрирует распространенную угрозу для конкурентности, именуемую *состоянием гонки* (race condition)[1]. При такой ситуации возврат уникального значения при вызове из разных потоков посредством `nextValue` зависит от того, как рабочая среда перемежает операции.

Поскольку потоки совместно используют одно и то же адресное пространство памяти и работают конкурентно, они могут обращаться к переменным либо изменять переменные, которые уже используются другими потоками. В этом заключается их удобство при совместной работе с данными. Но это качество несет за собой значительный риск: потоки могут быть сбиты с толку, работая с данными, которые изменяются неожиданно. Для того чтобы поведение многопоточной программы было предсказуемым, доступ к совместным переменным должен быть скоординирован так, чтобы потоки не мешали друг другу. К счастью, Java обеспечивает механизмы синхронизации.

Класс `UnsafeSequence` можно исправить, сделав метод `getNext` синхронизированным, как показано в классе `Sequence` в листинге 1.2[2], тем самым предотвратив неудачное взаимодействие, показанное на рис. 1.1. (Подробности описаны в главах 2 и 3.)

---

[1] *Состояние гонки* — это недостаток в системе, характеризующийся разным результатом в зависимости от того, в каком порядке действуют игроки системы. — *Примеч. пер.*

[2] Аннотация `@GuardedBy` описана в разделе 2.4. Она документирует *политику синхронизации* для класса `Sequence`.

Листинг 1.2. Потокобезопасный генератор последовательности

```
@ThreadSafe
public class Sequence {
    @GuardedBy("this") private int nextValue;

    public synchronized int getNext() {
        return nextValue++;
    }
}
```

При отсутствии синхронизации компилятору, аппаратному обеспечению и рабочей среде разрешаются допущения в вопросах временной координации и упорядочивания действий, такие как кэширование переменных в регистрах или локальных для процессора кэшах, где действия временно (или даже постоянно) не видимы для других потоков. Эти хитрости помогают повысить производительность и, как правило, желательны, но они ложатся бременем на разработчика, в чьи обязанности входит четкая идентификация того, где данные используются между потоками совместно. (В главе 16 описано, какое именно упорядочивание гарантирует JVM и как на него влияет синхронизация, но, следуя правилам из глав 2 и 3, вы сможете игнорировать эти детали.)

1.3.2. Сбои жизнеспособности

Потокобезопасность необходима как многопоточным программам, так и однопоточным, но использование множества потоков создает для безопасности дополнительные сбои, например новые формы *сбоев в жизнеспособности* (liveness failure).

Безопасность (safety) подразумевает отсутствие плохих состояний, а жизнеспособность обозначает их преодоление с хорошим исходом. Сбой в жизнеспособности происходит, когда программа не способна продвигаться вперед, например, если вошла в бесконечный цикл. Если поток *A* ожидает ресурс, которым поток *B* владеет эксклюзивно, и *B* никогда его не освободит, то *A* будет ждать вечно. Глава 10 описывает различные формы сбоев в жизнеспособности и способы их предотвращения, включая взаимную блокировку (раздел 10.1), голодание (раздел 10.3.1) и активную блокировку (раздел 10.3.3). Ошибки конкурентности, вызывающие сбои в жизнеспособности, могут быть неуловимыми, поскольку зависят от

относительной временно́й координации событий в разных потоках, и поэтому не всегда проявляются при разработке или тестировании.

1.3.3. Угрозы производительности

С жизнеспособностью связана *производительность* (performance). Если жизнеспособность подразумевает хороший исход, то производительность определяет, насколько быстро он достигнут. Производительность характеризуют показатели периода обслуживания, отзывчивости, пропускной способности, потребления ресурсов или масштабируемости. Многопоточные программы подвержены всем рискам для производительности, присущим однопоточным программам, и хотя могут обеспечить прирост производительности, провоцируют некоторые издержки.

К ним относятся *контекстные переключения* (context switches) — приостановки работы активных потоков для работы других потоков. Они сохраняют и восстанавливают контекст выполнения, но приводят к потере локальности и процессорного времени, затрачиваемого на планирование. В главе 11 рассматриваются технические решения для анализа и сокращения издержек.

1.4. Потоки есть везде

Даже если ваша программа не создает поток явно, структуры могут создавать потоки от вашего имени, и код, вызываемый из этих потоков, должен быть потокобезопасным. Это добавляет разработчикам проблем.

JVM-машина при запуске создает потоки для служебных задач (например, сбора мусора) и главный поток для работы метода `main`. Инструментальный пакет AWT (Abstract Window Toolkit) и платформа программирования пользовательского интерфейса Swing создают потоки для управления событиями пользовательского интерфейса. Класс `Timer` создает потоки для выполнения отложенных задач. Компонентные структуры создают пулы (или пучки) потоков и активируют в них компонентные методы.

Мы понимаем, что воспринимать конкурентность как необязательный расширенный функционал языка очень удобно, но реальность такова, что почти все приложения Java являются многопоточными, и имеющиеся структуры не избавляют вас от необходимости правильно координировать доступ к состоянию приложения.

Требование к потокобезопасности нельзя удовлетворить созданием структуры: оно распространяется на все ветви кода, его можно назвать «заразным».

> Программные структуры вносят конкурентность в приложения, вызывая компоненты приложений из структурных потоков. Компоненты неизменно обращаются к состоянию приложения, требуя, чтобы *все* ветви кода, которые обращаются к этому состоянию, были потокобезопасными.

Все описанные ниже механизмы помогут вызывать код приложения из потоков, которые не управляются приложением.

Таймер. Класс `Timer` — это механизм для планирования задач, которые будут работать позже, один раз или периодически. Но введение этого класса может усложнить последовательную программу, поскольку задачи `TimerTask` выполняются в потоке, который управляется таймером `Timer`, а не приложением. Если задача `TimerTask` обращается к обьектам вместе с другими потоками приложения, то за потокобезопасность эти потоки будут отвечать тоже *вместе*, поэтому рекомендуем обеспечить потокобезопасность самих объектов.

Сервлеты и страницы JavaServer (JSP). Сервлет предназначен для обработки всей инфраструктуры развертывания приложения и доставки запросов от удаленных клиентов HTTP. Запрос, поступающий на сервер, может доставляться через цепочку фильтров в соответствующий сервлет или JSP. На крупных веб-сайтах многочисленным клиентам могут сразу потребоваться услуги одного сервлета. Поэтому он должен быть потокобезопасным, даже если вызывается из одного потока.

Вызов удаленного метода. Вызов удаленного метода позволяет активировать методы на объектах, работающих в другой виртуальной машине JVM. При вызове удаленного метода с помощью интерфейса RMI (remote method invocation) аргументы метода упаковываются (маршалируются) в байтовый поток и отправляются по сети в удаленную виртуальную машину JVM, где распаковываются (демаршалируются) и передаются удаленному методу.

Код RMI вызывает удаленный объект в некоем потоке, управляемом RMI. Сколько потоков создает RMI? Может ли один и тот же удаленный метод на одном и том же удаленном объекте вызываться одновременно в многочисленных потоках RMI?[1]

[1] Ответ: да. Но он не вытекает из Javadoc — прочтите спецификацию RMI.

Удаленный объект должен координировать доступ не только к состоянию, используемому совместно с другими объектами, но и к состоянию самого удаленного объекта (поскольку один объект может быть вызван в нескольких потоках одновременно). Как и сервлеты, объекты RMI должны быть подготовлены к многочисленным одновременным вызовам и обязаны обеспечивать собственную потокобезопасность.

Swing и AWT. GUI-приложения по определению являются асинхронными. Пользователи в любое время могут выбрать пункт меню или нажать кнопку. Поэтому Swing и AWT создают отдельные потоки для обработки инициируемых пользователем событий и обновления графического представления.

Компоненты Swing, такие как `JTable`, не являются потокобезопасными. Swing обеспечивает потокобезопасность, ограничивая весь доступ к компонентам GUI-интерфейса событийным потоком. Если приложение хочет управлять GUI-интерфейсом вне событийного потока, оно должно побудить управляющий код выполняться в событийном потоке.

Когда пользователь выполняет действие, в событийном потоке вызывается событийный обработчик для выполнения любой запрошенной операции. Если обработчику необходимо получить доступ к состоянию приложения из других потоков, то событийный обработчик вместе с любым другим кодом, обращающимся к этому состоянию, должен выполнить запрос потокобезопасным способом.

Часть I
Основы

2 Потокобезопасность

Возможно, вас удивит, что конкурентное программирование связано с потоками или замка́ми[1] не более, чем гражданское строительство связано с заклепками и двутавровыми балками. Разумеется, строительство мостов требует правильного использования большого количества заклепок и двутавровых балок, и то же самое касается построения конкурентных программ, которое требует правильного использования потоков и замко́в. Но это всего лишь *механизмы* — средства достижения цели. Написание потокобезопасного кода — это, по сути, управление доступом к *состоянию* и, в частности, к *совместному* (shared) *мутируемому состоянию* (mutable state).

В целом *состояние* объекта — это его данные, хранящиеся в *переменных состояния* (state variables), таких как экземплярные и статические поля или поля из других зависимых объектов. Состояние хеш-массива `HashMap` частично хранится в самом объекте `HashMap`, но также и во многих объектах `Map.Entry`. Состояние объекта включает любые данные, которые могут повлиять на его поведение.

---

[1] В тексте встречаются термины lock и block, которые часто переводятся одним словом «блокировка», что может подразумевать и объект, и процесс. В английском языке для процесса блокирования как приостановки продвижения есть термин blocking. Под термином lock имеется в виду «замок», «замковый защитный механизм». Во избежание путаницы термин lock переводится, как замок, кроме устоявшихся выражений, где принят перевод «блокировка». Замок — это механизм контроля доступа к данным с целью их защиты. В программировании замки часто используются для того, чтобы несколько программ или программных потоков могли использовать ресурс совместно, например, обращаться к файлу для его обновления на поочередной основе. — *Примеч. науч. ред.*

К *совместной* переменной могут обратиться несколько потоков, *мутируемая* — меняет свое значение. На самом деле мы пытаемся защитить от неконтролируемого конкурентного доступа не код, а *данные*.

Создание потокобезопасного объекта требует синхронизации для координации доступа к мутируемому состоянию, невыполнение которой может привести к повреждению данных и другим нежелательным последствиям.

Всякий раз, когда более чем один поток обращается к переменной состояния и один из потоков, возможно, в нее пишет, все потоки должны координировать свой доступ к ней с помощью синхронизации. Синхронизацию в Java обеспечивают ключевое слово `synchronized`, дающее эксклюзивную *блокировку*, а также волатильные (`volatile`) и атомарные переменные и явные замки́.

Удержитесь от соблазна думать, что существуют ситуации, не требующие синхронизации. Программа может работать и проходить свои тесты, но оставаться неисправной и завершиться аварийно в любой момент.

Если многочисленные потоки обращаются к одной и той же переменной, имеющей мутируемое состояние, без соответствующей синхронизации, то *ваша программа неисправна*. Существует три способа ее исправить:

- *не использовать* переменную состояния совместно во всех потоках;
- сделать переменную состояния *немутируемой*;
- при каждом доступе к переменной состояния использовать *синхронизацию*.

Исправления могут потребовать значительных проектных изменений, поэтому *гораздо проще проектировать класс потокобезопасным сразу, чем модернизировать его позже*.

Будут или нет многочисленные потоки обращаться к той или иной переменной, узнать сложно. К счастью, объектно-ориентированные технические решения, которые помогают создавать хорошо организованные и удобные в сопровождении классы — такие как инкапсуляция и сокрытие данных, — также помогают создавать потокобезопасные классы. Чем меньше потоков имеет доступ к определенной переменной, тем проще обеспечить синхронизацию и задать условия, при которых к данной переменной можно обращаться. Язык Java не заставляет вас инкапсулировать состояние — вполне допустимо хранить состояние в публичных

полях (даже публичных статических полях) или публиковать ссылку на объект, который в иных случаях является внутренним, — но чем лучше инкапсулировано состояние вашей программы, тем проще сделать вашу программу потокобезопасной и помочь сопроводителям поддерживать ее в таком виде.

> При проектировании потокобезопасных классов хорошие объектно-ориентированные технические решения: инкапсуляция, немутируемость и четкая спецификация инвариантов — будут вашими помощниками.

Если хорошие объектно-ориентированные проектные технические решения расходятся с потребностями разработчика, стоит поступиться правилами хорошего проектирования ради производительности либо обратной совместимости с устаревшим кодом. Иногда абстракция и инкапсуляция расходятся с производительностью — хотя и не так часто, как считают многие разработчики, — но образцовая практика состоит в том, чтобы сначала делать код правильным, а *затем* — быстрым. Старайтесь задействовать оптимизацию только в том случае, если измерения производительности и потребности говорят о том, что вы обязаны это сделать[1].

Если вы решите, что вам необходимо нарушить инкапсуляцию, то не все потеряно. Вашу программу по-прежнему можно сделать потокобезопасной, но процесс будет сложнее и дороже, а результат — ненадежнее. Глава 4 характеризует условия, при которых можно безопасно смягчать инкапсуляцию переменных состояния.

До сих пор мы использовали термины «потокобезопасный класс» и «потокобезопасная программа» почти взаимозаменяемо. Строится ли потокобезопасная программа полностью из потокобезопасных классов? Не обязательно: программа, которая состоит полностью из потокобезопасных классов, может не быть потокобезопасной, и потокобезопасная программа может содержать классы, которые не являются потокобезопасными. Вопросы, связанные с компоновкой потокобезопасных классов, также

[1] В конкурентном коде следует придерживаться этой практики даже больше, чем обычно. Поскольку ошибки конкурентности чрезвычайно трудно воспроизводимы и не просты в отладке, преимущество небольшого прироста производительности на некоторых редко используемых ветвях кода может вполне оказаться ничтожным по сравнению с риском, что программа завершится аварийно в условиях эксплуатации.

рассматриваются в главе 4. В любом случае понятие потокобезопасного класса имеет смысл только в том случае, если класс инкапсулирует собственное состояние. Термин «потокобезопасность» может применяться к *коду*, но он говорит о *состоянии* и может применяться только к тому массиву кода, который инкапсулирует его состояние (это может быть объект или вся программа целиком).

2.1. Что такое потокобезопасность?

Дать определение потокобезопасности непросто. Быстрый поиск в Google выдает многочисленные варианты, подобные этим:

> ...может вызываться из многочисленных потоков программы без нежелательных взаимодействий между потоками.

> ...может вызываться двумя или более потоками одновременно, не требуя никаких других действий с вызывающей стороны.

Учитывая подобные определения, неудивительно, что мы находим потокобезопасность запутанной! Как отличить потокобезопасный класс от небезопасного? Что мы вообще подразумеваем под словом «безопасный»?

В основе любого разумного определения потокобезопасности лежит понятие *правильности* (correctness).

Правильность подразумевает *соответствие* класса *своей спецификации*. Спецификация определяет *инварианты* (invariants), ограничивающие состояние объекта, и *постусловия* (postconditions), описывающие эффекты от операций. Как узнать, что спецификации для классов являются правильными? Никак, но это не мешает нам их использовать после того, как мы убедили себя, что код работает. Поэтому давайте допустим, что однопоточная правильность — это нечто видимое. Теперь можно предположить, что потокобезопасный класс ведет себя правильно во время доступа из многочисленных потоков.

> Класс является *потокобезопасным*, если он ведет себя правильно во время доступа из многочисленных потоков, независимо от того, как выполнение этих потоков планируется или перемежается рабочей средой, и без дополнительной синхронизации или другой координации со стороны вызывающего кода.

Многопоточная программа не может быть потокобезопасной, если она не является правильной даже в однопоточной среде[1]. Если объект реализован правильно, то никакая последовательность операций — обращения к публичным методам и чтение или запись в публичные поля — не должна нарушать его инварианты или постусловия. *Ни один набор операций, выполняемых последовательно либо конкурентно на экземплярах потокобезопасного класса, не может побудить экземпляр находиться в недопустимом состоянии.*

Потокобезопасные классы инкапсулируют любую необходимую синхронизацию сами и не нуждаются в помощи клиента.

2.1.1. Пример: сервлет без поддержки внутреннего состояния

В главе 1 мы перечислили структуры, которые создают потоки и вызывают из них компоненты, за потокобезопасность которых ответственны вы. Теперь мы намерены разработать сервлетную службу разложения на множители и постепенно расширить ее функционал, сохраняя потокобезопасность.

В листинге 2.1 показан простой сервлет, который распаковывает число из запроса, раскладывает его на множители и упаковывает результаты в отклик.

Листинг 2.1. Сервлет без поддержки внутреннего состояния

```
@ThreadSafe
public class StatelessFactorizer implements Servlet {
    public void service(ServletRequest req, ServletResponse resp) {
        BigInteger i = extractFromRequest(req);
        BigInteger[] factors = factor(i);
        encodeIntoResponse(resp, factors);
    }
}
```

Класс `StatelessFactorizer`, как и большинство сервлетов, не имеет внутреннего состояния: не содержит полей и не ссылается на поля из других классов. Состояние для конкретного вычисления существует только в локальных

[1] Если нестрогое использование термина *правильность* здесь вас беспокоит, то вы можете думать о потокобезопасном классе как о классе, который неисправен в конкурентной среде, как и в однопоточной среде.

переменных, которые хранятся в потоковом стеке и доступны только для выполняющего потока. Один поток, обращающийся к `StatelessFactorizer`, не может повлиять на результат другого потока, делающего то же самое, поскольку эти потоки не используют состояние совместно.

> Объекты без поддержки внутреннего состояния всегда являются потокобезопасными.

Тот факт, что большинство сервлетов могут быть реализованы без поддержки внутреннего состояния, значительно снижает бремя по обеспечению потокобезопасности самих сервлетов. И только когда сервлеты должны что-то запомнить, требования к их потокобезопасности возрастают.

2.2. Атомарность

Что происходит при добавлении элемента состояния в объект без поддержки внутреннего состояния? Предположим, мы хотим добавить счетчик посещений, который измеряет число обработанных запросов. Можно добавить в сервлет поле с типом `long` и приращивать его при каждом запросе, как показано в `UnsafeCountingFactorizer` в листинге 2.2.

Листинг 2.2. Сервлет, подсчитывающий запросы без необходимой синхронизации. *Так делать не следует*

```
@NotThreadSafe
public class UnsafeCountingFactorizer implements Servlet {
    private long count = 0;

    public long getCount() { return count; }

    public void service(ServletRequest req, ServletResponse resp) {
        BigInteger i = extractFromRequest(req);
        BigInteger[] factors = factor(i);
        ++count;
        encodeIntoResponse(resp, factors);
    }
}
```

К сожалению, класс `UnsafeCountingFactorizer` не является потокобезопасным, даже если отлично работает в однопоточной среде. Так же, как `UnsafeSequence`, он предрасположен к *потерянным обновлениям* (lost updates). Хотя операция приращения `++count` имеет компактный синтаксис, она не является *атомарной* (atomic), то есть неделимой, а представляет собой последовательность из трех операций: доставки текущего значения, прибавления к нему единицы и записи нового значения обратно. В операциях «прочитать, изменить, записать» результирующее состояние является производным от предыдущего.

На рис. 1.1 показано, что может произойти, если два потока попытаются увеличить счетчик одновременно, без синхронизации. Если счетчик равен 9, то из-за неудачной временной координации оба потока увидят значение 9, добавят в него единицу, и установят значение 10. Так счетчик посещений начнет отставать на единицу.

Вы можете подумать, что наличие немного неточного счетчика посещений в веб-службе является приемлемой потерей, и иногда это так. Но если счетчик используется для создания последовательностей или уникальных идентификаторов объектов, то возвращение одного и того же значения из многочисленных активаций может привести к серьезным проблемам целостности данных[1]. Возможность появления неправильных результатов из-за неудачной временно́й координации возникает при *состоянии гонки*.

2.2.1. Состояния гонки

Класс `UnsafeCountingFactorizer` имеет несколько состояний гонки[2]. Наиболее распространенным типом состояния гонки является ситуация «про-

---

[1] Подход, принятый в `UnsafeSequence` и `UnsafeCountingFactorizer`, имеет другие серьезные проблемы, включая возможность устаревших данных (см. раздел 3.1.1).

[2] Термин *состояние гонки* часто путают с родственным термином *гонка данных* (data race). Гонка данных возникает, когда синхронизация не используется для координации всего доступа к общему нефинальному полю. Вы рискуете попасть в гонку данных всякий раз, когда поток пишет переменную, которая затем может быть прочитана другим потоком, либо считывает переменную, которая в последний раз могла быть записана другим потоком, если оба потока не используют синхронизацию. Код с гонками данных не имеет полезной формально определенной семантики в рамках модели памяти Java. Не все состояния гонки являются гонками данных, и не все гонки данных являются состояниями гонки, но оба типа ситуаций могут вызывать аварийный сбой конкурентных программ

верить и затем действовать», где потенциально устаревшее наблюдение используется для принятия решения о том, что делать дальше.

Мы часто сталкиваемся с состоянием гонки в реальной жизни. Допустим, вы планируете встретиться с другом в полдень в кафе «Старбакс» на Университетском проспекте. Но вы узнаете, что на Университетском проспекте находятся *два* «Старбакса». В 12:10 вы не видите своего друга в кафе *A* и идете в кафе *B*, но там его тоже нет. Либо ваш друг опаздывает, либо он прибыл в кафе *A* сразу после того, как вы ушли, либо он *был* в кафе *B*, но пошел вас искать и теперь находится на пути к кафе *A*. Примем последний, то есть самый худший вариант. Сейчас 12:15, и вы оба задаетесь вопросом, а сдержал ли друг обещание. Вы вернетесь в другое кафе? Сколько раз вы будете ходить туда и обратно? Если вы не согласовали протокол, то можете провести весь день, гуляя по Университетскому проспекту в кофеиновой эйфории.

Проблема подхода «прогуляться и посмотреть, не находится ли он там» заключается в том, что прогулка по улице между двумя кафе занимает несколько минут, и за это время *состояние системы может измениться*.

Пример со «Старбаксом» иллюстрирует зависимость результата от относительной временно́й координации событий (от того, как долго вы ждете друга, находясь в кафе, и т. д.). Наблюдение, что он не находится в кафе *A*, становится потенциально утратившим силу: как только вы выходите из парадной двери, он может войти через заднюю дверь. Большинство состояний гонки вызывают такие проблемы, как неожиданное исключение, перезаписанные данные и повреждение файла.

2.2.2. Пример: состояния гонки в ленивой инициализации

Распространенным приемом, использующим подход «проверить и затем действовать», является *ленивая инициализация* (`LazyInitRace`). Ее цель — отложить инициализацию объекта до тех пор, пока он не понадобится, и обеспечить, чтобы он инициализировался только один раз. В листинге 2.3 метод `getInstance` убеждается в выполнении инициализации `ExpensiveObject` и возвращает существующий экземпляр, или,

---

самым непредсказуемым образом. `UnsafeCountingFactorizer` содержит оба типа. Подробнее гонки данных описаны в главе 16.

в противном случае, создает новый экземпляр и возвращает его после сохранения ссылки на него.

Листинг 2.3. Состояние гонки в ленивой инициализации. *Так делать не следует*

```
@NotThreadSafe
public class LazyInitRace {
    private ExpensiveObject instance = null;

    public ExpensiveObject getInstance() {
        if (instance == null)
            instance = new ExpensiveObject();
        return instance;
    }
}
```

Класс `LazyInitRace` содержит состояния гонки. Предположим, что потоки *A* и *B* выполняют метод `getInstance` в одно и то же время. *A* видит, что поле `instance` равно `null`, и создает новый `ExpensiveObject`. Поток *B* также проверяет, равно ли поле `instance` тому же значению `null`. Наличие в поле значения `null` в этот момент зависит от временно́й координации, включая капризы планирования и количество времени, нужного для создания экземпляра объекта `ExpensiveObject` и установки значения в поле `instance`. Если поле `instance` равно `null`, когда *B* его проверяет, два элемента кода, вызывающих метод `getInstance`, могут получить два разных результата, даже если метод `getInstance` предположительно должен всегда возвращать один и тот же экземпляр.

Счетчик посещений в `UnsafeCountingFactorizer` тоже содержит состояния гонки. Подход «прочитать, изменить, записать» подразумевает, что для приращения счетчика поток должен знать его предыдущее значение *и* убедиться, что в процессе обновления никто другой не изменяет и не использует это значение.

Как и большинство ошибок конкурентности, состояния гонки не *всегда* приводят к сбою: временна́я координация бывает удачной. Но если класс `LazyInitRace` используется для инстанциации реестра всего приложения, то, когда из многочисленных активаций он будет возвращать разные экземпляры, регистрации будут утеряны либо действия получат противо-

речивые представления набора зарегистрированных объектов. Или если класс `UnsafeSequence` используется для генерирования идентификаторов сущностей в структуре консервации данных, то два разных объекта могут иметь один и тот же идентификатор, нарушая ограничения идентичности.

2.2.3. Составные действия

И `LazyInitRace`, и `UnsafeCountingFactorizer` содержат последовательность операций, которые должны быть *атомарными*. Но для предотвращения состояния гонки должно существовать препятствие тому, чтобы другие потоки использовали переменную, пока один поток ее изменяет.

> Операции *A* и *B* являются *атомарными*, если, с точки зрения потока, выполняющего операцию *A*, операция *B* либо была целиком выполнена другим потоком, либо не выполнена даже частично.

Атомарность операции приращения в `UnsafeSequence` позволила бы избежать состояния гонки, показанного на рис. 1.1. Операции «проверить и затем действовать» и «прочитать, изменить, записать» всегда должны быть атомарными. Они называются *составными действиями* (compound actions) — последовательностями операций, которые должны выполняться атомарно, для того чтобы оставаться потокобезопасными. В следующем разделе мы рассмотрим *блокировку* — встроенный в Java механизм, который обеспечивает атомарность. А пока мы исправим проблему другим способом, применив существующий потокобезопасный класс, как показано в `Countingfactorizer` в листинге 2.4.

Листинг 2.4. Сервлет, подсчитывающий запросы с помощью AtomicLong

```
@ThreadSafe
public class CountingFactorizer implements Servlet {
    private final AtomicLong count = new AtomicLong(0);

    public long getCount() { return count.get(); }

    public void service(ServletRequest req, ServletResponse resp) {
        BigInteger i = extractFromRequest(req);
        BigInteger[] factors = factor(i);
        count.incrementAndGet();
        encodeIntoResponse(resp, factors);
    }
}
```

Пакет `java.util.concurrent.atomic` содержит *атомарные переменные* (atomic variable) для управления состояниями классов. Заменив тип счетчика с `long` на `AtomicLong`, мы гарантируем, что все действия, которые обращаются к состоянию счетчика, являются атомарными[1]. Поскольку состояние сервлета является состоянием счетчика, а счетчик является потокобезопасным, наш сервлет становится потокобезопасным.

При добавлении *единственного* элемента состояния в класс, который не поддерживает внутреннее состояние, результирующий класс будет потокобезопасным, если состояние полностью управляется потокобезопасным объектом. Но, как мы увидим в следующем разделе, переход от одной переменной состояния к следующим будет не так прост, как переход от нуля к единице.

> Там, где это удобно, используйте существующие потокобезопасные объекты, такие как `AtomicLong`, для управления состоянием вашего класса. Возможные состояния существующих потокобезопасных объектов и их переходы в другие состояния легче поддерживать и проверять на потокобезопасность, нежели произвольные переменные состояния.

2.3. Блокировка

Мы смогли добавить в сервлет одну переменную состояния, при этом сохранив потокобезопасность путем использования потокобезопасного объекта для управления всем состоянием сервлета. Но если мы хотим добавить больше состояний, можно ли просто добавить несколько потокобезопасных переменных состояния?

Представим, что мы хотим улучшить производительность нашего сервлета, кэшируя последний вычисленный результат на случай, если два клиента подряд запросят разложение на множители одного и того же числа. (Эта стратегия кэширования вряд ли будет эффективной, и в разделе 5.6 мы

[1] `CountingFactorizer` вызывает `incrementAndGet` для приращения счетчика, который тоже возвращает приращенное значение, поэтому возвращаемое значение игнорируется.

предлагаем более оптимальную стратегию.) Для этого нужно запомнить две вещи: последнее разложенное число и его множители.

Мы использовали `AtomicLong` для управления состоянием счетчика потокобезопасным способом. Можно ли для управления последним числом и его множителями применить `AtomicReference`[1]? Попытка сделать это показана в `Unsafecachingfactorizer` в листинге 2.5.

Листинг 2.5. Сервлет, пытающийся кэшировать свой последний результат без адекватной атомарности. *Так делать не следует*

```
@NotThreadSafe
public class UnsafeCachingFactorizer implements Servlet {
    private final AtomicReference<BigInteger> lastNumber
        = new AtomicReference<BigInteger>();
    private final AtomicReference<BigInteger[]> lastFactors
        = new AtomicReference<BigInteger[]>();

    public void service(ServletRequest req, ServletResponse resp) {
        BigInteger i = extractFromRequest(req);
        if (i.equals(lastNumber.get()))
            encodeIntoResponse(resp, lastFactors.get());
        else {
            BigInteger[] factors = factor(i);
            lastNumber.set(i);
            lastFactors.set(factors);
            encodeIntoResponse(resp, factors);
        }
    }
}
```

К сожалению, такой подход не работает. Несмотря на то что атомарные ссылки потокобезопасны, класс `UnsafeCachingFactorizer` содержит состояния гонки.

---

[1] Как и в случае с классом `AtomicLong`, который является потокобезопасным держателем целого числа с типом `long`, класс `AtomicReference` является потокобезопасным держателем объектной ссылки. Атомарные переменные и их преимущества рассматриваются в главе 15.

Определение потокобезопасности требует соблюдения инвариантов независимо от временной координации или перемежения операций в многочисленных потоках. Один инвариант класса `UnsafeCachingFactorizer` заключается в том, что кэшированное в поле `lastFactors` произведение множителей равно значению, кэшированному в поле `lastNumber`. Наш сервлет является правильным, только если этот инвариант всегда соблюдается. Когда в инварианте участвуют многочисленные переменные, поля не являются *независимыми*: значение одного ограничивает допустимые значения другого. Следовательно, при обновлении одного необходимо обновлять другие *в той же атомарной операции*.

При неудачной временной координации класс `UnsafeCachingFactorizer` может нарушить инвариант. Используя атомарные ссылки, мы не можем обновлять оба поля, `lastNumber` и `lastfactors`, одновременно. Даже если каждый вызов метода `set` будет атомарным, останется окно уязвимости, когда одно поле изменено, а другое нет, и потоки видят, что инвариант не соблюдается. Схожим образом, два значения не могут быть доставлены одновременно: в промежутке, когда поток *A* доставляет два значения, поток *B* может их изменить, и *A* заметит несоблюдение инварианта.

> Для сохранения непротиворечивости состояний обновляйте родственные переменные состояния в единой атомарной операции.

2.3.1. Внутренние замки́

Java предоставляет встроенный замковый механизм для усиления атомарности — *синхронизированный блок*, состоящий из ссылки на объект-*замок* (lock), и блока кода, который будет им защищен. Ключевое слово `synchronized` является условным обозначением и метода, и замка. (Статические синхронизированные методы используют объект `Class`.)

```
synchronized (lock) {
    // Обратиться к защищаемому замком совместному состоянию либо его
    // изменить
}
```

Каждый объект Java может неявно действовать как замок для целей синхронизации, то есть являться *внутренним замком* (intrinsic locks)[1] или *мониторным замком* (monitor locks). Замок автоматически приобретается выполняющим потоком перед входом в синхронизированный блок и автоматически освобождается, когда управление выходит из синхронизированного блока, либо обычным путем выполнения кода, либо путем исключения из блока. Приобрести внутренний замок можно только при входе в синхронизированный блок или в метод, защищенный этим замком.

Внутренние замки в Java действуют как взаимоисключающие замки — *мьютексы* (mutual exclusion locks). Это означает, что замком может владеть не более чем один поток. Когда поток *A* пытается приобрести замок, которым владеет поток *B*, он должен ждать или *блокировать* продвижение до тех пор, пока *B* его не освободит. Если *B* не освободит замок никогда, то *A* будет ждать вечно.

Поскольку только один поток за раз может выполнять блок кода, защищенный замком, синхронизированные блоки, защищенные тем же замком, выполняются атомарно. Никакой поток, выполняющий синхронизированный блок, не может наблюдать другой поток в синхронизированном блоке, защищенном тем же замком.

Механизм синхронизации позволяет легко восстановить потокобезопасность сервлета разложения на множители. В листинге 2.6 метод `service` делается синхронизированным, то есть впускающим один поток за раз. Класс `SynchronizedFactorizer` теперь является потокобезопасным, но многочисленные клиенты больше не могут использовать сервлет одновременно, что приводит к неприемлемо низкой отзывчивости. Эта проблема — которая касается производительности, а не потокобезопасности — рассматривается в разделе 2.5.

---

[1] *Внутренний замок*, иногда *мониторный замок* — это внутренняя сущность, вокруг которой строится синхронизация (в спецификация API часто называется *монитором*). Внутренние замки играют роль в обоих аспектах синхронизации: обеспечении эксклюзивного доступа к состоянию объекта и установлении связей, необходимых для обеспечения видимости. См. https://docs.oracle.com/javase/tutorial/essential/concurrency/locksync.html. — *Примеч. пер.*

Листинг 2.6. Сервлет, кэширующий последний результат с неприемлемо слабой конкурентностью. *Так делать не следует*

```
@ThreadSafe
public class SynchronizedFactorizer implements Servlet {
    @GuardedBy("this") private BigInteger lastNumber;
    @GuardedBy("this") private BigInteger[] lastFactors;

    public synchronized void service(ServletRequest req,
                                     ServletResponse resp) {
        BigInteger i = extractFromRequest(req);
        if (i.equals(lastNumber))
            encodeIntoResponse(resp, lastFactors);
        else {
            BigInteger[] factors = factor(i);
            lastNumber = i;
            lastFactors = factors;
            encodeIntoResponse(resp, factors);
        }
    }
}
```

2.3.2. Повторная входимость

Когда поток запрашивает замок, которым уже владеет другой поток, он блокирует продвижение. Но так как внутренние замки являются повторно *входимыми* (reentrant), если поток пытается приобрести замок, которым он уже владеет, то запрос выполнится успешно. Повторная входимость означает, что замки приобретаются в расчете на один поток, а не в расчете на один вызов[1], путем ассоциирования потоков с замками. Когда счетчик приобретения равен нулю, замок считается не занятым. Когда поток приобретает ранее не занятый замок, JVM регистрирует владельца и устанавливает счетчик приобретения, равный единице. Если поток снова приобретает замок, то счетчик приращивает единицу, и когда владеющий поток выходит из синхронизированного блока,

[1] Это отличается от замкового поведения, принятого по умолчанию для *p*-поточных (POSIX-поточных) мьютексов, которые предоставляются на основе каждого вызова.

счетчик уменьшает значение. Замок освобождается, когда счетчик достигает нуля.

Повторная входимость способствует инкапсуляции замкового поведения и упрощает разработку объектно-ориентированного конкурентного кода. Без повторно входимых замков код в листинге 2.7 был бы заперт *взаимной блокировкой* (deadlock), а вызов `super.doSomething` никогда не смог бы приобрести замок, считающийся занятым.

Листинг 2.7. Код, запертый взаимной блокировкой, так как внутренние замки не являются повторно входимыми

```
public class Widget {
    public synchronized void doSomething() {
        ...
    }
}

public class LoggingWidget extends Widget {
    public synchronized void doSomething() {
        System.out.println(toString() + ": calling doSomething");
        super.doSomething();
    }
}
```

2.4. Защита состояния с помощью замков

Поскольку замки задействуют последовательный[1] доступ к защищаемым ими ветвям кода, мы можем использовать их для создания протоколов, чтобы обеспечить эксклюзивный доступ к совместному состоянию и непротиворечивость состояний.

Составные действия в совместном состоянии, такие как приращение счетчика посещений или ленивая инициализация, должны быть атомарными, чтобы избежать состояний гонки. Удержание блокировки в течение *всего времени* составного действия может сделать это действие атомарным, однако если синхронизация используется для координации доступа к переменной, то она необходима *везде, где есть доступ к этой*

[1] Сериализация доступа к объекту не имеет ничего общего с сериализацией объекта (превращением объекта в байтовый поток) и означает, что потоки обращаются к объекту по очереди эксклюзивно, а не конкурентно.

переменной. Причем должен использоваться *один и тот же* замок везде, где осуществляется доступ к этой переменной.

Распространенная ошибка — считать, что синхронизация должна использоваться только во время *записи* в совместные переменные (см. раздел 3.1).

> *Все* обращения к мутируемой переменной состояния должны выполняться с удержанием *одного и того же* замка. Только тогда переменная *защищена* этим замком.

В классе `SynchronizedFactorizer` в листинге 2.6 `lastNumber` и `lastFactors` защищены внутренним замком сервлетного объекта, что документируется аннотацией `@GuardedBy`.

Между внутренним замком объекта и его состоянием нет обязательной связи: поля объекта не всегда защищены внутренним замком, несмотря на допустимое замковое соглашение, используемое многими классами. Приобретение ассоциированного с объектом замка *не* препятствует тому, чтобы другие потоки обращались к этому объекту. Единственное, что приобретение замка не дает сделать другому потоку, — это приобрести тот же самый замок. Тот факт, что каждый объект имеет встроенный замок, является просто удобством, благодаря которому не нужно создавать замковые объекты явно[1]. За создание и непротиворечивое использование замковых *протоколов*, или *политик*, *синхронизации* отвечает разработчик.

> Каждая совместная мутируемая переменная должна быть защищена только одним замком. Дайте сопроводителям четкое понимание, какой это замок.

Инкапсуляция всего мутируемого состояния внутри объекта обеспечивает его защиту от конкурентного доступа. Благодаря ей все переменные внутри состояния объекта защищаются внутренним замком объекта. Однако ни компилятор, ни рабочая среда не усиливают этот (или другой)

[1] В ретроспективе это проектное решение было, вероятно, плохим: оно не только может сбивать с толку, но и заставляет разработчиков JVM идти на компромисс между размером объекта и производительностью замковой защиты.

замковый шаблон[1], поэтому протокол подвержен сбоям при добавлении нового метода или ветви кода без синхронизации.

Не все данные должны быть защищены замками, а только мутируемые и запрашиваемые несколькими потоками. В главе 1 мы описали, как добавление простого асинхронного события, такого как задача `TimerTask`, может создавать требования к потокобезопасности для всей программы. Рассмотрим однопоточную программу, обрабатывающую большой объем данных, в которую нужно добавить функционал создания периодических снимков продвижения ее работы, чтобы в случае аварийной остановки она не вернулась в самое начало. Добавим объект `TimerTask`, который срабатывает каждые десять минут, сохраняя состояние программы в файле.

Поскольку задача `TimerTask` будет вызываться из другого потока (управляемого объектом `Timer`), все задействованные в снимке данные начнут запрашиваться двумя потоками: главным потоком программы и таймерным. То есть при обращении к состоянию программы использовать синхронизацию должны и программный код задачи `TimerTask`, и любая ветвь кода, которая касается тех же данных.

Когда *каждый* доступ к переменной осуществляется с удержанием замка, только один поток за раз может к ней обратиться. Когда класс имеет инварианты, включающие более одной переменной состояния, каждая переменная, участвующая в инварианте, должна быть защищена *тем же* замком. Это позволит обращаться к переменным или обновлять их в единой атомарной операции, соблюдая инвариант. Класс `SynchronizedFactorizer` демонстрирует это правило: и кэшированное число, и кэшированные множители защищены внутренним замком сервлетного объекта.

> Для каждого инварианта, который включает более одной переменной, *все* переменные, участвующие в инварианте, должны быть защищены *тем же* замком.

Если синхронизация спасает от состояний гонки, почему бы просто не объявлять каждый метод синхронизированным? Оказывается, что такое

[1] Инструменты программного аудита, такие как `FindBugs`, могут идентифицировать ситуации, когда к переменной часто, но не всегда, обращаются с удержанием замка, что может указывать на дефект.

неизбирательное применение механизма `synchronized` может спровоцировать либо недостаточную, либо чрезмерную синхронизацию. Простой синхронизации каждого метода, как это делает `Vector`, недостаточно для того, чтобы сделать составные действия на `Vector` атомарными:

```
if (!vector.contains(element))
    vector.add(element);
```

Операция «добавить, если отсутствует» содержит состояния гонки, даже если методы `contains` и `add` являются атомарными. Хотя синхронизированные методы могут делать отдельные операции атомарными, дополнительная блокировка требуется, когда многочисленные операции объединяются в составное действие. (В разделе 4.4 описаны несколько технических решений для безопасного добавления атомарных операций в потокобезопасные объекты.) В то же время синхронизация каждого метода может привести к проблемам жизнеспособности и производительности, которые мы видели в классе `SynchronizedFactorizer`.

2.5. Живучесть и производительность

В классе `UnsafeCachingFactorizer` мы ввели в наш сервлет разложения на множители кэширование в надежде улучшить производительность. Кэширование требует некоего совместного состояния, которое, в свою очередь, требует синхронизации. Но нужно понимать, что политика синхронизации для `SynchronizedFactorizer` состоит в защите каждой переменной состояния внутренним замком сервлетного объекта, и эта политика была реализована путем синхронизации всего метода `service` целиком. Этот простой подход восстановил безопасность, но за высокую цену.

Поскольку метод `service` синхронизирован, только один поток может его выполнять за один раз. Если сервлет будет занят разложением на множители большого числа, другим клиентам придется ждать завершения текущего запроса: процессоры будут простаивать даже при высокой нагрузке, а выполнение кратковременных запросов, для которых значение кэшируется, наоборот, растянется.

На рис. 2.1 показано, как запросы помещаются в очередь и обрабатываются последовательно. Мы назовем это приложение демонстрирующим *слабую конкурентность*: число одновременных активаций ограничено не наличи-

ем обрабатываемых ресурсов, а структурой самого приложения. К счастью, конкурентность сервлета можно улучшить, причем сохранив потокобезопасность путем сужения области действия синхронизированного блока. Будьте осторожны, чтобы не сделать эту область *слишком* малой, ведь вы не хотите делить операцию, которая должна быть атомарной, на несколько синхронизированных блоков. Разумно попытаться исключить из синхронизированных блоков длительные операции, которые не влияют на совместное состояние, для того чтобы другие потоки не были лишены доступа к совместному состоянию в то время, когда длительная операция находится в процессе работы.

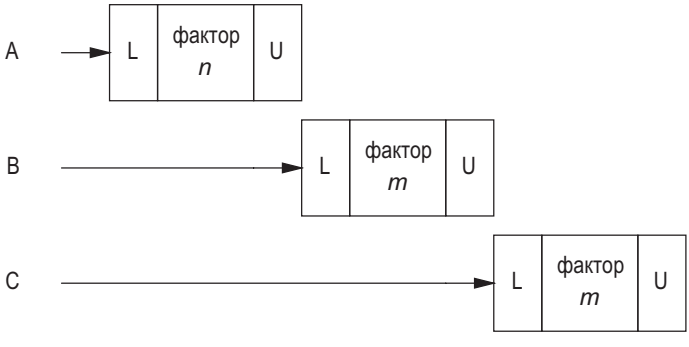

Рис. 2.1. Слабая конкурентность класса SynchronizedFactorizer

Класс `CachedFactorizer` в листинге 2.8 реструктурирует сервлет за счет использования двух синхронизированных блоков, где каждый ограничен коротким фрагментом кода. Один защищает последовательность «проверить и затем действовать» и проверяет возможность вернуть кэшированный результат, а другой защищает обновление кэшированного числа и кэшированных множителей. В качестве бонуса мы вновь ввели счетчик посещений и добавили еще один счетчик посещений в кэш, обновив их в первоначальном синхронизированном блоке. Поскольку эти счетчики также составляют совместно используемое мутируемое состояние, мы должны добавить синхронизацию везде, где к ним осуществляется доступ. Части кода, находящиеся вне синхронизированных блоков, работают эксклюзивно на локальных (стековых) переменных, которые не используются совместно во всех потоках и поэтому не требуют синхронизации.

Листинг 2.8. Сервлет, который кэширует свой последний запрос и результат

```
@ThreadSafe
public class CachedFactorizer implements Servlet {
    @GuardedBy("this") private BigInteger lastNumber;
    @GuardedBy("this") private BigInteger[] lastFactors;
    @GuardedBy("this") private long hits;
    @GuardedBy("this") private long cacheHits;

    public synchronized long getHits() { return hits; }
    public synchronized double getCacheHitRatio() {
        return (double) cacheHits / (double) hits;
    }

    public void service(ServletRequest req, ServletResponse resp) {
        BigInteger i = extractFromRequest(req);
        BigInteger[] factors = null;
        synchronized (this) {
            ++hits;
            if (i.equals(lastNumber)) {
                ++cacheHits;
                factors = lastFactors.clone();
            }
        }
        if (factors == null) {
            factors = factor(i);
            synchronized (this) {
                lastNumber = i;
                lastFactors = factors.clone();
            }
        }
        encodeIntoResponse(resp, factors);
    }
}
```

Класс `CachedFactorizer` больше не использует `AtomicLong` для счетчика посещений, поскольку здесь он не так полезен, как в классе `CountingFactorizer`. Мы уже используем синхронизированные блоки для построения атомарных операций, и второй механизм синхронизации не нужен.

Реструктуризация класса `CachedFactorizer` обеспечивает баланс между простотой (синхронизация всего метода целиком) и конкурентностью (синхронизация кратчайших из возможных ветвей кода). Приобретение и освобождение замка имеет некоторые издержки, поэтому

нежелательно разводить синхронизированные блоки *слишком* далеко (например, раскладывать на множители `++hits` в отдельный синхронизированный блок), даже если это не подвергнет атомарность риску. Класс `CachedFactorizer` владеет замком во время обращения к переменным состояния и в течение всего времени составных действий, но освобождает его перед выполнением потенциально длительной операции разложения на множители.

Решение о том, насколько большими или насколько малыми сделать синхронизированные блоки, может потребовать компромисса между безопасностью, простотой и производительностью.

> При реализации политики синхронизации не поддавайтесь искушению пожертвовать простотой (что может поставить безопасность под угрозу) ради производительности.

Используя блокировку, обращайте внимание, насколько долгим будет выполнение программного кода. Длительное удержание блокировки создает риск возникновения проблем жизнеспособности и производительности.

> Избегайте удержания блокировки во время длительных вычислений или операций, таких как сетевой или консольный ввод-вывод.

3 Совместное использование объектов

В главе 2 мы обозначили, что написание правильных конкурентных программ прежде всего связано с доступом к совместному мутируемому состоянию, и описали механизмы синхронизации для предотвращения одновременного доступа многочисленных потоков к одним и тем же данным. В этой главе мы рассмотрим технические решения для совместного использования и публикации объектов при безопасном доступе к ним со стороны многочисленных потоков. Вместе эти решения закладывают основу для построения потокобезопасных классов и безопасного структурирования конкурентных приложений с использованием классов библиотеки `java.util.concurrent`.

Мы видели, как синхронизированные блоки и методы могут обеспечивать атомарность выполнения операций, однако существует распространенное заблуждение, что механизм `synchronized` затрагивает *исключительно* атомарность или демаркацию «критических секций». Синхронизация также имеет еще один важный аспект: *видимость памяти* (memory visibility). Мы обеспечим потокам во время изменения объекта возможность *видеть* внесенные другим потоком изменения и опубликуем объекты с помощью синхронизации (явной или встроенной в библиотечные классы).

3.1. Видимость

Видимость вещей, появления которых вы никак не ожидаете, весьма затруднена. Если в однопоточной среде вы пишете значение в переменную

и позже читаете эту переменную без промежуточных записей, то вы надеетесь увидеть то же самое значение. Но когда чтение и запись происходят в разных потоках, *нет* гарантии (без синхронизации), что читающий поток своевременно увидит значение, записанное другим потоком.

Класс `NoVisibility` в листинге 3.1 иллюстрирует, как потоки используют данные совместно, без синхронизации. Два потока — главный и читающий — обращаются к совместным переменным `ready` и `number`. Главный поток запускает читающий поток, а затем устанавливает `number` значение `42`, и `ready` — значение `true`. Читающий поток видит, что `ready` равна `true`, и распечатывает `number`. Очевидно, что `NoVisibility` должен напечатать `42`, но этого может не произойти!

Листинг 3.1. Совместное использование переменных без синхронизации. *Так делать не следует*

```
public class NoVisibility {
    private static boolean ready;
    private static int number;

    private static class ReaderThread extends Thread {
        public void run() {
            while (!ready)
                Thread.yield();
            System.out.println(number);
        }
    }

    public static void main(String[] args) {
        new ReaderThread().start();
        number = 42;
        ready = true;
    }
}
```

Объект класса `NoVisibility` может попасть в бесконечный цикл, если значение `ready` невидимо для читающего потока. Или напечатать ноль, если операция записи в `ready` станет видимой читающему потоку *перед* операцией записи в `number`, — явление, известное как *переупорядочива-*

ние (reordering). Нет гарантии, что операции в одном потоке будут выполняться в заданном программой порядке[1].

> Без синхронизации компилятор, процессор и рабочая среда могут запутать порядок выполнения операций. Не стоит ожидать естественного порядка действий памяти в недостаточно синхронизированных многопоточных программах.

Даже `NoVisibility` — максимально простая конкурентная программа из двух потоков и двух совместных переменных — может стать неисправной.

Как бы проблематично это ни выглядело, но придется *применять правильную синхронизацию, когда данные используются потоками совместно*.

3.1.1. Устаревшие данные

Класс `NoVisibility` продемонстрировал один из путей появления *устаревших данных* (stale data), которые видит читающий поток, если синхронизация не используется *всякий раз при обращении к переменной*. Но бывают случаи хуже этого: поток может увидеть актуальное значение одной переменной и устаревшее значение другой переменной, которая была записана первой.

Устаревшие данные опасны не столько для счетчика посещений[2], сколько для объектных ссылок, таких как связные указатели в связанном списке, где *устаревшие значения могут вызывать сбои в безопасности или жизне-*

---

[1] Эта схема может показаться неисправной, но она предназначена для того, чтобы позволить JVM-машинам в полной мере воспользоваться производительностью современного многопроцессорного оборудования. Например, при отсутствии синхронизации модель памяти Java позволяет компилятору переупорядочивать операции и значения кэша в регистрах, а процессорам — переупорядочивать операции и значения кэша в кэшах, специфичных для процессора. За более подробной информацией обратитесь к главе 16.

[2] Чтение данных без синхронизации аналогично использованию уровня изоляции `READ_UNCOMMITTED` в базе данных, где можно выменивать производительность на точность. Однако в случае несинхронизированных операций чтения вы жертвуете более высокой степенью точности, поскольку видимое значение совместной переменной может быть произвольно устаревшим.

способности: неожиданные исключения, поврежденные структуры данных, неточные вычисления и бесконечные циклы.

Класс `MutableInteger` в листинге 3.2 не является потокобезопасным, поскольку доступ к полю `value` осуществляется как из `get`, так и из `set` без синхронизации. Причем он подвержен устаревшим значениям: если один поток вызывает `set`, то другие потоки, вызывающие `get`, могут не увидеть обновления.

Чтобы сделать `MutableInteger` потокобезопасным, необходимо синхронизировать методы доступа `get` и `set`, как показано в `Synchronizedinteger` в листинге 3.3. Синхронизации только метода `set` будет недостаточно: потоки, вызывающие `get`, все равно будут видеть устаревшие значения.

Листинг 3.2. Непотокобезопасный мутируемый держатель целого числа

```
@NotThreadSafe
public class MutableInteger {
    private int value;

    public int get() { return value; }
    public void set(int value) { this.value = value; }
}
```

Листинг 3.3. Потокобезопасный мутируемый держатель целого числа

```
@ThreadSafe
public class SynchronizedInteger {
    @GuardedBy("this") private int value;

    public synchronized int get() { return value; }
    public synchronized void set(int value) { this.value = value; }
}
```

3.1.2. Неатомарные 64-разрядные операции

Когда поток читает переменную без синхронизации, он может увидеть устаревшее значение, но можно утверждать, что это значение было по-

мещено туда каким-то потоком, а не возникло случайно. Эта гарантия безопасности называется *безопасностью из ниоткуда* (out-of-thin-air).

Безопасность из ниоткуда применима ко всем переменным с одним исключением: 64-разрядные числовые переменные (с типом `double` и `long`), которые не объявлены волатильными (см. раздел 3.1.4). Модель памяти Java требует, чтобы операции доставки из памяти и сохранения в память были атомарными, но переменным типов `double` и `long` разрешено воспринимать 64-разрядное чтение и запись как две отдельные 32-разрядные операции. Если чтения и записи происходят в разных потоках, то при чтении переменной `long` можно получить назад верхние 32 бита одного значения и нижние 32 бита другого[1]. Таким образом, использовать в многопоточных программах совместные мутируемые переменные с типом `double` и `long` небезопасно, если они не объявлены волатильными или не защищены замком.

3.1.3. Блокировка и видимость

Как показано на рисунке 3.1, блокировка внутренним замком может использоваться для того, чтобы гарантировать видимость действий одного потока другими. Когда поток *A* выполняет синхронизированный блок, а затем поток *B* входит в синхронизированный блок, защищенный тем же замком, значения переменных, которые были видны потоку *A* до освобождения замка, будут видны потоку *B* по приобретении замка. *Без синхронизации видимость не гарантирована*.

Гарантия видимости — еще одна причина синхронизации всех потоков на *одинаковом* замке во время доступа к совместной мутируемой переменной.

> Чтобы обеспечить видимость актуальных значений совместных волатильных переменных, синхронизируйте читающие и пишущие потоки на общем замке.

[1] Когда была написана спецификация виртуальной машины Java, многие широко используемые тогда процессорные архитектуры не могли эффективно обеспечивать атомарные 64-разрядные арифметические операции.

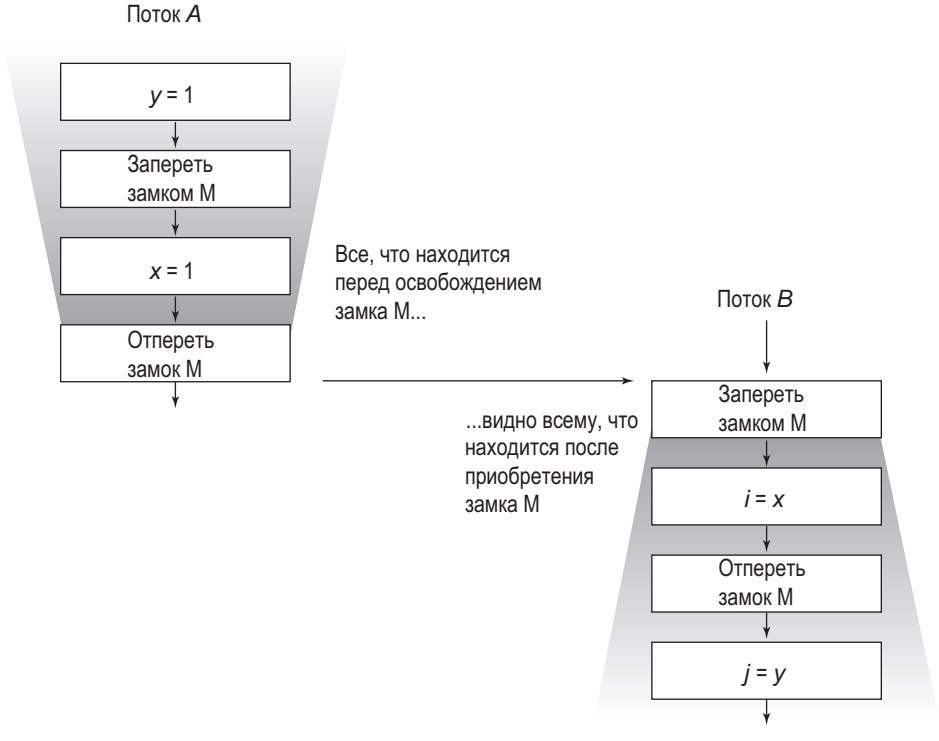

Рис. 3.1. Видимость гарантирует синхронизацию

3.1.4. Волатильные переменные

Язык Java также предоставляет альтернативную, более слабую форму синхронизации — использование *волатильных переменных*, обновления которых распространяются предсказуемо всеми потоками. Переменная `volatile` для компилятора и рабочей среды является совместной, то есть операции над ней не будут переупорядочены с другими операциями в памяти. Волатильные переменные не кэшируются в регистрах или кэшах, где данные скрыты от других процессоров, поэтому их чтение всегда возвращает самый последний результат операций записи.

Приятно думать, что волатильные переменные ведут себя примерно так же, как методы `get` и `set`[1] класса `SynchronizedInteger` в листинге 3.3.

[1] Аналогия неточна: эффекты видимости памяти класса `SynchronizedInteger` сильнее возможностей волатильных переменных. См. главу 16.

Однако обращение к волатильной переменной не может побудить выполняющий поток к блокированию, что делает ее легковесным механизмом синхронизации[1].

Эффекты видимости волатильной переменной выходят за пределы ее значения. Когда поток *A* пишет значение в волатильную переменную и затем поток *B* его читает, значения *всех* переменных, которые были видны до этой записи, становятся видимыми потоку *B*. Запись в волатильную переменную похожа на выход из синхронизированного блока, а ее чтение — на вход в него. Однако мы не рекомендуем слишком полагаться на волатильные переменные в вопросах видимости: код, который использует волатильные переменные ради видимости произвольного состояния, является более хрупким и трудным для понимания, чем код, содержащий блокировку.

> Используйте волатильные переменные только тогда, когда они упрощают реализацию и проверку политики синхронизации. И избегайте их использования, когда диагностика правильности требует утонченных рассуждений о видимости. Волатильные переменные должны обеспечивать видимость их собственного состояния, состояния объекта, на который они ссылаются, или важного события жизненного цикла (например, инициализации или выключения).

Листинг 3.4 иллюстрирует типичное использование волатильных переменных. Они должны проверить статусный флажок и определить время выхода из цикла. Поток пытается «уснуть» с помощью метода подсчета овец. Флажок `asleep` должен быть волатильным, иначе поток может не заметить, когда флажок был установлен другим потоком[2]. Обеспечить

---

[1] На большинстве современных процессорных архитектур волатильные чтения ненамного дороже неволатильных.

[2] Совет по отладке: в серверном приложении всегда указывайте переключатель командной строки JVM `-server` во время вызова JVM даже для разработки и тестирования. Серверная JVM выполняет больше оптимизационных задач, чем клиентская JVM, например извлекает из цикла переменные, которые в нем не модифицируются. Код, который может показаться работающим в рабочей среде (клиентская JVM), может нарушиться в среде развертывания (серверная JVM). Если бы в листинге 3.4 мы забыли объявить переменную `asleep` как волатильную, то серверная JVM могла бы вынуть проверку из цикла (превратив его в бесконечный цикл), а клиентская JVM — не могла бы. Бесконечный цикл, который проявляется при разработке, стоит намного дешевле, чем тот, который обнаруживает себя в процессе эксплуатации кода.

видимость изменений во флажке `asleep` может и блокировка, но она сделает код громоздким.

Листинг 3.4. Подсчет овец

```
volatile boolean asleep;
...
    while (!asleep)
        countSomeSheep();
```

Волатильные переменные удобны и часто используются в качестве флажка завершения, прерывания или статуса. Однако механизм `volatile` недостаточно силен, для того чтобы сделать операцию приращения (`count++`) атомарной в многопоточной среде. (Атомарные переменные обеспечивают атомарную поддержку операции «прочитать, изменить, записать» и часто могут использоваться как «более качественные волатильные переменные»; см. главу 15.)

> Блокировка может гарантировать как видимость, так и атомарность, а волатильные переменные гарантируют только видимость.

Использование волатильных переменных оправданно при следующих условиях:

- записи в переменную не зависят от ее текущего значения, либо есть гарантия, что значения переменной обновляются только одним потоком;
- переменная не участвует в инвариантах с другими переменными состояния;
- при обращении к переменной заранее не требуется блокировка.

3.2. Публикация и ускользание

Публикация (publishing) объекта означает его доступность за пределами текущей области действия. Например, ссылка на объект может позволить другому коду вернуть его из неприватного метода или передать его методу в другом классе. Публикация переменных внутреннего состояния препятствует инкапсуляции и соблюдению инвариантов, а публикация объектов до момента их полного конструирования ставит под угрозу по-

токобезопасность. Объект, который не вовремя публикуется, называется *ускользнувшим* (escaped). Рассмотрим, как объект может ускользнуть.

Самая простая форма публикации — это ссылка в публичном статическом поле. Метод `initialize` создает и публикует экземпляр нового хеш-множества, сохраняя ссылку на него в `knownSecrets`.

Листинг 3.5. Публикация объекта

```
public static Set<Secret> knownSecrets;

public void initialize() {
    knownSecrets = new HashSet<Secret>();
}
```

Публикация одного объекта может косвенно публиковать другие. Если вы добавите `Secret` в опубликованное множество `knownSecrets`, то дополнение также опубликуется, потому что любой код может выполнить итеративный обход множества и получить ссылку на новый `Secret`. Схожим образом возвращение ссылки из неприватного метода заодно публикует возвращаемый объект. Класс `UnsafeStates` в листинге 3.6 старается опубликовать массив аббревиатур приватно.

Листинг 3.6. Позволяем ускользнуть внутреннему мутируемому состоянию. *Так делать не следует*

```
class UnsafeStates {
    private String[] states = new String[] {
        "AK", "AL" ...
    };
    public String[] getStates() { return states; }
}
```

Любой вызывающий массив `states` элемент кода может изменить его содержимое. В данном случае предположительно приватный объект стал публичным.

Публикация объекта также открывает доступ к любым объектам, на которые ссылаются его неприватные поля.

С точки зрения класса *C*, *чужой* (alien) метод — это метод, поведение которого не полностью определено классом *C*. Сюда входят методы, находящиеся в других классах, а также переопределяемые методы (не приватные и не финальные) в самом *C*. Передача объекта чужому методу должна рассматриваться как публикация этого объекта.

Неважно, делает ли другой поток что-то с опубликованной ссылкой или нет, потому что остается риск ее неправильного использования[1]. Это веская причина для применения инкапсуляции, которая позволяет анализировать правильность программы и затрудняет случайное нарушение проектных ограничений.

Еще одним механизмом публикации объекта или его внутреннего состояния является публикация экземпляра внутреннего класса, как показано в листинге 3.7. Когда класс `ThisEscape` публикует слушателя `EventListener`, он неявно публикует и окаймляющий его экземпляр `ThisEscape`, потому что экземпляры внутреннего класса содержат скрытую ссылку на него.

Листинг 3.7. Неявное разрешение ссылке this ускользнуть. *Так делать не следует*

```
public class ThisEscape {
    public ThisEscape(EventSource source) {
        source.registerListener(
            new EventListener() {
                public void onEvent(Event e) {
                    doSomething(e);
                }
            });
    }
}
```

[1] Если кто-то украдет ваш пароль и опубликует его в новостной группе alt.free-passwords, то эта информация ускользнет: независимо от того, воспользовался ли (уже) кто-то этими учетными данными для нанесения вреда или нет, ваша учетная запись по-прежнему находится под угрозой. Публикация ссылки представляет такой же риск.

3.2.1. Приемы безопасного конструирования

Объект находится в предсказуемом и непротиворечивом состоянии только при полноценности его конструктора. В противном случае возникает риск публикации неполно сконструированного объекта, даже если *публикация является последней инструкцией в конструкторе*. Объект с ускользнувшей ссылкой `this` считается *ненадлежаще сконструированным*[1].

Не позволяйте ссылке `this` ускользнуть во время конструирования.

Распространенной ошибкой, позволяющей ссылке `this` ускользнуть, является запуск потока из конструктора. Когда объект создает поток из своего конструктора, он почти всегда делится своей ссылкой `this` с новым потоком, явно или неявно. Тогда новый поток видит владеющий объект до своего окончательного конструирования. Нет ничего плохого в *создании* потока в конструкторе, но лучше не *запускать* поток сразу. Вместо этого добавьте метод `start` или `initialize`, запускающий собственный поток. (Дополнительные сведения о вопросах жизненного цикла службы см. в главе 7.) Вызов переопределяемого экземплярного метода (который не является ни приватным, ни финальным) из конструктора также может дать ссылке `this` ускользнуть.

Чтобы зарегистрировать слушателя событий или запустить поток из конструктора, воспользуйтесь приватным конструктором и публичным фабричным методом, как показано в `SafeListener` в листинге 3.8.

Листинг 3.8. Использование фабричного метода для предотвращения ускользания ссылки this во время конструирования

```
public class SafeListener {
    private final EventListener listener;

    private SafeListener() {
        listener = new EventListener() {
            public void onEvent(Event e) {
                doSomething(e);
```

[1] Ссылка `this` не должна ускользать из *потока* до тех пор, пока конструктор не возвратится. Ссылка `this` может храниться где-то конструктором, при условии что она не *используется* другим потоком до завершения конструирования. `SafeListener` в листинге 3.8 использует это техническое решение.

```
            }
        };
    }

    public static SafeListener newInstance(EventSource source) {
        SafeListener safe = new SafeListener();
        source.registerListener(safe.listener);
        return safe;
    }
}
```

3.3. Ограничение одним потоком

Обращение к совместным мутируемым данным требует трудозатрат на синхронизацию, поэтому попробуем *не использовать данные совместно*. *Ограничение одним потоком* (thread confinement), является одним из самых простых технических решений достижения потокобезопасности. Однопоточное использование объекта потокобезопасно, даже если сам ограниченный объект безопасным не является [CPJ 2.3.2].

В платформе Swing используется ограничение непотокобезопасных визуальных компонентов и объектов модели данных одним потоком диспетчеризации событий. То есть код, выполняемый в потоках, отличных от событийного, не обращается к этим объектам. (Swing предоставляет механизм планирования `invokeLater` для выполнения объектов `Runnable` в событийном потоке.) Многие ошибки конкурентности в приложениях Swing возникают из-за неправильного использования ограниченных объектов.

Еще одним примером ограничения одним потоком является объединение в пул JDBC (Java Database Connectivity) объектов `Connection`. Спецификация JDBC не требует, чтобы объекты соединения были потокобезопасными[1]. В типичных серверных приложениях поток приобретает соединение из пула, использует его для обработки одиночного запроса и возвращает его. Большинство запросов, таких как сервлетные запросы или вызовы EJB (Enterprise JavaBeans) обрабатываются синхронно одним потоком, и пул не распределяет соединение потоку, до тех пор пока оно

[1] Реализации пула соединений, предоставляемые серверами приложений, являются потокобезопасными. Доступ к пулам соединений неизбежно осуществляется из многочисленных потоков, поэтому непотокобезопасная реализация не имеет смысла.

не будет возвращено другим потоком. Этот паттерн управления соединениями неявно ограничивает объект `Connection` одним потоком на время выполнения запроса.

Сам язык не содержит механизмов ограничения объекта одним потоком. Такое ограничение станет элементом проекта вашей программы. Языковые и ядерные библиотеки предоставляют механизмы поддержки ограничения — локальные переменные и класс `ThreadLocal`, — но ответственность за их реализацию лежит на разработчике.

3.3.1. Узкоспециальное ограничение одним потоком

Узкоспециальное ограничение одним потоком (ad-hoc thread confinement) необходимо, когда ни один из элементов функционала языка не помогает ограничить объект целевым потоком. На самом деле ссылки на ограниченные одним потоком объекты, такие как визуальные компоненты или модели данных в GUI-приложениях, часто хранятся в публичных полях.

Решение использовать ограничение обычно связано с желанием реализовать некую подсистему, такую как GUI, в качестве однопоточной. Простота однопоточных подсистем иногда ценнее прочности, которую не может обеспечить узкоспециальное ограничение[1].

Безопасное выполнение операций «прочитать, изменить, записать» с совместными волатильными переменными возможно, только если переменная записана одним потоком. В этом случае предотвращены состояния гонки и гарантирована видимость наиболее актуального значения переменной.

Более прочными формами ограничения одним потоком являются ограничение стеком и `ThreadLocal`.

3.3.2. Ограничение стеком

При *ограничении стеком* (stack confinement) объект может быть достигнут только через локальные переменные. Они внутренне ограничены стеком выполняющего потока, в котором находятся. Ограничение стеком также называют *внутрипоточным* или *локальным* для потока использованием.

---

[1] Еще одна причина сделать подсистему однопоточной заключается в предотвращении взаимной блокировки. Однопоточные подсистемы рассматриваются в главе 9.

Примитивные локальные переменные, такие как `numPairs` в `loadTheArk` в листинге 3.9, всегда ограничены стеком, и получить на них ссылку невозможно.

Листинг 3.9. Ограничение локальных примитивных и ссылочных переменных одним потоком

```
public int loadTheArk(Collection<Animal> candidates) {
    SortedSet<Animal> animals;
    int numPairs = 0;
    Animal candidate = null;

    // не дайте animals ускользнуть!
    animals = new TreeSet<Animal>(new SpeciesGenderComparator());
    animals.addAll(candidates);
    for (Animal a : animals) {
        if (candidate == null || !candidate.isPotentialMate(a))
            candidate = a;
        else {
            ark.load(new AnimalPair(candidate, a));
            ++numPairs;
            candidate = null;
        }
    }
    return numPairs;
}
```

Ограничение стеком объектных ссылок требует особого внимания. Чтобы объект, на который указывает ссылка, не ускользал, нужно в `loadTheArk` создать экземпляр `TreeSet` и сохранить ссылку на него в переменной `animals`. Есть только одна ссылка на объект `Set`, содержащаяся в локальной переменной и, следовательно, ограниченная выполняющим потоком. Но если опубликовать ссылку на `Set` (или ее внутренний элемент), то ограничение нарушится, и `animals` ускользнет.

Будьте аккуратны: всегда документируйте ограничение выполняющим потоком либо информацию о непотокобезопасности объекта, чтобы будущие сопроводители не позволили объекту ускользнуть.

3.3.3. ThreadLocal

Класс `ThreadLocal` позволяет ассоциировать каждое значение в потоке с объектом, владеющим этим значением. Он предоставляет методы до-

ступа get и set, поддерживающие отдельную копию значения для каждого потока, который его использует, поэтому get возвращает самое последнее значение, переданное в set *из выполняющегося в настоящее время потока*.

Однопоточное приложение может поддерживать глобальное подключение к базе данных, инициализируемое при запуске, чтобы избежать необходимости передавать объект Connection в каждый метод. Если соединения JDBC непотокобезопасны, многопоточное приложение, использующее глобальное соединение без дополнительной координации, также не является потокобезопасным. Используя класс ThreadLocal для хранения соединения JDBC, как в ConnectionHolder в листинге 3.10, вы обеспечите каждому потоку свое собственное соединение.

Листинг 3.10. Использование класса ThreadLocal для ограничения одним потоком

```
private static ThreadLocal<Connection> connectionHolder
    = new ThreadLocal<Connection>() {
        public Connection initialValue() {
            return DriverManager.getConnection(DB_URL);
        }
    };

public static Connection getConnection() {
    return connectionHolder.get();
}
```

Если часто используемая операция требует временного объекта, такого как буфер, описанное решение позволит обойтись без повторного выделения такого объекта при каждом вызове. До появления Java 5.0, в методе Integer.toString использовался класс ThreadLocal для хранения такого 12-байтового буфера[1].

Когда поток вызывает метод ThreadLocal.get в первый раз, переменная initialvalue предоставляет начальное значение для данного потока. ThreadLocal<T> не является объектом, владеющим Map<Thread,T>, который хранит специфичные для потока значения. Эти значения хранятся

[1] Чтобы это техническое решение не ухудшило производительность, операция должна выполняться очень часто, а выделение памяти не должно быть дорогим. В Java 5.0 этот подход был заменен более простым подходом на основе выделения нового буфера для каждого вызова.

в самом объекте `Thread`, и когда поток терминируется, они могут быть собраны сборщиком мусора.

Когда вы переносите однопоточное приложение в многопоточную среду, вы можете сохранить потокобезопасность, конвертировав совместные глобальные переменные в объекты `ThreadLocal`, если их семантика позволяет это сделать. В масштабе всего приложения будет не так полезен кэш, превращенный в ряд локальных кэшей.

Класс `ThreadLocal` широко используется в реализации структур приложений. Например, контейнеры J2EE ассоциируют транзакционный контекст с выполняющим потоком в течение всего времени вызова EJB: `ThreadLocal` предоставляет транзакционный контекст структурному коду, сообщая, какая транзакция активна. Это уменьшает необходимость передачи информации о выполняющем контексте в каждый метод, но связывает любой код, использующий этот механизм, со структурой.

Помните, что класс `ThreadLocal` — это не лицензия на использование глобальных переменных и не средство создания «скрытых» аргументов в методах. Локальные для потоков переменные тоже могут создавать скрытые связи между классами, поэтому их следует использовать с осторожностью.

3.4. Немутируемость

Избежать синхронизации также можно использованием *немутируемых* (immutable) объектов [EJ пункт 13]. Почти все описанные нами до сих пор сбои атомарности и видимости, такие как устаревшие значения, утеря обновлений или противоречивое состояние объекта, связаны с тем, что многочисленные потоки обращаются к одному и тому же мутируемому состоянию в одно и то же время.

Состояние немутируемого объкта нельзя изменить после его конструирования, поэтому установленные в нем инварианты всегда соблюдаются.

> Немутируемые объекты всегда являются потокобезопасными.

Немутируемые объекты — *просты*. Они могут находиться только в одном состоянии, которое тщательно контролируется конструктором.

Немутируемые объекты — *безопасны*. Они не могут быть изменены вредоносным или дефектным кодом, и их можно совместно использовать и публиковать без защитных копий [EJ пункт 24].

Ни спецификация языка, ни модель памяти Java формально не определяют немутируемость. Объект, все поля которого являются финальными, может быть мутируемым, так как финальные поля могут содержать ссылки на мутируемые объекты.

Объект является *немутируемым*, если:

- его состояние невозможно изменить после конструирования;
- все поля являются финальными[1];
- он *надлежаще сконструирован* (ссылка `this` не ускользает).

Немутируемые объекты могут использовать мутируемые объекты для управления своим состоянием, как показано в листинге 3.11. Объект `Set`, который хранит имена, является мутируемым, но построение класса `ThreeStooges` делает невозможным изменение этого объекта после конструирования. Ссылка `stooges` является финальной, и состояние объекта достигается через финальное поле. Конструктор не делает ничего, что могло бы сделать ссылку `this` доступной для кода, отличного от конструктора и элемента кода, вызывающего объект.

Листинг 3.11. Немутируемый класс, построенный из мутируемых базовых объектов

```
@Immutable
public final class ThreeStooges {
    private final Set<String> stooges = new HashSet<String>();
```

[1] Технически возможно иметь немутируемый объект без финальности всех полей — `String` является таким классом, — но его использование требует глубокого понимания модели памяти Java. (Для любопытных: `String` лениво вычисляет хеш-код при первом вызове метода `hashCode` и кэширует его в нефинальном поле, которое может принимать только одно значение не по умолчанию, всегда одинаковое при вычислении, потому что оно детерминированно выводится из немутируемого состояния. Не пытайтесь повторить.)

```
    public ThreeStooges() {
        stooges.add("Moe");
        stooges.add("Larry");
        stooges.add("Curly");
    }

    public boolean isStooge(String name) {
        return stooges.contains(name);
    }
}
```

Поскольку состояние программы постоянно меняется, можно предположить, что немутируемые объекты имеют ограниченное применение, но это не так. Существует разница между немутируемым *объектом* и *ссылкой* на него. Состояние программы, хранящееся в немутируемых объектах, может быть обновлено путем замены немутируемых объектов новым экземпляром, содержащим новое состояние. Следующий раздел предлагает пример этого технического решения[1].

3.4.1. Финальные поля

Ключевое слово `final`, более ограниченная версия механизма `const` из языка C++, поддерживает конструирование немутируемых объектов. Финальные поля не могут быть изменены (в отличие от мутируемых объектов, на которые они ссылаются), а их специальная семантика в модели памяти Java гарантирует *безопасность инициализации* (см. раздел 3.5.2), которая позволяет свободно обращаться к немутируемым объектам и совместно их использовать без синхронизации.

Даже если объект является мутируемым, объявление некоторых его полей финальными может упростить рассуждения о его состоянии, так как ограничение мутируемости объекта лимитирует множество его возможных состояний и документирует для сопроводителя тот факт, что они не должны изменяться.

---

[1] Многие разработчики опасаются, что такой подход создаст проблемы с производительностью, но эти опасения обычно необоснованные. Выделение памяти обходится дешевле, чем может показаться, а немутируемые объекты обеспечивают дополнительные преимущества в производительности, такие как сокращение количества блокировок и защитных копий и оптимизация поколенческого сбора мусора.

Объявляйте поля приватными, если они не нуждаются в большей видимости [EJ пункт 12], и финальными — если они не должны быть мутируемыми.

3.4.2. Пример: использование volatile для публикации немутируемых объектов

В классе `UnsafeCachingFactorizer` на с. 59 мы использовали два объекта `AtomicReference` для хранения последнего числа и множителей, но мы не смогли доставлять или обновлять родственные значения атомарно. Использование волатильных переменных для этих значений тоже не гарантировало бы потокобезопасность. Но немутируемые объекты иногда могут обеспечивать слабую форму атомарности.

Сервлет разложения на множители выполняет две операции, которые должны быть атомарными: обновление кэшированного результата и условную доставку кэшированных множителей, если кэшированное число совпадает с запрошенным числом. Всякий раз, когда группа родственных элементов данных должна действовать атомарно, рассмотрите возможность создания для них немутируемого держателя, такого как `OneValueCache`[1] в листинге 3.12.

Листинг 3.12. Немутируемый держатель для кэширования числа и его множителей

```
@Immutable
class OneValueCache {
    private final BigInteger lastNumber;
    private final BigInteger[] lastFactors;

    public OneValueCache(BigInteger i,
                         BigInteger[] factors) {
        lastNumber = i;
        lastFactors = Arrays.copyOf(factors, factors.length);
    }

    public BigInteger[] getFactors(BigInteger i) {
        if (lastNumber == null || !lastNumber.equals(i))
```

[1] Объект `OneValueCache` не был бы немутируемым без вызовов `copyOf` в конструкторе и методе `get`. Метод `Arrays.copyOf` был добавлен в качестве удобства в Java 6; метод `clone` также будет работать.

```
            return null;
        else
            return Arrays.copyOf(lastFactors, lastFactors.length);
    }
}
```

Состояния гонки при доступе или обновлении родственных переменных могут быть устранены в немутируемом держателе даже без блокировки: когда поток приобретет на него ссылку, другой поток уже не сможет изменить его состояние. Чтобы обновить переменные, потребуется создать новый держатель, но потоки, работающие с предыдущим держателем, продолжат видеть его в непротиворечивом состоянии.

Класс `VolatileCachedFactorizer` в листинге 3.13 использует кэш `OneValueCache` для хранения кэшированного числа и множителей. Когда поток задаст для волатильного поля `cache` ссылку на новый объект `OneValueCache`, новые кэшированные данные станут видимыми для других потоков.

Операции, связанные с кэшем, не могут взаимодействовать, поскольку объект `OneValueCache` является немутируемым, а поле `cache` доступно только один раз в каждой из релевантных ветвей кода. Комбинация немутируемого держателя родственных через инвариант переменных состояния и мутируемой ссылки, обеспечивающей его своевременную видимость, позволяет классу `VolatileCachedFactorizer` быть потокобезопасным без блокировки.

Листинг 3.13. Кэширование последнего результата с помощью изменчивой ссылки на немутируемый держатель объекта

```
@ThreadSafe
public class VolatileCachedFactorizer implements Servlet
{
    private volatile OneValueCache cache =
        new OneValueCache(null, null);

    public void service(ServletRequest req, ServletResponse resp) {
        BigInteger i = extractFromRequest(req);
        BigInteger[] factors = cache.getFactors(i);
        if (factors == null) {
            factors = factor(i);
            cache = new OneValueCache(i, factors);
        }
        encodeIntoResponse(resp, factors);
    }
}
```

3.5. Безопасная публикация

Иногда *необходимо* использовать объекты в потоках совместно. Как показано в листинге 3.14, простого сохранения объектной ссылки в публичном поле *недостаточно*, для того чтобы опубликовать объект безопасно.

Листинг 3.14. Публикация объекта без надлежащей синхронизации. *Так делать не следует*

```
// Небезопасная публикация
public Holder holder;

public void initialize() {
    holder = new Holder(42);
}
```

Вы будете удивлены, но из-за проблем видимости держатель `Holder` может показаться другому потоку в противоречивом состоянии, даже если его инварианты были правильно установлены его конструктором! Именно ненадлежащая публикация заставит другой поток наблюдать *частично сконструированный объект*.

3.5.1. Ненадлежащая публикация: хорошие объекты становятся плохими

Не стоит надеяться на целостность частично сконструированных объектов. Наблюдающий поток может увидеть сначала объект в противоречивом состоянии, а затем внезапное изменение состояния. Если держатель `Holder` в листинге 3.15 будет опубликован так, как в листинге 3.14, при вызове метода `assertSanity` возникнет ошибка `AssertionError`![1]

[1] Проблема здесь не в самом классе `Holder`, а в его публикации. `Holder` может быть сделан невосприимчивым к ненадлежащей публикации, если объявить поле *n* финальным, что сделает класс `Holder` немутируемым (см. раздел 3.5.2).

Листинг 3.15. Риск возникновения сбоя при ненадлежащей публикации

```
public class Holder {
    private int n;

    public Holder(int n) { this.n = n; }

    public void assertSanity() {
        if (n != n)
            throw new AssertionError("Эта инструкция является ложной.");
    }
}
```

Поскольку синхронизация не использовалась для того, чтобы сделать класс `Holder` видимым для других потоков, другие потоки увидят либо устаревшее значение в поле `holder` (пустую ссылку `null` или другое), либо актуальное значение ссылки на `holder`, но устаревшие значения *состояния* `Holder`[1], либо устаревшее значение при первом чтении поля, а в следующий раз — более актуальное значение.

Да, без достаточной синхронизации могут происходить очень странные вещи, если данные используются потоками совместно.

3.5.2. Немутируемые объекты и безопасность при инициализации

Модель памяти Java предлагает специальную гарантию *безопасности при инициализации* (initialization safety) для совместного использования немутируемых объектов.

К немутируемым объектам можно безопасно обращаться, *даже когда синхронизация не используется для публикации объектной ссылки*, только если соблюдены три условия: состояние немутируемое, все поля финаль-

[1] Можно предположить, что значения полей, заданные в конструкторе, являются первыми значениями, записанными в эти поля, и увидеть в них устаревшие значения невозможно. Но в действительности до начала работы конструкторов подклассов конструктор `Object` записывает во все поля значения по умолчанию.

ные и конструирование надлежащее. (Если класс `Holder` в листинге 3.15 является немутируемым, то метод `assertSanity` не может выдать ошибку `AssertionError`, даже если `Holder` не был надлежаще опубликован.)

> *Немутируемые* объекты могут безопасно использоваться потоками без дополнительной синхронизации, даже когда синхронизация для их публикации не используется.

Однако если финальные поля ссылаются на мутируемые объекты, то синхронизация по-прежнему необходима для доступа к состоянию этих объектов.

3.5.3. Приемы безопасной публикации

Объекты, которые не являются немутируемыми, должны быть *безопасно опубликованы*, что обычно влечет за собой синхронизацию, осуществляемую как публикующим, так и потребляющим потоками. Сначала мы обеспечим потребляющему потоку видимость объекта в его опубликованном состоянии, после чего займемся видимостью изменений, сделанных после публикации.

> Безопасную публикацию объекта, при которой ссылка на него и его состояние видна всем потокам в одно и то же время, можно провести с помощью:
>
> - инициализации объектной ссылки из статического инициализатора;
> - сохранения ссылки на него в волатильном поле либо в `AtomicReference`;
> - сохранения ссылки на него в финальном поле надлежаще сконструированного объекта;
> - сохранения ссылки на него в поле, которое надлежаще защищается замком.

Если поток *A* помещает объект *X* в потокобезопасную коллекцию `Vector` или `synchronizedList`, а затем поток *B* извлекает его, то *B* гарантированно видит состояние *X* в том виде, в каком *A* его оставил, даже если передаю-

щий код не имеет явной синхронизации. Потокобезопасные библиотечные коллекции предлагают следующие гарантии безопасной публикации:

- ключ или значение, размещенные в `Hashtable`, `synchronizedMap` либо `ConcurrentMap`, безопасно публикуются в любом потоке, который извлекает их из ассоциативного массива `Map` (напрямую или через итератор);
- элемент, размещенный в `Vector`, `CopyOnWriteArrayList`, `CopyOnWriteArraySet`, `synchronizedList` либо `synchronizedSet`, безопасно публикуется в любом потоке, который извлекает его из коллекции;
- элемент, размещенный в `BlockingQueue` либо `ConcurrentLinkedQueue`, безопасно публикуется в любом потоке, который извлекает его из очереди.

Другие механизмы эстафетной передачи в библиотеке классов, такие как `Future` и `Exchanger`, также образуют безопасную публикацию, которую вы увидите позднее.

Использование статического инициализатора является самым простым и безопасным способом публикации объектов, которые могут конструироваться статически:

```
public static Holder holder = new Holder(42);
```

Статические инициализаторы выполняются JVM-машиной во время инициализации класса, гарантируя дальнейшую безопасную публикацию инициализированных объектов [JLS 12.4.2].

3.5.4. Фактически немутируемые объекты

Механизмы безопасной публикации гарантируют безопасное обращение потоков к состоянию объекта при условии видимости ссылки на него и отсутствии дальнейших изменений.

Объекты, которые не являются немутируемыми, но состояние которых не будет изменено после публикации, называются *фактически немутируемыми* (effectively immutable). Они не соответствуют определению немутируемости из раздела 3.4: немутируемыми их воспринимает программа. Их использование упрощает разработку и повышает производительность за счет уменьшения необходимости в синхронизации.

Безопасно опубликованные *фактически немутируемые* объекты могут безопасно использоваться любым потоком без дополнительной синхронизации.

Например, класс `Date` является мутируемым[1], но если вы используете его как немутируемый, то сэкономите на блокировке. Предположим, вы хотите поддержать ассоциативный массив `Map`, хранящий время последнего входа каждого пользователя в систему:

```
public Map<String, Date> lastLogin =
    Collections.synchronizedMap(new HashMap<String, Date>());
```

Если значения `Date` не изменятся после их размещения в массиве `Map`, то синхронизации в реализации `synchronizedMap` достаточно для безопасной публикации значений `Date`, и при доступе к ним дополнительная синхронизация не потребуется.

3.5.5. Мутируемые объекты

Если объект может быть изменен после конструирования, то безопасная публикация с использованием синхронизации обеспечит видимость его состояния, актуального на время публикации. Совместное использование таких объектов подразумевает их безопасную публикацию *и* потокобезопасность или защищенность замком.

Требования к публикации объекта зависят от его мутируемости:

- *немутируемые объекты* могут быть опубликованы любым механизмом;
- *фактически немутируемые объекты* должны быть безопасно опубликованы;
- *мутируемые объекты* должны быть безопасно опубликованы и быть либо потокобезопасными, либо защищенными замком.

[1] Вероятно, это было недочетом при проектировании библиотеки классов.

3.5.6. Безопасное совместное использование объектов

Всякий раз, когда вы приобретаете ссылку на объект, вы должны знать, что именно вам разрешено с ним делать. Нужно ли перед его использованием приобрести замок? Разрешено ли изменять его состояние или же только читать его? Многие ошибки конкурентности возникают из-за непонимания этих «правил ведения боевых действий» для совместного объекта. Когда вы публикуете объект, следует задокументировать то, как к нему можно обращаться.

Ниже приведены наиболее полезные политики для применения и совместного использования объектов в конкурентной программе.

Ограничение одним потоком. Объект, ограниченный одним потоком, принадлежит эксклюзивно владеющему потоку, который может его изменять.

Совместный доступ только для чтения. Потоки могут обращаться к объекту, предназначенному только для чтения, конкурентно, без дополнительной синхронизации и возможности его изменять. Совместные объекты только для чтения включают немутируемые и фактически немутируемые объекты.

Совместная потокобезопасность. Потокобезопасный объект выполняет синхронизацию внутренне, поэтому потоки могут свободно обращаться к нему через его публичный интерфейс без дополнительной синхронизации.

Защищенность. С удержанием конкретного замка можно обращаться к объекту, инкапсулированному в другие потокобезопасные объекты, а также к опубликованному объекту, защищенному замком.

4 Компоновка объектов

До сих пор мы рассматривали низкоуровневые основы потокобезопасности и синхронизации. Но мы не хотим анализировать доступы к памяти — нас интересует возможность брать потокобезопасные компоненты и компоновать программу или ее части. В этой главе мы рассмотрим паттерны для безопасного структурирования классов.

4.1. Проектирование потокобезопасного класса

Если написать потокобезопасную программу, которая хранит свое состояние в публичных статических полях, ее будет трудно проверить или безопасно изменить. Поэтому лучше использовать надлежащую инкапсуляцию, которая позволит определить потокобезопасность класса без полной проверки программы.

Проектирование потокобезопасного класса включает три этапа:

- идентификация переменных, формирующих состояние объекта;
- идентификация инвариантов, ограничивающих переменные состояния;
- создание политики для управления конкурентным доступом к состоянию объекта.

Если все поля объекта имеют примитивный тип, то можно говорить, что они содержат все состояние объекта. В листинге 4.1 единственное поле `value` содержит состояние счетчика. Несколько примитивных полей представляют собой *n*-элементый кортеж значений. Состоянием двумерной точки `Point` является ее значение (*x*, *y*). Состояние списка `LinkedList` включает состояния всех объектов связных узлов, входящих в список, если содержит поля-ссылки на эти узлы.

Листинг 4.1. Простой потокобезопасный счетчик с использованием Java-паттерна — монитор

```
@ThreadSafe
public final class Counter {
    @GuardedBy("this") private long value = 0;

    public synchronized long getValue() {
        return value;
    }
    public synchronized long increment() {
        if (value == Long.MAX_VALUE)
            throw new IllegalStateException("переполнение счетчика");
        return ++value;
    }
}
```

Политика синхронизации (synchronization policy) определяет, как объект координирует доступ к своему состоянию, не нарушая его инвариантов или постусловий. Она указывает, какая комбинация немутируемости, ограничения одним потоком и блокировки используется сейчас, какие переменные защищены и какими замками. Ее документирование упрощает анализ и сопровождение класса.

4.1.1. Сбор требований к синхронизации

Потокобезопасность класса подразумевает соблюдение его инвариантов в условиях конкурентного доступа. Объекты и переменные имеют *пространства состояний* (state space) — диапазоны возможных состояний, которые лучше уменьшить с помощью финальных полей для более простого анализа.

Многие классы имеют инварианты, идентифицирующие состояния как *допустимые* (valid) или *недопустимые* (invalid). Поле `value` в `Counter`

имеет тип `long`. Состояние пространства типа `long` находится в диапазоне от `Long.MIN_VALUE` до `Long.MAX_VALUE`, но `Counter` ограничивает `value` только положительными значениями.

Постусловия в операциях могут идентифицировать *переход из состояния в состояние* (state transitions) как недопустимый. Если текущее состояние объекта `Counter` равно 17, то *единственным* допустимым следующим состоянием должно быть 18.

Названные ограничения создают дополнительные требования к синхронизации или инкапсуляции. Если некие состояния признаны недопустимыми, то лежащие в их основе переменные состояния должны быть инкапсулированы, иначе клиентский код может поставить в недопустимое состояние весь объект. Если операция имеет недопустимые переходы из состояния в состояние, то она должна быть атомарной. Помните, что если класс не накладывает ограничений, то можно ослабить требования к инкапсуляции или сериализации, чтобы улучшить гибкость и производительность.

В свою очередь, класс диапазона чисел, такой как `NumberRange` в листинге 4.10, ограничивает переменные состояния условием, что нижняя граница значений меньше или равна верхней. А многопеременные инварианты требуют, чтобы родственные переменные доставлялись или обновлялись в единой атомарной операции. Вы не сможете обновить, освободить и повторно защитить замком одну из родственных переменных, так как объект попадет в недопустимое состояние.

> Обеспечивайте потокобезопасность, учитывая роль инвариантов и постусловий.

4.1.2. Операции, зависимые от состояния

Помимо инвариантов классов и постусловий методов, ограничивающих допустимые состояния объекта и его переходы из состояния в состояние, некоторые объекты имеют методы с *предусловиями*, основанными на состоянии. Предусловия, например, запрещают удалять элемент из пустой очереди, требуя для удаления состояние очереди «не пусто». Операции с предусловиями, основанными на состоянии, именуются *зависимыми от состояния* (state-dependent) [CPJ 3].

Если в однопоточной программе предусловие не соблюдается, то операция завершается безрезультатно. Конкурентные же программы ожидают появление потока, для которого предусловие станет истинным, и затем продолжают выполнение операции.

Встроенные механизмы ожидания момента истинности предусловия — ожидание `wait` и уведомление `notify` — тесно связаны с внутренней блокировкой и могут быть сложны в применении. Для включения в программу таких механизмов часто используют библиотечные классы, такие как блокирующие очереди или семафоры. Блокирующие библиотечные классы (`BlockingQueue`, `Semaphore` и другие *синхронизаторы*) рассматриваются в главе 5, а создание зависимых от состояния классов с помощью платформенной библиотеки описано в главе 14.

4.1.3. Владение состоянием

В разделе 4.1 мы косвенно указали на то, что состояние объекта может быть подмножеством полей в объектном графе. Почему подмножеством? При каких условиях поля объекта достижимы, *не* являясь частью состояния этого объекта?

При определении переменных, формирующих состояние объекта, мы учитываем только те данные, которыми объект *владеет*. Владение не воплощено в языке явным образом, а является элементом проектирования класса. Если вы выделяете место под объект `HashMap`, то, заполняя его, создаете не только `HashMap`, но и ряд объектов `Map.Entry` и, возможно, другие внутренние объекты. Все они объединяются общим логическим состоянием, даже если реализуются отдельно.

В C++, передавая объект методу, вы должны продумать, отдаете ли вы право владения, выдаете ли краткосрочный кредит или планируете долгосрочное совместное владение. В Java возможны все эти модели, но сборщик мусора снижает стоимость многих ошибок при совместном использовании ссылок, поэтому точный выбор не требуется.

Во многих случаях владение связано с инкапсуляцией — объект инкапсулирует состояние, которым владеет, и владеет состоянием, которое инкапсулирует. Именно владелец переменной состояния принимает решение о замковом протоколе, используемом для поддержания целостности ее состояния. Из факта владения вытекает обязанность контроля, которая после публикации ссылки на мутируемый объект приобретет

совместный характер. Класс не владеет объектами, поставляемыми в его методы или конструкторы, если сам метод не предназначен для явной передачи права владения.

Коллекционные классы часто демонстрируют форму раздельного владения: они владеют состоянием коллекции, но хранящиеся у себя объекты передают клиентскому коду. Например, объект `ServletContext` предоставляет сервлетам контейнер для регистрации и извлечения `Map`-подобных объектов. `ServletContext` должен быть потокобезопасным, поскольку к нему обращаются многочисленные потоки. Но синхронизацию при вызове методов извлечения `setAttribute` и `getAttribute` сервлеты использовать не обязаны: она потребуется им только при непосредственной работе с объектами, хранящимися в `ServletContext`[1].

4.2. Ограничение одним экземпляром

Если объект не является потокобезопасным, то ограничение одним потоком или блокировка позволяют безопасно использовать его в многопоточной программе.

В свою очередь, инкапсуляция упрощает создание потокобезопасных классов, обеспечивая *ограничение одним экземпляром* (instance confinement), которое часто называется *запиранием* (confinement) [CPJ 2.3.3]. Объект инкапсулируется в другой объект, к которому имеют доступ только некоторые известные ветви кода, а не вся программа. Сочетание ограничения одним экземпляром и блокировки может обеспечить безопасное использование непотокобезопасных объектов.

> Инкапсуляция данных в объекте ограничивает доступ к ним только методами объекта, что упрощает обеспечение постоянного доступа с удержанием замка.

---

[1] Интересно, что к объекту `HttpSession`, который выполняет аналогичную функцию в сервлетной платформе, могут предъявляться более строгие требования. Сервлетный контейнер обращается к объектам в `HttpSession` для их сериализации с целью репликации или пассивации как к веб-приложению, поэтому их потокобезопасность обязательна. (Мы говорим «могут предъявляться», так как репликация и пассивация находятся вне спецификации сервлета, но являются распространенным функционалом сервлетных контейнеров.)

Ограниченные экземпляром объекты не должны выходить за пределы своей области действия. Объект может быть ограничен экземпляром класса (например, приватным членом класса), лексической областью (например, локальной переменной) или потоком (например, объектом, который передается из метода в метод внутри одного потока). Публикуйте объекты осторожно.

Класс `PersonSet` в листинге 4.2 иллюстрирует, как ограничение одним экземпляром и блокировка работают вместе. Состояние объекта `PersonSet` управляется непотокобезопасным объектом `HashSet`. Но `mySet` является приватным и не может ускользнуть, поэтому объект `HashSet` ограничен объектом `PersonSet`. Единственными ветвями кода, которые могут обратиться к `mySet`, являются методы `addPerson` и `containsPerson`, и каждый из них приобретает замок на `PersonSet`, все состояние которого защищено его внутренним замком.

Листинг 4.2. Использование ограничения одним экземпляром

```
@ThreadSafe
public class PersonSet {
    @GuardedBy("this")
    private final Set<Person> mySet = new HashSet<Person>();

    public synchronized void addPerson(Person p) {
        mySet.add(p);
    }

    public synchronized boolean containsPerson(Person p) {
        return mySet.contains(p);
    }
}
```

Если класс `Person` является мутируемым, то доступ к объекту `Person` из множества `PersonSet` потребует дополнительной синхронизации. Проектирование класса `Person` потокобезопасным будет надежнее, чем защита объектов `Person` замком.

Ограничение одним экземпляром допускает гибкость в выборе замковой стратегии. В нашем примере `PersonSet` использует внутренний замок, но любой непротиворечиво используемый замок будет так же эффективен. Разные переменные состояния могут быть защищены разными замками. (Пример класса, использующего многочисленные замковые объекты, показан в разделе `ServerStatus` на с. 273.)

Базовые коллекционные классы, такие как `ArrayList` и `HashMap`, непотокобезопасны, но их можно использовать в многопоточных средах. Оберточные фабричные методы (`Collections.synchronizedList` и др.) с помощью паттерна декоратора (Gamma, 1995) обертывают коллекции синхронизированными объектами, заставляя методы интерфейса пересылать запросы базовым коллекционным объектам. Оберточный объект потокобезопасен, если содержит только достижимую ссылку на базовую коллекцию. Документация Javadoc предупреждает, что доступ к базовой коллекции должен выполняться через обертку.

Разумеется, возможность нарушить ограничение останется и будет связана с ненадлежащей публикацией объекта или его итераторов и экземпляров внутренних классов, которые могут косвенным путем публиковать окаймляющие объекты.

> Ограничение одним экземпляром упрощает создание потокобезопасных классов, позволяя анализировать их потокобезопасность без проверки всей программы.

4.2.1. Мониторный шаблон Java

По принципу ограничения одним экземпляром работает и *мониторный шаблон Java* (Java monitor pattern)[1]: объект, подчиняющийся шаблону, инкапсулирует мутируемое состояние под защитой внутреннего замка.

В листинге 4.1 вы видели пример работы шаблона: класс `Counter` инкапсулировал одну переменную состояния, `value`, и весь доступ к переменной состояния через методы объекта `Counter`, поддерживая синхронизацию.

Мониторный шаблон Java благодаря своей простоте используется многими библиотечными классами, такими как `Vector` и `Hashtable`. В главе 11 мы показали, как с помощью более сложных замковых стратегий улучшить масштабируемость.

---

[1] Мониторный шаблон Java создан на основе работ Хоара над *мониторами* (Hoare, 1974), но с существенными отличиями. Байт-кодовые инструкции для входа и выхода из синхронизированного блока даже называются `monitorenter` и `monitorexit`, а встроенные в Java (внутренние) замки иногда называются *мониторными замками*, или *мониторами*.

Вместо мониторного шаблона Java, который является просто соглашением, для защиты состояния объекта можно использовать другой замковый объект, например приватный замок, как показано в листинге 4.3.

Листинг 4.3. Защита состояния с помощью приватного замка

```
public class PrivateLock {
    private final Object myLock = new Object();
    @GuardedBy("myLock") Widget widget;

    void someMethod() {
        synchronized(myLock) {
            // Обратиться и изменить состояние виджета
        }
    }
}
```

Преимущество использования приватного замка — в его инкапсуляции, которая не позволяет клиентскому коду приобрести его и участвовать в политике его синхронизации, избавляя от необходимости проверки всей программы.

4.2.2. Пример: трекинг такси

Давайте построим пример мониторного шаблона Java, но сделаем его менее тривиальным, чем класс `Counter` в листинге 4.1. Наш трекер такси будет программой для диспетчера таксопарка. Сначала мы построим его с помощью мониторного шаблона, а затем ослабим требования к инкапсуляции, сохраняя потокобезопасность.

Каждое такси будет идентифицироваться объектом `String` и иметь местоположение в координатах (x, y). Классы `VehicleTracker` инкапсулируют идентичность и местоположения известных такси, чтобы сделать из них модель данных в GUI-приложении. С помощью схемы «модель-представление-контроллер» GUI-приложение сможет совместно использоваться потоками. Просмотровый поток будет доставлять имена и местоположения такси и выводить их на дисплей:

```
Map<String, Point> locations = vehicles.getLocations();
for (String key : locations.keySet())
    renderVehicle(key, locations.get(key));
```

Схожим образом обновляющие потоки будут изменять местоположение такси с помощью данных, полученных от GPS-устройств, либо введенных вручную диспетчером через GUI-интерфейс:

```
void vehicleMoved(VehicleMovedEvent evt) {
    Point loc = evt.getNewLocation();
    vehicles.setLocation(evt.getVehicleId(), loc.x, loc.y);
}
```

Потоки будут обращаться к модели данных конкурентно, и она должна быть потокобезопасной. В листинге 4.4 показана реализация отслеживателя такси с использованием мониторного шаблона Java и класса `MutablePoint` из листинга 4.5 для представления местоположений.

В отличие от класса `MutablePoint`, класс трекера потокобезопасен. Ни ассоциативный массив, ни какие-либо мутируемые точки, которые он содержит, не публикуются. Чтобы вернуть значение местоположения такси вызывающему элементу кода, мы копируем его с помощью конструктора копирования в классе `MutablePoint` или метода `deepCopy`, который создает новый ассоциативный массив `Map`, значения которого являются копиями ключей и значений из старого массива[1].

Копирование большого количества мутируемых данных отрицательно повлияет на производительность[2]. Еще одним следствием копирования данных при каждом вызове метода `getLocation` является неизменность содержимого возвращаемой коллекции. Она удовлетворит требованию непротиворечивости среди множества местоположений, но, если вызывающим элементам кода требуется актуальная информация по каждому такси, придется чаще обновлять снимки.

---

[1] Обратите внимание, что `deepCopy` не может обернуть ассоциативный массив `Map` методом `unmodifiableMap`, потому что обертка защищает от модификации только *коллекцию*, но не мешает вызывающим элементам кода модифицировать хранящиеся в нем мутируемые объекты. По этой же причине заполнение хеш-массива `HashMap` в `deepCopy` с помощью конструктора копирования также не будет работать, поскольку будут скопированы только *ссылки* на точки, а не сами точечные объекты.

[2] Так как метод `deepCopy` вызывается из синхронизированного метода, внутренний замок трекера удерживается в течение всего времени операции копирования. Обработка данных большого количества такси может привести к потере отзывчивости пользовательского интерфейса.

4.3. Делегирование потокобезопасности

Все объекты, кроме самых тривиальных, являются композитными (составными). Мониторный шаблон Java полезен при создании классов с нуля или компоновки классов из непотокобезопасных объектов. Но если компоненты нашего класса уже являются потокобезопасными, нужно ли добавлять дополнительный уровень потокобезопасности? Смотрите по обстоятельствам. В некоторых случаях композит из потокобезопасных компонентов является потокобезопасным (листинги 4.7 и 4.9), а в других — просто хорошей отправной точкой (листинг 4.10).

Листинг 4.4. Мониторная реализация трекера такси

```
@ThreadSafe
public class MonitorVehicleTracker {
    @GuardedBy("this")
    private final Map<String, MutablePoint> locations;

    public MonitorVehicleTracker(
            Map<String, MutablePoint> locations) {
        this.locations = deepCopy(locations);
    }

    public synchronized Map<String, MutablePoint> getLocations() {
        return deepCopy(locations);
    }

    public synchronized MutablePoint getLocation(String id) {
        MutablePoint loc = locations.get(id);
        return loc == null ? null : new MutablePoint(loc);
    }

    public synchronized void setLocation(String id, int x, int y) {
        MutablePoint loc = locations.get(id);
        if (loc == null)
            throw new IllegalArgumentException("No such ID: " + id);
        loc.x = x;
        loc.y = y;
    }

    private static Map<String, MutablePoint> deepCopy(
            Map<String, MutablePoint> m) {
        Map<String, MutablePoint> result =
                new HashMap<String, MutablePoint>();
```

продолжение ⇩

Листинг 4.4 *(продолжение)*

```
        for (String id : m.keySet())
            result.put(id, new MutablePoint(m.get(id)));
        return Collections.unmodifiableMap(result);
    }
}

public class MutablePoint { /* Listing 4.5 */ }
```

Листинг 4.5. Изменяемый класс точки, подобный java.awt.Point

```
@NotThreadSafe
public class MutablePoint {
    public int x, y;

    public MutablePoint() { x = 0; y = 0; }
    public MutablePoint(MutablePoint p) {
        this.x = p.x;
        this.y = p.y;
    }
}
```

В классе `CountingFactorizer` на с. 57 мы добавили `AtomicLong` в объект, который не всегда поддерживал внутреннее состояние, и результирующий композитный объект стал потокобезопасным. Поскольку состояние класса `CountingFactorizer` *являлось* состоянием потокобезопасного `AtomicLong`, а сам класс не накладывал дополнительных ограничений валидности на состояние счетчика, было легко убедиться, что `CountingFactorizer` потокобезопасен. Можно сказать, он *делегировал* свои обязанности по потокобезопасности объекту `AtomicLong`[1].

[1] Важно, что счетчик `count` был финальным. Если бы `CountingFactorizer` мог модифицировать счетчик `count`, чтобы ссылаться на другой `AtomicLong`, нам пришлось бы обеспечивать видимость этого обновления всем потокам, которые могли бы обратиться к `count`, и отсутствие состояний гонки относительно значения ссылки на `count`.

4.3.1. Пример: трекер такси с использованием делегирования

Давайте сконструируем версию трекера такси, которая делегирует свои обязанности по потокобезопасности неизменяемому классу. Местоположения хранятся в ассоциативном массиве `Map`, поэтому мы начнем с потокобезопасной реализации хеш-массива `ConcurrentHashMap`. Как показано в листинге 4.6, хранение местоположений происходит с помощью немутируемого класса `Point` вместо `MutablePoint`.

Листинг 4.6. Неизменяемый класс точки Point, используемый трекером DelegatingVehicleTracker

```
@Immutable
public class Point {
    public final int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

Класс `Point` является потокобезопасным, поэтому нам больше не нужно копировать местоположения при их возврате.

В листинге 4.7 класс `DelegatingVehicleTracker` не использует явной синхронизации. Доступ к состоянию управляется хеш-массивом `ConcurrentHashMap`, где все ключи и значения являются немутируемыми.

Листинг 4.7. Делегирование потокобезопасности в хеш-массив ConcurrentHashMap

```
@ThreadSafe
public class DelegatingVehicleTracker {
    private final ConcurrentMap<String, Point> locations;
    private final Map<String, Point> unmodifiableMap;

    public DelegatingVehicleTracker(Map<String, Point> points) {
        locations = new ConcurrentHashMap<String, Point>(points);
        unmodifiableMap = Collections.unmodifiableMap(locations);
    }
```

продолжение ⇩

Листинг 4.7 *(продолжение)*

```
    public Map<String, Point> getLocations() {
        return unmodifiableMap;
    }

    public Point getLocation(String id) {
        return locations.get(id);
    }

    public void setLocation(String id, int x, int y) {
        if (locations.replace(id, new Point(x, y)) == null)
            throw new IllegalArgumentException(
                "недопустимое имя такси: " + id);
    }
}
```

Использование исходного класса `MutablePoint` вместо класса `Point` нарушит инкапсуляцию, позволив методу `getLocations` публиковать ссылку на мутируемое непотокобезопасное состояние. Обратите внимание, что мы немного изменили поведение класса трекера такси: мониторная версия возвращала снимок местоположений, а делегирующая версия возвращает их неизменяемое, но «живое» представление. Это означает, что если поток *A* вызывает метод `getLocations`, а поток *B* позже изменяет местоположение некоторых точек, эти изменения отражаются в хеш-массиве, возвращенном потоку *A*. Как мы отмечали ранее, это может быть преимуществом (более актуальные данные) либо недостатком (потенциально противоречивое представление таксопарка), в зависимости от ваших требований.

Если требуется неизменное представление парка, то метод `getLocations` может вернуть неглубокую копию хеш-массива местоположений. Поскольку содержимое хеш-массива является немутируемым, необходимо копировать только структуру хеш-массива, а не его содержимое. Как показано в листинге 4.8, возвращается простой хеш-массив `HashMap`.

Листинг 4.8. Возврат статической копии множества местоположений вместо «живой» копии

```
public Map<String, Point> getLocations() {
    return Collections.unmodifiableMap(
            new HashMap<String, Point>(locations));
}
```

4.3.2. Независимые переменные состояния

Делегировать потокобезопасность более чем одной базовой переменной состояния возможно, если эти переменные *независимы*, то есть композитный класс не накладывает инварианты с их участием.

Класс `VisualComponent` в листинге 4.9 представляет собой графический компонент, который поддерживает список зарегистрированных слушателей в виде двух независимых множеств (слушателей мыши и слушателей клавиш), чтобы при наступлении события активировать соответствующих слушателей. `VisualComponent` может делегировать свои обязательства по потокобезопасности двум потокобезопасным спискам.

Листинг 4.9. Делегирование потокобезопасности нескольким базовым переменным состояния

```
public class VisualComponent {
    private final List<KeyListener> keyListeners
        = new CopyOnWriteArrayList<KeyListener>();
    private final List<MouseListener> mouseListeners
        = new CopyOnWriteArrayList<MouseListener>();

    public void addKeyListener(KeyListener listener) {
        keyListeners.add(listener);
    }

    public void addMouseListener(MouseListener listener) {
        mouseListeners.add(listener);
    }

    public void removeKeyListener(KeyListener listener) {
        keyListeners.remove(listener);
    }

    public void removeMouseListener(MouseListener listener) {
        mouseListeners.remove(listener);
    }
}
```

Класс `VisualComponent` использует список `CopyOnWriteArrayList` для хранения каждого списка слушателей (причины для этого описаны в разделе 5.2.3). Каждый список является потокобезопасным, и поскольку нет ограничений между их состояниями, `VisualComponent` может делегировать свои обязанности по потокобезопасности базовым объектам `mouseListeners` и `keyListeners`.

4.3.3. Случаи безуспешного делегирования

Большинство композитных классов не так просты, как `VisualComponent`: у них есть инварианты, которые связывают их компонентные переменные состояния. Класс `NumberRange` в листинге 4.10 для управления своим состоянием использует два объекта `AtomicInteger`, но накладывает дополнительное ограничение — первое число должно быть меньше или равно второму.

Листинг 4.10. Класс диапазона чисел, недостаточно защищающий свои инварианты. *Так делать не следует*

```
public class NumberRange {
    // ИНВАРИАНТ: lower <= upper
    private final AtomicInteger lower = new AtomicInteger(0);
    private final AtomicInteger upper = new AtomicInteger(0);

    public void setLower(int i) {
        // Предупреждение – небезопасная операция "проверить и затем
        // действовать"
        if (i > upper.get())
            throw new IllegalArgumentException(
                "не могу установить lower равным " + i + " > upper");
        lower.set(i);
    }

    public void setUpper(int i) {
        // Предупреждение – небезопасная операция "проверить и затем
        // действовать"
        if (i < lower.get())
            throw new IllegalArgumentException(
                "не могу установить upper равным " + i + " < lower");
        upper.set(i);
    }

    public boolean isInRange(int i) {
        return (i >= lower.get() && i <= upper.get());
    }
}
```

Класс `NumberRange` не является потокобезопасным: он не соблюдает инвариант, который ограничивает `lower` и `upper`. Методы `setLower` и `setUpper` *пытаются* соблюсти этот инвариант, но безуспешно, так как являются последовательностями операций «проверить и затем действовать», но их блокировка не обеспечивает им атомарности. Если в диапазоне (0, 10) один поток вызовет `setLower(5)`, а другой в то же время — `setUpper(4)`, то при неудачной временно́й координации оба потока пройдут проверку в методах доступа `set`, и выполнятся два изменения, создав диапазон (5, 4) — недопустимое состояние. В отличие от базовых `AtomicInteger`, составной класс непотокобезопасен, а класс `NumberRange` не может делегировать потокобезопасность переменным состояния `lower` и `upper`, так как они не являются независимыми.

Потокобезопасность класса `NumberRange` может обеспечить блокировка, которая сохранит его инварианты и защитит `lower` и `upper` замком, чтобы препятствовать их публикации.

Составные действия не позволят только одному делегированию обеспечить потокобезопасность. Классу потребуется собственная блокировка, которая обеспечит атомарность составных действий, если каждое из них не может быть целиком делегировано базовым переменным состояния.

> Если класс составлен из *независимых* потокобезопасных переменных состояния и не имеет недопустимых переходов из состояния в состояние, то он может делегировать потокобезопасность базовым переменным состояния.

Класс `NumberRange` не мог стать потокобезопасным, даже если его компоненты состояния были потокобезопасными, так же как в разделе 3.1.4 переменная не могла стать волатильной, пока не начала участвовать в инвариантах, связанных с другими переменными состояния.

4.3.4. Публикация базовых переменных состояния

Каковы условия публикации базовых переменных, которым делегированы обязанности по потокобезопасности? Ответ зависит от того, какие инварианты класс накладывает на эти переменные. Базовое поле `value` в классе `Counter` может принимать любое целочисленное значение, но `Counter`

ограничивает его только положительными значениями, а операция приращения значения ограничивает множество допустимых следующих состояний для любого текущего состояния. Если сделать поле `value` публичным, клиенты смогут присваивать ему недопустимые значения, и класс будет не реализован. Но если переменная представляет текущую температуру или идентифицирует последнего вошедшего в систему пользователя, то изменение извне не нарушит инварианты. (Публикация мутируемых переменных ограничивает будущую разработку подклассирования, но она *не всегда* лишает класс потокобезопасности.)

> Если переменная состояния является потокобезопасной, не участвует в инвариантах, ограничивающих ее значение, и не имеет запрещенных переходов из состояния в состояние для любой из ее операций, то она может быть безопасно опубликована.

Поскольку класс `VisualComponent` не накладывает ограничений на допустимые состояния списков слушателей, поля `mouseListeners` или `keyListeners` можно сделать публичными либо опубликовать иным образом без ущерба для потокобезопасности.

4.3.5. Пример: трекер такси, публикующий свое состояние

Рассмотрим версию трекера такси с публикацией базового мутируемого состояния путем размещения на мутируемых, но потокобезопасных точках.

Листинг 4.11. Класс потокобезопасной мутируемой точки

```
@ThreadSafe
public class SafePoint {
    @GuardedBy("this") private int x, y;

    private SafePoint(int[] a) { this(a[0], a[1]); }

    public SafePoint(SafePoint p) { this(p.get()); }

    public SafePoint(int x, int y) {
        this.x = x;
        this.y = y;
    }
```

```
    public synchronized int[] get() {
        return new int[] { x, y };
    }

    public synchronized void set(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

Класс `SafePoint` в листинге 4.11 предоставляет метод доступа `get`, который извлекает значения *x* и *y* одновременно, возвращая двухэлементный массив[1]. Наличие двух отдельных методов `get` для *x* и *y* заставило бы вызывающий элемент кода видеть противоречивое значение. Используя `SafePoint`, мы можем создать отслеживатель такси, который публикует базовое мутируемое состояние без ущерба для потокобезопасности, как показано в классе `PublishingVehicleTracker` в листинге 4.12.

Листинг 4.12. Трекер такси, безопасно публикующий базовое состояние

```
@ThreadSafe
public class PublishingVehicleTracker {
    private final Map<String, SafePoint> locations;
    private final Map<String, SafePoint> unmodifiableMap;

    public PublishingVehicleTracker(
                            Map<String, SafePoint> locations) {
        this.locations
            = new ConcurrentHashMap<String, SafePoint>(locations);
        this.unmodifiableMap
            = Collections.unmodifiableMap(this.locations);
    }

    public Map<String, SafePoint> getLocations() {
        return unmodifiableMap;
    }

    public SafePoint getLocation(String id) {
        return locations.get(id);
    }
```

продолжение ⇩

[1] Приватный конструктор существует для того, чтобы избежать состояния гонки, возникающего при реализации конструктора копирования, такого как `this(p.x, p.y)`; это пример *захвата частного конструктора* (private constructor capture) (Bloch и Gafte, 2005).

Листинг 4.12 *(продолжение)*

```
    public void setLocation(String id, int x, int y) {
        if (!locations.containsKey(id))
            throw new IllegalArgumentException(
                "недопустимое имя транспортного средства: " + id);
        locations.get(id).set(x, y);
    }
}
```

Класс `PublishingVehicleTracker` делегирует свои обязанности по потокобезопасности базовому хеш-массиву `ConcurrentHashMap`, содержимым которого на этот раз являются потокобезопасные мутируемые точки. Метод `getLocation` возвращает неизменяемую копию базового ассоциативного массива `Map`. Вызывающие элементы кода не могут добавлять или удалять такси, но могут изменять местоположение одного из них путем мутирования значений `SafePoint` в возвращенном ассоциативном массиве `Map`. «Живой» характер ассоциативного массива `Map` может быть преимуществом либо недостатком, в зависимости от требований. Класс `PublishingVehicleTracker` является потокобезопасным, благодаря тому что он не накладывает дополнительные ограничения на допустимые значения местоположений такси. Чтобы запретить изменения местоположения такси или снабдить их дополнительными свойствами, нужно отказаться от класса `PublishingVehicleTracker`.

4.4. Добавление функциональности в существующие потокобезопасные классы

Библиотека классов Java содержит много полезных классов, выступающих как «строительные блоки». Их использование предпочтительнее разработки новых классов, так как не подразумевает дополнительных усилий, рисков и стоимости сопровождения. Потокобезопасный класс, который поддерживает нужные нам операции, не так полезен, как класс, поддерживающий *потенциально* нужные нам операции и позволяющий добавлять новую операцию, не подрывая потокобезопасность.

Например, чтобы создать потокобезопасный список с атомарной операцией «добавить, если отсутствует», нам потребуется синхронизированный список `List`, который предоставляет методы `contains` и `add`.

«Добавить, если отсутствует» — значит перед добавлением элемента в коллекцию проверить его наличие. (Где-то рядом операция «проверить и затем действовать».) Потокобезопасность класса требует *атомарности* операций, чтобы в список, не содержащий *X*, «класть» сколько угодно *X* из разных потоков, но в результирующей коллекции увидеть только одну копию *X*.

Самый безопасный способ добавить атомарную операцию — это изменить класс таким образом, чтобы он начал ее поддерживать. Мы не всегда имеем доступ к исходному коду, однако, если мы его получили, нам нужно понимать принятую в коде политику синхронизации, которая контролируется в одном исходном файле.

Можно расширить класс (предназначенный для расширения). Класс `BetterVector` в листинге 4.13 расширяет класс `Vector`, добавляя метод `putIfAbsent`.

Но после расширения реализация политики синхронизации распределится по нескольким отдельно сопровождаемым исходным файлам. То есть, если базовый класс изменит замок для защиты своих переменных состояния, то подкласс станет использовать неправильный замок для управления конкурентным доступом к состоянию базового класса. (Политика синхронизации класса `Vector` зафиксирована в его спецификации, поэтому `BetterVector` от этой проблемы не пострадал.)

Листинг 4.13. Расширение класса Vector для использования метода «добавить, если отсутствует»

```
@ThreadSafe
public class BetterVector<E> extends Vector<E> {
    public synchronized boolean putIfAbsent(E x) {
        boolean absent = !contains(x);
        if (absent)
            add(x);
        return absent;
    }
}
```

4.4.1. Блокировка на стороне клиента

Для списка `ArrayList` в обертке `Collections.synchronizedList` не подходят добавление метода в исходный класс или расширение класса, по-

тому что клиентский код даже не знает класса спискового объекта `List`, возвращаемого из синхронизированных оберточных фабрик. Ему нужно расширение функциональности класса без расширения самого класса путем помещения кода во вспомогательный класс.

В листинге 4.14 показана безуспешная попытка создания вспомогательного класса с атомарной операцией «добавить, если отсутствует» для работы со списком `List`.

Листинг 4.14. Непотокобезопасная попытка реализовать метод «добавить, если отсутствует». *Так делать не следует*

```
@NotThreadSafe
public class ListHelper<E> {
    public List<E> list =
        Collections.synchronizedList(new ArrayList<E>());
    ...
    public synchronized boolean putIfAbsent(E x) {
        boolean absent = !list.contains(x);
        if (absent)
            list.add(x);
        return absent;
    }
}
```

Почему безуспешная? Метод `putIfAbsent` синхронизирован? Да, но на *неправильном замке*. Класс `ListHelper` обеспечивает только *иллюзию синхронизации*: на самом деле операции используют разные замки, и метод `putIfAbsent` *не* воспринимается другими операциями как атомарный.

Исправить этот пример поможет *блокировка на стороне клиента* (client-side locking) или *внешняя блокировка*. Мы защитим клиентский код, который использует некий объект *X*, собственным замком объекта *X*. Необходимо будет узнать, что это за замок.

Документация для `Vector` и синхронизированных оберточных классов утверждает, хотя и косвенно, что они поддерживают блокировку на стороне клиента, используя внутренний замок для `Vector` или оберточной коллекции (не путайте с обернутой коллекцией). Листинг 4.15 показывает

в потокобезопасном списке `List`, как операция `putIfAbsent` правильно использует блокировку на стороне клиента.

Листинг 4.15. Реализация метода «добавить, если отсутствует» с блокировкой на стороне клиента

```
@ThreadSafe
public class ListHelper<E> {
    public List<E> list =
        Collections.synchronizedList(new ArrayList<E>());
    ...
    public boolean putIfAbsent(E x) {
        synchronized (list) {
            boolean absent = !list.contains(x);
            if (absent)
                list.add(x);
            return absent;
        }
    }
}
```

Блокировка на стороне клиента влечет за собой внедрение замкового кода для класса *C* в классы, которые никак не связаны с *C*, поэтому проявляйте осторожность при использовании такой блокировки на некоторых классах.

В то время как расширение класса нарушает инкапсуляцию реализации [EJ пункт 14], блокировка на стороне клиента нарушает инкапсуляцию политики синхронизации.

4.4.2. Компоновка

Добавить атомарную операцию в существующий класс поможет *компоновка* (composition). Класс `ImprovedList` в листинге 4.16 реализует списковые операции, передавая их в базовый экземпляр `List` и добавляя атомарный метод `putIfAbsent`, через который клиент должен обращаться к базовому списку.

Листинг 4.16. Реализация метода «добавить, если отсутствует» с использованием компоновки

```
@ThreadSafe
public class ImprovedList<T> implements List<T> {
    private final List<T> list;
```

продолжение ⇩

Листинг 4.16 *(продолжение)*

```
    public ImprovedList(List<T> list) { this.list = list; }

    public synchronized boolean putIfAbsent(T x) {
        boolean contains = list.contains(x);
        if (contains)
            list.add(x);
        return !contains;
    }

    public synchronized void clear() { list.clear(); }
    // ... делегировать остальные списковые методы схожим образом
}
```

Класс `ImprovedList` добавляет новый уровень блокировки с помощью своего внутреннего замка, не обращая внимания на наличие или отсутствие потокобезопасности у `List`. Дополнительная синхронизация может наложить штраф на производительность[1], но такое использование мониторного шаблона Java для инкапсуляции существующего списка гарантированно обеспечит потокобезопасность, при условии что класс содержит единственную ожидающую обработки ссылку на базовый список.

4.5. Документирование политик синхронизации

Документация представляет собой один из самых мощных (и, к сожалению, наиболее недооцененных) инструментов управления потокобезопасностью. Она призвана сообщать пользователям о потокобезопасности того или иного класса и помогать сопроводителям понять стратегию разработчика.

> Документируйте гарантии потокобезопасности класса для клиентов и политику синхронизации — для сопроводителей.

Каждое применение механизмов `synchronized`, `volatile` или любого потокобезопасного класса отражает *политику синхронизации*. Как проектный элемент программы она должна быть задокументирована, причем лучше всего — на этапе разработки.

---

[1] Штраф будет небольшим, так как синхронизация в базовом списке не будет оспариваться и, следовательно, будет быстрой (см. главу 11).

Определитесь, какие переменные сделать волатильными, какие — защитить замками (и какими замками), какие — сделать немутируемыми или ограничить одним потоком, какие операции сделать атомарными и т. д. Детали реализации документируются для сопроводителей, а элементы, влияющие на публично наблюдаемое поведение класса, документируются как часть его спецификации.

Обозначьте, является ли класс потокобезопасным, делает ли он обратные вызовы с удерживаемым замком и существуют ли замки, которые влияют на его поведение.

Текущее состояние дел в документации по потокобезопасности, даже в платформенных библиотеках классов, не обнадеживает. Сколько раз вы обращались к Javadoc с вопросом, является ли класс потокобезопасным?[1] По большинству классов не найдется никаких подсказок.

Мы стараемся следовать спецификациям, но в работе мы сталкиваемся с необходимостью допущений. Должны ли мы исходить из допущения, что объект является потокобезопасным, потому что нам так кажется? Или нужно приобрести замок на объект? (Это рискованное техническое решение работает, только если мы контролируем *весь* код, который обращается к этому объекту.)

Наша интуиция может ошибаться. Например, `java.text.SimpleDateFormat` не является потокобезопасным, но Javadoc упоминул об этом только в JDK 1.4, поразив многих разработчиков. Сколько было создано совместных экземпляров непотокобезопасного объекта, использование которого в многопоточной среде может привести к ошибочным результатам при большой нагрузке?

Невозможно разработать сервлетное приложение, не принимая допущений о потокобезопасности поставляемых контейнером объектов типа `HttpSession`. Не заставляйте своих клиентов или коллег рисковать.

4.5.1. Толкование расплывчатой документации

Спецификации технологии Java мало сообщают о гарантиях потокобезопасности и требованиях к интерфейсам, таким как `ServletContext`,

[1] Если вы никогда об этом не задумывались, то мы восхищаемся вашим оптимизмом.

`HttpSession` и `DataSource`[1]. Поскольку эти интерфейсы реализуются поставщиком контейнера или базы данных, часто невозможно просмотреть их код. И что же нам делать?

Догадываться. Интерпретировать спецификацию с точки зрения того, кто будет ее *реализовывать* (например, поставщика контейнера или базы данных), а не клиента. Допустим, что контейнер знает о существовании других потоков, потому что он делает доступными определенные объекты, которые будут обращаться к многочисленным сервлетам, таким как `HttpSession` или `ServletContext`.

Допустим также, что объекты создаются потокобезопасными. Во-первых, невозможно вообразить их однопоточное использование. Во-вторых, нигде не говорится о замке, на котором синхронизирован клиентский код, и примеры в спецификации и официальных руководствах показывают, как обращаться к `ServletContext` или `HttpSession`, не используя блокировку на стороне клиента.

Однако объекты с `setAttribute`, помещенные в сервлетный контейнер, принадлежат веб-приложению. Спецификация не предлагает механизма для координации конкурентного доступа к совместным атрибутам, поэтому допустим, что они создаются потокобезопасными или фактически немутируемыми и, возможно, защищенными замком. Но если контейнеру потребуется сериализовать объекты в `HttpSession` в целях репликации или пассивации, а сервлетный контейнер не будет знать замковый протокол, то обеспечение их потокобезопасности станет вашей заботой.

`DataSource` представляет собой пул многократно используемых соединений с базой данных. Трудно вообразить его работу без вызова метода `getConnection` из разных потоков. Поскольку спецификация не требует для этого блокировки на стороне клиента, допустим, что метод потокобезопасен.

Но объекты соединения JDBC, раздаваемые интерфейсом `DataSource`, не обязательно предназначены для совместного использования до возвращения в пул. Поэтому, если действие, получающее соединение JDBC, охватывает многочисленные потоки, его следует защитить с помощью синхронизации. (В большинстве приложений действия, использующие соединение JDBC, ограничиваются одним потоком.)

---

[1] Особенно грустно, что эти упущения сохраняются после ревизий спецификаций.

5 Строительные блоки

В предыдущей главе мы описали несколько технических решений для конструирования потокобезопасных классов.

Эта глава посвящена использованию конкурентных строительных блоков, таких как коллекции и *синхронизаторы*.

5.1. Синхронизированные коллекции

К *синхронизированным классам коллекций* (synchronized collection classes) относятся `Vector` и `Hashtable`, часть исходного JDK, а также добавленные в JDK 1.2 синхронизированные оберточные классы, создаваемые фабричными методами `Collections.synchronizedXxx`. Эти классы обеспечивают потокобезопасность, инкапсулируя свое состояние и синхронизуя каждый публичный метод таким образом, чтобы только один поток за один раз мог обратиться к состоянию коллекции.

5.1.1. Проблемы синхронизированных коллекций

Синхронизированные коллекции являются потокобезопасными, но, когда потоки имеют возможность конкурентно изменять коллекцию, требуется блокировка на стороне клиента для защиты составных действий. Такие действия включают итеративный обход (доставку элементов из

коллекции), навигацию (отыскание следующего элемента) и условные операции (связывание ключа *K* с неким значением).

В листинге 5.1 показаны два метода, работающие над объектом `Vector`: `getLast` и `deleteLast`, которые являются последовательностями действий «проверить и затем действовать». Каждый вызывает метод `size` для определения размера массива и использует результирующее значение для извлечения или удаления последнего элемента.

Листинг 5.1. Составные действия над объектом Vector, приводящие к запутанным результатам

```
public static Object getLast(Vector list) {
    int lastIndex = list.size() - 1;
    return list.get(lastIndex);
}

public static void deleteLast(Vector list) {
    int lastIndex = list.size() - 1;
    list.remove(lastIndex);
}
```

Сами методы не могут повредить `Vector` — опасен элемент кода, вызывающий их. Если поток *A* вызывает метод `getLast`, а поток *B* — метод `deleteLast`, то эти операции перемежаются, как показано на рисунке 5.1, и метод `getLast` выдает исключение `ArrayIndexOutOfBoundsException`. Между вызовом метода `size` и последующим вызовом метода `getlast` объект `Vector` сжимается и индекс, вычисленный на первом шаге, перестает быть допустимым.

Поскольку синхронизированные коллекции поддерживают блокировку на стороне клиента[1], существует возможность создания новых операций, атомарных по отношению к другим операциям над коллекцией, если известен замок. Синхронизированные коллекционные классы защищают каждый метод замком на самом синхронизированном коллекционном объекте. Как показано в листинге 5.2, приобретая коллекционный замок,

[1] Это задокументировано в Javadoc для Java 5.0 как пример правильной итерации.

мы можем сделать методы `getLast` и `deleteLast` атомарными, обеспечив неизменность размера `Vector` на время между вызовами методов `size` и `get`.

Риск изменения размера списка между вызовами методов также присутствует во время итеративного обхода элементов `Vector`, как показано в листинге 5.3.

Если другой поток удалит элемент во время итеративного обхода `Vector`, появится исключение `ArrayIndexOutOfBoundsException`.

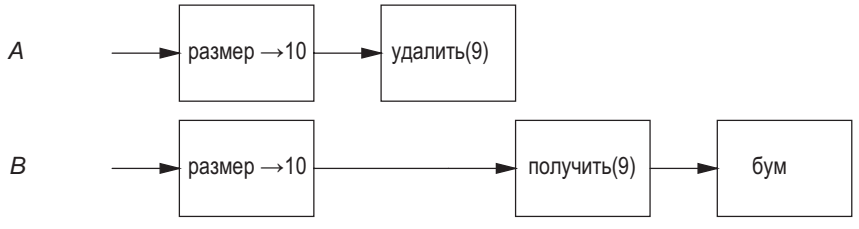

Рис. 5.1. Чередование методов getLast и deleteLast, которое выдает исключение ArrayIndexOutOfBoundsException

Листинг 5.2. Составные действия над объектом Vector с использованием блокировки на стороне клиента

```
public static Object getLast(Vector list) {
    synchronized (list) {
        int lastIndex = list.size() - 1;
        return list.get(lastIndex);
    }
}

public static void deleteLast(Vector list) {
    synchronized (list) {
        int lastIndex = list.size() - 1;
        list.remove(lastIndex);
    }
}
```

Листинг 5.3. Итеративный обход, который может выдавать исключение ArrayIndexOutOfBoundsException

```
for (int i = 0; i < vector.size(); i++)
    doSomething(vector.get(i));
```

Объект `Vector` остается потокобезопасным, а исключение соответствует его спецификации. Но то, что доставка последнего элемента из коллекции или итеративный обход выдают исключение, конечно, нежелательно.

Блокировка на стороне клиента влияет на масштабируемость. В течение всего времени итеративного обхода владея замком на `Vector`, мы не позволяем другим потокам изменять его (см. листинг 5.4). К сожалению, от этого страдает конкурентность.

Листинг 5.4. Итеративный обход с блокировкой на стороне клиента

```
synchronized (vector) {
    for (int i = 0; i < vector.size(); i++)
        doSomething(vector.get(i));
}
```

5.1.2. Итераторы и исключение ConcurrentModificationException

Как и класс `Vector`, более современные классы коллекций содержат проблемы составных действий. Стандартный способ итеративного обхода объекта `Collection` заключается в использовании объекта `Iterator` (явно либо посредством синтаксиса цикла `for-each`) в сочетании с замком. Итераторы синхронизированных коллекций не проектировались для конкурентного выполнения изменений, поэтому, обнаружив во время итеративного обхода изменения в коллекции, они выдают исключение `ConcurrentModificationException`, то есть *быстро отказывают* (fail-fast).

Такие итераторы не предназначены для защиты объекта — они действуют только в качестве индикаторов проблем конкурентности. Их методы `hasNext` или `next` выдают исключение, если во время итеративного обхода обнаруживают изменения счетчика изменений. Они ведут проверку без синхронизации и могут увидеть устаревшее значение счетчика, благодаря чему сокращено влияние кода обнаружения изменений на производительность[1].

В листинге 5.5 показан итеративный обход коллекции с использованием синтаксиса цикла `for-each`. `javac` генерирует код, который использует

[1] Исключение `ConcurrentModificationException` также может возникать в однопоточном коде, когда объекты удаляются из коллекции напрямую, а не через `Iterator.remove`.

итератор, многократно вызывая `hasNext` и `next`. Чтобы предотвратить исключение `ConcurrentModificationException`, надо владеть коллекционным замком в течение всего времени итеративного обхода.

Листинг 5.5. Итеративный обход списка с помощью итератора

```
List<Widget> widgetList
    = Collections.synchronizedList(new ArrayList<Widget>());
...
// Может выдать ConcurrentModificationException
for (Widget w : widgetList)
    doSomething(w);
```

Однако блокировка коллекции во время итеративного обхода не всегда уместна. При больших коллекциях и задачах ожидание завершения итерации становится слишком долгим. Если коллекция заперта, как в листинге 5.4, то метод `doSomething` вызывается с удержанием замка, что является фактором риска возникновения взаимной блокировки (см. главу 10). Любое запирание коллекций на значительные периоды времени снижает масштабируемость приложения: чем дольше замок занят, тем больше вероятность того, что он будет оспариваться другими потоками (см. главу 11).

Альтернативой запиранию коллекции во время итеративного обхода является ее клонирование и итеративный обход клона. Клон ограничен одним потоком, но коллекция все равно должна быть защищена замком во время клонирования. Насколько клонирование оправданно, зависит от размера коллекции, объема работы, выполняемой для каждого элемента, относительной частоты итераций, а также требований к отзывчивости и пропускной способности.

5.1.3. Скрытые итераторы

Использовать блокировку везде, где может выполняться итеративный обход совместной коллекции, не так просто. Итераторы бывают скрытыми, как в `HiddenIterator` в листинге 5.6, где явного итеративного обхода нет, но код, выделенный жирным шрифтом, его подразумевает. Конкатенация строк преобразуется компилятором в вызов `StringBuilder.append(Object)`,

который, в свою очередь, активирует метод `toString`, который в стандартных коллекциях выполняет итеративный обход.

Пока коллекция перебирается методом `toString`, метод `addTenThings` может выдавать исключение `ConcurrentModificationException`. Дело в том, что класс `HiddenIterator` не является потокобезопасным, и замок его итератора должен быть приобретен перед использованием метода `set` в вызове `println`, но код отладки и журналирования обычно этим пренебрегает.

Чем больше расстояние между состоянием и синхронизацией, защищающей его, тем больше вероятность того, что кто-то забудет применить надлежащую синхронизацию при доступе к этому состоянию. Если `HiddenIterator` обернет хеш-множество `HashSet` в `synchronizedSet`, инкапсулируя синхронизацию, ошибки не будет.

Инкапсулирование синхронизации упрощает усиление ее политики.

Листинг 5.6. Итеративный обход, скрытый внутри конкатенации строк. *Так делать не следует*

```
public class HiddenIterator {
    @GuardedBy("this")
    private final Set<Integer> set = new HashSet<Integer>();

    public synchronized void add(Integer i) { set.add(i); }
    public synchronized void remove(Integer i) { set.remove(i); }

    public void addTenThings() {
        Random r = new Random();
        for (int i = 0; i < 10; i++)
            add(r.nextInt());
        System.out.println("ОТЛАДКА: добавлено десять элементов в " + set);
    }
}
```

Итеративный обход также косвенно активируется коллекционными методами `hashCode` и `equals`, которые могут вызываться, если коллекция используется в качестве элемента или ключа другой коллекции. Методы

containsall, removeAll и retainAll, а также конструкторы, принимающие коллекции в качестве аргументов, также выполняют итеративный обход коллекции. Все эти косвенные итеративные обходы могут выдавать исключение ConcurrentModificationException.

5.2. Конкурентные коллекции

Синхронизированные коллекции обеспечивают потокобезопасность, сериализуя весь доступ к своему состоянию. Стоимость этого подхода равна слабой конкурентности. Совершенствуя синхронизированные коллекции, Java 5.0 предоставляет несколько *конкурентных* коллекционных классов.

Версия предлагает хеш-массив ConcurrentHashMap вместо синхронизированных хешированных реализаций ассоциативного массива Map, и CopyOnWriteArrayList вместо синхронизированных реализаций списка List для случаев, где обход является доминирующей операцией. Новый интерфейс ассоциативного массива ConcurrentMap поддерживает распространенные составные действия, такие как операция «добавить, если отсутствует», замена и условное удаление.

> Замена синхронизированных коллекций конкурентными коллекциями может предложить значительные улучшения масштабируемости при небольшом риске.

Есть два типа коллекций: Queue и BlockingQueue. Класс Queue предназначен для временного хранения множества элементов во время ожидания ими обработки. Он представлен в нескольких реализациях, в том числе ConcurrentLinkedQueue (традиционной очереди с дисциплиной доступа FIFO) и PriorityQueue (неконкурентной упорядоченной очереди с приоритетом). Операции над очередью не блокируют продвижение. Если очередь является пустой, то операция извлечения возвращает значение null. Поведение очереди можно симулировать с помощью списка List — связный список LinkedList тоже реализует очередь Queue.

Класс BlockingQueue расширяет класс Queue, добавляя блокирующие операции вставки и извлечения. Если очередь является пустой, то операция извлечения будет заблокирована до тех пор, пока в ней не появится один элемент. Соответственно, если ограниченная очередь заполнена, то

операция вставки будет заблокирована до появления свободного места. Блокирующие очереди полезны в паттернах проектирования «производитель-потребитель» и подробнее рассмотрены в разделе 5.3.

Начиная с Java 6 появились `ConcurrentSkipListMap` и `ConcurrentSkipListSet` — конкурентные версии синхронизированного массива `SortedMap` и множества `SortedSet` (`TreeMap` и `TreeSet`, обернутых в `synchronizedMap`).

5.2.1. ConcurrentHashMap

Некоторые операции, такие как `HashMap.get` или `List.contains`, могут быть сопряжены с большим объемом работы: обход хеш-корзины или списка для отыскания определенного объекта иногда требует вызова метода `equals`. Если функция `hashCode` распределяет значения хеша неэффективно, то элементы могут неравномерно разместиться в корзинах или даже превратить хеш-массив в связный список. Обход длинного списка и вызовы метода `equals` могут занять много времени, и за это время другие потоки не должны обращаться к коллекции.

Класс `ConcurrentHashMap` — это хешированный ассоциативный массив `Map`, аналогичный хеш-массиву `HashMap`, но использующий другую замковую стратегию. Вместо синхронизации каждого метода на общем замке и ограничения доступа одним потоком за раз он использует *замковое расщепление на полосы* (lock striping, см. раздел 11.4.3), расширяющее возможности совместного доступа к ассоциативному массиву. Оно обеспечивает конкурентность между читающими потоками, между читателями и писателями и между писателями. Результатом является высокая пропускная способность в рамках конкурентного доступа с небольшим штрафом на производительность для однопоточного доступа.

Также класс `ConcurrentHashMap` предоставляет итераторы, которые не выдают исключение `ConcurrentModificationException`, являются не быстро отказывающими, а *слабо непротиворечивыми* (weakly consistent). Они допускают конкурентное выполнение изменений, перебирают элементы в том порядке, в каком они существовали при конструировании итератора, и могут (не обязательно) отражать изменения в коллекцию.

Однако семантика методов, работающих на всем ассоциативном массиве `Map`, таких как `size` и `isEmpty`, была немного ослаблена: в конкурентных средах эти методы менее полезны, поскольку ищут движущиеся цели.

В некоторых случаях, таких как атомарное добавление нескольких соответствий «ключ-значение» или итеративный обход ассоциативного массива несколько раз, нам нужно видеть одни и те же элементы в одном и том же порядке. Тогда с помощью хеш-таблицы `Hashtable` и ассоциативного массива `synchronizedMap` мы можем пойти на компромисс и приобрести замок эксклюзивного доступа ассоциативного массива `Map`.

В большинстве случаев замена синхронизированных реализаций ассоциативного массива `Map` на хеш-массив `ConcurrentHashMap` приводит к более высокой масштабируемости. Но если вам необходимо только защитить ассоциативный массив замком для эксклюзивного доступа[1], то `ConcurrentHashMap` вам не подойдет.

5.2.2. Дополнительные атомарные операции над ассоциативным массивом

Поскольку хеш-массив `ConcurrentHashMap` не заперт для эксклюзивного доступа, мы не можем использовать блокировку на стороне клиента для создания новых атомарных операций как в разделе 4.4.1. Однако ряд составных операций («добавить, если отсутствует», «удалить, если равно» и «заменить, если равно») сразу задаются интерфейсом `ConcurrentMap` как атомарные (см. листинг 5.7).

5.2.3. CopyOnWriteArrayList

Класс `CopyOnWriteArrayList` является конкурентной версией синхронизированного спискового класса `List`, который предлагает более высокую конкурентность и устраняет необходимость в блокировке или клонировании коллекции во время итеративного обхода.

Пока фактически немутируемый объект коллекции «копировать при записи» публикуется правильно, при обращении к нему синхронизация не требуется. При создании и публикации копии коллекции после каждого ее изменения реализуется мутируемость. Итераторы сохраняют ссылку на резервный массив, который был актуальным в начале итеративного обхода, и так как он никогда не изменяется, синхронизация им нужна ненадолго — чтобы обеспечить видимость содержимого массива. Потоки могут выполнять итеративный обход коллекции, не мешая

[1] Либо вы активно используете побочные эффекты синхронизации массива `Map`.

потокам, желающим изменить ее. Итераторы не выдают исключения `ConcurrentModificationException` и возвращают элементы, не измененные с момента создания итератора.

Листинг 5.7. Интерфейс ConcurrentMap

```
public interface ConcurrentMap<K,V> extends Map<K,V> {
    // Вставить в ассоциативный массив, только если ни одно значение
    // не соответствует ключу K
    V putIfAbsent(K key, V value);

    // Удалить, только если ключу K соответствует значение V
    boolean remove(K key, V value);

    // Заменить значение, только если ключу K соответствует старое
    // значение oldValue
    boolean replace(K key, V oldValue, V newValue);

    // Заменить значение, только если ключу K соответствует некое значение
    V replace(K key, V newValue);
}
```

Очевидно, что копирование резервного массива сопряжено с определенной стоимостью, и его целесообразно использовать только тогда, когда итеративный обход встречается гораздо чаще, чем изменение [CPJ 2.4.4]. Например, доставка уведомления требует итеративного обхода списка зарегистрированных слушателей и вызова каждого из них, а регистрация или дерегистрация слушателей происходят гораздо реже.

5.3. Блокирующие очереди и паттерн «производитель-потребитель»

Блокирующие очереди предоставляют блокирующие методы `put` и `take`, а также хронометрированные эквиваленты `offer` и `poll`. Метод `put` блокирует продвижение до тех пор, пока не освободится место в заполненной очереди, а метод `take` блокирует продвижение до появления элемента в пустой очереди. Неимитированные очереди никогда не заполняются полностью, поэтому метод `put` в них не используется.

Блокирующие очереди поддерживают паттерн проектирования *производитель-потребитель* (producer-consumer), который помещает рабочие

элементы в список предстоящих дел для последующей обработки, вместо того чтобы обрабатывать их сразу по мере их идентификации. Шаблон устраняет зависимости от кода между производящими и потребляющими классами и упрощает управление рабочей нагрузкой путем расстыковки действий, выполняемых с разной скоростью.

Согласно шаблону производители помещают данные в очередь, ничего не зная о потребителях, которые будут извлекать эти данные. Точно так же потребители не должны знать, откуда берется работа. Очередь `BlockingQueue` упрощает реализацию шаблона с любым числом производителей и потребителей. Примером этого шаблона является поточный пул в соединении с рабочей очередью (он воплощен в структуре выполнения задач `Executor`, описанной в главах 6 и 8).

Представьте двух человек, один из которых моет посуду и помещает ее на стеллаж, а другой — забирает ее с полок и сушит. Стеллаж действует как блокирующая очередь: если на стеллаже нет посуды, то потребитель ждет ее появления, а если стеллаж заполняется, то производитель должен приостановить мытье.

Понятия «производитель» и «потребитель» относительны. Сушка посуды «потребляет» влажную посуду и «производит» сухую. Если третий человек начнет убирать сухую посуду, сушка окажется в двух рабочих очередях.

Блокирующие очереди упрощают кодирование потребителей благодаря методу `take`. Если производители не генерируют работу достаточно быстро, потребители просто ждут. Это приемлемо как в серверном приложении, где ни один клиент не запрашивает заданий, так и в веб-обходчике, в котором существует бесконечная работа.

Если производители неуклонно генерируют работу быстрее, чем потребители могут ее обработать, то со временем у приложения заполняется память. Метод `put` упрощает кодирование производителей, благодаря чему в *ограниченной очереди* у потребителей есть время, чтобы нивелировать свое отставание.

В свою очередь, метод `offer` возвращает статус сбоя, если элемент не может быть размещен в очереди. Это позволяет создавать более гибкие политики для работы с перегрузкой, такие как сброс нагрузки, сериализа-

ция избыточных рабочих элементов и запись их на диск и регулирование числа производящих потоков.

Ограниченные очереди делают вашу программу устойчивой к перегрузке.

Встраивайте управление ресурсами в проект с помощью блокирующих очередей, причем гораздо проще делать это заранее, чем модернизировать код позже. Если блокирующие очереди не вписываются в ваш проект, то вы можете создать другие блокирующие структуры данных с помощью семафора `Semaphore` (см. раздел 5.5.3).

Библиотека классов содержит несколько реализаций блокирующей очереди `BlockingQueue`. Очереди `LinkedBlockingQueue` и `ArrayBlockingQueue` являются очередями с дисциплиной доступа FIFO, аналогичными связному списку `LinkedList` и массиву-списку `ArrayList`, но с лучшей конкурентной производительностью, чем синхронизированный список `List`. Очередь `PriorityBlockingQueue` является очередью с приоритетом, которая полезна при обработке элементов в порядке, отличном от FIFO. Как и другие сортированные коллекции, `PriorityBlockingQueue` может сравнивать элементы в соответствии с их естественным порядком с помощью `Comparable` или `Comparator`.

Реализация `SynchronousQueue` не содержит места для хранения элементов, но поддерживает список *потоков*, ожидающих постановки элемента в очередь или его удаления. При отсутствии стеллажа и непосредственной передаче вымытой посуды явно экономится время. Прямая эстафетная передача также подает больше обратных сигналов производителю о состоянии отданной им задачи. В синхронизированной очереди методы `put` и `take` блокируют продвижение, когда другой поток отказывается от ожидания эстафетной передачи, поэтому данная реализация уместна при достаточном количестве потребителей и их готовности брать работу.

5.3.1. Пример: поиск на рабочем столе

В листинге 5.8 разложим на производителей и потребителей работу агента, который сканирует локальные диски на наличие документов и индексирует их для последующего поиска (подобно Google Desktop или службе индексирования Windows). Класс `DiskCrawler` показывает задачу производителя по поиску нужных файлов и постановке их имен в рабочую

очередь. Класс `Indexer` показывает задачу потребителя по принятию имен файлов из очереди и их индексации.

Разложение обхода файлов и индексации на отдельные действия приводит к лучшей читаемости кода.

Производитель и потребитель могут выполняться конкурентно: если один привязан к вводу-выводу, а другой — к процессору, то конкурентное их выполнение даст более высокую совокупную пропускную способность, чем последовательное.

Листинг 5.9 запускает несколько обходчиков и индексаторов в отдельных потоках. Потребляющие потоки никогда не завершаются, что не дает программе терминироваться (мы рассмотрим решения этой проблемы в главе 7). В этом примере используются явно управляемые потоки, но паттерны «производитель-потребитель» могут быть выражены и с помощью структуры выполнения задач `Executor`.

5.3.2. Серийное ограничение одним потоком

Все реализации блокирующей очереди в `java.util.concurrent` содержат внутреннюю синхронизацию, достаточную для безопасной публикации объектов из производящего потока в потребляющий.

Паттерны «производитель-потребитель» и блокирующие очереди поддерживают *серийное ограничение одним потоком* (serial thread confinement) мутируемых объектов. Эксклюзивное владение объектом может быть «перенесено» другому потоку путем безопасной публикации. Исходный владелец не будет касаться объекта снова, а новый владелец может свободно его изменять.

Объектные пулы используют такое ограничение для «одалживания» объекта. Если пул содержит внутреннюю синхронизацию, а клиенты не собираются публиковать принадлежащий пулу объект или использовать его после возвращения в пул, владение может безопасно передаваться из потока в поток.

Для передачи права владения мутируемым объектом можно использовать и другие механизмы публикации, но необходимо обеспечивать передачу эстафеты только одному потоку. Лучше всего с этим справляются блокирующие очереди. Немного дополнительной работы в этом направлении по-

требуют атомарный метод `remove` ассоциативного массива `ConcurrentMap` или метод `compareAndSet` атомарной ссылки `AtomicReference`.

Листинг 5.8. Задачи производителя и потребителя в настольном поисковом приложении

```
public class FileCrawler implements Runnable {
    private final BlockingQueue<File> fileQueue;
    private final FileFilter fileFilter;
    private final File root;
    ...
    public void run() {
        try {
            crawl(root);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
         }
    }

    private void crawl(File root) throws InterruptedException {
        File[] entries = root.listFiles(fileFilter);
        if (entries != null) {
            for (File entry : entries)
                if (entry.isDirectory())
                    crawl(entry);
                else if (!alreadyIndexed(entry))
                    fileQueue.put(entry);
        }
    }
}

public class Indexer implements Runnable {
    private final BlockingQueue<File> queue;

    public Indexer(BlockingQueue<File> queue) {
        this.queue = queue;
    }

    public void run() {
        try {
            while (true)
                indexFile(queue.take());
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }
}
```

Листинг 5.9. Запуск поиска на рабочем столе

```
public static void startIndexing(File[] roots) {
    BlockingQueue<File> queue = new LinkedBlockingQueue<File>(BOUND);
    FileFilter filter = new FileFilter() {
        public boolean accept(File file) { return true; }
    };

    for (File root : roots)
        new Thread(new FileCrawler(queue, filter, root)).start();

    for (int i = 0; i < N_CONSUMERS; i++)
        new Thread(new Indexer(queue)).start();
}
```

5.3.3. Двухсторонние очереди и кража работы

Java 6 также предоставляет очереди `Deque` и `BlockingDeque`, которые расширяют очереди `Queue` и `BlockingQueue`. Класс `Deque` — это двусторонняя очередь, которая позволяет эффективно вставлять и удалять как голову очереди, так и ее хвост. Ее реализации включают `ArrayDeque` и `LinkedBlockingDeque`.

Блокирующие очереди поддаются паттерну «производитель-потребитель», а двусторонние очереди — родственному паттерну под названием *кража работы* (work stealing). Паттерн «производитель-потребитель» имеет одну совместную рабочую очередь для всех потребителей. В схеме кражи работы каждый потребитель имеет свою собственную двустороннюю очередь. Если потребитель исчерпывает работу в собственной очереди, он может украсть работу из чужой, что улучшает масштабируемость из-за сокращения конфликта: потребитель чаще обращается к собственной очереди, а просмотр чужой очереди начинает с хвоста.

Кража работы подходит для решения задач, в которых потребители одновременно являются производителями: например, в алгоритмах разведывания графов, таких как маркировка кучи во время сбора мусора, или обработках страницы веб-обходчиком, которые приводят к идентификации новых страниц для обхода. Когда потребитель выявляет новую единицу работы, он помещает ее в конец собственной очереди (либо, по схеме *совместного использования работы*, в чужую очередь), а когда его очередь пуста, он ищет работу в конце чьей-то очереди. Таким образом каждый потребитель занят.

5.4. Блокирующие и прерываемые методы

Потоки могут *блокировать* продвижение, если ожидают завершения ввода-вывода, приобретения замка, пробуждения ото сна `Thread.sleep` или результата вычисления в другом потоке. Помещаясь в состояние `BLOCKED`, `WAITING` или `TIMED_WAITING`, поток не контролирует задачу, завершения которой ожидает, и возвращается в состояние работоспособности `RUNNABLE` только после возникновения внешнего события.

Методы `put` и `take` класса `BlockingQueue` выдают проверяемое исключение `InterruptedException`, как и ряд других библиотечных методов, таких как `Thread.sleep`. Данное исключение отличает блокирующие методы, которые активно сопротивляются собственному *прерыванию*.

Класс `Thread` предоставляет метод `interrupt` для прерывания потока и для запроса о подтверждении прерывания. Каждый поток имеет булево свойство, представляющее статус прерванности.

Прерывание представляет собой *кооперативный* механизм. Когда поток *A* пытается прерывать поток *B*, то просто просит поток *B* прекратить это делать. Отменой длительных действий занимаются не потоки, а блокирующие методы.

Вызывая метод, который выдает исключение `InterruptedException`, ваш метод также становится блокирующим и должен уметь откликаться на прерывание. В случае библиотечного кода существует два варианта:

Распространить исключение `InterruptedException`**.** Эта политика предусматривает неотлавливание исключения либо его отлавливание и повторную выдачу после выполнения краткой очистки.

Восстановить прерывание. Когда распространить исключение невозможно (например, код является частью `Runnable`), необходимо перехватить исключение и восстановить статус прерванности, вызвав метод `interrupt` в текущем потоке, чтобы код выше в стеке вызовов видел выполненное прерывание (см. листинг 5.10).

Вариантов еще много, но существует одна вещь, которую вы *не* должны делать с исключением `InterruptedException` — отлавливать его и ничего не делать в ответ. Это лишает код выше в стеке вызовов возможности действовать по прерыванию, из-за невидимости последнего. *Единственная ситуация, в которой приемлемо проглатывание прерывания, — это*

расширение класса `Thread`, *когда вы контролируете весь код выше в стеке вызовов*. Отмена и выключение подробнее рассмотрены в главе 7.

Листинг 5.10. Восстановление прерванного статуса

```
public class TaskRunnable implements Runnable {
    BlockingQueue<Task> queue;
    ...
    public void run() {
        try {
            processTask(queue.take());
        } catch (InterruptedException e) {
            // восстановить статус прерванности
            Thread.currentThread().interrupt();
        }
    }
}
```

5.5. Синхронизаторы

Блокирующие очереди не только играют роль контейнеров для объектов, но и могут координировать поток управления в производящих и потребляющих потоках, поскольку методы `take` и `put` блокируют продвижение, до тех пор пока очередь не перейдет в нужное состояние (будет не пустой и не полной).

Синхронизатор (synchronizer) — это любой объект, координирующий поток управления в остальных потоках, основываясь на их состоянии. В качестве синхронизаторов могут выступать блокирующие очереди, семафоры, барьеры и защелки. В платформенной библиотеке существует несколько классов синхронизаторов. Если они не отвечают вашим потребностям, то вы можете создать свои (см. главу 14).

Все синхронизаторы инкапсулируют состояние, которое определяет, пропускать или отправлять в ожидание поступающие потоки. Для этого они предоставляют методы эффективного ожидания нужного состояния и управления им.

5.5.1. Защелки

Защелка (latch) представляет собой синхронизатор, который может задерживать продвижение потоков до достижения своего *конечного* состояния

[CPJ 3.4.2]. Он действует как закрытый шлюз, который в определенный момент разрешает всем потокам пройти, и навсегда остается открытым. Защелки сдерживают некие действия до завершения других важных действий, приведенных ниже.

- Инициализация ресурсов. На простой двоичной (с двумя состояниями) защелке будет указано, что «ресурс *R* был инициализирован», и любое действие, которое потребует *R*, станет ожидать на этой защелке.
- Запуск служб, от которых зависит действие. Запуск службы *S*, имеющей ассоциированную двоичную защелку, будет включать ожидание защелок для других служб, от которых зависит *S*, а затем освобождение защелки *S* после завершения запуска, для того чтобы любые службы, зависящие от *S*, могли продолжить работу.
- Готовность всех участников (например, в многопользовательской игре). Защелка достигает конечного состояния, после того как все игроки будут готовы.

Класс `CountDownLatch` представляет собой гибкую реализацию защелки. Состояние защелки состоит из счетчика, инициализируемого положительным числом ожидаемых событий. Метод `countDown` уменьшает счетчик, сигнализируя о том, что произошло событие, и методы `await` ожидают до тех пор, пока счетчик не достигнет нуля, т. е. все события завершатся.

В листинге 5.11 `TestHarness` создает несколько потоков, выполняющих одну задачу конкурентно, и использует две защелки: начальную (инициализируется числом 1) и конечную (инициализируется числом рабочих потоков). Потоки ожидают на начальной защелке готовности каждого из них к запуску. В конце выполнения задачи каждый поток проведет отсчет в обратном направлении на конечной защелке, чтобы главный поток дождался завершения всех потоков и вычислил затраченное время.

Защелки в `TestHarness` нужны именно для того, чтобы измерить время, необходимое для выполнения задачи *n* раз *конкурентно*.

5.5.2. FutureTask

`FutureTask` действует как защелка: он реализует `Future`, описывающий абстрактные вычисления, приносящие результат [CPJ 4.3.3], которые реа-

лизуются с помощью интерфейса `Callable`, эквивалентного `Runnable`. Они могут находиться в состоянии ожидания выполнения, самого выполнения либо завершения (нормального завершения, отмены или прерывания). `FutureTask`, приняв завершенное состояние, остается в нем навсегда.

Поведение метода `Future.get` зависит от состояния задачи. Если она завершена, то метод возвратит результат немедленно. В противном случае он будет блокировать продвижение до тех пор, пока задача не перейдет в завершенное состояние, а затем возвратит результат или создаст исключение. `FutureTask` перенесет результат из потока, выполняющего вычисление, в поток, извлекающий результат. Спецификация `FutureTask` гарантирует безопасную публикацию результата через такой трансфер.

Листинг 5.11. Использование CountDownLatch для запуска и остановки потоков в тестах с хронометражем

```
public class TestHarness {
    public long timeTasks(int nThreads, final Runnable task)
            throws InterruptedException {
    final CountDownLatch startGate = new CountDownLatch(1);
    final CountDownLatch endGate = new CountDownLatch(nThreads);

    for (int i = 0; i < nThreads; i++) {
        Thread t = new Thread() {
            public void run() {
                try {
                        startGate.await();
                        try {
                            task.run();
                        } finally {
                            endGate.countDown();
                        }
                    } catch (InterruptedException ignored) { }
                }
            };
            t.start();
        }

        long start = System.nanoTime();
        startGate.countDown();
        endGate.await();
        long end = System.nanoTime();
        return end-start;
    }
}
```

FutureTask используется структурой выполнения задач Executor для представления асинхронных задач и потенциально длительных вычислений, запускаемых до того, как понадобятся результаты. Класс предварительной загрузки Preloader в листинге 5.12 использует FutureTask для выполнения дорогостоящих вычислений, результаты которых потребуются позже. Досрочные вычисления сэкономят время в будущем.

Листинг 5.12. Использование FutureTask для предварительной загрузки данных, которые потребуются позже

```
public class Preloader {
    private final FutureTask<ProductInfo> future =
        new FutureTask<ProductInfo>(new Callable<ProductInfo>() {
            public ProductInfo call() throws DataLoadException {
                return loadProductInfo();
            }
        });

    private final Thread thread = new Thread(future);

    public void start() { thread.start(); }

    public ProductInfo get()
            throws DataLoadException, InterruptedException {
        try {
            return future.get();
        } catch (ExecutionException e) {
            Throwable cause = e.getCause();
            if (cause instanceof DataLoadException)
                throw (DataLoadException) cause;
            else
                throw launderThrowable(cause);
        }
    }
}
```

Класс Preloader создает FutureTask, который описывает задачу загрузки информации из базы данных, и поток, в котором будет выполняться вычисление. Также он предоставляет метод start для запуска потока, так как не рекомендуется запускать поток из конструктора или статического инициализатора. Когда программе позже потребуется ProductInfo, она сможет вызвать метод get, который возвратит загруженные данные, если они готовы, либо сначала дождется завершения загрузки.

Задачи, описываемые интерфейсом Callable, могут создавать проверяемые и непроверяемые исключения, и любой код может выдавать Error.

Все, что выдает код, обернуто в `ExecutionException` и повторно выдается из `Future.get`, что усложняет код, вызывающий метод `get`. Сложности возникают не только из-за исключений, но и потому, что причина исключения `ExecutionException` возвращается как суперкласс `Throwable`, с которым неудобно иметь дело.

Когда метод `get` выдает исключение `ExecutionException` в классе `Preloader`, причиной этого явления могут стать проверяемое исключение, исключение `RuntimeException` или ошибка `Error`. Нужно обрабатывать каждый из этих случаев отдельно, но в листинге 5.13 мы воспользуемся служебным методом `launderThrowable`. Перед его вызовом `Preloader` проверит наличие известных проверяемых исключений и их возвратит, после чего воспользуется служебным методом для обработки непроверяемых исключений. Если `launderThrowable` сочтет `Throwable` ошибкой `Error`, он непосредственно выдаст его повторно. Если решит, что `Throwable` не является исключением `RuntimeException`, — выдаст исключение `IllegalStateException`, сигнализируя о логической ошибке. В случае идентификации исключения как `RuntimeException` метод `launderThrowable` вернет `Throwable` вызывающему его элементу кода.

Листинг 5.13. Приведение непроверяемого Throwable к исключению RuntimeException

```
/** Если Throwable является ошибкой Error, то выдать ее;
* если же это исключение RuntimeException, то вернуть его,
* в противном случае выдать IllegalStateException
*/
public static RuntimeException launderThrowable(Throwable t) {
    if (t instanceof RuntimeException)
        return (RuntimeException) t;
    else if (t instanceof Error)
        throw (Error) t;
    else
        throw new IllegalStateException("Не является непроверяемым", t);
}
```

5.5.3. Семафоры

Счетные семафоры (counting semaphores) регулируют число действий, способных обращаться к определенному ресурсу или выполнять одну и ту же задачу в одно и то же время [CPJ 3.4.1]. Счетные семафоры можно

использовать для реализации ресурсных пулов или наложения лимита на коллекцию.

Класс `Semaphore` управляет набором виртуальных *разрешений* (permits). Начальное число разрешений передается конструктору класса `Semaphore`. Действия могут приобретать разрешения (не все сразу) и освобождать их, когда работа с ними закончена. Если разрешения отсутствуют, то метод `acquire` блокирует продвижение до тех пор, пока не появится хотя бы одно из них (либо пока не наступит прерывание или время операции не истечет). Метод `release` возвращает разрешение семафору[1]. Двоичный семафор с начальным счетчиком, равным единице, можно использовать в качестве *мьютекса* с семантикой повторно невходимого замка, и любой владелец единственного разрешения будет владеть мьютексом.

Семафоры полезны для реализации ресурсных пулов. Можно построить пул фиксированного размера, который выдаст сбой, если запросить из него лишний ресурс. Но лучше *блокировать* продвижение пустого пула, которое можно при пополнении пула разблокировать. Инициализируя `Semaphore` размером пула, вы приобретаете разрешение (методом `acquire`) до того, как будет сделана попытка доставить ресурс из пула, и освобождаете разрешение (методом `release`) после помещения ресурса обратно в пул. Метод `acquire` блокирует продвижение, до тех пор пока пул не перестанет быть пустым. Это техническое решение используется в классе ограниченного буфера, описанном в главе 12. Более простой способ конструирования блокирующего пула — очередь `BlockingQueue` для хранения объединенных в пул ресурсов.

Как показано в классе `BoundedHashSet` в листинге 5.14, `Semaphore` превращает любую коллекцию в блокирующую связанную коллекцию. Он инициализируется желаемым максимальным размером коллекции. Операция `add` приобретает разрешение для добавления элемента в базовую коллекцию, и если она ничего не добавляет, то разрешение освобождается. Схожим образом действует операция `remove`.

---

[1] Реализация не имеет фактических объектов-разрешений, и класс `Semaphore` не ассоциирует раздаваемые разрешения с потоками, поэтому разрешение, приобретенное в одном потоке, может быть освобождено из другого потока. Таким образом, метод `acquire` — потребитель, а метод `release` — создатель разрешения. Объект `Semaphore` не ограничен числом разрешений, с которыми он был создан.

5.5.4. Барьеры

Мы видели, как защелки могут облегчить запуск группы родственных действий или ожидание завершения их выполнения. Защелки представляют собой одноразовые объекты: если защелка попала в конечное состояние, ее уже нельзя обнулить.

Барьеры (barriers) подобны защелкам в том, что они блокируют группу потоков до наступления какого-то события [CPJ 4.4.3]. Но в отличие от защелки барьер заставляет потоки вместе проходить барьерную точку *в одно и то же время*, чтобы продолжить работу. Защелки предназначены для ожидания *событий*, а барьеры — для ожидания *других потоков*. Барьер реализует протокол, который похож на назначение групповой встречи: «Подходите к “Макдоналдсу” в 18:00, оставайтесь там, пока все не соберутся, потом решим, что будем делать дальше».

Класс `CyclicBarrier` позволяет фиксированному числу сторон неоднократно назначать встречу в *барьерной точке*. Он полезен в параллельных итеративных алгоритмах, которые разбивают задачу на фиксированное число независимых подзадач. Потоки достигают барьерной точки и вызывают метод `await`, который блокирует продвижение, пока *все* потоки не сделают то же самое. Если барьер успешно пройден, все потоки выпускаются, и барьер переустанавливается для следующего использования. Если время на вызов метода `await` истекает либо блокированный им поток прерывается, барьер считается *неисправным*, и все вызовы метода `await`, ожидающие обработки, терминируются с исключением `BrokenBarrierException`. Если барьер успешно пройден, то метод `await` возвращает уникальный индекс прибытия каждому потоку. `CyclicBarrier` также позволяет передавать *барьерное действие* конструктору, реализуя интерфейс `Runnable` в одном из подзадачных потоков, после прохождения барьера, но до освобождения блокированных потоков.

Листинг 5.14. Использование семафора для связывания коллекции

```
public class BoundedHashSet<T> {
    private final Set<T> set;
    private final Semaphore sem;

    public BoundedHashSet(int bound) {
        this.set = Collections.synchronizedSet(new HashSet<T>());
        sem = new Semaphore(bound);
    }
```

продолжение ⇩

Листинг 5.14 *(продолжение)*

```
    public boolean add(T o) throws InterruptedException {
        sem.acquire();
        boolean wasAdded = false;
        try {
            wasAdded = set.add(o);
            return wasAdded;
        }
        finally {
            if (!wasAdded)
                sem.release();
         }
    }

    public boolean remove(Object o) {
        boolean wasRemoved = set.remove(o);
        if (wasRemoved)
            sem.release();
        return wasRemoved;
    }
}
```

Барьеры часто используются в симуляциях, где шаги рассчитываются параллельно, а работа, ассоциированная с шагами, выполняется поочередно. Например, в симуляциях n-частиц шаг вычисляет обновление позиции частицы на основе характеристик других частиц. Ожидание у барьера обеспечивает завершенность всех обновлений для шага k, прежде чем будет выполнен переход к шагу $k + 1$.

Класс `CellularAutomata` в листинге 5.15 демонстрирует использование барьера в клеточных автоматах, таких как игра «Жизнь» Конвея (Gardner, 1970). Во время параллелизации симуляции нецелесообразно назначать отдельный поток каждому элементу (ячейке в игре «Жизнь») и принято разбивать задачу на несколько *подчастей*, каждую из которых выполнит отдельный поток, а затем результаты объединятся. `CellularAutomata` разделяет доску на N_{cpu} частей (где N_{cpu} — это число имеющихся процессоров) и назначает потокам подчасти[1]. На каждом шаге рабочие потоки вычисляют новые значения для всех ячеек в их части доски. Когда

[1] Для вычислительных задач, которые не делают ввода-вывода и не обращаются к совместно используемым данным, потоки N_{cpu} или $N_{cpu} + 1$ выдают оптимальную пропускную способность. Увеличение числа потоков может ухудшить производительность из-за конфликта за ресурсы процессора и памяти.

потоки достигают барьера, барьерное действие передает новые значения в модель данных. После этого потоки освобождаются для вычисления следующего шага, который включает консультации с методом `isDone` по определению необходимости в дальнейших итерациях.

Еще одной формой барьера является обменник `Exchanger` — двухсторонний барьер, в котором стороны обмениваются данными в барьерной точке [CPJ 3.4.3]. Обменники полезны, когда стороны выполняют асимметричные действия, например, когда один поток заполняет буфер данными, а другой потребляет данные из буфера. Потоки могут использовать `Exchanger` для встречи и обмена полного буфера на пустой. Такой обмен является безопасной публикацией объектов.

Временная координация обмена зависит от требований к отзывчивости приложения. Самый простой подход заключается в том, что заполняющая задача обменивается тогда, когда буфер заполнен, а опустошающая задача — когда буфер пуст, чтобы минимизировать число обменов. Однако, если скорость поступления новых данных непредсказуема, обработка некоторых из них будет задержана. Другой подход состоит в обмене заполняющей задачи, когда буфер заполнен частично и прошло определенное количество времени.

5.6. Создание эффективного масштабируемого кэша результатов

Почти каждое серверное приложение использует кэширование. Многоразовое использование результатов предыдущих вычислений может увеличить пропускную способность за счет дополнительного использования памяти.

Листинг 5.15. Координирование вычислений в клеточном автомате с помощью барьера CyclicBarrier

```
public class CellularAutomata {
    private final Board mainBoard;
    private final CyclicBarrier barrier;
    private final Worker[] workers;

    public CellularAutomata(Board board) {
        this.mainBoard = board;
        int count = Runtime.getRuntime().availableProcessors();
```

продолжение ⇩

Листинг 5.15 *(продолжение)*

```
        this.barrier = new CyclicBarrier(count,
                new Runnable() {
                    public void run() {
                        mainBoard.commitNewValues();
                    }});
        this.workers = new Worker[count];
        for (int i = 0; i < count; i++)
            workers[i] = new Worker(mainBoard.getSubBoard(count, i));
    }

    private class Worker implements Runnable {
        private final Board board;

        public Worker(Board board) { this.board = board; }
        public void run() {
            while (!board.hasConverged()) {
                for (int x = 0; x < board.getMaxX(); x++)
                    for (int y = 0; y < board.getMaxY(); y++)
                        board.setNewValue(x, y, computeValue(x, y));
                    try {
                        barrier.await();
                    } catch (InterruptedException ex) {
                        return;
                    } catch (BrokenBarrierException ex) {
                        return;
                    }
                }
            }
        }

        public void start() {
            for (int i = 0; i < workers.length; i++)
                new Thread(workers[i]).start();
            mainBoard.waitForConvergence();
        }
    }
```

Мы разработаем эффективный и масштабируемый кэш для вычислительно затратной функции. И начнем с очевидного подхода — простого хеш-массива, а затем рассмотрим некоторые из его недостатков конкурентности и способы их устранения.

Интерфейс `Computable<A,V>` в листинге 5.16 описывает функцию с типом *A* на входе и типом *V* на выходе. Класс `ExpensiveFunction`, который

реализует интерфейс `Computable`, занимает много времени для вычисления своего результата. Мы создадим оболочку для `Computable`, которая запомнит результаты предыдущих вычислений и инкапсулирует процесс кэширования. (Это техническое решение называется *мемоизацией* — memoization.)

Листинг 5.16. Первоначальный подход к кэшированию с использованием HashMap и синхронизации

```
public interface Computable<A, V> {
    V compute(A arg) throws InterruptedException;
}

public class ExpensiveFunction
        implements Computable<String, BigInteger> {
    public BigInteger compute(String arg) {
        // после глубокого раздумья...
        return new BigInteger(arg);
    }
}

public class Memoizer1<A, V> implements Computable<A, V> {
    @GuardedBy("this")
    private final Map<A, V> cache = new HashMap<A, V>();
    private final Computable<A, V> c;

    public Memoizer1(Computable<A, V> c) {
        this.c = c;
    }

    public synchronized V compute(A arg) throws InterruptedException {
        V result = cache.get(arg);
        if (result == null) {
            result = c.compute(arg);
            cache.put(arg, result);
        }
        return result;
    }
}
```

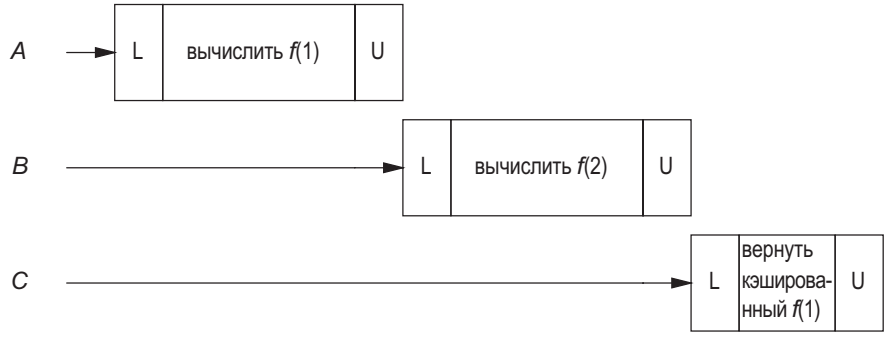

Рис. 5.2. Слабая конкурентность класса Memoizer1

Класс `Memoizer1` в листинге 5.16 показывает использование хеш-массива `HashMap` для хранения результатов предыдущих вычислений. Метод `compute` проверяет, находится ли нужный результат в кэше, и возвращает предварительно вычисленное значение, если это так. В противном случае перед возвращением результат вычисляется и кэшируется в `HashMap`.

Класс `HashMap` не является потокобезопасным. Чтобы два потока не обращались к `HashMap` одновременно, `Memoizer1` синхронизирует метод `compute` целиком, создавая проблему масштабируемости, так как выполнять вычисления может только один поток за раз. На рис. 5.2 показано, что может произойти, если потоки попытаются использовать функцию, которую запомнила обертка. Это не то улучшение производительности, которого мы надеялись достичь с помощью кэширования.

Класс `Memoizer2` в листинге 5.17 улучшает конкурентное поведение класса `Memoizer1`, заменив `HashMap` на потокобезопасный `ConcurrentHashMap`, который устраняет необходимость в синхронизации при обращении к резервному ассоциативному массиву `Map` и сериализацию, вызванную синхронизацией метода `compute`.

Но у класса есть ошибки в кэше. Он содержит окно уязвимости, в котором два потока, вызывающие метод `compute` одновременно, могут вычислить одно и то же значение, что противоречит цели кэширования.

Если один поток в `Memoizer2` запускает дорогостоящее вычисление, другие потоки, не зная о нем, могут начать то же самое вычисление, как показано на рисунке 5.3. Мы же хотим, чтобы поток, который ищет некую функцию, уже вычисляемую другим потоком, видел этот поток, наблюдал за его работой и пользовался его результатом.

Листинг 5.17. Замена HashMap на ConcurrentHashMap

```
public class Memoizer2<A, V> implements Computable<A, V> {
    private final Map<A, V> cache = new ConcurrentHashMap<A, V>();
    private final Computable<A, V> c;

    public Memoizer2(Computable<A, V> c) { this.c = c; }

    public V compute(A arg) throws InterruptedException {
        V result = cache.get(arg);
        if (result == null) {
            result = c.compute(arg);
            cache.put(arg, result);
        }
        return result;
    }
}
```

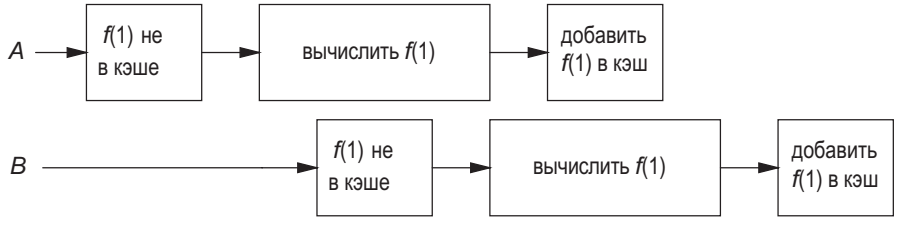

Рис. 5.3. Два потока вычисляют одно и то же значение при использовании Memoizer2

Мы уже видели класс, который делает почти то же самое: `FutureTask`. Метод `FutureTask.get`, имея результат вычисления, сразу возвращает его или блокирует продвижение, пока результат не будет вычислен, а затем возвращает его.

В листинге 5.18 использован `ConcurrentHashMap<A,Future<V>>` вместо `ConcurrentHashMap<A,V>`. Класс `Memoizer3` сначала проверяет факт запуска вычисления (в `Memoizer2` проверялось завершение) и ожидает результата существующего вычисления. В отсутствие подтверждения запуска он создает объект `FutureTask`, регистрирует его в ассоциативном массиве и запускает вычисление. Результат прозрачен для элемента кода, вызывающего метод `Future.get`.

Реализация класса `Memoizer3` демонстрирует очень хорошую конкурентность, но остается окно уязвимости (оно меньше, чем в `Memoizer2`), в котором два потока могут вычислить одно и то же значение. Блок `if` в методе `compute` по-прежнему является неатомарной последовательностью «проверить и затем действовать», и два потока могут одновременно вызвать метод `compute` с одним и тем же значением, увидеть, что кэш не содержит желаемого значения, и начать вычисление. Неудачная временная координация показана на рис. 5.4.

Листинг 5.18. Запоминающая обертка с использованием FutureTask

```
public class Memoizer3<A, V> implements Computable<A, V> {
    private final Map<A, Future<V>> cache
            = new ConcurrentHashMap<A, Future<V>>();
    private final Computable<A, V> c;

    public Memoizer3(Computable<A, V> c) { this.c = c; }

    public V compute(final A arg) throws InterruptedException {
        Future<V> f = cache.get(arg);
        if (f == null) {
            Callable<V> eval = new Callable<V>() {
                public V call() throws InterruptedException {
                    return c.compute(arg);
                }
            };
            FutureTask<V> ft = new FutureTask<V>(eval);
            f = ft;
            cache.put(arg, ft);
            ft.run(); // вызов c.compute происходит тут
        }
        try {
            return f.get();
        } catch (ExecutionException e) {
            throw launderThrowable(e.getCause());
        }
    }
}
```

Класс `Memoizer3` уязвим, потому что составная операция «добавить, если отсутствует» выполняется на резервном ассоциативном массиве, и ее нельзя сделать атомарной с помощью замка. Класс `Memoizer` в листинге 5.19 закрывает окно уязвимости атомарным методом `putIfAbsent` ассоциативного массива `ConcurrentMap`.

Кэширование `Future` создает угрозу *загрязнения кэша*: если вычисление отменяется или завершится сбоем, то дальнейшие попытки вычислить результат тоже будут указывать на отмену или сбой. Поэтому `Memoizer` удаляет `Future` из кэша, если обнаруживает, что вычисление было отменено. Также можно удалить `Future` при обнаружении исключения `RuntimeException`, если вычисление может быть успешным при будущей попытке. `Memoizer` не решает задачу с истечением срока действия кэша, но он может обратиться к подклассу `FutureTask`, который ассоциирует время истечения срока с каждым результатом и периодически проверяет кэш на наличие данных с истекшим сроком. (Также `Memoizer` не решает проблему вытеснения кэша, при котором старые записи удаляются для разгрузки памяти.)

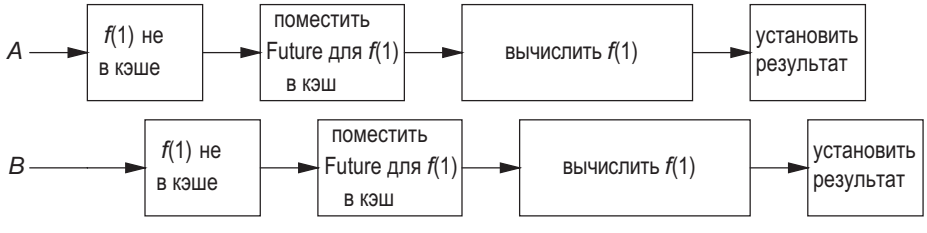

Рис. 5.4. Неудачная временна́я координация, которая может побудить Memoizer3 вычислять одно и то же значение дважды

Теперь, когда конкурентная реализация кэша завершена, мы можем добавить реальное кэширование в сервлет разложения на множители из главы 2, как и обещали. Класс `Factorizer` в листинге 5.20 эффективно использует класс `Memoizer` для кэширования ранее вычисленных значений.

Листинг 5.19. Окончательная реализация класса Memoizer

```
public class Memoizer<A, V> implements Computable<A, V> {
    private final ConcurrentMap<A, Future<V>> cache
        = new ConcurrentHashMap<A, Future<V>>();
    private final Computable<A, V> c;
```

продолжение ⇗

Листинг 5.19 *(продолжение)*

```
    public Memoizer(Computable<A, V> c) { this.c = c; }

    public V compute(final A arg) throws InterruptedException {
        while (true) {
            Future<V> f = cache.get(arg);
            if (f == null) {
                Callable<V> eval = new Callable<V>() {
                    public V call() throws InterruptedException {
                        return c.compute(arg);
                    }
                };
                FutureTask<V> ft = new FutureTask<V>(eval);
                f = cache.putIfAbsent(arg, ft);
                if (f == null) { f = ft; ft.run(); }
            }
            try {
                return f.get();
            } catch (CancellationException e) {
                cache.remove(arg, f);
            } catch (ExecutionException e) {
                throw launderThrowable(e.getCause());
            }
        }
    }
}
```

Листинг 5.20. Сервлет разложения на множители, который кэширует результаты, используя Memoizer

```
@ThreadSafe
public class Factorizer implements Servlet {
    private final Computable<BigInteger, BigInteger[]> c =
        new Computable<BigInteger, BigInteger[]>() {
            public BigInteger[] compute(BigInteger arg) {
                return factor(arg);
            }
        };
    private final Computable<BigInteger, BigInteger[]> cache
        = new Memoizer<BigInteger, BigInteger[]>(c);

    public void service(ServletRequest req,
                        ServletResponse resp) {
        try {
            BigInteger i = extractFromRequest(req);
```

```
            encodeIntoResponse(resp, cache.compute(i));
        } catch (InterruptedException e) {
            encodeError(resp, "факторизация прервана");
        }
    }
}
```

Итоги

Мы рассмотрели довольно много материала! В приведенной ниже памятке кратко изложены основные понятия и правила, представленные в части I.

- *Это мутируемое состояние, дурачок*[1].

  Все вопросы конкурентности сводятся к координированию доступа к мутируемому состоянию. Чем менее мутируемо состояние, тем легче обеспечить потокобезопасность.

- *Объявляйте поля финальными, если нет необходимости в том, чтобы они были мутируемыми.*

- *Немутируемые объекты автоматически являются потокобезопасными*.

  Немутируемые объекты упрощают конкурентное программирование. Они могут свободно использоваться без блокировки или защитного копирования.

- *Инкапсуляция позволяет управлять сложностью*.

  Можно написать потокобезопасную программу, в которой все данные хранятся в глобальных переменных, но зачем? Инкапсулирование данных внутри объектов упрощает соблюдение их инвариантов, а инкапсулирование синхронизации внутри объектов упрощает соблюдение их политики синхронизации.

- *Защищайте каждую мутируемую переменную с помощью замка́.*

[1] Во время президентских выборов в США 1992 г. Джеймс Карвилл повесил в предвыборном штабе Билла Клинтона табличку с надписью «Это экономика, дурачок» для постоянного напоминания о ней участникам кампании.

- *Защищайте все переменные в инварианте с помощью одинаковых замко́в.*
- *Удерживайте замки́ в течение всего времени составных действий.*
- *Программа, которая обращается к мутируемой переменной из многочисленных потоков без синхронизации, неисправна.*
- *Не верьте доводам, что обойдетесь без синхронизации.*
- *Включайте потокобезопасность в процесс проектирования либо явным образом документируйте то, что класс не является потокобезопасным.*
- *Документируйте политику синхронизации.*

Часть II
Структурирование конкурентных приложений

6 Выполнение задач

Большинство конкурентных приложений организованы вокруг выполнения *задач* (tasks) — абстрактных дискретных единиц работы. Разделение работы приложения на задачи упрощает организацию программы, облегчает обнаружение и исправление ошибок, предоставляя естественные транзакционные границы, и способствует конкурентности, обеспечивая естественную структуру для параллелизации работы.

6.1. Выполнение задач в потоках

Первым шагом в организации программы вокруг выполнения задачи является определение разумных *границ задач* (task boundaries). *Независимые* друг от друга задачи могут выполняться параллельно при наличии достаточных обрабатывающих ресурсов. Для большей гибкости в планировании и распределении нагрузки каждая задача должна представлять собой небольшую часть вычислительной емкости приложения.

Серверные приложения должны демонстрировать *хорошую пропускную способность*, *высокую отзывчивость*, а также *плавную деградацию* по мере перегрузки. В разделе 6.2.2 мы описали выбор задачных границ в зависимости от разумной *политики* их *выполнения*.

Большинство серверных приложений предлагают в качестве границы задачи использовать индивидуальные клиентские запросы, которые

обеспечивают как независимость, так и надлежащий размер задачи. Веб-серверы, почтовые серверы, файловые серверы, контейнеры EJB и серверы баз данных принимают запросы от удаленных клиентов через сетевые подключения и обрабатывают сообщения одновременно и независимо друг от друга.

6.1.1. Последовательное выполнение задач

Некоторые политики планирования задач используют потенциал конкурентности лучше других. Одна из них предлагает выполнять задачи последовательно в одном потоке. Класс `SingleThreadWebServer` в листинге 6.1 обрабатывает HTTP-запросы, поступающие на порт 80, последовательно. Детали обработки запроса сейчас не важны — мы стараемся оценить конкурентность различных политик планирования.

Листинг 6.1. Последовательный веб-сервер

```
class SingleThreadWebServer {
    public static void main(String[] args) throws IOException {
        ServerSocket socket = new ServerSocket(80);
        while (true) {
            Socket connection = socket.accept();
            handleRequest(connection);
        }
    }
}
```

Простой и правильный класс `SingleThreadedWebServer` неэффективен в рабочей среде, поскольку обрабатывает только один запрос за раз. Пока главный поток попеременно принимает соединения и обрабатывает ассоциированный запрос, новые соединения должны ожидать завершения текущего запроса и вызова метода `accept`. К сожалению, `handleRequest` не может возвращаться мгновенно.

Обработка веб-запроса состоит из вычислений и ввода-вывода, помогающего избежать блокирования продвижения. Сокетный ввод-вывод используется для чтения запроса, записи отклика, выполнения файлового ввода-вывода и запросов к базе данных.

В большинстве серверных приложений последовательная обработка не обеспечивает хорошую пропускную способность и высокую отзывчивость[1].

6.1.2. Явное создание потоков для задач

Создание отдельного потока для обслуживания каждого запроса, как показано в классе `ThreadPerTaskWebServer` в листинге 6.2, повышает отзывчивость.

Листинг 6.2. Веб-сервер, запускающий новый поток для каждого запроса

```
class ThreadPerTaskWebServer {
    public static void main(String[] args) throws IOException {
        ServerSocket socket = new ServerSocket(80);
        while (true) {
            final Socket connection = socket.accept();
            Runnable task = new Runnable() {
                    public void run() {
                        handleRequest(connection);
                    }
                };
            new Thread(task).start();
        }
    }
}
```

Главный поток попеременно принимает входящие соединения и отправляет запросы, но вместо того чтобы самостоятельно обрабатывать запросы, он создает для них отдельные потоки, вследствие чего:

- главный цикл может принимать новые подключения до завершения предыдущих запросов;

[1] В некоторых ситуациях последовательная обработка может обеспечить простоту или безопасность. Большинство платформ GUI обрабатывают задачи последовательно, используя один поток. Мы вернемся к последовательной модели в главе 9.

- задачи могут обрабатываться параллельно, что повышает пропускную способность, если имеются многочисленные процессоры либо задача нуждается в блокировании;
- возрастает необходимость потокобезопасности кода обработки.

При небольшой и умеренной нагрузке подход «поток в расчете на задачу» уместнее последовательного выполнения. До тех пор пока скорость поступления запросов не превышает емкость сервера для обработки запросов, этот подход обеспечивает высокую отзывчивость и пропускную способность.

6.1.3. Недостатки создания неограниченных потоков

Вместе с тем подход «поток в расчете на задачу» имеет практические недостатки:

Пул потоков фиксированного размера. Создание потока занимает время и требует некоторой обрабатывающей деятельности со стороны JVM и ОС, поэтому не приносит пользы при высокой частоте запросов.

Ресурсопотребление. Активные потоки потребляют системные ресурсы, в особенности память. Свободные потоки в большом количестве могут связывать часть памяти, оказывая давление на сборщика мусора, а избыток потоков, которые конфликтуют за процессоры, может вносить другие стоимости для производительности. Если у вас достаточно потоков, чтобы занять все процессоры, создание большего числа потоков нецелесообразно.

Стабильность. Существует лимит на число создаваемых потоков, который зависит от многих факторов[1]. Достижение лимита приводит к появлению ошибки `OutOfMemoryError`, после которой восстановление рискованно. Структурируйте программу так, чтобы избежать попадания в лимит.

---

[1] На 32-разрядных машинах каждый поток поддерживает два стека выполнения: один для кода Java, другой — для машинного кода. Типичные параметры JVM дают комбинированный размер стека около половины мегабайта, который можно изменить с помощью флага `-Xss` или конструктора класса `Thread`. Разделив размер поточного стека на 2^{32}, вы получите предел в несколько тысяч или десятков тысяч потоков. Другие факторы, такие как лимиты ОС, накладывают более строгие ограничения.

Сначала увеличение числа потоков повышает пропускную способность, но затем замедляет работу и может привести к сбою всего приложения. Необходимо установить лимит на число потоков, создаваемых приложением, и обеспечить, чтобы даже при достижении этого лимита ресурсы не заканчивались (с помощью тестирования).

Подход «поток в расчете на задачу» ограничивает число создаваемых потоков только скоростью, с которой удаленные пользователи могут отправлять запросы HTTP. Для серверного приложения, которое должно обеспечивать высокую доступность и плавную деградацию под нагрузкой, это является серьезным недостатком.

6.2. Фреймворк Executor

Задачи — это логические единицы работы, а потоки — это механизм их асинхронного выполнения. Мы рассмотрели две политики выполнения задач с помощью потоков: последовательное выполнение задач в одном потоке и выполнение каждой задачи в отдельном потоке. И та, и другая политика имеют серьезные ограничения.

Ряд преимуществ предлагают *поточные пулы* (thread pools), и `java.util.concurrent` предоставляет их гибкую реализацию в структуре `Executor`. Первостепенной абстракцией для выполнения задач в библиотеках классов Java является *не* класс `Thread`, а интерфейс `Executor`, показанный в листинге 6.3.

Листинг 6.3. Интерфейс Executor

```
public interface Executor {
    void execute(Runnable command);
}
```

Интерфейс `Executor` обеспечивает отделение *предоставления задачи* от *ее выполнения* с помощью интерфейса `Runnable` и поддерживает жизненный цикл программы и перехватчиков для добавления функционала сбора статистики, управления приложением и мониторинга.

Использование `Executor` *является одним из самых простых способов реализации паттерна «производитель-потребитель».*

6.2.1. Пример: веб-сервер с использованием Executor

`TaskExecutionWebServer` в листинге 6.4 использует одну из стандартных реализаций исполнителя `Executor` — пул в 100 потоков фиксированного размера.

Предоставление задачи обработки запросов отделено от ее выполнения и зависит от выбранной реализации исполнителя. Изменение реализаций или конфигурации исполнителя `Executor` менее инвазивно, чем изменение способа предоставления задач. Конфигурация исполнителя `Executor` обычно представляет собой одно событие, в то время как код предоставления задачи разбросан по всей программе.

Листинг 6.4. Веб-сервер, использующий пул потоков

```
class TaskExecutionWebServer {
    private static final int NTHREADS = 100;
    private static final Executor exec
        = Executors.newFixedThreadPool(NTHREADS);

    public static void main(String[] args) throws IOException {
        ServerSocket socket = new ServerSocket(80);
        while (true) {
            final Socket connection = socket.accept();
            Runnable task = new Runnable() {
                public void run() {
                    handleRequest(connection);
                }
            };
            exec.execute(task);
        }
    }
}
```

После внедрения исполнителя `Executor` класс `TaskExecutionWebServer` станет вести себя как `ThreadPerTaskWebServer` (см. листинг 6.5).

Листинг 6.5. Исполнитель, запускающий отдельный поток для каждой задачи

```
public class ThreadPerTaskExecutor implements Executor {
    public void execute(Runnable r) {
        new Thread(r).start();
    };
}
```

Также с помощью исполнителя можно побудить `TaskExecutionWebServer` вести себя как один поток и выполнять каждую задачу синхронно перед возвращением из метода `execute`, как показано в `WithinThreadExecutor` в листинге 6.6.

Листинг 6.6. Исполнитель, синхронно выполняющий задачи в вызывающем потоке

```
public class WithinThreadExecutor implements Executor {
    public void execute(Runnable r) {
        r.run();
    };
}
```

6.2.2. Политики выполнения

Отделение предоставления задачи от ее выполнения позволяет легко специфицировать и впоследствии без особых трудностей изменять *политику выполнения* для выбранного класса задач. Спецификация политики выполнения отвечает на вопросы:

- В каком потоке будут выполняться задачи?
- Каков порядок выполнения задач (FIFO, LIFO, приоритетный порядок)?
- Сколько задач может выполняться конкурентно?
- Сколько задач может быть поставлено в очередь?
- Какую задачу удалить из-за перегрузки системы и как уведомить приложение?
- Какие действия предпринимать до и после выполнения задачи?

Политика выполнения — это инструмент управления ресурсами, выбор которого зависит от имеющихся вычислительных ресурсов и требований к качеству обслуживания. Ограничивая число конкурентных задач, вы можете контролировать риск завершения приложения из-за нехватки ресурсов[1].

[1] Аналогично транзакционный монитор в корпоративном приложении может регулировать скорость выполнения транзакции при ограниченных ресурсах.

Код в форме:

```
new Thread(runnable).start()
```

сигнализирует о возможности его замены на исполнитель `Executor`.

6.2.3. Пулы потоков

Пул потоков тесно привязан к *рабочей очереди* (work queue), содержащей задачи, которые ожидают выполнения. Рабочие потоки запрашивают следующую задачу из очереди, выполняют ее и возвращаются к ожиданию другой задачи.

Многоразовое использование существующего потока амортизирует стоимости, которые могли быть потрачены на отдельные потоки для запросов и улучшает отзывчивость. Правильно настроив размер пула потоков, вы обеспечите занятость процессоров и сохраните ресурсы.

Библиотека классов предоставляет гибкую реализацию пула потоков, который можно создать, вызвав один из статических фабричных методов в исполнителях `Executor`:

`newFixedThreadPool`. Пул потоков фиксированного размера создает определенное число потоков по мере предоставления задач, а затем старается держать размер пула постоянным.

`newCachedThreadPool`. Более гибкий кэшированный пул потоков убирает простаивающие потоки и при необходимости добавляет новые, не накладывая лимит на размер пула.

`newSingleThreadExecutor`. Исполнитель создает один поток для последовательной обработки задач, который при необходимости можно заменить[1].

`newScheduledThreadPool`. Пул потоков фиксированного размера, который поддерживает отложенное и периодическое выполнение задач аналогично классу `Timer` (см. раздел 6.2.5).

[1] Однопоточные исполнители обеспечивают внутреннюю синхронизацию, гарантируя, что записи в память со стороны задач видны другим задачам, а объекты могут быть ограничены «потоком задачи», даже если этот поток будет заменен на другой.

Фабричные методы `newFixedThreadPool` и `newCachedThreadPool` возвращают экземпляры универсального исполнителя `ThreadPoolExecutor`, который также может использоваться для построения более специализированных исполнителей. Мы подробно обсудим варианты конфигурации пула потоков в главе 8.

Веб-сервер в `TaskExecutionWebServer` использует класс `Executor` с ограниченным пулом рабочих потоков. Предоставление задачи с помощью метода `execute` добавляет задачу в рабочую очередь, и рабочие потоки непрерывно удаляют задачи из рабочей очереди, выполняя их.

Благодаря политике на основе пула веб-сервер не отказывает при большой нагрузке[1].

Также он деградирует более плавно, так как не создает лишние потоки, которые конфликтуют за ресурсы. Использование исполнителя `Executor` открывает расширенные возможности по настройке, управлению, мониторингу, журналированию, отчетности об ошибках и позволяет вносить другие дополнения, доступные благодаря структуре выполнения задач.

6.2.4. Жизненный цикл исполнителя Executor

Исполнитель `Executor` отключается, когда все (не являющиеся демонами) потоки терминированы, после чего следует выход JVM.

Поскольку исполнитель обрабатывает задачи асинхронно, их состояния могут отличаться. В `shutdown` есть спектр выключений от плавного (закончить начатое и не принимать новой работы) до внезапного (выключить питание). Исполнитель `Executor` тоже содержит варианты выключений и подает информацию о статусах задач.

Интерфейс `ExecutorService` расширяет интерфейс `Executor`, добавляя ряд методов управления жизненным циклом, которые показаны в листинге 6.7.

---

[1] Но если скорость поступления задач превышает скорость их обслуживания достаточно долго, остается риск исчерпания памяти из-за растущей очереди объектов `Runnable`, ожидающих выполнения. Эту проблему можно решить в рамках платформы `Executor`, используя ограниченную рабочую очередь (см. раздел 8.3.2).

Листинг 6.7. Методы жизненного цикла в интерфейсе ExecutorService

```
public interface ExecutorService extends Executor {
    void shutdown();
    List<Runnable> shutdownNow();
    boolean isShutdown();
    boolean isTerminated();
    boolean awaitTermination(long timeout, TimeUnit unit)
        throws InterruptedException;
    // ... дополнительные удобные методы предоставления задач
}
```

Жизненный цикл имеет три состояния: *работает*, *выключается* и *терминирован*. Службы `ExecutorService` создаются в *рабочем* состоянии, после чего метод `shutdown` инициирует выключение.

Задачами, предоставленными службе `ExecutorService` после ее выключения, занимается *обработчик отклоненного выполнения* (см. раздел 8.3.3), который может отклонить задачу или вызвать метод `execute` для выдачи непроверяемого исключения `RejectedExecutionException`. Как только все задачи завершены, служба `ExecutorService` переходит в *терминированное* состояние. Обычно сразу за методом `shutdown` следует метод `awaitTermination`, создающий эффект синхронного выключения службы `ExecutorService`. (Выключение исполнителя `Executor` и отмена задач рассматриваются более подробно в главе 7.)

Веб-сервер `LifecycleWebServer` в листинге 6.8 расширяет поддержку жизненного цикла веб-сервера. Его можно выключить двумя способами: программным путем, вызвав метод `stop`, и через запрос клиента, отправив веб-серверу специально отформатированный HTTP-запрос.

Листинг 6.8. Веб-сервер с поддержкой выключения

```
class LifecycleWebServer {
    private final ExecutorService exec = ...;

    public void start() throws IOException {
        ServerSocket socket = new ServerSocket(80);
        while (!exec.isShutdown()) {
            try {
                final Socket conn = socket.accept();
                exec.execute(new Runnable() {
                    public void run() { handleRequest(conn); }
                });
```

продолжение ⇩

Листинг 6.8 *(продолжение)*

```
            } catch (RejectedExecutionException e) {
                if (!exec.isShutdown())
                    log("предоставленная задача отклонена", e);
            }
        }
    }

    public void stop() { exec.shutdown(); }

    void handleRequest(Socket connection) {
        Request req = readRequest(connection);
        if (isShutdownRequest(req))
            stop();
        else
            dispatchRequest(req);
    }
}
```

6.2.5. Отложенные и периодические задачи

Механизм `Timer` управляет выполнением отложенных и периодических задач. Его альтернативой является исполнитель `ScheduledThreadPoolExecutor`[1], который можно создать с помощью его собственного конструктора либо фабричного метода `newScheduledThreadPool`.

Класс `Timer` создает только один поток выполнения таймерных задач. Если таймерная задача выполняется слишком долго, то точность хронометража других задач `TimerTask` может пострадать. Если периодическая задача `TimerTask` должна выполняться каждые 10 мс, а выполнение другой задачи `TimerTask` занимает 40 мс, то периодическая задача (в зависимости от того, была ли она запланирована с фиксированной скоростью или с фиксированной задержкой) либо быстро вызывается четыре раза подряд после завершения длительной задачи, либо пропускает четыре вызова. Пулы запланированных потоков предоставляют многочисленные потоки для выполнения отложенных и периодических задач.

---

[1] `Timer` поддерживает планирование на основе абсолютного времени, вследствие чего задачи могут быть чувствительны к изменениям в системных часах. Исполнитель `ScheduledThreadPoolExecutor` поддерживает только относительное время.

Еще одна проблема `Timer` заключается в том, что его поток не может отлавливать непроверяемое исключение, выдаваемое из задачи `TimerTask`, которое его терминирует. В этой ситуации `Timer` не восстанавливает поток, считая себя отмененным, задачи `TimerTask`, которые уже запланированы, но еще не выполнены, никогда не запускаются, и новые задачи не могут быть запланированы. (Эта проблема, именуемая *утечкой потока* (thread leakage), и ее решение описаны в разделе 7.3.)

Класс `OutOfTime` в листинге 6.9 иллюстрирует, как `Timer` путается сам и путает следующий вызывающий элемент кода, который пытается предоставить задачу `TimerTask`. Программа терминируется через одну секунду с исключением `IllegalStateException` и текстом сообщения «`Timer` отменен». Исполнитель `ScheduledThreadPoolExecutor` надлежащим образом справляется с неблагополучными задачами, и в Java 5.0 `Timer` почти не используется.

Чтобы построить свою службу планирования, воспользуйтесь очередью `DelayQueue`, предоставляющей функциональные возможности планирования в исполнителе `ScheduledThreadPoolExecutor`. Она управляет сбором отложенных объектов `Delayed`, каждый из которых имеет ассоциированное с ним время задержки: очередь `DelayQueue` позволяет принимать элемент (метод `take`), только если его задержка истекла. Объекты возвращаются из очереди `DelayQueue`, упорядоченной по времени, ассоциированному с их задержкой.

6.3. Поиск эксплуатационно-пригодного параллелизма

Чтобы использовать исполнитель `Executor`, вы должны описать свою задачу как `Runnable`. В большинстве серверных приложений существует очевидная граница задачи: единственный клиентский запрос. Но иногда хорошие границы задачи не так очевидны, как во многих настольных приложениях. В серверных приложениях также может существовать эксплуатационно-пригодный параллелизм в рамках единственного клиентского запроса, как это иногда бывает на серверах баз данных. (О конфликтующих проектных силах при выборе задачных границ см. в [CPJ 4.4.1.1].)

Листинг 6.9. Класс, иллюстрирующий запутанное поведение таймера

```
public class OutOfTime {
    public static void main(String[] args) throws Exception {
        Timer timer = new Timer();
        timer.schedule(new ThrowTask(), 1);
        SECONDS.sleep(1);
        timer.schedule(new ThrowTask(), 1);
        SECONDS.sleep(5);
    }

    static class ThrowTask extends TimerTask {
        public void run() { throw new RuntimeException(); }
    }
}
```

В этом разделе мы разработали несколько версий компонента, который допускает варианты степени конкурентности. Демонстрационный компонент представлен фрагментом отрисовки страницы браузерного приложения, который выводит страницу HTML в буфер изображений. Для простоты предположим, что HTML состоит только из размеченного текста с элементами изображений заранее заданных размеров и URL.

6.3.1. Пример: последовательный страничный отрисовщик

Последовательная обработка HTML-документа подразумевает при появлении текстовой разметки ее отрисовку в буфере изображений, а при появлении ссылок на изображение — их доставку по сети и отрисовку в буфере изображений. Такая обработка требует прикосновения к каждому входному элементу всего один раз, но может занять много времени.

Другой (тоже последовательный) подход включает в себя отрисовку сначала текстовых элементов, а затем — изображений в соответствующем заполнителе. Этот подход показан в классе `SingleThreadRenderer` в листинге 6.10.

Скачивание изображения в основном включает в себя ожидание завершения ввода-вывода. Поэтому последовательный подход может недо-

использовать процессор, а также заставит пользователя долго ожидать завершения отрисовки. Мы можем добиться лучшей задействованности ресурсов и отзывчивости приложения, разбив задачу на независимые подзадачи, которые могут выполняться конкурентно.

Листинг 6.10. Последовательная отрисовка элементов страницы

```
public class SingleThreadRenderer {
    void renderPage(CharSequence source) {
        renderText(source);
        List<ImageData> imageData = new ArrayList<ImageData>();
        for (ImageInfo imageInfo : scanForImageInfo(source))
            imageData.add(imageInfo.downloadImage());
        for (ImageData data : imageData)
            renderImage(data);
    }
}
```

6.3.2. Задачи, приносящие результаты: Callable и Future

Структура `Executor` использует для представления задач интерфейс `Runnable`, содержащий метод `run`, неспособный возвращать значение или выдавать проверяемые исключения и допускающий записи в файл журнала или размещение результата в совместной структуре данных.

Для отложенных задач интерфейс `Callable` является более подходящей абстракцией: он ожидает, что главная точка входа, `call`, вернет значение, и готов, если потребуется, выдать исключение[1]. Исполнитель `Executor` включает в себя несколько вспомогательных методов для обертывания других типов задач, включая `Runnable` и `java.security.PrivilegedAction`, с помощью `Callable`.

Интерфейсы `Runnable` и `Callable` описывают абстрактные вычислительные задачи, у которых есть четкая отправная точка, чтобы в итоге терминиро-

[1] Для того чтобы с помощью `Callable` выразить задачу, не возвращающую значения, следует использовать `Callable<Void>`.

ваться. Жизненный цикл задачи, выполняемой исполнителем, состоит из четырех этапов: *создана*, *предоставлена*, *запущена* и *завершена*. Поскольку выполнение задач может занимать много времени, мы хотим иметь возможность отменить задачу. В структуре `Executor` задачи, которые были предоставлены, но не были запущены, всегда могут быть отменены, а задачи, которые были запущены, могут быть отменены, если они откликаются на прерывание. Отмена задачи, которая была уже завершена, не имеет эффекта.

Интерфейс `Future` предоставляет методы, которые проверяют, была ли задача завершена или отменена, извлекают ее результат и при необходимости отменяют ее. Интерфейсы `Callable` и `Future` показаны в листинге 6.11. В спецификации `Future` подразумевается, что жизненный цикл задачи может двигаться только вперед, а не назад — как и жизненный цикл службы `ExecutorService`. Как только задача завершена, она остается в этом состоянии навсегда.

Поведение метода `get` зависит от состояния задачи. Он немедленно возвращается или выдает исключение, если задача завершена, и в противном случае блокирует продвижение до завершения задачи. Если задача завершается выдачей исключения, то метод `get` повторно выдает его, обернутым в исключение `ExecutionException`. Если задача отменена, то метод `get` выдает исключение `CancellationException`. Если метод `get` выдает исключение `ExecutionException`, то базовое исключение может быть получено с помощью метода `getCause`.

Листинг 6.11. Интерфейсы Callable и Future

```
public interface Callable<V> {
    V call() throws Exception;
}

public interface Future<V> {
    boolean cancel(boolean mayInterruptIfRunning);
    boolean isCancelled();
    boolean isDone();
    V get() throws InterruptedException, ExecutionException,
                   CancellationException;
    V get(long timeout, TimeUnit unit)
        throws InterruptedException, ExecutionException,
               CancellationException, TimeoutException;
}
```

Существует несколько способов создания объекта `Future` для описания задачи. Все методы `submit` в классе `ExecutorService` возвращают объект

Future, что позволяет предоставлять объекты Runnable или Callable исполнителю и получать назад объект Future, который может быть использован для получения результата или отмены задачи. Также можно явным образом создать объект, инстанцировав экземпляр FutureTask для данного Runnable или Callable. (Поскольку класс FutureTask реализует Runnable, его экземпляр может быть передан исполнителю для выполнения либо выполнен непосредственно путем вызова метода run.)

Начиная с Java 6, реализации службы ExecutorService могут переопределять метод newTaskFor в абстрактном классе AbstractExecutorService, для того чтобы контролировать создание объекта Future, соответствующего предоставленным объектам Callable либо Runnable. По умолчанию принята реализация, которая просто создает новый объект FutureTask, как показано в листинге 6.12.

Листинг 6.12. Реализация по умолчанию метода newTaskFor в классе ThreadPoolExecutor

```
protected <T> RunnableFuture<T> newTaskFor(Callable<T> task) {
    return new FutureTask<T>(task);
}
```

Предоставление объекта Runnable (или Callable) исполнителю Executor образует его безопасную публикацию (см. раздел 3.5) из предоставляющего потока в поток, который выполнит задачу. Схожим образом установка результирующего значения для объекта Future образует безопасную публикацию результата из потока, в котором он был вычислен, в любой поток, который извлекает его через метод get.

6.3.3. Пример: страничный отрисовщик с объектом Future

В качестве первого шага к тому, чтобы сделать страничный отрисовщик более конкурентным, разделим его на две подзадачи: отрисовку текста и скачивание изображений. (Поскольку одна задача в значительной степени привязана к процессору, а другая — к вводу-выводу, этот подход может привести к улучшениям даже в однопроцессорных системах.)

Объекты Callable и Future способны помочь нам выразить взаимодействие между подзадачами. В классе FutureRenderer в листинге 6.13 мы создадим объект Callable для скачивания всех изображений и предоставим его в службу ExecutorService, которая возвратит объект Future,

описывающий выполнение задачи. Когда главная задача дойдет до точки, где ей потребуется заполнить пустые области изображениями, она вызовет метод `Future.get`. Даже если при изолированном скачивании изображений мы не сможем получить результаты по первому требованию, все равно получим преимущество в производительности.

Зависимый от состояния метод `get` означает, что вызывающий его элемент кода не должен знать о состоянии задачи, а свойства безопасной публикации предоставленной задачи и получения результатов делают этот подход потокобезопасным. Код обработки исключений вокруг метода `Future.get` занимается двумя возможными проблемами: появление исключения либо прерывание потока, вызывающего метод `get`, прежде чем появились результаты. (См. разделы 5.5.2 и 5.4.)

Класс `FutureRenderer` позволяет отрисовывать текст конкурентно со скачиванием данных изображения. Когда все изображения скачаны, они отрисовываются на странице. Пользователям нет необходимости ждать, когда *все* изображения будут скачаны; они, скорее, предпочтут видеть изображения по мере их появления.

6.3.4. Ограничения параллелизации разнородных задач

Получить значительное повышение производительности с помощью параллелизации последовательных разнородных задач достаточно непросто.

Два человека могут эффективно разделить работу с посудой: один моет, другой вытирает. Тем не менее закрепление разного типа задач за каждым работником не очень хорошо масштабируется: если появится еще несколько помощников, они значительно перестроят разделение труда.

Дополнительная проблема с разделением разнородных задач между многочисленными работниками заключается в том, что задачи могут иметь несопоставимые размеры. Если вы разделите задачи *A* и *B* между двумя работниками, но *A* будет занимать в десять раз больше времени, чем *B*, то вы ускорите суммарный процесс только на 9 %. Наконец, для того чтобы разделение было оправданным, издержки на координацию должны компенсироваться повышением производительности.

Класс `FutureRenderer` использует две задачи: одну для отрисовки текста и одну для скачивания изображений. Если отрисовка текста происходит

намного быстрее, чем скачивание изображений, то результирующая производительность не будет отличаться от полученной в последовательной версии, но код будет намного сложнее. Существует предел количества дополнительной конкурентности, выигранной от разделения задачи. (Еще один пример того же феномена см. в разделах 11.4.2 и 11.4.3.)

Листинг 6.13. Ожидание загрузки изображения с объектом Future

```
public class FutureRenderer {
    private final ExecutorService executor = ...;

    void renderPage(CharSequence source) {
        final List<ImageInfo> imageInfos = scanForImageInfo(source);
        Callable<List<ImageData>> task =
                new Callable<List<ImageData>>() {
                    public List<ImageData> call() {
                        List<ImageData> result
                                = new ArrayList<ImageData>();
                        for (ImageInfo imageInfo : imageInfos)
                            result.add(imageInfo.downloadImage());
                        return result;
                    }
                };

        Future<List<ImageData>> future = executor.submit(task);
        renderText(source);

        try {
            List<ImageData> imageData = future.get();
            for (ImageData data : imageData)
                renderImage(data);
        } catch (InterruptedException e) {
            // Переподтвердить статус прерванности потока
            Thread.currentThread().interrupt();
            // Нам не нужен результат, поэтому отменить задачу
            future.cancel(true);
        } catch (ExecutionException e) {
            throw launderThrowable(e.getCause());
        }
    }
}
```

Реальная отдача от разделения рабочей нагрузки программы на задачи достигается при наличии большого числа независимых *однородных* задач, которые могут обрабатываться конкурентно.

6.3.5. CompletionService: исполнитель Executor встречается с очередью BlockingQueue

Если у вас есть пакет вычислений для предоставления исполнителю `Executor` и вы хотите извлекать их результаты по мере появления, то вы можете сохранять объект `Future`, ассоциированный с каждой задачей, и периодически опрашивать его о ее завершении, вызывая метод `get` с тайм-аутом, равным нулю. Это возможно, но утомительно. К счастью, существует оптимальный способ: *служба завершения* (`CompletionService`).

Она сочетает в себе функциональность исполнителя `Executor` и блокирующей очереди `BlockingQueue`. Вы можете предоставлять ей задачи `Callable` на выполнение и использовать методы `take` и `poll`, аналогичные тем, которые используются в очереди, для получения завершенных результатов. Класс `ExecutorCompletionService` реализует `CompletionService`, делегируя вычисление исполнителю `Executor`.

Реализация службы `ExecutorCompletionService` довольно проста. Конструктор создает очередь `BlockingQueue` для хранения завершенных результатов. Класс `FutureTask` содержит метод `done`, который вызывается по завершении вычислений. Когда задача предоставлена, она обертывается в `QueueingFuture`, подкласс класса `FutureTask`, который переопределяет метод `done`, помещая результат в очередь `BlockingQueue`, как показано в листинге 6.14. Методы `take` и `poll` делегируют очереди `BlockingQueue`, блокируя продвижение, если результаты еще отсутствуют.

Листинг 6.14. Класс QueueingFuture, используемый службой ExecutorCompletionService

```
private class QueueingFuture<V> extends FutureTask<V> {
    QueueingFuture(Callable<V> c) { super(c); }
    QueueingFuture(Runnable t, V r) { super(t, r); }

    protected void done() {
        completionQueue.add(this);
    }
}
```

6.3.6. Пример: страничный отрисовщик со службой CompletionService

Служба `CompletionService` позволяет повысить производительность страничного отрисовщика двумя путями: более коротким суммарным временем работы и улучшенной отзывчивостью. Мы можем создать отдельную задачу для скачивания *каждого* изображения и его выполнения в пуле потоков, превратив последовательное скачивание в параллельное: это сократит время скачивания всех изображений. А путем доставки результатов из службы `CompletionService` и отрисовки каждого изображения, как только оно будет в наличии, мы дадим пользователю динамичный и отзывчивый пользовательский интерфейс. Эта реализация показана в `Renderer` в листинге 6.15.

Листинг 6.15. Использование службы CompletionService для отрисовки страничных элементов по мере их доступности

```
public class Renderer {
    private final ExecutorService executor;

    Renderer(ExecutorService executor) { this.executor = executor; }

    void renderPage(CharSequence source) {
        final List<ImageInfo> info = scanForImageInfo(source);
        CompletionService<ImageData> completionService =
            new ExecutorCompletionService<ImageData>(executor);
        for (final ImageInfo imageInfo : info)
            completionService.submit(new Callable<ImageData>() {
                public ImageData call() {
                    return imageInfo.downloadImage();
                }
            });

        renderText(source);

        try {
            for (int t = 0, n = info.size(); t < n; t++) {
                Future<ImageData> f = completionService.take();
                ImageData imageData = f.get();
                renderImage(imageData);
            }
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        } catch (ExecutionException e) {
            throw launderThrowable(e.getCause());
        }
    }
}
```

Многочисленные службы `ExecutorCompletionService` могут использовать один исполнитель `Executor` совместно, поэтому целесообразно создать службу `ExecutorCompletionService`, которая является приватной для конкретного вычисления. Во время такого использования служба `CompletionService` действует как дескриптор для пакета вычислений. Запоминая число задач, предоставленных в службу `CompletionService`, и подсчитывая число извлеченных завершенных результатов, можно узнать, когда все результаты для данного пакета были извлечены, даже если был применен совместный исполнитель `Executor`.

6.3.7. Наложение временны́х ограничений на задачи

Иногда, если действие не завершается в течение определенного времени, то его результат становится невостребованным. Например, веб-приложение может доставлять рекламные сообщения из внешнего сервера, но если они там отсутствуют в течение двух секунд, оно выводит рекламное сообщение, принятое по умолчанию, чтобы не снижать отзывчивость сайта.

Хронометрированная версия метода `Future.get` возвращается, как только результат готов, но выдает исключение `TimeoutException`, если результат не готов в пределах тайм-аута.

Важно, чтобы хронометрированные задачи вычисляли результат, который не будет использоваться. Для этого задачи управляются собственным бюджетом времени и прерываются, когда истекает время, либо отменяются, когда истекает тайм-аут. Если хронометрированный метод `get` завершается с исключением `TimeoutException`, то вы можете отменить задачу посредством объекта `Future`. Если задача написана как отменяемая (см. главу 7), то она может быть терминирована досрочно, для того чтобы не потреблять излишних ресурсов, как показано в листингах 6.13 и 6.16.

Типичное приложение хронометрируемого метода `Future.get` в листинге 6.16 генерирует составную веб-страницу с запрошенным содержимым и рекламой, доставленной из сервера. Оно предоставляет задачу доставки рекламы исполнителю, вычисляет остальную часть содержимого страницы, а затем ожидает рекламного сообщения, до тех пор пока не истечет его бюджет времени[1]. Если время ожидания метода `get` истечет, то задача до-

[1] Тайм-аут, передаваемый в метод `get`, вычисляется путем вычитания текущего времени из предельного срока, в результате чего может получиться отрицатель-

ставки рекламы отменяется[1], и вместо нее будет использовано рекламное сообщение, принятое по умолчанию.

6.3.8. Пример: портал бронирования поездок

Рассмотрим программу, где пользователь вводит даты и требования, а портал доставляет и выводит коммерческие предложения от ряда авиакомпаний, отелей и компаний по прокату автомобилей. В зависимости от компании доставка коммерческого предложения может быть связана с обращением к веб-службе, консультацией с базой данных, выполнением транзакции EDI или другим механизмом. Если поставщик не откликается вовремя, страница либо пропускает его полностью, либо выводит заполнитель, например «От Air Java ответ не получен вовремя».

Листинг 6.16. Получение рекламы с бюджетом времени

```
Page renderPageWithAd() throws InterruptedException {
    long endNanos = System.nanoTime() + TIME_BUDGET;
    Future<Ad> f = exec.submit(new FetchAdTask());
    // Отрисовать страницу в ожидании рекламы
    Page page = renderPageBody();
    Ad ad;
    try {
        // Ожидать только на протяжении оставшегося бюджета времени
        long timeLeft = endNanos - System.nanoTime();
        ad = f.get(timeLeft, NANOSECONDS);
    } catch (ExecutionException e) {
        ad = DEFAULT_AD;
    } catch (TimeoutException e) {
        ad = DEFAULT_AD;
        f.cancel(true);
    }
    page.setAd(ad);
    return page;
}
```

Доставка предложения одной компании не зависит от доставки предложения другой, и задачная граница обеспечивает конкурентность. Можно

ное число, но все синхронизированные методы в `java.util.concurrent` рассматривают отрицательные тайм-ауты как ноль, и дополнительный код не требуется.

[1] Параметр `true` в методе `Future.cancel` означает, что задачный поток может быть прерван, если задача в данный момент работает (см. главу 7).

создать *n* задач, предоставить их в пул потоков, сохранить объекты `Future` и использовать хронометрированный метод `get` для доставки каждого результата последовательно посредством его объекта `Future`, но есть более простой способ — метод `invokeAll`.

В листинге 6.17 используется хронометрированная версия метода `invokeAll` для предоставления многочисленных задач в службу `ExecutorService` и получения результатов. Метод принимает коллекцию задач и возвращает коллекцию объектов `Future`. Две коллекции имеют идентичные структуры; `invokeAll` добавляет объекты `Future` в возвращаемую коллекцию в порядке, установленном итератором коллекции задач, позволяя вызывающему элементу кода связать объект `Future` с объектом `Callable`, который он представляет. Хронометрированная версия метода `invokeAll` возвратится после завершения всех задач, прерывания вызывающего потока либо истечения тайм-аута. Все задачи, которые не были завершены по истечении тайм-аута, отменяются. По возвращении из метода `invokeAll` каждая задача будет либо завершена нормально, либо отменена. Клиентский код сможет вызывать метод `get` или метод `isCancelled` для того, чтобы выяснять, что произошло с задачей.

Листинг 6.17. Запрос цен на поездки в рамках бюджета времени

```
private class QuoteTask implements Callable<TravelQuote> {
    private final TravelCompany company;
    private final TravelInfo travelInfo;
    ...
    public TravelQuote call() throws Exception {
        return company.solicitQuote(travelInfo);
    }
}

public List<TravelQuote> getRankedTravelQuotes(
        TravelInfo travelInfo, Set<TravelCompany> companies,
        Comparator<TravelQuote> ranking, long time, TimeUnit unit)
        throws InterruptedException {
    List<QuoteTask> tasks = new ArrayList<QuoteTask>();
    for (TravelCompany company : companies)
        tasks.add(new QuoteTask(company, travelInfo));

    List<Future<TravelQuote>> futures =
        exec.invokeAll(tasks, time, unit);

    List<TravelQuote> quotes =
```

```
            new ArrayList<TravelQuote>(tasks.size());
        Iterator<QuoteTask> taskIter = tasks.iterator();
        for (Future<TravelQuote> f : futures) {
            QuoteTask task = taskIter.next();
            try {
                quotes.add(f.get());
            } catch (ExecutionException e) {
                quotes.add(task.getFailureQuote(e.getCause()));
            } catch (CancellationException e) {
                quotes.add(task.getTimeoutQuote(e));
            }
        }

        Collections.sort(quotes, ranking);
        return quotes;
    }
```

Итоги

Структурирование приложений вокруг выполнения *задач* упрощает разработку и способствует конкурентности. Структура `Executor` позволяет отстыковывать предоставление задач от их выполнения и поддерживает широкий спектр политик выполнения. Всякий раз, когда вы создаете потоки для выполнения задач, рассмотрите возможность использования исполнителя `Executor`. Для того чтобы извлечь максимальную пользу из разбиения приложения на задачи, необходимо определить разумные границы задач. В некоторых приложениях хорошо работают очевидные границы задач, в то время как в других может потребоваться более детальный анализ.

7 Отмена и выключение

Иногда нам требуется останавливать задачи или потоки раньше, чем они завершатся сами.

Java не предоставляет механизм для безопасной принудительной остановки работы потока[1], но обеспечивает *прерывание* (interruption) — кооперативный механизм, который позволяет одному потоку просить другой поток прекратить действовать.

Кооперативный подход необходим, поскольку мы редко хотим, чтобы задача, поток или служба останавливались *немедленно*, так как это может оставить совместные структуры данных в противоречивом состоянии. Мы будем кодировать задачи и службы таким образом, чтобы при запросе они очищали любую работу, которая в настоящий момент продолжается, а *затем* терминировали ее. Это обеспечит высокую гибкость, поскольку задачный код лучше способен применять очистку, чем код, запрашивающий отмену.

Вопросы конца жизненного цикла могут усложнить проектирование и реализацию задач, служб и приложений, поэтому столь важный элемент программирования часто игнорируется. В этой главе мы опишем

---

[1] Устаревшие методы `Thread.stop` и `suspend` с подобными функциями имеют серьезные недостатки, и их следует избегать. Проблемы этих методов описаны на странице http://java.sun.com/j2se/1.5.0/docs/guide/misc/threadPrimitiveDeprecation.html.

механизмы *отмены* (cancellation) и *прерывания* и способы кодирования задач и служб, откликающихся на запросы об отмене.

7.1. Отмена задачи

Действие доступно для отмены, если внешний код может направить его к остановке до момента его нормального завершения в следующих случаях:

Отмена по запросу пользователя. Пользователь нажал на кнопку «Отмена» в GUI-приложении или запросил отмену через интерфейс управления, такой как JMX (Java Management Extensions).

Ограниченное по времени действие. Приложение ведет поиск наилучшего решения в течение определенного времени. По истечении срока все задачи в процессе поиска отменяются.

События приложения. Приложение ищет лучшее решение, разбивая поиск на несколько задач, работающих в разных участках пространства. Когда одна задача находит решение, все остальные задачи отменяются.

Ошибки. Если задача веб-обходчика обнаруживает ошибку, другие задачи обхода отменяются, но остается запись их текущего состояния для перезапуска.

Выключение. При плавном выключении те задачи, которые в настоящий момент продолжаются, могут быть доведены до завершения. В случае немедленного выключения задачи, которые в настоящий момент выполняются, отменяются.

В Java нет других способов безопасно остановить задачу, кроме использования кооперативных механизмов и их непротиворечивых протоколов.

Одним из таких кооперативных механизмов является установка флажка «запрошена отмена», который задача периодически проверяет. Обнаружив, что флажок установлен, задача терминируется досрочно. Класс `PrimeGenerator` в листинге 7.1, в котором простые числа выводятся до тех пор, пока действие не будет отменено, иллюстрирует это техническое решение. Метод `cancel` устанавливает флажок `cancelled`, и главный цикл опрашивает этот флажок перед поиском следующего простого числа. (Флажок `cancelled` должен быть волатильным.)

Листинг 7.1. Использование волатильного поля для хранения состояния отмены

```
@ThreadSafe
public class PrimeGenerator implements Runnable {
    @GuardedBy("this")
    private final List<BigInteger> primes
            = new ArrayList<BigInteger>();
    private volatile boolean cancelled;

    public void run() {
        BigInteger p = BigInteger.ONE;
        while (!cancelled) {
            p = p.nextProbablePrime();
            synchronized (this) {
                primes.add(p);
            }
        }
    }

    public void cancel() { cancelled = true; }

    public synchronized List<BigInteger> get() {
        return new ArrayList<BigInteger>(primes);
    }
}
```

В листинге 7.2 показано, как генератор простых чисел может работать в течение одной секунды перед остановкой. Конечно, возникает задержка между временем запроса на отмену и временем следующей проверки флажка циклом `run`, но метод `cancel`, вызываемый из блока `finally`, все равно обеспечивает отмену генерирования простых чисел, даже если вызов метода `sleep` прерван. Без вызова метода `cancel` поток поиска простых чисел работал бы вечно, потребляя ресурсы и не позволяя JVM выйти.

Задача, задуманная как отменяемая, должна иметь *политику отмены*, отвечающую на вопросы: как другой код может запросить отмену, когда задача проверяет наличие запроса на отмену, какие действия задача предпринимает в ответ на запрос об отмене.

Порядок отмены платежа по чеку является реальным примером политики отмены.

Листинг 7.2. Генерирование простого числа каждую секунду

```
List<BigInteger> aSecondOfPrimes() throws InterruptedException {
    PrimeGenerator generator = new PrimeGenerator();
    new Thread(generator).start();
    try {
        SECONDS.sleep(1);
    } finally {
        generator.cancel();
    }
    return generator.get();
}
```

Класс `PrimeGenerator` использует простую политику отмены: клиентский код запрашивает отмену, вызывая метод `cancel`, а генератор `PrimeGenerator` проверяет наличие отмены один раз на одно найденное простое число и выходит, когда обнаруживает, что была запрошена отмена.

7.1.1. Прерывание

Если задача, которая использует описанный выше подход, вызовет блокирующий метод, такой как `BlockingQueue.put`, то она может ни разу не проверить флажок отмены и не терминироваться.

В классе `BrokenPrimeProducer` в листинге 7.3 производящий поток генерирует простые числа и помещает их в блокирующую очередь. Если производитель опережает потребителя, то очередь заполнится и метод `put` станет блокировать продвижение. Чтобы отменить задачу производителя, потребитель может вызвать метод `cancel`, который установит флажок `cancelled`. Но производитель ни разу не проверит этот флажок, потому что он никогда не выберется из блокирующего метода `put`.

В главе 5 мы намекнули, что некоторые блокирующие библиотечные методы поддерживают *прерывание*.

> В спецификации API или языка прерывание не привязано к какой-либо конкретной семантике отмены, но на практике использование прерывания не для отмены является хрупким и трудно сопроводимым механизмом.

Каждый поток имеет булев *статус прерванности* (interrupted status): `true` или `false`. В листинге 7.4 объект `Thread` содержит метод `interrupt`,

прерывающий целевой поток, и метод `isInterrupted`, возвращающий потоку статус прерванности. *Очистить* статус прерванности можно только статическим методом `interrupted`.

Блокирующие библиотечные методы, такие как `Thread.sleep` и `Object.wait`, быстро отзываются на прерывание, очищая статус прерванности и выдавая исключение `InterruptedException`.

Листинг 7.3. Ненадежная отмена, которая может оставить производителей застрявшими в блокирующей операции. *Так делать не следует*

```
class BrokenPrimeProducer extends Thread {
    private final BlockingQueue<BigInteger> queue;
    private volatile boolean cancelled = false;

    BrokenPrimeProducer(BlockingQueue<BigInteger> queue) {
        this.queue = queue;
    }

    public void run() {
        try {
            BigInteger p = BigInteger.ONE;
            while (!cancelled)
                queue.put(p = p.nextProbablePrime());
        } catch (InterruptedException consumed) { }
    }

    public void cancel() { cancelled = true; }
}

void consumePrimes() throws InterruptedException {
    BlockingQueue<BigInteger> primes = ...;
    BrokenPrimeProducer producer = new BrokenPrimeProducer(primes);
    producer.start();
    try {
        while (needMorePrimes())
            consume(primes.take());
    } finally {
        producer.cancel();
    }
}
```

Листинг 7.4. Методы прерывания в классе Thread

```
public class Thread {
    public void interrupt() { ... }
    public boolean isInterrupted() { ... }
    public static boolean interrupted() { ... }
    ...
}
```

Если прерываение не вызывает исключение `InterruptedException`, то статус прерванности `true` сохраняется до тех пор, пока кто-то намеренно его не очистит.

> Вызов метода `interrupt` не обязательно побуждает целевой поток прекратить действия — он только доставляет сообщение о том, что прерывание было запрошено.

Прерывание только *просит*, чтобы поток прервался при удобной возможности. (Такие возможности называются *точками отмены* — cancellation points.) Некоторые методы, такие как `wait`, `sleep` и `join`, воспринимают эти просьбы серьезно, выдавая исключение. Благополучные методы могут полностью игнорировать эти запросы и оставлять их вызывающему коду. Неблагополучные методы проглатывают запрос на прерывание, отказывая коду в возможности предпринять по нему какие-либо решения.

Статический метод `interrupted` следует использовать с осторожностью. Если вы вызываете метод `interrupted`, и он возвращает `true`, при условии что вы не планируете проглатывать прерывание, то вы должны что-то сделать — либо выдать исключение `InterruptedException`, либо восстановить статус прерванности, вызвав метод `interrupt` снова, как в листинге 5.10.

Класс `BrokenPrimeProducer` иллюстрирует, как пользовательские механизмы отмены не всегда хорошо взаимодействуют с блокирующими библиотечными методами.

> Прерывание обычно является наиболее разумным способом реализации отмены.

Класс `BrokenPrimeProducer` можно упростить, используя прерывание для запроса отмены вместо флажка, как показано в листинге 7.5. В каж-

дой итерации цикла есть две точки обнаружения прерывания: в вызове метода `put` и в заголовке цикла. Явная проверка здесь не является строго необходимой из-за вызова `put`, но она делает класс `PrimeProducer` более отзывчивым к прерыванию, так как он проверяет прерывание *перед* началом длительной задачи поиска простого числа, а не после нее.

Листинг 7.5. Использование прерывания для отмены

```
class PrimeProducer extends Thread {
    private final BlockingQueue<BigInteger> queue;

    PrimeProducer(BlockingQueue<BigInteger> queue) {
        this.queue = queue;
    }

    public void run() {
        try {
            BigInteger p = BigInteger.ONE;
            while (!Thread.currentThread().isInterrupted())
                queue.put(p = p.nextProbablePrime());
        } catch (InterruptedException consumed) {
            /* Разрешить потоку выйти */
        }
    }
    public void cancel() { interrupt(); }
}
```

7.1.2. Политики прерывания

Потокам необходима *политика прерывания*, определяющая, как они интерпретируют запрос на прерывание: что и с какой скоростью они делают при обнаружении запроса и какие единицы работы считают атомарными по отношению к прерыванию.

Наиболее разумной политикой является отмена на уровне потока или службы: максимально быстрый выход потока или службы, очистка при необходимости и уведомление владеющей сущности о выходе. Существует возможность установки других политик прерывания, таких как приостановка или возобновление службы, но поток или пулы потоков с нестандартными политиками прерывания могут быть ограничены задачами, осведомленными об этой политике.

Важно различать то, как *задачи* и *потоки* должны откликаться на прерывание. Одиночный запрос на прерывание может иметь более одного получателя — прерывание рабочего потока в пуле может означать отмену текущей задачи и выключение самого рабочего потока.

Задачи не выполняются в потоках, которыми они владеют, — они заимствуют потоки, принадлежащие службе, такой как пул потоков. Код, который не является владельцем потока (любой код вне реализации пула), должен следить за сохранением статуса прерванности, чтобы владеющий код мог предпринять по нему какие-либо действия. (Присматривая за чьим-то домом, вы не выбрасываете приходящую почту, пока хозяева в отъезде — вы ее сохраняете, даже если читаете чужие журналы.)

Вот почему большинство блокирующих библиотечных методов в ответ на прерывание выдают исключение `InterruptedException`. Они не выполняются в потоке, которым владеют, а сообщают о находке вызывающему элементу кода.

Задача не обязательно все отбрасывает, когда обнаруживает запрос на прерывание, — она может отложить его до более подходящего момента, плавно завершить начатое, а *затем* выдать исключение `InterruptedException` или иным образом указать на прерывание. Этот способ защищает структуры данных от повреждения.

Если задача не может распространить исключение `InterruptedException` на вызывающий ее элемент кода, то она должна восстановить статус прерванности после отлова исключения `InterruptedException`:

```
Thread.currentThread().interrupt();
```

Поток должен быть прерван только его владельцем, который может инкапсулировать осведомленность о политике прерывания потока в соответствующем механизме отмены, таком как метод выключения.

> Не прерывайте поток, если не знаете, как он интерпретирует прерывание.

Критики высмеяли механизм прерывания в Java, потому что он не допускает упреждающее прерывание и заставляет программистов обрабатывать исключение `InterruptedException`. Однако возможность отложить запрос на прерывание позволяет разработчикам создавать гибкие по-

литики прерывания, которые помогают сбалансировать в приложении отзывчивость и надежность.

7.1.3. Отклик на прерывание

Существуют две практические стратегии для обработки исключения `InterruptedException`:

- распространение исключения (возможно, после очистки, специфичной для задачи), после которого метод станет прерываемым и блокирующим;
- восстановление статуса прерванности, чтобы код выше в стеке вызовов мог с ним работать.

Распространение исключения `InterruptedException` сравнимо по простоте с его добавлением в спецификатор `throws`, как показано в листинге 7.6.

Листинг 7.6. Распространение исключения InterruptedException на вызывающие элементы кода

```
BlockingQueue<Task> queue;
...
public Task getNextTask() throws InterruptedException {
    return queue.take();
}
```

Если вы не хотите или не можете распространить исключение `InterruptedException` (возможно, потому, что ваша задача определяется интерфейсом `Runnable`), то сохраните запрос на прерывание через восстановление статуса прерванности путем повторного вызова метода `interrupt`. *Не* следует проглатывать исключение, если ваш код не реализует политику прерывания для потока. Класс `PrimeProducer` проглатывает прерывание, но лишь потому, что поток вот-вот терминируется и в стеке вызовов выше нет кода, который должен знать о прерывании.

> Проглатывание запроса на прерывание разрешено только коду, реализующему политику прерывания потока.

Действия, которые не поддерживают отмену, должны вызывать прерываемые блокирующие методы в цикле, повторяя попытки вызова при

обнаружении прерывания. Им следует сохранять статус прерванности локально и восстанавливать его непосредственно перед возвратом, как показано в листинге 7.7. Преждевременное установление статуса прерванности может привести к бесконечному циклу.

Если ваш код не вызывает прерываемые блокирующие методы, то он может отзываться на прерывание путем опроса статуса прерванности текущего потока в коде задачи. Вы можете назначить частоту опроса в зависимости от требований к эффективности и отзывчивости.

Отмена может быть связана с состоянием, отличным от статуса прерванности. Прерывание может использоваться для привлечения внимания потока, а информация, хранимая прерывающим потоком в другом месте, может потребоваться для предоставления дальнейших инструкций по работе с прерванным потоком. (Во время обращения к этой информации обязательно используйте синхронизацию.)

Листинг 7.7. Неотменяемая задача, восстанавливающая прерывание перед выходом

```
public Task getNextTask(BlockingQueue<Task> queue) {
    boolean interrupted = false;
    try {
        while (true) {
           try {
               return queue.take();
           } catch (InterruptedException e) {
                interrupted = true;
                // проскочить и попытаться снова
            }
        }
    } finally {
        if (interrupted)
            Thread.currentThread().interrupt();
    }
}
```

Когда рабочий поток, принадлежащий исполнителю `ThreadPoolExecutor`, обнаруживает прерывание, он проверяет, выключается ли пул в данный момент. Если это так, то он выполняет очистку пула перед его терминированием. В противном случае он может создать новый поток для восстановления пула до нужного размера.

7.1.4. Пример: хронометрированный прогон

Существуют задачи, которые можно решать вечно (например, перечисление всех простых чисел), поэтому нас интересует возможность ограничить обработку по времени.

Метод `aSecondOfPrimes` в листинге 7.2 запускает генератор простых чисел `PrimeGenerator` и прерывает его через одну секунду. Для остановки потребуется больше времени, чем секунда, генератор заметит прерывание и даст потоку терминироваться. Если `PrimeGenerator` создаст непроверяемое исключение до истечения тайм-аута, то оно, вероятно, останется незамеченным, так как генератор работает в отдельном потоке, не занимающемся исключениями явным образом.

В листинге 7.8 объект `Runnable` запускает задачу в вызывающем потоке и планирует вторую задачу, которая отменит первую после заданного интервала времени.

Данный пример нарушает правило: прежде чем прерывать поток, узнать его политику прерывания. Метод `timedRun` вызывается из произвольного потока, и если первая задача завершится до истечения тайм-аута, то задача отмены может начать действовать *после* возвращения метода к вызвавшему его элементу кода. Чтобы устранить риск нежелательного результата, нужно использовать объект `ScheduledFuture`, возвращаемый методом `schedule`, чтобы отменить задачу отмены.

Листинг 7.8. Планирование прерывания на заимствованном потоке. *Так делать не следует*

```
private static final ScheduledExecutorService cancelExec = ...;

public static void timedRun(Runnable r,
                            long timeout, TimeUnit unit) {
    final Thread taskThread = Thread.currentThread();
    cancelExec.schedule(new Runnable() {
        public void run() { taskThread.interrupt(); }
    }, timeout, unit);
    r.run();
}
```

Если задача не отзывается на прерывание, то метод `timedRun` не вернется до тех пор, пока она не завершится, что может произойти не скоро.

В листинге 7.9 после запуска задачного потока метод `timedRun` выполняет хронометрированный метод присоединения `join`[1] с только что созданным потоком. После своего возвращения метод `join` проверяет, выдала ли задача исключение, и если да, то повторно выдает его в потоке, вызывающем метод `timedRun`. Сохраненный объект `Throwable` используется двумя потоками, и поэтому объявляется волатильным с целью его безопасной публикации из задачного потока в поток метода `timedRun`.

Мы устранили проблемы, описанные в предыдущих примерах, но пока не знаем, возвращено ли управление и истек ли тайм-аут метода `join`[2].

7.1.5. Отмена с помощью Future

Мы уже использовали абстракцию для управления жизненным циклом задачи, занимающуюся исключениями и облегчающую отмену, — `Future`. Построим метод `timedRun` с использованием `Future` и структуры выполнения задач.

Листинг 7.9. Прерывание задачи в выделенном потоке

```
public static void timedRun(final Runnable r,
                            long timeout, TimeUnit unit)
                            throws InterruptedException {
    class RethrowableTask implements Runnable {
        private volatile Throwable t;
        public void run() {
            try { r.run(); }
            catch (Throwable t) { this.t = t; }
        }
```

продолжение ⇗

[1] *Присоединение* (`join`) к потоку означает ожидание блокирования его продвижения до момента завершения другого потока — *Примеч. пер.*

[2] Успешность или неуспешность завершения метода `join` имеет последствия для видимости памяти в модели памяти Java. Сам метод не возвращает статус, говорящий о его успешности или неуспешности.

Листинг 7.9 *(продолжение)*

```
        void rethrow() {
            if (t != null)
                throw launderThrowable(t);
        }
    }

    RethrowableTask task = new RethrowableTask();
    final Thread taskThread = new Thread(task);
    taskThread.start();
    cancelExec.schedule(new Runnable() {
        public void run() { taskThread.interrupt(); }
    }, timeout, unit);
    taskThread.join(unit.toMillis(timeout));
    task.rethrow();
}
```

Объект `Future` содержит метод `cancel`, который принимает булев аргумент `mayInterruptIfRunning` и возвращает значение, указывающее на успешность или неуспешность попытки отмены. (Оно сообщает о доставке прерывания, а не об обнаружении задачи или каких-то действиях по прерыванию.) Аргумент, равный `true`, сообщает о прерывании потока, в котором работает задача.

Когда можно вызывать метод `cancel` с аргументом `true`? Можно безопасно устанавливать аргумент `mayInterruptIfRunning` при отмене задач, работающих в стандартном исполнителе `Executor`. Не следует прерывать принадлежащий пулу поток непосредственно при попытке отменить задачу, так как вы не будете знать, какая задача работает во время доставки запроса на прерывание: делайте это только через объект `Future` задачи. Кодируя задачи так, чтобы они рассматривали прерывание как запрос на отмену, вы обеспечиваете возможность их отмены через `Future`.

В листинге 7.10 метод `timedRun` отправляет задачу в службу `ExecutorService` и извлекает результат с помощью хронометрированного метода `Future.get`. Если метод `get` завершается исключением `TimeoutException`, то задача отменяется через его объект `Future`. (Для упрощения метод `Future.cancel` вызывается в блоке `finally`.) Исключение, выданное до отмены, повторно выдается из метода `timedRun`, чтобы вызывающему элементу кода было удобнее с ним работать. В листинге 7.10 показана отмена задач, результат выполнения которых не востребован. (Данное техническое решение см. в листингах 6.13 и 6.16.)

Листинг 7.10. Отмена задачи с помощью Future

```
public static void timedRun(Runnable r,
                            long timeout, TimeUnit unit)
                            throws InterruptedException {
    Future<?> task = taskExec.submit(r);
    try {
        task.get(timeout, unit);
    } catch (TimeoutException e) {
        // задача будет отменена ниже
    } catch (ExecutionException e) {
        // исключение выдано в задаче; выдать повторно
        throw launderThrowable(e.getCause());
    } finally {
        // Безвредно, если задача уже завершена
        task.cancel(true); // прервать, если работает
    }
}
```

Когда метод `Future.get` выдает исключение `InterruptedException` (или `TimeoutException`), и вы знаете, что результат больше не нужен, то отмените задачу с помощью метода `Future.cancel`.

7.1.6. Работа с непрерываемым блокированием

Прерывание потока, заблокированного во время выполнения синхронного сокетного ввода-вывода или ожидания приобретения внутреннего замка, не состоится (будет только установлен статус прерванности потока). Чтобы убедить поток, блокированный в непрерываемом действии, остановиться с помощью средств, подобных прерыванию, нужно понимать механизм его блокирования.

Синхронный сокетный ввод-вывод в `java.io`. Распространенной формой блокирующего ввода-вывода в серверных приложениях является чтение из сокета или запись в сокет. К сожалению, методы чтения и записи в `InputStream` и `OutputStream` не отзываются на прерывание, но закрытие базового сокета побуждает любые потоки, блокированные в методах `read` или `write`, выдавать исключение `SocketException`.

Синхронный ввод-вывод в `java.nio`. Прерывание потока, ожидающего на канале `InterruptibleChannel`, побуждает все потоки, блокированные на этом канале, выдавать исключение `ClosedByInterruptException`

и закрывать канал. Закрытие канала `InterruptibleChannel` побуждает потоки, блокированные на канальных операциях, выдавать исключение `AsynchronousCloseException`.

Асинхронный ввод-вывод с помощью `Selector`. Если поток заблокирован в методе `Selector.select` (в `ava.nio.channels`), то метод `wakeup` побуждает его возвращаться досрочно с исключением `ClosedSelectorException`.

Приобретение замка. Нельзя остановить поток, заблокированный во время ожидания внутреннего замка. Но можно помочь ему приобрести замок и продвинуться. А явные замковые классы `Lock` предлагают метод `lockInterruptibly`, который позволяет ожидать замок и при этом отзываться на прерывания (см. главу 13).

Класс `ReaderThread` в листинге 7.11 показывает техническое решение для инкапсуляции нестандартной отмены. `ReaderThread` управляет отдельным сокетным соединением, синхронно читая данные из сокета и передавая их в буфер `processBuffer`. Для того чтобы облегчить терминацию пользовательского соединения или выключение сервера, класс `ReaderThread` переопределяет метод `interrupt` для доставки стандартного прерывания и закрытия базового сокета. Так поток `ReaderThread` прерывается, независимо от того, блокирован ли он в методе `read` или в непрерываемом блокирующем методе.

7.1.7. Инкапсуляция нестандартной отмены с помощью newTaskFor

Техническое решение, используемое в классе `ReaderThread`, для инкапсуляции нестандартной отмены, может быть уточнено с помощью перехватчика `newTaskFor`, добавленного в исполнитель `ThreadPoolExecutor` в Java 6. При отправке вызываемого объекта в службу `ExecutorService` метод `submit` возвращает объект `Future`, который можно использовать для отмены задачи. Перехватчик `newTaskFor` представляет собой фабричный метод, который создает объект `Future`, представляющий задачу. Он возвращает `RunnableFuture` — интерфейс, расширяющий `Future` и `Runnable`, который реализуется классом `FutureTask`.

Настройка `Future` позволяет переопределить метод `Future.cancel` для отмены задачи. Пользовательский код отмены может выполнять журналирование или сбор статистики по отмене, а также может использоваться для

отмены действий, которые не откликаются на прерывание. `ReaderThread` инкапсулирует отмену потоков, использующих сокеты, путем переопределения метода `interrupt`.

Класс `CancellableTask` в листинге 7.12 определяет интерфейс `CancellableTask`, который расширяет `Callable` и добавляет метод `cancel` и фабричный метод `newTask` для конструирования `RunnableFuture`. Класс `CancellingExecutor` расширяет исполнитель `ThreadPoolExecutor` и переопределяет метод `newTaskFor`, позволяя `CancellableTask` создавать свой объект `Future`.

Листинг 7.11. Сокрытие нестандартной отмены в потоке Thread путем переопределения метода interrupt

```
public class ReaderThread extends Thread {
    private final Socket socket;
    private final InputStream in;

    public ReaderThread(Socket socket) throws IOException {
        this.socket = socket;
        this.in = socket.getInputStream();
    }

    public void interrupt() {
        try {
            socket.close();
        }
        catch (IOException ignored) { }
        finally {
            super.interrupt();
        }
    }

    public void run() {
        try {
            byte[] buf = new byte[BUFSZ];
            while (true) {
                int count = in.read(buf);
                if (count < 0)
                    break;
                else if (count > 0)
                    processBuffer(buf, count);
            }
        } catch (IOException e) { /* Позволяет выход потока */ }
    }
}
```

Класс `SocketUsingTask` реализует интерфейс `CancellableTask` и определяет метод `Future.cancel`, для того чтобы закрыть сокет, а также вызвать метод `super.cancel`. Если `SocketUsingTask` отменяется через объект `Future`, то сокет закрывается, а выполняющий поток прерывается. Это улучшает отзывчивость задачи на отмену и позволяет потоку безопасно вызывать различные блокирующие методы.

7.2. Остановка поточной службы

Приложения обычно создают службы, которые владеют потоками, такие как пулы потоков, и время существования этих служб обычно превышает время существования метода, который их создает. Если приложение планируется выключить плавно, то потоки, находящиеся во владении этих служб, должны быть терминированы: их нужно убедить выключиться самостоятельно.

Листинг 7.12. Инкапсуляция нестандартной отмены в задаче с помощью метода newTaskFor

```
public interface CancellableTask<T> extends Callable<T> {
    void cancel();
    RunnableFuture<T> newTask();
}

@ThreadSafe
public class CancellingExecutor extends ThreadPoolExecutor {
    ...
    protected<T> RunnableFuture<T> newTaskFor(Callable<T> callable) {
    if (callable instanceof CancellableTask)
        return ((CancellableTask<T>) callable).newTask();
    else
        return super.newTaskFor(callable);
    }
}

public abstract class SocketUsingTask<T>
        implements CancellableTask<T> {
    @GuardedBy("this") private Socket socket;

    protected synchronized void setSocket(Socket s) { socket = s; }

    public synchronized void cancel() {
        try {
```

```
            if (socket != null)
                socket.close();
        } catch (IOException ignored) { }
    }

    public RunnableFuture<T> newTask() {
        return new FutureTask<T>(this) {
            public boolean cancel(boolean mayInterruptIfRunning) {
                try {
                    SocketUsingTask.this.cancel();
                } finally {
                    return super.cancel(mayInterruptIfRunning);
                }
            }
        };
    }
}
```

API потока не содержит формального понятия о владении: поток представлен объектом `Thread`, который может использоваться совместно, как любой другой объект. Однако фактически потоки имеют владельца, и обычно он представляет собой класс, который их создал. Пул потоков владеет своими рабочими потоками и должен при необходимости позаботиться об их остановке.

Приложение может владеть службой, а служба — рабочими потоками, но приложение не владеет рабочими потоками и не должно пытаться останавливать их напрямую. Вместо этого служба должна предоставлять *методы жизненного цикла* для самовыключения, которые также выключат принадлежащие ей потоки. Например, служба `ExecutorService` предоставляет методы `shutdown` и `shutdownNow`.

> Предоставляйте методы жизненного цикла, если владеющая потоком служба должна жить дольше метода, который ее создал.

7.2.1. Пример: служба журналирования

Большинство серверных приложений используют журналирование. Оно может быть не более чем простой вставкой инструкций `println` в исходный код. Классы, такие как `PrintWriter`, являются потокобезопасными,

поэтому журналирование в них не требует явной синхронизации[1]. Однако, как мы увидим в разделе 11.6, встроенное в код журналирование имеет некоторую стоимость для производительности. Вызов метода `log` помещает журнальное сообщение в очередь для обработки другим потоком.

Класс `LogWriter` в листинге 7.13 показывает отдельный журнальный поток, который получает сообщение по эстафете от источника через очередь `BlockingQueue`, а затем пишет его. Это шаблон с многочисленными производителями и единственным потребителем, где очередь блокирует источники, если журнальный поток отстает.

Листинг 7.13. Служба журналирования «производитель-потребитель» без поддержки выключения

```
public class LogWriter {
    private final BlockingQueue<String> queue;
    private final LoggerThread logger;

    public LogWriter(Writer writer) {
        this.queue = new LinkedBlockingQueue<String>(CAPACITY);
        this.logger = new LoggerThread(writer);
    }

    public void start() { logger.start(); }

    public void log(String msg) throws InterruptedException {
        queue.put(msg);
    }

    private class LoggerThread extends Thread {
        private final PrintWriter writer;
        ...
        public void run() {
            try {
                while (true)
                    writer.println(queue.take());
```

[1] Если два потока вносят многострочные стековые трассировки в один и тот же стандартный поток одним вызовом инструкции `println` на строку, то может потребоваться дополнительная блокировка на стороне клиента, чтобы результаты не перемежались.

```
            } catch(InterruptedException ignored) {
            } finally {
                writer.close();
            }
        }
    }
}
```

Чтобы служба `LogWriter` не препятствовала нормальному выключению JVM, журнальный поток нужно уметь терминировать (через метод `take`, который отзывается на прерывание). Если журнальный поток способен выходить при отлавливании исключения `InterruptedException`, то его прерывание будет останавливать службу.

Помните, что внезапное выключение выбрасывает журнальные сообщения, ожидающие запись, а заблокированные в методе `log` потоки *никогда не будут разблокированы*. Отмена в шаблоне подразумевает отмену как производителей, так и потребителей, но первых отменить сложнее, так как они не являются выделенными потоками.

В листинге 7.14 для выключения писателя `LogWriter` установлен флажок «запрошено выключение», позволяющий прекратить поступление новых сообщений. Такой подход содержит состояния гонки. Реализация метода `log` представляет собой последовательность «проверить и затем действовать»: производители видят, что служба еще не выключена, и размещают сообщения в очереди после ее выключения с риском стать заблокированными навсегда. Если потребитель подождет несколько секунд, прежде чем объявит очередь пустой, вероятность сбоя уменьшится, но проблема не исчезнет.

Листинг 7.14. Ненадежный способ добавления поддержки выключения в службу журналирования

```
public void log(String msg) throws InterruptedException {
    if (!shutdownRequested)
        queue.put(msg);
    else
        throw new IllegalStateException("диспетчер журналирования
                                                        выключен");
}
```

Чтобы избежать состояния гонки, нужно сделать поступление нового сообщения атомарным. Наличие замка заставит метод `put` блокировать продвижение, поэтому мы будем «резервировать» право отправлять сообщение с помощью условного приращения счетчика, атомарно проверяя выключение, как показано в листинге 7.15.

Листинг 7.15. Добавление надежной отмены в LogWriter

```
public class LogService {
    private final BlockingQueue<String> queue;
    private final LoggerThread loggerThread;
    private final PrintWriter writer;
    @GuardedBy("this") private boolean isShutdown;
    @GuardedBy("this") private int reservations;

    public void start() { loggerThread.start(); }

    public void stop() {
        synchronized (this) { isShutdown = true; }
        loggerThread.interrupt();
    }

    public void log(String msg) throws InterruptedException {
        synchronized (this) {
            if (isShutdown)
                throw new IllegalStateException(...);
            ++reservations;
        }
        queue.put(msg);
    }

    private class LoggerThread extends Thread {
        public void run() {
            try {
                while (true) {
                    try {
                        synchronized (this) {
                            if (isShutdown && reservations == 0)
                                break;
                        }
                        String msg = queue.take();
                        synchronized (this) { --reservations; }
                        writer.println(msg);
                    } catch (InterruptedException e) { /* повторная
                                                         попытка */ }
                }
```

```
            } finally {
                writer.close();
            }
        }
    }
}
```

7.2.2. Выключение службы ExecutorService

В разделе 6.2.4 мы показали два способа выключения: плавное — с помощью метода `shutdown` и внезапное — с помощью метода `shutdownNow`, который возвращает список задач, еще не запущенных после попытки отмены выполняемых задач.

Определите приоритет: безопасность и откликаемость. Внезапная терминация быстрее, но рискованнее, а нормальная терминация медленнее, но безопаснее. Рассмотрите возможность предоставления выбора режимов выключения.

В простых программах безвреден запуск выключения глобальной службы `ExecutorService` из `main`. Более сложные программы, скорее всего, будут инкапсулировать службу `ExecutorService` на фоне более высокоуровневой службы, которая предоставляет свои методы жизненного цикла, такие как вариант службы `LogService` в листинге 7.16, который делегирует управление в службу `ExecutorService` вместо управления своими потоками. Инкапсулирование службы `ExecutorService` удлиняет цепочку владения путем добавления еще одной ссылки.

Листинг 7.16. Служба журналирования, использующая ExecutorService

```
public class LogService {
    private final ExecutorService exec = newSingleThreadExecutor();
    ...
    public void start() { }

    public void stop() throws InterruptedException {
        try {
            exec.shutdown();
            exec.awaitTermination(TIMEOUT, UNIT);
        } finally {
            writer.close();
        }
    }
```

продолжение ⇩

Листинг 7.16 *(продолжение)*

```
    public void log(String msg) {
        try {
            exec.execute(new WriteTask(msg));
        } catch (RejectedExecutionException ignored) { }
    }
}
```

7.2.3. Ядовитые таблетки

Еще один способ убедить службу «производитель-потребитель» выключиться представлен *ядовитой таблеткой* (poison pill) — узнаваемым объектом очереди, получатель которого останавливается. При FIFO ядовитая таблетка заставляет потребителей закончить начатое, так как производители не смогут предоставлять работу после ее помещения в очередь. Служба `IndexingService` в листингах 7.17, 7.18 и 7.19 показывает поиск на рабочем столе из листинга 5.8 на с. 134, в котором для выключения службы использована ядовитая таблетка.

Листинг 7.17. Выключение с ядовитой таблеткой

```
public class IndexingService {
    private static final File POISON = new File("");
    private final IndexerThread consumer = new IndexerThread();
    private final CrawlerThread producer = new CrawlerThread();
    private final BlockingQueue<File> queue;
    private final FileFilter fileFilter;
    private final File root;

    class CrawlerThread extends Thread { /* Листинг 7.18 */ }
    class IndexerThread extends Thread { /* Листинг 7.19 */ }

    public void start() {
        producer.start();
        consumer.start();
    }

    public void stop() { producer.interrupt(); }

    public void awaitTermination() throws InterruptedException {
        consumer.join();
    }
}
```

Ядовитые таблетки надежно работают с известным числом производителей и потребителей и неограниченными очередями. В службе `IndexingService`

можно увеличить число производителей и потребителей, размещая достаточное число таблеток.

Листинг 7.18. Поток производителя для службы IndexingService

```
public class CrawlerThread extends Thread {
    public void run() {
        try {
            crawl(root);
        } catch (InterruptedException e) { /* проскочить */ }
        finally {
            while (true) {
                try {
                    queue.put(POISON);
                    break;
                } catch (InterruptedException e1) { /* попытаться снова */ }
            }
        }
    }

    private void crawl(File root) throws InterruptedException {
        ...
    }
}
```

Листинг 7.19. Поток потребителя для службы IndexingService

```
public class IndexerThread extends Thread {
    public void run() {
        try {
            while (true) {
                File file = queue.take();
                if (file == POISON)
                    break;
                else
                    indexFile(file);
            }
        } catch (InterruptedException consumed) { }
    }
}
```

7.2.4. Пример: служба однократного выполнения

Если метод обязан обработать пакет задач, и он не возвращается до их завершения, то он может упростить управление жизненным циклом службы с помощью приватного исполнителя `Executor`, срок службы которого им ограничен. (Например, методы `invokeAll` и `invokeAny`.)

Метод `checkMail` в листинге 7.20 проверяет наличие новой почты параллельно на нескольких хостах. Он создает приватного исполнителя и предоставляет задачу для каждого хоста, после чего выключает исполнителя и ожидает терминации, которая происходит, когда все задачи проверки почты завершены[1].

Листинг 7.20. Использование приватного исполнителя Executor, время жизни которого ограничено вызовом метода

```
boolean checkMail(Set<String> hosts, long timeout, TimeUnit unit)
        throws InterruptedException {
    ExecutorService exec = Executors.newCachedThreadPool();
    final AtomicBoolean hasNewMail = new AtomicBoolean(false);
    try {
        for (final String host : hosts)
            exec.execute(new Runnable() {
                public void run() {
                    if (checkMail(host))
                        hasNewMail.set(true);
                }
            });
    } finally {
        exec.shutdown();
        exec.awaitTermination(timeout, unit);
    }
    return hasNewMail.get();
}
```

7.2.5. Ограничения метода shutdownNow

При внезапном выключении с помощью метода `shutdownNow` служба `ExecutorService` отменяет выполняемые задачи и возвращает список незапущенных задач, чтобы записать их в журнал или сохранить для последующей обработки[2].

Но нет способа узнать о состоянии задач, находившихся в процессе работы во время выключения, если сами задачи не занимаются установкой своего рода

---

[1] `AtomicBoolean` используется вместо `volatile boolean`, потому что для доступности из внутреннего `Runnable` флажок `hasNewMail` должен быть финальным, что исключает его модификацию.

[2] Объекты `Runnable`, возвращаемые методом `shutdownNow`, могут отличаться от объектов, которые были предоставлены в службу `ExecutorService`, и могут быть *обернутыми* экземплярами предоставленных задач.

контрольных точек. Для того чтобы узнать, какие задачи не были завершены, вам нужно знать, не только какие задачи не были запущены, но и какие задачи находились в процессе работы, когда исполнитель быть выключен[1].

Класс `TrackingExecutor` в листинге 7.21 показывает техническое решение для нахождения задач, выполняемых во время выключения. Инкапсулируя службу `ExecutorService` и запоминая методом `execute` (или `submit`) задачи, отмененные после выключения, исполнитель `TrackingExecutor` идентифицирует, какие задачи были запущены, но не завершались корректно. После завершения работы исполнителя метод `getCancelledTasks` возвращает список отмененных задач. Задачи должны сохранять статус прерванности потока при возвращении.

Листинг 7.21. Служба ExecutorService, отслеживающая отмененные задачи после выключения

```
public class TrackingExecutor extends AbstractExecutorService {
    private final ExecutorService exec;
    private final Set<Runnable> tasksCancelledAtShutdown =
        Collections.synchronizedSet(new HashSet<Runnable>());
    ...
    public List<Runnable> getCancelledTasks() {
        if (!exec.isTerminated())
            throw new IllegalStateException(...);
        return new ArrayList<Runnable>(tasksCancelledAtShutdown);
    }

    public void execute(final Runnable runnable) {
        exec.execute(new Runnable() {
            public void run() {
                try {
                    runnable.run();
                } finally {
                    if (isShutdown()
                        && Thread.currentThread().isInterrupted())
                            tasksCancelledAtShutdown.add(runnable);
                }
            }
        });
    }
    // делегировать другие методы службы ExecutorService исполнителю
}
```

[1] К сожалению, отсутствует вариант выключения, в котором еще не запущенные задачи возвращаются вызывающему элементу кода, а задачам, находящимся в процессе работы, разрешено завершиться.

Класс `WebCrawler` в листинге 7.22 показывает применение службы `TrackingExecutor`. Работа веб-обходчика часто не ограничена, и нам удобно сохранять его состояние для перезапуска. `CrawlTask` предоставляет метод `getPage`, который идентифицирует текущую страницу. При выключении обходчика сканируются незапущенные и отмененные задачи и записываются их URL-адреса, которые можно добавить в очередь.

Исполнитель `TrackingExecutor` содержит состояние гонки, которое допускает идентификацию завешенных задач как отмененных. Пул потоков может быть выключен между моментом выполнения последней инструкции задачи и моментом записи завершения задачи. Задачи обходчика *идемпотентны* (их двукратное выполнение имеет тот же эффект, что и однократное). В других случаях будьте готовы к работе с ложными утверждениями.

Листинг 7.22. Использование TrackingExecutorService для сохранения незаконченных задач для последующего выполнения

```
public abstract class WebCrawler {
    private volatile TrackingExecutor exec;
    @GuardedBy("this")
    private final Set<URL> urlsToCrawl = new HashSet<URL>();
    ...
    public synchronized void start() {
        exec = new TrackingExecutor(
                Executors.newCachedThreadPool());
        for (URL url : urlsToCrawl) submitCrawlTask(url);
        urlsToCrawl.clear();
    }

    public synchronized void stop() throws InterruptedException {
        try {
            saveUncrawled(exec.shutdownNow());
            if (exec.awaitTermination(TIMEOUT, UNIT))
                saveUncrawled(exec.getCancelledTasks());
        } finally {
            exec = null;
        }
    }

    protected abstract List<URL> processPage(URL url);

    private void saveUncrawled(List<Runnable> uncrawled) {
        for (Runnable task : uncrawled)
            urlsToCrawl.add(((CrawlTask) task).getPage());
    }
```

```
    private void submitCrawlTask(URL u) {
        exec.execute(new CrawlTask(u));
    }
    private class CrawlTask implements Runnable {
        private final URL url;
        ...
        public void run() {
            for (URL link : processPage(url)) {
                if (Thread.currentThread().isInterrupted())
                    return;
                submitCrawlTask(link);
            }
        }
        public URL getPage() { return url; }
    }
}
```

7.3. Обработка аномальной терминации потоков

Когда однопоточное консольное приложение терминируется из-за неотловленного исключения, программа прекращает работу и создает нетипичную стековую трассировку. В конкурентном приложении сбой потока не всегда так очевиден. Стековая трассировка может быть напечатана на консоли и остаться незамеченной. К счастью, существуют средства обнаружения и предотвращения сбоя потоков.

При появлении исключения `RuntimeException`, которое распространяется вверх по стеку, распечатайте стековую трассировку на консоли и дайте потоку терминироваться.

Потеря потока из пула может иметь последствия для производительности, но не помешает работе приложения. Намного заметнее потеря потока диспетчеризации событий в GUI-приложении: она ведет к остановке обработки событий и замерзанию GUI-интерфейса. В классе `OutOfTime` на с. 168 показано, как вышла из строя служба, представленная объектом `Timer`.

Почти любой код может выдавать исключение `RuntimeException`. Чем меньше вы знакомы с вызываемым кодом, тем внимательнее отнеситесь к его поведению.

Потоки обработки задач, такие как рабочие потоки в пуле или поток диспетчеризации событий `Swing`, проводят всю жизнь, вызывая неизвестный

код через абстрактный барьер, такой как `Runnable`. Они должны вызывать задачи в блоке `try-catch`, который отлавливает непроверяемые исключения, либо в блоке `try-finally`, информирующем струкутру об аномальном выходе потока. Отлов исключения `RuntimeException` уместен, когда вы вызываете ненадежный код с помощью абстракции, такой как `Runnable`[1].

Листинг 7.23 показывает способ создания рабочего потока в пуле. Если задача создает непроверяемое исключение, то она дает потоку умереть, но не раньше, чем уведомит структуру об этом событии. Затем структура может заменить поток новым при необходимости. Исполнитель `ThreadPoolExecutor` и платформа `Swing` используют это техническое решение, чтобы неблагополучная задача не препятствовала выполнению последующих задач. Если вы пишете класс рабочего потока, который выполняет предоставленные задачи, или вызываете ненадежный внешний код (например, динамически загружаемые подключаемые модули), используйте один из этих подходов, для того чтобы не дать плохо написанной задаче или подключаемому модулю уничтожить поток, которому случится его вызвать.

Листинг 7.23. Типичная структура рабочего потока пула потоков

```
public void run() {
    Throwable thrown = null;
    try {
        while (!isInterrupted())
            runTask(getTaskFromWorkQueue());
    } catch (Throwable e) {
        thrown = e;
    } finally {
        threadExited(this, thrown);
    }
}
```

7.3.1. Обработчики неотловленных исключений

Обработчик `UncaughtExceptionHandler` позволяет обнаруживать сбой потока из-за неотловленного исключения и дополняет описанный выше подход.

---

[1] Существуют разногласия по поводу безопасности этого технического решения: когда поток выдает непроверяемое исключение, под угрозу может быть поставлено все приложение в целом. Но выключение всего приложения непрактично.

Когда поток выходит из-за неотловленного исключения, JVM сообщает об этом событии обработчику `UncaughtExceptionHandler`, поставляемому приложением (см. листинг 7.24). Если обработчика не существует, то принятым по умолчанию поведением будет печать стековой трассировки в `System.err`[1].

Листинг 7.24. Интерфейс UncaughtExceptionHandler

```
public interface UncaughtExceptionHandler {
    void uncaughtException(Thread t, Throwable e);
}
```

Действия обработчика в отношении неотловленных исключений определяются заранее. Например, в листинге 7.25 показана запись сообщения об ошибке и стековой трассировке в журнал приложения. Также обработчики могут выполнять более прямые действия: перезапустить поток, выключить приложение или пролистать оператор.

Листинг 7.25. Обработчик UncaughtExceptionHandler, регистрирующий исключение в журнале

```
public class UEHLogger implements Thread.UncaughtExceptionHandler {
    public void uncaughtException(Thread t, Throwable e) {
        Logger logger = Logger.getAnonymousLogger();
        logger.log(Level.SEVERE,
             "Поток терминирован с исключением: " + t.getName(),
             e);
    }
}
```

> Для обработки длительных задач всегда используйте обработчики неотловленных исключений для всех потоков, которые могут регистрировать исключение в журнале.

Чтобы задать обработчик `UncaughtExceptionHandler` для потоков пула, предоставьте фабрику `ThreadFactory` конструктору исполнителя `Thread-`

[1] Можно настроить обработчик `UncaughtExceptionHandler` на основе соответствующего потока или установить его по умолчанию. Реализация используемого по умолчанию обработчика в `ThreadGroup` делегирует свою родительскую поточную группу вверх по цепочке, до тех пор пока один из обработчиков `ThreadGroup` не разберется с неотловленным исключением или не выведет стековую трассировку на консоль.

PoolExecutor. (Только владелец потока должен изменять обработчик UncaughtExceptionHandler.) Стандартные пулы потоков позволяют неотловленному задачному исключению терминировать поток пула, используя блок try-finally для создания уведомления о времени терминации, необходимого для замены потока. Для получения достоверной информации о сбое оберните задачу с помощью объекта Runnable или Callable, отлавливающего исключение, либо переопределите перехватчик afterExecute в исполнителе ThreadPoolExecutor.

Если задача, предоставленная с помощью submit, терминируется с исключением, то оно выдается повторно методом Future.get в обертке исключения ExecutionException.

7.4. Выключение JVM

JVM может выключаться *упорядоченно* или *внезапно*. Упорядоченное выключение инициируется, когда терминируется последний не являющийся демоном поток, с помощью вызова System.exit или других средств (например, отправки SIGINT или нажатия Ctrl-C). Хотя этот способ выключения JVM является стандартным и предпочтительным, машина также может быть выключена внезапно путем вызова метода Runtime.halt или отправки SIGKILL.

7.4.1. Хуки

В упорядоченном выключении JVM-машина сначала запускает *хуки* (shutdown hooks) — незапущенные потоки, которые зарегистрированы с помощью метода Runtime.addShutdownHook. JVM не дает гарантий относительно порядка их запуска. Потоки приложения (являющиеся либо не являющиеся демонами) могут продолжать работать конкурентно с процессом выключения. Когда все хуки завершены, JVM может выполнить финализаторы, если флажок runFinalizersOnExit равен true, и затем остановиться, заставляя потоки терминироваться внезапно. Если хуки или финализаторы не завершаются, то упорядоченный процесс выключения зависает и JVM необходимо выключить внезапно (в этом случае хуки работать не будут).

Хуки должны быть потокобезопасными — использовать синхронизацию при доступе к совместным данным, избегать взаимной блокировки, не

делать допущений о состоянии приложения и выходить быстро, так как их работа задерживает терминацию JVM.

Хуки можно использовать для очистки служб или приложений, например для удаления временных файлов или очистки ресурсов, которые не удаляются операционной системой автоматически. В листинге 7.26 показано, как `LogService` в листинге 7.16 может зарегистрировать хук из метода `start`, обеспечив закрытие журнального файла на выходе.

Поскольку хуки работают конкурентно, закрытие журнального файла может вызвать проблемы совместного использования диспетчера журналирования. Хуки не должны опираться на службы, которые могут быть выключены приложением или другим хуком. Поэтому лучше использовать один хук для всех служб, который будет вызывать серию действий по выключению последовательно в одном потоке, без состояний гонки и взаимной блокировки. В приложениях с явными сведениями о зависимостях между службами это техническое решение поддержит правильный порядок действий по выключению.

Листинг 7.26. Регистрация хука для остановки службы журналирования

```
public void start() {
    Runtime.getRuntime().addShutdownHook(new Thread() {
        public void run() {
            try { LogService.this.stop(); }
            catch (InterruptedException ignored) {}
        }
    });
}
```

7.4.2. Потоки-демоны

Поток-демон (daemon thread) является вспомогательным потоком, который не мешает выключению JVM.

Потоки делятся на два типа: обычные и демоны. При запуске JVM все создаваемые ею потоки (например, сборщик мусора и другие служебные потоки) являются демонами, за исключением главного потока. Новый поток наследует демон-статус создавшего его потока, поэтому потоки, созданные главным потоком, являются обычными.

Обычные потоки и демоны отличаются только последствиями своего выхода. Когда поток выходит, JVM выполняет инвентаризацию запу-

щенных потоков. Если она фиксирует, что остались только демоны, она инициирует упорядоченное выключение. Когда JVM останавливается, все оставшиеся потоки-демоны покидаются — блоки `finally` не выполняются, стеки не разматываются, — JVM просто выходит.

Потоки-демоны следует использовать осторожно — существует немного обрабатывающих операций, которые можно безопасно покинуть без очистки в любое время. В частности, опасно использовать потоки-демоны для задач, которые могут выполнять любой вид ввода-вывода. Однако они подходят для служебных задач, таких как фоновое удаление устаревших записей из кэша в памяти.

Потоки-демоны не подходят для управления жизненным циклом служб внутри приложения.

7.4.3. Финализаторы

Сборщик мусора хорошо справляется с высвобождением ресурсов памяти, но некоторые ресурсы, такие как файловые или сокетные дескрипторы, должны возвращаться операционной системе. Поэтому объекты, содержащие метод `finalize`, после своего высвобождения могут высвобождать и персистентные ресурсы.

Поскольку финализаторы могут работать в потоке, управляемом машиной JVM, то к любому состоянию, к которому они обращаются, будет обращаться более чем один поток и, следовательно, потребуется синхронизация. Финализаторы вносят значительную стоимость для производительности, и их чрезвычайно трудно написать правильно[1]. Сочетание блоков `finally` и явных методов `close` справляется с управлением ресурсами лучше, чем финализаторы, если речь идет не об управлении объектами, содержащими ресурсы, приобретаемые нативными методами. По этим и другим причинам старайтесь избегать написания или использования классов с финализаторами (кроме платформенных библиотечных классов) [EJ пункт 6].

Избегайте финализаторов.

[1] О проблемах, связанных с написанием финализаторов, есть более подробная информация (Boehm, 2005).

Итоги

Вопросы конца жизненного цикла задач, потоков, служб и приложений могут усложнить их проектирование и реализацию. Язык Java не предоставляет упреждающий механизм для отмены действий или терминации потоков. Вместо этого он предоставляет кооперативный механизм прерывания, который может быть использован для облегчения отмены, но конструирование протоколов отмены и их непротиворечивое применение будут зависеть от вас. Использование `FutureTask` и структуры `Executor` упрощает построение отменяемых задач и служб.

8 Применение пулов потоков

В главе 7 были рассмотрены некоторые проблемы жизненного цикла службы, возникающие при использовании структуры выполнения задач в реальных приложениях. В этой главе мы опишем дополнительные параметры пула потоков и исполнителя `Executor`, а также сбои при использовании структуры выполнения задач.

8.1. Неявные стыковки между задачами и политиками выполнения

Ранее мы утверждали, что структура `Executor` отстыковывает предоставление задачи от ее выполнения. Добавим, что существуют типы задач, совместимые только с определенными политиками выполнения:

Зависимые задачи. Наиболее благополучные задачи *не зависимы* от присутствия других задач. При их выполнении в пуле вы можете свободно варьировать размер и конфигурацию пула (пострадает только производительность). Предоставляя в пул потоков задачи, зависящие от других задач, вы неявно создаете ограничения в политике выполнения, которыми необходимо управлять, чтобы избежать проблем жизнеспособности (см. раздел 8.1.1).

Задачи, которые задействуют ограничение одним потоком. Задачи, созданные для работы в одном потоке, требуют, чтобы их исполнитель был однопоточным[1].

Задачи, чувствительные ко времени отклика. Предоставление длительной задачи однопоточному исполнителю либо нескольких длительных задач — пулу с малым числом потоков может ухудшить отзывчивость службы, управляемой этим исполнителем.

Задачи, которые используют класс `ThreadLocal`. Класс `ThreadLocal` позволяет каждому потоку иметь свою версию переменной. Его разумно использовать, если локальное для потока значение имеет жизненный цикл, ограниченный задачей. `ThreadLocal` не должен использоваться в потоках пула для обмена значениями между задачами.

Пулы потоков лучше всего работают, когда задачи *однородны* и *независимы*. Смешивание длительных и кратковременных задач засорит небольшой пул, а предоставление задач, зависящих от других задач, может привести к взаимной блокировке, если пул ограничен. К счастью, запросы в типичных сетевых серверных приложениях — веб-, почтовых или файловых серверах — соответствуют этим рекомендациям.

Документируйте требования задач к политике выполнения.

8.1.1. Взаимная блокировка с ресурсным голоданием

Если в пуле выполняются задачи, зависящие от других задач, то они могут быть заперты взаимной блокировкой. В однопоточном исполнителе задача, которая предоставляет тому же исполнителю еще одну задачу и ожидает ее результат, всегда будет запираться взаимной блокировкой. Вторая задача просидит в рабочей очереди, до тех пор пока не завершится первая задача, которая ожидает результат второй задачи. То же самое может произойти в больших пулах, если все потоки выполняют задачи, заблокированные в ожидании других задач в рабочей очереди. *Взаимная*

[1] Это требование является не совсем строгим: достаточно обеспечить, чтобы задачи не выполнялись конкурентно, и предоставить синхронизацию, чтобы эффекты памяти одной задачи были видны следующей. Такая гарантия предлагается исполнителем `newSingleThreadExecutor`.

блокировка с ресурсным голоданием (thread starvation deadlock) может возникать всякий раз, когда задача пула инициирует неограниченное блокирующее ожидание ресурса или условия, которое может совершиться только после действия другой задачи пула.

В листинге 8.1 `RenderPageTask` предоставляет две дополнительные задачи исполнителю по доставке верхнего и нижнего колонтитулов, отрисовывает тело страницы, ожидает результатов задач по доставке колонтитулов, а затем объединяет верхний колонтитул, тело и нижний колонтитул на готовой странице. С однопоточным исполнителем класс `ThreadDeadlock` всегда будет запираться взаимной блокировкой. Задачи, которые координируются между собой с помощью барьера, могут тоже вызвать взаимную блокировку с ресурсным голоданием, если пул недостаточно велик.

> Всякий раз, когда вы предоставляете исполнителю `Executor` зависимые задачи, документируйте в коде или конфигурационном файле для `Executor` любые лимиты на размер пула или конфигурацию.

Помимо явных лимитов на размер пула, существуют неявные ограничения на другие ресурсы. Если ваше приложение использует пул соединений JDBC с десятью соединениями и каждой задаче требуется подключение к базе данных, то пул потоков должен иметь только десять потоков.

Листинг 8.1. Задача, запираемая взаимной блокировкой в однопоточном исполнителе Executor. *Так делать не следует*

```
public class ThreadDeadlock {
    ExecutorService exec = Executors.newSingleThreadExecutor();

    public class RenderPageTask implements Callable<String> {
        public String call() throws Exception {
            Future<String> header, footer;
            header = exec.submit(new LoadFileTask("header.html"));
            footer = exec.submit(new LoadFileTask("footer.html"));
            String page = renderBody();
            // Запирается взаимной блокировкой – задача, ожидающая
            // результат подзадачи
```

```
            return header.get() + page + footer.get();
        }
    }
}
```

8.1.2. Длительные задачи

Если размер пула слишком мал по сравнению с ожидаемым числом длительных задач, то пострадает отзывчивость.

Чтобы смягчить негативные последствия длительных задач, можно использовать хронометрированные ожидания ресурсов вместо неограниченных. Большинство блокирующих методов в структурных библиотеках поставляются в виде нехронометрированных и хронометрированных версий, таких как `Thread.join`, `BlockingQueue.put`, `CountDownLatch.await` и `Selector.select`. Если время ожидания истекло, вы можете пометить задачу как безуспешную и прервать ее или поставить ее в очередь заново, чтобы выполнить позже. Этим гарантируется, что каждая задача продвигается в направлении успешного или безуспешного завершения, освобождая потоки для задач, которые могут завершиться быстрее. Если пул потоков заполнен заблокированными задачами, это может означать, что он слишком мал.

8.2. Определение размера пула потоков

Размер пула потоков зависит от типов задач, которые будут предоставляться, и характеристик системы развертывания. Он не кодируется жестко, а предоставляется конфигурационным механизмом или вычисляется динамически, консультируясь с `Runtime.availableProcessors`.

В слишком большом пуле потоки конфликтуют за ресурсы процессора и памяти, чрезмерно потребляя их. В слишком малом пуле страдает пропускная способность, поскольку процессоры остаются невостребованными, несмотря на имеющуюся работу.

Чтобы правильно определить размер пула потоков, нужно понимать вычислительную среду, ресурсный бюджет и характер задач. Сколько имеется процессоров в системе развертывания? Сколько памяти? Выполняют ли задачи в основном вычисления, ввод-вывод или какую-то их комбинацию? Требуют ли они дефицитных ресурсов, таких как соедине-

ние JDBC? Если у вас имеются разные категории задач с очень разными формами поведения, рассмотрите возможность использования нескольких пулов, каждый из которых можно отрегулировать в соответствии с его рабочей нагрузкой.

Для вычислительно-емких задач N_{cpu}-процессорная система обычно достигает оптимальной работы с пулом $N_{cpu} + 1$. (Даже вычислительно-емкие потоки периодически делают страничный отказ или берут паузу по какой-либо причине, поэтому «лишний» работоспособный поток защищает от недоиспользования процессорных циклов.) Для задач, которые включают операции ввода-вывода или другие блокирующие операции, требуется больший пул. Чтобы правильно определить размер пула, вы должны оценить отношение времени ожидания ко времени вычисления ваших задач; эта оценочная величина необязательно должна быть точной и может быть получена с помощью профилирования или контрольных измерений. Например, размер пула можно регулировать, выполняя приложение с использованием нескольких разных размеров под контрольной нагрузкой и наблюдая уровень задействованности процессоров.

Оптимальный размер пула для поддержания процессоров на желаемом уровне задействованности равен:

$$N_{потоков} = N_{cpu} \times U_{cpu} \times (1 + \frac{W}{C}),$$

где

N_{cpu} = *число процессоров*;

U_{cpu} = *целевая задействованность процессоров*, $0 \le U_{cpu} \le 1$;

$\frac{W}{C}$ = *отношение времени ожидания ко времени вычисления*.

Число процессоров можно определить с помощью статического класса `Runtime`:

```
int N_CPUS = Runtime.getRuntime().availableProcessors();
```

Конечно, циклы процессора — это не единственный ресурс, которым вы, возможно, захотите управлять с помощью пула потоков. Другие ресурсы,

которые могут способствовать определению размера пула, — это память, файловые и сокетные дескрипторы и подключения к базам данных. Вычислять ограничения на размер пула для этих типов ресурсов проще: разделите количество ресурса, требуемого каждой задаче, на суммарное имеющееся количество. Результатом будет верхняя граница размера пула.

Когда задачам требуется объединенный в пул ресурс, например подключения к базе данных, размер пула потоков и размер ресурсного пула влияют друг на друга. Если для каждой задачи требуется подключение, то эффективный размер пула потоков ограничен размером пула соединений. Схожим образом, когда единственными потребителями соединений являются задачи пула, эффективный размер пула соединений ограничен размером пула потоков.

8.3. Конфигурирование класса ThreadPoolExecutor

Класс `ThreadPoolExecutor` предоставляет базовую реализацию для исполнителей, возвращаемых фабриками `newCachedThreadPool`, `newFixedThreadPool` и `newScheduledThreadExecutor` в исполнителях `Executor`. Исполнитель `ThreadPoolExecutor` представляет собой гибкую, надежную реализацию пула, которая позволяет выполнять различные настройки.

Если принятая по умолчанию политика выполнения не соответствует вашим потребностям, то вы можете создать экземпляр класса `ThreadPoolExecutor` с помощью его конструктора и настроить его по своему усмотрению: обратиться к исходному коду исполнителей `Executor` и использовать в качестве отправной точки политики выполнения для стандартных конфигураций. `ThreadPoolExecutor` имеет несколько конструкторов, наиболее общий из которых показан в листинге 8.2.

8.3.1. Создание и удаление потоков

Ядерный размер пула, максимальный размер пула и время поддержания потока в активном состоянии управляют созданием и удалением потоков. Ядерный размер — это целевой размер: реализация пытается поддерживать пул в этом размере, даже если нет задач для выполнения, и не будет создавать новые потоки, количеством превышающие это число, если ра-

бочая очередь не заполнена[1]. Максимальный размер пула — это верхняя граница числа потоков пула, которые могут быть активны одновременно. Поток, который простаивал дольше периода поддержания потока в активном состоянии, будет терминирован, если текущий размер пула превысит ядерный размер.

Листинг 8.2. Общий конструктор для класса ThreadPoolExecutor

```
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue,
                          ThreadFactory threadFactory,
                          RejectedExecutionHandler handler) { ... }
```

Регулируя ядерный размер пула и время поддержания потоков в активном состоянии, вы можете поощрять пул к высвобождению ресурсов, используемых другими простаивающими потоками, что сделает их доступными для полезной работы. (Убирая простаивающие потоки, вы вызываете дополнительную задержку из-за создания потока, отложенного до момента увеличения спроса на потоки.)

Фабрика `newFixedThreadPool` устанавливает максимальный размер пула равным `Integer.MAX_VALUE` и ядерный размер пула равным нулю, с таймаутом в одну минуту, создавая эффект бесконечно расширяемого пула потоков, который будет сокращаться при уменьшении спроса на потоки.

---

[1] Иногда разработчики устанавливают ядерный размер равным нулю, для того чтобы рабочие потоки в конечном итоге были демонтированы и не препятствовали выходу JVM. Но это может вызвать странное поведение пула, который не использует очередь `SynchronousQueue` в качестве своей рабочей очереди (как это делает `newCachedThreadPool`). Если пул уже имеет ядерный размер, то исполнитель `ThreadPoolExecutor` создает новый поток, только если рабочая очередь заполнена. Поэтому задачи, предоставленные пулу потоков с рабочей очередью, которая имеет любую емкость и нулевой ядерный размер, не будут выполняться, до тех пор пока очередь не заполнится, что нежелательно. В Java 6 метод `allowCoreThreadTimeOut` позволяет запрашивать остановку потоков по тайм-ауту. Активируйте этот функционал с помощью нулевого ядерного размера, если вы хотите иметь ограниченный пул потоков с ограниченной рабочей очередью, но при этом иметь возможность демонтировать потоки, когда нет работы.

Другие комбинации возможны с помощью конструктора исполнителя `ThreadPoolExecutor`.

8.3.2. Управление задачами очереди

Ограниченный пул потоков задает предел числа задач, которые могут выполняться конкурентно. (Однопоточные исполнители — частный случай такого пула — гарантируют, что никакие задачи не будут выполняться конкурентно, предлагая возможность достижения потокобезопасности через ограничение одним потоком.)

В разделе 6.1.2 мы использовали пул потоков фиксированного размера вместо создания нового потока для каждого запроса, но оставили риск исчерпания ресурсов под большой нагрузкой. Если скорость прибытия новых запросов превышает скорость их обработки, то запросы будут выстраиваться в очереди. В пуле потоков они будут ждать в очереди из объектов `Runnable`, управляемой исполнителем `Executor`, вместо того чтобы выстраиваться в очереди как потоки, конфликтующие за процессор. Предоставление ожидающей задачи с помощью `Runnable` и спискового узла, безусловно, намного дешевле, чем с помощью потока, но риск исчерпания ресурсов не исчезнет, если клиенты забросают сервер запросами быстрее, чем он сможет их обрабатывать.

Запросы часто поступают пакетами, даже если их средняя скорость стабильна. Очереди помогают сглаживать кратковременные всплески прибытия задач, но если задачи продолжают поступать слишком быстро, то вам придется регулировать скорость их прибытия, чтобы избежать нехватки памяти[1]. Даже до того, как закончится память, время ответа будет замедляться по мере роста задачной очереди.

Исполнитель `ThreadPoolExecutor` позволяет предоставить очередь `BlockingQueue` для хранения задач, ожидающих выполнения. Существует три основных подхода к организации очередности задач: неограниченная очередь, ограниченная очередь и синхронная эстафетная передача. Выбор очереди взаимодействует с другими параметрами конфигурации, такими как размер пула.

---

[1] Аналогично поток управляется в коммуникационных сетях: даже используя буфер, вы захотите побудить другую сторону прекратить отправку данных и будете надеяться, что отправитель вышлет их повторно в удобное для вас время.

По умолчанию для `newFixedThreadPool` и `newSingleThreadExecutor` используется неограниченная очередь `LinkedBlockingQueue`. Задачи помещаются в очередь, если все рабочие потоки заняты, но очередь может расти неограниченно, если задачи продолжают поступать быстрее, чем они могут быть выполнены.

Более стабильной стратегией управления ресурсами является использование ограниченной очереди, такой как `ArrayBlockingQueue`, `LinkedBlockingQueue` или `PriorityBlockingQueue`. Она помогает предотвратить исчерпание ресурсов, но ставит вопрос о том, что делать с новыми задачами, когда очередь заполнена. (Возможные *политики насыщения* описаны в разделе 8.3.3.) Размер ограниченной рабочей очереди и размер пула должны соответствовать друг другу.

Очередь `SynchronousQueue` является механизмом управления эстафетной передачей между потоками. Элемент помещается в `SynchronousQueue` с помощью ожидающего его потока. Если ни один поток не ожидает, но текущий размер пула меньше максимума, то исполнитель `ThreadPoolExecutor` создает новый поток либо задача отклоняется согласно политике насыщения. Благодаря прямой эстафетной передаче задача может быть передана непосредственно выполняющему потоку, минуя очередь. `SynchronousQueue` полезна, если пул неограничен или допустимо отклонение избыточных задач. Фабрика `newCachedThreadPool` использует этот тип очереди.

В очередях с FIFO, таких как `LinkedBlockingQueue` или `ArrayBlockingQueue`, задачи запускаются в порядке поступления. Очередь `PriorityBlockingQueue` упорядочивает задачи согласно приоритету, определенному естественным порядком (если задачи реализуют интерфейс `Comparable`) либо через `Comparator`.

Фабрика `newCachedThreadPool` является хорошим выбором для исполнителя `Executor`, так как обеспечивает более высокую производительность очереди, чем фиксированный пул потоков[1], который, в свою очередь, уместен при ограниченном числе конкурентных задач.

[1] Разница в производительности вызвана использованием `SynchronousQueue` вместо `LinkedBlockingQueue`. Очередь `SynchronousQueue` была заменена в Java 6 новым неблокирующим алгоритмом, который улучшил пропускную способность в эталонных тестах исполнителя задач `Executor` в три раза по сравнению с реализацией `SynchronousQueue` в Java 5.0 (Scherer и соавт., 2006).

Ограничение пула потоков или рабочей очереди подходит только в том случае, если задачи являются независимыми. С задачами, зависящими от других задач, ограниченные пулы потоков или очереди могут вызывать взаимную блокировку с ресурсным голоданием. Используйте конфигурацию неограниченного пула, такого как `newCachedThreadPool`[1].

8.3.3. Политика насыщения

Когда ограниченная рабочая очередь заполнена, в игру вступает *политика насыщения*. Для исполнителя `ThreadPoolExecutor` она может быть изменена путем вызова обработчика `setRejectedExecutionHandler`. (Политика насыщения также используется, когда задача передается исполнителю `Executor`, который был выключен.) Есть несколько реализаций обработчика `RejectedExecutionHandler`, поддерживающих разные политики насыщения: `AbortPolicy`, `CallerRunsPolicy`, `DiscardPolicy` и `DiscardOldestPolicy`.

По умолчанию применяется политика *абортирования* (abort), при которой метод `execute` выдает непроверяемое исключение `RejectedExecutionException`. Вызывающий элемент кода может отловить это исключение и реализовать обработку переполнения по своему усмотрению. Политика *отбрасывания* (discard) молчаливо отбрасывает только что предоставленную задачу, если ее нельзя поставить в очередь на выполнение. Политика *отбрасывания самой старой задачи* (discard-oldest) отбрасывает задачу, которая в иных случаях была бы выполнена позднее, и пытается повторно предоставить новую задачу. (Если рабочая очередь имеет приоритеты, то отбрасывается элемент с наивысшим приоритетом, что противоречит функциям очереди.)

Политика под управлением вызывающего элемента кода (caller-runs policy) старается замедлить поток новых задач, направляя некоторую работу назад вызывающему элементу кода. Она выполняет только что предоставленную задачу не в пуле, а в потоке, который вызывает метод `execute`. Если бы мы изменяли пример `WebServer` так, чтобы он использовал ограниченную очередь и политику под управлением вызывающего элемента кода, то в конце концов все потоки пула были бы заняты, и рабочая очередь,

---

[1] Альтернативной конфигурацией для задач, предоставляющих другие задачи и ожидающих их результаты, является использование ограниченного пула потоков, очереди `SynchronousQueue` в качестве рабочей очереди и политики насыщения под управлением вызывающего кода.

заполненная до конца следующей задачей, выполнялась бы во время вызова метода `execute` в главном потоке. Некоторое время главный поток не смог бы предоставлять задачи, давая рабочим потокам шанс нивелировать отставание, и не вызвал бы метод `accept` в это время. Поэтому входящие запросы стали бы накапливаться в очереди на уровне TCP, а не в приложении. При длительной перегрузке TCP тоже начал бы отбрасывать запросы, на этот раз — клиенту и с более плавной деградацией.

Выбор политики насыщения или внесение других изменений в политику выполнения могут быть сделаны во время создания исполнителя. Листинг 8.3 иллюстрирует создание пула потоков фиксированного размера с политикой насыщения под управлением вызывающего элемента кода.

Листинг 8.3. Создание пула потоков фиксированного размера с ограниченной очередью и политикой насыщения под управлением вызывающего элемента кода

```
ThreadPoolExecutor executor
    = new ThreadPoolExecutor(N_THREADS, N_THREADS,
        0L, TimeUnit.MILLISECONDS,
        new LinkedBlockingQueue<Runnable>(CAPACITY));
executor.setRejectedExecutionHandler(
    new ThreadPoolExecutor.CallerRunsPolicy());
```

Политика насыщения, при которой метод `execute` блокирует продвижение, когда рабочая очередь заполнена, отсутствует. Однако блокирование может быть достигнуто с помощью `Semaphore`, ограничивающего скорость внесения задач, как показано в `BoundedExecutor` в листинге 8.4. Используйте неограниченную очередь и установите границу семафора равной сумме размера пула и числа задач очереди, которое вы хотите допустить (семафор ограничивает число задач, как выполняемых в данный момент, так и ожидающих выполнения).

8.3.4. Фабрики потоков

Пул создает потоки посредством *фабрики потоков* (см. листинг 8.5), которая по умолчанию создает поток, не являющийся демоном, без специальной конфигурации. Кастомизация фабрики потоков позволяет настроить конфигурацию потоков пула. Фабрика `ThreadFactory` имеет единственный метод, `newThread`, который вызывается всякий раз, когда пулу необходимо создать новый поток.

Существует ряд причин для использования настраиваемой фабрики потоков. Вы, возможно, захотите указать обработчика `UncaughtExceptionHandler` для потоков пула или создать экземпляр настраиваемого класса `Thread`, например выполняющего журналирование отладочной информации. Или захотите изменить приоритет (не лучшая идея; см. раздел 10.3.1) или установить потоку демон-статус (тоже не очень хорошая идея; см. раздел 7.4.2). Или дать потокам более содержательные имена, для того чтобы упростить интерпретацию поточных дампов и журналов регистрации ошибок.

Листинг 8.4. Использование семафора для регулирования запуска задач

```
@ThreadSafe
public class BoundedExecutor {
    private final Executor exec;
    private final Semaphore semaphore;

    public BoundedExecutor(Executor exec, int bound) {
        this.exec = exec;
        this.semaphore = new Semaphore(bound);
    }

    public void submitTask(final Runnable command)
            throws InterruptedException {
        semaphore.acquire();
        try {
            exec.execute(new Runnable() {
                public void run() {
                    try {
                        command.run();
                    } finally {
                        semaphore.release();
                    }
                }
            });
        } catch (RejectedExecutionException e) {
            semaphore.release();
        }
     }
}
```

Листинг 8.5. Интерфейс ThreadFactory

```
public interface ThreadFactory {
    Thread newThread(Runnable r);
}
```

Класс MyThreadFactory в листинге 8.6 иллюстрирует настраиваемую фабрику потоков. Он создает экземпляр, инстанцировав класс MyAppThread и передав специфичное для пула имя конструктору. Класс MyAppThread также можно использовать в других местах приложения, благодаря чему все потоки смогут воспользоваться его отладочным функционалом.

Листинг 8.6. Настраиваемая фабрика потоков

```
public class MyThreadFactory implements ThreadFactory {
    private final String poolName;

    public MyThreadFactory(String poolName) {
        this.poolName = poolName;
    }

    public Thread newThread(Runnable runnable) {
        return new MyAppThread(runnable, poolName);
    }
}
```

Как показано в листинге 8.7, в классе MyAppThread предоставляется имя потока и задается настраиваемый обработчик UncaughtExceptionHandler, который пишет сообщение в диспетчер журналирования Logger, ведет статистику количества созданных и уничтоженных потоков и записывает отладочное сообщение в журнал регистрации, когда поток создан или терминируется.

Листинг 8.7. Базовый класс настраиваемого потока

```
public class MyAppThread extends Thread {
    public static final String DEFAULT_NAME = "MyAppThread";
    private static volatile boolean debugLifecycle = false;
    private static final AtomicInteger created = new AtomicInteger();
    private static final AtomicInteger alive = new AtomicInteger();
    private static final Logger log = Logger.getAnonymousLogger();

    public MyAppThread(Runnable r) { this(r, DEFAULT_NAME); }

    public MyAppThread(Runnable runnable, String name) {
        super(runnable, name + "-" + created.incrementAndGet());
        setUncaughtExceptionHandler(
            new Thread.UncaughtExceptionHandler() {
                public void uncaughtException(Thread t,
                                              Throwable e) {
                    log.log(Level.SEVERE,
                        "НЕ ОТЛОВЛЕН в потоке " + t.getName(), e);
```

```
                }
            });
    }

    public void run() {
        // Отладочный флажок копирования для повсеместного
        // обеспечения непротиворечивого значения.
        boolean debug = debugLifecycle;
        if (debug) log.log(Level.FINE, "Создан "+getName());
        try {
            alive.incrementAndGet();
            super.run();
        } finally {
            alive.decrementAndGet();
            if (debug) log.log(Level.FINE, "Завершаю "+getName());
        }
    }

    public static int getThreadsCreated() { return created.get(); }
    public static int getThreadsAlive() { return alive.get(); }
    public static boolean getDebug() { return debugLifecycle; }
    public static void setDebug(boolean b) { debugLifecycle = b; }
}
```

Если ваше приложение пользуется преимуществом *политик безопасности*, используйте фабричный метод `privilegedThreadFactory` в исполнителях для создания фабрики потоков. Он создает потоки в пуле, которые имеют те же разрешения, `AccessControlContext` и `contextClassLoader`, что и поток, создающий фабрику `privilegedThreadFactory`. Помните, что потоки, созданные пулом, наследуют разрешения от любого клиента, который будет вызывать методы `execute` или `submit`, что может привести к исключениям, связанным с безопасностью.

8.3.5. Настройка класса ThreadPoolExecutor после конструирования

Большинство параметров, передаваемых конструкторам класса `ThreadPoolExecutor`, могут быть изменены после конструирования посредством методов доступа `set` (в частности, ядерный и максимальный размеры пула потоков, время поддержания потока в активном состоянии, фабрика потоков и обработчик отклоненного выполнения). Если `Executor` создается с помощью одного из фабричных методов в классах `Executor` (кроме класса `newSingleThreadExecutor`), то для обращения к методам доступа `set`

вы можете привести результат к типу `ThreadPoolExecutor`, как показано в листинге 8.8.

Исполнители содержат фабричный метод, `unconfigurableExecutorService`, который берет существующую службу `ExecutorService` и обертывает ее, предоставляя только методы службы `ExecutorService`, в результате чего она не может быть сконфигурирована далее. В отличие от объединенных в пул реализаций, `newSingleThreadExecutor` возвращает службу `ExecutorService`, обернутую описанным образом. В то время как однопоточный исполнитель фактически реализован как пул потоков, он обещает не выполнять задачи конкурентно. Если бы какой-то заблудший код увеличил размер пула на однопоточном исполнителе, это подорвало бы вмененную семантику выполнения.

Листинг 8.8. Изменение исполнителя, созданного с помощью стандартных фабрик

```
ExecutorService exec = Executors.newCachedThreadPool();
if (exec instanceof ThreadPoolExecutor)
    ((ThreadPoolExecutor) exec).setCorePoolSize(10);
else
    throw new AssertionError("Опаньки! Плохое допущение");
```

Вы можете использовать это техническое решение со своими исполнителями для предотвращения изменения политики выполнения. Если вы будете предоставлять службу `ExecutorService` коду, который может ее изменить, оберните ее в службу `unconfigurableExecutorService`.

8.4. Расширение класса ThreadPoolExecutor

Класс `ThreadPoolExecutor` был спроектирован для расширения с помощью перехватчиков — методов `beforeExecute`, `afterExecute` и `terminate`.

Перехватчики `beforeExecute` и `afterExecute` вызываются в потоке, выполняющем задачу, и используются для добавления журналирования, хронометрирования, мониторинга или сбора статистики. Перехватчик `afterExecute` вызывается независимо от того, как задача завершается: возвращаясь из метода `run` нормальным образом либо выдавая исключение. (Если задача завершается с ошибкой `Error`, то перехватчик `afterExecute` не вызывается.) Если перехватчик `beforeExecute` выдает исключение `RuntimeException`, то задачи не выполняются, и перехватчик `afterExecute` не вызывается.

Перехватчик `terminated` вызывается, когда пул потоков завершает процесс выключения. Он может использоваться для высвобождения ресурсов, выполнения уведомлений или журналирования, а также финализирования сбора статистики.

8.4.1. Пример: добавление статистики в пул потоков

Класс `TimingThreadPool` в листинге 8.9 показывает настраиваемый пул потоков, который задействует перехватчиков `beforeExecute`, `afterExecute` и `terminated` для добавления журналирования и сбора статистики. Для того чтобы измерить время работы задачи, перехватчик `beforeExecute` записывет время начала и сохраняет его там, где перехватчик `afterExecute` его найдет. Класс `TimingThreadPool` использует пару объектов `AtomicLong`, для того чтобы отслеживать суммарное число обработанных задач и суммарное время обработки, и использует перехватчик `terminated`, для того чтобы печатать журнальное сообщение, показывающее среднее время задачи.

Листинг 8.9. Пул потоков, расширенный журналированием и хронометрированием

```
public class TimingThreadPool extends ThreadPoolExecutor {
    private final ThreadLocal<Long> startTime
            = new ThreadLocal<Long>();
    private final Logger log = Logger.getLogger("TimingThreadPool");
    private final AtomicLong numTasks = new AtomicLong();
    private final AtomicLong totalTime = new AtomicLong();

    protected void beforeExecute(Thread t, Runnable r) {
        super.beforeExecute(t, r);
        log.fine(String.format("Поток %s: начало %s", t, r));
        startTime.set(System.nanoTime());
    }

    protected void afterExecute(Runnable r, Throwable t) {
        try {
            long endTime = System.nanoTime();
            long taskTime = endTime - startTime.get();
            numTasks.incrementAndGet();
            totalTime.addAndGet(taskTime);
            log.fine(String.format("Поток %s: конец %s, время=%dns",
                     t, r, taskTime));
        } finally {
```

продолжение ⇩

Листинг 8.9 *(продолжение)*

```
            super.afterExecute(r, t);
        }
    }

    protected void terminated() {
        try {
            log.info(String.format("Terminated: avg time=%dns",
                    totalTime.get() / numTasks.get()));
        } finally {
            super.terminated();
        }
    }
}
```

8.5. Параллелизация рекурсивных алгоритмов

Циклы, тела которых содержат нетривиальные вычисления или выполняют потенциально блокирующий ввод-вывод, часто являются хорошими кандидатами для параллелизации, если их итерации являются независимыми.

Используйте исполнитель `Executor` для преобразования последовательного цикла в параллельный, как показано в методах `processSequentially` и `processInParallel` в листинге 8.10.

Листинг 8.10. Преобразование последовательного выполнения в параллельное

```
void processSequentially(List<Element> elements) {
    for (Element e : elements)
        process(e);
}

void processInParallel(Executor exec, List<Element> elements) {
    for (final Element e : elements)
        exec.execute(new Runnable() {
            public void run() { process(e); }
        });
}
```

Вызов метода `processInParallel` возвращается быстро, как только все задачи поставлены в очередь в исполнителе до их завершения. Если

вам требуются завершенные задачи, примените метод `ExecutorService.invokeAll`. Вы сможете получать результаты по мере их появления и использовать службу `CompletionService`, как в классе `Renderer`.

> Последовательные итерации цикла подходят для параллелизации, когда каждая итерация является независимой от других, и работа, выполняемая в каждой итерации тела цикла, является достаточно значительной, для того чтобы компенсировать стоимость управления новой задачей.

Параллелизация циклов также может быть применена к некоторым рекурсивным проектам. В рекурсивном алгоритме часто встречаются последовательные циклы, которые могут быть параллелизованы, как в листинге 8.10. Если каждая итерация не требует результатов рекурсивных итераций, которые она инициирует, параллелизация пройдет еще проще. Например, `sequentialRecursive` в листинге 8.11 делает обход дерева сперва в глубину, выполняя вычисление на каждом узле и помещая результат в коллекцию. Преобразованная версия `parallelRecursive` также делает обход сперва в глубину, но вместо вычисления результата при посещении каждого узла она предоставляет задачу для вычисления результата узла.

Листинг 8.11. Преобразование последовательной хвостовой рекурсии в параллелизованную рекурсию

```
public<T> void sequentialRecursive(List<Node<T>> nodes,
                                   Collection<T> results) {
    for (Node<T> n : nodes) {
        results.add(n.compute());
        sequentialRecursive(n.getChildren(), results);
    }
}

public<T> void parallelRecursive(final Executor exec,
                                 List<Node<T>> nodes,
                                 final Collection<T> results) {
    for (final Node<T> n : nodes) {
        exec.execute(new Runnable() {
            public void run() {
                results.add(n.compute());
            }
        });
        parallelRecursive(exec, n.getChildren(), results);
    }
}
```

Когда `parallelRecursive` возвращается, это значит, что каждый узел в дереве был посещен (обход по-прежнему является последовательным: только вызовы метода `compute` выполняются параллельно), и вычисление для каждого узла было поставлено в очередь в исполнителе. Элементы кода, вызывающие метод `parallelRecursive`, могут ожидать все результаты, создав специфичный для обхода исполнитель и используя методы `shutdown` и `awaitTermination`, как показано в листинге 8.12.

Листинг 8.12. Ожидание результатов, которые будут вычислены параллельно

```
public<T> Collection<T> getParallelResults(List<Node<T>> nodes)
        throws InterruptedException {
    ExecutorService exec = Executors.newCachedThreadPool();
    Queue<T> resultQueue = new ConcurrentLinkedQueue<T>();
    parallelRecursive(exec, nodes, resultQueue);
    exec.shutdown();
    exec.awaitTermination(Long.MAX_VALUE, TimeUnit.SECONDS);
    return resultQueue;
}
```

8.5.1. Пример: фреймворк головоломки

Решение головоломок, таких как «Скользящие блоки»[1], «Солитер», «Мгновенное безумие» и т. п., предусматривает поиск последовательности преобразований некоего начального состояния для достижения целевого состояния.

Мы определяем головоломку как комбинацию начальной и целевой позиций и правил, определяющих допустимые ходы. Множество правил состоит из двух частей: вычисление списка законных ходов из заданной позиции и вычисление результата применения хода к позиции. В листинге 8.13 показана абстракция головоломки. Параметры типа `P` и `M` представляют классы для позиции и хода. Из этого интерфейса мы можем написать простой последовательный решатель, который выполняет поиск в пространстве головоломки, до тех пор пока не будет найдено решение либо пространство головоломки не будет исчерпано.

[1] См. http://www.puzzleworld.org/SlidingBlockPuzzles

Листинг 8.13. Абстракция для головоломок, таких как «Скользящие блоки»

```
public interface Puzzle<P, M> {
    P initialPosition();
    boolean isGoal(P position);
    Set<M> legalMoves(P position);
    P move(P position, M move);
}
```

Узел в листинге 8.14 представляет собой позицию, достигнутую после серии ходов, содержащую ссылку на ход, создавший эту позицию, и предыдущий узел. Следуя по ссылкам назад от узла, можно восстановить последовательность ходов, которые привели к текущей позиции.

Листинг 8.14. Узел связи для структуры решателя головоломок

```
@Immutable
static class Node<P, M> {
    final P pos;
    final M move;
    final Node<P, M> prev;

    Node(P pos, M move, Node<P, M> prev) {...}

    List<M> asMoveList() {
        List<M> solution = new LinkedList<M>();
        for (Node<P, M> n = this; n.move != null; n = n.prev)
            solution.add(0, n.move);
        return solution;
    }
}
```

Класс `SequentialPuzzleSolver` в листинге 8.15 показывает последовательный решатель для структуры головоломок, который начинает вести поиск в глубину пространства головоломки. Он терминируется, когда находит решение (не обязательно найденное кратчайшим путем).

Переработка решателя с целью задействования конкурентности позволит вычислять следующие ходы и оценивать целевое условие параллельно, так как процесс оценивания одного хода в основном не зависит от оценивания других ходов. (При этом задачи используют совместно некое мутируемое состояние, такое как множество видимых позиций.) Многочисленные процессоры могут сократить время поиска решения.

Листинг 8.15. Последовательная версия решателя головоломок

```
public class SequentialPuzzleSolver<P, M> {
    private final Puzzle<P, M> puzzle;
    private final Set<P> seen = new HashSet<P>();

    public SequentialPuzzleSolver(Puzzle<P, M> puzzle) {
        this.puzzle = puzzle;
    }

    public List<M> solve() {
        P pos = puzzle.initialPosition();
        return search(new Node<P, M>(pos, null, null));
    }

    private List<M> search(Node<P, M> node) {
        if (!seen.contains(node.pos)) {
            seen.add(node.pos);
            if (puzzle.isGoal(node.pos))
                return node.asMoveList();
            for (M move : puzzle.legalMoves(node.pos)) {
                P pos = puzzle.move(node.pos, move);
                Node<P, M> child = new Node<P, M>(pos, move, node);
                List<M> result = search(child);
                if (result != null)
                    return result;
            }
        }
        return null;
    }
    static class Node<P, M> { /* Listing 8.14 */ }
}
```

Класс `ConcurrentPuzzleSolver` в листинге 8.16 использует внутренний класс `SolverTask`, который расширяет `Node` и реализует `Runnable`. Основная часть работы выполняется в методе `run`: оценка множества возможных следующих позиций, подрезание уже найденных позиций, оценка достижения успеха (этой задачей или какой-либо другой) и предоставление неразведанных позиций исполнителю.

Листинг 8.16. Конкурентная версия решателя головоломок

```
public class ConcurrentPuzzleSolver<P, M> {
    private final Puzzle<P, M> puzzle;
    private final ExecutorService exec;
    private final ConcurrentMap<P, Boolean> seen;
```

```
    final ValueLatch<Node<P, M>> solution
            = new ValueLatch<Node<P, M>>();
    ...
    public List<M> solve() throws InterruptedException {
        try {
            P p = puzzle.initialPosition();
            exec.execute(newTask(p, null, null));
            // блокировать до тех пор, пока решение не будет найдено
            Node<P, M> solnNode = solution.getValue();
            return (solnNode == null) ? null : solnNode.asMoveList();
        } finally {
            exec.shutdown();
        }
    }

    protected Runnable newTask(P p, M m, Node<P,M> n) {
        return new SolverTask(p, m, n);
    }

    class SolverTask extends Node<P, M> implements Runnable {
        ...
        public void run() {
            if (solution.isSet()
                    || seen.putIfAbsent(pos, true) != null)
                return; // already solved or seen this position
            if (puzzle.isGoal(pos))
                solution.setValue(this);
            else
                for (M m : puzzle.legalMoves(pos))
                    exec.execute(
                        newTask(puzzle.move(pos, m), m, this));
        }
    }
}
```

Чтобы избежать бесконечных циклов, последовательная версия поддерживала множество `Set` ранее разведанных позиций. Класс `ConcurrentPuzzleSolver` для этой цели использует хеш-массив `ConcurrentHashMap`. Он обеспечивает потокобезопасность и позволяет избежать состояния гонки, присущего условному обновлению совместной коллекции, применяя метод `putIfAbsent` («добавить, если отсутствует») для атомарного добавления ранее не известной позиции. Для хранения состояния поиска класс `ConcurrentPuzzleSolver` использует внутреннюю рабочую очередь пула потоков вместо стека вызовов.

Конкурентный подход ищет форму, подходящую для предметной области. Последовательная версия начинала вести поиск в глубину, поэтому он ограничивался доступным размером стека. Конкурентная версия начинает поиск в ширину и свободна от такого ограничения (но по-прежнему может исчерпать память при чрезмерном количестве позиций).

Если мы хотим принимать первое найденное решение, то нам также нужно обновлять решение, только если ни одна другая задача еще не нашла ни одного решения. Эти требования описывают разновидность защелки (см. раздел 5.5.1) и, в частности, *защелки, приносящей результат*, которую можно построить по рекомендациям из главы 14, но проще использовать существующие библиотечные классы, а не низкоуровневые языковые механизмы. Для обеспечения необходимого поведения класс `ValueLatch` в листинге 8.17 использует защелку `CountDownLatch` и блокировку, обеспечивающую установку решения только один раз.

Листинг 8.17. Защелка, приносящая результат, используемый решателем ConcurrentPuzzleSolver

```
@ThreadSafe
public class ValueLatch<T> {
    @GuardedBy("this") private T value = null;
    private final CountDownLatch done = new CountDownLatch(1);

    public boolean isSet() {
        return (done.getCount() == 0);
    }

    public synchronized void setValue(T newValue) {
        if (!isSet()) {
            value = newValue;
            done.countDown();
        }
    }

    public T getValue() throws InterruptedException {
        done.await();
        synchronized (this) {
            return value;
        }
    }
}
```

Каждая задача сначала консультируется с защелкой решения и останавливается, если решение уже найдено. Главный поток должен ожидать

нахождение решения. Метод `getValue` в классе `ValueLatch` блокирует продвижение до тех пор, пока какой-либо поток не установит значение. `ValueLatch` хранит значение таким образом, чтобы только первый вызов фактически устанавливал значение, а вызывающие элементы кода проверяли, было ли оно установлено, и блокировали продвижение в ожидании этого события. При первом вызове метода `setValue` решение обновляется и защелка `CountDownLatch` уменьшается, освобождая главный поток из метода `getValue`.

Первый поток, нашедший решение, выключает исполнитель, не давая принимать новые задачи. Для того чтобы избежать необходимости заниматься исключением `RejectedExecutionException`, отклоненный обработчик выполнения должен быть настроен так, чтобы отбрасывать предоставленные задачи. Незавершенные задачи дойдут до полного завершения, а попытки выполнить новые задачи не сработают, позволив исполнителю терминироваться. (Если работа задач продолжится дольше, то мы, возможно, захотим прервать их, вместо того чтобы позволить им завершиться.)

В классе `ConcurrentPuzzleSolver`, если все возможные ходы и позиции были оценены и решение не было найдено, метод `solve` будет ждать вечно в вызове метода `getSolution`. Последовательная версия завершается, когда она исчерпала пространство поиска, но добиться терминации конкурентных программ сложнее. Одним из решений является хранение счетчика активных задач решателя и установка решения равным `null`, когда этот счетчик падает до нуля, как показано в листинге 8.18.

Листинг 8.18. Решатель, распознающий отсутствие решения

```
public class PuzzleSolver<P,M> extends ConcurrentPuzzleSolver<P,M> {
    ...
    private final AtomicInteger taskCount = new AtomicInteger(0);

    protected Runnable newTask(P p, M m, Node<P,M> n) {
        return new CountingSolverTask(p, m, n);
    }

    class CountingSolverTask extends SolverTask {
        CountingSolverTask(P pos, M move, Node<P, M> prev) {
            super(pos, move, prev);
            taskCount.incrementAndGet();
        }
        public void run() {
            try {
```

продолжение ⇩

Листинг 8.18 *(продолжение)*

```
                super.run();
            } finally {
                if (taskCount.decrementAndGet() == 0)
                    solution.setValue(null);
            }
        }
    }
}
```

К дополнительным условиям завершения, которые мы можем наложить на поиск, относятся: ограничение по времени хронометрированным методом `getValue` в классе `ValueLatch` и введение специфичного для головоломки метрического показателя, такого как поиск только до определенного числа позиций. Еще мы можем предоставить механизм отмены и позволить клиенту самостоятельно принять решение о том, когда прекратить поиск.

Итоги

`Executor` — это мощная и гибкая структура для конкурентного выполнения задач. Он предлагает ряд регулировочных параметров, таких как политики для создания и удаления потоков, обработки задач в очереди и действий с избыточными задачами, а также предоставляет несколько перехватчиков для расширения своего поведения. Однако, как и в большинстве мощных структур, в нем существуют комбинации параметров, которые плохо работают вместе: некоторые типы задач требуют специфических политик выполнения, а некоторые комбинации регулировочных параметров могут привести к непредвиденным результатам.

9 Приложения с GUI

Поточная обработка GUI-приложения на платформе Swing имеет ряд особенностей. Структуры данных Swing не являются потокобезопасными.

Почти все инструментальные средства GUI, включая Swing и SWT, реализованы как *однопоточные подсистемы*. Вы можете использовать действия, которые частично работают в потоке приложения и частично — в потоке событий, но идентифицировать сбой программы, вызванный некорректной работой этих действий, будет трудно.

9.1. Почему GUI-интерфейсы являются однопоточными?

Изначально GUI-приложения были однопоточными, а события GUI обрабатывались из главного событийного цикла. Современные структуры GUI используют модель, которая отличается лишь немногим: для обработки событий GUI они создают выделенный *поток диспетчеризации событий* (EDT).

Однопоточные структуры GUI относятся не только к Java. Структуры Qt, NextStep, MacOS, Cocoa, X Windows и многие другие тоже являются однопоточными. Попыток написать многопоточные структуры GUI было много, но прийти к успеху мешали состояния гонки и взаимная блокировка.

Существует проблема взаимодействия между обработкой входных событий и объектно-ориентированным моделированием компонентов GUI. Действия, инициируемые пользователем, направляются из операционной системы в приложение, а инициируемые приложением действия движутся в обратном направлении. Желание сделать объекты потокобезопасными приводило разработчиков к противоречивому упорядочиванию замков и возникновению взаимной блокировки (см. главу 10).

В многопоточных структурах GUI преобладает паттерн «модель, представление, контроллер», который повышает риск противоречивого упорядочивания замков. Контроллер обращается к модели, которая уведомляет представление о том, что что-то изменилось. Но контроллер также может обратиться к представлению, которое, в свою очередь, может обратиться к модели для запроса об ее состоянии.

В своем блоге[1] вице-президент компании Sun Грэм Гамильтон резюмирует эти сложности:

> *Я считаю, что успешно программировать с помощью многопоточных инструментов GUI можно, если этот инструмент очень тщательно спроектирован, предоставляет свою замковую методику в мельчайших подробностях, а вы — очень умный, очень осторожный и имеете глобальное понимание всей структуры инструмента. Иначе вы получите случайные зависания (из-за взаимной блокировки) или аппаратные сбои (из-за состояний гонки). Многопоточный подход лучше всего подойдет людям, которые принимали непосредственное участие в разработке инструмента.*

> *К сожалению, я не думаю, что он годится для широкого коммерческого использования. То, с чем в конечном счете вы останетесь, — это приложения, работающие не совсем надежно по причинам, которые не очевидны. И вы будете разочарованы.*

[1] См. http://weblogs.java.net/blog/kgh/archive/2004/10

Поэтому все объекты GUI, включая визуальные компоненты и модели данных, доступны исключительно из событийного потока. Разумеется, часть бремени потокобезопасности ложится на разработчика приложения, который должен убедиться, что эти объекты ограничены должным образом.

9.1.1. Последовательная обработка событий

GUI-приложения ориентированы на обработку *событий* (events), таких как щелчки мыши, нажатия клавиш или истечения таймера. События — это своего рода задачи, механизм обработки которых похож на работу исполнителя `Executor`.

Поскольку для обработки задач GUI существует только один поток, они обрабатываются последовательно — одна задача завершается до начала следующей.

Недостатком последовательной обработки задач является то, что если выполнение одной задачи занимает много времени, то другие задачи должны ожидать ее завершения. Поэтому длительную задачу лучше выполнять в другом потоке, чтобы управление смогло быстро вернуться в событийный поток.

9.1.2. Ограничение одним потоком в Swing

Все компоненты Swing (такие как `JButton` и `JTable`) и объекты модели данных (такие как `TableModel` и `TreeModel`) ограничены событийным потоком, поэтому любой код, который обращается к этим объектам, должен работать в событийном потоке.

> Правило однопоточности: компоненты и модели Swing должны создаваться, модифицироваться и запрашиваться только из потока диспетчеризации событий.

У данного правила есть исключения. Несколько методов Swing могут вызываться безопасно из любого потока, поскольку определены в Javadoc как

потокобезопасные. Приведем другие методы, которые могут вызываться не из событийного потока:

- метод `SwingUtilities.isEventDispatchThread` определяет, является или нет текущий поток событийным;
- метод `SwingUtilities.invokeLater` планирует объект `Runnable` для выполнения в событийном потоке;
- метод `SwingUtilities.invokeAndWait` планирует задачу `Runnable` для выполнения в событийном потоке и блокирует текущий поток, до тех пор пока задача не завершится (вызывается *только* из потока, не являющегося GUI);
- методы размещения в событийной очереди запроса на перерисовку или повторную валидацию;
- методы добавления и удаления слушателей (слушатели будут активироваться в событийном потоке).

Методы `invokeLater` и `invokeAndWait` функционируют во многом как исполнитель `Executor`. На самом деле, как показано в листинге 9.1, методы из класса `SwingUtilities`, связанные с поточной обработкой, реализуются с помощью однопоточного исполнителя `Executor`.

Событийный поток Swing можно рассматривать как однопоточный исполнитель, обрабатывающий из событийной очереди задачи, которые видят, как рабочие потоки умирают и заменяются новыми. Последовательное однопоточное выполнение является разумной политикой выполнения, когда задачи скоротечны, прогнозируемость планирования не важна либо необходимо, чтобы задачи не выполнялись конкурентно.

Класс `GuiExecutor` в листинге 9.2 является исполнителем, который делегирует выполнение задач классу `SwingUtilities`. Он также может быть реализован в терминах других структур GUI: например, SWT обеспечивает метод `Display.asyncExec`, который подобен методу Swing `invokeLater`.

9.2. Кратковременные задачи GUI

Обработка кратковременных задач может происходить в событийном потоке, тогда как часть обработки длительных задач должна быть выгружена в другой поток.

В листинге 9.3 создается кнопка, цвет которой изменяется случайным образом при нажатии. Когда пользователь нажимает на кнопку, инструментарий доставляет событие `ActionEvent` в событийном потоке всем зарегистрированным слушателям действий. В ответ слушатели изменяют фоновый цвет кнопки. Таким образом, событие возникает в инструментарии GUI и доставляется приложению, и приложение модифицирует GUI-интерфейс в ответ на действие пользователя. Контроль никогда не покидает событийный поток (рис. 9.1).

Этот пример характеризует большинство взаимодействий между GUI-приложениями и инструментами GUI.

Листинг 9.1. Реализация SwingUtilities с использованием Executor

```
public class SwingUtilities {
    private static final ExecutorService exec =
        Executors.newSingleThreadExecutor(new SwingThreadFactory());
    private static volatile Thread swingThread;

    private static class SwingThreadFactory implements ThreadFactory {
        public Thread newThread(Runnable r) {
            swingThread = new Thread(r);
            return swingThread;
        }
    }

    public static boolean isEventDispatchThread() {
        return Thread.currentThread() == swingThread;
    }

    public static void invokeLater(Runnable task) {
        exec.execute(task);
    }

    public static void invokeAndWait(Runnable task)
            throws InterruptedException, InvocationTargetException {
        Future f = exec.submit(task);
        try {
            f.get();
        } catch (ExecutionException e) {
            throw new InvocationTargetException(e);
        }
    }
}
```

Листинг 9.2. Исполнитель Executor построен поверх SwingUtilities

```
public class GuiExecutor extends AbstractExecutorService {
    // Синглтоны имеют приватный конструктор и публичную фабрику
    private static final GuiExecutor instance = new GuiExecutor();

    private GuiExecutor() { }

    public static GuiExecutor instance() { return instance; }

    public void execute(Runnable r) {
        if (SwingUtilities.isEventDispatchThread())
            r.run();
        else
            SwingUtilities.invokeLater(r);
    }
    // Плюс тривиальные реализации методов жизненного цикла
}
```

Листинг 9.3. Простой слушатель событий

```
final Random random = new Random();
final JButton button = new JButton("Change Color");
...
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        button.setBackground(new Color(random.nextInt()));
    }
});
```

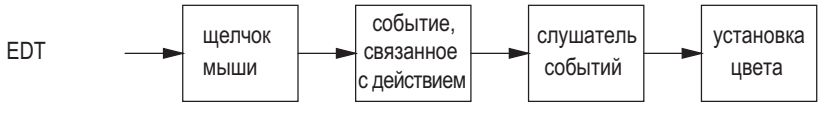

Рис. 9.1. Поток управления простым щелчком кнопки

Более сложная версия этого сценария, показанная на рис. 9.2, предусматривает использование формальной модели данных, такой как `TableModel` или `TreeModel`. Swing разделяет большинство визуальных компонентов на два объекта: модель и представление. Данные, подлежащие выводу на экран, находятся в модели, а правила, регулирующие их показ, — в представлении. Модельные объекты могут запускать события, говорящие об изменении модельных данных, и представления могут подписываться на эти события. Когда представление получит событие, говорящее о том,

что модельные данные могли измениться, оно запросит у модели новые данные и обновит экран. Таким образом, в слушателе кнопок, который модифицирует содержимое таблицы, слушатель действий обновит модель и вызовет один из методов `fireXxx`, который, в свою очередь, вызовет находящихся в представлении слушателей табличной модели, которые и обновят экран. (Методы модели данных Swing `fireXxx` всегда вызывают слушателей модели напрямую, а не отправляют новое событие в событийную очередь, поэтому методы `fireXxx` должны вызываться только в событийном потоке.)

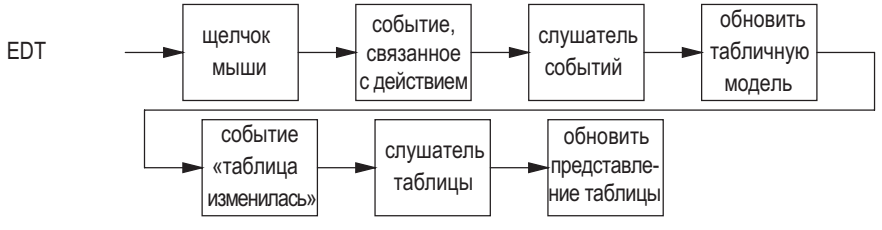

Рис. 9.2. Поток управления с отдельными объектами модели и представления

9.3. Длительные задачи GUI

Сложные GUI-приложения могут выполнять длительные задачи в другом потоке, чтобы графический интерфейс оставался отзывчивым во время их работы.

Swing упрощает работу задачи в событийном потоке, но (до Java 6) не предоставляет механизма, помогающего задачам GUI выполнять код в других потоках. Мы можем создать собственный исполнитель, например кэшированный пул потоков: GUI-приложения инициируют большое число длительных задач крайне редко, поэтому риск того, что пул будет расти неограниченно, крайне мал.

Начнем с простых задач, которые не поддерживают отмену или индикацию хода выполнения и не обновляют GUI-интерфейс по завершению. В листинге 9.4 показан слушатель действий, привязанный к визуальному компоненту, который предоставляет длительную задачу исполнителю. Несмотря на два уровня внутренних классов, побудить GUI инициировать задачу таким образом довольно просто: слушатель действий пользователь-

ского интерфейса вызывается в событийном потоке и отправляет объект `Runnable` для выполнения в пуле потоков.

Так мы извлекаем длительную задачу из событийного потока, используя прием «запустить и забыть», который, вероятно, не очень полезен. Обычно после завершения длительной задачи возникает своего рода визуальная обратная связь. Но вы не можете обратиться к объектам представления из фонового потока, поэтому по завершении задача должна предложить еще одну задачу для запуска в событийном потоке, чтобы обновить пользовательский интерфейс.

Листинг 9.4. Привязка длительной задачи к визуальному компоненту

```
ExecutorService backgroundExec = Executors.newCachedThreadPool();
...
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        backgroundExec.execute(new Runnable() {
            public void run() { doBigComputation(); }
        });
}});
```

Листинг 9.5 иллюстрирует способ добавления задачи, но с тремя уровнями внутренних классов. Слушатель действий сначала приглушает кнопку и задает надпись, указывающую на то, что вычисление находится в процессе работы, а затем предоставляет задачу фоновому исполнителю. Когда эта задача завершается, он помещает в очередь еще одну задачу для выполнения в событийном потоке, которая активирует кнопку заново и восстанавливает текст надписи.

Листинг 9.5. Длительная задача с обратной связью пользователя

```
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        button.setEnabled(false);
        label.setText("busy");
        backgroundExec.execute(new Runnable() {
            public void run() {
                try {
                    doBigComputation();
                } finally {
                    GuiExecutor.instance().execute(new Runnable() {
                        public void run() {
                            button.setEnabled(true);
```

```
                        label.setText("idle");
                    }
                });
            }
        }
    });
}
});
```

Задача, запускаемая при нажатии кнопки, состоит из трех последовательных подзадач, выполнение которых перемежается между событийным и фоновым потоками. Первая подзадача обновляет пользовательский интерфейс, показывая, что началась длительная операция, и запускает вторую подзадачу в фоновом потоке. По завершении вторая подзадача помещает в очередь третью подзадачу, для того чтобы она работала в событийном потоке, тем самым обновляя пользовательский интерфейс и отражая завершение операции. Этот вид «перескакивания от потока к потоку» типичен для обработки длительных задач в GUI-приложениях.

9.3.1. Отмена

Вы можете отменить задачу с помощью прерывания потока, но гораздо проще использовать `Future`, который был спроектирован для управления отменяемыми задачами.

При вызове метода `cancel` в `Future` с параметром `mayInterruptIfRunning`, равным `true`, реализация `Future` прерывает поток, выполняющий задачу, если он работает в данный момент. Если ваша задача откликается на прерывание, то она может вернуться досрочно в случае отмены. В листинге 9.6 показана задача, которая опрашивает статус прерванности потока и возвращается досрочно по прерыванию.

Листинг 9.6. Отмена длительной задачи

```
Future<?> runningTask = null; // ограниченная одним потоком
...
startButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        if (runningTask != null) {
            runningTask = backgroundExec.submit(new Runnable() {
                public void run() {
                    while (moreWork()) {
                        if (Thread.currentThread().isInterrupted()) {
```

продолжение ⇩

Листинг 9.6 *(продолжение)*

```
                            cleanUpPartialWork();
                            break;
                        }
                        doSomeWork();
                    }
                }
            });
    };
}});

cancelButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent event) {
        if (runningTask != null)
            runningTask.cancel(true);
}});
```

Поскольку задача `runningTask` ограничена событийным потоком, при ее настройке или проверке синхронизации не требуется, а слушатель кнопки пуска обеспечивает, чтобы выполнялась только одна фоновая задача за раз. Однако было бы еще лучше получать уведомления о завершении задачи, чтобы, например, отключить кнопку «Отмена». Мы рассмотрим это в следующем разделе.

9.3.2. Индикация хода выполнения и завершения

Использование `Future` для представления длительной задачи упростило реализацию отмены. `FutureTask` имеет перехватчика `done`, который способствует уведомлению об отмене (он вызывается после завершения фонового объекта `Callable`). Давая перехватчику `done` запустить задачу завершения в событийном потоке, мы можем сконструировать класс `BackgroundTask`, предоставляющий перехватчик `onCompletion`, вызываемый в событийном потоке, как показано в листинге 9.7.

Класс `BackgroundTask` также поддерживает индикацию хода выполнения. Метод `compute` может вызывать метод `setProgress`, показывающий ход выполнения в числовом выражении. Это приводит к вызову из событийного потока метода `onProgress`, который может обновлять пользовательский интерфейс для визуального информирования о ходе работы.

Для реализации `BackgroundTask` необходимо только реализовать метод `compute`, который вызывается в фоновом потоке. У вас также есть воз-

можность переопределить методы `onCompletion` и `onProgress`, которые активируются в событийном потоке.

Базирование класса `BackgroundTask` на `FutureTask` также упрощает отмену. Вместо того чтобы опрашивать статус прерванности потока, метод `compute` может вызывать метод `Future.isCancelled`. Листинг 9.8 дополняет пример из листинга 9.6 классом `BackgroundTask`.

9.3.3. SwingWorker

Мы построили простую структуру, используя `FutureTask` и `Executor`, которая может применяться к любой однопоточной структуре GUI, а не только к платформе Swing. В Swing многие функциональные элементы предоставляются классом `SwingWorker`, включая отмену, уведомление о завершении и индикацию хода выполнения. Различные версии класса `SwingWorker` были опубликованы в книгах *The Swing Connection* («Взаимодействие с платформой Swing») и *The Java Tutorial* («Учебное руководство по Java»), а его обновленная версия включена в Java 6.

9.4. Совместные модели данных

Объекты представления Swing, включая объекты модели данных, такие как `TableModel` или `TreeModel`, ограничены событийным потоком. В простых GUI-программах все мутируемое состояние хранится в объектах представления, а единственным потоком, кроме событийного, является главный поток. В этих программах легко обеспечивать соблюдение правила однопоточности: не обращаться к модели данных или компонентам представления из главного потока. Более сложные программы могут использовать другие потоки для перемещения данных в постоянное хранилище или из него, например в файловую систему или базу данных, для того чтобы не ставить под угрозу отзывчивость.

Данные вводятся в модель данных пользователем или статически загружаются из файла или другого источника данных при запуске приложения, то есть не затрагиваются только событийным потоком. Но иногда объектом модели представления является вид на другой источник данных, например базу данных, файловую систему или удаленную службу. В этом случае данные могут затрагиваться более чем одним потоком на входе или выходе из приложения.

Листинг 9.7. Класс фоновой задачи, поддерживающий отмену, уведомление о завершении и уведомление о ходе выполнения

```
abstract class BackgroundTask<V> implements Runnable, Future<V> {
    private final FutureTask<V> computation = new Computation();

    private class Computation extends FutureTask<V> {
        public Computation() {
            super(new Callable<V>() {
                public V call() throws Exception {
                    return BackgroundTask.this.compute();
            }
        });
    }
    protected final void done() {
        GuiExecutor.instance().execute(new Runnable() {
            public void run() {
                V value = null;
                Throwable thrown = null;
                boolean cancelled = false;
                try {
                    value = get();
                } catch (ExecutionException e) {
                    thrown = e.getCause();
                } catch (CancellationException e) {
                    cancelled = true;
                } catch (InterruptedException consumed) {
                } finally {
                    onCompletion(value, thrown, cancelled);
                }
            };
        });
        }
    }
    protected void setProgress(final int current, final int max) {
        GuiExecutor.instance().execute(new Runnable() {
            public void run() { onProgress(current, max); }
        });
    }
    // Вызывается в фоновом потоке
    protected abstract V compute() throws Exception;
    // Вызывается в событийном потоке
    protected void onCompletion(V result, Throwable exception,
                                boolean cancelled) { }
    protected void onProgress(int current, int max) { }
    // Другие методы Future, переадресованные для вычисления
}
```

Листинг 9.8. Инициирование длительной отменяемой задачи с помощью BackgroundTask

```
public void runInBackground(final Runnable task) {
    startButton.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            class CancelListener implements ActionListener {
                BackgroundTask<?> task;
                public void actionPerformed(ActionEvent event) {
                    if (task != null)
                        task.cancel(true);
                }
            }
            final CancelListener listener = new CancelListener();
            listener.task = new BackgroundTask<Void>() {
                public Void compute() {
                    while (moreWork() && !isCancelled())
                        doSomeWork();
                    return null;
                }
                public void onCompletion(boolean cancelled, String s,
                                         Throwable exception) {
                    cancelButton.removeActionListener(listener);
                    label.setText("done");
                }
            };
            cancelButton.addActionListener(listener);
            backgroundExec.execute(task);
        }
    });
}
```

Например, выводя на экран содержимое удаленной файловой системы с помощью древовидного элемента управления, вы не хотите перечислять всю файловую систему перед выводом древовидного элемента управления на экран. Вместо этого дерево может лениво заполняться по мере раскрытия узлов. Перечисление даже одного каталога на удаленном томе может занять много времени, поэтому вам может потребоваться выполнять перечисление в фоновой задаче. После завершения фоновой задачи можно переместить данные в событийный поток потокобезопасной древовидной модели с помощью публикации задачи методом `invokeLater` или опроса событийного потока о наличии данных.

9.4.1. Потокобезопасные модели данных

Если модель данных поддерживает конкурентность, то событийный поток и фоновые потоки должны иметь возможность совместной работы без проблем с отзывчивостью. Потокобезопасные модели данных должны генерировать события при обновлении модели, чтобы представления могли обновляться при изменении данных.

Иногда появляется возможность получить потокобезопасность, непротиворечивость и хорошую отзывчивость с помощью *версионной модели данных*, такой как `CopyOnWriteArrayList` [CPJ 2.2.3.3]. Когда вы приобретаете итератор для коллекции «копировать при записи», он обходит коллекцию в том виде, в каком она существовала при его создании. Однако коллекции «копировать при записи» обеспечивают хорошую производительность только в том случае, если число проходов превышает число изменений, что не относится, например, к приложению с отслеживанием такси. Более специализированные версионные структуры данных могут избежать этого ограничения, но создавать версионные структуры данных, которые обеспечивают эффективный конкурентный доступ и не сохраняют старые версии данных дольше, чем это необходимо, нелегко, и поэтому их следует рассматривать только тогда, когда другие подходы не практичны.

9.4.2. Раздвоенные модели данных

С точки зрения GUI классы табличной модели Swing, такие как `TableModel` и `TreeModel`, являются официальным репозиторием для данных, подлежащих выводу на экран. Однако эти модельные объекты часто сами являются представлениями других объектов, управляемых приложением. Считается, что программа, которая имеет модель данных с областью представления и областью приложения, имеет паттерн *раздвоенной модели* (split-model, Fowler, 2005).

В этом паттерне модель представления ограничена событийным потоком, а *совместная модель* является потокобезопасной и может оцениваться как событийным потоком, так и потоками приложения. Модель представления регистрирует слушателей в совместной модели, давая ей возможность получать уведомления об обновлениях, после чего сама может обновляться из совместной модели путем встраивания снимка релевантного состояния в сообщение об обновлении.

Подход на основе снимка прост, но ограничен. Он хорошо работает, когда модель данных мала, обновления не слишком часты и структуры двух моделей подобны друг другу. Если же модель данных является большой, или обновления очень часты, или хотя бы одна сторона паттерна содержит информацию, которая не видна другой стороне, то эффективнее сработает отправка инкрементных обновлений вместо целых снимков. Этот подход позволит сериализовать обновления в совместно используемой модели и воссоздавать их в событийном потоке для модели представления. Еще одним преимуществом инкрементных обновлений является то, что более детальная информация об изменениях улучшает их восприятие — если движется только один автомобиль, то нам нужно перерисовывать не весь экран, а только затронутые участки.

> Рассматривайте применение паттерна раздвоенной модели, когда модель данных должна использоваться несколькими потоками совместно, а реализация потокобезопасной модели данных нецелесообразна по причинам блокирования, непротиворечивости или сложности.

9.5. Другие формы однопоточных подсистем

Иногда ограничение одним потоком навязывается разработчику по причинам, которые не связаны с потребностью избежать синхронизации или взаимной блокировки. Например, некоторые нативные библиотеки требуют, чтобы весь доступ к ним, даже загрузка с помощью `System.loadLibrary`, осуществлялся из одного потока.

Заимствуя подход, применяемый структурами GUI, можно легко создавать выделенный поток или однопоточный исполнитель для доступа к нативной библиотеке и предоставлять объект-посредник, который перехватывает вызовы ограниченного одним потоком объекта и предоставляет их выделенному потоку в виде задач. Для этого `Future` и `newSingleThreadExecutor` работают вместе. Метод-посредник может предоставлять (`submit`) задачу и немедленно вызывать метод `Future.get`, для того чтобы ожидать результата. (Если класс, который ограничен одним потоком, реализует интерфейс, то вы можете автоматизировать процесс, в котором каждый метод

предоставляет объект `Callable` исполнителю фонового потока и ожидает результата, используя динамические посредники.)

Итоги

Структуры GUI почти всегда реализуются как однопоточные подсистемы, в которых весь код, связанный с представлением, работает в виде задач в событийном потоке. Длительные задачи могут отрицательно повлиять на отзывчивость и поэтому должны выполняться в фоновых потоках. Построенные здесь вспомогательные классы, такие как класс `SwingWorker` или `BackgroundTask`, которые обеспечивают поддержку отмены, индикации хода выполнения и индикации завершения, могут упростить разработку длительных задач.

Часть III
Жизнеспособность, производительность и тестирование

10 Предотвращение сбоев жизнеспособности

Нередко между безопасностью и жизнеспособностью возникает напряжение. Мы используем блокировку для обеспечения потокобезопасности, но беспорядочное использование блокировки может привести к *возникновению взаимной блокировки из-за порядка блокировки* (lock-ordering deadlock). Схожим образом мы используем пулы потоков и семафоры для ограничения ресурсопотребления, но неспособность понять ограничиваемые действия может вызвать *ресурсные взаимные блокировки* (resource deadlock). После возникновения взаимной блокировки приложения Java не восстанавливаются, поэтому стоит позаботиться, чтобы ваш проект исключал условия, которые могли бы вызывать такую ситуацию. В этой главе рассматриваются некоторые из причин сбоев жизнеспособности и что можно сделать, чтобы их предотвратить.

10.1. Взаимная блокировка

Взаимная блокировка (deadlock) иллюстрируется классической, хотя и несколько антисанитарной, задачей «обедающих философов». Пять философов собираются поесть китайской еды и садятся за круглый стол. На столе лежат пять палочек для еды (не пять пар), по одной между каждой парой посетителей. Философы чередуют размышление с приемом пищи. Каждый должен владеть двумя палочками достаточно долго, чтобы поесть, но затем может положить палочки обратно и вернуться к размышлению.

Существует несколько алгоритмов управления палочками, которые позволяют каждому есть на более-менее регулярной основе (голодный философ хватает обе соседние палочки, но если не хватает одной свободной, то кладет вторую назад и ждет минуту или около того, прежде чем попытаться снова). Некоторые из этих алгоритмов могут привести к тому, что некоторые или все философы умрут от голода (каждый философ тут же хватает палочку слева от него и ждет, когда освободится палочка справа, прежде чем положить левую). Последняя ситуация, когда каждый из них имеет ресурс, необходимый другому, и ждет ресурса, занимаемого другим, и не высвобождает ресурс, который он занимает сам, до тех пор пока не приобретет ресурс, которого у него нет, и иллюстрирует ситуацию взаимной блокировки.

Когда поток владеет замком вечно, другие потоки, пытающиеся приобрести этот замок, будут блокированы, находясь в бесконечном ожидании. Когда поток *A* владеет замком *L* и пытается приобрести замок *M*, но в то же время поток *B* владеет *M* и пытается приобрести *L*, то *оба* потока будут ждать вечно. Эта ситуация является простейшим случаем взаимной блокировки, в котором многочисленные потоки ждут вечно из-за циклической замковой зависимости. (Считайте потоки узлами ориентированного графа, ребра которого представляют связь «Поток *A* ожидает ресурс, занятый потоком *B*». Если этот граф является циклическим, то возникает взаимная блокировка.)

Системы баз данных спроектированы так, чтобы обнаруживать и восстанавливаться от взаимной блокировки. Транзакция может приобрести много замков, и замки удерживаются до фиксации транзакции. Поэтому вполне возможно, и на самом деле это не редкость, что две транзакции будут заперты взаимной блокировкой. Без вмешательства они будут ждать вечно (удерживая замки, которые, вероятно, потребуются для других транзакций). Но сервер базы данных этого не допускает. Когда он обнаруживает, что набор транзакций заперт взаимной блокировкой (что он делает путем поиска циклов в графе транзакций, *находящихся в ожидании*), он выбирает жертву и прерывает эту транзакцию. Это освобождает замки, занимаемые жертвой, позволяя другим транзакциям продолжиться. Приложение затем может повторить прерванную транзакцию, которая теперь может завершиться после того, как все конфликтующие транзакции завершились.

JVM не так полезна в разрешении проблем взаимной блокировки, как серверы баз данных. Когда набор потоков Java запирается взаимной

блокировкой, это означает конец игры — эти потоки выходят из строя навсегда. В зависимости от того, что делают эти потоки, может полностью застопориться приложение, определенная подсистема либо пострадать производительность. Единственный способ восстановить работоспособность приложения — прервать его и перезапустить и надеяться, что все это не повторится.

Как и многие другие сбои конкурентности, взаимные блокировки редко проявляются сразу. Тот факт, что в классе есть потенциальная взаимная блокировка, не значит, что класс обязательно *будет* им заперт, просто существует такая возможность. Когда взаимные блокировки все-таки себя проявляют, то это зачастую происходит в самый неподходящий момент — под большой производственной нагрузкой.

10.1.1. Взаимные блокировки из-за порядка блокировки

Класс `LeftRightDeadlock` в листинге 10.1 рискует быть запертым взаимной блокировкой. Оба метода, `leftRight` и `rightLeft`, приобретают левый и правый замки. Если один поток вызывает метод `leftRight`, а другой вызывает метод `rightLeft` и их действия перемежаются, как показано на рис. 10.1, то они будут заперты взаимной блокировкой.

Взаимная блокировка в `LeftRightDeadlock` возникла, потому что два потока попытались приобрести одинаковые замки в *разном порядке*. Если бы они запросили замки в том же самом порядке, то не было бы никакой циклической замковой зависимости и, следовательно, никакой взаимной блокировки. Если вы можете обеспечить, чтобы каждый поток, которому нужны замки L и M в одно и то же время, всегда приобретал L и M в том же самом порядке, то никакой взаимной блокировки не будет.

> Программа будет свободна от возникновения взаимной блокировки, возникающей из-за порядка блокировки, если все потоки приобретают замки, в которых они нуждаются, в фиксированном глобальном порядке.

Верификация непротиворечивого порядка блокировки требует глобального анализа замкового поведения вашей программы. Недостаточно изучить

ветви кода, которые приобретают многочисленные замки индивидуально; и `leftRight`, и `rightLeft` являются «разумными» способами приобретения двух замков, они просто несовместимы. Когда дело доходит до замковой защиты, левая рука должна знать, что делает правая.

Листинг 10.1. Простая взаимная блокировка из-за порядка блокировки. *Так делать не следует*

```
// Предупреждение: предрасположен к возникновению взаимной блокировки!
public class LeftRightDeadlock {
    private final Object left = new Object();
    private final Object right = new Object();

    public void leftRight() {
        synchronized (left) {
            synchronized (right) {
                doSomething();
            }
        }
    }

    public void rightLeft() {
        synchronized (right) {
            synchronized (left) {
                doSomethingElse();
            }
        }
    }
}
```

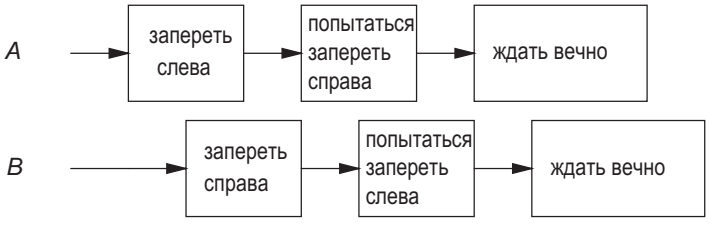

Рис. 10.1. Неудачная временная координация в классе LeftRightDeadlock

10.1.2. Взаимная блокировка из-за динамического порядка следования замков

Иногда не очевидно, что вы располагаете достаточным контролем над порядком блокировки, для того чтобы предотвратить взаимные блокировки. Рассмотрим безобидный код в листинге 10.2, который переводит средства с одного счета на другой. Перед выполнением перевода он приобретает замки на обоих объектах `Account`, обеспечивая, чтобы остатки обновлялись атомарно и без нарушения инвариантов, таких как «счет не может иметь отрицательный остаток».

Листинг 10.2. Взаимная блокировка из-за динамического порядка следования замков. *Так делать не следует*.

```
// Предупреждение: предрасположен к возникновению взаимной блокировки!
public void transferMoney(Account fromAccount,
                          Account toAccount,
                          DollarAmount amount)
        throws InsufficientFundsException {
    synchronized (fromAccount) {
        synchronized (toAccount) {
            if (fromAccount.getBalance().compareTo(amount) < 0)
                throw new InsufficientFundsException();
            else {
                fromAccount.debit(amount);
                toAccount.credit(amount);
            }
        }
    }
}
```

Каким образом метод `transferMoney` может быть заперт взаимной блокировкой? Может показаться, что все потоки приобретают свои замки в одном и том же порядке, но на самом деле порядок следования замков зависит от порядка следования аргументов, передаваемых методу `transferMoney`, и они, в свою очередь, могут зависеть от внешних входных данных. Взаимная блокировка может возникнуть, если два потока одно-

временно вызывают метод transferMoney, один из которых переводит со счета *X* на счет *Y*, а другой — наоборот:

```
A: transferMoney(myAccount, yourAccount, 10);
B: transferMoney(yourAccount, myAccount, 20);
```

При неудачной временной координации *A* приобретет замок на счете myAccount и будет ожидать замок на счете yourAccount, в то время как *B* владеет замком на yourAccount и ожидает замок на счете myAccount.

Взаимные блокировки подобного рода можно обнаружить так же, как и в листинге 10.1, — отыскать приобретения вложенных замков. Поскольку порядок следования аргументов выходит из-под нашего контроля, то для того, чтобы решить эту проблему, мы должны *индуцировать* порядок блокировки и приобретать их в соответствии с индуцированным порядком единообразно во всем приложении.

Один из способов индуцирования порядка на объектах — использовать метод System.identityHashCode, возвращающий значение, которое будет возвращено методом Object.hashCode. В листинге 10.3 показана версия метода transferMoney, использующая метод System.identityHashCode для индуцирования порядка блокировки. Он задействует несколько дополнительных строк кода, но исключает возможность взаимной блокировки.

В редких случаях, когда два объекта имеют один и тот же хеш-код, мы должны использовать произвольные средства порядка блокировки, и это вновь создает возможность возникновения взаимной блокировки. Для того чтобы предотвратить противоречивый порядок блокировки в этом случае, используется третий замок, который «разрывает связь». Приобретая замок разрыва связи до приобретения любого из замков Account, мы обеспечиваем, чтобы только один поток за раз выполнял рискованную задачу приобретения двух замков в произвольном порядке, устраняя возможность возникновения взаимной блокировки (при условии что этот механизм используется непротиворечиво). Если бы хеш-коллизии были бы распространены, то это техническое решение могло бы стать узким местом конкурентности (так же как наличие единственного общепрограммного замка), но поскольку хеш-коллизии благодаря методу System.identityHashCode исчезающе редки, то данное техническое решение обеспечивает последнюю каплю безопасности при небольшой стоимости.

Листинг 10.3. Создание порядка блокировки, чтобы избежать взаимоблокировки

```
private static final Object tieLock = new Object();

public void transferMoney(final Account fromAcct,
                          final Account toAcct,
                          final DollarAmount amount)
        throws InsufficientFundsException {
    class Helper {
        public void transfer() throws InsufficientFundsException {
            if (fromAcct.getBalance().compareTo(amount) < 0)
                throw new InsufficientFundsException();
            else {
                fromAcct.debit(amount);
                toAcct.credit(amount);
            }
        }
    }
    int fromHash = System.identityHashCode(fromAcct);
    int toHash = System.identityHashCode(toAcct);

    if (fromHash < toHash) {
        synchronized (fromAcct) {
            synchronized (toAcct) {
                new Helper().transfer();
            }
        }
    } else if (fromHash > toHash) {
        synchronized (toAcct) {
            synchronized (fromAcct) {
                new Helper().transfer();
            }
        }
    } else {
        synchronized (tieLock) {
            synchronized (fromAcct) {
                synchronized (toAcct) {
                    new Helper().transfer();
                }
            }
        }
    }
}
```

Если счет `Account` имеет уникальный немутируемый сопоставимый ключ, такой как номер счета, создать порядок блокировки еще проще: упорядо-

чить объекты по их ключу, тем самым устраняя необходимость в замке разрыва связи.

Вы можете подумать, что мы завышаем риск возникновения взаимной блокировки, потому что замками обычно владеют недолго, однако взаимные блокировки являются серьезной проблемой в реальных системах. Производственное приложение может выполнять миллиарды циклов приобретения-освобождения замков в день. И только одному из них нужно оказаться плохо скоординированным по времени, чтобы запереть приложение взаимной блокировкой, и даже тщательный режим нагрузочного тестирования не способен раскрыть все потайные взаимные блокировки[1]. Класс `DemonstrateDeadlock` в листинге 10.4[2] запирается взаимной блокировкой довольно быстро на большинстве систем.

Листинг 10.4. Цикл драйвера, который вызывает взаимоблокировку в типичных условиях

```
public class DemonstrateDeadlock {
    private static final int NUM_THREADS = 20;
    private static final int NUM_ACCOUNTS = 5;
    private static final int NUM_ITERATIONS = 1000000;

    public static void main(String[] args) {
        final Random rnd = new Random();
        final Account[] accounts = new Account[NUM_ACCOUNTS];

        for (int i = 0; i < accounts.length; i++)
            accounts[i] = new Account();

        class TransferThread extends Thread {
            public void run() {
                for (int i=0; i<NUM_ITERATIONS; i++) {
                    int fromAcct = rnd.nextInt(NUM_ACCOUNTS);
                    int toAcct = rnd.nextInt(NUM_ACCOUNTS);
                    DollarAmount amount =
                        new DollarAmount(rnd.nextInt(1000));
```

продолжение ⇩

[1] По иронии судьбы, удержание замков в течение коротких периодов времени, которое вы предположительно должны делать с целью уменьшить конфликт блокировки, увеличивает вероятность того, что тестирование не будет раскрывать скрытые риски взаимной блокировки.

[2] В целях упрощения `DemonstrateDeadlock` игнорирует вопрос отрицательных остатков на счетах.

Листинг 10.4 *(продолжение)*

```
                    transferMoney(accounts[fromAcct],
                                  accounts[toAcct], amount);
                }
            }
        }
        for (int i = 0; i < NUM_THREADS; i++)
            new TransferThread().start();
    }
}
```

10.1.3. Взаимные блокировки между взаимодействующими объектами

Многочисленное замковое приобретение не всегда так очевидно, как в `LeftRightDeadlock` или `transferMoney`; два замка не обязательно должны быть приобретены одним и тем же методом. Рассмотрим взаимодействующие классы в листинге 10.5, которые могут использоваться в приложении диспетчеризации такси. Класс `Taxi` представляет отдельное такси с местоположением и пунктом назначения, класс `Dispatcher` представляет парк такси.

Листинг 10.5. Взаимная блокировка из-за порядка блокировки между взаимодействующими объектами. *Так делать не следует*

```
// Предупреждение: предрасположен к возникновению взаимной блокировки!
class Taxi {
    @GuardedBy("this") private Point location, destination;
    private final Dispatcher dispatcher;

    public Taxi(Dispatcher dispatcher) {
        this.dispatcher = dispatcher;
    }

    public synchronized Point getLocation() {
        return location;
    }

    public synchronized void setLocation(Point location) {
```

```
        this.location = location;
        if (location.equals(destination))
            dispatcher.notifyAvailable(this);
    }
}

class Dispatcher {
    @GuardedBy("this") private final Set<Taxi> taxis;
    @GuardedBy("this") private final Set<Taxi> availableTaxis;

    public Dispatcher() {
        taxis = new HashSet<Taxi>();
        availableTaxis = new HashSet<Taxi>();
    }

    public synchronized void notifyAvailable(Taxi taxi) {
        availableTaxis.add(taxi);
    }

    public synchronized Image getImage() {
        Image image = new Image();
        for (Taxi t : taxis)
            image.drawMarker(t.getLocation());
        return image;
    }
}
```

Хотя ни один метод не приобретает два замка *явным* образом, элементы кода, вызывающие методы `setLocation` и `getImage`, все равно могут приобрести два замка. Если поток вызывает метод `setLocation` в ответ на обновление, поступающее от приемника GPS, то он сначала обновляет местоположение такси, а затем проверяет, достигло или нет такси пункта назначения. Если да, то он сообщает диспетчеру, что ему нужен новый пункт назначения. Поскольку оба метода, `setLocation` и `notifyAvailable`, синхронизированы, поток, вызывающий метод `setLocation`, приобретает замок `Taxi` и затем замок `Dispatcher`. Схожим образом поток, вызывающий метод `getImage`, приобретает замок `Dispatcher`, а затем каждый замок `Taxi` (один за раз). Точно так же, как в `LeftRightDeadlock`, два замка приобретаются двумя потоками в разных порядках, рискуя быть запертыми взаимной блокировкой.

В методах `LeftRightDeadlock` или `transferMoney` легко было засечь возможность возникновения взаимной блокировки, проведя поиск методов, которые приобретают два замка. Обнаружить возможность возникновения

взаимной блокировки в `Taxi` и `Dispatcher` немного сложнее: предупреждающим признаком является то, что *чужой* метод вызывается, удерживая замок.

> Активировать чужой метод с удержанием замка — значит нарываться на неприятности с жизнеспособностью. Чужой метод может приобретать другие замки (рискуя быть запертым взаимной блокировкой) или блокировать неожиданно долго, застопоривая другие потоки, нуждающиеся в замке, которым вы владеете.

10.1.4. Открытые вызовы

Разумеется, классы `Taxi` и `Dispatcher` не знали, что каждый из них являлся половиной взаимной блокировки, ожидающей своего часа. И они не должны этого делать; вызов метода является барьером-абстракцией, в чьи обязанности вменено защищать вас от деталей того, что происходит на другой стороне. Но поскольку вы не знаете, что происходит на другой стороне вызова, *вызов чужого метода с удержанием замка трудно анализируется и, следовательно, является рискованным*.

Вызов метода без удерживаемых замков называется *открытым вызовом* (open call) [CPJ 2.4.1.3], и классы, которые опираются на открытые вызовы, являются более благополучными и композиционно пригодными, чем классы, которые выполняют вызовы с занятыми замками. Использование открытых вызовов во избежание взаимной блокировки аналогично использованию инкапсуляции в целях обеспечения потокобезопасности. Хотя, конечно, можно создать потокобезопасную программу без инкапсуляции, анализ потокобезопасности программы, которая эффективно использует инкапсуляцию, намного проще той, которая этого не делает. Точно так же анализ жизнеспособности программы, которая опирается исключительно на открытые вызовы, намного проще, чем той, которая этого не делает. Ограничиваясь открытыми вызовами, вы значительно упрощаете идентификацию ветвей кода, которые приобретают многочисленные замки, и, следовательно, обеспечиваете, чтобы замки приобретались в непротиворечивом порядке[1].

---

[1] Необходимость опираться на открытые вызовы и тщательное замковое упорядочивание отражает фундаментальную беспорядочность составления синхронизированных объектов, а не синхронизации составных объектов.

Классы `Taxi` и `Dispatcher` в листинге 10.5 могут быть легко перестроены для использования открытых вызовов, тем самым риск возникновения взаимной блокировки будет устранен. Как показано в листинге 10.6, это предусматривает сжатие синхронизированных блоков для защиты только тех операций, которые касаются совместного состояния. Очень часто причиной проблем, подобных описанным в листинге 10.5, является использование синхронизированных методов вместо меньших синхронизированных блоков по причине компактного синтаксиса или простоты, а не потому, что весь метод должен быть защищен замком. (В качестве бонуса: сжатие синхронизированного блока также может улучшить масштабируемость; см. раздел 11.4.1 о размере синхронизуемых блоков.)

> Старайтесь использовать открытые вызовы по всей программе. Программы, которые опираются на открытые вызовы, гораздо легче анализировать на предмет отсутствия взаимной блокировки, чем те, которые допускают вызовы чужих методов с удержанием замков.

Листинг 10.6. Использование открытых вызовов, чтобы избежать тупиковых ситуаций между взаимодействующими объектами

```
@ThreadSafe
class Taxi {
    @GuardedBy("this") private Point location, destination;
    private final Dispatcher dispatcher;
    ...
    public synchronized Point getLocation() {
        return location;
    }

    public synchronized void setLocation(Point location) {
        boolean reachedDestination;
        synchronized (this) {
            this.location = location;
            reachedDestination = location.equals(destination);
        }
    if (reachedDestination)
        dispatcher.notifyAvailable(this);
    }
}

@ThreadSafe
class Dispatcher {
```

продолжение ⇩

Листинг 10.6 *(продолжение)*

```
    @GuardedBy("this") private final Set<Taxi> taxis;
    @GuardedBy("this") private final Set<Taxi> availableTaxis;
    ...
    public synchronized void notifyAvailable(Taxi taxi) {
        availableTaxis.add(taxi);
    }

    public Image getImage() {
        Set<Taxi> copy;
        synchronized (this) {
            copy = new HashSet<Taxi>(taxis);
        }
        Image image = new Image();
        for (Taxi t : copy)
            image.drawMarker(t.getLocation());
        return image;
    }
}
```

Реструктуризация синхронизированного блока для допущения открытых вызовов иногда может иметь нежелательные последствия, так как она берет операцию, которая была атомарной, и делает ее неатомарной. Во многих случаях потеря атомарности вполне допустима; нет никакой причины, почему обновление местоположения такси и уведомление диспетчера о том, что оно готово ехать к новому месту назначения, должно быть атомарной операцией. В других случаях потеря атомарности заметна, но семантические изменения по-прежнему приемлемы. В версии, предрасположенной к возникновению взаимной блокировки, метод `getImage` создает полный снимок местоположений парка такси в тот же самый момент; в перестроенной версии он доставляет местоположение каждого такси немного в разные времена.

В некоторых случаях потеря атомарности является проблемой, и здесь придется использовать еще одно техническое решение для достижения атомарности. Одним из таких технических решений является структурирование конкурентного объекта таким образом, чтобы только один поток мог выполнять ветвь кода после открытого вызова. Например, при выключении службы может потребоваться дождаться завершения операций, находящихся в процессе работы, а затем высвободить ресурсы, используемые службой. Удержание блокировки службы во время ожидания завершения операций по своей сути предрасположено к возникновению взаимной блокировки, но освобождение замка службы до выключения службы мо-

жет позволить другим потокам начать новые операции. Решение состоит в том, чтобы владеть замком достаточно долго и в течение этого времени обновить состояние службы до состояния «служба выключается», для того чтобы другие потоки, желающие начать новые операции — в том числе выключение службы, — видели, что служба отсутствует, и даже не пытались. Тогда вы можете дождаться, когда выключение завершится, зная, что только поток выключения имеет доступ к состоянию службы после завершения открытого вызова. Таким образом, чтобы использовать замковую защиту, которая держала бы другие потоки вне критической секции кода, это техническое решение опирается на конструирование протоколов, благодаря которым другие потоки не пытаются войти.

10.1.5. Ресурсные взаимные блокировки

Подобно тому как потоки могут быть заперты взаимной блокировкой, когда каждый из них ожидает замка, занятого другим, и потоком не освобождается, они точно так же могут быть заперты взаимной блокировкой во время ожидания ресурсов.

Предположим, у вас есть два ресурса, объединенных в пул, например пулы соединений для двух разных баз данных. Ресурсные пулы обычно реализуются с помощью семафоров (см. раздел 5.5.3), облегчающих блокирование продвижения, когда пул является пустым. Если задача требует подключения к базам данных и ресурсы не всегда запрашиваются в одинаковом порядке, то поток A мог бы удерживать подключение к базе данных D_1 в ожидании подключения к базе D_2, а поток B — удерживать подключение к D_2 в ожидании подключения к D_1. (Чем крупнее пулы, тем реже это происходит; если каждый пул имеет N соединений, то взаимная блокировка требует N наборов циклически ожидающих потоков и много неудачной временной координации.)

Еще одной формой взаимной блокировки на основе ресурсов является *взаимная блокировка с ресурсным голоданием*. Пример такого сбоя мы видели в разделе 8.1.1, где задача, которая предоставляет некую задачу и ждет ее результата, выполняется в однопоточном исполнителе. В этом случае первая задача будет ожидать вечно, навсегда застопорив выполнение этой и всех остальных задач в исполнителе. Задачи, ожидающие результатов других задач, являются первичным источником взаимной блокировки с ресурсным голоданием; ограниченные пулы и взаимозависимые задачи плохо сочетаются.

10.2. Предотвращение и диагностирование взаимной блокировки

Программа, которая никогда не приобретает более одного замка за один раз, не может столкнуться с взаимной блокировкой, которая инициируется порядком блокировки. Разумеется, это не всегда практично, но если вам это сходит с рук, то, значит, работы будет намного меньше. Если требуется приобретать многочисленные замки, то замковая упорядоченность должна стать частью проекта. Постарайтесь минимизировать число потенциальных замковых взаимодействий, а также следуйте протоколу порядка блокировки и документируйте ее в отношении замков, которые могут приобретаться вместе.

В программах, использующих блокировку с высокой степенью детализации, следует выполнить аудит своего кода на предмет отсутствия взаимной блокировки с помощью двухчастной стратегии: сначала идентифицировать экземпляры, где могут приобретаться многочисленные замки (попробуйте сделать это на небольшом наборе), а затем выполнить глобальный анализ всех таких экземпляров с целью обеспечения непротиворечивости порядка блокировки во всей программе. Использование открытых вызовов, где это возможно, существенно упрощает этот анализ. Без неоткрытых вызовов найти экземпляры, где приобретаются многочисленные замки, довольно легко: это достигается либо путем ревизии кода, либо с помощью автоматического анализа байт-кода или исходного кода.

10.2.1. Хронометрированные замки

Еще одно техническое решение для обнаружения и восстановления после взаимной блокировки заключается в использовании хронометрированного функционала `tryLock` явных замковых классов `Lock` (см. главу 13) вместо внутренней замковой защиты. Там, где внутренние замки ждут вечно, если они не могут приобрести замок, явные замки позволяют указывать тайм-аут, после которого метод `tryLock` возвращает сбой. Используя тайм-аут, который намного дольше, чем, по вашим ожиданиям, уйдет времени на приобретение замка, вы можете восстановить контроль, когда происходит нечто непредвиденное. (В листинге 13.3 показана альтернативная реализация метода перевода денег `transferMoney` с использованием опра-

шиваемого метода `tryLock` с повторными попытками для вероятностного избегания взаимной блокировки.)

Когда попытка с помощью хронометрированного замка завершается безуспешно, вы не обязательно будете знать *причину*. Возможно, возникла взаимная блокировка; возможно, поток ошибочно вошел в бесконечный цикл, владея этим замком; либо, возможно, какое-то действие выполняется намного медленнее, чем вы ожидали. Тем не менее у вас есть возможность записать, что ваша попытка оказалась безуспешной, внести в журнал любую полезную информацию о том, что вы пытались сделать, и перезапустить вычисление несколько более плавно, чем убить весь процесс целиком.

Использование захвата временного замка для получения нескольких замков может быть эффективным против тупиковой ситуации, даже если временный замок не используется постоянно в программе. Если время приобретения замков истекло, то вы можете освободить замки, отступить и подождать некоторое время, а затем повторить попытку, возможно, очистив условие возникновения взаимной блокировки и позволив программе восстановиться. (Это техническое решение работает только тогда, когда два замка приобретаются вместе; если многочисленные замки приобретаются вследствие вложенности вызовов методов, то вы не сможете просто освободить внешний замок, даже если знаете, что им владеете.)

10.2.2. Анализ взаимной блокировки с помощью поточных дампов

Хотя предотвращение взаимных блокировок в основном является вашей задачей, JVM помогает их идентифицировать, когда они встречаются, используя для этого *поточные дампы*. Поточный дамп содержит стековую трассировку для каждого работающего потока, аналогичную стековой трассировке, сопровождающей исключение. Поточные дампы также содержат информацию о замковой защите, например о том, какие замки заняты каждым потоком, в каком стековом фрейме они были приобретены и какой замок блокированный поток ожидает приобрести[1]. Перед генери-

[1] Эта информация полезна для отладки, даже если взаимной блокировки нет; периодический запуск поточных дампов позволяет наблюдать за замковым поведением программы.

рованием поточного дампа JVM проводит поиск циклов в графе ожидания с целью отыскать взаимные блокировки. Если среда JVM находит один такой замок, то включает о нем информацию, идентифицирующую, какие замки и потоки участвуют и где в программе расположены нарушающие замковые приобретения.

Для того чтобы инициировать поточный дамп, вы можете отправить процессу JVM сигнал `SIGQUIT` (`kill -3`) на платформах Unix или нажать комбинацию клавиш Ctrl-\ (Unix) или Ctrl-Break (Windows). Многие интегрированные среды разработки также могут запрашивать поточный дамп.

Если вы используете явные замковые классы `Lock` вместо внутренней замковой защиты, то знайте, что Java не поддерживает ассоциирование информации о `Lock` с поточным дампом; явные замки вообще не показываются в поточных дампах. Java 6 все-таки включает поддержку поточного дампа и обнаружение взаимной блокировки с помощью явных замков, но информация о том, где замки приобретены, с неизбежностью менее точна, чем для внутренних замков. Внутренние замки ассоциированы со стековым фреймом, в котором они были приобретены; явные замки ассоциированы только с приобретающим потоком.

В листинге 10.7 показаны фрагменты поточного дампа, взятые из производственного приложения J2EE. Сбой, приведший к взаимной блокировке, связан с тремя компонентами — приложением J2EE, контейнером J2EE и драйвером JDBC, каждый из которых принадлежит разным поставщикам. (Все имена были изменены, чтобы защитить виновников.) Все три были коммерческими продуктами, которые прошли через обширные циклы тестирования; у каждого был дефект, который был безвреден до тех пор, пока все они не стали взаимодействовать и не вызвали фатальный сбой сервера.

Мы показали только фрагмент поточного дампа, относящийся к идентификации взаимной блокировки. JVM проделала для нас много работы по диагностированию взаимной блокировки — какие замки вызывают проблему, какие потоки вовлечены, какие другие замки они удерживают и создало ли это другим потокам косвенные неудобства. Один поток владеет замком на `MumbleDBConnection` и ожидает приобретения замка на `MumbleDBCallableStatement`; другой владеет замком на `MumbleDBCallableStatement` и ожидает замка на `MumbleDBConnection`.

Листинг 10.7. Часть дампа потока после взаимоблокировки

```
Found one Java-level deadlock:
============================
"ApplicationServerThread":
  waiting to lock monitor 0x080f0cdc (a MumbleDBConnection),
  which is held by "ApplicationServerThread"
"ApplicationServerThread":
  waiting to lock monitor 0x080f0ed4 (a MumbleDBCallableStatement),
  which is held by "ApplicationServerThread"

Java stack information for the threads listed above:
"ApplicationServerThread":
        at MumbleDBConnection.remove_statement
        - waiting to lock <0x650f7f30> (a MumbleDBConnection)
        at MumbleDBStatement.close
        - locked <0x6024ffb0> (a MumbleDBCallableStatement)
    ...

"ApplicationServerThread":
        at MumbleDBCallableStatement.sendBatch
        - waiting to lock <0x6024ffb0> (a MumbleDBCallableStatement)
        at MumbleDBConnection.commit
        - locked <0x650f7f30> (a MumbleDBConnection)
    ...
```

Используемый здесь драйвер JDBC явно имеет ошибку в порядке блокировок: разные цепочки вызовов через драйвер JDBC приобретают многочисленные замки в разных порядках. Но эта проблема не проявилась бы, если бы не еще один дефект: многочисленные потоки пытались использовать один и тот же объект JDBC `Connection` в одно и то же время. Это совсем не то, как данное приложение предположительно должно было работать — разработчики были удивлены, увидев, что один и тот же объект `Connection` используется конкурентно двумя потоками. В спецификации JDBC нет ничего, что требовало бы, чтобы соединение `Connection` было потокобезопасным, и обычно использование объекта `Connection` ограничивается единственным потоком, как и было здесь предусмотрено. Этот поставщик пытался предоставить потокобезопасный драйвер JDBC, о чем свидетельствует синхронизация многочисленных объектов JDBC в коде драйвера. К сожалению, поскольку поставщик не принял во внимание порядок блокировки, драйвер был предрасположен к запиранию взаимной блокировкой, но данная проблема была раскрыта только в результате взаимодействия такого драйвера и неправильного совместного использования приложением объекта `Connection`. Поскольку ни один из дефектов

не был фатальным в отдельности, оба оставались нетронутыми, несмотря на обширное тестирование.

10.3. Другие сбои жизнеспособности

В то время как взаимная блокировка является наиболее широко встречающейся угрозой жизнеспособности, существует несколько других угроз жизнеспособности, с которыми вы можете столкнуться в конкурентных программах, включая голодание, пропущенные сигналы и активную блокировку. (Пропущенные сигналы рассматриваются в разделе 14.2.3.)

10.3.1. Голодание

Голодание (starvation) происходит, когда поток постоянно лишается доступа к ресурсам, в которых нуждается для продвижения вперед; процессорные циклы являются ресурсом, нехватка которого испытывается наиболее часто. Голодание в приложениях Java может быть вызвано неправильным использованием поточных приоритетов. Оно также может быть вызвано выполнением нетерминирующих конструкций (бесконечных циклов или ожиданий ресурсов, которые не терминируются) с удержанием замка, так как другие потоки, которым нужен этот замок, никогда не смогут его приобрести.

Поточные приоритеты, определенные в API Thread, — это всего-навсего советы по планированию. API Thread определяет десять уровней приоритетности, которые JVM может соотносить с приоритетами планирования операционной системы по своему усмотрению. Это соотнесение является платформенно-специфичным, вследствие чего два приоритета Java могут соотноситься с одним и тем же приоритетом ОС на одной системе и другими приоритетами ОС на другой. Некоторые операционные системы имеют менее десяти уровней приоритетности, в каковом случае многочисленные приоритеты Java соотносятся с тем же самым приоритетом ОС.

Планировщики ОС идут на многое, чтобы обеспечить справедливость и жизнеспособность планирования сверх того, что требуется спецификацией языка Java. В большинстве приложений Java все потоки приложений имеют одинаковый приоритет, `Thread.NORM_PRIORITY`. Механизм поточных приоритетов является тупым инструментом, и не всегда очевидно,

какой эффект изменение приоритетов будет иметь; повышение поточного приоритета может ничего не сделать либо всегда может приводить к тому, что один поток будет планироваться в предпочтение другому, вызывая голодание.

В общем случае разумно не регулировать поточные приоритеты. Как только вы начинаете модифицировать приоритеты, поведение вашего приложения становится платформенно-специфичным, и вы вносите риск голодания. Вы нередко можете опознать программу, которая пытается восстановиться после регулировки приоритета или других проблем с отзывчивостью, по наличию вызовов `Thread.sleep` или `Thread.yield` там, где их не ждешь, в попытке дать больше времени низкоприоритетным потокам[1].

> Избегайте соблазна использовать поточные приоритеты, так как они увеличивают зависимость от платформы и могут вызывать проблемы жизнеспособности. Большинство конкурентных приложений могут использовать стандартный приоритет для всех потоков.

10.3.2. Слабая отзывчивость

На одном шаге от голодания находится слабая отзывчивость, что не редкость в GUI-приложениях, использующих фоновые потоки. В главе 9 разработана структура для выгрузки длительных задач в фоновые потоки с целью предотвращения заморозки пользовательского интерфейса. Процессорно-интенсивные фоновые задачи по-прежнему могут влиять на отзывчивость, поскольку могут конфликтовать за процессорные циклы с событийным потоком. Это тот случай, когда изменение поточных приоритетов имеет смысл; когда вычислительно интенсивные фоновые вычисления повлияли бы на отзывчивость. Если работа, выполняемая другими потоками, является по-настоящему фоновыми задачами, снижение их приоритета может сделать фронтальные задачи более отзывчивыми.

---

[1] Семантика `Thread.yield` (и `Thread.sleep(0)`) не определена [JLS 17.9]; JVM может свободно реализовывать их как инструкции без операций или рассматривать их как советы по планированию. В частности от них не требуется иметь семантику `sleep(0)` в системах Unix — поместить текущий поток в конец рабочей очереди для этого приоритета, уступая место другим потокам с тем же приоритетом — хотя некоторые JVM реализуют `yield` таким образом.

Слабая отзывчивость может также быть обусловлена плохим замковым управлением. Если поток владеет замком в течение длительного времени (возможно, итеративно перебирая большую коллекцию и выполняя существенную работу для каждого элемента), то другим потокам, которым требуется доступ к этой коллекции, может потребоваться очень продолжительное время.

10.3.3. Активная блокировка

Активная блокировка (livelock) — это форма сбоя жизнеспособности, в котором поток, хотя и не блокирован, по-прежнему не может продвигаться вперед, потому что продолжает повторять операцию, которая всегда будет безуспешной. Активная блокировка часто происходит в транзакционных приложениях обмена сообщениями, где структура обмена сообщениями откатывает транзакцию, если сообщение не удается успешно обработать, и помещает его обратно в начало очереди. Если дефект в обработчике определенного типа сообщения вызывает его сбой, то всякий раз, когда сообщение удаляется из очереди и передается обработчику, выполняется откат транзакции. Так как сообщение теперь снова находится в голове очереди, обработчик вызывается снова и снова с тем же результатом. (Такая ситуация иногда называется проблемой *вредоносного сообщения*.) Поток обработки сообщений не блокируется, но при этом он никогда не будет продвигаться вперед. Эта форма активной блокировки часто происходит от избыточного кода обнаружения и исправления ошибок, который неправильно рассматривает неустранимую ошибку как устранимую.

Активная блокировка также может произойти, когда многочисленные взаимодействующие потоки изменяют свое состояние в ответ на другие таким образом, что ни один поток никогда не сможет добиться продвижения вперед. Это похоже на то, что происходит, когда два крайне вежливых человека идут в противоположных направлениях по коридору: каждый уступает дорогу другому, и в итоге не дают пройти друг другу. Они снова и снова отходят в сторону, и так до бесконечности.

Решение проблемы такого разнообразия активных блокировок состоит во внесении некоторой случайности в механизм повтора. Например, когда две станции в Ethernet-сети пытаются передать пакет на совместном носителе в одно и то же время, пакеты сталкиваются. Станции обнаруживают коллизию, и каждая из них пытается передать свой пакет снова позже. Если каждая из них повторяет попытку *ровно* через секунду, то они стал-

киваются снова и снова, и ни один пакет никогда не пройдет, даже если в распоряжении имеется много пропускной способности. Для того чтобы избежать этого, мы заставляем каждую станцию ожидать определенное количество времени, которое включает случайный компонент. (Протокол Ethernet также включает экспоненциальный откат после повторяющихся столкновений, уменьшая как перегрузку, так и риск многократного сбоя в многочисленных сталкивающихся станциях.) Выполнение повторной попытки со случайными ожиданиями и откатами может быть одинаково эффективным для предотвращения активной блокировки в конкурентных приложениях.

Итоги

Сбои жизнеспособности являются серьезной проблемой, потому что невозможно восстановиться, не прерывая приложения. Наиболее распространенной формой сбоя в жизнеспособности является взаимная блокировка из-за порядка блокировки. Предотвращение взаимной блокировки из-за порядка блокировки начинается во время разработки: обеспечить, чтобы во время приобретения потоками многочисленных замков они делали это в непротиворечивом порядке. Лучший способ сделать это — использовать открытые вызовы по всей программе. Это значительно уменьшает число мест, где удерживается сразу несколько замков, и делает более очевидными места их расположения.

11 Производительность и масштабирование

Одной из основных причин использования потоков является повышение производительности[1]. Использование потоков может улучшить использование ресурсов, позволяя приложениям легче эксплуатировать доступные вычислительные мощности, а также улучшить отзывчивость, давая приложениям начинать обработку новых задач немедленно, пока существующие задачи все еще работают.

В этой главе рассматриваются технические решения для анализа, мониторинга и повышения производительности конкурентных программ. К сожалению, многие технические решения для повышения производительности также увеличивают сложность, тем самым повышая вероятность сбоев в безопасности и жизнеспособности. Хуже того, некоторые технические решения, предназначенные для повышения производительности, на самом деле контрпродуктивны или обменивают одну проблему производительности на другую. В то время как более высокая производительность часто желательна — и повышение производительности бывает очень удовлетворительным, — безопасность всегда стоит на первом месте. Сначала сделать свою программу правильной, затем сделать ее быстрой — и исключительно тогда, когда ваши требования к производительности и ее измерения говорят вам, что она должна быть выше. При разработке конкурентного

[1] Некоторые могут возразить, что это единственная причина, почему мы миримся со сложностью, вносимой потоками.

приложения выжимание последней капли производительности часто является наименьшей проблемой.

11.1. Некоторые мысли о производительности

Повышение производительности означает выполнение большего объема работы при меньшем объеме ресурсов. Смысл термина «ресурсы» может варьироваться; для данного вида деятельности некоторые конкретные ресурсы, как правило, бывают дефицитными, будь то процессорные циклы, память, пропускная способность сети, пропускная способность ввода-вывода, запросы к базе данных, дисковое пространство или любое число других ресурсов. Когда производительность действия ограничена наличием определенного ресурса, мы говорим, что она *привязана* к этому ресурсу: привязана к процессору, привязана к базе данных и т. д.

Хотя целью может быть повышение производительности в целом, использование многочисленных потоков всегда вносит некоторую стоимость для производительности по сравнению с однопоточным подходом. К ним относятся издержки, ассоциированные с координацией между потоками (замковой защитой, сигнализацией и синхронизацией памяти), повышенным переключением контекста, созданием и удалением потоков, а также издержки на планирование. При эффективном использовании потоков эти стоимости с лихвой компенсируются большей пропускной способностью, отзывчивостью или емкостью. С другой стороны, плохо спроектированное конкурентное приложение может работать даже хуже, чем аналогичное последовательное[1].

Используя конкурентность для повышения производительности, мы пытаемся сделать две вещи: более эффективно задействовать обрабатывающие ресурсы, которые у нас есть, и позволить нашей программе эксплуатировать дополнительные обрабатывающие ресурсы, если они

[1] Коллега рассказал забавный случай: он участвовал в тестировании дорогого и сложного приложения, которое управляло своей работой через регулируемый пул потоков. После того как система была завершена, тестирование показало, что оптимальное число потоков для пула равно... 1. Это должно было быть очевидным с самого начала; целевая система была однопроцессорной, и приложение было почти полностью привязано к процессору.

появятся. С точки зрения мониторинга производительности это означает, что мы хотим, чтобы процессоры были максимально заняты. (Конечно, это вовсе не означает наличие циклов записи с бесполезными вычислениями; мы хотим, чтобы процессоры были заняты *полезной* работой.) Если программа привязана к вычислениям, то мы можем увеличить ее емкость, добавив больше процессоров; если она не может занять работой даже те процессоры, которые у нас есть, то добавление большего числа не поможет. Поточная обработка предлагает средство поддержания процессора(-ов) в «нагретом» состоянии, разбивая приложение на части, тем самым всегда обеспечивая работу, которая должна быть выполнена имеющимся процессором.

11.1.1. Производительность и масштабируемость

Производительность приложения может измеряться несколькими мерами, такими как время обслуживания, задержка, пропускная способность, эффективность, масштабируемость или емкость. Некоторые из них (время обслуживания, задержка) являются мерами того, «как быстро» данная единица работы может быть обработана или подтверждена; другие (емкость, пропускная способность) являются мерами того, «как много» работы может быть выполнено с данным количеством вычислительных ресурсов.

> *Масштабируемость* описывает возможность повышения пропускной способности или емкости при добавлении дополнительных вычислительных ресурсов (таких как дополнительные процессоры, память, хранилище или пропускная способность ввода-вывода).

Проектирование и регулировка конкурентных приложений в целях масштабируемости может сильно отличаться от традиционной оптимизации производительности. Во время регулировки производительности обычно цель состоит в том, чтобы выполнять одну и *ту же* работу с *меньшими* усилиями, например многократно использовать ранее вычисленные результаты через кэширование или заменить алгоритм $O(n^2)$ алгоритмом $O(n \log n)$. Во время регулировки масштабируемости вместо этого вы пытаетесь найти способы параллелизации задачи, благодаря которой сможете использовать дополнительные обрабатывающие ресурсы, для того чтобы выполнять *больше* работы с *большим* количеством ресурсов.

Эти два аспекта производительности — *как быстро* и *как много* — полностью разделены, а иногда даже противоречат друг другу. Для достижения более высокой масштабируемости или лучшего использования оборудования мы часто *увеличиваем* объем работы, выполняемой для обработки каждой *отдельной* задачи, например при разделении задач на многочисленные «конвейерные» подзадачи. По иронии судьбы, многие приемы, повышающие производительность однопоточных программ, плохо влияют на масштабируемость (см. пример в разделе 11.4.4).

Всем знакомая трехъярусная модель приложения — в которой представление, бизнес-логика и персистентность разделены и могут обрабатываться разными системами — иллюстрирует, как улучшения масштабируемости часто достигаются за счет снижения производительности. Монолитное приложение, в котором переплетены представление, бизнес-логика и персистентность, почти наверняка обеспечит более высокую производительность для *первой* единицы работы, чем хорошо продуманная многоярусная реализация, распределенная по многочисленным системам. А как же иначе? Монолитное приложение не будет иметь сетевую задержку, присущую эстафетной передаче задач между ярусами. Кроме того, оно не должно будет оплачивать стоимости, кроющиеся в разделении вычислительного процесса на отдельные абстрактные ярусы (например, издержки очередности, издержки координации и копирование данных).

Когда монолитная система достигнет своей обрабатывающей мощности, у нас может возникнуть серьезная проблема: значительно увеличить мощность может оказаться непомерно сложно. Поэтому мы часто уживаемся со стоимостью более длительного срока службы или большего числа вычислительных ресурсов, используемых на единицу работы, за счет которых наше приложение может масштабироваться, для того чтобы справляться с большей нагрузкой путем добавления большего числа ресурсов.

Из различных аспектов производительности аспекты «как много» — масштабируемость, пропускная способность и емкость — обычно имеют большее значение для серверных приложений, чем аспекты «как быстро». (В случае интерактивных приложений задержка, как правило, имеет более высокую значимость, поэтому пользователям не нужно ждать индикаций хода выполнения и задаваться вопросом, что происходит.) В этой главе основное внимание уделяется масштабируемости, а не сырой однопоточной производительности.

11.1.2. Оценивание компромиссов производительности

Почти все инженерные решения предполагают некий компромисс. Использование более толстой стали в пролете моста может увеличить не только его грузоподъемность и безопасность, но и стоимость возведения. Хотя решения в области программной инженерии обычно не сопряжены с компромиссами между деньгами и риском для человеческой жизни, у нас часто меньше информации, с помощью которой можно принимать правильные решения. Например, алгоритм быстрой сортировки очень эффективен для больших совокупностей данных, но менее сложная пузырьковая сортировка в действительности эффективнее в случае малых совокупностей данных. Если вам поручено реализовать эффективную процедуру сортировки, то нужно знать, какие размеры совокупностей данных придется обрабатывать, а также метрические показатели, говорящие о том, пытаетесь ли вы оптимизировать время для среднего случая, время для худшего случая либо прогнозируемость. К сожалению, эта информация часто не входит в требования, предъявляемые к автору библиотечной процедуры сортировки. Это одна из причин, почему большинство оптимизаций преждевременны: *они часто предпринимаются до того, как доступен четкий набор требований*.

> Избегайте преждевременной оптимизации. Сначала выполните программную задачу правильно, затем сделайте ее быстрой — если она еще недостаточно быстра.

Во время принятия инженерных решений иногда вы обмениваете одну форму стоимости на другую (время обслуживания и потребление памяти); иногда обмениваете стоимость на безопасность. Безопасность необязательно означает риск для человеческих жизней, как это было в примере с мостом. Многие способы оптимизации производительности происходят за счет удобочитаемости или сопровождаемости — чем «заумнее» или неочевиднее код, тем труднее его понять и сопровождать. Иногда оптимизация влечет за собой компрометацию хороших объектно-ориентированных принципов проектирования, таких как нарушение инкапсуляции; иногда они сопряжены с бóльшим риском ошибок, поскольку более быстрые алгоритмы обычно сложнее. (Если вы не можете идентифицировать стои-

мости или риски, то вы, вероятно, не продумали все достаточно тщательно, чтобы продолжать.)

Большинство решений относительно производительности включают в себя многочисленные переменные и являются ситуативными. Прежде чем принять решение о том, что один подход «быстрее», чем другой, задайте себе несколько вопросов:

- Что вы подразумеваете под словом «быстрее»?
- При каких условиях этот подход будет быстрее? В условиях легкой или тяжелой нагрузки? С большими или малыми совокупностями данных? Можете ли вы поддержать свой ответ измерениями?
- Как часто эти условия могут возникать в вашей ситуации? Можете вы поддержать свой ответ измерениями?
- Может ли этот код использоваться в других ситуациях, когда условия могут отличаться?
- Какие скрытые стоимости, такие как повышенный риск для разработки или сопровождения, вы используете для повышения производительности? Является ли это хорошим компромиссом?

Эти соображения применимы к любому инженерному решению, связанному с производительностью, но это книга о конкурентности. Почему мы рекомендуем такой консервативный подход к оптимизации? *Стремление к повышению производительности, вероятно, является единственным источником ошибок конкурентности.* Убеждение в том, что синхронизация была «слишком медленной», привело к появлению многих на вид умных, но опасных идиом сокращения синхронизации (таких как блокировка с двойной проверкой, обсуждаемая в разделе 16.2.4) и часто цитируется как оправдание для нарушения правил синхронизации. Но поскольку дефекты конкурентности являются одними из самых трудных для отслеживания и устранения, все решения должны быть внедрены тщательнейшим образом.

Хуже того: когда вы обмениваете безопасность на производительность, вы можете не получить ни того, ни другого. В особенности когда дело доходит до конкурентности, интуиция обманывает многих разработчиков: они неверно представляют, где находится слабое звено производительности или какой подход будет быстрее или более масштабируемым. Поэтому крайне важно, чтобы любое усилие по регулировке производительности сопро-

вождалось конкретными требованиями к производительности (благодаря которым вы будете знать, когда регулировать, а когда *прекращать* регулировать), а также программой измерений с использованием реалистичной конфигурации и профиля нагрузки. Выполните повторные измерения после регулировки, для того чтобы убедиться, что вы достигли желаемых улучшений. Риски для безопасности и сопровождения, связанные со многими оптимизациями, достаточно плохи: вы не хотите оплачивать эти стоимости, если в этом нет необходимости, и вы определенно не хотите оплачивать их, если вы даже не получаете желаемой выгоды.

Измеряйте, а не гадайте.

На рынке существуют сложные инструменты профилирования для измерения производительности и отслеживания узких мест в производительности, но вам вовсе не нужно тратить много денег, чтобы понять, что делает ваша программа. Например, бесплатное приложение `perfbar` может дать хорошее представление о том, насколько процессоры заняты, и поскольку цель, как правило, состоит в том, чтобы процессоры оставались занятыми, это очень хороший способ оценить, следует ли выполнить регулировку производительности или насколько эффективной была ваша регулировка.

11.2. Закон Амдала

Некоторые задачи могут решаться быстрее с бóльшим количеством ресурсов — чем больше работников для уборки урожая, тем быстрее может быть завершен сбор урожая. Другие же задачи являются принципиально последовательными — никакое количество дополнительных работников не заставит урожай расти быстрее. Если одной из первостепенных причин использования потоков является использование мощи многочисленных процессоров, то мы также должны обеспечить, чтобы задача поддавалась параллельной декомпозиции и чтобы наша программа эффективно эксплуатировала этот потенциал параллелизации.

Большинство конкурентных программ имеют много общего с фермерством, так как состоят из набора параллелизуемых и последовательных частей. *Закон Амдала* описывает, насколько программа может быть теоретически ускорена с помощью дополнительных вычислительных ресурсов,

исходя из соотношения параллелизируемых и последовательных компонентов. Если F — это доля вычислений, которые должны выполняться последовательно, то закон Амдала гласит, что на машине с N процессорами мы можем достичь:

$$Ускорение \leq \frac{1}{F + \frac{(1 - F)}{N}}.$$

По мере того как N приближается к бесконечности, максимальное ускорение сходится к $1/F$. Это значит, что программа, в которой 50 % обработки должны выполняться последовательно, может быть ускорена только в два раза, независимо от того, сколько имеется процессоров. А программа, в которой должны выполняться последовательно 10 %, может быть ускорена более чем в десять раз. Закон Амдала также квантифицирует стоимостную эффективность сериализации (то есть преобразования из одновременной формы в последовательную). С десятью процессорами программа с 10 %-ной сериализацией может достигнуть ускорения максимум 5,3 (при 53 %-ной задействованности), а со 100 процессорами может достичь ускорения не более 9,2 (при 9 %-ной задействованности). Иными словами, для того чтобы никогда не добраться до кратности 10, требуется много неэффективно задействованных процессоров.

Рисунок 11.1 показывает максимально возможную задействованность процессора для разной степени сериализованного выполнения и количества процессоров. (Задействованность определяется как ускорение, деленное на количество процессоров.) Очевидно, что по мере увеличения числа процессоров даже небольшой процент сериализованного выполнения ограничивает величину пропускной способности, которую можно увеличить с помощью дополнительных вычислительных ресурсов.

В главе 6 рассматриваются логические границы для разбиения приложений на задачи. Но для того, чтобы предсказать, какого рода ускорение возможно от работы вашего приложения в многопроцессорной системе, необходимо также определить источники сериализации в ваших задачах.

Представьте себе приложение, в котором N потоков выполняют `doWork` в листинге 11.1, доставляя задачи из совместной рабочей очереди и обрабатывая их; предположим, что задачи не зависят от результатов либо побочных эффектов других задач. Игнорируя на мгновение то, как задачи

попадают в очередь, насколько хорошо это приложение будет масштабироваться по мере добавления процессоров? На первый взгляд может показаться, что приложение полностью параллелизуемо: задачи не ожидают друг друга, и чем больше процессоров, тем больше задач может обрабатываться конкурентно. Однако есть и последовательный компонент — доставка задачи из рабочей очереди. Рабочая очередь используется всеми рабочими потоками совместно, и для поддержания ее целостности перед лицом конкурентного доступа потребуется некоторый объем синхронизации. Если для защиты состояния очереди используется замковая защита, то в момент, когда один поток удаляет задачу из очереди, другие потоки, которые должны удалить свою следующую задачу из очереди, должны ждать — и именно в этом месте сериализуется обработка задачи.

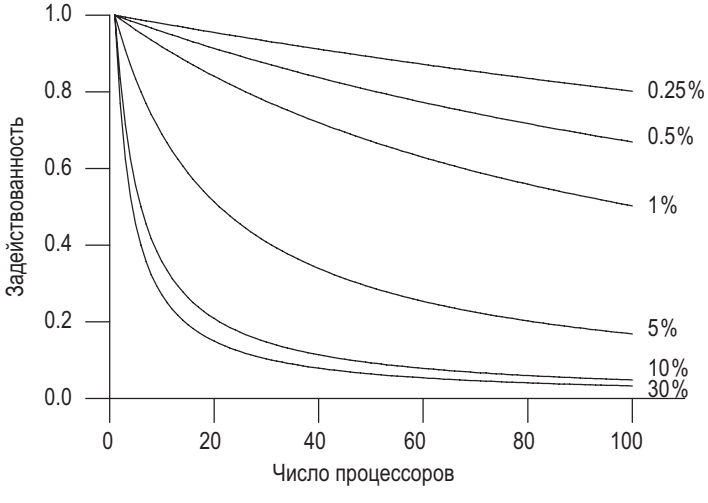

Рис. 11.1. Максимальная задействованность по закону Амдала для разных процентов сериализации

Время обработки отдельной задачи включает в себя не только время выполнения задачи `Runnable`, но и время удаления задачи из совместной рабочей очереди. Если рабочей очередью является `LinkedBlockingQueue`, то операция удаления из очереди `dequeue` может блокировать меньше, чем с синхронизированным связным списком `LinkedList`, поскольку очередь `LinkedBlockingQueue` использует более масштабируемый алгоритм, но доступ к любой совместной структуре данных принципиально вносит в программу элемент сериализации.

Листинг 11.1. Последовательный доступ к очереди задач

```
public class WorkerThread extends Thread {
    private final BlockingQueue<Runnable> queue;

    public WorkerThread(BlockingQueue<Runnable> queue) {
        this.queue = queue;
    }

    public void run() {
        while (true) {
            try {
                Runnable task = queue.take();
                task.run();
            } catch (InterruptedException e) {
                break; /* Дать потоку выйти */
            }
        }
    }
}
```

В этом примере также игнорируется еще один общий источник сериализации: обработка результатов. Все полезные вычисления дают что-то вроде результата или побочного эффекта — если нет, то их можно исключить как мертвый код. Поскольку задача `Runnable` не предоставляет явной обработки результатов, эти задачи должны иметь какой-то побочный эффект, например запись результатов в файл журнала или помещение их в структуру данных. Журнальные файлы и контейнеры результатов обычно используются многочисленными рабочими потоками совместно и поэтому также являются источником сериализации. Если вместо этого каждый поток поддерживает свою собственную структуру данных для результатов, которые сливаются воедино после выполнения всех задач, то окончательное слияние является источником сериализации.

> Все конкурентные приложения имеют некие источники сериализации; если вы думаете, что ваше не имеет, то прочитайте еще раз.

11.2.1. Пример: сериализация, скрытая в фреймворках

Чтобы увидеть, как сериализация может быть скрыта в структуре приложения, мы можем сравнивать пропускную способность по мере добавления

потоков и определять разницы в сериализации на основе наблюдаемых разниц в масштабируемости. На рис. 11.2 показано простое приложение, в котором многочисленные потоки многократно удаляют элемент из совместной очереди и обрабатывают его, как показано в листинге 11.1. Шаг обработки включает только локальное для потока вычисление. Если поток обнаруживает, что очередь пуста, то он помещает пакет новых элементов в очередь, для того чтобы другие потоки могли что-то обрабатывать на следующей итерации. Доступ к совместной очереди явно влечет за собой некоторую степень сериализации, но шаг обработки полностью параллелизуем, так как он не предусматривает совместные данные.

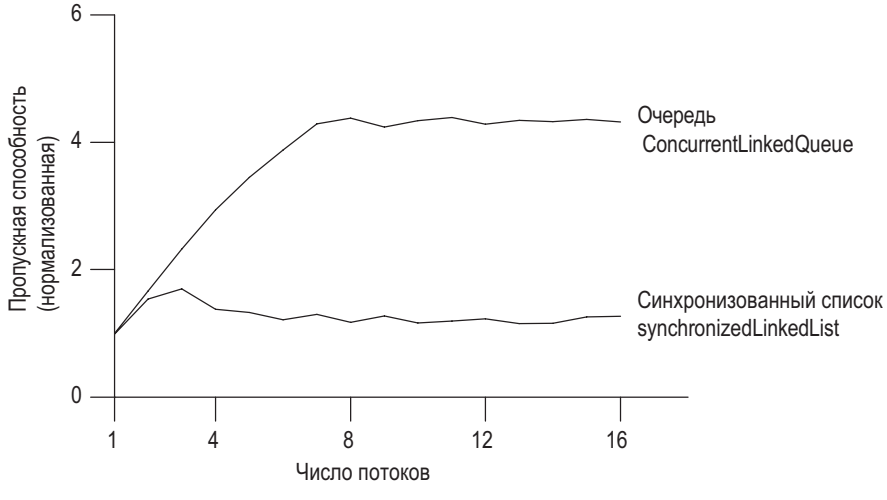

Рис. 11.2. Сравнение реализаций очереди

Кривые на рис. 11.2 сравнивают пропускную способность двух потокобезопасных реализаций очереди: связный список `LinkedList`, обернутый в `synchronizedList`, и очередь `ConcurrentLinkedQueue`. Тесты проводились на 8-ядерной системе Sparc V880 под управлением Solaris. Хотя каждый прогон представляет одинаковый объем «работы», мы видим, что простая замена реализаций очереди может оказать большое влияние на масштабируемость.

Пропускная способность очереди `ConcurrentLinkedQueue` продолжает улучшаться, до тех пор пока не достигнет числа процессоров, а затем остается почти постоянной. С другой стороны, пропускная способность

синхронизированного связного списка `LinkedList` показывает некоторое улучшение вплоть до трех потоков, но затем падает по мере увеличения издержек синхронизации. К тому времени, когда она достигает четырех или пяти потоков, конфликт настолько велик, что каждый доступ к очередному замку оспаривается, а над пропускной способностью доминирует переключение контекста.

Разница в пропускной способности происходит из разных степеней сериализации между двумя реализациями очереди. Синхронизированный связный список `LinkedList` защищает все состояние очереди единственным замком, который удерживается в течение всего времени вызова операций `offer` или `remove`; связная очередь `ConcurrentLinkedQueue` применяет сложный неблокирующий связанный список (см. раздел 15.4.2), который использует атомарные ссылки для обновления отдельных связных указателей. В одном случае сериализуются все операции вставки или удаления целиком; в другом сериализуются только обновления отдельных указателей.

11.2.2. Качественное применение закона Амдала

Закон Амдала квантифицирует возможное ускорение при наличии большего количества вычислительных ресурсов, в случае если мы можем точно оценить долю сериализованного выполнения. Хотя измерение сериализации напрямую может быть затруднено, закон Амдала все же может быть полезен без такого измерения.

Поскольку наши ментальные модели находятся под влиянием окружающей среды, многие привыкли думать, что многопроцессорная система имеет два или четыре процессора, или, может быть (если у нас большой бюджет), целых несколько десятков, потому что эта технология была широко доступна в последние годы. Но по мере того как многоядерные процессоры будут становиться магистральным направлением, системы будут иметь сотни или даже тысячи процессоров[1]. Алгоритмы, которые кажутся масштабируемыми в четырехъядерной системе, могут иметь скрытые узкие места масштабируемости, которые еще не были обнаружены.

---

[1] Обновление рыночной информации: на данный момент Sun поставляет серверные системы низкого класса aNiagara на основе восьмиядерного процессора и серверные системы высокого класса Azul (96, 192 и 384-ядерные) на основе 24-ядерного процессора Vega.

Во время оценивания алгоритма размышление «в пределе» о том, что произойдет с сотнями или тысячами процессоров, может дать некоторое представление о том, где могут возникнуть лимиты масштабируемости. Например, в разделах 11.4.2 и 11.4.3 рассматриваются два технические решения для сокращения замковой детализации: замковое расщепление на два замка и замковое расщепление на полосы. Глядя на них через призму закона Амдала, мы видим, что замковое расщепление на два замка не продвигает нас очень далеко в сторону задействованности многочисленных процессоров, но замковое расщепление на полосы кажется гораздо более перспективным, потому что размер набора полос может быть увеличен с увеличением числа процессоров. (Конечно, приемы оптимизации производительности всегда следует рассматривать в свете фактических требований к производительности; в некоторых случаях для удовлетворения требований бывает достаточным разделить замок на две части.)

11.3. Стоимость, вносимая потоком

Однопоточные программы не требуют издержек ни на планирование, ни на синхронизацию и не требуют для поддержания непротиворечивости структур данных использовать замки. Планирование и координация между потоками требуют издержек, которые имеют стоимость для производительности; для повышения производительности потоков преимущества параллелизации должны перевешивать стоимости, вносимые конкурентностью.

11.3.1. Переключение контекста

Если главный поток является единственным планово-способным потоком, то он почти никогда не будет вынесен за пределы планирования. С другой стороны, если существует больше работоспособных потоков, чем процессоров, то в итоге ОС будет вытеснять один поток, давая другому возможность использовать процессор. В результате получается *переключение контекста*, которое требует сохранения контекста выполнения текущего выполняющего потока и восстановления контекста выполнения только что запланированного потока.

Переключения контекста небесплатны; поточное планирование требует управления совместными структурами данных в ОС и JVM. ОС и JVM используют те же процессоры, что и ваша программа; больше времени

процессора, потраченного в коде JVM и ОС, означает, что для вашей программы доступно меньше. Но деятельность ОС и JVM не является единственной стоимостью переключений контекста. Когда подключается новый поток, данные, которые ему нужны, вряд ли будут в локальном кэше процессора, поэтому переключение контекста вызывает шквал непопаданий в кэше, и в результате потоки работают немного медленнее, когда они только что были запланированы. Это одна из причин, почему планировщики предоставляют каждому работоспособному потоку определенный минимальный квант времени, даже когда многие другие потоки ожидают: этим амортизируется стоимость переключения контекста и его последствия в течение большего непрерывного времени выполнения, улучшая общую пропускную способность (за счет некоторого снижения отзывчивости).

Когда поток блокирует продвижение, потому что ожидает оспариваемого замка, JVM обычно приостанавливает поток и разрешает ему отключиться. Если потоки часто блокируются, то они не могут использовать свой полный квант планирования. Программа, которая задействует больше блокирования (блокирующий ввод-вывод, ожидание оспариваемых замков или ожидание условных переменных), требует больше переключений контекста, чем та, которая привязана к процессору, увеличивая издержки на планирование и уменьшая пропускную способность. (Неблокирующие алгоритмы также помогают сократить переключение контекста; см. главу 15.)

Фактическая стоимость переключения контекста варьируется между платформами, но хорошее эмпирическое правило состоит в том, что стоимость переключателя контекста эквивалентна 5000–10 000 тактовым циклам или нескольким микросекундам на большинстве нынешних процессорах.

Команда `vmstat` в системах Unix и инструмент `perfmon` в системах Windows сообщают число переключений контекста и процент времени, затрачиваемого в ядре. Высокая загрузка ядра (более 10 %) часто указывает на высокую активность планирования, которая может быть вызвана замковой защитой из-за ввода-вывода или конфликта за замок.

11.3.2. Синхронизации памяти

Стоимость синхронизации для производительности проистекает из нескольких источников. Гарантии видимости, предоставляемые механизмами `synchronized` и `volatile`, могут повлечь за собой использование специальных инструкций, называемых *барьерами памяти*, которые могут

сбрасывать или обнулять кэши, сбрасывать аппаратные буферы записи и защелкивать конвейеры выполнения. Барьеры памяти могут косвенно влиять на производительность, поскольку препятствуют другой оптимизации компиляторов; большинство операций не пригодны для переупорядочивания с помощью барьеров памяти.

При оценивании влияния синхронизации на производительность важно проводить различие между *оспариваемой* и *неоспариваемой* синхронизацией. Механизм `synchronized` оптимизирован для неоспариваемого случая (`volatile` всегда не оспаривается), и на момент написания этих строк стоимость для производительности неоспариваемой синхронизации «при более короткой длине инструкции» (fast-path) колеблется от 20 до 250 тактов для большинства систем. Хотя это, конечно, не ноль, эффект необходимой, неоспариваемой синхронизации редко бывает значительным в общей производительности приложения, и альтернатива включает в себя угрозу для безопасности и собственное (или вашего преемника) согласие на весьма болезненное отлавливание дефектов впоследствии.

Современные JVM могут снизить стоимость инцидентной синхронизации путем оптимизации замковой защиты, которая, что можно доказать, никогда не оспаривается. Если замковый объект доступен только текущему потоку, то JVM разрешено в рамках оптимизации исключать замковое приобретение, потому что другой поток никоим образом не способен синхронизироваться на том же замке. Например, замковое приобретение в листинге 11.2 всегда может быть устранено JVM.

Листинг 11.2. Синхронизация, не имеющая эффекта. *Так делать не следует*

```
synchronized (new Object()) {
    // сделать что-то
}
```

Более изощренные JVM могут использовать *анализ ускользания* с целью определения момента, когда ссылка на локальный объект никогда не публикуется в куче и, следовательно, является локальной для потока. В методе `getStoogeNames` в листинге 11.3 единственной ссылкой на список является локальная переменная `stooges`, и переменные, ограниченные

стеком, автоматически являются локальными для потока. Наивное выполнение метода `getStoogeNames` будет приобретать и снимать замок на векторе четыре раза, один раз для каждого вызова методов `add` или `toString`. Однако умный компилятор рабочей среды может внедрить эти вызовы в код, а затем увидеть, что `stooges` и ее внутреннее состояние никогда не ускользали, и поэтому все четыре приобретения замка могут быть устранены[1].

Листинг 11.3. Кандидат на снятие замка

```
public String getStoogeNames() {
    List<String> stooges = new Vector<String>();
    stooges.add("Moe");
    stooges.add("Larry");
    stooges.add("Curly");
    return stooges.toString();
}
```

Даже без анализа ускользания компиляторы могут выполнять *укрупнение замков*, объединяя соседние синхронизированные блоки с одним и тем же замком. Для метода `getStoogeNames` JVM, выполняющая укрупнение замков, может объединить три вызова метода `add` и вызов `toString` в единое замковое приобретение и снятие, используя эвристики на относительной стоимости синхронизации против инструкций внутри синхронизированного блока[2]. Благодаря этому не только снижаются издержки на синхронизацию, но и оптимизатор получает гораздо больший блок для работы, при этом, вполне вероятно, активируя другие оптимизации.

> Не беспокойтесь излишне о стоимости неоспариваемой синхронизации. Базовый механизм уже достаточно быстр, и JVM-машины могут делать дополнительные оптимизации, которые еще больше снижают или устраняют эту стоимость. Вместо этого сосредоточьте оптимизационные усилия на участках, где конфликт блокировки происходит фактически.

---

[1] Эта оптимизация компилятора, именуемая *снятием замка* (lock elision), выполняется JVM-машиной IBM, и ее появление ожидается в HotSpot в Java 7.

[2] Умный динамический компилятор может понять, что этот метод всегда возвращает одну и ту же строку, и после первого выполнения перекомпилировать `getStoogeNames`, для того чтобы просто возвращать значение, возвращенное первым выполнением.

Синхронизация, выполняемая одним потоком, также может повлиять на производительность других потоков. Синхронизация создает трафик на шине совместно используемой памяти; эта шина имеет ограниченную пропускную способность и используется всеми процессорами совместно. Если потоки должны конфликтовать за пропускную способность синхронизации, то будут страдать все потоки, использующие синхронизацию[1].

11.3.3. Блокирование

Неоспариваемая синхронизация может быть полностью обработана внутри JVM (Bacon и соавт., 1998); оспариваемая синхронизация может потребовать действий ОС, что увеличивает стоимость. Когда замок оспаривается, проигравший поток(и) должен блокировать продвижение. JVM может реализовать блокирование либо путем ожидания в *холостом цикле* (спин-ожидания, т. е. многократно пытаясь приобрести замок до тех пор, пока попытка не будет успешной) либо путем *приостановки* блокированного потока через операционную систему. Ответ на вопрос, что из этого эффективнее, зависит от связи между издержками на переключение контекста и временем, пока замок не появится; холостой цикл предпочтителен для коротких ожиданий, и приостановка предпочтительна для длинных. Некоторые JVM выбирают между этими двумя вариантами адаптивно, на основе данных профилирования прошлых времен ожидания, но большинство просто приостанавливает потоки, которые ожидают замка.

Приостановка потока вследствие того, что он не может получить замок, блокирован ожиданием по условию или блокирующей операцией ввода-вывода, влечет за собой два дополнительных переключения контекста и дальнейшие действия операционной системы и кэша: блокированный поток отключается, прежде чем его квант истек, а затем подключается обратно, после того как появляется замок или другой ресурс. (Блокирование из-за конфликта блокировки также имеет стоимость для потока, владеющего замком: когда он освобождает замок, то должен затем запросить ОС возобновить блокированный поток.)

---

[1] Этот аспект иногда применяется, для того чтобы возразить против использования неблокирующих алгоритмов без какого-либо отката, потому что в условиях сильного конфликта неблокирующие алгоритмы генерируют больше синхронизационного трафика, чем замковые. См. главу 15.

11.4. Сокращение конфликта блокировки

Мы видели, что сериализация снижает масштабируемость, а переключение контекста снижает производительность. Конфликт блокировки вызывает и то, и другое, поэтому уменьшение конфликта блокировки может повысить производительность и масштабируемость.

Доступ к ресурсам, защищенным исключающей блокировкой, является последовательным — к нему может обращаться только один поток за раз. Конечно, мы используем замки по веским причинам, таким как предотвращение повреждения данных, но эта безопасность имеет свою цену. Постоянный конфликт блокировки ограничивает масштабируемость.

> Основной угрозой для масштабируемости в конкурентных приложениях является исключающая ресурсная блокировка.

Вероятность конфликта блокировки зависит от двух факторов: как часто этот замок запрашивается и как долго он удерживается после его приобретения[1]. Если произведение этих факторов достаточно мало, то большинство попыток приобрести замок не будут оспариваться, и случаи конфликта блокировки не будут представлять значительного препятствия для масштабируемости. Если же замок пользуется достаточно большим спросом, то потоки будут блокировать в ожидании замка; в экстремальном случае процессоры будут простаивать, даже если есть много работы.

> Существует три способа сокращения случаев конфликта блокировки:
>
> - Сократить время, в течение которого замок удерживается;
> - Сократить частоту, с которой замки запрашиваются;
> - Заменить исключающие блокировки механизмами координации, обеспечивающими большую конкурентность.

[1] Это следствие *закона Литтла*, являющегося результатом теории очередей, которая гласит: «Среднее число клиентов в стабильной системе равно их средней скорости прибытия, умноженной на их среднее время в системе» (Little, 1961).

11.4.1. Сужение области действия замка («вошел, вышел»)

Эффективный способ снизить вероятность конфликта — владеть замками максимально быстро. Это можно сделать путем выноса кода, который не требует замка, за пределы синхронизированных блоков, в особенности это касается дорогостоящих операций и потенциально блокирующих операций, таких как ввод-вывод.

Легко увидеть, как слишком долгое удержание «горячей» блокировкой может ограничивать масштабируемость; мы видели пример тому в классе `SynchronizedFactorizer` в главе 2. Если операция владеет замком в течение 2 миллисекунд и для каждой операции требуется этот замок, то пропускная способность не может превышать 500 операций в секунду, независимо от числа имеющихся процессоров. Уменьшение времени удержания блокировки до 1 миллисекунды улучшает лимит индуцированной замком пропускной способности до тысячи операций в секунду[1].

Класс `AttributeStore` в листинге 11.4 показывает пример удержания блокировки дольше, чем это необходимо. Метод `userLocationMatches` отыскивает позицию пользователя в ассоциативном массиве `Map` и использует регулярное выражение, для того чтобы увидеть, соответствует или нет полученное значение предоставленному шаблону. Весь метод `userLocationMatches` синхронизирован, но единственная часть кода, которая действительно нуждается в замке, — это вызов метода `Map.get`.

Листинг 11.4. Удержание блокировки дольше чем необходимо

```
@ThreadSafe
public class AttributeStore {
    @GuardedBy("this") private final Map<String, String>
            attributes = new HashMap<String, String>();
```

[1] Фактически это вычисление занижает стоимость удержания блокировки в течение слишком продолжительного времени, потому что оно не учитывает издержки на переключение контекста, генерируемые увеличением конфликта блокировки.

```
    public synchronized boolean userLocationMatches(String name,
                                                    String regexp) {
        String key = "users." + name + ".location";
        String location = attributes.get(key);
        if (location == null)
            return false;
        else
            return Pattern.matches(regexp, location);
    }
}
```

Класс `BetterAttributeStore` в листинге 11.5 переписывает `AttributeStore` с целью значительного сокращения длительности замка. Первым шагом является создание ключа ассоциативного массива `Map`, ассоциированного с позицией пользователя, строки в форме `users.`name`.location`. Это влечет за собой инстанциацию объекта `StringBuilder`, добавление в него нескольких строк и инстанциацию результата в качестве `String`. После того как позиция извлечена, регулярное выражение сопоставляется с результирующей строкой позиции. Поскольку конструирование строкового значения с ключом и обработка регулярного выражения не имеют доступа к совместному состоянию, они не должны выполняться с удержанием замка. `BetterAttributeStore` выводит эти шаги за скобки синхронизированного блока, тем самым сокращая время, когда замок занят.

Листинг 11.5. Уменьшение продолжительности блокировки

```
@ThreadSafe
public class BetterAttributeStore {
    @GuardedBy("this") private final Map<String, String>
            attributes = new HashMap<String, String>();

    public boolean userLocationMatches(String name, String regexp) {
        String key = "users." + name + ".location";
        String location;
        synchronized (this) {
            location = attributes.get(key);
        }
        if (location == null)
            return false;
        else
            return Pattern.matches(regexp, location);
    }
}
```

Сокращение области действия замка в `userLocationMatches` существенно сокращает число инструкций, которые выполняются с удержанием замка. По закону Амдала это устраняет препятствие перед масштабируемостью, поскольку объем последовательного кода сокращается.

Поскольку класс `AttributeStore` имеет только одну переменную состояния, `attributes`, мы можем усовершенствовать его дальше техническим решением *делегировать потокобезопасность* (раздел 4.3). Поменяв `attributes` на потокобезопасный ассоциативный массив `Map` (`Hashtable`, `synchronizedMap` или `ConcurrentHashMap`), класс `AttributeStore` может делегировать все свои обязательства по потокобезопасности базовой потокобезопасной коллекции. Это устраняет необходимость явной синхронизации в `AttributeStore`, сводит область действия замка к продолжительности доступа к ассоциативному массиву и устраняет риск того, что будущий сопроводитель подорвет потокобезопасность, забыв приобрести соответствующий замок перед доступом к переменной `attributes`.

Хотя сжатие синхронизированных блоков может улучшить масштабируемость, синхронизированный блок может быть *слишком* малым — операции, которые должны быть атомарными (например, обновление многочисленных участвующих в инварианте переменных), должны содержаться в единственном синхронизированном блоке. И поскольку стоимость синхронизации не равна нулю, замковое расщепление синхронизированного блока на многочисленные синхронизированные блоки (если это позволяет правильность) в какой-то момент становится контрпродуктивным с точки зрения производительности[1]. Идеальный баланс, конечно, зависит от платформы, но на практике имеет смысл беспокоиться о размере синхронизированного блока только тогда, когда вы можете вынести из него «существенные» вычисления или блокирующие операции.

11.4.2. Сокращение степени детализации замка

Другой вариант сократить долю времени, в течение которого замок удерживается (и, следовательно, вероятность того, что он будет оспариваться), состоит в том, чтобы потоки запрашивали его менее часто. Это может быть выполнено *замковым разделением* (lock splitting) и *чередованием блокировок* (lock striping), которые предусматривают использование от-

[1] Если JVM выполняет укрупнение замка, она в любом случае может отменить разделение синхронизированных блоков.

дельных замков для защиты многочисленных независимых переменных состояния, ранее защищенных единственным замком. Эти технические решения сокращают степень детализации, при которой осуществляется замковая защита, потенциально допуская более высокую масштабируемость, — но использование большего числа замков также увеличивает риск возникновения взаимной блокировки.

В качестве мысленного эксперимента представьте, что произойдет, если для всего приложения будет *только один* замок вместо отдельного замка для каждого объекта. Тогда выполнение всех синхронизированных блоков, независимо от их замков, будет сериализовано. Поскольку многие потоки конфликтуют за глобальный замок, вероятность того, что двум потокам требуется этот замок в одно и то же время, возрастает, что приводит к большему конфликту. Поэтому, если бы замковые запросы были распределены по *большему* множеству замков, то существовало бы меньше случаев конфликта. Меньшее число потоков будет блокировано в ожидании замков, что повысит масштабируемость.

Если замок защищает более чем одну *независимую* переменную состояния, то вы можете улучшить масштабируемость, расщепив его на многочисленные замки, каждый из которых защищает разные переменные. В результате каждый замок запрашивается реже.

Класс `ServerStatus` в листинге 11.6 показывает фрагмент мониторингового интерфейса для сервера баз данных, который поддерживает множество пользователей, вошедших в систему, и множество выполняемых в данный момент запросов. Когда пользователь входит в систему или выходит из нее или выполнение запроса начинается или заканчивается, объект `ServerStatus` обновляется путем вызова соответствующего метода `add` или `remove`. Эти два типа информации полностью независимы; `ServerStatus` можно даже расщепить на два отдельных класса без потери функциональности.

Вместо защиты пользователей `users` и запросов `queries` с помощью замка `ServerStatus` мы можем защитить каждый из них отдельным замком, как показано в листинге 11.7. После разделения замка каждый новый более мелкий замок будет видеть меньше замкового трафика, чем исходный более грубый замок. (Делегирование в реализацию потокобезопасного множества `Set` для `users` и `queries` вместо явной синхронизации неявным образом обеспечит разделение замков, так как каждый `Set` будет использовать для защиты своего состояния другой замок.)

Листинг 11.6. Кандидат на разделение блокировки

```
@ThreadSafe
public class ServerStatus {
    @GuardedBy("users") public final Set<String> users;
    @GuardedBy("queries") public final Set<String> queries;
    ...
    public void addUser(String u) {
        synchronized (users) {
            users.add(u);
        }
    }

    public void addQuery(String q) {
        synchronized (queries) {
            queries.add(q);
        }
    }
    // удалить методы, перестроенные схожим образом, для использования
    // разделенных замков
}
```

Листинг 11.7. ServerStatus реорганизован для использования разделенных блокировок

```
@ThreadSafe
public class ServerStatus {
    @GuardedBy("this") public final Set<String> users;
    @GuardedBy("this") public final Set<String> queries;
    ...
    public synchronized void addUser(String u) { users.add(u); }
    public synchronized void addQuery(String q) { queries.add(q); }
    public synchronized void removeUser(String u) {
        users.remove(u);
    }
    public synchronized void removeQuery(String q) {
        queries.remove(q);
    }
}
```

Разделение на два замка предусматривает наибольшую возможность для улучшения, когда замок испытывает умеренный, но не сильный конфликт. Разделение замков, которые испытывают малый конфликт, дает мало Листинг 10.5 чистого улучшения производительности или пропускной способности, хотя это может увеличить порог нагрузки, при котором

производительность начинает снижаться из-за конфликта. Разделение замков, которые испытывают умеренный конфликт, может фактически превратить их в основном в неоспариваемые замки, что является наиболее желательным результатом как для производительности, так и для масштабируемости.

11.4.3. Чередование блокировок

Разделение одного сильно оспариваемого замка на два, вероятно, приведет к двум сильно оспариваемым замкам. Хотя это породит небольшое улучшение масштабируемости, позволив двум потокам выполняться конкурентно вместо одного, это все же незначительно улучшает перспективы конкурентности в системе с большим числом процессоров. Пример замкового разделения в классах `ServerStatus` не предоставляет очевидной возможности для дальнейшего чередования замков.

Замковое разделение иногда может быть расширено до секционной замковой защиты на переменно-размерном множестве независимых объектов. В этом случае оно называется *чередованием блокировок* (lock striping). Например, реализация хеш-массива `ConcurrentHashMap` использует массив из 16 замков, каждый из которых защищает 1/16 хеш-корзин; корзина N защищена замком N mod 16. Исходя из того, что хеш-функция обеспечивает разумные характеристики разброса, а ключи доступны равномерно, это должно уменьшить спрос на любой конкретный замок примерно в 16 раз. Именно такое техническое решение позволяет хеш-массиву `ConcurrentHashMap` поддерживать до 16 конкурентных писателей. (Число замков может быть увеличено, обеспечив еще более высокую конкурентность при интенсивном доступе в многопроцессорных системах, но число полос следует увеличивать за пределы стандартных 16, только если у вас есть веские доказательства того, что конкурентные писатели генерируют объем конфликта, достаточный для того, чтобы оправдать повышение лимита.)

Одним из недостатков чередования блокировок является то, что замковая защита коллекции для эксклюзивного доступа сложнее и более дорогостоящая, чем с единственным замком. Обычно операция может выполняться путем приобретения не более одного замка, но иногда необходимо запереть всю коллекцию, например, когда хеш-массиву `ConcurrentHashMap` необходимо расширить массив и заново хешировать значения в более

крупное множество корзин. Обычно это делается путем приобретения всех замков в наборе полос[1].

Класс `StripedMap` в листинге 11.8 иллюстрирует реализацию хеш-массива с помощью чередования блокировок. Имеется `N_LOCKS` замков, каждый из которых защищает подмножество корзин. Большинство методов, таких как `get`, нуждаются в приобретении только одного корзинного замка. Некоторые методы, возможно, должны приобретать все замки, но, как и в реализации для `clear`, возможно, не должны приобретать их все одновременно[2].

11.4.4. Недопущение горячих полей

Техники lock splitting и lock striping могут улучшить масштабируемость, так как позволяют разным потокам работать с разными данными (или разными порциями одной и той же структуры данных), не мешая друг другу. Программа, которая выиграла бы от чередования блокировок, с неизбежностью демонстрирует конфликт за *замок* чаще, чем за *данные*, защищаемые этим замком. Если замок защищает две независимые переменные *X* и *Y*, и поток *A* хочет обратиться к *X*, а поток *B* хочет обратиться к *Y* (как было бы в случае, если бы один поток вызывал метод `adduser`, в то время как другой вызывал метод `addQuery`, в классе `ServerStatus`), то эти два потока не конфликтуют ни за какие данные, даже если они конфликтуют за замок.

---

[1] Единственный способ получить произвольное множество внутренних замков — это рекурсия.

[2] Очистка ассоциативного массива `Map` таким способом не является атомарной, поэтому не обязательно существует время, когда массив `stripedmap` фактически является пустым, если другие потоки конкурентно добавляют элементы; создание атомарной операции потребует приобретения всех замков сразу. Однако для конкурентных коллекций, которые клиенты обычно не могут защитить замком для эксклюзивного доступа, результат таких методов, как `size` или `isEmpty`, в любом случае может быть устаревшим к тому времени, когда они возвращаются, поэтому такое поведение обычно является приемлемым, хотя, возможно, и несколько удивительным.

Листинг 11.8. Таблица на основе хеша с использованием чередования блокировок

```
@ThreadSafe
public class StripedMap {
    // Политика синхронизации: корзины[n], защищаемые замками[n%N_LOCKS]
     private static final int N_LOCKS = 16;
     private final Node[] buckets;
     private final Object[] locks;

     private static class Node { ... }

     public StripedMap(int numBuckets) {
         buckets = new Node[numBuckets];
         locks = new Object[N_LOCKS];
         for (int i = 0; i < N_LOCKS; i++)
             locks[i] = new Object();
     }

     private final int hash(Object key) {
         return Math.abs(key.hashCode() % buckets.length);
     }

     public Object get(Object key) {
         int hash = hash(key);
         synchronized (locks[hash % N_LOCKS]) {
             for (Node m = buckets[hash]; m != null; m = m.next)
                 if (m.key.equals(key))
                     return m.value;
         }
         return null;
     }

     public void clear() {
         for (int i = 0; i < buckets.length; i++) {
             synchronized (locks[i % N_LOCKS]) {
                 buckets[i] = null;
             }
         }
     }
     ...
}
```

Замковую гранулярность нельзя сократить, если для каждой операции требуются переменные. Это еще один участок, где сырая производитель-

ность и масштабируемость часто расходятся друг с другом; общепринятые приемы оптимизации, такие как кэширование часто вычисляемых значений, могут вносить «горячие поля», которые ограничивают масштабируемость.

Если бы вы реализовывали хеш-массив `HashMap`, то у вас был бы выбор того, как метод `size` вычисляет число элементов на ассоциативном массиве. Самый простой подход заключается в подсчете числа элементов при каждом его вызове. Общепринятой оптимизацией является обновление отдельного счетчика по мере добавления или удаления элементов; это немного увеличивает стоимость операции `put` или `remove`, которые поддерживают счетчик в актуальном состоянии, но сокращает стоимость операции `size` с $O(n)$ до $O(1)$.

Хранение отдельного счетчика для ускорения операций, таких как `size` и `isEmpty`, отлично работает для однопоточной или полностью синхронизированной реализации, но значительно усложняет улучшение масштабируемости реализации, поскольку каждая операция, модифицирующая ассоциативный массив, теперь должна обновлять совместный счетчик. Даже если вы используете чередование блокировок для хеш-цепочек, синхронизация доступа к счетчику заново привносит проблемы масштабируемости, присущие защите от исключающих блокировок. То, что выглядело как оптимизация производительности — кэширование результатов операции `size`, — превратилось в помеху для масштабируемости. В этом случае счетчик называется «*горячим полем*» (hot field), потому что каждая мутационная операция должна иметь к нему доступ.

Хеш-массив `ConcurrentHashMap` позволяет избежать этой проблемы, имея операцию `size`, которая перечисляет полосы и складывает число элементов в каждой полосе, вместо поддержания глобального счетчика. Для того чтобы избежать перечисления каждого элемента, класс `ConcurrentHashMap` поддерживает отдельное поле `count` для каждой полосы, также защищенное полосным замком[1].

[1] Если операция `size` вызывается часто по сравнению с мутационными операциями, то полосные структуры данных могут оптимизировать это путем кэширования размера коллекции в `volatile` при каждом вызове `size` и обнуляя кэш (устанавливая его равным –1) всякий раз, когда коллекция изменена. Если кэшированное значение является неотрицательным на входе в `size`, то оно является точным и может быть возвращено; в противном случае оно вычисляется заново.

11.4.5. Альтернативы исключающим блокировкам

Третье техническое решение для смягчения последствий конфликта блокировки заключается в отказе от использования исключающих блокировок в пользу более дружественных к конкурентности средств управления совместным состоянием. К ним относятся использование конкурентных коллекций, замков чтения-записи, немутируемых объектов и атомарных переменных.

Класс `ReadWriteLock` (см. главу 13) обеспечивает соблюдение замковой дисциплины «многочисленные читатели — единственный писатель»: более одного читателя могут обращаться к совместному ресурсу конкурентно, если никто из них не хочет его модифицировать, но писатели должны приобретать замок исключительно. Для структур данных с преобладанием чтения класс `ReadWriteLock` может предложить более высокую конкурентность, чем исключающая блокировка; для структур данных только для чтения немутируемость может полностью устранить потребность в замковой защите.

Атомарные переменные (см. главу 15) предлагают средства сокращения стоимости обновления «горячих полей», таких как статистические счетчики, генераторы последовательностей или ссылки на первый узел в связной структуре данных. (Мы использовали `AtomicLong` для поддержания счетчика посещений в примерах сервлетов в главе 2.) Классы атомарных переменных предоставляют очень мелкие (и, следовательно, более масштабируемые) атомарные операции над целыми числами или объектными ссылками и реализуются с помощью низкоуровневых конкурентных примитивов (таких как «сравнить и обменять»), предоставляемых большинством современных процессоров. Если ваш класс имеет небольшое число горячих полей, которые не участвуют в инвариантах с другими переменными, то замена их атомарными переменными может улучшить масштабируемость. (Изменение вашего алгоритма на меньшее число горячих полей может улучшить масштабируемость еще больше — атомарные переменные сокращают стоимость обновления горячих полей, но они не устраняют эти издержки.)

11.4.6. Мониторинг задействованности процессоров

Во время тестировании на масштабируемость целью обычно является поддержание процессоров в полностью задействованном состоянии. Такие

инструменты, как `vmstat` и `mpstat` в системах Unix или `perfmon` в системах Windows, могут сообщить вам, насколько «горячими» процессоры являются во время работы.

Если процессоры используются асимметрично (некоторые процессоры работают на повышенной температуре, а другие — нет), то первая цель должна состоять в том, чтобы найти в программе повышенный параллелизм. Асимметричная задействованность означает, что бóльшая часть вычислений происходит в небольшом наборе потоков, и ваше приложение не сможет воспользоваться преимуществами дополнительных процессоров.

Если процессоры задействованы не полностью, то вам необходимо выяснить причину. Существует несколько вероятных причин.

Недостаточная нагрузка. Возможно, тестируемое приложение просто не подвергается достаточной нагрузке. Это можно проверить путем увеличения нагрузки и измерения изменений в задействованности, времени отклика или времени обслуживания. Создание достаточной нагрузки для насыщения приложения может потребовать значительной компьютерной мощности; проблема может заключаться в том, что на пределе возможностей работает не тестируемая система, а клиентские системы.

Привязанность к вводу-выводу. С помощью `iostat` или `perfmon` вы можете установить, привязано или нет приложение к диску, а также путем мониторинга уровня трафика в сети выяснить, ограничена или нет пропускная способность.

Внешняя ограниченность. Если приложение зависит от внешних служб, таких как база данных или веб-служба, то узкое место, возможно, находится не в вашем коде. Это можно проверить с помощью профилировщика или средств администрирования баз данных, которые позволяют определить, сколько времени тратится на ожидание откликов от внешней службы.

Конфликт блокировки. Средства профилирования могут указать на то, сколько в приложении возникает конфликта блокировки и какие замки являются «горячими». Часто можно получить ту же информацию без профилировщика путем случайной выборки, запустив несколько поточных дампов и произведя поиск потоков, конфликту-

ющих за замки. Если поток блокирован в ожидании замка, то соответствующий стековый фрейм в поточном дампе выдает сообщение «ожидание замкового монитора...» (waiting to lock monitor). Замки, за которые в основном нет конфликта, появляются в поточном дампе редко; сильно оспариваемые замки почти всегда имеют хотя бы один поток, который ожидает его приобретения, и поэтому появляются в поточных дампах часто.

Если ваше приложение поддерживает процессоры в достаточно «горячем» состоянии, то вы можете применить средства мониторинга, для того чтобы определить полезность для него дополнительных процессоров. Программа, имеющая только четыре потока, возможно, будет способна задействовать 4-ядерную систему полностью, но вряд ли увидит повышение производительности при переходе на 8-ядерную систему, так как там ей придется ждать, когда работоспособные потоки воспользуются преимуществами дополнительных процессоров. (Вы также можете переконфигурировать программу с целью разделить ее рабочую нагрузку на большее число потоков, например настроить размер поточного пула.) Одним из столбцов, о котором сообщает `vmstat`, является число потоков, которые работоспособны, но не работают в данный момент, поскольку нет в наличии процессора. Если использование процессора высоко и всегда имеются работоспособные потоки, ожидающие процессор, то от большего числа процессоров ваше приложение, скорее всего, выиграет.

11.4.7. Объединению объектов в пул — «нет»!

В ранних версиях JVM выделение объектов и сбор мусора были медленными[1], но с тех пор их производительность значительно улучшилась. На самом деле выделение памяти в Java теперь быстрее, чем `malloc` в *C*: обычная ветвь кода для инструкции `new Object` в HotSpot 1.4.X и 5.0 составляет примерно десять машинных команд.

Для того чтобы обойти «медленные» жизненные циклы объектов, многие разработчики обратились к объединению объектов в пул, где объекты рециркулируются, а не собираются сборщиком мусора и выделяются заново при необходимости. Даже с учетом сокращения издержек на сбор мусора

[1] Как и все остальное — синхронизация, графика, запуск JVM, рефлексия — предсказуемо являются такими в первой версии экспериментальной технологии.

было показано, что объединение объектов в пул приводит к потере производительности[1] для всех, кроме самых дорогостоящих объектов (и серьезной потере для легковесных и средневесных объектов) в однопоточных программах (Click, 2005).

В конкурентных приложениях объединение в пул находится в еще худшем положении. Когда потоки осуществляют выделение новых объектов, требуется очень мало координации между потоками, так как распределители ресурсов обычно используют локальные для потоков блоки выделения, устраняя бóльшую часть синхронизации на кучевых структурах данных. И напротив, если эти потоки запрашивают объект из пула, то необходима некоторая синхронизация для координации доступа к структуре данных пула, что создает возможность того, что поток будет блокирован. Поскольку блокирование потока из-за конфликта блокировки в сотни раз дороже, чем выделение объекта, даже небольшой индуцированный пулом конфликт будет узким местом масштабируемости. (Даже незапланированная синхронизация обычно обходится дороже, чем выделение объекта.) Это еще одно техническое решение, которому приписывалась оптимизация производительности, но которое превратилось в угрозу для масштабируемости. Объединение в пул имеет свои применения[2], но оно имеет ограниченную полезность в качестве оптимизации производительности.

Выделение объектов обычно обходится дешевле, чем синхронизация.

---

[1] В дополнение к тому, что оно является потерей в терминах процессорных циклов, объединение объектов в пул имеет ряд других проблем, среди которых проблема правильной установки размера пула (слишком малый пул не имеет никакого эффекта, слишком крупный оказывает давление на сборщика мусора, удерживая память, которая могла бы быть использована эффективнее на что-то другое); риск того, что объект не будет надлежащим образом инициализироваться заново в только что выделенное состояние, привнося едва различимые дефекты; риск того, что поток будет возвращать объект в пул, но продолжать его использовать; и что это создает больше работы для поколенческих сборщиков мусора, поощряя шаблон ссылок от старых к молодым.

[2] В ограниченных средах, таких как некоторые целевые объекты J2ME или RTSJ, объединение объектов в пул по-прежнему может потребоваться для эффективного управления памятью или отзывчивостью.

11.5. Пример: сравнение производительности ассоциативного массива Map

Однопоточная производительность хеш-массива `ConcurrentHashMap` слегка лучше, чем у синхронизированного хеш-массива `HashMap`, но именно в конкурентном применении она действительно блещет. Реализация хеш-массива `ConcurrentHashMap` исходит из допущения, что наиболее распространенной операцией является извлечение значения, которое уже существует, и поэтому оптимизирована для обеспечения максимальной производительности и конкурентности для успешных операций `get`.

Главным препятствием для масштабируемости в синхронизированных реализациях ассоциативного массива `Map` является то, что есть только один замок для всего массива, поэтому только один поток может обратиться к массиву за раз. С другой стороны, хеш-массив `ConcurrentHashMap` не имеет замковой защиты для большинства успешных операций чтения и использует замковое разделение на полосы для операций записи и тех немногих операций чтения, которые требуют замковой защиты. В результате многочисленные потоки могут обращаться к массиву конкурентно без блокирования.

Рис. 11.3 иллюстрирует различия в масштабируемости между несколькими ассоциативными массивами: `ConcurrentHashMap`, `ConcurrentSkipListMap`, а также `HashMap` и `TreeMap`, обернутыми в `synchronizedMap`. Первые два являются потокобезопасными по своей конструкции; последние два сделаны потокобезопасными синхронизированной оберткой. В каждом прогоне N потоков конкурентно выполняют плотный цикл[1], который отбирает случайный ключ и пытается получить значение по этому ключу. Если значение отсутствует, то оно добавляется в ассоциативный массив с вероятностью $p = .6$, и если оно присутствует, то удаляется с вероятностью $p = .02$. Тесты прогонялись в предрелизной сборке Java 6 на 8-ядерном Sparc V880, и график показывает пропускную способность, нормализованную по однопоточному случаю для хеш-массива `ConcurrentHashMap`. (Разрыв

[1] Плотный цикл (*tight loop*) — это цикл, который содержит мало инструкций и повторяется много раз. Такой цикл сильно задействует ресурсы ввода-вывода или обработки. — *Примеч. пер.*

по масштабируемости между конкурентными и синхронизированными коллекциями еще больше в Java 5.0.)

Данные для массивов `ConcurrentHashMap` и `ConcurrentSkipListMap` показывают, что они хорошо масштабируются для больших чисел потоков; пропускная способность продолжает улучшаться по мере добавления потоков. Хотя число потоков на рис. 11.3 может показаться небольшим, эта тестовая программа создает больше конфликтов на поток, чем типичное приложение, поскольку мало что делает кроме того, что «барабанит» по массиву; реальная программа каждую итерацию будет выполнять дополнительную локальную для потока работу.

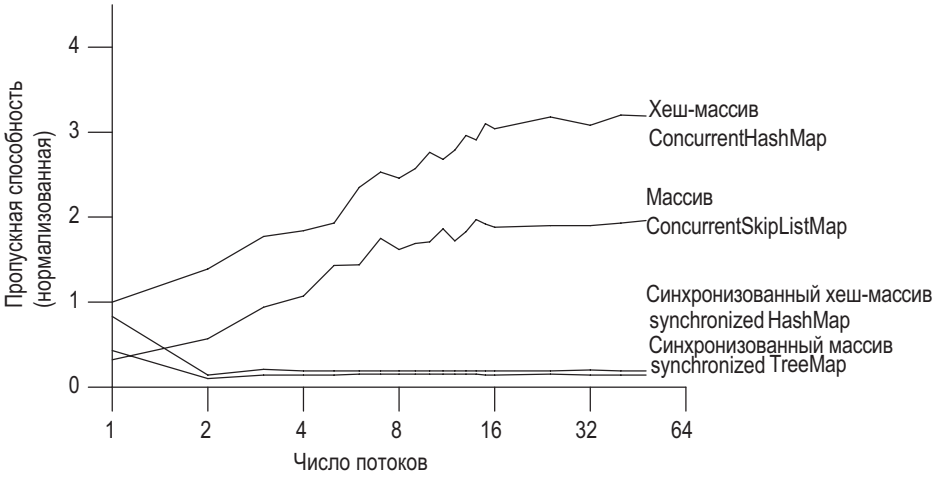

Рис. 11.3. Сравнение масштабируемости реализаций ассоциированного массива Map

Цифры для синхронизированных коллекций не столь обнадеживают. Производительность для однопоточного случая сравнима с хеш-массивом `ConcurrentHashMap`, но как только нагрузка переходит от в основном не оспариваемой к в основном оспариваемой — что и происходит здесь в двух потоках, — синхронизированные коллекции сильно страдают. Это обычное поведение для кода, масштабируемость которого ограничена конфликтом блокировки. До тех пор пока конфликт блокировки является низким, над показателем времени на операцию доминирует время, затрачиваемое на то, чтобы сделать работу фактически, и пропускная способность может улучшиться по мере добавления потоков. Как только конфликт блоки-

ровки становится значительным, над показателем времени на операцию доминирует переключение контекста и задержки планирования, и добавление большего числа потоков имеет небольшое влияние на пропускную способность.

11.6. Сокращение издержек на переключение контекста

Многие задачи предусматривают операции, которые могут блокировать; переход между работающим и блокированным состояниями влечет за собой переключение контекста. Одним из источников блокирования в серверных приложениях является создание журнальных сообщений в процессе обработки запросов; в качестве иллюстрации того, как можно повысить пропускную способность за счет уменьшения переключений контекста, мы проанализируем планировочное поведение двух подходов к журналированию.

Большинство структур журналирования представляют собой тонкие обертки вокруг инструкции `println`; когда у вас есть что-то для записи в журнал регистрации, просто напишите это прямо здесь и сейчас. Еще один подход был показан в классе `LogWriter` на с. 198: журналирование выполняется не запрашивающим потоком, а в выделенном фоновом потоке. С точки зрения разработчика оба подхода примерно равноценны. Однако производительность может отличаться — в зависимости от объема журналирования, числа потоков, выполняющих журналирование, и других факторов, таких как стоимость переключения контекста[1].

Время обслуживания операции журналирования включает в себя все вычисления, ассоциированные с классами стандартного потока ввода-вы-

[1] Создание диспетчера журналирования, который переносит ввод-вывод в другой поток, может повысить производительность, но это также вносит ряд проектных осложнений, таких как прерывание (что происходит, если поток, блокированный в операции журналирования, прерывается?), гарантии службы (гарантирует ли диспетчер, что успешно помещенное в очередь журнальное сообщение будет зарегистрировано в журнале до выключения?), политика насыщения (что происходит, когда производители регистрируют сообщения быстрее, чем поток диспетчера может их обработать?) и жизненный цикл службы (как мы завершаем работу диспетчера и как мы сообщаем состояние службы производителям?).

вода; если операция ввода-вывода блокирует продвижение, то оно также включает в себя время, в течение которого поток блокирован. Операционная система выводит блокированный поток за рамки планирования до тех пор, пока не завершится ввод-вывод, и, вероятно, еще немного дольше. Когда ввод-вывод завершится, другие потоки, вероятно, будут активными, и им будет разрешено завершить свои кванты планирования, и, возможно, потоки уже будут ожидать впереди нас в планировочной очереди — еще более увеличивая срок обслуживания. В качестве альтернативы, если многочисленные потоки выполняют журналирование одновременно, то может возникнуть конфликт блокировки за стандартный выходной поток, в каковом случае результат будет таким же, как и с блокирующим вводом-выводом — поток блокирует продвижение в ожидании замка и отключается. Встроенное журналирование предусматривает ввод-вывод и замковую защиту, что может привести к увеличению переключения контекста и, следовательно, к увеличению времени обслуживания.

Увеличение времени обслуживания запросов является нежелательным по нескольким причинам. Во-первых, время обслуживания влияет на качество обслуживания: более длительное время обслуживания означает, что кто-то ждет результата дольше. Но что более важно, более длительное время обслуживания в данном случае означает больше случаев конфликта блокировки. Принцип «вошел, вышел» из раздела 11.4.1 говорит нам, что мы должны владеть замками как можно короче, потому что чем дольше замок занят, тем больше вероятность того, что замок будет оспариваться. Если поток блокируется в ожидании ввода-вывода, владея замком, то другой поток, скорее всего, захочет иметь этот замок, пока первый поток им владеет. Конкурентные системы работают намного лучше, когда большинство замковых приобретений не являются обязательными, потому что приобретение оспариваемого замка означает больше переключений контекста. Стиль кодирования, который поощряет больше переключений контекста, следовательно, дает более низкую совокупную пропускную способность.

Перемещение ввода-вывода за пределы потоков обработки запросов может сократить среднее время обслуживания для обработки запросов. Потоки, вызывающие операцию `log`, больше не блокируются в ожидании замка стандартного выходного потока либо завершения ввода-вывода; им нужно только разместить сообщение в очередь и затем вернуться к своей задаче. С другой стороны, мы внесли возможность конфликта за очередь сообщений, но операция `put` имеет меньший вес, чем журнальный

ввод-вывод (который может потребовать системных вызовов), и поэтому с меньшей вероятностью блокируется в фактическом использовании (если очередь не заполнена). Поскольку поток запроса теперь с меньшей вероятностью блокируется, он с меньшей вероятностью будет контекстно выключен в середине запроса. То, что мы сделали, превратило сложную и неопределенную ветвь кода с вводом-выводом и возможным конфликтом блокировки в прямолинейную ветвь кода.

В какой-то степени мы просто переносим работу туда-сюда, перемещая ввод-вывод в поток, где его стоимость не воспринимается пользователем (что само по себе может быть выигрышем). Но, перемещая *весь* ввод-вывод журналирования в единственный поток, мы также исключаем возможность конфликта за стандартный выходной поток и, таким образом, устраняем источник блокирования. Это повышает совокупную пропускную способность, поскольку потребляется меньше ресурсов при планировании, переключении контекста и управлении замками.

Вынесение ввода-вывода из многих потоков обработки запросов в единственный поток диспетчера журналирования аналогично разнице между ведерной бригадой и пожарной командой. В подходе «беготни ста парней туда-сюда с ведрами» у вас больше шансов конфликтовать у источника воды и у огня (в результате чего в целом меньше воды будет доставлено к огню) плюс более высокая неэффективность, потому что каждый работник постоянно переключает режимы (заполнение водой, бег, выливание, бег и т. д.). При подходе «ведерная бригада» поток воды от источника к горящему зданию является постоянным, затрачивается меньше энергии на транспортировку воды к огню, и каждый работник сосредоточивается на выполнении одной работы непрерывно. Как и в случае прерываний, которые создают помехи людям и снижают их производительность, блокирование и переключение контекста создают помехи для потоков.

Итоги

Поскольку одной из наиболее распространенных причин использования потоков является эксплуатирование многочисленных процессоров, при обсуждении производительности конкурентных приложений нас обычно больше волнует пропускная способность либо масштабируемость, чем сырое время обслуживания. Закон Амдала гласит, что масштабируемость приложения определяется долей кода, который должен выполняться

сериализованно (последовательно). Поскольку основным источником сериализации в программах Java является исключающая ресурсная блокировка, масштабируемость часто можно улучшить, тратя меньше времени на удержание блокировки, что достигается путем уменьшения степени детализации замков, сокращения продолжительности удержания блокировки либо замены исключающих блокировок неэксклюзивными или неблокирующими альтернативами.

12 Тестирование конкурентных программ

Конкурентные программы задействуют проектные принципы и паттерны, схожие с последовательными программами. Разница лишь в том, что конкурентные программы имеют степень недетерминизма, которую последовательные программы не имеют, увеличивая число потенциальных взаимодействий и режимов сбоя, которые должны быть спланированы и проанализированы.

Схожим образом в тестировании конкурентных программ используются и расширяются идеи тестирования последовательных программ. Те же самые технические решения, которые применимы для тестирования правильности и производительности последовательных программ, могут применяться и к конкурентным программам, но в случае конкурентных программ пространство того, что может пойти не так, намного больше. Главная проблема при конструировании тестов для конкурентных программ заключается в том, что потенциальные сбои могут быть скорее редкими вероятностными происшествиями, чем детерминированными; тесты, раскрывающие такие сбои, должны быть более обширными и выполняться дольше, чем типичные последовательные тесты.

Большинство тегов конкурентных классов относятся к одной или обеим классическим категориям — *безопасность* и *жизнеспособность*. В главе 1 мы определили безопасность как «ничего плохого никогда не происходит», а жизнеспособность как «что-то хорошее в конце концов происходит».

Тесты на безопасность, которые верифицируют соответствие поведения класса его спецификации, обычно принимают форму тестовых инвариантов. Например, в реализации связного списка, которая кэширует размер списка всякий раз, когда он модифицируется, один из тестов на безопасность состоит в сравнении числа кэшированных элементов с фактическим числом элементов в списке. В однопоточной программе это делается легко, так как содержимое списка не изменяется во время проверки его свойств. Но в конкурентной программе такой тест, скорее всего, будет чреват гонками, если вы не сможете отслеживать поле счетчика и подсчитывать элементы за единую атомарную операцию. Это можно сделать, заперев список для исключающего доступа, привлекая некий предоставляемый реализацией функционал «атомарного снимка» либо применяя предоставляемые им же «тестовые точки», которые позволяют тестам подтверждать (с помощью `assert`) инварианты или выполнять тестовый код атомарно.

В этой книге мы использовали хронометражные диаграммы для описания «неудачных» взаимодействий, которые могут приводить к сбоям в неправильно сконструированных классах; тестовые программы пытаются разведать объем пространства состояний, достаточный для того чтобы такая неудача в итоге произошла. К сожалению, тестовый код может содержать артефакты временной координации или синхронизации, которые могут маскировать дефекты, в иных случаях не проявляющиеся бы[1].

Свойства жизнеспособности привносят свои собственные сложности для тестирования. Тесты на жизнеспособность включают тесты на продвижение и непродвижение, которые трудно квантифицировать, — как проверить, что метод блокируется, а не просто работает медленно? И то же самое: как проверить, что алгоритм *не* был заперт взаимной блокировкой? Как долго следует ждать, прежде чем объявить его безуспешность?

С тестами на жизнеспособность связаны тесты на производительность. Производительность может измеряться несколькими путями, включая:

пропускную способность: скорость, с которой завершился набор конкурентных задач;

отзывчивость: задержка между запросом и завершением какого-либо действия (также именуемая *запаздыванием*);

[1] Ошибки, которые исчезают при добавлении отладочного или проверочного кода, игриво называются *гейзенбагами* (Heisenbug).

масштабируемость: улучшение пропускной способности (либо ее отсутствие) по мере увеличения количества ресурсов (обычно процессоров).

12.1. Тестирование на правильность

Разработка модульных тестов для конкурентного класса начинается с того же анализа, что и для последовательного класса, — идентификации инвариантов и постусловий, поддающихся механической проверке. Если вам повезет, то многие из них присутствуют в спецификации; в остальное время написание тестов выливается в приключение по итеративному обнаружению спецификации.

В качестве конкретной иллюстрации мы намерены построить набор тестовых случаев для ограниченного буфера. В листинге 12.1 показана наша реализация `BoundedBuffer`, использующая семафор для реализации необходимого ограничения и блокирования.

Класс `BoundedBuffer` реализует очередь на основе постоянно-размерного массива с блокирующими методами `put` и `take`, управляемыми парой счетных семафоров. Семафор `availableItems` представляет число элементов, которые могут быть *удалены* из буфера, и изначально равен нулю (так как буфер изначально пуст). Схожим образом семафор `availableSpaces` представляет число элементов, которые можно вставить в буфер, и инициализируется размером буфера.

Операция `take` сначала требует, чтобы из семафора `availableItems` было получено разрешение. Эта операция сразу же является успешной, если буфер не пустой, и в противном случае блокируется до тех пор, пока буфер не станет не пустым. Как только разрешение получено, из буфера удаляется следующий элемент, и разрешение освобождается и возвращается в семафор `availableSpaces`[1]. Операция `put` определяется, наоборот, так, чтобы на выходе из методов `put` либо `take` сумма счетчиков обоих семафоров всегда была равна лимиту. (На практике, если вам нужен ограниченный буфер, вы должны использовать очереди `ArrayBlockingQueue` или `LinkedBlockingQueue`, а не разворачивать свой собственный, но ис-

[1] В счетном семафоре разрешения не представлены явно или не ассоциированы с владеющим потоком; операция `release` создает разрешение, а операция `acquire` его потребляет.

пользуемое здесь техническое решение иллюстрирует то, как вставками и удалениями можно управлять и в других структурах данных.)

Листинг 12.1. Ограниченный буфер с использованием семафора

```
@ThreadSafe
public class BoundedBuffer<E> {
    private final Semaphore availableItems, availableSpaces;
    @GuardedBy("this") private final E[] items;
    @GuardedBy("this") private int putPosition = 0, takePosition = 0;

    public BoundedBuffer(int capacity) {
        availableItems = new Semaphore(0);
        availableSpaces = new Semaphore(capacity);
        items = (E[]) new Object[capacity];
    }

    public boolean isEmpty() {
        return availableItems.availablePermits() == 0;
    }

    public boolean isFull() {
        return availableSpaces.availablePermits() == 0;
    }

    public void put(E x) throws InterruptedException {
        availableSpaces.acquire();
        doInsert(x);
        availableItems.release();
    }

    public E take() throws InterruptedException {
        availableItems.acquire();
        E item = doExtract();
        availableSpaces.release();
        return item;
    }

    private synchronized void doInsert(E x) {
        int i = putPosition;
        items[i] = x;
        putPosition = (++i == items.length)? 0 : i;
    }

    private synchronized E doExtract() {
        int i = takePosition;
```

```
        E x = items[i];
        items[i] = null;
        takePosition = (++i == items.length)? 0 : i;
        return x;
    }
}
```

12.1.1. Базовые модульные тесты

Самые базовые модульные тесты для `BoundedBuffer` аналогичны тем, которые мы использовали бы в последовательном контексте, — создать ограниченный буфер, вызывать его методы и подтверждать постусловия и инварианты. Некоторые инварианты, которые быстро приходят на ум, состоят в том, что только что созданный буфер должен идентифицировать себя как пустой и как не полный. Аналогичный, но несколько более сложный тест на безопасность состоит в том, чтобы вставить N элементов в буфер емкостью N (что должно быть успешно выполнено без блокирования) и проверить, что буфер распознает, что он полон (а не пуст). Тестовые методы JUnit для этих свойств показаны в листинге 12.2.

Листинг 12.2. Базовые модульные тесты для BoundedBuffer

```
class BoundedBufferTest extends TestCase {
    void testIsEmptyWhenConstructed() {
        BoundedBuffer<Integer> bb = new BoundedBuffer<Integer>(10);
        assertTrue(bb.isEmpty());
        assertFalse(bb.isFull());
    }

    void testIsFullAfterPuts() throws InterruptedException {
        BoundedBuffer<Integer> bb = new BoundedBuffer<Integer>(10);
        for (int i = 0; i < 10; i++)
            bb.put(i);
        assertTrue(bb.isFull());
        assertFalse(bb.isEmpty());
    }
}
```

Эти простые тестовые методы являются полностью последовательными. Включение набора последовательных тестов в свой комплект тестов часто бывает полезным, так как они могут раскрывать ситуации, когда задача *не* связана с вопросами конкурентности, прежде чем вы начнете искать ситуации гонки данных.

12.1.2. Тестирование блокирующих операций

Тесты существенных свойств конкурентности требуют введения более одного потока. Большинство тестовых структур не очень дружественны к конкурентности: они редко включают механизмы для создания потоков или их мониторинга для обеспечения того, чтобы они не умирали неожиданно. Если вспомогательный поток, созданный тестовым случаем, обнаруживает сбой, то структура обычно не знает, с каким тестом этот поток ассоциирован, поэтому может потребоваться некоторая работа для предоставления данных при возврате к главному, выполняющему тесты потоку для отчетности.

В тестах на соответствие для `java.util.concurrent` было важно, чтобы сбои были четко ассоциированы с конкретным тестом. В результате экспертная группа JSR 166 создала базовый класс[1], который предоставляет методы для ретрансляции и сообщения о сбоях во время операции демонтажа `tearDown`, следуя соглашению, что каждый тест должен ждать до тех пор, пока все потоки, которые он создал, терминируются. Вам, возможно, не потребуется углубляться в такие дебри; ключевые требования заключаются в том, чтобы было ясно, прошли или нет тесты успешно, и что информация о сбое где-то сообщается для использования в диагностировании проблемы.

Если метод предположительно будет блокироваться при определенных условиях, то тест такого поведения должен завершаться успешно только в том случае, если поток *не* продвигается вперед. Тестирование того, что метод блокируется, похоже на тестирование того, что метод выдает исключение; если метод возвращается нормально, то тест был безуспешным.

Тестирование того факта, что метод блокирует продвижение, вносит дополнительную сложность: как только метод успешно блокируется, вы должны убедить его как-то разблокировать продвижение. Очевидным способом сделать это является прерывание — запустить блокирующее действие в отдельном потоке, дождаться, когда поток будет блокироваться, прервать его, а затем подтвердить, что блокирующая операция завершена. Конечно, это требует, чтобы ваши блокирующие методы откликались на прерывание, возвращаясь досрочно или выдавая исключение `InterruptedException`.

---

[1] См. http://gee.cs.oswego.edu/cgi-bin/viewcvs.cgi/jsr166/src/test/tck/JSR166TestCase.java

Часто «дождаться, когда поток будет блокироваться» легче сказать, чем сделать; на практике вы должны принять произвольное решение о том, сколько времени могут занимать эти несколько выполняемых инструкций, и ждать дольше, чем указано. Вы должны быть готовы увеличить это значение, в случае если ошибаетесь (в этом случае вы увидите мнимые сбои теста).

В листинге 12.3 показан подход к тестированию блокирующих операций. Он создает «берущий» поток, который пытается взять (операцией `take`) элемент из пустого буфера. Если операция `take` оказывается успешной, он регистрирует неудачу. Тестирующий поток запускает берущий поток, долго ожидает, а затем прерывает его. Если берущий поток правильно блокирован в операции `take`, то он будет выдавать исключение `InterruptedException`, и блок кода `catch` для этого исключения рассматривает его как успех и позволяет потоку выйти. Затем главный тестирующий поток пытается выполнить операцию присоединения `join` с берущим потоком и проверяет, что данная операция успешно возвращается, вызывая `Thread.isAlive`; если берущий поток откликнулся на прерывание, то операция `join` должна завершиться быстро.

Листинг 12.3. Тестирование блокировок и отзывчивости на прерывание

```
void testTakeBlocksWhenEmpty() {
    final BoundedBuffer<Integer> bb = new BoundedBuffer<Integer>(10);
    Thread taker = new Thread() {
        public void run() {
            try {
                int unused = bb.take();
                fail(); // если мы добрались сюда, то значит ошибка
            } catch (InterruptedException success) { }
        }};
    try {
        taker.start();
        Thread.sleep(LOCKUP_DETECT_TIMEOUT);
        taker.interrupt();
        taker.join(LOCKUP_DETECT_TIMEOUT);
        assertFalse(taker.isAlive());
    } catch (Exception unexpected) {
        fail();
    }
}
```

Хронометрированная операция `join` обеспечивает, чтобы тест завершался, даже если операция `take` каким-то неожиданным образом застрянет. Этот

тестовый метод проверяет несколько свойств операции `take` — не только то, что он блокируется, но и то, что при прерывании он выдает исключение `InterruptedException`. Это один из немногих случаев, в которых уместно подклассировать `Thread` явно вместо использования `Runnable` в пуле: это делается для того, чтобы протестировать надлежащую терминацию с операцией `join`. Такой же подход можно использовать для проверки того, что берущий поток разблокируется после того, как главный поток помещает элемент в очередь.

Заманчиво использовать метод `Thread.getState` для верификации того, что поток фактически блокирован на ожидании по условию, но этот подход не является надежным. Нет ничего, что требовало бы от блокированного потока входить *когда-либо* в состояния `WAITING` или `TIMED_WAITING`, так как JVM вместо этого может реализовать блокирование за счет ожидания в холостом цикле (спин-ожидания). Аналогичным образом, поскольку разрешены мнимые пробуждения от `Object.wait` или `Condition.await` (см. главу 14), поток, находящийся в состоянии `WAITING` или `TIMED_WAITING`, может временно перейти в состояние `RUNNABLE`, даже если условие, которого он ожидает, еще не является истинным. Даже игнорируя эти варианты реализации, целевому потоку может потребоваться некоторое время для того, чтобы принять состояние блокирования. *Результат метода* `Thread.getState` *не следует использовать для управления конкурентностью, и он имеет ограниченную полезность для проведения тестов: его основная полезность — быть источником отладочной информации.*

12.1.3. Тестирование на безопасность

Тесты, приведенные в листингах 12.2 и 12.3, проверяют важные свойства ограниченного буфера, но вряд ли они обнаружат ошибки, вытекающие из ситуаций гонки данных. Для того чтобы протестировать, что конкурентный класс работает правильно при непредсказуемом конкурентном доступе, нам нужно выполнить настройку многочисленных потоков, дав им возможность выполнять операции `put` и `take` в течение некоторого времени, а затем каким-то образом проверить, что ничего не пошло неправильно.

Конструирование тестов для выявления ошибок безопасности в конкурентных классах является примером дилеммы «курица или яйцо»: сами тестовые программы являются конкурентными. Разработка хороших

конкурентных тестов может быть сложнее, чем разработка классов, которые они тестируют.

> Задача, стоящая перед конструированием эффективных тестов на безопасность для конкурентных классов, заключается в выявлении легко проверяемых свойств, которые с высокой вероятностью завершатся безуспешно, если что-то пойдет не так, в то же время не позволяя коду, который проводит аудит на предмет безуспешности, искусственно ограничивать конкурентность. Лучше всего, если проверка тестируемого свойства не будет требовать синхронизации.

Один подход, который хорошо работает с классами, используемыми в патернах «производитель-потребитель» (такими как ограниченный буфер `BoundedBuffer`), состоит в том, чтобы проверять, что все поставленное в очередь или буфер из него выходит и что ничего больше не делается. Наивная реализация этого подхода будет вставлять элемент в «теневой» список, когда он будет ставиться в очередь, удалять его из списка при удалении из очереди и подтверждать, что теневой список пуст по завершении теста. Но такой подход исказил бы планирование тестовых потоков, поскольку изменение теневого списка потребовало бы синхронизации и, возможно, блокирования.

Более качественным подходом является вычисление контрольных сумм элементов, которые помещаются в очередь и удаляются из нее с помощью чувствительной к порядку функции контрольных сумм, и их сравнение. Если они совпадают, то тест проходит успешно. Этот подход лучше всего работает, когда существует единственный производитель, помещающий элементы в буфер, и единственный потребитель, вынимающий их оттуда, потому что он может проверять не только то, что правильные элементы (возможно) вышли, но и то, что они вышли в правильном порядке.

Расширение этого подхода до ситуации с многочисленными производителями и многочисленными потребителями требует использования функции контрольной суммы, *не чувствительной* к порядку, в которой элементы объединяются, для того чтобы многочисленные контрольные суммы можно было объединить после теста. В противном случае синхронизация доступа к совместному полю контрольной суммы может стать узким местом конкурентности или исказить время теста. (Этим требованиям отвечает любая коммутативная операция, такая как сложение или XOR.)

Для обеспечения того, чтобы тест действительно тестировал то, что, по вашему мнению, он тестирует, важно, чтобы сами контрольные суммы не были очевидными для компилятора. Было бы плохой идеей в качестве тестовых данных использовать целые числа, идущие подряд, потому что тогда результат всегда будет одинаковым и умный компилятор может просто предварительно его вычислить.

Для того чтобы избежать этой проблемы, тестовые данные должны генерироваться случайно, но многие другие эффективные тесты компрометируются плохим выбором генератора случайных чисел (ГСЧ). Генерация случайных чисел может создавать стыки между классами и артефактами синхронизации, потому что большинство классов генератора случайных чисел — потокобезопасны и поэтому вносят дополнительную синхронизацию[1]. Предоставление каждого потока собственного ГСЧ позволяет использовать непотокобезопасный ГСЧ.

Вместо использования универсального ГСЧ лучше использовать простые псевдослучайные функции. Вам не нужна высококачественная случайность; все, что вам нужно, это случайность, достаточная для обеспечения изменчивости чисел от прогона к прогону. Функция `xorShift` в листинге 12.4 (Marsaglia, 2003) является одной из самых дешевых среднекачественных функций случайных чисел. Ее запуск со значениями, основанными на `hashCode` и `nanoTime`, делает суммы как неочевидными, так и почти всегда разными для каждого прогона.

Листинг 12.4. Среднекачественный генератор случайных чисел, подходящий для тестирования

```
static int xorShift(int y) {
    y ^= (y << 6);
    y ^= (y >>> 21);
    y ^= (y << 7);
    return y;
}
```

Класс `PutTakeTest` в листингах 12.5 и 12.6 запускает N производящих потоков, которые генерируют элементы и помещают их в очередь, и N потребляющих потоков, которые их вынимают из очереди. Каждый поток обновляет контрольную сумму элементов по мере их входа или выхода,

[1] Многие эталонные тесты, неведомые разработчикам или пользователям, просто проверяют, насколько ГСЧ является узким местом конкурентности.

используя контрольную сумму для каждого потока, который объединяется в конце тестового прогона с целью добавить не больше синхронизации или конфликта, чем требуется для тестирования буфера.

Листинг 12.5. Тестовая программа на основе паттерна «производитель-потребитель» для буфера BoundedBuffer

```
public class PutTakeTest {
    private static final ExecutorService pool
            = Executors.newCachedThreadPool();
    private final AtomicInteger putSum = new AtomicInteger(0);
    private final AtomicInteger takeSum = new AtomicInteger(0);
    private final CyclicBarrier barrier;
    private final BoundedBuffer<Integer> bb;
    private final int nTrials, nPairs;

    public static void main(String[] args) {
        new PutTakeTest(10, 10, 100000).test(); // образец параметров
        pool.shutdown();
    }

    PutTakeTest(int capacity, int npairs, int ntrials) {
        this.bb = new BoundedBuffer<Integer>(capacity);
        this.nTrials = ntrials;
        this.nPairs = npairs;
        this.barrier = new CyclicBarrier(npairs * 2 + 1);
    }

    void test() {
        try {
            for (int i = 0; i < nPairs; i++) {
                pool.execute(new Producer());
                pool.execute(new Consumer());
            }
            barrier.await(); // ждать, когда все потоки будут готовы
            barrier.await(); // ждать, когда все потоки завершатся
            assertEquals(putSum.get(), takeSum.get());
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

    class Producer implements Runnable { /* Листинг 12.6 */ }

    class Consumer implements Runnable { /* Листинг 12.6 */ }
}
```

Листинг 12.6. Классы производителя и потребителя, используемые в тесте PutTakeTest

```
/* внутренние для теста PutTakeTest классы (листинг 12.5) */
class Producer implements Runnable {
    public void run() {
        try {
            int seed = (this.hashCode() ^ (int)System.nanoTime());
            int sum = 0;
            barrier.await();
            for (int i = nTrials; i > 0; --i) {
                bb.put(seed);
                sum += seed;
                seed = xorShift(seed);
            }
            putSum.getAndAdd(sum);
            barrier.await();
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
}

class Consumer implements Runnable {
    public void run() {
        try {
            barrier.await();
            int sum = 0;
            for (int i = nTrials; i > 0; --i) {
                sum += bb.take();
            }
            takeSum.getAndAdd(sum);
            barrier.await();
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
}
```

В зависимости от вашей платформы создание и запуск потока может быть умеренно тяжеловесной операцией. Если поток является кратковременным и вы запускаете несколько потоков в цикле, то в худшем случае потоки работают последовательно, а не конкурентно. Даже не совсем в худшем случае тот факт, что первый поток имеет фору перед другими, означает, что вы можете получить меньше перемежений, чем ожидалось:

первый поток работает сам по себе в течение некоторого времени, затем первые два потока работают конкурентно в течение некоторого времени, и только в конце все потоки работают конкурентно. (То же самое происходит в конце прогона: потоки, которые получили фору, также заканчиваются досрочно.)

Для смягчения этой проблемы в разделе 5.5.1 мы представили техническое решение, используя защелку `CountDownLatch` в качестве стартового шлюза и еще одну в качестве финишного шлюза. Еще один способ получить тот же эффект — использовать циклический барьер `CyclicBarrier`, инициализируемый числом рабочих потоков плюс один, и дать рабочим потокам и тестовому драйверу ждать у барьера в начале и конце их прогона. Этим обеспечивается, что все потоки находятся в состоянии готовности до начала работы. Тест `PutTakeTest` использует это техническое решение для координирования запуска и останова рабочих потоков, создавая больше потенциальных конкурентных перемежений. Мы по-прежнему не можем гарантировать, что планировщик не будет прогонять каждый поток до завершения последовательно, но создание достаточно длинных прогонов уменьшает степень, до которой планирование искажает наши результаты.

Последняя хитрость, которая применена в тесте `PutTakeTest`, состоит в использовании критерия детерминированной терминации, благодаря которому не требуется никакой дополнительной координации между потоками, для того чтобы выяснить, когда тест будет завершен. Метод `test` запускает ровно столько производителей, сколько потребителей, и каждый из них помещает (`put`) или вынимает (`take`) одинаковое число элементов, поэтому общее число добавляемых и вынимаемых элементов является одинаковым.

Тесты, такие как `PutTakeTest`, как правило, хороши в поиске нарушений безопасности. Например, распространенной ошибкой при реализации буферов, управляемых семафорами, является забывание о том, что код, фактически выполняющий вставку и извлечение, требует взаимного исключения (с использованием `synchronized` или `ReentrantLock`). Демонстрационный прогон теста `PutTakeTest` с версией буфера `BoundedBuffer`, который пренебрегает синхронизацией методов `doInsert` и `doExtract`, довольно быстро заканчивается безуспешно. Прогон теста `PutTakeTest` с несколькими десятками потоков с несколькими миллионами итераций

с буферами различной емкости на различных системах повышает нашу уверенность в отсутствии повреждения данных в методах `put` и `take`.

> Тесты должны прогоняться на многопроцессорных системах для увеличения многообразия потенциальных перемежений. Однако наличие нескольких процессоров необязательно делает тесты эффективнее. Для максимизации вероятности обнаружения чувствительной ко времени гонки данных должно быть больше активных потоков, чем процессоров; благодаря этому в любой момент некоторые потоки работают, а некоторые отключаются, тем самым сокращая предсказуемость взаимодействий между потоками.

В тестах, прогоняемых до тех пор, пока не будет выполнено фиксированное число операций, существует возможность, что тестовый случай никогда не завершится, если в тестируемом коде возникнет исключение из-за дефекта. Наиболее распространенный способ, который помогает справиться с этой ситуацией, — побудить тестовую структуру прерывать тесты, которые не терминируются в течение определенного периода; продолжительность ожидания должна определяться эмпирически, а сбои должны анализироваться с целью обеспечения того, чтобы проблема возникала не из-за того, что вы просто не ждали достаточно долго. (Эта проблема не является уникальной для тестирования конкурентных классов; последовательные тесты также должны проводить различие между длительными и бесконечными циклами.)

12.1.4. Тестирование на управление ресурсами

До настоящего момента тесты занимались проверкой соблюдения классом его спецификации — проверкой того, что он делает именно то, что предположительно должен делать. Вторичный аспект теста заключается в проверке того, что он *не* делает того, что он предположительно *не* должен делать, например утечки ресурсов. Любой объект, который владеет другими объектами или управляет ими, не должен продолжать поддерживать ссылки на эти объекты дольше, чем это необходимо. Такие утечки памяти не позволяют сборщикам мусора освобождать память (либо потоки, файловые дескрипторы, сокеты, подключения к базам данных или другие ограниченные ресурсы) и могут привести к исчерпанию ресурсов и сбою приложения.

Вопросы управления ресурсами особенно важны для таких классов, как `BoundedBuffer` — вся причина ограничения буфера заключается в предотвращении сбоя приложения из-за исчерпания ресурсов, когда производители слишком далеко опережают потребителей. Ограничение побуждает чрезмерно продуктивных производителей блокироваться, вместо того чтобы продолжать создавать работу, которая будет потреблять все больше памяти или других ресурсов.

Нежелательное удержание памяти можно легко протестировать с помощью инструментов инспектирования кучи, которые измеряют потребление памяти приложением; это можно сделать с помощью различных коммерческих и открытых инструментов профилирования кучи. Метод `testLeak` в листинге 12.7 содержит заполнители для инструмента инспектирования кучи для взятия снимка кучи, который побуждает сбор мусора[1], а затем записывает информацию о размере кучи и памяти.

Листинг 12.7. Тест на утечку ресурсов

```
class Big { double[] data = new double[100000]; }

void testLeak() throws InterruptedException {
    BoundedBuffer<Big> bb = new BoundedBuffer<Big>(CAPACITY);
    int heapSize1 = /* куча для снимка */;
    for (int i = 0; i < CAPACITY; i++)
        bb.put(new Big());
    for (int i = 0; i < CAPACITY; i++)
        bb.take();
    int heapSize2 = /* куча для снимка */;
    assertTrue(Math.abs(heapSize1-heapSize2) < THRESHOLD);
}
```

Метод `testLeak` вставляет несколько больших объектов в ограниченный буфер, а затем вынимает их; потребление памяти в снимке #2 должно быть примерно таким же, как в снимке #1. С другой стороны, если метод `doExtract` забыл обнулить ссылку на возвращаемый элемент (`items[i]=null`), то сообщаемое потребление памяти в двух снимках

[1] *Побудить* сбор мусора технически невозможно; `System.gc` лишь *намекает* JVM-машине, что, возможно, наступило благоприятное время для выполнения сбора мусора. С помощью `-XX:+DisableExplicitGC` виртуальную машину HotSpot можно проинструктировать, чтобы она игнорировала вызовы `System.gc`.

определенно не будет одинаковым. (Это один из немногих случаев, когда необходимо явное обнуление; в большинстве случаев это либо не полезно, либо фактически приносит вред [EJ пункт 5].)

12.1.5. Использование обратных вызовов

При конструировании тестовых случаев могут быть полезны обратные вызовы клиентского кода; обратные вызовы часто выполняются в известных точках жизненного цикла объекта, представляющих собой хорошие возможности для подтверждения инвариантов. Например, исполнитель `ThreadPoolExecutor` вызывает объекты `Runnable` задач и фабрику `ThreadFactory`.

Тестирование поточного пула предусматривает тестирование ряда элементов политики выполнения: что дополнительные потоки создаются только тогда, когда они должны, а не иначе; что простаивающие потоки убираются только тогда, когда они должны, и т. д. Создание всестороннего комплекта тестов, охватывающего все возможные случаи, стоит немалых усилий, но многие из них могут быть протестированы довольно просто по отдельности.

Мы можем оборудовать создание потоков настраиваемой фабрики потоков. Фабрика `TestingThreadFactory` в листинге 12.8 поддерживает счетчик созданных потоков; он позволяет тестовым случаям проверять число потоков, созданных во время прогона теста. Фабрика `TestingThreadFactory` может быть расширена и возвращать настраиваемый объект `Thread`, который также регистрирует время, когда поток терминируется, благодаря чему тестовые случаи могут верифицирововать, что потоки убираются в соответствии с политикой выполнения.

Листинг 12.8. Фабрика потоков для тестирования ThreadPoolExecutor

```
class TestingThreadFactory implements ThreadFactory {
    public final AtomicInteger numCreated = new AtomicInteger();
    private final ThreadFactory factory
            = Executors.defaultThreadFactory();

    public Thread newThread(Runnable r) {
        numCreated.incrementAndGet();
        return factory.newThread(r);
    }
}
```

Если размер ядерного пула меньше максимального, то пул потоков должен увеличиваться по мере увеличения спроса на выполнение. Предоставление пулу длительных задач делает число выполняемых задач постоянным достаточно долго, что позволяет сделать несколько подтверждений, таких как проверка того, что пул расширен, как и ожидалось. Это показано в листинге 12.9.

Листинг 12.9. Тестовый метод для проверки расширения пула потоков

```
public void testPoolExpansion() throws InterruptedException {
    int MAX_SIZE = 10;
    ExecutorService exec = Executors.newFixedThreadPool(MAX_SIZE);

    for (int i = 0; i < 10 * MAX_SIZE; i++)
        exec.execute(new Runnable() {
            public void run() {
                try {
                    Thread.sleep(Long.MAX_VALUE);
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                }
            }
        });
    for (int i = 0;
        i < 20 && threadFactory.numCreated.get() < MAX_SIZE;
        i++)
        Thread.sleep(100);
    assertEquals(threadFactory.numCreated.get(), MAX_SIZE);
    exec.shutdownNow();
}
```

12.1.6. Генерирование большего числа расслоений

Поскольку многие потенциальные сбои в конкурентном коде являются маловероятными событиями, тестирование на наличие ошибок конкурентности представляет собой игру чисел, но есть нечто, что вы можете сделать, чтобы улучшить свои шансы. Мы уже упоминали о том, как выполнение на многопроцессорных системах с меньшим числом процессоров, чем число активных потоков, может генерировать больше расслоений (interleadings), чем однопроцессорная система либо многопроцессорная. Схожим образом тестирование на целом ряде систем с разным количеством процессоров,

операционных систем и процессорных архитектур может выявлять проблемы, которые могут возникнуть не во всех системах.

Полезный хитрый прием увеличения числа расслоений и, следовательно, более эффективного разведывания пространства состояний ваших программ заключается в использовании метода `Thread.yield`, который содействует большему числу переключений контекста во время операций доступа к совместному состоянию. (Эффективность этого технического решения зависит от платформы, так как JVM может свободно обрабатывать `Thread.yield` как инструкцию без операций [JLS 17.9]; использование короткого, но ненулевого сна было бы медленнее, но надежнее.) Метод в листинге 12.10 перечисляет средства с одного счета на другой; между двумя операциями обновления инварианты типа «сумма всех счетов равна нулю» не соблюдаются. Иногда выполняя метод `yield` в середине операции, вы можете активировать чувствительные ко времени дефекты в коде, в котором не используется адекватная синхронизация для доступа к состоянию. Неудобство добавления этих вызовов для тестирования и их удаления из кода для производственных целей может быть уменьшено за счет добавления их с помощью инструментов аспектно-ориентированного программирования (AOP).

Листинг 12.10. Использование Thread.yield для генерации большего числа расслоений

```
public synchronized void transferCredits(Account from,
                                         Account to,
                                         int amount) {
    from.setBalance(from.getBalance() - amount);
    if (random.nextInt(1000) > THRESHOLD)
        Thread.yield();
    to.setBalance(to.getBalance() + amount);
}
```

12.2. Тестирование на производительность

Тесты на производительность часто представляют собой расширенные версии тестов на функциональность. На самом деле в тесты на производительность почти всегда стоит включать элементарное тестирование функциональности с целью обеспечения того, чтобы вы не тестировали производительность неисправного кода.

Хотя между тестами на производительность и функциональность определенно существует наложение, у них разные цели. Тесты на производительность стремятся измерить сквозные показатели производительности для репрезентативных случаев использования. Выбор разумного набора сценариев использования не всегда прост; в идеале тесты должны отражать то, как тестируемые объекты используются в приложении фактически.

В некоторых случаях надлежащий сценарий тестирования очевиден. Ограниченные буферы почти всегда используются в паттернах «производитель-потребитель», поэтому целесообразно измерять пропускную способность производителей, передающих данные потребителям. Для этого сценария мы можем легко расширить тест `PutTakeTest`, превратив его в тест на производительность.

Распространенной вторичной целью проведения тестов на производительность является эмпирический выбор размеров для различных лимитов — числа потоков, емкостей буфера и т. д. Хотя эти значения могут оказаться достаточно чувствительными к характеристикам платформы (таким как тип процессора или даже шаговый уровень процессора, число процессоров или размер памяти), требуя динамической конфигурации, одинаково распространено, что разумные варианты этих значений хорошо работают в широком диапазоне систем.

12.2.1. Расширение теста PutTakeTest за счет хронометрирования

Первостепенное расширение, которое нам надо внести в тест `PutTakeTest`, состоит в измерении времени, затрачиваемого на прогон теста. Вместо того чтобы пытаться измерить время отдельной операции, мы получаем более точную меру путем хронометрирования всего прогона и деления на число операций, в результате получив время на операцию. Мы уже используем циклический барьер `CyclicBarrier` для запуска и остановки рабочих потоков, поэтому можем расширить его с помощью барьерного действия, измеряющего время начала и окончания, как показано в листинге 12.11.

Мы можем модифицировать инициализацию барьера за счет применения в нем этого барьерного действия, используя конструктор для `CyclicBarrier`, который принимает барьерное действие:

Листинг 12.11. Таймер на основе барьера

```
this.timer = new BarrierTimer();
this.barrier = new CyclicBarrier(npairs * 2 + 1, timer);

public class BarrierTimer implements Runnable {
    private boolean started;
    private long startTime, endTime;

    public synchronized void run() {
        long t = System.nanoTime();
        if (!started) {
            started = true;
            startTime = t;
        } else
            endTime = t;
    }
    public synchronized void clear() {
        started = false;
    }
    public synchronized long getTime() {
        return endTime - startTime;
    }
}
```

Измененный метод `test` с использованием барьерного таймера показан в листинге 12.12.

Листинг 12.12. Тестирование с помощью барьерного таймера

```
public void test() {
    try {
        timer.clear();
        for (int i = 0; i < nPairs; i++) {
            pool.execute(new Producer());
            pool.execute(new Consumer());
        }
        barrier.await();
        barrier.await();
        long nsPerItem = timer.getTime() / (nPairs * (long)nTrials);
        System.out.print("Throughput: " + nsPerItem + " ns/item");
        assertEquals(putSum.get(), takeSum.get());
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}
```

Из прогона теста `TimedPutTakeTest` мы можем узнать несколько вещей. Одна из них — пропускная способность операции передачи данных между производителем и потребителем для различных комбинаций параметров; другая — как ограниченный буфер масштабируется вместе с разным числом потоков; третья — каким образом можно было бы выбрать размер лимита. Для ответа на эти вопросы требуется выполнять тест для различных параметрических комбинаций, поэтому нам понадобится главный тестовый драйвер, показанный в листинге 12.13.

Листинг 12.13. Драйверная программа для TimedPutTakeTest

```
public static void main(String[] args) throws Exception {
    int tpt = 100000; // испытания на поток
    for (int cap = 1; cap <= 1000; cap *= 10) {
        System.out.println("Capacity: " + cap);
        for (int pairs = 1; pairs <= 128; pairs *= 2) {
            TimedPutTakeTest t =
                new TimedPutTakeTest(cap, pairs, tpt);
            System.out.print("Pairs: " + pairs + "\t");
            t.test();
            System.out.print("\t");
            Thread.sleep(1000);
            t.test();
            System.out.println();
            Thread.sleep(1000);
        }
    }
    pool.shutdown();
}
```

На рис. 12.1 показаны некоторые примеры результатов на 4-ядерной машине с емкостями буфера, равными 1, 10, 100 и 1000. Мы сразу видим, что размер буфера, равный единице, приводит к очень слабой пропускной способности; это вызвано тем, что каждый поток может делать лишь крошечное продвижение перед блокированием и ожиданием другого потока. Увеличение размера буфера до десяти существенно помогает, но числа после десяти демонстрируют уменьшение отдачи.

Сначала, возможно, вызывает недоумение то, что добавление гораздо большего числа потоков ухудшает производительность только слегка. Причину этого трудно понять из данных, но легко увидеть на измерителе производительности процессора, таком как `perfbar`, во время прогона

теста: даже с большим числом потоков происходит не так много вычислений, и бóльшая их часть тратится на блокирование и разблокирование потоков. В связи с этим существует много процессорного люфта, в котором большее число потоков делают то же самое, не особо причиняя вреда производительности.

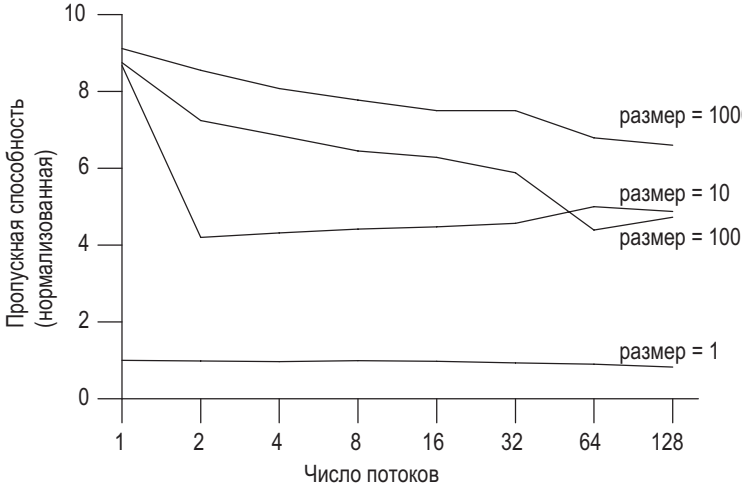

Рис. 12.1. Тест TimedPutTakeTest с различными емкостями буфера

Однако будьте осмотрительны насчет того, чтобы заключить из этих данных, что в программу «производитель-потребитель», использующую ограниченный буфер, вы всегда можете добавить еще больше потоков. Этот тест довольно искусственен в том, как симулирует *приложение*; производители почти не делают никакой работы для генерирования очередного элемента, который помещен в очередь, а потребители почти не делают никакой работы с элементом, который извлечен из нее. Если рабочие потоки в реальном приложении «производитель-потребитель» выполняют некую нетривиальную работу по производству и потреблению элементов (как это обычно бывает), то этот люфт исчезнет и эффекты наличия слишком большого числа потоков могут стать очень заметными. Первостепенная цель этого теста состоит в измерении того, какие ограничения на совокупную пропускную способность накладывает эстафетная передача между производителем и потребителем через ограниченный буфер.

12.2.2. Сравнение многочисленных алгоритмов

В то время как ограниченный буфер `BoundedBuffer` реализован довольно прочно в программном отношении, показывая достаточно хорошую производительность, он не сравнится ни с очередью `ArrayBlockingQueue`, ни с очередью `LinkedBlockingQueue` (что объясняет, почему данный буферный алгоритм не был отобран для включения в библиотеку классов). Алгоритмы `java.util.concurrent` были отобраны и отрегулированы, частично с использованием подобных описанных здесь тестов, чтобы сделать их настолько эффективными и предложить широкий спектр функциональных возможностей[1]. Главная причина, почему `BoundedBuffer` работает неэффективно, заключается в том, что методы `put` и `take` имеют многочисленные операции, которые могут столкнуться с конфликтом — приобрести семафор, приобрести замок, освободить семафор. Другие подходы к реализации имеют меньше точек, в которых они могут конфликтовать с другим потоком.

На рис. 12.2 показана сравнительная пропускная способность на двухмашинном гиперпоточном вычислительном комплексе для всех трех классов с 256-элементными буферами с использованием варианта теста `TimedPutTakeTest`. Этот тест говорит о том, что связная очередь `LinkedBlockingQueue` масштабируется лучше, чем очередь `ArrayBlockingQueue` на основе массива. Этот факт сначала может показаться странным: связная очередь должна выделять объект связного узла для каждой вставки и, следовательно, выполнять больше работы, чем очередь на основе массива. Несмотря на то что она имеет большее число операций выделения ресурсов и больше издержек на сбор мусора, связная очередь допускает более конкурентный доступ посредством операций `put` и `take`, чем очередь на основе массива, поскольку лучшие алгоритмы на основе связных очередей допускают обновление головы и хвоста независимо. Ввиду того что распределение ресурсов обычно является для потоки локальным, алгоритмы, которые могут уменьшить конфликт, делая больше выделений ресурсов, обычно масштабируются лучше. (Это еще один пример, в котором интуиция, основанная на традиционной регулировке производительности, идет вразрез с тем, что необходимо для масштабируемости.)

---

[1] Вы вполне можете превзойти их, если являетесь одновременно экспертом по конкурентности и способны отказаться от некоторой предоставляемой функциональности.

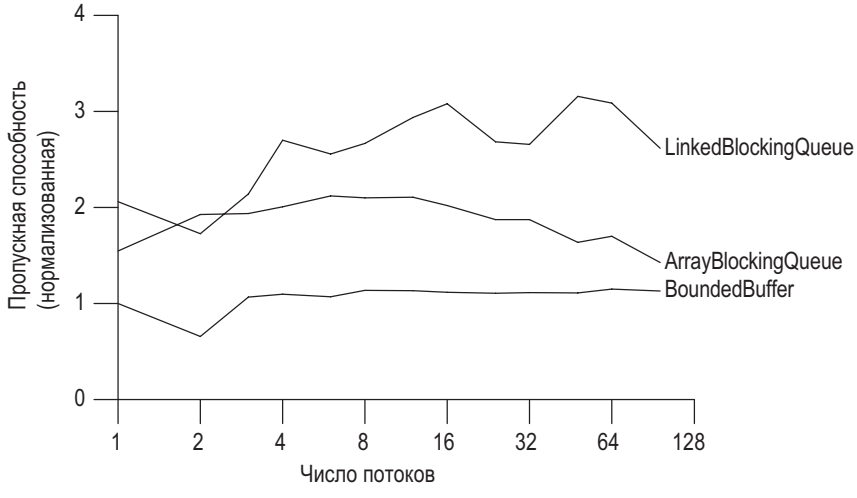

Рис. 12.2. Сравнение реализаций блокирующих очередей

12.2.3. Измерение отзывчивости

До сих пор мы фокусировались на измерении пропускной способности, которая обычно является наиболее важным показателем производительности для конкурентных программ. Но иногда важнее знать, сколько времени может занять завершение отдельного действия, и в этом случае мы хотим измерить *дисперсию* времени обслуживания. Иногда имеет смысл допустить более продолжительное среднее время обслуживания, в случае если это позволяет нам получить меньшую дисперсию; ценной характеристикой производительности также является предсказуемость. Измерение дисперсии позволяет оценивать ответы на вопросы относительно качества обслуживания типа «какой процент операций будет успешным в пределах 100 миллисекунд?».

Гистограммы времени завершения задачи обычно являются лучшим способом визуализации дисперсии времени обслуживания. Дисперсии лишь немного сложнее измерить, чем средние значения, — в дополнение к совокупному времени завершения вам нужно отслеживать времена в расчете на завершение задачи. Поскольку гранулярность таймера может быть фактором в оценивании времени отдельной задачи (на отдельную задачу может уходить времени меньше или близко к наименьшему «таймерному тику», что исказило бы измерения длительности задачи), в целях предотвращения артефактов измерения вместо этого мы можем измерить время выполнения небольших пакетов операций `put` и `take`.

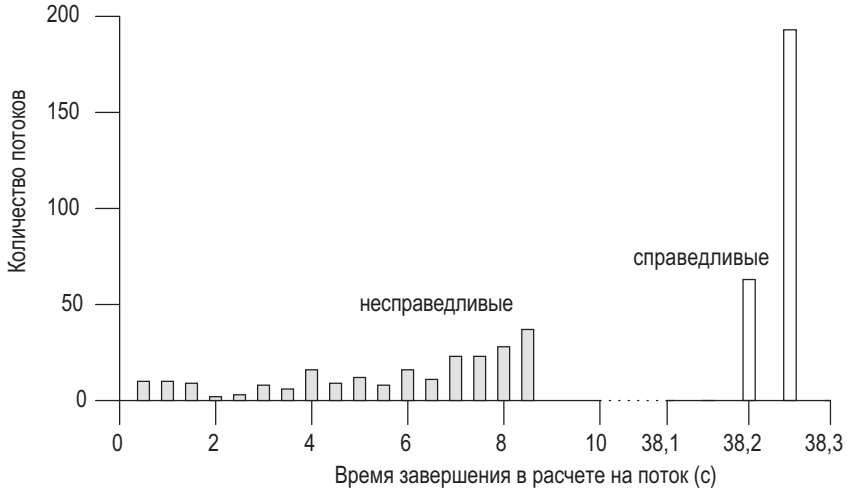

Рис. 12.3. Гистограмма времени завершения для теста TimedPutTakeTest со стандартными (несправедливыми) и справедливыми семафорами

На рис. 12.3 показаны варианты завершения теста `TimedPutTakeTest` в расчете на задачу с использованием буфера размером 1000, в котором каждая из 256 конкурентных задач выполняет итеративный обход только 1000 элементов для несправедливого (затененные столбики) и справедливого (пустые столбики) семафоров. (В разделе 13.3 дается объяснение справедливой очередности в сопоставлении с несправедливой для замков и семафоров.) Времена завершения для несправедливых находятся в диапазоне от 104 до 8,714 мс, с коэффициентом свыше восьмидесяти. Существует возможность уменьшить этот диапазон, побуждая проявлять больше справедливости в управлении конкурентностью; это легко сделать в `BoundedBuffer`, инициализируя семафоры в справедливом режиме. Как показано на рис. 12.3, за счет этого дисперсия значительно сокращается (теперь она составляет от 38,194 до 38,207 мс), но, к сожалению, также значительно снижается пропускная способность. (Более длительный тест с более типичными типами задач, вероятно, покажет еще большее снижение пропускной способности.)

Ранее мы видели, что очень малые размеры буфера вызывают очень частое переключение контекста и слабую пропускную способность даже в несправедливом режиме, потому что почти каждая операция сопряжена с переключением контекста. Как показатель того, что стоимость справедливости вытекает прежде всего из блокирующих потоков, мы можем про-

гнать этот тест заново с размером буфера, равным единице, и увидеть, что несправедливые семафоры теперь работают сравнимо со справедливыми. На рис. 12.4 показано, что в этом случае справедливость не делает среднее значение намного хуже или дисперсию намного лучше.

Таким образом, если только потоки не блокируют продвижение непрерывно из-за жестких требований к синхронизации, несправедливые семафоры обеспечивают гораздо более высокую пропускную способность, а справедливые семафоры обеспечивают меньшую дисперсию. Поскольку эти результаты сильно различаются, `Semaphore` вынуждает своих клиентов решать, какой из этих двух факторов следует оптимизировать.

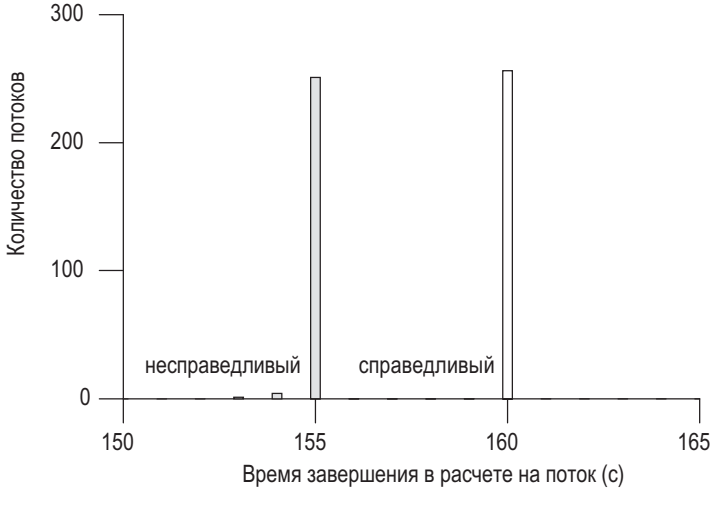

Рис. 12.4. Гистограмма времени завершения для теста TimedPutTakeTest с одноэлементными буферами

12.3. Предотвращение ошибок при тестировании

Теоретически разработка тестов на производительность проста — найти типичный сценарий использования, написать программу, которая выполняет этот сценарий много раз, и засечь время. На практике следует остерегаться ряда ловушек кодирования, которые не дают тестам на производительность получать содержательные результаты.

12.3.1. Сбор мусора

Хронометрирование сбора мусора непредсказуемо, поэтому всегда существует вероятность того, что сборщик мусора будет работать во время измеряемого тестового прогона. Если тестовая программа делает *N* итераций и не запускает сбор мусора, но итерация *N* + 1 вызовет сбор мусора, то небольшая вариация размера прогона может оказать большой (но мнимый) эффект на измеряемое время в расчете на итерацию.

Существует две стратегии, которые предотвращают смещение результатов сбора мусора. Одна из них — обеспечивать, чтобы сбор мусора вообще не работал во время теста (вы можете это выяснить, инициализировав JVM с параметром `-verbose:gc`); как вариант, вы можете обеспечить, чтобы во время прогона сборщик мусора работал несколько раз, благодаря чему тестовая программа сможет адекватно отражать стоимость регулярного выделения ресурсов и сбор мусора. Последняя стратегия часто бывает лучше — она требует более длительного тестирования и с большей вероятностью отражает реальную производительность.

Большинство приложений «производитель-потребитель» сопряжено с изрядным числом выделения ресурсов и сбора мусора — производители выделяют новые объекты, которые используются и отбрасываются потребителями. Прогон теста с ограниченным буфером, достаточно продолжительный для того, чтобы инициировать многократные операции сбора мусора, дает более точные результаты.

12.3.2. Динамическая компиляция

Написание и интерпретация эталонных тестов на производительность для динамически компилируемых языков, таких как Java, намного сложнее, чем для статически компилируемых языков, таких как C или C++. JVM-машина HotSpot (и другие современные JVM) использует комбинацию байт-кодовой интерпретации и динамической компиляции. При первой загрузке класса JVM выполняет его, интерпретируя байт-код. В какой-то момент, если метод выполняется достаточно часто, срабатывает динамический компилятор и конвертирует этот код в машинный; по завершении компиляции он переключается с интерпретации на прямое выполнение.

Хронометрирование компиляции ведет себя непредсказуемо. Хронометрирующие тесты должны прогоняться только после компиляции всего

кода; нет смысла измерять скорость интерпретируемого кода, так как большинство программ работают достаточно долго, в результате чего все часто выполняемые ветви кода компилируются. Разрешение компилятору работать во время измеряемого тестового прогона может сместить результаты теста двумя путями: компиляция потребляет ресурсы процессора, и оценивание времени работы комбинации интерпретируемого и скомпилированного кода не является содержательным метрическим показателем производительности. На рис. 12.5 показано, как это может повлиять на результаты. Три временных шкалы представляют выполнение одного и того же числа итераций: временная шкала *A* представляет все интерпретируемое выполнение, *B* представляет компиляцию в середине выполнения и *C* представляет компиляцию в начале выполнения. Точка, в которой работает компиляция, серьезно влияет на время работы, измеряемое в расчете на одну операцию[1].

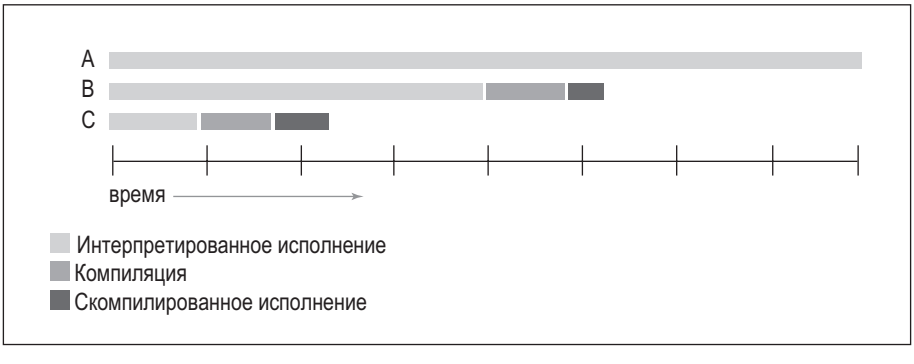

Рис. 12.5. Результаты, смещенные динамической компиляцией

Код также может быть декомпилирован (возвращен к интерпретируемому выполнению) и перекомпилирован по различным причинам, таким как загрузка класса, аннулирующие предположения, сделанные предыдущими компиляциями, или сбор данных профилирования, достаточных для того, чтобы решить, что ветвь кода должна быть перекомпилирована с другими оптимизациями.

[1] JVM может решить выполнить компиляцию в потоке приложения или в фоновом потоке; каждая может смещать результаты хронометрирования поразному.

Одним из способов, которые предотвращают смещение результатов компиляции, состоит в длительной работе вашей программы (по крайней мере несколько минут), с тем чтобы компиляция и интерпретируемое выполнение представляли собой небольшую часть общего времени работы. Еще один подход заключается в использовании неизмеряемого «разминающего» прогона, при котором ваш код выполняется достаточно долго, чтобы быть полностью скомпилированным во время фактического запуска хронометрирования. В HotSpot-машине запуск вашей программы с опцией `-XX:+PrintCompilation` выводит сообщение в момент, когда происходит динамическая компиляция; это позволяет вам удостовериться, что она происходит до, а не во время измеряемых тестовых прогонов.

Прогон одного и того же теста несколько раз в одном экземпляре JVM может использоваться для проверки методологии тестирования. Первую группу результатов следует отбросить как разминку; наличие противоречивых результатов в остальных группах свидетельствует о том, что тест следует изучать дальше, с целью установления того, почему результаты хронометрирования невоспроизводимы.

JVM использует различные фоновые потоки для служебных задач. Во время измерения многочисленных *неродственных* вычислительно интенсивных действий рекомендуется размещать явные паузы между измеряемыми испытаниями в однократном прогоне, чтобы дать JVM возможность нивелировать отставание от фоновых задач с минимальным вмешательством со стороны измеряемых задач. (Однако при измерении многочисленных родственных действий, таких как многократные прогоны одного теста, такое исключение фоновых задач JVM-машины может дать нереально оптимистичные результаты.)

12.3.3. Нереалистичный отбор ветвей кода

Компиляторы рабочей среды используют информацию профилирования для оптимизации компилируемого кода. JVM разрешено использовать специфичную для выполнения информацию, для того чтобы порождать более качественный код, а это значит, что компиляция метода *M* в одной программе может генерировать отличающийся код, чем компиляция M в другой. В некоторых случаях JVM может делать оптимизацию, опираясь на допущения, которые могут быть истинными только временно, а затем

возвращать их обратно, лишая достоверности скомпилированный код, в случае если они становятся ложными[1].

В результате очень важно, чтобы тестовые программы адекватно приближались не только к шаблонам использования типичного приложения, но и к набору ветвей кода, используемых таким приложением. В противном случае динамический компилятор может провести специальные оптимизации чисто однопоточной тестовой программы, которая не может быть применена в реальных приложениях, содержащих хотя бы периодический параллелизм. Поэтому тесты на многопоточную производительность обычно следует смешивать с тестами на однопоточную производительность, даже если требуется измерить только однопоточную производительность. (Этот вопрос не возникает в тесте `TimedPutTakeTest`, так как даже самый малый тестовый случай использует два потока.)

12.3.4. Нереалистичные уровни конфликта

Конкурентные приложения, как правило, чередуют два очень разных вида работы: доступ к общим данным, например доставка следующей задачи из совместной рабочей очереди, и локальное для потока вычисление (выполнение задачи, если исходить из того, что сама задача не обращается к совместным данным). В зависимости от относительных пропорций двух типов работы приложение будет испытывать разные уровни конфликта и демонстрировать разные формы поведения производительности и масштабирования.

Если *N* потоков доставляют задачи из совместной рабочей очереди и выполняют их и задачи являются вычислительно-емкими и длительными (и не обращаются к совместным данным часто), то конфликта почти не будет; над пропускной способностью доминирует наличие процессорных ресурсов. С другой стороны, если задачи являются очень скоротечными, то будет много конфликтов за рабочую очередь, и тогда над пропускной способностью доминирует стоимость синхронизации.

---

[1] Например, JVM может использовать *преобразование «мономорфный вызов»*, которое конвертирует вызов виртуального метода в прямой вызов метода, в случае если ни один из загруженных классов не переопределяет этот метод, но это преобразование лишает скомпилированный код достоверности, если впоследствии загружается класс, который этот метод переопределяет.

Для того чтобы получить реалистичные результаты, тесты на конкурентную производительность должны стремиться к локальным для потока вычислениям, выполняемым типичным приложением, в дополнение к изучаемой конкурентной координации. Если работа, выполняемая для каждой задачи в приложении, существенно отличается по характеру или объему от тестовой программы, то легко прийти к необоснованным заключениям о том, где лежат узкие места производительности. В разделе 11.5 мы видели, что для классов, основанных на замковой защите, таких как синхронизированные реализации ассоциативного массива `Map`, ответ на вопрос, является ли доступ к замку в основном оспариваемым или в основном не оспариваемым, может иметь драматическое влияние на пропускную способность. Тесты в том разделе не делают ничего, кроме того, что барабанят по массиву; даже с двумя потоками все попытки обратиться к массиву оспариваются. Однако если бы приложение делало значительный объем локального для потока вычисления для каждого случая, когда оно обращается к совместной структуре данных, то уровень конфликта мог бы быть низким настолько, что обеспечивал бы хорошую производительность.

В этом отношении для некоторых приложений тест `TimedPutTakeTest` может быть слабой моделью. Поскольку рабочие потоки делают не слишком много, над пропускной способностью доминируют издержки на координацию, и это не обязательно относится ко всем приложениям, которые обмениваются данными между производителями и потребителями через ограниченные буферы.

12.3.5. Устранение мертвого кода

Одна из проблем написания хороших тестов (на любом языке) заключается в том, что оптимизирующие компиляторы умеют выявлять и устранять мертвый код — код, который не влияет на результат. Поскольку эталонные тесты часто ничего не вычисляют, они являются легкой мишенью для оптимизатора. В большинстве случаев совсем неплохо, когда оптимизатор вырезает мертвый код из программы, но для эталонного теста это большая проблема, потому что в этом случае вы измеряете меньше выполнения программы, чем вы думаете. Если вам повезет, то оптимизатор очистит *всю* вашу программу, и тогда будет очевидно, что ваши данные являются фиктивными. Если вам не повезет, устранение мертвого кода просто

ускорит программу на какой-то коэффициент, который *можно* объяснить другими способами.

Устранение мертвого кода также является проблемой в эталонном тестировании статически компилируемых языков, но обнаружить, что компилятор устранил хороший кусок вашего эталонного теста, намного проще, потому что вы можете взглянуть на машинный код и увидеть, что часть вашей программы отсутствует. С динамически компилируемыми языками эта информация недоступна.

Многие эталонные микротесты работают гораздо «лучше», когда они запускают компилятор HotSpot с опцией `-server`, чем с `-client`, не только потому, что серверный компилятор может производить более эффективный код, но и потому, что он более умело оптимизирует мертвый код. К сожалению, процедура устранения мертвого кода, которая быстро справилась с вашим эталонным текстом, не будет работать так же хорошо с кодом, который на самом деле что-то делает. Но на многопроцессорных системах вы все равно должны предпочесть опцию `-server` опции `-client` как для производства, так и для тестирования — вам просто нужно писать свои тесты так, чтобы они не были подвержены устранению мертвого кода.

> Написание эффективных тестов на производительность требует обмана оптимизатора, чтобы тот не оптимизировал ваш эталонный тест как мертвый код. Это обусловливает необходимость, чтобы каждый вычисленный результат каким-то образом использовался вашей программой — каким-то способом, который не требует синхронизации или существенных вычислений.

В тесте `PutTakeTest` мы вычисляем контрольную сумму элементов, добавляемых и удаляемых из очереди, и объединяем эти контрольные суммы по всем потокам, но данная работа тоже может быть оптимизирована, если мы на самом деле не *используем* значение контрольной суммы. Это значение может потребоваться для верификации правильности алгоритма. Правда, использование любого значения можно обеспечить путем его вывода. Однако вам следует избегать ввода-вывода во время работы теста, чтобы не исказить измерение времени работы.

Дешевый трюк, который позволяет избежать оптимизации вычисления без слишком больших издержек, состоит в вычислении метода `hashCode` поля некоторого производного объекта, сравнении его с произвольным

значением, таким как текущее значение `System.nanoTime`, и распечатки бесполезного и игнорируемого сообщения, если они совпадают:

```
if (foo.x.hashCode() == System.nanoTime())
    System.out.print(" ");
```

Это сравнение редко будет успешным, и если это произойдет, то его единственным эффектом будет вставка безвредного символа пробела в вывод. (Метод `print` буферизует вывод, до тех пор пока не будет вызван метод `println`, поэтому в редком случае, когда `hashCode` и `System.nanoTime` равны, никакого ввода/вывода на самом деле не происходит.)

Каждый вычисленный результат не только должен использоваться, но и результаты должны быть неочевидными. В противном случае интеллектуальный динамический оптимизирующий компилятор может заменить действия предварительно вычисленными результатами. Мы обращались к этому вопросу при построении теста `PutTakeTest`, но любая тестовая программа, на чей вход поступают статические данные, уязвима для этой оптимизации.

12.4. Комплементарные подходы к тестированию

Хотя мы хотели бы верить, что эффективная тестирующая программа должна «найти все ошибки», эта цель является нереалистичной. NASA выделяет больше своих инженерных ресурсов на тестирование (по оценкам, они используют 20 тестировщиков для каждого разработчика), чем любое коммерческое предприятие может себе позволить, — и порожденный код по-прежнему не свободен от дефектов. В сложных программах никакое количество тестирования не способно отыскать все ошибки кодирования.

Целью тестирования является не столько *поиск ошибок*, сколько *повышение уверенности* в том, что код работает, как и ожидается. Поскольку нереалистично допускать, что вы можете найти все ошибки, цель плана обеспечения качества (QA) должна заключаться в достижении максимально возможной уверенности с учетом имеющихся ресурсов тестирования. В конкурентной программе может произойти больше ошибок, чем в последовательной, и поэтому для достижения того же уровня уверенности требуется больше тестирования. До сих пор мы сосредоточивались в основном на технических решениях для конструирования эффективных

модульных тестов и тестов на производительность. Тестирование имеет критическую важность для укрепления уверенности в том, что конкурентные классы ведут себя благополучно, но оно должно быть лишь одной из задействуемых вами методологий QA.

Разные методологии QA являются более эффективными в отыскании одних типов дефектов и менее эффективными в отыскании других. Используя комплементарные методологии тестирования, такие как ревизия кода и статический анализ, вы можете добиться большей уверенности, чем при использовании какого-либо отдельного подхода.

12.4.1. Ревизия кода

Хотя модульные и стресс-тесты эффективны и важны для отыскания дефектов конкурентности, они не заменят тщательной ревизии кода несколькими людьми. (С другой стороны, ревизия кода также не может заменить тестирование.) Вы можете и должны проектировать тесты, которые максимизируют их шансы на обнаружение ошибок безопасности, и вы должны прогонять их часто, но вы не должны пренебрегать тем, чтобы конкурентный код подвергался тщательной ревизии кем-то, кроме его автора. Оплошности допускают даже конкурентные эксперты; почти всегда стоит тратить время на то, чтобы кто-то другой занимался ревизией кода. Опытные конкурентные программисты лучше справляются с отысканием едва различимых гоночных условий, чем большинство тестовых программ. (Кроме того, платформенные вопросы, такие как детали реализации JVM или модели процессорной памяти, могут предотвратить появление дефектов в определенных конфигурациях оборудования или программного обеспечения.) Ревизия кода также имеет и другие преимущества; она не только может находить ошибки, но и часто улучшает качество комментариев, описывающих детали реализации, тем самым уменьшая будущую стоимость сопровождения и риск.

12.4.2. Инструменты статического анализа

На момент написания этих строк стали быстро появляться *инструменты статического анализа* в качестве эффективного дополнения к формальному тестированию и ревизии кода. Статический анализ кода представляет собой подробный разбор кода без его выполнения, и инструменты аудирования кода могут анализировать классы в поисках экземпляров распространенных *шаблонов дефектов*. Инструменты статического анализа,

такие как FindBugs[1] с открытым исходным кодом, содержат *детекторы* многих распространенных *ошибок* кодирования, многие из которых могут быть легко пропущены при тестировании или ревизии кода.

Инструменты статического анализа создают список предупреждений, которые необходимо проверить вручную, для того чтобы выяснить, что эти предупреждения представляют фактические ошибки. Исторически такие инструменты, как `lint`, производили так много ложных предупреждений, что распугали всех разработчиков, но такие инструменты, как FindBugs, были отрегулированы так, чтобы производить как можно меньше ложных сигналов. Инструменты статического анализа все еще несколько примитивны (в особенности в их интеграции с инструментами разработки и жизненным циклом), но уже достаточно эффективны, чтобы стать ценным дополнением к процессу тестирования.

На момент написания этих строк FindBugs включает детекторы следующих шаблонов ошибок, связанных с конкурентностью, и с каждым разом их добавляется все больше.

Противоречивая синхронизация. Многие объекты следуют политике синхронизации, состоящей в защите всех переменных с помощью внутреннего замка объекта. Если к полю делаются частые обращения, но не всегда с удержанием замка `this`, то это может означать, что политика синхронизации не соблюдается постоянно.

Средства анализа должны угадывать политику синхронизации, поскольку классы Java не имеют формальных спецификаций конкурентности. В будущем, если такие аннотации, как `@GuardedBy`, будут стандартизованы, то инструменты аудирования смогут интерпретировать аннотации, вместо того чтобы гадать о взаимосвязи между переменными и замками, что в итоге повысит качество анализа.

Активирование `Thread.run`. Класс `Thread` реализует интерфейс `Runnable` и поэтому имеет метод `run`. Однако вызов метода `Thread.run` напрямую почти всегда является промашкой; обычно программист имеет в виду вызов метода `Thread.start`.

Неосвобожденный замок. В отличие от внутренних замков явные замки (см. главу 13) не освобождаются автоматически, когда управление выходит из области действия, в которой они были приобретены. Стан-

[1] См. http://findbugs.sourceforge.net

дартная идиома — освободить замок из блока `finally`; в противном случае замок может остаться не освобожденным в случае исключения `Exception`.

Пустой синхронизированный блок. Хотя пустые синхронизированные блоки действительно имеют семантику в модели памяти Java, они часто используются неправильно, и обычно существуют более качественные решения любой задачи, которую разработчик пытался решить.

Блокировка с двойной проверкой. Блокировка с двойной проверкой — это неисправная идиома для сокращения издержек синхронизации при ленивой инициализации (см. раздел 16.2.4), которая предусматривает чтение совместного мутируемого поля без соответствующей синхронизации.

Запуск потока из конструктора. Запуск потока из конструктора вносит риск возникновения проблем подклассирования и может дать ссылке `this` ускользнуть из конструктора.

Ошибки уведомлений. Методы `notify` и `notifyAll` указывают на то, что состояние объекта могло измениться, в результате чего потоки, ожидающие в ассоциированной очереди условий, были бы разблокированы. Эти методы должны вызываться, только когда состояние, ассоциированное с очередью условий, изменилось. Ситуация, когда снхронизованный блок, который вызывает `notify` или `notifyAll`, но не модифицирует никакого состояния, скорее всего, будет ошибкой. (См. главу 14.)

Ошибки ожидания по условию. Во время ожидания в очереди условий метод `Object.wait` или `Condition.await` должен вызываться в цикле с удержанием соответствующего замка после проверки некоторого предиката состояния (см. главу 14). Вызов метода `Object.wait` или `Condition.await` без удержания замка не в цикле либо без проверки некоторого предиката состояния почти наверняка является ошибкой.

Неправильное использование `Lock` и `Condition`. Использование замкового объекта `Lock` в качестве замкового аргумента для синхронизированного блока может быть опечаткой, как и вызов метода `Condition.wait` вместо `await` (хотя последняя опечатка, вероятно, будет отловлена при проверке, так как этот метод выдал бы исключение `IllegalMonitorStateException` при первом его вызове).

Сон или ожидание с удержанием замка. Вызов метода `Thread.sleep` с удержанием замка может не дать другим потокам продвигаться вперед

в течение длительного времени и, следовательно, является потенциально серьезной угрозой для жизнеспособности. Вызов метода `Object.wait` или `Condition.await` с удержанием двух замков представляет аналогичную угрозу.

Холостые циклы. Код, который ничего не делает, кроме как непрерывно проверяет поле на ожидаемое значение (busy wait, spin-wait, занятое ожидание), может тратить процессорное время впустую, и если поле не является мутируемым, то не гарантирует завершение. Защелки или ожидания по условию часто являются лучшим техническим решением во время ожидания перехода из состояния в состояние.

12.4.3. Аспектно-ориентированные методы тестирования

На момент написания этих строк технические решения аспектно-ориентированного программирования (AOP) имели только ограниченную применимость к конкурентности, поскольку большинство популярных инструментов AOP еще не поддерживают точечные врезки в точках синхронизации. Однако AOP может применяться для подтверждения инвариантов или некоторых аспектов соответствия политикам синхронизации. Например, в работе (Laddad, 2003) представлен пример использования аспекта для обертывания всех вызовов непотокобезопасных методов Swing, с подтверждением, что вызов происходит в событийном потоке. Поскольку это техническое решение не требует никаких изменений кода, оно является простым в применении и может раскрывать едва различимые ошибки публикации и ограничения одного потока.

12.4.4. Средства профилирования и мониторинга

Большинство коммерческих средств профилирования поддерживают потоки. Они различаются по набору функционала и эффективности, но часто могут дать представление о том, что делает ваша программа (хотя инструменты профилирования обычно навязчивы и могут существенно влиять на синхронизацию и поведение программы). Большинство из них предлагают вывод на экран временной шкалы каждого потока с разными цветами для разных состояний (работоспособен, блокирован в ожидании замка, блокирован в ожидании ввода-вывода и т. д.). Такой вывод может показывать, насколько эффективно программа использует имеющиеся

процессорные ресурсы, и если она работает плохо, то где искать причину. (Многие профилировщики также заявляют о наличии функционала по идентификации того, какие замки вызывают конфликт, но на практике этот функционал часто является более грубым инструментом, чем требуется для анализа замкового поведения программы.)

Встроенный агент JMX также предлагает некоторый ограниченный функционал для мониторинга поточного поведения. Класс `ThreadInfo` включает текущее состояние потока, и если поток блокирован, то замок либо очередь условий, в которой он блокирован. Если включена функция «мониторинг поточного конфликта» (thread contention monitoring) (по умолчанию она отключена из-за ее влияния на производительность), то `ThreadInfo` также включает количество раз, когда этот поток блокировал в ожидании замка или уведомления, и суммарное время, которое он провел в ожидании.

Итоги

Задача тестирования конкурентных программ на правильность может быть чрезвычайно сложной, поскольку многие из возможных режимов сбоя конкурентных программ являются низковероятностными событиями, чувствительными ко временной координации, нагрузке и другим трудновоспроизводимым условиям. Кроме того, тестовая структура может вносить дополнительные ограничения на временну́ю координацию или синхронизацию, которые могут маскировать проблемы конкурентности в тестируемом коде. Задача тестирования конкурентных программ на производительность может быть не менее сложной; программы Java сложнее тестировать, чем программы, написанные на статически компилируемых языках, таких как C, потому что измерения времени могут зависеть от динамической компиляции, сбора мусора и адаптивной оптимизации.

Для того чтобы иметь наилучшие шансы отыскать скрытые дефекты до их появления в производстве, объединяйте традиционные технические решения по тестированию (стараясь избегать описанных здесь ловушек) с ревизиями кода и средствами автоматического анализа. Каждый из этих методов находит проблемы, которые другие, скорее всего, пропустят.

Часть IV
Продвинутые темы

13 Явные замки

Раньше единственными механизмами координации доступа к совместным данным были `synchronized` и `volatile`. Теперь в Java есть еще один вариант: класс повторно входимого замка `ReentrantLock`. Повторно входимый замок `ReentrantLock` не является заменой для внутренней замковой защиты, а скорее представляет собой альтернативу с расширенным функционалом, когда внутренняя замковая защита оказывается слишком ограниченной.

13.1. Lock и ReentrantLock

Интерфейс `Lock`, показанный в листинге 13.1, определяет ряд абстрактных замковых операций. В отличие от внутренней замковой защиты, `Lock` предлагает выбор между безусловным, опрашиваемым, хронометрированным и прерываемым замковым приобретением, и все операции `lock` и `unlock` являются явными. Реализации интерфейса `Lock` должны обеспечивать ту же самую семантику видимости памяти, что и у внутренних замков, но могут различаться по их замковой семантике, алгоритмам планирования, гарантиям упорядоченности и характеристикам производительности. (`Lock.newCondition` рассматривается в главе 14.)

Повторно входимый замок `ReentrantLock` реализует интерфейс `Lock`, обеспечивая те же гарантии взаимного исключения и видимости памяти, что и у `synchronized`. Приобретение замка `ReentrantLock` имеет ту же

семантику памяти, как и вход в синхронизированный блок `synchronized`, и освобождение замка `ReentrantLock` имеет ту же семантику памяти, как и выход из синхронизированного блока `synchronized`. (Видимость памяти рассматривается в разделе 3.1 и в главе 16.) И, как и `synchronized`, замок `ReentrantLock` предлагает семантику повторно входимой замковой защиты (см. раздел 2.3.2). `ReentrantLock` поддерживает все режимы замкового приобретения, определенные интерфейсом `Lock`, обеспечивая бóльшую гибкость для работы с отсутствием замков, чем у `synchronized`.

Листинг 13.1. Интерфейс Lock

```
public interface Lock {
    void lock();
    void lockInterruptibly() throws InterruptedException;
    boolean tryLock();
    boolean tryLock(long timeout, TimeUnit unit)
        throws InterruptedException;
    void unlock();
    Condition newCondition();
}
```

Зачем создавать новый замковый механизм, который так похож на внутреннюю замковую защиту? Внутренняя замковая защита отлично работает в большинстве ситуаций, но имеет некоторые функциональные ограничения — невозможно прервать поток, ожидающий приобретения замка, или попытаться приобрести замок, не желая ждать его вечно. Внутренние замки также должны освобождаться в том же блоке кода, в котором они приобретены; это упрощает кодирование и прекрасно взаимодействует с обработкой исключений, но делает невозможной дисциплину неблоковоструктурированной замковой защиты. Ничего из этого не является причиной отказаться от `synchronized`, но в некоторых случаях более гибкий замковый механизм предлагает более высокую жизнеспособность или производительность.

В листинге 13.2 показана каноническая форма использования замкового объекта `Lock`. Эта идиома несколько сложнее, чем использование внутренних замков: замок *должен* быть освобожден в блоке `finally`. В противном случае замок никогда не будет освобожден, если защищенный код состоит в том, чтобы выдавать исключение. При использовании замковой защиты вы также должны учитывать, что происходит, если исключение выдается из блока `try`; если объект может быть оставлен в противоречивом состоянии, то могут потребоваться дополнительные блоки `try-catch` или `try-`

finally. (Вам следует всегда рассматривать эффект исключений во время использования любой формы замковой защиты, включая внутреннюю замковую защиту.)

Отказ от использования finally для освобождения замка является бомбой замедленного действия. Когда она сработает, вам будет трудно отследить ее происхождение, поскольку не будет зарегистрировано, где и когда замок должен был быть освобожден. Это одна из причин не использовать замок ReentrantLock в качестве замены synchronized: он — «опаснее», потому что не очищает замок автоматически, когда контроль покидает защищаемый блок. Хотя помнить об освобождении замка из блока finally вовсе не так сложно, забыть об этом не есть нечто невозможное[1].

Листинг 13.2. Защита состояния объекта с помощью ReentrantLock

```
Lock lock = new ReentrantLock();
...
lock.lock();
try {
    // обновить состояние объекта
    // отловить исключения и восстановить инварианты при
    // необходимости
} finally {
    lock.unlock();
}
```

13.1.1. Опрашиваемое и хронометрируемое приобретение замка

Режимы хронометрированного и опрашиваемого приобретения замка, предоставляемые методом tryLock, обеспечивают более изощренное обнаружение и исправление ошибок, чем безусловное приобретение. В случае внутренних замков взаимная блокировка является фатальной. Единственный способ восстановления состоит в перезапуске приложения, а единственная защита состоит в конструировании вашей программы таким образом, чтобы противоречивый порядок блокировки был невозможен. Хронометрированная и опрашиваемая замковая защита предлагает еще один вариант: вероятностное предотвращение взаимной блокировки.

[1] FindBugs имеет детектор «неосвобожденного замка», идентифицирующий, когда замковый объект Lock не освобождается во всех ветвях кода из блока, в котором он был приобретен.

Использование хронометрированного и опрашиваемого приобретения замка (tryLock) позволяет восстановить контроль, если вы не можете приобрести все необходимые замки, освободить те, которые вы все-таки приобрели, и повторить попытку (или, по крайней мере, зарегистрировать сбой в журнале и сделать что-то еще). В листинге 13.3 показан альтернативный способ устранения взаимной блокировки из-за динамического упорядочивания из раздела 10.1.2: использовать tryLock, для того чтобы попытаться приобрести оба замка, но отступить и повторить попытку, если оба замка не могут быть приобретены. Время сна имеет фиксированный и случайный компоненты, которые сокращают вероятность возникновения активной блокировки (livelock). Если замок не удается приобрести в течение указанного времени, то метод transferMoney возвращает статус сбоя, благодаря чему операция отказывает плавно. (См. [CPJ 2.5.1.2] и [CPJ 2.5.1.3], где приводятся дополнительные примеры использования опрашиваемых замков для предотвращения взаимной блокировки.)

Листинг 13.3. Предотвращение блокировки из-за порядка замков с помощью метода tryLock

```
public boolean transferMoney(Account fromAcct,
                             Account toAcct,
                             DollarAmount amount,
                             long timeout,
                             TimeUnit unit)
        throws InsufficientFundsException, InterruptedException {
    long fixedDelay = getFixedDelayComponentNanos(timeout, unit);
    long randMod = getRandomDelayModulusNanos(timeout, unit);
    long stopTime = System.nanoTime() + unit.toNanos(timeout);

    while (true) {
        if (fromAcct.lock.tryLock()) {
            try {
                if (toAcct.lock.tryLock()) {
                    try {
                        if (fromAcct.getBalance().compareTo(amount)
                                < 0)
                            throw new InsufficientFundsException();
                        else {
                            fromAcct.debit(amount);
                            toAcct.credit(amount);
                            return true;
                        }
```

продолжение ⇩

Листинг 13.3 *(продолжение)*

```
                    } finally {
                        toAcct.lock.unlock();
                    }
                }
            } finally {
                fromAcct.lock.unlock();
            }
        }
        if (System.nanoTime() < stopTime)
            return false;
        NANOSECONDS.sleep(fixedDelay + rnd.nextLong() % randMod);
    }
}
```

Хронометрированные замки также полезны при реализации действий, которые управляют бюджетом времени (см. раздел 6.3.7). Когда действие с бюджетом времени вызывает блокирующий метод, оно может предоставить тайм-аут, соответствующий оставшемуся в бюджете времени. Это дает действиям завершаться досрочно, если они не могут доставить результат в течение требуемого времени. В случае внутренних замков отменить приобретение замка после его запуска невозможно, поэтому внутренние замки ставят под угрозу возможность реализации бюджетированных по времени действий.

Пример с поездками в листинге 6.17 создает отдельную задачу для каждой компании по прокату автомобилей, от которой он запрашивал коммерческие предложения. Запрос предложения, вероятно, предусматривает какой-то механизм сетевого запроса, такой как запрос веб-службы. Однако для получения предложения может также потребоваться исключающий доступ к дефицитному ресурсу, например прямой линии связи с компанией.

Мы видели один из способов обеспечения последовательного доступа к ресурсу в разделе 9.5: однопоточный исполнитель. Еще один подход заключается в использовании исключающей блокировки для защиты доступа к ресурсу. Код, приведенный в листинге 13.4, пытается отправить сообщение по совместной линии связи, защищенной замком, но терпит неудачу, если не может этого сделать в рамках своего бюджета времени. Хронометрированный метод `tryLock` делает практичным встраивание защиты от исключающей блокировки в такое ограниченное по времени действие.

Листинг 13.4. Блокировка с бюджетом времени

```
public boolean trySendOnSharedLine(String message,
                                   long timeout, TimeUnit unit)
                                   throws InterruptedException {
    long nanosToLock = unit.toNanos(timeout)
                     - estimatedNanosToSend(message);
    if (!lock.tryLock(nanosToLock, NANOSECONDS))
        return false;
    try {
        return sendOnSharedLine(message);
    } finally {
        lock.unlock();
    }
}
```

13.1.2. Прерываемое приобретение замка

Точно так же, как хронометрированное приобретение замка, которое позволяет использовать защиту от исключающей блокировки в ограниченных по времени действиях, прерываемое приобретение замка позволяет использовать замковую защиту внутри отменяемых действий. В разделе 7.1.6 определено несколько механизмов, таких как приобретение внутреннего замка, который не откликается на прерывание. Эти непрерываемые блокирующие механизмы усложняют реализацию отменяемых задач. Метод `lockInterruptibly` позволяет вам предпринимать попытки приобретения замка, оставаясь при этом отзывчивым к прерыванию, а его включение в `Lock` позволяет избежать создания еще одной категории непрерываемых блокирующих механизмов.

Каноническая структура прерываемого замкового приобретения немного сложнее, чем у нормального замкового приобретения, так как необходимы два блока `try`. (Если прерываемое замковое приобретение может выдать исключение `InterruptedException`, то стандартная идиома замковой защиты на основе `try-finally` будет работать.) В листинге 13.5 используется метод `lockInterruptibly` для реализации метода `sendOnSharedLine` из листинга 13.4, благодаря чему мы можем вызывать его из отменяемой задачи. Хронометрированный метод `tryLock` также откликается на прерывание и поэтому может быть использован, когда вам нужно как хронометрирование, так и приобретение прерываемого замка.

Листинг 13.5. Приобретение прерываемого замка

```
public boolean sendOnSharedLine(String message)
        throws InterruptedException {
    lock.lockInterruptibly();
    try {
        return cancellableSendOnSharedLine(message);
    } finally {
        lock.unlock();
    }
}

private boolean cancellableSendOnSharedLine(String message)
    throws InterruptedException { ... }
```

13.1.3. Неблочно структурированная замковая защита

С внутренними замками пары «приобретение замка — освобождение замка» являются блочно структурированными: замок всегда освобождается в том же базовом блоке, в котором он был приобретен, независимо от того, как управление выходит из блока. Автоматическое освобождение замка упрощает анализ и предотвращает возможные ошибки в написании кода, но иногда требуется более гибкая дисциплина замковой защиты.

В главе 11 мы увидели, как сокращение замковой детализации может повысить масштабируемость. Разделение замка на полосы позволяет различным цепочкам хешей в хешированных коллекциях использовать разные замки. Мы можем применить аналогичный принцип для сокращения замковой детализации в связном списке путем использования отдельного замка для *каждого связного узла*, позволяя разным потокам работать независимо на разных частях списка. Замок для данного узла защищает связные указатели и данные, хранящиеся в этом узле, поэтому при обходе или изменении списка мы должны владеть замком на одном узле, до тех пор пока не приобретем замок на следующем узле; только тогда мы сможем освободить замок на первом узле. Пример этого технического решения, именуемый *замковой защитой «из рук в руки»*, или *сцеплением замков*, приведен в [CPJ 2.5.1.4].

13.2. Соображения по поводу производительности

Когда класс `ReentrantLock` был добавлен в Java 5.0, он предлагал гораздо более высокую оспариваемую (т. е. в условиях конфликта) производитель-

ность, чем у внутренней замковой защиты. Для примитивов синхронизации ключом к масштабируемости является оспариваемая производительность: чем больше ресурсов расходуется на замковое управление и планирование, тем меньше доступно для приложения. Более качественная замковая реализация делает меньше системных вызовов, побуждает применять меньше переключений контекста и инициирует меньше трафика синхронизации памяти на шине совместной памяти, т. е. операций, которые потребляют много времени и отвлекают вычислительные ресурсы от программы.

Java 6 использует усовершенствованный алгоритм управления внутренними замками, подобный используемому в `ReentrantLock`, который значительно закрывает разрыв в масштабируемости. На рис. 13.1 показана разница в производительности между внутренними замками и замком `ReentrantLock` в Java 5.0 и в предрелизной сборке Java 6 на четырехъядерной системе `Opteron` под управлением Solaris. Кривые представляют «ускорение» замка `ReentrantLock` над внутренним замком на единой версии JVM. На Java 5.0 замок `ReentrantLock` предлагает значительно более высокую пропускную способность, но на Java 6 они довольно близки[1]. Тестовая программа является той же, которая используется в разделе 11.5, на этот раз сравнивая пропускную способность хеш-массива `HashMap`, защищенного внутренним замком и замком `ReentrantLock`.

Производительность внутреннего замка резко падает при переходе от единственного потока (без конфликта) более чем к одному; производительность замка `ReentrantLock` падает гораздо меньше, показывая его более высокую масштабируемость. Но на Java 6 это совсем другая история — внутренние замки больше не разваливаются под грузом конфликта, и оба масштабируются одинаково.

Графики, подобные рис. 13.1, напоминают о том, что утверждения типа «*X* быстрее *Y*» в лучшем случае недолговечны. Производительность и масштабируемость зависят от таких факторов платформы, как процессор, число процессоров, размер кэша и характеристики JVM, которые со временем могут меняться[2].

---

[1] Хотя данный график это и не показывает, но разница в масштабируемости между Java 5.0 и Java 6 действительно происходит за счет усовершенствования внутреннего замка, а не регрессии замка `ReentrantLock`.

[2] Когда мы начали эту книгу, замок `ReentrantLock` казался последним словом в замковой масштабируемости. Сейчас внутренняя замковая защита демонстрирует хороший ход за свои деньги. Производительность является не просто движущейся целью, а иногда она бывает быстро движущейся целью.

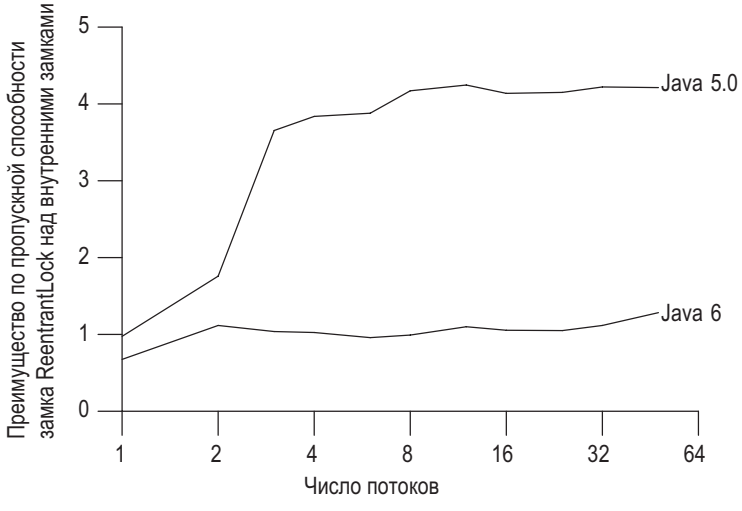

Рис. 13.1. Производительность внутреннего замка против замка ReentrantLock в Java 5.0 и Java 6

Производительность является движущейся целью; вчерашний эталонный тест показывающий, что *X* быстрее *Y*, возможно, уже устарел сегодня.

13.3. Справедливость

Конструктор класса `ReentrantLock` предлагает выбор из двух вариантов *справедливости*: создать *несправедливый* замок (по умолчанию) либо *справедливый* замок. Потоки приобретают справедливый замок в том порядке, в котором они его запрашивали, в то время как несправедливый замок разрешает *проталкивание* (баржирование): запрашивающие замок потоки могут перепрыгивать вперед очереди из ожидающих потоков, если замку случилось оказаться в наличии при его запросе. (Семафор `Semaphore` также предлагает выбор справедливого или несправедливого упорядочивания приобретений.) Несправедливые замки `ReentrantLock` не стараются изо всех сил, чтобы способствовать проталкиванию, — они просто не препятствуют потокам в том, чтобы они проталкивали, если появляются в нужное время. Со справедливым замком только что запросивший его поток ставится в очередь, если замок занят другим потоком либо если потоки стоят

в очереди в ожидании замка; с несправедливым замком поток ставится в очередь, только если замок в настоящее время занят[1].

Разве мы не хотим, чтобы все замки были справедливыми? В конце концов, справедливость — это хорошо, а несправедливость — плохо, верно? (Спросите у своих детей.) Однако когда дело доходит до замковой защиты, справедливость имеет значительную стоимость для производительности из-за издержек на приостановку и возобновление потоков. На практике гарантия статистической справедливости — обещающая, что блокированный поток *в конце концов* получит замок, — часто достаточно хороша и гораздо дешевле в доставке. Некоторые алгоритмы полагаются на справедливую очередность с целью обеспечения их правильности, но эти случаи являются необычными. В большинстве случаев преимущества по производительности у несправедливых замков перевешивают преимущества справедливой очередности.

На рис. 13.2 показан еще один прогон теста на производительность ассоциативного массива `Map`, на этот раз сравнивающий хеш-массив `HashMap`, обернутый справедливыми и несправедливыми замками `ReentrantLock` на четырехъядерной системе `Opteron` под управлением Solaris, выведенный на график с логарифмической шкалой[2]. Штраф за справедливость составляет почти два порядка. *Не платите за справедливость, если она вам не нужна*.

Одна из причин, почему проталкивающие замки работают намного лучше, чем справедливые замки, в условиях жесткого конфликта, заключается в том, что между тем, когда приостановленный поток возобновляется, и тем, когда он работает фактически, может быть значительная задержка. Допустим, поток *A* владеет замком и поток *B* запрашивает этот замок. По-

---

[1] Опрашиваемый метод `tryLock` всегда проталкивает, даже для справедливых замков.

[2] График для хеш-массива `ConcurrentHashMap` довольно волнистый на участке между четырьмя и восемью потоками. Эти изменения почти наверняка происходят от шума в измерениях, который мог быть внесен случайно совпавшими взаимодействиями с хеш-кодами элементов, планированием потоков, изменением размера массива, сбором мусора или другими эффектами системы памяти, или ОС, решившей запустить некоторую периодическую служебную задачу во время работы тестового случая. Реальность такова, что существуют самые разные вариации тестов на производительность, которые обычно не заслуживают того, чтобы их контролировать. Мы не пытались вычистить графики искусственно, потому что реальные измерения производительности также полны шума.

скольку данный замок занят, поток *B* приостанавливается. Когда поток *A* освобождает замок, поток *B* возобновляется, поэтому он может повторить попытку. Между тем, однако, если этот замок запрашивает поток *C*, то существует хороший шанс, что *C* сможет приобрести этот замок, применить его и освободить его до того, как *B* даже закончит просыпаться. В этом случае выигрывают все: поток *B* приобретает замок не позже, чем в противном случае, поток *C* приобретает его намного раньше, и пропускная способность улучшается.

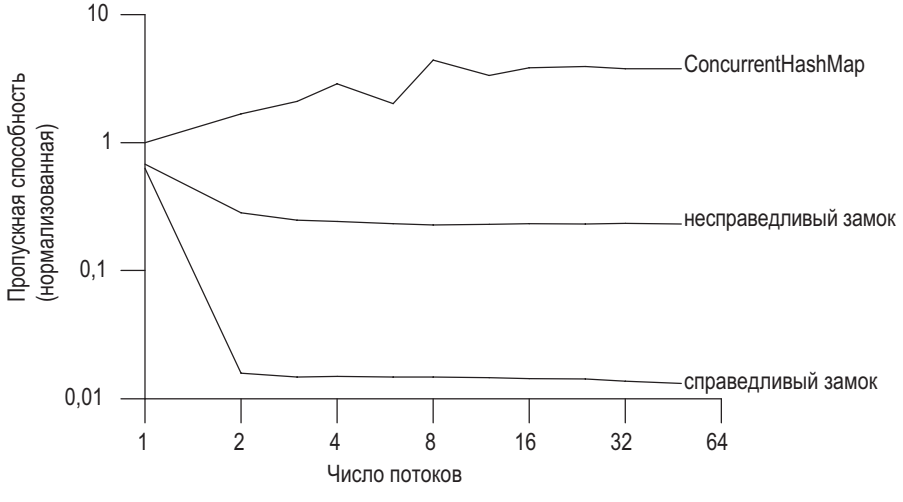

Рис. 13.2. Производительность справедливого замка против несправедливого

Справедливые замки, как правило, лучше всего работают, когда они заняты в течение относительно длительного времени или когда среднее время между замковыми запросами относительно велико. В этих случаях условие, при котором проталкивание обеспечивает преимущество в пропускной способности — когда замок освобожден, но поток в настоящее время просыпается, чтобы претендовать на него, — будет соблюдаться с меньшей вероятностью.

Как и принятый по умолчанию замок `ReentrantLock`, внутренний замок не дает детерминированных гарантий справедливости, но статистические гарантии справедливости большинства реализаций замка достаточно хороши практически для всех ситуаций. Спецификация языка не требует, чтобы JVM-машина реализовывала внутренние замки справедливо, и никакие производственные JVM этого не делают. Замок `ReentrantLock`

не опускает замковую справедливость до новых минимумов — он только делает явным то, что присутствовало все время.

13.4. Выбор между synchronized и ReentrantLock

Замок `ReentrantLock` предоставляет ту же семантику замковой защиты и памяти, что и внутренняя замковая защита, а также дополнительный функционал, такой как хронометрированные замковые ожидания, прерываемые замковые ожидания, справедливость и возможность реализовать неблоково структурированную замковую защиту. Производительность замка `ReentrantLock`, по всей видимости, доминирует над производительностью внутренней замковой защиты, выигрывая немного в Java 6 и резко в Java 5.0. Почему бы тогда не отменить `synchronized` и не рекомендовать всему новому конкурентному коду использовать замок `ReentrantLock`? Некоторые авторы фактически это и предложили, рассматривая `synchronized` как «унаследованную» конструкцию. Но этот взгляд заходит слишком *далеко*.

Внутренние замки по-прежнему имеют значительные преимущества по сравнению с явными замками. Форма их записи знакома и компактна, и многие существующие программы уже используют внутреннюю замковую защиту — и смешивание этих двух инструментов может оказаться запутанным и подверженным ошибкам. Класс `ReentrantLock`, безусловно, является более опасным инструментом, чем синхронизация; если вы забудете обернуть вызов метода `unlock` в блок `finally`, то ваш код, вероятно, будет работать должным образом, но вы создали бомбу замедленного действия, которая может покалечить невинных наблюдателей. Придержите класс `ReentrantLock` для ситуаций, в которых вам нужно что-то, что обеспечивает класс `ReentrantLock`, но что внутренняя замковая защита не делает.

> Класс `ReentrantLock` является продвинутым инструментом для ситуаций, где внутренняя замковая защита оказывается непрактичной. Используйте его, если вам нужен расширенный функционал: хронометрированное, опрашиваемое или прерываемое замковое приобретение, справедливая очередность или неблочно структурированная замковая защита. В противном случае отдавайте предпочтение конструкции `synchronized`.

В Java внутренняя замковая защита имеет еще одно преимущество по сравнению с замком `ReentrantLock`: потоковые дампы показывают, какие фреймы вызовов приобрели какие замки, и могут обнаруживать и идентифицировать потоки, запертые взаимной блокировкой. JVM ничего не знает о том, какие потоки владеют замками `ReentrantLock`, и поэтому не способна помочь в отладке поточных проблем с помощью замка `ReentrantLock`. Это несоответствие устранено путем предоставления интерфейса управления и мониторинга, с помощью которого замки могут регистрироваться, давая возможность замковой информации о замках `ReentrantLock` появляться в поточных дампах и через другие интерфейсы управления и отлаживания. Наличие этой информации для отладки является существенным, если в основном не временным, преимуществом для `synchronized`; замковая информации в поточных дампах спасла многих программистов от полного ужаса. Неблочно структурированная природа замка `ReentrantLock` по-прежнему означает, что замковое приобретение не может быть привязано к определенным стековым фреймам, как это происходит с внутренними замками.

Будущие улучшения производительности, вероятно, будут благоприятствовать механизму `synchronized` более, чем `ReentrantLock`. Поскольку механизм `synchronized` встроен в JVM, он может выполнять такие оптимизации, как замковое выталкивание для ограниченных одним потоком замковых объектов и замковое укрупнение с целью устранения синхронизации с внутренними замками (см. раздел 11.3.2); делать это с библиотечными замками представляется гораздо менее вероятным. Если только вы не развертываете Java 5.0 в обозримом будущем и у вас есть *продемонстрированная* потребность в преимуществах масштабируемости замка `ReentrantLock` на той платформе, то по соображениям производительности не будет хорошей идеей отдать предпочтение замку `ReentrantLock` вместо `synchronized`.

13.5. Замки чтения-записи

Класс `ReentrantLock` реализует стандартный замок взаимного исключения: владеть замком `ReentrantLock` может не более одного потока за раз. Но взаимное исключение часто является более строгой замковой дисциплиной, чем это необходимо для сохранения целостности данных, и поэтому оно ограничивает конкурентность больше, чем необходимо. Взаимное исключение — это консервативная замковая стратегия, кото-

рая предотвращает наложение запись/запись и запись/чтение, а также предотвращает наложение чтение/чтение. Во многих случаях структуры данных характеризуются «преобладанием чтения» — они являются мутируемыми и иногда модифицируются, но большинство обращений к ним связано только с чтением. В этих случаях было бы неплохо ослабить требования к замковой защите, для того чтобы позволить многочисленным читателям обращаться к структуре данных в одно и то же время. До тех пор пока для каждого потока гарантируется актуальное представление данных, и никакой другой поток не модифицирует данные, в то время как читатели их просматривают, никаких проблем не будет. Это то, что замки чтения-записи позволяют делать: ресурс может быть доступен многочисленным читателям либо единственному писателю за раз, но не обоим.

Интерфейс `ReadWriteLock`, показанный в листинге 13.6, предоставляет два замковых объекта `Lock` — один для чтения и один для записи. Для чтения данных, защищаемых интерфейсом `ReadWriteLock`, вы должны сначала приобрести замок чтения, а для модификации данных, защищаемых интерфейсом `ReadWriteLock`, вы должны сначала приобрести замок записи. Хотя может показаться, что существует два отдельных замка, замок чтения и замок записи, — это просто разные представления интегрированного замкового объекта чтения-записи.

Листинг 13.6. Интерфейс ReadWriteLock

```
public interface ReadWriteLock {
    Lock readLock();
    Lock writeLock();
}
```

Замковая стратегия, реализованная замками чтения-записи, позволяет использовать многочисленных одновременных читателей, но только одного единственного писателя. Как и замковый объект `Lock`, замок `ReadWriteLock` допускает многочисленные реализации, которые могут варьироваться в производительности, гарантиях планирования, предпочтениях приобретения, справедливости или семантике замковой защиты.

Замки чтения-записи представляют собой оптимизацию производительности, предназначенную для обеспечения большей конкурентности в определенных ситуациях. На практике замки чтения-записи могут повысить производительность часто используемых структур данных с пре-

обладанием чтения в многопроцессорных системах; в других условиях они работают немного хуже, чем исключающие блокировки из-за их большей сложности. Ответить на вопрос, являются ли они улучшением в любой конкретной ситуации, лучше всего можно с помощью профилирования; поскольку замок `ReadWriteLock` использует `Lock` для замковых частей чтения и записи, относительно легко обменять замок чтения-записи на исключающий, если профилирование определяет, что замок чтения-записи не является выигрышным.

Взаимодействие между замками чтения-записи позволяет реализовать ряд возможных реализаций. Ниже приведено несколько вариантов реализации для замка `ReadWriteLock`.

Предпочтение освобождения замка. Когда писатель освобождает замок записи и читатели и писатели ставятся в очередь, кому следует отдать предпочтение — читателям, писателям или тому, кто запросил первым?

Проталкивание читателя. Если замком владеют читатели, но существуют ожидающие писатели, то следует ли только что прибывшим читателям предоставлять немедленный доступ, или же они должны ждать позади писателей? Разрешение читателям опережать писателей повышает конкурентность, но рискует вызвать голод у писателей.

Повторная входимость. Являются ли замки чтения-записи повторно входимыми?

Понижение. Если поток владеет замком записи, то может ли он приобрести замок чтения, не освобождая замка записи? Это позволило бы писателю «понизиться» до замка чтения, не позволяя другим писателям в это время модифицировать защищенный ресурс.

Повышение. Можно ли замок чтения повысить до замка записи в предпочтении перед другими ожидающими читателями или писателями? Большинство реализаций замка чтения-записи не поддерживают повышение, поскольку без явной операции повышения она предрасположена к возникновению взаимной блокировки. (Если два читателя одновременно попытаются повыситься до замка записи, то ни один из них не освободит замок чтения.)

Замок `ReentrantReadWriteLock` обеспечивает семантику повторно входимой замковой защиты для обоих замков. Как и замок `ReentrantLock`,

замок ReentrantReadWriteLock может быть сконструирован как несправедливый (по умолчанию) либо как справедливый. В случае справедливого замка предпочтение отдается потоку, который ждал дольше всего; если замком владеют читатели и поток запрашивает замок записи, то ни одному читателю больше не разрешается приобретать замок чтения, до тех пор пока писатель не будет обслужен и не освободит замок записи. В случае несправедливого замка порядок, в котором потокам предоставляется доступ, не определен. Разрешается понижение с писателя до читателя; повышение с читателя до писателя не разрешается (попытка сделать это приводит к возникновению взаимной блокировки).

Как и замок ReentrantLock, замок записи в замке ReentrantReadWriteLock имеет уникального владельца и может быть освобожден только потоком, который его приобрел. В Java 5.0 замок чтения ведет себя больше как семафор Semaphore, чем замок, поддерживая только количество активных читателей, а не их идентичности. Это поведение было изменено в Java 6, для того чтобы отслеживать также, каким потокам был предоставлен замок чтения[1].

Замки чтения-записи могут улучшать конкурентность, когда замки обычно удерживаются в течение умеренно длительного времени, и операции в своем большинстве не модифицируют ресурсы. Массив ReadWriteMap в листинге 13.7 использует замок ReentrantReadWriteLock для обертывания ассоциативного массива Map, для того чтобы он мог безопасно использоваться многочисленными читателями совместно и при этом предотвращать конфликты чтение-запись или запись-запись[2]. На самом деле производительность хеш-массива ConcurrentHashMap настолько высока, что вы, вероятно, использовали бы его, а не этот подход, если бы все, что вам было нужно, — это конкурентный хеш-массив, но это техническое решение было бы полезным, если вы хотите предоставить более конкурентный доступ к альтернативной реализации ассоциативного массива Map, такой как LinkedHashMap.

---

[1] Одна из причин этого изменения заключается в том, что в рамках Java 5.0 реализация замка не может различать поток, запрашивающий замок чтения в первый раз, и запрос повторно входимого замка, который сделал бы справедливые замки чтения-записи подверженными взаимной блокировке.

[2] Массив ReadWriteMap не реализует ассоциативный массив Map, потому что реализация методов представления, таких как entrySet и values, была бы трудной, и «легкие» методы обычно достаточны.

Листинг 13.7. Обертывание ассоциативного массива Map блокировкой чтения-записи

```
public class ReadWriteMap<K,V> {
    private final Map<K,V> map;
    private final ReadWriteLock lock = new ReentrantReadWriteLock();
    private final Lock r = lock.readLock();
    private final Lock w = lock.writeLock();

    public ReadWriteMap(Map<K,V> map) {
        this.map = map;
    }

    public V put(K key, V value) {
        w.lock();
        try {
            return map.put(key, value);
        } finally {
            w.unlock();
        }
    }
    // Сделать то же самое для remove(), putAll(), clear()

    public V get(Object key) {
        r.lock();
        try {
            return map.get(key);
        } finally {
            r.unlock();
        }
    }
    // Сделать то же самое для других методов Map только для чтения
}
```

На рис. 13.3 показано сравнение пропускной способности между списком `ArrayList`, обернутым замком `ReentrantLock` и замком `ReadWriteLock` в четырехпутной системе `Opteron` под управлением Solaris. Используемая здесь тестовая программа похожа на тест на производительность ассоциативного массива `Map`, который мы использовали на протяжении всей книги, — каждая операция случайным образом отбирает значение и ищет его в коллекции, и небольшой процент операций модифицирует содержимое коллекции.

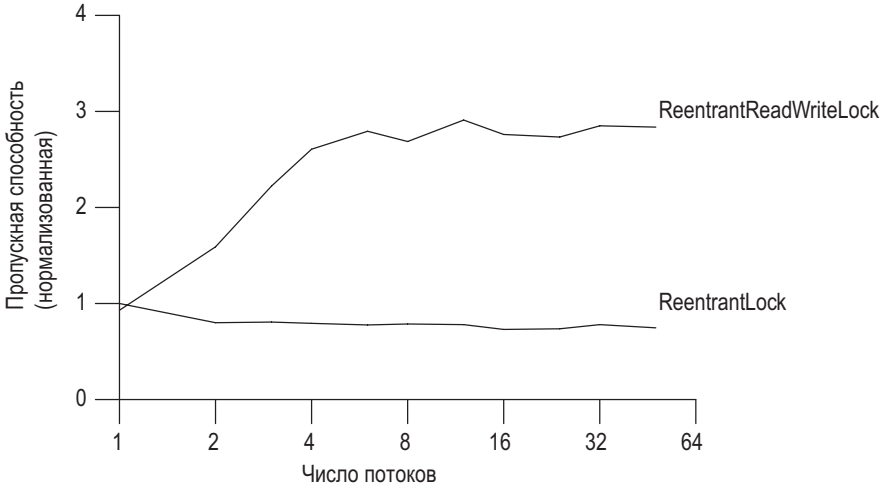

Рис. 13.3. Производительность замка чтения-записи

Итоги

Явные замки предлагают расширенный функциональный набор по сравнению с внутренней замковой защитой, включая бóльшую гибкость при работе с отсутствием замков и больший контроль над поведением, связанным с соблюдением очередности. Но замок `ReentrantLock` не является полной заменой механизма `synchronized`; используйте его только тогда, когда вам нужен функционал, которого не хватает в механизме `synchronized`.

Замки чтения-записи позволяют многочисленным читателям обращаться к защищаемому объекту конкурентно, предлагая потенциал для улучшения масштабируемости при обращении к структурам данных с преобладанием чтения.

14 Построение настраиваемых синхронизаторов

Библиотеки классов включают ряд *зависимых от состояния* классов — имеющих операции *с предусловиями на основе состояния*, — таких как `FutureTask`, `Semaphore` и `BlockingQueue`. Например, вы не можете удалить элемент из пустой очереди или извлечь результат задачи, которая еще не завершена; перед выполнением этих операций необходимо дождаться, когда очередь перейдет в состояние «не пустая» или задача перейдет в состояние «завершена».

Самый простой способ конструирования класса, зависящего от состояния, обычно заключается в надстраивании поверх существующего библиотечного класса, зависящего от состояния; мы сделали это в классе `ValueLatch` на с. 236, используя защелку `CountDownLatch` для обеспечения требуемого поведения замковой защиты. Но если библиотечные классы не предоставляют необходимого вам функционала, то вы также можете построить свои собственные синхронизаторы, используя низкоуровневые механизмы, предоставляемые языком и библиотеками, включая внутренние *очереди условий*, явные объекты `Condition` и структуру `AbstractQueuedSynchronizer`. В этой главе разведываются различные варианты реализации зависимости от состояния и правила использо-

вания механизмов зависимости от состояния, предусмотренные данной платформой.

14.1. Управление зависимостью от состояния

Если в однопоточной программе предусловие на основе состояния (например, «пул соединений не пустой») не соблюдается при вызове метода, то оно никогда не станет истинным. Поэтому классы в последовательных программах могут быть закодированы так, чтобы выдавать сбой, если их предусловия не соблюдаются. Но в конкурентной программе условия на основе состояний могут изменяться в результате действий других потоков: пул, который был пуст несколько инструкций назад, может стать не пустым, поскольку другой поток вернул элемент. Зависящие от состояния методы на конкурентных объектах иногда могут выйти со сбоем, когда их предусловия не удовлетворены, но часто существует оптимальная альтернатива: дождаться, когда предусловие станет истинным.

Зависящие от состояния операции, которые *блокируются*, до тех пор пока операция не сможет продолжаться, более удобны и менее подвержены ошибкам, чем те, которые просто дают сбой. Встроенный механизм очереди условий позволяет потокам блокироваться, до тех пор пока объект не войдет в состояние, допускающее продвижение, и пробуждать блокированные потоки, когда они могут быть способны продвигаться дальше. Мы рассмотрим детали очередей условий в разделе 14.2, но для того, чтобы мотивировать ценность эффективного механизма ожидания по условию, мы сначала покажем, как можно (с большими усилиями) бороться с зависимостью от состояния с помощью опроса и сна.

Зависящее от состояния блокирующее действие принимает форму, показанную в листинге 14.1. Паттерн замковой защиты несколько необычен в том, что замок освобождается и приобретается заново в середине действия. Переменные состояния, составляющие предусловие, должны быть защищены замком объекта, для того чтобы они могли оставаться постоянными во время проверки предусловия. Но если предусловие не соблюдается, то замок должен быть освобожден, для того чтобы другой поток мог модифицировать состояние объекта — в противном случае предусловие никогда не станет истинным. Замок затем должен быть приобретен заново, прежде чем снова проверить предусловие.

Листинг 14.1. Структура блокирующих действий, зависящих от состояния

```
void blockingAction() throws InterruptedException {
    acquire lock on object state
    while (precondition does not hold) {
        release lock
        wait until precondition might hold
        optionally fail if interrupted or timeout expires
        reacquire lock
    }
    perform action
}
```

Ограниченные буферы, такие как очередь `ArrayBlockingQueue`, обычно используются в паттернах «производитель-потребитель». Ограниченный буфер обеспечивает операции `put` и `take`, каждая из которых имеет предусловия: нельзя взять элемент из пустого буфера или поместить элемент в полный буфер. Операции, зависящие от состояния, могут справляться со сбоем предусловия путем выдачи исключения или возврата статуса ошибки (делая ее проблемой вызывающего элемента кода), либо путем блокирования, до тех пор пока объект не перейдет в нужное состояние.

Мы намерены разработать несколько реализаций ограниченного буфера, которые используют разные подходы к обработке сбоя предусловия. Каждый из них расширяет `BaseBoundedBuffer` в листинге 14.2, который реализует классический кольцевой буфер на основе массива, где буферные переменные состояния (`buf`, `head`, `tail` и `count`) защищаются внутренним замком буфера. Он предоставляет синхронизированные методы `doPut` и `doTake`, которые используются подклассами для реализации операций `put` и `take`; базовое состояние скрыто от подклассов.

14.1.1. Пример: распространение сбоя предусловия на вызывающие элементы кода

Класс `GrumpyBoundedBuffer` в листинге 14.3 представляет собой первую грубую попытку реализации ограниченного буфера. Методы `put` и `take` синхронизуются для обеспечения исключительного доступа к состоянию буфера, так как оба используют логику «проверить и затем действовать» при обращении к буферу.

Хотя этот подход достаточно прост в реализации, он надоедлив в использовании. Исключения предположительно должны относиться к ис-

ключительным условиям [EJ пункт 39]. Условие «буфер заполнен» не является исключительным для ограниченного буфера, так же как условие «красный» является исключительным для сигнала светофора. Упрощение реализации буфера (побуждая вызывающий элемент кода управлять зависимостью от состояния) более чем компенсируется существенной сложностью его использования, так как теперь вызывающий элемент кода должен быть готов перехватывать исключения и, возможно, повторять попытку для каждой буферной операции[1]. Хорошо структурированный вызов метода `take` показан в листинге 14.4 — не очень красивый, в особенности если `put` и `take` вызываются по всей программе.

Листинг 14.2. Базовый класс для реализации ограниченного буфера

```
@ThreadSafe
public abstract class BaseBoundedBuffer<V> {
    @GuardedBy("this") private final V[] buf;
    @GuardedBy("this") private int tail;
    @GuardedBy("this") private int head;
    @GuardedBy("this") private int count;

    protected BaseBoundedBuffer(int capacity) {
        this.buf = (V[]) new Object[capacity];
    }

    protected synchronized final void doPut(V v) {
        buf[tail] = v;
        if (++tail == buf.length)
            tail = 0;
        ++count;
    }

    protected synchronized final V doTake() {
        V v = buf[head];
        buf[head] = null;
        if (++head == buf.length)
            head = 0;
        --count;
        return v;
    }
```

продолжение ⇗

[1] Перенос зависимости от состояния обратно вызывающему элементу кода также делает почти невозможным такие действия, как сохранение упорядоченности FIFO; заставляя вызывающий элемент кода повторять попытку, вы теряете информацию о том, кто прибыл первым.

Листинг 14.2 *(продолжение)*

```
    public synchronized final boolean isFull() {
        return count == buf.length;
    }

    public synchronized final boolean isEmpty() {
        return count == 0;
    }
}
```

Листинг 14.3. Ограниченный буфер, который блокируется при невыполнении предусловий

```
@ThreadSafe
public class GrumpyBoundedBuffer<V> extends BaseBoundedBuffer<V> {
    public GrumpyBoundedBuffer(int size) { super(size); }

    public synchronized void put(V v) throws BufferFullException {
        if (isFull())
            throw new BufferFullException();
        doPut(v);
    }

    public synchronized V take() throws BufferEmptyException {
        if (isEmpty())
            throw new BufferEmptyException();
        return doTake();
    }
}
```

Листинг 14.4. Логика клиента для вызова буфера GrumpyBoundedBuffer

```
while (true) {
    try {
        V item = buffer.take();
        // использовать элемент
        break;
    } catch (BufferEmptyException e) {
        Thread.sleep(SLEEP_GRANULARITY);
    }
}
```

Вариант этого подхода заключается в возврате значения ошибки, когда буфер находится в неправильном состоянии. Незначительное улучшение здесь в том, что он не злоупотребляет механизмом исключения, выдавая исключение, которое в действительности означает «извините, повторите попытку», но это не решает фундаментальной задачи «вызывающие элементы кода должны сами справляться со сбоями предусловий»[1].

Клиентский код в листинге 14.4 не является единственным способом реализации логики повтора. Вызывающий элемент кода мог немедленно повторить метод `take`, не засыпая — подход, известный как *ожидание в холостом цикле,* или *занятое ожидание* (busy waiting), или спин-ожидание (spin waiting). Этот подход может потребить довольно много процессорного времени, если состояние буфера не изменяется в течение некоторого времени. С другой стороны, если потребитель решает заснуть, для того чтобы не потреблять так много процессорного времени, то он может легко «проспать», если состояние буфера изменится вскоре после вызова метода `sleep`. Поэтому клиентский код остается перед выбором между плохим потреблением процессора со стороны процедуры ожидания в холостом цикле и слабой отзывчивостью сна. (Где-то между занятым ожиданием и сном будет находиться вызов `Thread.yield` в каждой итерации, который является намеком планировщику о том, что сейчас самое разумное время позволить другому потоку работать. Если вы ждете, когда другой поток что-то сделает, то это может произойти быстрее, если вы уступите процессор, а не потребите свой полный квант планирования.)

14.1.2. Пример: грубая блокировка с помощью опрашивания и сна

Класс `SleepyBoundedBuffer` в листинге 14.5 пытается избавить вызывающие элементы кода от неудобства реализации логики повтора при каждом вызове, инкапсулируя один и тот же грубый механизм повтора «опроса и сна» в операциях `put` и `take`. Если буфер пуст, то операция `take` спит до тех пор, пока другой поток не поместит немного данных в буфер;

---

[1] Класс `Queue` предлагает оба этих варианта — `poll` возвращает `null`, если очередь пуста, и `remove` выдает исключение — но класс `Queue` не предназначен для использования в паттернах «производитель-потребитель». Класс `BlockingQueue`, операции которого блокируются, до тех пор пока очередь не перейдет в нужное состояние, является более качественным выбором, когда производители и потребители будут выполняться конкурентно.

если буфер полон, то операция `take` спит до тех пор, пока другой поток не освободит место, удалив немного данных. Этот подход инкапсулирует управление предусловиями и упрощает использование буфера — определенно шаг в правильном направлении.

Листинг 14.5. Ограниченный буфер с использованием грубой блокировки

```
@ThreadSafe
public class SleepyBoundedBuffer<V> extends BaseBoundedBuffer<V> {
    public SleepyBoundedBuffer(int size) { super(size); }

    public void put(V v) throws InterruptedException {
        while (true) {
            synchronized (this) {
                if (!isFull()) {
                    doPut(v);
                    return;
                }
            }
            Thread.sleep(SLEEP_GRANULARITY);
        }
    }

    public V take() throws InterruptedException {
        while (true) {
            synchronized (this) {
                if (!isEmpty())
                    return doTake();
            }
            Thread.sleep(SLEEP_GRANULARITY);
        }
    }
}
```

Реализация класса `SleepyBoundedBuffer` сложнее, чем предыдущая попытка[1]. Буферный код должен проверить соответствующее условие состояния с занятым замком буфера, поскольку переменные, представляющие условие состояния, защищаются замком буфера. Если тест не проходит, то выполня-

[1] Мы избавим вас от деталей других пяти реализаций ограниченных буферов по протоколу Snow White, в особенности `SneezyBoundedBuffer`.

ющий поток засыпает на некоторое время, сначала освобождая замок, для того чтобы другие потоки могли обратиться к буферу[1]. Когда поток просыпается, он приобретает замок и пытается снова, чередуя сон и проверку условия состояния, до тех пор пока операция не сможет быть продолжена.

С точки зрения вызывающего элемента кода это работает безупречно — если операция может быть продолжена немедленно, он это делает, в противном случае он блокируется — и вызывающему элементу кода не нужно иметь дело с механикой сбоя и повтора. Выбор детализации сна является компромиссом между отзывчивостью и процессорным потреблением; чем меньше гранулярность сна, тем больше отзывчивость, но и больше потребляется процессорных ресурсов. На рис. 14.1 показано, как гранулярность сна может повлиять на отзывчивость: между тем, когда буферное пространство становится доступным, и тем, когда поток, просыпаясь, делает повторную попытку, может возникнуть задержка.

Буфер `SleepyBoundedBuffer` также создает еще одно требование для вызывающего элемента кода — работа с исключением `InterruptedException`. Когда метод блокирует в ожидании, когда условие станет истинным, вежливым поведением является предоставление механизма отмены (см. главу 7). Как и большинство благополучных блокирующих библиотечных методов, `SleepyBoundedBuffer` поддерживает отмену через прерывание, возвращаясь досрочно и выдавая исключение `InterruptedException`, если он прерван.

Эти попытки синтезировать блокирующую операцию из опроса и сна стоили довольно больших усилий. Было бы неплохо иметь способ приостановки потока, но такой, который обеспечивал бы быстрое пробуждение, когда определенное условие (такое как буфер больше не полон) становится истинным. Это именно то, что делают *очереди условий*.

14.1.3. Очереди условий на освобождение

Очереди условий (condition queue) — это все равно, что сигнал «тост готов» от тостера. Если вы его слушаете, то будете незамедлительно уведомлены, когда тост будет готов, и можете прекратить делать то, что вы делаете (или, может быть, хотите сначала покончить с газетой), и взять

[1] Плохая идея, когда поток засыпает или, в противном случае, блокируется с удержанием замка, но в данном случае это даже хуже, потому что желаемое условие (буфер полон/пуст) никогда не может стать истинным, если замок не будет освобожден!

свой тост. Если вы его не слушаете (возможно, вы вышли на улицу купить газету), то можете пропустить уведомление. Однако вернувшись на кухню, вы можете взглянуть на состояние тостера и либо вынуть тост, если он готов, либо начать слушать звонок снова, если он еще не готов.

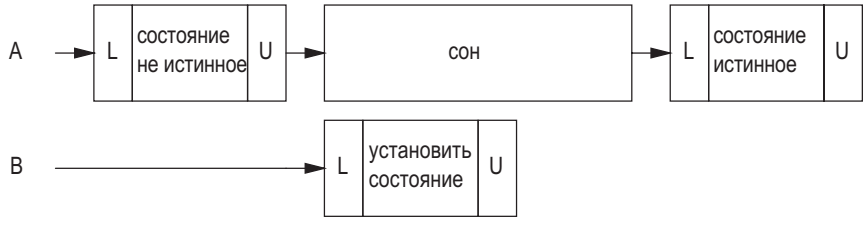

Рис. 14.1. Поток проспал, потому что состояние стало истинным сразу после того, как он уснул

Очередь условий так называется потому, что она предоставляет группе потоков — так называемому *набору ожидания* — возможность ждать, когда конкретное условие станет истинным. В отличие от обычных очередей, в которых элементы являются элементами данных, элементами очереди условий являются потоки, ожидающие наступления условия.

Так же, как всякий объект Java, который может действовать как замок, всякий объект может также действовать как очередь условий, и методы `wait`, `notify` и `notifyAll` в `Object` образуют API для внутренних очередей условий. Внутренний замок объекта и его внутренняя очередь условий между собой связаны: для того чтобы вызвать любой из методов очереди условий для объекта *X*, вы должны владеть замком *X*. Это объясняется тем, что механизм ожидания условий на основе состояний неизбежно тесно связан с механизмом сохранения непротиворечивости состояния: нельзя дожидаться условия, если не удается проверить состояние, и нельзя освободить другой поток из ожидания по условию, если не удается модифицировать состояние.

Метод `Object.wait` атомарно освобождает замок и просит ОС приостановить текущий поток, давая другим потокам приобрести замок и, следовательно, модифицировать состояние объекта. Проснувшись, он снова приобретает замок перед возвращением. Интуитивно вызов метода `wait` означает «Я хочу поспать, но разбудите меня, когда произойдет что-то интересное», а вызов методов уведомления означает «что-то интересное произошло».

Класс `BoundedBuffer` в листинге 14.6 реализует ограниченный буфер с использованием методов `wait` и `notifyAll`. Он проще, чем спящая версия, и эффективнее (пробуждение происходит реже, если состояние буфера не изменяется), и отзывчивее (пробуждение происходит незамедлительно, когда происходит интересное изменение состояния). Большое улучшение, но обратите внимание, что введение очередей условий не изменило семантику по сравнению со спящей версией. Это просто оптимизация в нескольких размерностях: эффективность процессора, издержки на переключение контекста и отзывчивость. Очереди условий не позволяют вам делать то, что вы не можете делать с засыпанием и опрашиванием[1], но они делают намного проще и эффективнее возможность выражать и управлять зависимостью от состояния.

Листинг 14.6. Ограниченный буфер с использованием очередей условий

```
@ThreadSafe
public class BoundedBuffer<V> extends BaseBoundedBuffer<V> {
    // УСЛОВНЫЙ ПРЕДИКАТ: неполный (!isFull())
    // УСЛОВНЫЙ ПРЕДИКАТ: непустой (!isEmpty())

    public BoundedBuffer(int size) { super(size); }

    // БЛОКИРУЕТ ДО ТЕХ ПОР, ПОКА: неполный
    public synchronized void put(V v) throws InterruptedException {
        while (isFull())
            wait();
        doPut(v);
        notifyAll();
    }

    // БЛОКИРУЕТ ДО ТЕХ ПОР, ПОКА: непустой
    public synchronized V take() throws InterruptedException {
        while (isEmpty())
            wait();
        V v = doTake();
        notifyAll();
        return v;
    }
}
```

[1] Это не совсем верно; справедливая очередь условий может гарантировать относительный порядок, в котором потоки освобождаются из набора ожиданий. Внутренние очереди условий, подобно внутренним замкам, не предлагают справедливую очередность; явный объект `Conditions` предлагает на выбор справедливую либо несправедливую очередность.

Класс `BoundedBuffer` наконец-то достаточно хорош в использовании — он прост в использовании и разумно управляет зависимостью от состояния[1]. Производственная версия должна также включать хронометрированные версии методов `put` и `take`, для того чтобы блокирующие операции могли истекать, если они не могут быть завершены в рамках бюджета времени. Хронометрированная версия метода `Object.wait` облегчает их реализацию.

14.2. Использование очередей условий

Очереди условий упрощают создание эффективных и отзывчивых классов, зависящих от состояния, но их по-прежнему легко использовать неправильно; существует ряд правил их использования, которые не применяются компилятором или платформой. (Это одна из причин, почему следует делать надстройку поверх классов, таких как `LinkedBlockingQueue`, `CountDownLatch`, `Semaphore` и `FutureTask`, когда это возможно; это намного проще, если вы можете обойтись этим.)

14.2.1. Условный предикат

Ключом к правильному использованию очередей условий является идентификация условных предикатов, которых объект может ожидать. Условный предикат вызывает бóльшую часть путаницы вокруг методов `wait` и `notify`, потому что у него нет инстанциации в API и ничего ни в спецификации языка, ни в реализации JVM не обеспечивает его правильное использование. Он практически совсем не упоминается непосредственно в спецификации языка или Javadoc. Но без него ожидания по условию работать не будут.

Прежде всего, условный предикат — это предусловие, которое делает операцию зависимой от состояния. В ограниченном буфере метод `take` может продолжаться только в том случае, если буфер не пуст; в противном случае он должен ждать. Для метода `take` условным предикатом является «буфер не пуст», который метод `take` должен проверять перед продолжением. Схожим образом, условным предикатом для метода `put` является «буфер не заполнен». Условные предикаты — это выражения,

[1] `ConditionBoundedBuffer` в разделе 14.3 еще лучше: он эффективнее, потому что может использовать разовое уведомление вместо `notifyAll`.

сконструированные из переменных состояния, принадлежащих классу; `BaseBoundedBuffer` выполняет проверку на возникновение условия «буфер не пуст», сравнивая счетчик `count` с нулем, и выполняет проверку на возникновение условия «буфер не заполнен», сравнивая счетчик `count` с размером буфера.

> Документируйте условные предикаты, ассоциированные с очередью условий, и операции, которые ожидают их наступления.

Существует важная трехсторонняя связь в ожидании по условию, состоящая из замковой защиты, метода `wait` и условного предиката. Условный предикат включает переменные состояния, а переменные состояния защищаются замком, поэтому перед проверкой условного предиката мы должны владеть этим замком. Замковый объект и объект очереди условий (объект, на котором активируются методы `wait` и `notify`) также должны быть одним и тем же объектом.

В `BoundedBuffer` состояние буфера защищается замком буфера, а объект-буфер используется в качестве очереди условия. Метод `take` приобретает замок буфера, а затем проверяет условный предикат (что буфер не пустой). Если буфер действительно не пустой, он удаляет первый элемент; он может сделать это, потому что по-прежнему владеет замком, защищающим состояние буфера.

Если условный предикат не является истинным (буфер пуст), то метод `take` должен подождать до тех пор, пока другой поток не поместит объект в буфер. Это делается путем вызова метода `wait` на внутренней очереди условий, принадлежащей буферу, для чего требуется владеть замком очереди условий. Как и полагается в тщательном проекте, метод `take` уже владеет этим замком, который нужен для проверки условного предиката (и если условный предикат был истинным, то модифицирует состояние буфера в той же атомарной операции). Метод `wait` освобождает замок, блокирует текущий поток и ожидает истечения указанного тайм-аута, поток прерывается либо пробуждается уведомлением. После того как поток пробуждается, метод `wait` повторно приобретает замок перед возвращением. Поток, пробуждающийся от ожидания, не получает никакого особого приоритета в повторном приобретении замка; он конфликтует за замок, как и любой другой поток, пытающийся войти в синхронизированный блок `synchronized`.

Каждый вызов метода `wait` неявно ассоциирован с конкретным *условным предикатом*. Во время вызова метода `wait` по тому или иному условному предикату вызывающий элемент кода должен уже владеть замком, ассоциированным с очередью условий, и этот замок также должен защищать переменные состояния, из которых состоит условный предикат.

14.2.2. Слишком раннее пробуждение

Как будто трехсторонняя связь между замком, условным предикатом и очередью условий не была уже достаточно сложной, то, что метод `wait` возвращается, не обязательно означает, что условный предикат, которого поток ожидает, стал истинным.

Одна-единственная внутренняя очередь условий может использоваться с более чем одним условным предикатом. Когда ваш поток пробуждается, потому что кто-то вызвал метод `notifyAll`, это не означает, что условный предикат, которого *вы* ждали, теперь истинен. (Это все равно что ваш тостер и кофеварка имеют единый звонок; когда он звонит, вам все равно нужно посмотреть, какое устройство подняло сигнал.[1]) Вдобавок методу `wait` даже разрешено возвращаться «мнимо» — не в ответ на вызов метода `notify` любым потоком[2].

Когда управление повторно входит в код, который вызывает метод `wait`, он повторно запрашивает замок, связанный с очередью условий. Является ли сейчас условный предикат истинным? Возможно. Он мог быть истинным в момент, когда уведомляющий поток вызвал метод `notifyAll`, но мог стать опять ложным к тому времени, когда *вы* повторно приобрели замок. Между моментом, когда ваш поток пробудился, и моментом, когда метод `wait` повторно приобрел замок, другие потоки могли приобрести замок и изменить состояние объекта. Либо, возможно, он вовсе не был истинным, так как вы вызвали метод `wait`. Вы не знаете, почему другой

---

[1] Эта ситуация на самом деле довольно хорошо описывает кухню Тима; там так много устройств, которые издают сигналы, что, когда вы слышите один, то должны проверить тостер, микроволновку, кофеварку и еще несколько других, чтобы определить источник сигнала.

[2] В продолжение аналогии с завтраком — это похоже на тостер с неплотным соединением, при котором звонок срабатывает, когда тост готов, а также иногда, когда он не готов.

поток вызвал метод `notify` или `notifyAll`; возможно, это произошло потому, что *другой* условный предикат, ассоциированный с той же очередью условий, стал истинным. Весьма распространены многочисленные условные предикаты в расчете на очередь условий — `BoundedBuffer` использует ту же очередь условий как для предиката «не полный», так и для предиката «не пустой»[1].

По всем этим причинам, когда вы просыпаетесь от ожидания `wait`, вы должны *снова* проверить условный предикат и вернуться к ожиданию (или выдать сбой), если оно еще не истинно. Так как вы можете просыпаться многократно и не иметь истинного условного предиката, то, следовательно, должны всегда вызывать `wait` изнутри цикла, проверяя условный предикат в каждой итерации. Каноническая форма ожидания по условию показана в листинге 14.7.

Листинг 14.7. Каноническая форма для методов, зависящих от состояния

```
void stateDependentMethod() throws InterruptedException {
    // условный предикат должен быть защищен замком
    synchronized(lock) {
    while (!conditionPredicate())
        lock.wait();
    // теперь объект находится в желаемом состоянии
    }
}
```

При использовании ожиданий по условию (`Object.wait` или `Condition.await`) следует:

- всегда иметь условный предикат — некую проверку состояния объекта, которая должна удовлетворяться перед продолжением;
- всегда проверять условный предикат перед вызовом метода `wait` и после возврата из метода `wait`;
- всегда вызывать метод `wait` в цикле;
- обеспечивать, чтобы переменные состояния, составляющие условный предикат, были защищены замком, ассоциированным с очередью условий;

[1] Для потоков фактически возможна ситуация, когда они одновременно ожидают предикаты «не полный» и «не пустой»! Это может произойти, когда число производителей/потребителей превышает емкость буфера.

- владеть замком, ассоциированным с очередью условий, при вызове методов `wait`, `notify` или `notifyAll`;
- освобождать замок не после проверки условного предиката, а перед тем, как действовать по нему.

14.2.3. Пропущенные сигналы

В главе 10 обсуждались сбои в жизнеспособности, такие как взаимная блокировка и активная блокировка. Еще одна форма сбоя в жизнеспособности — это *пропущенные сигналы* (missed signal). Пропущенный сигнал возникает, когда поток должен ожидать наступление определенного условия, которое уже истинно, но при этом не проверяет условный предикат перед ожиданием. Теперь поток ожидает уведомления о событии, которое уже произошло. Это как если запустить тостер, выйти наружу за газетой, при этом звонок срабатывает, пока вы находитесь снаружи, а затем сесть за кухонный стол, ожидая звонка тостера. Вы будете ждать долго, потенциально — вечно[1]. В отличие от джема для вашего тоста, уведомление не является «липким» — если поток *A* уведомляет об очереди условий и поток *B* впоследствии ожидает в той же очереди условий, *B* пробуждается не сразу — для того чтобы разбудить *B*, требуется еще одно уведомление. Пропущенные сигналы являются результатом ошибок кодирования, как те, о которых предупреждают элементы приведенного выше списка, например отказ от проверки условного предиката перед вызовом метода `wait`. Если вы структурируете ожидания по условию, как показано в листинге 14.7, то у вас не возникнет проблем с пропущенными сигналами.

14.2.4. Уведомление

На данный момент мы описали половину того, что происходит в ожидании по условию: ожидание. Другая половина — уведомление. В ограниченном буфере метод `take` блокирует продвижение, если он вызывается, когда буфер пуст. Для того чтобы метод `take` *разблокировал* продвижение, когда буфер становится не пустым, мы должны обеспечить, чтобы *каждая* ветвь кода, в которой буфер может стать не пустым, отправляла уведомление. В `BoundedBuffer` есть только одно такое место — после ме-

[1] Для того чтобы выйти из этого ожидания, кто-то другой должен бы сделать тост, но это только ухудшит ситуацию; когда прозвучит сигнал, у вас возникнет разногласие о принадлежности тоста.

тода put. Поэтому метод put вызывает метод notifyAll после успешного добавления объекта в буфер. Схожим образом метод take вызывает метод notifyAll после удаления элемента, что указывает на то, что буфер больше не может быть полным в случае, если какие-либо потоки ожидают по условию «не полный».

> Всякий раз, когда вы ожидаете по условию, обеспечьте, чтобы кто-то делал уведомление, когда условный предикат становится истинным.

Существует два метода уведомления в API очереди условий — notify и notifyAll. Для вызова любого из них необходимо владеть замком, ассоциированным с объектом очереди условий. Вызов notify побуждает JVM выбрать один поток, ожидающий пробуждения в этой очереди условий; вызов метода notifyAll пробуждает *все* потоки, ожидающие в этой очереди условий. Поскольку во время вызова методов notify или notifyAll вы должны владеть замком на объекте очереди условий и ожидающие потоки не могут вернуться из wait без приобретения замка, уведомляющий поток должен быстро освободить замок с целью обеспечения того, чтобы ожидающие потоки разблокировались как можно скорее.

Поскольку многочисленные потоки могут ожидать разных условных предикатов в одной и той же очереди условий, использование метода notify вместо метода notifyAll может быть опасным, прежде всего потому, что разовое уведомление подвержено проблеме, родственной пропущенным сигналам.

BoundedBuffer является хорошей иллюстрацией того, почему метод notifyAll следует предпочесть разовому методу notify в большинстве случаев. Очередь условий используется для двух разных условных предикатов: «не полный» и «не пустой». Предположим, поток A ожидает в очереди условий предикат P_A, а поток B ожидает в той же очереди условий предикат P_B. Теперь предположим, что P_B становится истинным, и поток C выполняет разовое уведомление notify: JVM разбудит один поток по своему выбору. Если выбран поток A, то он проснется, увидит, что P_A еще не истинен, и вернется к ожиданию. Между тем, поток B, который мог бы сейчас продвинуться вперед, не пробуждается. Это не совсем пропущенный сигнал — это, скорее, «угнанный сигнал» (hijacked signal), — но проблема та же: поток ожидает сигнала, который уже произошел (или должен был произойти).

Разовое уведомление notify можно использовать вместо notifyAll только при соблюдении обоих следующих условий:

Единообразные ожидатели. С очередью условий ассоциирован один и только один условный предикат, и каждый поток выполняет одинаковую логику по возвращении из ожидания wait; и

Один вошел, один вышел. Уведомление об условной переменной позволяет продолжиться не более чем одному потоку.

BoundedBuffer отвечает требованиям «один вошел, один вышел», но не отвечает требованиям единообразных ожидателей, поскольку ожидающие потоки могут ожидать либо условия «не полный», либо условия «не пустой». Защелка «стартовый шлюз», подобная той, которая используется в классе TestHarness на с. 139, в котором единственное событие освобождает набор потоков, не соответствует требованию «один вошел, один вышел», потому что открытие стартового шлюза позволяет многочисленным потокам проследовать дальше.

Большинство классов не отвечают этим требованиям, поэтому вместо разового метода notify рекомендуется использовать метод notifyAll. Хотя это может быть неэффективно, но гораздо проще обеспечить правильное поведение классов при использовании notifyAll вместо notify.

Эта «господствующая мудрость» заставляет некоторых людей чувствовать себя некомфортно, и не зря. Использование метода notifyAll, когда только один поток может продвигаться вперед, является неэффективным — иногда немного, иногда чересчур. Если десять потоков ожидают в очереди условий, вызов метода notifyAll побуждает каждый из них проснуться и конфликтовать за замок; затем большинство из них или все они снова засыпают. Из этого следует очень много переключений контекста и очень много оспариваемых замковых приобретений для каждого события, которое позволяет (возможно) единственному потоку продвигаться вперед. (В худшем случае использование метода notifyAll приводит к $O(n^2)$ пробуждениям, где n было бы достаточно.) Это еще одна ситуация, когда озабоченности по поводу производительности поддерживают один подход, а озабоченности по поводу безопасности поддерживают другой.

Уведомление, выполняемое методами put и take в BoundedBuffer, является консервативным: уведомление выполняется всякий раз, когда объект по-

мещается в буфер или удаляется из него. Это может быть оптимизировано при наблюдении, что поток может быть освобожден от ожидания, только если буфер переходит от пустого к не пустому или от полного к не полному, и с уведомлением, только если методы `put` или `take` произвели один из этих переходов из состояния в состояние. Это называется *условным уведомлением*. В то время как условное уведомление может улучшить производительность, его сложно настроить (а также оно усложняет реализацию подклассов), и поэтому должно использоваться осторожно. В листинге 14.8 показано использование условного уведомления в методе `BoundedBuffer.put`.

Разовое уведомление и условное уведомление — это оптимизации. Как всегда, при использовании этих оптимизаций следуйте принципу «сначала сделай это правильно, а затем сделай это быстро — *если* это еще не работает достаточно быстро»; применяя их неправильно, легко внести странные сбои в жизнеспособности.

Листинг 14.8. Использование условного уведомления в методе BoundedBuffer.put

```
public synchronized void put(V v) throws InterruptedException {
    while (isFull())
        wait();
    boolean wasEmpty = isEmpty();
    doPut(v);
    if (wasEmpty)
        notifyAll();
}
```

14.2.5. Пример: шлюзовый класс

Защелка стартового шлюза в классе `TestHarness` на с. 139 была сконструирована с начальным счетчиком, равным единице, для *создания двоичной защелки*, то есть с двумя состояниями: начальным и конечным. Защелка не дает потокам проходить через стартовый шлюз до тех пор, пока он не будет открыт, и в этот момент все потоки могут проходить сквозь. Хотя этот механизм защелки часто является как раз тем, что нужно, иногда недостаток проявляется в том, что защелку, сконструированную таким образом, нельзя закрыть снова после ее открытия.

Как показано в листинге 14.9, легко можно разработать перезакрываемый класс `ThreadGate`, используя ожидания по условию. Класс `ThreadGate` позволяет шлюзам быть открытыми и закрытыми, предоставляя метод `await`, который блокируется до тех пор, пока ворота не будут открыты. Метод `open` использует метод `notifyAll`, поскольку семантика этого класса не проходит проверки «один вошел, один вышел» на разовое уведомление.

Условный предикат, используемый методом `await`, сложнее, чем простая проверка методом `isOpen`. Это необходимо, потому что если *N* потоков ожидают у шлюза в то время, когда он открыт, им всем должно быть разрешено проходить. Но если шлюз открывается и закрывается в быстрой последовательности, то все потоки не могли бы быть освобождены, если бы метод `await` проверял только `isOpen`: к тому времени, когда все потоки получат уведомление, повторно приобретут замок и выйдут из ожидания `wait`, шлюз, возможно, будет уже закрыт снова. Поэтому `ThreadGate` использует несколько более сложный условный предикат: всякий раз, когда шлюз закрыт, счетчик «поколения» наращивается и поток может пройти `await`, если шлюз открыт сейчас или если шлюз открыт с тех пор, как этот поток прибыл к шлюзу.

Поскольку класс `ThreadGate` поддерживает только ожидание момента, когда шлюз откроется, он выполняет уведомление только в `open`; для поддержки операций «ждать открытия» и «ждать закрытия» ему пришлось бы уведомлять как в `open`, так и в `close`. Это иллюстрирует, почему зависимые от состояния классы могут быть хрупкими в сопровождении — добавление новой зависящей от состояния операции может потребовать изменения многих ветвей кода, которые модифицируют состояние объекта, с целью обеспечить возможность выполнять соответствующие уведомления.

14.2.6. Вопросы безопасности подклассов

Использование условного или разового уведомления вводит ограничения, которые могут осложнить создание подкласса [CPJ 3.3.3.3]. Если вы хотите вообще поддерживать создание подкласса, то должны структурировать свой класс таким образом, чтобы подклассы могли добавлять соответствующие уведомления от имени базового класса, если он разделяется на подклассы, нарушающие одно из требований для разового или условного уведомления.

Листинг 14.9. Перезакрываемый шлюз с использованием методов wait и notifyAll

```
@ThreadSafe
public class ThreadGate {
    // УСЛОВНЫЙ ПРЕДИКАТ: opened-since(n) (isOpen || generation>n)
    @GuardedBy("this") private boolean isOpen;
    @GuardedBy("this") private int generation;

    public synchronized void close() {
        isOpen = false;
    }

    public synchronized void open() {
        ++generation;
        isOpen = true;
        notifyAll();
    }

    // БЛОКИРУЕТ ДО ТЕХ ПОР, ПОКА: opened-since(generation on entry)
    public synchronized void await() throws InterruptedException {
        int arrivalGeneration = generation;
        while (!isOpen && arrivalGeneration == generation)
            wait();
    }
}
```

Зависимый от состояния класс должен либо полностью предоставлять (и документировать) подклассам свои протоколы ожидания и уведомления, либо вообще запрещать участие в них подклассов. (Это правило является расширением формулировки «проектируй и документируй для наследования, или же запрети это» [EJ пункт 15].) По крайней мере, при проектировании зависящего от состояния класса для наследования требуется предоставить очереди условий и замки, а также задокументировать предикаты условий и политику синхронизации; также может потребоваться предоставить базовые переменные состояния. (Худшее, что может сделать зависящий от состояния класс, — это предоставить свое состояние подклассам, но при этом не задокументировать свои протоколы для ожидания и уведомления; это все равно что класс, который предоставляет свои переменные состояния, но который при этом не документирует свои инварианты.)

Один из путей достижения этого состоит в том, чтобы фактически запретить создание подклассов, сделав класс финальным либо скрыв от под-

классов очереди условий, замки и переменные состояния. В противном случае, если подкласс делает что-то, чтобы подорвать то, как базовый класс использует метод `notify`, он должен быть в состоянии возместить ущерб. Возьмем неограниченный блокирующий стек, в котором операция `pop` блокирует, если стек пуст, но операция `push` всегда может продолжаться. Это соответствует требованиям для разового уведомления. Если этот класс использует разовое уведомление, а подкласс добавляет блокирующий метод «вытолкнуть два элемента подряд», то теперь есть два класса ожидателей: те, кто ожидает выталкивания одного элемента, и те, кто ожидает выталкивания двух. Но если базовый класс предоставляет очередь условий и документирует свои протоколы для ее использования, то подкласс может переопределить метод `push` для выполнения метода `notifyAll`, восстанавливая безопасность.

14.2.7. Инкапсулирование очередей условий

Обычно лучше инкапсулировать очередь условий таким образом, чтобы она была недоступна вне классовой иерархии, в которой она используется. В противном случае у вызывающих элементов кода может возникнуть соблазн подумать, что они понимают ваши протоколы ожидания и уведомления и используют их способом, который противоречит вашему проекту. (Невозможно обеспечить соблюдение требования единообразных ожидателей для разового уведомления, если объект очереди условий недоступен для кода, который вы не контролируете; если чужой код ошибочно ожидает в очереди условий, то это может нарушить протокол уведомлений и вызвать угнанный сигнал.)

К сожалению, этот совет — инкапсулировать объекты, используемые в качестве очередей условий — противоречит наиболее распространенному паттерну проектирования для потокобезопасных классов, в которых внутренний замок объекта используется для защиты его состояния. Класс `BoundedBuffer` иллюстрирует эту распространенную идиому, где сам объект буфера является замком и очередью условий. Однако `BoundedBuffer` можно легко реструктурировать, для того чтобы использовать приватный замковый объект и очередь условий; единственное различие будет заключаться в том, что он больше не будет поддерживать любую форму замковой защиты на стороне клиента.

14.2.8. Протоколы входа и выхода

Веллингс (Wellings, 2004) характеризует надлежащее использование методов `wait` и `notify` с точки зрения *протоколов входа* и *выхода*. Для каждой операции, зависящей от состояния, и для каждой операции, модифицирующей состояние, от которой зависит состояние другой операции, вы должны определить и задокументировать протокол входа и выхода. Протокол входа является условным предикатом операции; протокол выхода включает в себя проверку всех переменных состояния, которые были изменены операцией, для того чтобы увидеть, могли ли они побудить какой-то другой условный предикат стать истинным, и если да, то протокол выхода включает уведомление об ассоциированной очереди условий.

Абстрактный класс `AbstractQueuedSynchronizer`, над которым надстроено большинство зависимых от состояния классов в `java.util.concurrent` (см. раздел 14.4), эксплуатирует концепцию протокола выхода. Вместо того чтобы позволить классам-синхронизаторам выполнять свои собственные уведомления, он требует, чтобы методы-синхронизаторы возвращали значение, указывающее на то, могло или нет его действие разблокировать один или несколько ожидающих потоков. Это явное требование API затрудняет «забывание» уведомить о некоторых переходах из состояния в состояние.

14.3. Явные объекты условий

Как мы видели в главе 13, явные замковые объекты `Lock` могут быть полезны в некоторых ситуациях, когда внутренние замки слишком негибки. Подобно тому как замок `Lock` является обобщением внутренних замков, `Condition` (см. листинг 14.10) является обобщением внутренних очередей условий.

Внутренние очереди условий имеют несколько недостатков. Каждый внутренний замок может иметь только одну ассоциированную очередь условий, что означает, что в таких классах, как `BoundedBuffer`, многочисленные потоки могут ожидать в одной и той же очереди условий разных условных предикатов, и наиболее распространенный шаблон замковой защиты предусматривает предоставление объекта очереди условий. Оба эти фактора делают невозможным обеспечить соблюдение требования

единообразных ожидателей для использования метода `notifyAll`. Если вы хотите написать конкурентный объект с многочисленными условными предикатами либо хотите осуществлять больший контроль над видимостью очереди условий, то явные классы `Lock` и `Condition` предлагают более гибкую альтернативу внутренним замкам и очередям условий.

Класс `Condition` ассоциирован с единственным классом `Lock` так же, как очередь условий ассоциирована с единственным внутренним замком; для того чтобы создать условие `Condition`, вызовите `Lock.newCondition` на ассоциированном замке. И так же, как замок предлагает более богатый функционал, чем внутренний замок, `Condition` предлагает более богатый функционал, чем внутренние очереди условий: многочисленные наборы ожиданий в расчете на замок, прерываемые и непрерываемые ожидания по условию, ожидания на основе предельного срока и выбор между справедливой и несправедливой очередностью.

Листинг 14.10. Интерфейс Condition

```
public interface Condition {
    void await() throws InterruptedException;
    boolean await(long time, TimeUnit unit)
        throws InterruptedException;
    long awaitNanos(long nanosTimeout) throws InterruptedException;
    void awaitUninterruptibly();
    boolean awaitUntil(Date deadline) throws InterruptedException;

    void signal();
    void signalAll();
}
```

В отличие от внутренних очередей условий вы можете иметь столько объектов `Condition` в расчете на объект `Lock`, сколько вам необходимо. Объекты `Condition` наследуют значение справедливости ассоциированного с ними объекта `Lock`; в случае справедливых замков потоки освобождаются от ожидания `Condition.await` в порядке FIFO.

> Предупреждение об угрозе: эквивалентами методов `wait`, `notify` и `notifyAll` для объектов `Condition` являются `await`, `signal` и `signalAll`. Однако `Condition` расширяет `Object`, это означает, что он также имеет методы `wait` и `notify`. Вместо них обязательно используйте правильные версии — методы `await` и `signal`!

В листинге 14.11 показана еще одна реализация ограниченного буфера, на этот раз с использованием двух объектов Condition, notFull и notEmpty, для явного представления условных предикатов «не полный» и «не пустой». Когда метод take блокирует, потому что буфер пуст, он ждет на notEmpty, и метод put разблокирует любые потоки, блокированные в методе take, сигнализируя на notEmpty.

Листинг 14.11. Ограниченный буфер с использованием явных переменных состояния

```
@ThreadSafe
public class ConditionBoundedBuffer<T> {
    protected final Lock lock = new ReentrantLock();
    // УСЛОВНЫЙ ПРЕДИКАТ: notFull (count < items.length)
    private final Condition notFull = lock.newCondition();
    // УСЛОВНЫЙ ПРЕДИКАТ: notEmpty (count > 0)
    private final Condition notEmpty = lock.newCondition();
    @GuardedBy("lock")
    private final T[] items = (T[]) new Object[BUFFER_SIZE];
    @GuardedBy("lock") private int tail, head, count;

    // БЛОКИРУЕТ ДО ТЕХ ПОР, ПОКА: notFull
    public void put(T x) throws InterruptedException {
        lock.lock();
        try {
            while (count == items.length)
                notFull.await();
            items[tail] = x;
            if (++tail == items.length)
                tail = 0;
            ++count;
            notEmpty.signal();
        } finally {
            lock.unlock();
        }
    }

    // БЛОКИРУЕТ ДО ТЕХ ПОР, ПОКА: notEmpty
    public T take() throws InterruptedException {
        lock.lock();
        try {
            while (count == 0)
                notEmpty.await();
            T x = items[head];
            items[head] = null;
```

продолжение ⇩

Листинг 14.11 *(продолжение)*

```
            if (++head == items.length)
                head = 0;
            --count;
            notFull.signal();
            return x;
        } finally {
            lock.unlock();
        }
    }
}
```

Поведение класса `ConditionBoundedBuffer` совпадает с поведением класса `BoundedBuffer`, но его использование очередей условий читаемо — легче проанализировать класс, который использует многочисленные объекты `Condition`, чем тот, который использует одну внутреннюю очередь условий с многочисленными условными предикатами. Разделяя два условных предиката на отдельные наборы ожидания, `Condition` упрощает соблюдение требований для разового уведомления. Использование более эффективного метода `signal` вместо `signalAll` уменьшает число переключений контекста и замковых приобретений, инициируемых каждой операцией буфера.

Как и во внутренних замках и очередях условий, трехсторонняя связь между замком, условным предикатом и условной переменной также должна соблюдаться при использовании явных замковых объектов `Lock` и объектов-условий `Condition`. Переменные, участвующие в условном предикате, должны быть защищены замком `Lock`, и замок `Lock` должен удерживаться во время проверки условного предиката и во время вызова методов `await` и `signal`[1].

Выбирайте между использованием явных условий `Condition` и внутренних очередей условий так же, как вы выбрали бы между `ReentrantLock` и `synchronized`: используйте условие `Condition`, если вам нужен его расширенный функционал, такой как справедливая очередь или многочисленные наборы ожидания в расчете на замок, и в противном случае отдавайте предпочтение внутренним очередям условий. (Если вы уже используете замок `ReentrantLock`, потому что вам нужен его дополнительный функционал, то выбор уже сделан.)

[1] Класс `ReentrantLock` требует, чтобы замок удерживался во время вызова методов `signal` или `signalAll`, но реализациям класса `Lock` разрешено конструировать условия `Condition`, которые не имеют этого требования.

14.4. Анатомия синхронизатора

Интерфейсы повторно входимого замка `ReentrantLock` и семафора `Semaphore` имеют много общего. Оба класса выступают в качестве «шлюза», допуская только ограниченное число потоков, которые проходят за раз; потоки прибывают к шлюзу и могут пройти через него (метод `lock` или `acquire` возвращают управление), ожидать (метод `lock` или `acquire` блокируются) и получить отказ (метод `tryLock` или `tryAcquire` возвращает ложь, говоря о том, что замок или разрешение не появляются в допускаемое время). Кроме того, оба допускают прерываемые, непрерываемые и хронометрированные попытки приобретения и оба допускают выбор справедливой или несправедливой очередности ожидающих потоков.

Учитывая эту общность, вы можете подумать, что `Semaphore` был реализован поверх `ReentrantLock`, или, возможно, `ReentrantLock` был реализован, как `Semaphore`, с одним разрешением. Это было бы вполне практичным; доказать, что счетный семафор может быть реализован с использованием замка (как в классе `SemaphoreOnLock` в листинге 14.12) и что замок может быть реализован с использованием счетного семафора, является обычным упражнением.

Листинг 14.12. Счетный семафор, реализованный с использованием замка Lock

```
// Не совсем то, как реализован java.util.concurrent.Semaphore
@ThreadSafe
public class SemaphoreOnLock {
    private final Lock lock = new ReentrantLock();
    // УСЛОВНЫЙ ПРЕДИКАТ: permitsAvailable (permits > 0)
    private final Condition permitsAvailable = lock.newCondition();
    @GuardedBy("lock") private int permits;

    SemaphoreOnLock(int initialPermits) {
        lock.lock();
        try {
            permits = initialPermits;
        } finally {
            lock.unlock();
        }
    }

    // БЛОКИРУЕТ ДО ТЕХ ПОР, ПОКА: permitsAvailable
    public void acquire() throws InterruptedException {
```

продолжение ⇗

Листинг 14.12 *(продолжение)*

```
        lock.lock();
        try {
            while (permits <= 0)
                permitsAvailable.await();
            --permits;
        } finally {
            lock.unlock();
        }
    }

    public void release() {
        lock.lock();
        try {
            ++permits;
            permitsAvailable.signal();
        } finally {
            lock.unlock();
        }
    }
}
```

На самом деле они оба реализованы с использованием общего базового класса `abstractQueuedSynchronizer` (AQS) — как и многие другие синхронизаторы. AQS является структурой для построения замков и синхронизаторов, и с его помощью легко и эффективно может быть построен удивительно широкий спектр синхронизаторов. С помощью AQS построены не только `ReentrantLock` и `Semaphore`, но и `CountDownLatch`, `ReentrantReadWriteLock`, `SynchronousQueue`[1] и `FutureTask`.

AQS решает многие детали реализации синхронизатора, такие как FIFO-очередность ожидающих потоков. Отдельные синхронизаторы могут определять гибкие критерии того, должен ли поток проходить или ждать.

Использование базового класса AQS для создания синхронизаторов дает ряд преимуществ. Он не только существенно сокращает усилия по реализации, но и от вас не требуется оплачивать многочисленные точки конфликта, что требовалось бы во время конструирования одного синхронизатора поверх другого. В `SemaphoreOnLock` приобретение разрешения

[1] Java 6 заменяет `SynchronousQueue` на основе AQS на (более масштабируемую) неблокирующую версию.

имеет два места, где он может блокироваться, — один раз в замке, защищающем состояние семафора, а затем еще раз, если разрешение отсутствует. Синхронизаторы, построенные с помощью AQS, имеют только одну точку, где они могли бы блокироваться, сокращая издержки переключения контекста и улучшая пропускную способность. AQS был разработан для масштабируемости, и все синхронизаторы в `java.util.concurrent`, которые построены с помощью AQS, от этого только выиграют.

14.5. AbstractQueuedSynchronizer

Большинство разработчиков, вероятно, никогда не будут использовать AQS напрямую; стандартный набор синхронизаторов охватывает довольно широкий спектр ситуаций. Но видя, как реализованы стандартные синхронизаторы, можно выяснить, как они работают.

Основные операции, которые выполняет синхронизатор на основе AQS, — это несколько вариантов *приобретения* (acquire) и *освобождения* (release). Операция приобретения зависит от состояния и всегда может блокироваться. С замком или семафором смысл данной операции прост: приобрести замок или разрешение — и вызывающему элементу кода может потребоваться подождать до тех пор, пока синхронизатор не окажется в состоянии, в котором это может произойти. С защелкой `CountDownLatch` операция приобретения означает «ждать до тех пор, пока защелка не достигнет своего конечного состояния», а с задачей `FutureTask` эта операция означает «ждать до тех пор, пока задача не будет завершена». Операция освобождения не является блокирующей; она может позволить потокам, блокированным в операции приобретения, продолжиться.

Для того чтобы класс зависел от состояния, он должен иметь некое состояние. AQS берет на себя задачу управления некоторыми состояниями для класса-синхронизатора: он управляет одним целым числом информации о состоянии, которым можно управлять с помощью защищенных методов `getState`, `setState` и `compareAndSetState`. Он может использоваться для представления произвольного состояния; например `ReentrantLock` использует его для отображения количества раз, когда владеющий поток приобретал замок, `Semaphore` использует его для представления числа оставшихся разрешений и `FutureTask` использует его для представления состояния задачи (еще на начата, работает, завершена, отменена). Синхронизаторы также могут управлять дополнительными переменными

состояния сами; например ReentrantLock отслеживает текущего владельца замка, для того чтобы различать запросы на приобретение повторно входимых и оспариваемых замков.

Приобретение и освобождение в AQS осуществляются в формах, указанных в листинге 14.13. В зависимости от синхронизатора приобретение может быть *эксклюзивным*, как в случае с повторно входимым замком ReentrantLock, либо *неэксклюзивным*, как в случае с семафором Semaphore и защелкой CountDownLatch. Операция приобретения состоит из двух частей. В первой части синхронизатор принимает решение о том, разрешает или нет текущее состояние выполнить операцию приобретения; если да, то потоки разрешается продолжить, и если нет, то операция приобретения блокируется или выдает сбой. Это решение определяется семантикой синхронизатора; например, приобретение замка может быть успешным, если замок не занят, а приобретение защелки может быть успешным, если защелка находится в конечном состоянии.

Листинг 14.13. Канонические формы приобретения и освобождения в AQS

```
boolean acquire() throws InterruptedException {
    while (state does not permit acquire) {
        if (blocking acquisition requested) {
            enqueue current thread if not already queued
            block current thread
    }
    else
        return failure
    }
    possibly update synchronization state
    dequeue thread if it was queued
    return success
}

void release() {
    update synchronization state
    if (new state may permit a blocked thread to acquire)
        unblock one or more queued threads
}
```

Вторая часть предусматривает, возможно, обновление состояния синхронизатора; один поток, приобретающий синхронизатор, может влиять на возможность его приобретения другими потоками. Например, приобретение замка изменяет состояние замка с «не занят» на «занят», а приоб-

ретение разрешения от семафора `Semaphore` сокращает число оставшихся разрешений. С другой стороны, приобретение защелки одним потоком не влияет на способность других потоков ее приобрести, поэтому приобретение защелки его состояние не изменяет.

Синхронизатор, поддерживающий эксклюзивное приобретение, должен реализовывать защищенные методы `tryAcquire`, `tryRelease` и `isHeldExclusively`, и синхронизаторы, поддерживающие совместное приобретение, должны реализовывать методы `tryAcquireShared` и `tryReleaseShared`. Методы `acquire`, `acquireShared`, `release` и `releaseShared` в AQS вызывают формы `try` этих методов в подклассе синхронизатора с целью установления, может или нет операция продолжиться. Подкласс синхронизатора может использовать методы `getState`, `setState` и `compareAndSetState` для проверки и обновления состояния в соответствии с его семантикой приобретения и освобождения, а также сообщает базовому классу через возвращаемое состояние об успешности попытки приобрести или освободить синхронизатор. Например, возвращение отрицательного значения из `tryAcquireShared` указывает на сбой приобретения; возвращение нулевого значения указывает на то, что синхронизатор был приобретен эксклюзивно, и возвращение положительного значения указывает на то, что синхронизатор был приобретен неэксклюзивно. Методы `tryRelease` и `tryReleaseShared` должны возвращать значение `true`, если освобождение могло разблокировать потоки, пытающиеся приобрести синхронизатор.

Чтобы упростить реализацию замков, поддерживающих очереди условий (например замка `ReentrantLock`), AQS также предоставляет механизм для конструирования условных переменных, ассоциированных с синхронизаторами.

14.5.1. Простая защелка

Класс `OneShotLatch` в листинге 14.14 представляет собой двоичную защелку, реализованную с помощью AQS. Она имеет два публичных метода, `await` и `signal`, которые соответствуют приобретению и освобождению. Первоначально защелка закрыта; любой поток, вызывающий метод `await`, блокируется до тех пор, пока защелка не будет открыта. Как только защелка становится открытой с помощью вызова метода `signal`, ожидающие потоки освобождаются, и потокам, которые впоследствии прибудут в защелку, будет разрешено продолжить.

Листинг 14.14. Двоичная защелка с использованием AbstractQueuedSynchronizer

```
@ThreadSafe
public class OneShotLatch {
    private final Sync sync = new Sync();

    public void signal() { sync.releaseShared(0); }

    public void await() throws InterruptedException {
        sync.acquireSharedInterruptibly(0);
    }

    private class Sync extends AbstractQueuedSynchronizer {
        protected int tryAcquireShared(int ignored) {
            // Успешно, если защелка открыта (состояние == 1), иначе сбой
            return (getState() == 1) ? 1 : -1;
        }

        protected boolean tryReleaseShared(int ignored) {
            setState(1); // Защелка теперь открыта
            return true; // Другие потоки теперь могут приобретать
        }
    }
}
```

В классе `OneShotLatch` состояние AQS содержит состояние защелки — закрыта (ноль) или открыта (единица). Метод `await` вызывает метод `acquireSharedInterruptibly` в AQS, который, в свою очередь, консультируется с методом `tryAcquireShared` в защелке `OneShotLatch`. Реализация метода `tryAcquireShared` должна возвращать значение, указывающее на то, можно или нет продолжить приобретение. Если защелка была ранее открыта, то `tryAcquireShared` возвращает успех, позволяя потоку пройти; в противном случае он возвращает значение, указывающее на то, что попытка приобретения оказалась безуспешной. Метод `acquireSharedInterruptibly` интерпретирует ошибку как означающую, что поток должен быть помещен в очередь ожидающих потоков. Схожим образом метод `signal` вызывает `releaseShared`, который вызывает метод `tryReleaseShared` для консультации. Реализация метода `tryReleaseShared` безусловно устанавливает состояние защелки открытым и указывает (через свое возвращаемое значение) на то, что синхронизатор находится в полностью освобожденном состоянии. Это заставляет AQS позволить всем ожидающим потокам попытаться повторно запросить синхронизатор,

и приобретение теперь будет успешно, потому что метод `tryAcquireShared` возвращает успех.

Класс `OneShotLatch` — это полнофункциональный, пригодный для использования, производительный синхронизатор, реализованный всего в двадцати строках кода или около того. Конечно, в нем отсутствует некоторый полезный функционал — такой как хронометрированное приобретение или возможность проинспектировать состояние защелки, — но их также легко реализовать, поскольку AQS предоставляет хронометрированные версии методов приобретения и вспомогательных методов для общих инспекционных операций.

Класс `OneShotLatch` можно было бы реализовать путем расширения AQS, а не делегирования ему, но это нежелательно по нескольким причинам [EJ пункт 14]. Это подорвет простой (двухметодный) интерфейс `OneShotLatch`, и хотя открытые методы AQS не позволят вызывающим элементам кода нарушить состояние защелки, вызывающие элементы кода могут легко использовать их неправильно. Ни один из синхронизаторов в `java.util.concurrent` не расширяет AQS напрямую — вместо этого все они делегируют частные внутренние подклассы абстрактного синхронизатора AQS.

14.6. AQS в классах синхронизатора библиотеки java.util.concurrent

Многие из блокирующих классов в `java.util.concurrent`, такие как `ReentrantLock`, `Semaphore`, `ReentrantReadWriteLock`, `CountDownLatch`, `SynchronousQueue` и `FutureTask`, строятся с помощью AQS. Не вдаваясь слишком глубоко в детали (исходный код является частью скачиваемого JDK[1]), давайте кратко рассмотрим, как каждый из этих классов использует AQS.

14.6.1. ReentrantLock

Класс `ReentrantLock` поддерживает только эксклюзивные приобретения, поэтому он реализует методы `tryAcquire`, `tryRelease` и `isHeldExclusively`; метод `tryAcquire` для несправедливой версии показан в листинге 14.15.

[1] Либо с меньшими лицензионными ограничениями на http://gee.cs.oswego.edu/dl/concurrency-interest.

Листинг 14.15. Реализация метода tryAcquire из несправедливого замка ReentrantLock

```
protected boolean tryAcquire(int ignored) {
    final Thread current = Thread.currentThread();
    int c = getState();
    if (c == 0) {
        if (compareAndSetState(0, 1)) {
            owner = current;
            return true;
        }
    } else if (current == owner) {
        setState(c+1);
        return true;
    }
    return false;
}
```

Класс `ReentrantLock` использует состояние синхронизации, которое содержит счетчик замковых приобретений и поддерживает переменную `owner`, содержащую идентификатор владеющего потока, который модифицируется только тогда, когда текущий поток только что приобрел замок либо вот-вот собирается его освободить[1]. В методе `tryRelease` он проверяет переменную `owner` с целью обеспечения того, чтобы текущий поток владел замком, перед тем как разрешить операции `unlock` продолжиться; в методе `tryAcquire` эта переменная используется для различения попытки повторно входимого приобретения и оспариваемого приобретения.

Когда поток пытается приобрести замок, метод `tryAcquire` сначала обращается к состоянию замка. Если он не занят, то данный метод пытается обновить состояние замка, тем самым указав, что он занят. Поскольку состояние могло измениться с тех пор, как оно инспектировалось в первый раз несколько инструкций назад, метод `tryAcquire` использует метод `compareAndSetState`, для того чтобы попытаться автоматически обновить состояние, тем самым указав, что замок теперь занят, и подтвердить, что

[1] Поскольку защищенные методы манипулирования состоянием имеют основанную на памяти семантику волатильного чтения и записи и класс `ReentrantLock` осторожен в том, что читает переменную `owner` только после вызова метода `getState` и записывает ее только перед вызовом метода `setState`, класс `ReentrantLock` может выезжать за счет основанной на памяти семантики состояния синхронизации и таким образом избегать дальнейшей синхронизации — см. раздел 16.1.4.

состояние не изменилось с прошлого наблюдения. (См. описание метода `compareAndSet` в разделе 15.3.) Если состояние замка указывает на то, что он уже занят, если текущий поток является владельцем замка, то число приобретений увеличивается; если текущий поток не является владельцем замка, то попытка приобретения завершается безуспешно.

Класс `ReentrantLock` также использует встроенную в AQS поддержку многочисленных условных переменных и наборов ожидания. Метод `Lock.newCondition` возвращает новый экземпляр `ConditionObject`, т. е. внутренний класс AQS.

14.6.2. Semaphore и CountDownLatch

Класс `Semaphore` использует состояние синхронизации AQS для хранения счетчика разрешений, имеющихся в настоящее время. Метод `tryAcquireShared` (см. листинг 14.16) сначала вычисляет число оставшихся разрешений, и если их недостаточно, то возвращает значение, указывающее на то, что приобретение не получилось. Если, похоже, осталось достаточное число разрешений, то он пытается атомарно уменьшить число разрешений с помощью метода `compareAndSetState`. Если это удастся (означая, что число разрешений не изменилось с момента последнего просмотра), то он возвращает значение, указывающее на то, что приобретение завершено успешно. Возвращаемое значение также кодирует возможность успешности других совместных попыток приобретения, и в этом случае *другие* ожидающие потоки также будут разблокированы.

Цикл `while` завершается либо при отсутствии достаточного числа разрешений, либо когда метод `tryAcquireShared` может атомарно обновить счетчик разрешений, для того чтобы отразить факт приобретения. В то время как любой данный вызов метода `compareAndSetState` может быть безуспешным из-за конфликта с другим потоком (см. раздел 15.3), побуждая его повторять попытку, один из этих двух критериев терминации станет истинным в пределах разумного числа повторений. Схожим образом метод `tryReleaseShared` увеличивает счетчик разрешений, потенциально снимая блокировку с ожидающих нитей, и повторяет попытки до тех пор, пока обновление не будет успешным. Возвращаемое из метода `tryReleaseShared` значение указывает на то, могли или нет другие потоки быть разблокированы операцией освобождения.

Класс `CountDownLatch` использует AQS аналогично классу `Semaphore`: состояние синхронизации содержит текущий счетчик. Метод `countDown` вызывает метод `release`, который приводит к уменьшению счетчика, и разблокирует ожидающие потоки, если счетчик достиг нуля; метод `await` вызывает метод `acquire`, который немедленно возвращается, если счетчик достиг нуля, и в противном случае блокируется.

Листинг 14.16. Методы tryAcquireShared и tryReleaseShared из семафора

```
protected int tryAcquireShared(int acquires) {
    while (true) {
        int available = getState();
        int remaining = available - acquires;
        if (remaining < 0
                || compareAndSetState(available, remaining))
            return remaining;
    }
}

protected boolean tryReleaseShared(int releases) {
    while (true) {
        int p = getState();
        if (compareAndSetState(p, p + releases))
            return true;
    }
}
```

14.6.3. FutureTask

На первый взгляд класс `FutureTask` даже не выглядит как синхронизатор. Но метод `Future.get` имеет семантику, очень похожую на защелку, — если произошло какое-либо событие (завершение или отмена задачи, представленной объекту `FutureTask`), то потоки могут продолжить, в противном случае они помещаются в очередь и находятся там до тех пор, пока это событие не произойдет.

Класс `FutureTask` использует состояние синхронизации AQS для хранения статуса задачи — работает, завершена или отменена. Он также поддерживает дополнительные переменные состояния для хранения результата вычисления или вызванного им исключения. Кроме того, он сохраняет ссылку на поток, в котором работает вычисление (если оно в данный

момент находится в состоянии работы), чтобы его можно было прервать, если задача отменяется.

14.6.4. ReentrantReadWriteLock

Интерфейс `ReadWriteLock` предполагает наличие двух замков — замок чтения и замок записи, — но в реализации `ReentrantReadWriteLock` на основе AQS единственный подкласс AQS управляет замковой защитой как чтения, так и записи. `ReentrantReadWriteLock` использует 16 бит состояния для счетчика замка записи, а остальные 16 бит — для счетчика замка чтения. Операции над замком чтения используют методы совместного приобретения и освобождения; операции над замком записи используют методы эксклюзивного приобретения и освобождения.

Внутренне AQS поддерживает очередь ожидающих потоков, отслеживая, запрашивал или нет поток эксклюзивный или совместный доступ. Если при появлении замка в классе `ReentrantReadWriteLock` поток в голове очереди стремился получить доступ для записи, то он его получит, и если поток в голове очереди стремился получить доступ для чтения, то его получат все ожидающие потоки вплоть до первого писателя[1].

Итоги

Если вам нужно реализовать зависимый от состояния класс — такой, методы которого должны блокировать продвижение, если предусловие на основе состояния не соблюдается, — то самой лучшей стратегией, как правило, является надстройка над существующей библиотекой классов, таких как `Semaphore`, `BlockingQueue` или `CountDownLatch`, как в `ValueLatch`

[1] Этот механизм не позволяет выбирать политику предпочтения чтения или предпочтения записи, как это делают некоторые реализации замка чтения-записи. Для этого либо очередь ожидания AQS должна быть чем-то иным, чем очередь FIFO, либо потребуются две очереди. Однако такая жесткая политика упорядоченности редко нужна на практике; если несправедливая версия класса `ReentrantReadWriteLock` не предлагает приемлемой жизнеспособности, то справедливая версия обычно дает удовлетворительную упорядоченность и гарантирует отсутствие голодания читателей и писателей.

на с. 236. Однако иногда существующие библиотечные классы не обеспечивают достаточной основы; в этих случаях вы можете построить свои собственные синхронизаторы с помощью внутренних очередей условий, явных объектов `Condition` или абстрактного синхронизатора `AbstractQueuedSynchronizer`. Внутренние очереди условий тесно связаны с внутренней замковой защитой, так как механизм управления зависимостью от состояний неизбежно связан с механизмом обеспечения непротиворечивости состояний. Схожим образом явные объекты `Condition` тесно связаны с явными замковыми объектами `Lock` и предлагают расширенный набор функциональных средств по сравнению с внутренними очередями условий, включая многочисленные наборы ожиданий в расчете на замок, прерываемые или непрерываемые ожидания по условию, справедливую или несправедливую очередность и ожидание на основе предельного срока.

15 Атомарные переменные и неблокирующая синхронизация

Многие классы в `java.util.concurrent`, такие как `Semaphore` и `ConcurrentLinkedQueue`, обеспечивают лучшую производительность и масштабируемость по сравнению с альтернативами, использующими механизм `synchronized`. В этой главе мы рассмотрим основной источник повышения производительности: атомарные переменные и неблокирующую синхронизацию.

Большая часть недавних исследований в области конкурентных алгоритмов была сосредоточена на *неблокирующих алгоритмах*, которые используют низкоуровневые атомарные машинные инструкции, такие как «*сравнить и обменять*», вместо замков для обеспечения целостности данных при конкурентном доступе. Неблокирующие алгоритмы широко используются в операционных системах и JVM для планирования потоков и процессов, сбора мусора, а также для реализации замков и других конкурентных структур данных.

Неблокирующие алгоритмы значительно сложнее проектировать и реализовывать, чем альтернативы на основе замков, но они могут предложить значительные преимущества по масштабируемости и жизнеспособности. Они координируются на более тонком уровне детализации и могут значительно сократить издержки на планирование, поскольку они не

блокируются, когда многочисленные потоки конфликтуют за одни и те же данные. Кроме того, они невосприимчивы к взаимным блокировкам и другим проблемам жизнеспособности. В замковых алгоритмах другие потоки не способны продвигаться вперед, если поток засыпает или циркулирует в холостом цикле, удерживая замок, тогда как неблокирующие алгоритмы непроницаемы для отдельных поточных сбоев. Начиная с Java 5.0 существует возможность строить эффективные неблокирующие алгоритмы на Java, используя *классы атомарных переменных*, такие как `AtomicInteger` и `AtomicReference`.

Атомарные переменные также могут использоваться как «более качественные волатильные переменные», даже если вы не разрабатываете неблокирующие алгоритмы. Атомарные переменные предлагают ту же семантику памяти, что и волатильные переменные, но с дополнительной поддержкой атомарных обновлений — что делает их идеальными для счетчиков, генераторов последовательностей и сбора статистики, предлагая более высокую масштабируемость, чем замковые альтернативы.

15.1. Недостатки замковой защиты

Координирование доступа к совместному состоянию с помощью непротиворечивого протокола замковой защиты обеспечивает то, что, какой бы поток ни содержал замок, защищающий набор переменных, он имеет эксклюзивный доступ к этим переменным, и что любые изменения, внесенные в эти переменные, видны другим потокам, которые впоследствии приобретают замок.

Современные JVM могут довольно эффективно оптимизировать неоспариваемое замковое приобретение и освобождение, но если многочисленные потоки запрашивают замок в одно и то же время, то JVM обращается за помощью к операционной системе. Если дело доходит до этого момента, то какой-то неудачный поток будет приостановлен и должен быть возобновлен позже[1]. Когда этот поток возобновляется, ему может потребоваться дождаться завершения квантов планирования другими потоками, и только

[1] Умная JVM не обязательно приостанавливает поток, если он конфликтует за замок; она могла бы использовать данные профилирования, для того чтобы адаптивно выбирать между приостановкой и замковой защитой в холостом цикле, основываясь на том, как долго замок удерживался во время предыдущих приобретений.

после этого он будет запланирован. Приостановка и возобновление потока имеет много издержек и обычно влечет за собой длительный перерыв. Для классов на основе замков с операциями с высокой степенью детализации (таких как классы синхронизированных коллекций, где большинство методов содержат только несколько операций), отношение издержек планирования к полезной работе может быть довольно высоким, *когда замок часто оспаривается*.

Волатильные переменные являются более легковесным механизмом синхронизации, чем замковая защита, поскольку они не предусматривают переключений контекста или планирования потоков. Однако волатильные переменные имеют некоторые ограничения по сравнению с замковой защитой: хотя они обеспечивают аналогичные гарантии видимости, их нельзя использовать для конструирования атомарных составных действий. Это означает, что волатильные переменные нельзя использовать, когда одна переменная зависит от другой либо когда новое значение переменной зависит от ее старого значения. Это ограничивает использование волатильных переменных, поскольку они не могут использоваться для надежной реализации общих инструментов, таких как счетчики или мьютексы[1].

Например, в то время как операция приращения (`++i`) может *выглядеть* как атомарная операция, на самом деле она представляет собой три разные операции — доставку текущего значения переменной из памяти, добавление к нему единицы и затем запись обновленного значения обратно. Для того чтобы не потерять обновление, вся операция «прочитать-модифицировать-записать» должна быть атомарной. До сих пор единственный способ сделать это состоял в замковой защите, как в классе `Counter` на с. 97.

Класс `Counter` является потокобезопасным, и при наличии небольшого или никакого конфликта работает просто отлично. Но в условиях конфликта производительность страдает из-за издержек на переключение контекста и задержек планирования. Когда замки удерживаются очень коротко, перевод в спящий режим является жестким наказанием за то, что замок был запрошен не вовремя.

Замок имеет несколько недостатков. Когда поток ожидает замка, он не может делать ничего другого. Если поток, владеющий замком, задерживает-

[1] Использовать семантику механизма `volatile` для конструирования мьютексов и других синхронизаторов теоретически возможно, хотя и совершенно непрактично; см. (Raynal, 1986).

ся (из-за страничной ошибки, задержки планирования и т. п.), то никакой поток, которому нужен этот замок, не может продвигаться вперед. Это может быть серьезной проблемой, если блокированный поток является высокоприоритетным, а поток, владеющий замком, является более низкоприоритетным, — опасность для производительности, известная как *инверсия приоритетов* (priority inversion). Даже при том, что более высокоприоритетный поток должен иметь преимущество, он должен ждать до тех пор, пока замок не будет освобожден, и это фактически понижает его приоритет до уровня более низкоприоритетного потока. Если поток, владеющий замком, постоянно блокирован (из-за бесконечного цикла, взаимной блокировки, активной блокировки или другого сбоя в жизнеспособности), то любые потоки, ожидающие этого замка, никогда не смогут двигаться дальше.

Даже игнорируя эти сбои, замковая защита является просто тяжеловесным механизмом для действий с высокой степенью детализации, таких как приращение счетчика. Было бы неплохо иметь техническое решение с высокой степенью детализации для управления конфликтом между потоками — что-то вроде волатильных переменных, но также предлагающее возможность атомарных обновлений. К счастью, современные процессоры предлагают именно такой механизм.

15.2. Аппаратная поддержка конкурентности

Защита от исключающей блокировки — это *пессимистическое* техническое решение — оно исходит из худшего (если вы не запрете дверь, то ворвутся гремлины и переставят все ваши вещи) и не будет продвигаться дальше до тех пор, пока вы не сможете гарантировать, приобретая соответствующие замки, что другие потоки не будут мешать.

Для операций с высокой степенью детализации существует альтернативный подход, который часто будет эффективнее — *оптимистичный* подход, при котором вы продолжаете обновление, надеясь, что сможете завершить его без вмешательства. Этот подход основан на *обнаружении коллизий* с целью установить наличие вмешательства со стороны других во время обновления, в каковом случае операция дает сбой и возможность повторной попытки. Оптимистический подход подобен старой поговорке: «легче добыть прощение, чем разрешение», где «легче» означает «эффективнее».

Процессоры, спроектированные для многопроцессорной работы, предоставляют специальные инструкции по управлению конкурентным доступом к совместным переменным. Ранние процессоры имели атомарные инструкции «*проверить и установить*», «*доставить и прирастить*» или «*обменять*», достаточные для реализации мьютексов, которые, в свою очередь, могли использоваться для реализации более изощренных конкурентных объектов. Сегодня почти каждый современный процессор имеет некоторую форму атомарных инструкций «*прочитать-изменить-записать*», таких как «*сравнить и обменять*» или «*загрузка-с-пометкой/попытка-записи*» (load-linked and store-conditional)[1]. Операционные системы и JVM используют эти инструкции для реализации замков и конкурентных структур данных, но до Java 5.0 они не были доступны непосредственно классам Java.

15.2.1. Сравнить и обменять

Подход, используемый большинством процессорных архитектур, включая IA_{32} и Sparc, заключается в реализации инструкции «*сравнить и обменять*» (compare-and-swap, CAS). (Другие процессоры, такие как PowerPC, реализуют ту же функциональность с помощью пары инструкций: «*загрузка-с-пометкой/попытка-записи*».) Операция CAS имеет три операнда — ячейка памяти *V*, на которой выполняются операции, ожидаемое старое значение *A* и новое значение *B*. CAS атомарно обновляет *V* до нового значения *B*, но только если значение в *V* соответствует ожидаемому старому значению *A*; в противном случае она ничего не делает. В любом случае она возвращает значение, находящееся в данный момент в *V*. (Вариант «сравнить и установить» возвращает признак успешного выполнения операции.) Инструкция CAS имеет в виду: «Считаю, что *V* должна иметь значение *A*; если это так, то помещу-ка туда *B*, в противном случае не поменяю его, но скажи мне, что я не прав». Операция CAS является оптимистичным методом — она продолжает обновление в надежде на успех и может обнаружить сбой, если другой поток обновил переменную

[1] Пара инструкций «загрузка-с-пометкой/попытка-записи» (load-linked/store-conditional) состоит из двух инструкций: первая загружает значение (load-link), возвращая текущее значение ячейки памяти, тогда как последующая операция условного сохранения (store-conditional) в ту же самую ячейку памяти сохраняет новое значение, только если никакие обновления не произошли в этой ячейке с момента операции загрузки load-link. — *Примеч. пер.*

с момента ее последнего наблюдения. Класс `SimulatedCAS` в листинге 15.1 иллюстрирует эту семантику (но не реализацию или производительность) операции CAS.

Когда многочисленные потоки пытаются обновить одну и ту же переменную одновременно с помощью CAS, одна выигрывает и обновляет значение переменной, а остальные проигрывают. Но проигравшие не наказываются приостановкой, как они могли бы, если бы они не смогли приобрести замок; вместо этого им говорится, что они не выиграли гонку в этот раз, но могут попробовать еще. Поскольку поток, который проигрывает операцию CAS, не блокируется, он может выбрать между тем, хочет ли он повторить попытку, выполнить какое-либо другое действие по восстановлению либо ничего не делать[1]. Эта гибкость исключает много из тех угроз для жизнеспособности, которые ассоциированы с замковой защитой (хотя в необычных случаях может внести риск возникновения *активной блокировки* — см. раздел 10.3.3).

Листинг 15.1. Эмуляция работы CAS

```
@ThreadSafe
public class SimulatedCAS {
    @GuardedBy("this") private int value;

    public synchronized int get() { return value; }

    public synchronized int compareAndSwap(int expectedValue,
                                           int newValue) {
        int oldValue = value;
        if (oldValue == expectedValue)
            value = newValue;
        return oldValue;
    }

    public synchronized boolean compareAndSet(int expectedValue,
                                              int newValue) {
        return (expectedValue
                == compareAndSwap(expectedValue, newValue));
    }
}
```

[1] Бездействие может быть вполне разумным ответом на безуспешность операции CAS; в некоторых неблокирующих алгоритмах, таких как алгоритм связной очереди в разделе 15.4.2, безуспешность CAS означает, что кто-то другой уже сделал работу, которую вы планировали сделать.

Типичная схема использования CAS состоит в том, чтобы сначала прочитать значение из *V*, вывести новое значение *B* из *A*, а затем использовать CAS, для того чтобы атомарно поменять *V* с *A* на *B*, при условии что никакой другой поток не изменил *V* на другое значение в это время. Операция CAS решает проблему реализации атомарных последовательностей операций «прочитать-модифицировать-записать» без замковой защиты, потому что она может обнаруживать вмешательство со стороны других потоков.

15.2.2. Неблокирующий счетчик

Класс `CasCounter` в листинге 15.2 реализует потокобезопасный счетчик с помощью CAS. Операция приращения следует канонической форме — доставить старое значение, преобразовать его в новое значение (добавив единицу) и применить CAS для установки нового значения. Если CAS отказывает, то данная операция немедленно повторяется. Повторная попытка, как правило, является разумной стратегией, хотя в случаях крайнего конфликта может быть желательным подождать или отступить, прежде чем повторять попытку, с целью избежать возникновения активной блокировки.

Класс `CasCounter` не блокирует продвижение, хотя ему может потребоваться повторить попытку несколько раз, если другие потоки обновляют счетчик в одно и то же время[1]. (На практике, если вам нужен только счетчик либо генератор последовательностей, то просто используйте `AtomicInteger` или `AtomicLong`, которые обеспечивают атомарное приращение и другие арифметические методы.)

Листинг 15.2. Неблокирующий счетчик с использованием CAS

```
@ThreadSafe
public class CasCounter {
    private SimulatedCAS value;

    public int getValue() {
        return value.get();
    }

    public int increment() {
        int v;
        do {
```

продолжение ⇗

[1] Теоретически, ему, возможно, пришлось бы повторять произвольно много раз, если бы другие потоки продолжили выигрывать гонку CAS; на практике, такого рода голодание случается редко.

Листинг 15.2 *(продолжение)*

```
            v = value.get();
        }
        while (v != value.compareAndSwap(v, v + 1));
        return v + 1;
    }
}
```

На первый взгляд, счетчик на основе CAS выглядит так, как будто должен работать хуже, чем счетчик на основе замка; он имеет больше операций и более сложный поток управления и зависит от, казалось бы, сложной операции CAS. Но на самом деле счетчики на основе CAS значительно превосходят счетчики на основе замка, если существует даже небольшой конфликт, и часто даже если конфликт отсутствует. Быстрый путь неоспариваемого замкового приобретения обычно требует по крайней мере одной операции CAS плюс другой связанной с замком служебной работой, поэтому больше работы происходит в лучшем случае для счетчика на основе замка, чем в обычном случае для счетчика на основе CAS. Так как операция CAS оказывается успешной большую часть времени (если исходить из конфликта от низкого до умеренного), оборудование будет правильно предсказывать ветвь, скрытую в цикле `while`, минимизируя издержки более сложной логики управления.

Языковой синтаксис для замковой защиты может быть компактным, но работа, выполняемая JVM и ОС для управления замками, таковой не является. Замковая защита влечет за собой прохождение относительно сложной ветви кода в JVM и может повлечь замковую защиту на уровне ОС, приостановку потока и переключение контекста. В лучшем случае замковая защита требует по крайней мере одной операции CAS, поэтому использование замков выводит CAS из поля зрения, но не экономит на фактической стоимости выполнения. С другой стороны, выполнение CAS изнутри программы не задействует код JVM, системные вызовы или действия по планированию. То, что на уровне приложения выглядит как более длинная ветвь кода, на самом деле является гораздо более короткой ветвью кода, когда учтена деятельность JVM и ОС. Основным недостатком операции CAS является то, что она вынуждает вызывающий элемент кода конфликтовать (путем выполнения повторных попыток, отступления или сдачи), в то время как замки справляются с конфликтом автоматически, блокируясь, до тех пор пока замок не станет доступным[1].

[1] На самом деле самым большим недостатком инструкции CAS является сложность правильного конструирования окружающих алгоритмов.

Производительность операции CAS широко варьируется между процессорами. В однопроцессорной системе операция CAS, как правило, принимает порядок нескольких тактовых циклов, так как синхронизации между процессорами не требуется. На момент написания этих строк стоимость неоспариваемой операции CAS на многопроцессорных системах находится в диапазоне от десяти до примерно 150 циклов; производительность операции CAS представляет собой быстро движущуюся цель и варьируется не только от архитектуры к архитектуре, но даже между версиями одного и того же процессора. Соперничающие силы, вероятно, приведут к дальнейшему повышению эффективности CAS в течение следующих нескольких лет. Хорошим эмпирическим правилом является то, что стоимость «быстрой ветви» для *неоспариваемого* замкового приобретения и освобождения на большинстве процессоров примерно в два раза превышает стоимость операции CAS.

15.2.3. Поддержка операции CAS в JVM

Так каким же образом код Java убеждает процессор выполнять операцию CAS от своего имени? Начиная с Java 5.0 в язык добавлена низкоуровневая поддержка по предоставлению операций CAS на `int`, `long` и объектных ссылках, и JVM компилирует их в наиболее эффективные средства, предоставляемые базовым оборудованием. На платформах, поддерживающих CAS, рабочая среда интегрирует их в соответствующие машинные инструкции; в худшем случае, если CAS-подобная инструкция отсутствует, то JVM использует замок на основе холостого цикла (`spin lock`). Такая низкоуровневая поддержка со стороны JVM используется классами атомарных переменных (`AtomicXxx` в `java.util.concurrent.atomic`) с целью обеспечить эффективную операцию CAS на числовых и ссылочных типах; эти классы атомарных переменных используются прямо или косвенно с целью реализации большинства классов в `java.util.concurrent`.

15.3. Классы атомарных переменных

Атомарные переменные имеют более высокую степень детализации и являются более легкими, чем замки, а также имеют решающее значение для реализации высокопроизводительного конкурентного кода в многопроцессорных системах. Атомарные переменные ограничивают область

конфликта одной переменной; это максимально возможная степень детализации (если исходить из того, что ваш алгоритм вообще может быть даже реализован с использованием такой тонкой детализации). Быстрая (неоспариваемая) ветвь обновления атомарной переменной не медленнее, чем быстрая ветвь приобретения замка, и обычно быстрее; медленная ветвь определенно быстрее, чем медленная ветвь для замков, поскольку она не включает приостановку и перепланирование потоков. С алгоритмами, основанными на атомарных переменных вместо замков, потоки с большей вероятностью смогут продолжить работу без задержки и легче восстанавливаться, если есть конфликт.

Классы атомарных переменных обеспечивают обобщение волатильных переменных для поддержки атомарных условных операций «прочитать-изменить-записать». Класс `AtomicInteger` представляет значение `int` и обеспечивает методы `get` и `set` с той же семантикой памяти, что и чтения, и записи в волатильное значение `int`. Он также предоставляет атомарный метод `compareAndSet` (который в случае успеха имеет эффекты памяти и чтения и записи волатильной переменной) и, для удобства, атомарные методы `add`, `increment` и `decrement`. Класс `AtomicInteger` имеет поверхностное сходство с расширенным классом `Counter`, но предлагает гораздо большую масштабируемость в условиях конфликта, поскольку он может напрямую использовать базовую аппаратную поддержку конкурентности.

Существует двенадцать классов атомарных переменных, разделенных на четыре группы: скаляры, обновители полей, массивы и составные переменные. Наиболее часто используемыми атомарными переменными являются скаляры: `AtomicInteger`, `AtomicLong`, `AtomicBoolean` и `AtomicReference`. Все они поддерживают операцию CAS; версии `Integer` и `Long` также поддерживают арифметику. (Для симулирования атомарных переменных других примитивных типов вы можете выполнять приведение типа `short` или `byte` в `int` и из `int`, а также использовать методы `floatToIntBits` или `doubleToLongBits` для чисел с плавающей запятой.)

Классы атомарных массивов (доступные в целочисленных `Integer`, длинных `Long` и ссылочных `Reference` версиях) являются массивами, элементы которых могут обновляться атомарно. Классы атомарных массивов предоставляют семантику волатильного доступа к элементам массива, функционал, не доступный для обычных массивов: волатильный (`volatile`) массив имеет волатильную семантику только для ссылки на массив, а не

для своих элементов. (Другие типы атомарных переменных рассматриваются в разделах 15.4.3 и 15.4.4.)

Хотя атомарные скалярные классы расширяют `Number`, они не расширяют примитивные оберточные классы, такие как `Integer` или `Long`. По сути дела, они на это не способны: примитивные оберточные классы не мутируемы, тогда как классы атомарных переменных мутируемы. Классы атомарных переменных также не переопределяют методы `hashCode` или `equals`; каждый экземпляр является неповторяющимся. Как и большинство мутируемых объектов, они не являются хорошими кандидатами для ключей в хеш-коллекциях.

15.3.1. Атомарные компоненты в качестве «более качественных волатильных»

В разделе 3.4.2 мы использовали волатильную (`volatile`) ссылку на немутируемый объект для атомарного обновления многочисленных переменных состояния. Тот пример полагался на операцию «проверить и затем действовать», но в том конкретном случае гонка была безвредной, потому что нам было все равно, будем ли мы иногда терять обновление или нет. В большинстве других ситуаций такая проверка не будет безвредной и может поставить под угрозу целостность данных. Например, класс `NumberRange` на с. 110 не может быть безопасно реализован ни с волатильной ссылкой на немутируемый владеющий объект для верхней и нижней границ, ни с использованием атомарных целых чисел для хранения границ. Поскольку инвариант ограничивает два числа и они не могут обновляться одновременно, соблюдая при этом инвариант, класс диапазона чисел, использующий волатильные ссылки или многочисленные атомарные целые числа, будет иметь небезопасные последовательности операций «проверить и затем действовать».

Мы можем объединить техническое решение из `OneValueCache` с атомарными ссылками, чтобы прекратить гоночное условие, *атомарно* обновляя ссылку на немутируемый объект, владеющий нижней и верхней границами. Класс `CasNumberRange` в листинге 15.3 применяет `AtomicReference` к `IntPair`, для того чтобы содержать состояние; с помощью `compareAndSet` он может обновлять верхнюю или нижнюю границу без гоночных условий класса `NumberRange`.

Листинг 15.3. Сохранение многопеременных инвариантов с использованием операции CAS

```
public class CasNumberRange {
    @Immutable
    private static class IntPair {
        final int lower; // Инвариант: lower <= upper
        final int upper;
        ...
    }
    private final AtomicReference<IntPair> values =
        new AtomicReference<IntPair>(new IntPair(0, 0));

    public int getLower() { return values.get().lower; }
    public int getUpper() { return values.get().upper; }

    public void setLower(int i) {
        while (true) {
            IntPair oldv = values.get();
            if (i > oldv.upper)
                throw new IllegalArgumentException(
                    "Нельзя установить lower равным " + i + " > upper");
            IntPair newv = new IntPair(i, oldv.upper);
            if (values.compareAndSet(oldv, newv))
                return;
        }
    }
    // аналогично для setUpper
}
```

15.3.2. Сравнение производительности: замки против атомарных переменных

Для того чтобы продемонстрировать различия в масштабируемости между замками и атомарными переменными, мы сконструировали эталонный тест, сравнивающий несколько реализаций генератора псевдослучайных чисел (ГПСЧ). В ГПСЧ следующее «случайное» число является детерминированной функцией предыдущего числа, поэтому ГПСЧ должен помнить предыдущее число как часть своего состояния.

В листингах 15.4 и 15.5 показаны две реализации потокобезопасного ГПСЧ, одна из которых использует класс `ReentrantLock`, а другая — класс

AtomicInteger. Тестовый драйвер многократно вызывает каждый из них; каждая итерация генерирует случайное число (которое доставляет и модифицирует совместное состояние seed), а также выполняет ряд «холостых» итераций, которые работают только с локальными для потока данными. Таким образом симулируются типичные операции, которые включают некоторую порцию работы на совместном состоянии и некоторую порцию работы на локальном для потока состоянии.

На рис. 15.1 и 15.2 показана пропускная способность с низким и умеренным уровнями симулированной работы в каждой итерации. При низком уровне локальных вычислений замок или атомарная переменная испытывает конфликт; с более локальными для потока вычислениями замок или атомарная переменная испытывает конфликта меньше, поскольку каждый поток обращается к ней реже.

Листинг 15.4. Генератор случайных чисел с использованием класса ReentrantLock

```
@ThreadSafe
public class ReentrantLockPseudoRandom extends PseudoRandom {
    private final Lock lock = new ReentrantLock(false);
    private int seed;

    ReentrantLockPseudoRandom(int seed) {
        this.seed = seed;
    }

    public int nextInt(int n) {
        lock.lock();
        try {
            int s = seed;
            seed = calculateNext(s);
            int remainder = s % n;
            return remainder > 0 ? remainder : remainder + n;
        } finally {
            lock.unlock();
        }
    }
}
```

Листинг 15.5. Генератор случайных чисел с использованием класса AtomicInteger

```
@ThreadSafe
public class AtomicPseudoRandom extends PseudoRandom {
    private AtomicInteger seed;

    AtomicPseudoRandom(int seed) {
        this.seed = new AtomicInteger(seed);
    }

    public int nextInt(int n) {
        while (true) {
            int s = seed.get();
            int nextSeed = calculateNext(s);
            if (seed.compareAndSet(s, nextSeed)) {
                int remainder = s % n;
                return remainder > 0 ? remainder : remainder + n;
            }
        }
    }
}
```

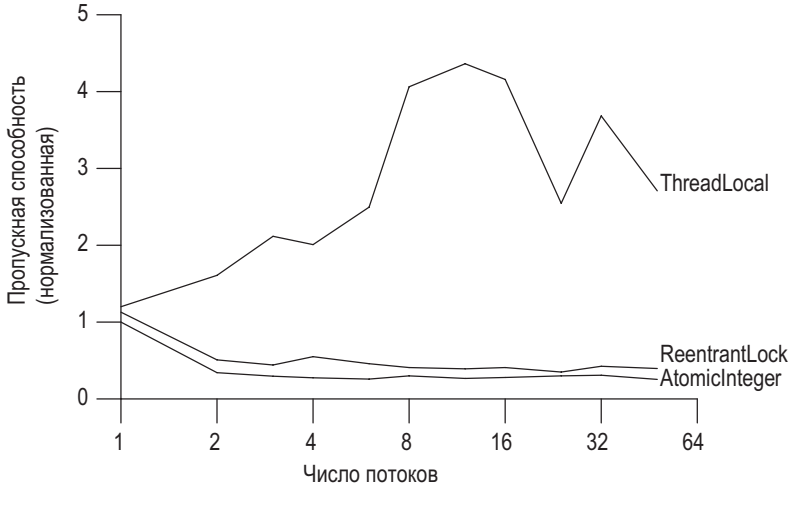

Рис. 15.1. Производительность классов Lock и AtomicInteger в условиях высокого конфликта

Как показывают эти графики, при высоких уровнях конфликта замковая защита имеет тенденцию превосходить атомарные переменные, но на более

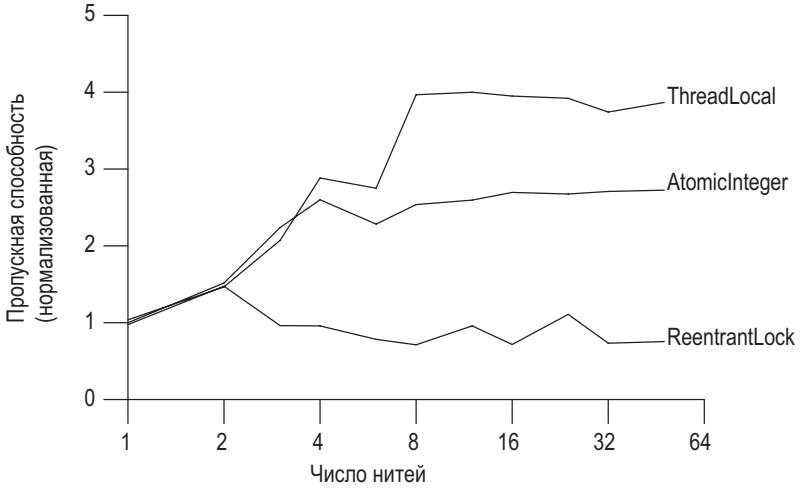

Рис. 15.2. Производительность классов Lock и AtomicInteger в условиях умеренного конфликта

реалистичных уровнях конфликта атомарные переменные превосходят замки[1]. Это обусловлено тем, что замок реагирует на конфликт путем приостановки потоков, сокращая потребление процессора и трафик синхронизации на шине совместно используемой памяти. (Это похоже на то, как блокировка производителей в паттерне «производитель-потребитель» сокращает нагрузку на потребителей и тем самым позволяет им нивелировать отставание.) С другой стороны, при использовании атомарных переменных управление конфликтом передается назад вызывающему классу. Как и большинство алгоритмов на основе CAS, `AtomicPseudoRandom` реагирует на конфликт, немедленно повторяя попытку, и этот подход обычно является правильным, но в высоко оспариваемой среде он просто создает больше конфликта.

[1] То же самое соблюдается и в других областях: светофоры обеспечивают более высокую пропускную способность при высокой интенсивности дорожного движения, а дорожные развязки обеспечивают более высокую пропускную способность при низкой интенсивности дорожного движения; схема конфликта, используемая Ethernet-сетями, показывает более высокие результаты при низком уровне трафика, но схема передачи маркера, используемая кольцевой сетью с маркерным доступом, работает лучше с интенсивным трафиком.

Прежде чем мы заклеймим `AtomicPseudoRandom` как плохо написанный или атомарные переменные как плохой выбор по сравнению с замками, мы должны понять, что уровень конфликта на рис. 15.1 является нереалистично высоким: нет ни одной реальной программы, которая не делает ничего, кроме как оспаривает замок или атомарную переменную. На практике атомарные объекты, как правило, масштабируются лучше, чем замки, потому что атомарные объекты более эффективно справляются с типичными уровнями конфликта.

Разворот производительности между замками и атомарными объектами на разных уровнях конфликта иллюстрирует сильные и слабые стороны каждого из них. С низким до умеренного конфликтом атомарные объекты предлагают более высокую масштабируемость; с высоким конфликтом замки предлагают лучшее предотвращение конфликта. (Алгоритмы на основе CAS также превосходят замковые алгоритмы на однопроцессорных системах, так как на однопроцессорной системе операция CAS всегда успешно работает, за исключением маловероятного случая, когда поток вытесняется в середине операции «прочитать-изменить-записать».)

Рисунки 15.1 и 15.2 включают третью кривую; реализация класса `PseudoRandom`, которая использует `ThreadLocal` для состояния ГПСЧ. Этот подход к реализации изменяет поведение класса — каждый поток видит свою собственную приватную последовательность псевдослучайных чисел, вместо того чтобы все потоки совместно использовали одну последовательность, — но иллюстрирует, что часто дешевле вообще не использовать состояние совместно, если этого можно избежать. Мы можем улучшить масштабируемость, более эффективно справляясь с конфликтом, но истинная масштабируемость достигается только путем полного устранения конфликта.

15.4. Неблокирующие алгоритмы

Замковые алгоритмы подвержены риску возникновения ряда сбоев в жизнеспособности. Если поток, владеющий замком, задерживается из-за блокирующего ввода-вывода, страничной ошибки или другой задержки, то существует возможность, что ни один поток не будет продвигаться вперед. Алгоритм называется *неблокирующим*, если сбой или приостановка какого-либо потока не способен вызвать сбой или приостановку другого потока; алгоритм называется *беззамковым*, если на каждом шаге *какой-либо* поток может продвигаться вперед. Алгоритмы, которые используют

операцию CAS исключительно для координации между потоками, могут, если сконструированы правильно, быть неблокирующими и беззамковыми. Неоспариваемая операция CAS всегда является успешной, и если многочисленные потоки конфликтуют за CAS, то один из них всегда выигрывает и, следовательно, продвигается вперед. Неблокирующие алгоритмы также невосприимчивы к взаимным блокировкам или инверсии приоритетов (хотя они могут демонстрировать голодание или активную блокировку, потому что могут задействовать повторные попытки). До сих пор мы видели только один неблокирующий алгоритм: `CasCounter`. Хорошие неблокирующие алгоритмы известны для многих распространенных структур данных, включая стеки, очереди, очереди с приоритетом и хеш-таблицы, — хотя проектирование новых представляет собой задачу, которую лучше оставить экспертам.

15.4.1. Неблокирующий стек

Неблокирующие алгоритмы значительно сложнее, чем их замковые эквиваленты. Ключом к созданию неблокирующих алгоритмов является выяснение того, как ограничить область атомарных изменений *одной* переменной при сохранении непротиворечивости данных. В классах связных коллекций, таких как очереди, иногда можно обойтись тем, чтобы выражать преобразования состояний в виде изменений отдельных ссылок и использования `AtomicReference` для представления каждой ссылки, которая должна обновляться атомарно.

Стеки представляют собой простейшую связную структуру данных: каждый элемент ссылается только на один элемент, и на каждый элемент ссылается только одна объектная ссылка. Класс `ConcurrentStack` в листинге 15.6 показывает, как конструировать стек с помощью атомарных ссылок. Стек представляет собой связный список узловых элементов `Node` с корнем в вершине, каждый из которых содержит значение и ссылку на следующий элемент. Метод `push` подготавливает новый связный узел, поле `next` которого ссылается на текущую вершину стека, и затем использует операцию CAS, для того чтобы попытаться установить его на вершине стека. Если тот же самый узел все еще находится на вершине стека, как и в момент, когда мы начали, то CAS заканчивается успешно; если верхний узел изменился (потому что другой поток добавлял или удалял элементы, с тех пор как мы начали), то CAS выдает сбой и метод `push` обновляет новый узел, основываясь на текущем состоянии стека, и пытается снова.

Листинг 15.6. Неблокирующий стек с использованием алгоритма Трейбера (Treiber, 1986)

```
@ThreadSafe
public class ConcurrentStack <E> {
    AtomicReference<Node<E>> top = new AtomicReference<Node<E>>();

    public void push(E item) {
        Node<E> newHead = new Node<E>(item);
        Node<E> oldHead;
        do {
            oldHead = top.get();
            newHead.next = oldHead;
        } while (!top.compareAndSet(oldHead, newHead));
    }

    public E pop() {
        Node<E> oldHead;
        Node<E> newHead;
        do {
            oldHead = top.get();
            if (oldHead == null)
                return null;
            newHead = oldHead.next;
        } while (!top.compareAndSet(oldHead, newHead));
        return oldHead.item;
    }

    private static class Node <E> {
        public final E item;
        public Node<E> next;

        public Node(E item) {
            this.item = item;
    }
}
}
```

В любом случае после операции CAS стек по-прежнему находится в непротиворечивом состоянии.

Классы `CasCounter` и `ConcurrentStack` иллюстрируют характеристики всех неблокирующих алгоритмов: некоторая работа делается спекулятивно и, возможно, ее придется переделывать. Когда мы конструируем узел `Node`, представляющий новый элемент, в `ConcurrentStack`, мы надеемся, что значение ссылки `next` будет по-прежнему правильным к тому времени, когда она будет установлена в стеке, но готовы повторить попытку в случае конфликта.

Неблокирующие алгоритмы, такие как `ConcurrentStack`, выводят свою потокобезопасность из того факта, что, как и замковая защита, метод `compareAndSet` предоставляет гарантии атомарности и видимости. Когда поток изменяет состояние стека, он делает это с помощью метода `compareAndSet`, который имеет эффекты памяти, присущие волатильной записи. Когда поток проверяет стек, он делает это, вызывая метод `get`, который имеет эффекты памяти, присущие волатильному чтению, на том же объекте `AtomicReference`. Таким образом, любые изменения, вносимые одним потоком, безопасно публикуются в любом другом потоке, который проверяет состояние списка. И список модифицируется с помощью метода `compareAndSet`, который атомарно обновляет ссылку на вершину либо выдает сбой, если обнаруживает вмешательство со стороны другого потока.

15.4.2. Неблокирующий связный список

Два неблокирующих алгоритма, которые мы видели до сих пор, счетчик и стек, иллюстрируют основной шаблон использования CAS для обновления значения спекулятивно, повторяя попытку, если обновление не удается. Хитрость построения неблокирующих алгоритмов заключается в ограничении области атомарных изменений одной переменной. Со счетчиками это тривиально, и со стеком это достаточно просто, но для более сложных структур данных, таких как очереди, хеш-таблицы или деревья, это может стать намного сложнее.

Связная очередь сложнее, чем стек, поскольку она должна поддерживать быстрый доступ как к голове, так и к хвосту. Для этого она поддерживает отдельные указатели на голову и хвост. Два указателя относят к узлу в хвосте: указатель `next` текущего последнего элемента и указатель на хвост. Для успешной вставки нового элемента оба указателя должны обновляться — атомарно. На первый взгляд это невозможно сделать с атомарными переменными; для обновления двух указателей требуются отдельные операции CAS, и если первая завершается успешно, а вторая завершается безуспешно, то очередь останется в противоречивом состоянии. И даже если обе операции завершатся успешно, другой поток может попытаться обратиться к очереди между первой и второй операциями. Построение неблокирующего алгоритма для связной очереди требует плана для обеих этих ситуаций.

Для того чтобы разработать этот план, нам нужно несколько хитростей. Первая — обеспечить, чтобы структура данных всегда находилась в не-

противоречивом состоянии, даже в середине многошагового обновления. Благодаря этому, если поток *A* находится в середине обновления, когда прибывает поток *B*, *B* может отличить, что операция была частично завершена, и знает, что не стоит пытаться немедленно применять свое собственное обновление. Тогда *B* может ожидать (многократно проверяя состояние очереди), до тех пор пока *A* не закончит, тем самым не мешая друг другу.

Хотя эта хитрость сама по себе была бы достаточной для того, чтобы дать потокам «по очереди» обращаться к структуре данных, не повреждая ее, если бы один поток выдал сбой в середине обновления, то ни один поток не смог бы обратиться к очереди вообще. Для того чтобы сделать этот алгоритм неблокирующим, мы должны обеспечить, чтобы сбой потока не мешал другим потокам продвигаться вперед. Следовательно, вторая хитрость заключается в том, чтобы убедиться, что если поток *B* прибывает, найдя структуру данных в середине обновления потоком *A*, то достаточно информации уже воплощено в структуре данных для *B*, для того чтобы закончить обновление для *A*. Если поток *B* «помогает» потоку *A*, завершая операцию потока *A*, то *B* может продолжить свою собственную операцию, не дожидаясь *A*. Когда *A* приступит к завершению своей операции, он обнаружит, что *B* уже выполнил эту работу для него.

Класс `LinkedQueue` в листинге 15.7 показывает вставочную порцию неблокирующего алгоритма на связной очереди Майкла — Скотта (Michael и Scott, 1996), который используется очередью `ConcurrentLinkedQueue`. Как и во многих очерёдных алгоритмах, пустая очередь состоит из «сторожевого» или «фиктивного» узла, а указатели на голову и хвост инициализируются так, чтобы ссылаться на сторожевой узел. Указатель на хвост всегда ссылается на сторожевой узел (если очередь пуста), последний элемент в очереди либо (в случае, если операция находится в середине обновления) на предпоследний элемент. На рис. 15.3 показана очередь с двумя элементами в нормальном состоянии, или состоянии *покоя*.

Листинг 15.7. Вставка в алгоритм неблокирующей очереди Майкла — Скотта (Michael — Scott, 1996)

```
@ThreadSafe
public class LinkedQueue <E> {
    private static class Node <E> {
        final E item;
        final AtomicReference<Node<E>> next;
```

```
        public Node(E item, Node<E> next) {
            this.item = item;
            this.next = new AtomicReference<Node<E>>(next);
        }
    }

    private final Node<E> dummy = new Node<E>(null, null);
    private final AtomicReference<Node<E>> head
            = new AtomicReference<Node<E>>(dummy);
    private final AtomicReference<Node<E>> tail
            = new AtomicReference<Node<E>>(dummy);

    public boolean put(E item) {
        Node<E> newNode = new Node<E>(item, null);
        while (true) {
            Node<E> curTail = tail.get();
            Node<E> tailNext = curTail.next.get();
            if (curTail == tail.get()) {
                if (tailNext != null) {
                   // Очередь в промежуточном состоянии, продвинуть
                   // хвост вперед
                    tail.compareAndSet(curTail, tailNext);
                } else {
                    // В состоянии покоя, попытаться вставить новый узел
                    if (curTail.next.compareAndSet(null, newNode)) {
                        // Вставка успешна, попытаться продвинуть
                        // хвост вперед
                        tail.compareAndSet(curTail, newNode);
                        return true;
                    }
                }
            }
        }
    }
}
```

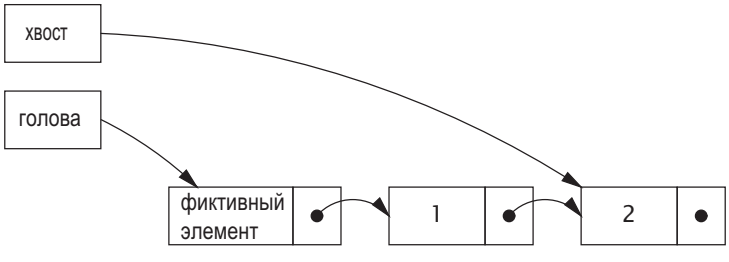

Рис. 15.3. Очередь с двумя элементами в состоянии покоя

Вставка нового элемента предусматривает обновление двух указателей. Первый связывает новый узел с концом списка, обновляя указатель `next` текущего последнего элемента; второй разворачивает указатель хвоста и указывает на новый последний элемент. Между этими двумя операциями очередь находится в *промежуточном* состоянии, показанном на рис. 15.4. После второго обновления очередь снова находится в состоянии покоя, показанном на рис. 15.5.

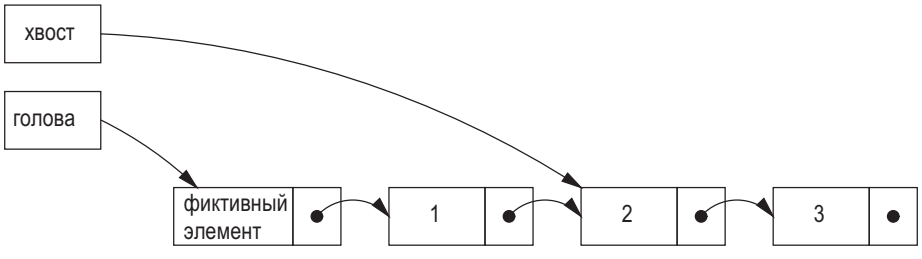

Рис. 15.4. Очередь в промежуточном состоянии во время вставки

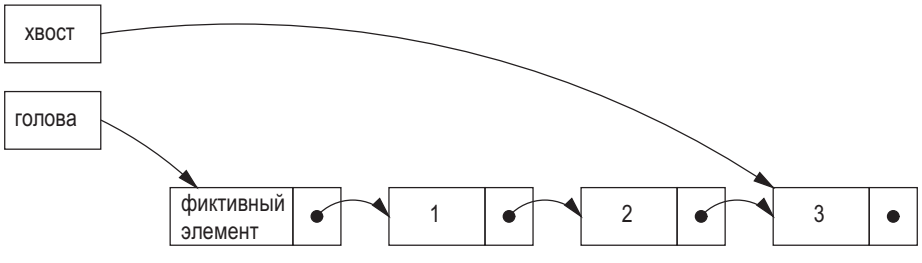

Рис. 15.5. Очередь снова в состоянии покоя после завершения вставки

Ключевое наблюдение, которое делает возможным обе требуемых хитрости, заключается в том, что если очередь находится в состоянии покоя, то поле `next` связного узла, на который указывает `tail`, равно `null`, а если оно находится в промежуточном состоянии, то `tail.next` не равно `null`. Поэтому любой поток может сразу определить состояние очереди, проверив `tail.next`. Более того, если очередь находится в промежуточном состоянии, то ее можно восстановить в состояние покоя, переместив указатель хвоста вперед на один узел, тем самым завершив операцию для любого потока, находящегося в середине вставки элемента[1].

[1] Полный отчет о правильности этого алгоритма см. в (Michael — Scott, 1996) или (Herlihy and Shavit, 2006).

Метод `LinkedQueue.put` сначала проверяет, находится ли очередь в промежуточном состоянии, и только потом пытается вставить новый элемент (шаг *A*). Если это так, то какой-то другой поток уже находится в процессе вставки элемента (между шагами *C* и *D*). Вместо того чтобы ждать, когда поток завершится, текущий поток помогает ему, завершая операцию и продвигая указатель хвоста (шаг *B*). Затем повторяет эту проверку в случае, если другой поток начал вставку нового элемента, продвигая указатель хвоста, до тех пор пока он не найдет очередь в состоянии покоя, для того чтобы можно было начать свою собственную вставку.

Операция CAS на шаге *C*, который связывает новый узел в конце очереди, может завершиться безуспешно, если два потока попытаются вставить элемент в одно и то же время. В этом случае ничего страшного не произойдет: изменения не были внесены, и текущий поток может просто перезагрузить указатель хвоста и повторить попытку. После того как шаг *C* завершен успешно, вставка считается вступившей в силу; вторая операция CAS (шаг *D*) является «очисткой», поскольку может быть выполнена либо вставляющим потоком, либо любым другим потоком. Если шаг *D* оказывается безуспешным, то вставляющий поток возвращается в любом случае, а не повторяет попытку выполнить операцию CAS, потому что никакая повторная попытка не требуется — другой поток уже закончил работу на своем шаге *B*! Это работает потому, что прежде чем какой-либо поток попытается связать новый узел с очередью, сначала он определяет необходимость очереди в очистке, проверяя, не равно ли поле `tail.next` значению `null`. Если не равно, то сначала перемещается указатель хвоста (возможно, многократно), до тех пор пока очередь не окажется в состоянии покоя.

15.4.3. Обновители атомарных полей

В листинге 15.7 показан алгоритм, используемый классом `ConcurrentLinkedQueue`, но его фактическая реализация немного отличается. Вместо представления каждого узла атомарной ссылкой `ConcurrentLinkedQueue` он использует обычную волатильную ссылку и обновляет ее с помощью обновителя `AtomicReferenceFieldUpdater` на основе отражения, как показано в листинге 15.8.

Классы обновителей атомарных полей (имеющиеся в целочисленных `Integer`, длинных `Long` и ссылочных `Reference` версиях) представляют

основанный на отражении «вид» существующего волатильного поля, с тем чтобы CAS можно было использовать на существующих волатильных полях. Классы-обновители не имеют конструкторов; для создания одного такого класса вы вызываете фабричный метод `newUpdater`, указав класс и имя поля. Классы — обновители полей не привязаны к определенному экземпляру; их можно использовать для обновления целевого поля любого экземпляра целевого класса. Гарантии атомарности для классов-обновителей слабее, чем для обычных атомарных классов, потому что нельзя гарантировать, что базовые поля не будут изменены непосредственно — метод `compareAndSet` и арифметические методы гарантируют атомарность только по отношению к другим потокам, использующим методы — обновители атомарных полей.

Листинг 15.8. Использование обновителя атомарного поля в очереди ConcurrentLinkedQueue

```
private class Node<E> {
    private final E item;
    private volatile Node<E> next;

    public Node(E item) {
        this.item = item;
    }
}

private static AtomicReferenceFieldUpdater<Node, Node> nextUpdater
        = AtomicReferenceFieldUpdater.newUpdater(
                Node.class, Node.class, "next");
```

В `ConcurrentLinkedQueue` обновления в поле `next` узла `Node` применяют, используя метод `compareAndSet` из `nextUpdater`. Этот несколько окольный подход используется исключительно по соображениям производительности. Для часто выделяемых, скоротечных объектов, таких как связные узлы очереди, устранение создания ссылки `AtomicReference` для каждого узла `Node` является достаточно значительным, для того чтобы снизить стоимость операций вставки.

Вместе с тем почти во всех ситуациях обычные атомарные переменные работают просто отлично — обновители атомарных полей потребуются только в нескольких случаях. (Обновители атомарных полей также полезны, когда требуется выполнить атомарные обновления с сохранением сериализованной формы существующего класса.)

15.4.4. Проблема ABA

Проблема ABA — это аномалия, которая может возникнуть из-за наивного использования операции «сравнить и обменять» в алгоритмах, где узлы могут быть переработаны (прежде всего в средах без сбора мусора). Операция CAS фактически спрашивает: «значение *V* по-прежнему равно *A*?» и продолжает обновление, если это так. В большинстве ситуаций, включая примеры, приведенные в этой главе, этого вполне достаточно. Но иногда мы хотим спросить: «Изменилось ли значение *V* с тех пор, как я наблюдал его равным *A*?» Для некоторых алгоритмов изменение *V* с *A* на *B*, а затем обратно на *A* по-прежнему считается изменением, которое требует от нас повторения некоторого алгоритмического шага.

Проблема ABA может возникнуть в алгоритмах, которые выполняют свое собственное управление памятью для объектов связных узлов. В этом случае недостаточно, что голова списка по-прежнему ссылается на ранее наблюдаемый узел, для того чтобы из этого следовало, что содержимое списка не изменилось. Если вы не можете избежать проблемы ABA, позволяя сборщику мусора управлять связными узлами за вас, то все еще существует относительно простое решение: вместо обновления значения ссылки обновить *пару* значений, ссылку и номер версии. Даже если значение изменяется с *A* на *B* и обратно на *A*, то номера версий будут отличаться. Класс `AtomicStampedReference` (и его двоюродный брат `AtomicMarkableReference`) обеспечивает атомарное условное обновление на паре переменных. `AtomicStampedReference` обновляет пару «объектная ссылка — целое число», допуская наличие «версионных» ссылок, которые невосприимчивы к проблеме ABA[1]. Схожим образом `AtomicMarkableReference` обновляет пару «объектная ссылка — булево значение», которая используется некоторыми алгоритмами, для того чтобы узел оставался в списке при его маркировке как удаленный[2].

---

[1] Во всяком случае, на практике; теоретически счетчик может циклически возвращаться к началу.

[2] Многие процессоры предоставляют операцию CAS двойной ширины (CAS_2 или CASX), которая может работать на паре «указатель — целое число», что делает эту операцию достаточно эффективной. Начиная с Java 6 `AtomicStampedReference` не использует операцию CAS двойной ширины даже на платформах, которые ее поддерживают. (CAS двойной ширины отличается от DCAS, которая работает на двух не связанных ячейках памяти; на момент написания этих строк ни один текущий процессор не реализует DCAS.)

Итоги

Неблокирующие алгоритмы поддерживают потокобезопасность с помощью низкоуровневых примитивов конкурентности, таких как «сравнить и обменять» вместо замков. Эти низкоуровневые примитивы предоставляются через классы атомарных переменных, которые также могут использоваться как «более качественные волатильные переменные», обеспечивая атомарные операции обновления для целых чисел и объектных ссылок.

Неблокирующие алгоритмы трудны в проектировании и реализации, но они могут предложить более высокую масштабируемость в типичных условиях и более высокую устойчивость к сбоям в жизнеспособности. Многие достижения в конкурентной производительности от одной версии JVM к другой происходят благодаря применению неблокирующих, алгоритмов как в JVM, так и в платформенных библиотеках.

16 Модель памяти Java

На протяжении всей книги мы почти избегали низкоуровневых деталей модели памяти Java (JMM) и вместо этого сосредоточивались на вопросах проектирования более высокого уровня, таких как безопасная публикация, спецификация и соблюдение политик синхронизации. Всем им обеспечивает безопасность JMM, и вам, возможно, будет проще использовать эти механизмы эффективно, когда вы понимаете, *почему* они работают. Эта глава приподнимает занавес и показывает низкоуровневые требования и гарантии модели памяти Java и обоснование некоторых правил проектирования более высокого уровня.

16.1. Что такое модель памяти и зачем она нужна?

Предположим, что один поток назначает значение переменной:

```
aVariable = 3;
```

Модель памяти отвечает на вопрос «при каких условиях поток, который читает переменную `aVariable`, видит значение 3?». Этот вопрос может выглядеть глупым, но в отсутствие синхронизации существует ряд причин, почему поток может сразу — или никогда — не увидеть результаты операции в другом потоке. Компиляторы могут генерировать инструкции

в порядке, отличном от «очевидного», предложенного исходным кодом, либо хранить переменные в регистрах, а не в памяти; процессоры могут выполнять инструкции параллельно или не по порядку; кэши могут варьировать порядок, в котором записи в переменные передаются в главную память; и значения, хранящиеся в локальных для процессора кэшах, могут быть не видны другим процессорам. Эти факторы могут препятствовать тому, чтобы поток видел самое актуальное значение переменной, и могут привести к тому, что действия памяти в других потоках будут внешне происходить не по порядку — если только вы не используете адекватную синхронизацию.

В однопоточной среде все эти трюки, проделываемые средой с нашей программой, скрыты и не имеют никакого предназначения, кроме как ускорить выполнение. Спецификация языка Java требует, чтобы JVM поддерживала *в потоке семантики последовательного выполнения* (*withinthread as-if-serial semantics*): пока программа имеет тот же результат, как если бы она была выполнена в программном порядке в строго последовательной среде, все эти проделки допустимы. И это тоже хорошо, потому что эти перестановки ответственны за значительное улучшение производительности вычислений, произошедшее за последние годы. Конечно, более высокие тактовые частоты способствовали повышению производительности и увеличили параллелизм — конвейерные суперскалярные исполнительные единицы, динамическое планирование инструкций, спекулятивное выполнение и сложные многоуровневые кэши памяти. По мере того как процессоры становились все сложнее, то же самое происходило с компиляторами, переставляющими инструкции для способствования оптимальному выполнению и использующими сложные алгоритмы глобального распределения регистров. И поскольку производители процессоров переходят на многоядерные процессоры, во многом из-за того, что увеличить тактовые частоты становится все труднее экономически, аппаратный параллелизм будет только увеличиваться.

В многопоточной среде иллюзия последовательности не может поддерживаться без значительной стоимости для производительности. Поскольку большую часть времени в конкурентном приложении каждый поток «делает свое дело», чрезмерная координация между потоками только замедлит работу приложения без реальной выгоды. Координировать их действия необходимо, только когда многочисленные потоки совместно используют данные, и JVM полагается на программу, которая идентифицирует время, когда это происходит, используя синхронизацию.

Модель памяти Java (JMM) специфицирует минимальные гарантии, которые JVM должна давать относительно того, когда записи в переменные становятся видимыми для других потоков. Она была спроектирована, чтобы балансировать потребность в предсказуемости и простоте разработки программ с реалиями реализации высокопроизводительных JVM на широком спектре популярных процессорных архитектур. Некоторые аспекты JMM могут поначалу беспокоить, если вы не знакомы с хитростями, используемыми современными процессорами и компиляторами с целью выжать дополнительную производительность из вашей программы.

16.1.1. Платформенные модели памяти

В многопроцессорной архитектуре с совместной памятью каждый процессор имеет собственный кэш, который периодически согласовывается с главной памятью. Процессорные архитектуры обеспечивают варьирующуюся степень *согласованности кэша* (cache coherence); некоторые предоставляют минимальные гарантии, позволяющие разным процессорам видеть разные значения для одной и той же ячейки в памяти практически в любое время. Операционная система, компилятор и рабочая среда (а иногда и программа) должны устранять разницу между тем, что предоставляет оборудование, и тем, что требует потокобезопасность.

Обеспечение того, чтобы каждый процессор знал, что делает другой, всегда стоит дорого. Бóльшую часть времени эта информация не нужна, поэтому процессоры ослабляют свою гарантию слаженности памяти с целью повышения производительности. *Модель памяти* архитектуры сообщает программам, какие гарантии они могут ожидать от системы памяти, и определяет специальные инструкции (именуемые *барьерами памяти*, или *ограждениями*), необходимые для получения дополнительных гарантий координации памяти, требующихся при совместном использовании данных. Для того чтобы защитить разработчика Java от различий между моделями памяти в разных архитектурах, Java предоставляет свою собственную модель памяти, и JVM имеет дело с различиями между JMM и моделью памяти базовой платформы, вставляя барьеры памяти в соответствующих местах.

Одна из удобных ментальных моделей выполнения программ заключается в том, чтобы представить, что существует единый порядок, в котором действия происходят в программе, независимо от того, на каком процессоре они выполняются, и что каждое чтение переменной увидит

последнюю запись в эту переменную в порядке выполнения на любом процессоре. Эта счастливая, хотя и нереалистичная, модель называется *последовательной согласованностью* (sequential consistency). Разработчики программного обеспечения часто ошибочно исходят из последовательной согласованности, но ни один современный мультипроцессор не предлагает последовательную согласованность, и JMM тоже. Классическая модель последовательных вычислений, модель фон Неймана, является лишь смутным приближением того, как ведут себя современные мультипроцессоры.

Подводя черту, современные мультипроцессоры с совместной памятью (и компиляторы) могут делать некоторые удивительные вещи, когда данные совместно используются потоками, если только вы не сказали им не делать этого, применив барьеры памяти. К счастью, программы Java не должны указывать расположение барьеров памяти; им нужно только идентифицировать, когда происходит обращение к совместному состоянию посредством надлежащего использования синхронизации.

16.1.2. Переупорядочивание

При описании гоночных условий и сбоев в атомарности в главе 2 мы использовали диаграммы взаимодействия, изображающие «неудачную временну́ю координацию», когда планировщик перемежал операции, что приводило к неправильным результатам в недостаточно синхронизированных программах. Что еще хуже, JMM может позволить действиям казаться, как будто они выполняются в разных порядках с точки зрения разных потоков, делая рассуждение об упорядоченности в отсутствие синхронизации еще более сложным. Различные причины, почему операции могут быть задержаны или казаться выполняемыми не по порядку, можно сгруппировать в общую категорию *переупорядочивания*.

Класс `PossibleReordering` в листинге 16.1 иллюстрирует то, как трудно рассуждать о поведении даже простейших конкурентных программ, если они не синхронизированы правильно. Довольно легко представить, как `PossibleReordering` может напечатать (1,0) или (0,1) и (1,1): поток мог бы выполниться до конца, прежде чем начнет поток *B*, *B* мог бы выполниться до конца, прежде чем начнется поток *A*, либо их действия могли бы перемежаться. Но, как ни странно, `PossibleReordering` может также напечатать (0,0)! Действия в каждом потоке не имеют зависимости друг от друга по потоку данных и, соответственно, могут быть выполнены не

по порядку. (Даже если они выполняются по порядку, временна́я координация, с помощью которой кэши сбрасываются в главную память, может создать впечатление, с точки зрения *B*, что назначения в *A* происходили в противоположном порядке.) На рис. 16.1 показано возможное перемежение с переупорядочиванием, которое побуждает напечатать (0,0).

Листинг 16.1. Недостаточно синхронизированная программа, которая может иметь удивительные результаты. *Так делать не следует*

```
public class PossibleReordering {
    static int x = 0, y = 0;
    static int a = 0, b = 0;

    public static void main(String[] args)
            throws InterruptedException {
        Thread one = new Thread(new Runnable() {
            public void run() {
                a = 1;
                x = b;
            }
        });
        Thread other = new Thread(new Runnable() {
            public void run() {
                b = 1;
                y = a;
            }
        });
        one.start(); other.start();
        one.join(); other.join();
        System.out.println("( "+ x + "," + y + ")");
    }
}
```

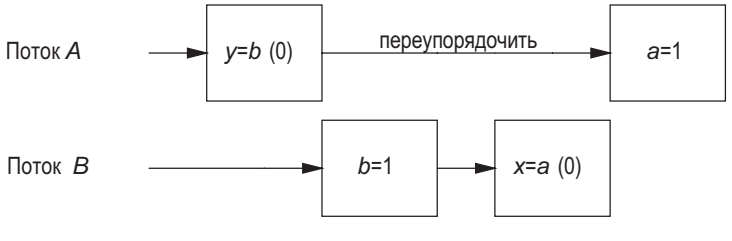

Рис. 16.1. Перемежение, показывающее переупорядочивание в PossibleReordering

Программа `PossibleReordering` — тривиальна, и все же удивительно сложно перечислить ее возможные результаты. Переупорядочивание на уровне памяти может привести к неожиданному поведению программ. В отсутствие синхронизации рассуждать об упорядоченности слишком сложно; гораздо проще обеспечить, чтобы ваша программа использовала синхронизацию надлежащим образом. Синхронизация запрещает компилятору, рабочей среде и оборудованию переупорядочивать операции памяти таким образом, чтобы это нарушало гарантии видимости, предоставляемые JMM-моделью[1].

16.1.3. Модель памяти Java в менее чем 500 словах

Модель памяти Java задана с точки зрения *действий*, которые включают чтения из переменных и записи в переменные, установку и снятие замков на мониторах и запуск и присоединение к потокам. JMM определяет частичное упорядочивание[2], именуемое «*происходит перед*» (happens-before), для всех действий в программе. Для того чтобы гарантировать, что поток, выполняющий действие *B*, может видеть результаты действия *A* (независимо от того, происходят или нет *A* и *B* в разных потоках), между *A* и *B* должно быть отношение «*происходит перед*». В отсутствие упорядоченности «*происходит перед*» между двумя операциями JVM-машина может переупорядочить их по своему усмотрению.

Гонка данных происходит, когда переменная считывается более чем одним потоком и пишется по крайней мере одним потоком, но операции чтения и записи не упорядочены отношением «*происходит перед*». *Правильно синхронизированная программа* — это программа без гонки данных; правильно синхронизированные программы демонстрируют последователь-

---

[1] На самых популярных процессорных архитектурах модель памяти сильна настолько, что стоимость для производительности волатильного чтения вписывается в стоимость неволатильного чтения.

[2] Частичное упорядочивание $\prec$ представляет собой отношение на множестве, которое является антисимметричным, рефлексивным и транзитивным, но для любых двух элементов x и y не обязательно должно иметь место, что $x \prec y$ или $y \prec x$. Мы используем частичные упорядочивания каждый день, для того чтобы выражать предпочтения; возможно, мы и предпочитаем суши чизбургерам, а Моцарта Малеру, но у нас совсем не обязательно имеются четкие предпочтения между чизбургерами и Моцартом.

ную согласованность, имея в виду, что все действия в программе внешне происходят в фиксированном глобальном порядке.

Правила для упорядочивания «*происходит перед*»:

Правило программного порядка. Каждое действие в потоке *происходит перед* каждым действием в нем, которое появляется позже в программном порядке.

Правило мониторного замка. Операция `unlock` на мониторном замке *происходит перед* каждой последующей операцией `lock` на том же самом мониторном замке[1].

Правило волатильной переменной. Запись в волатильное поле *происходит перед* каждым последующим чтением из этого же поля[2].

Правило запуска потока. Вызов `Thread.start` на потоки *происходит перед* каждым действием в запущенном потоке.

Правило терминации потока. Любое действие в потоке *происходит перед* тем, как любой другой поток обнаружит, что поток терминировался, успешно вернувшись из `Thread.join` либо вернув `false` с помощью `Thread.isAlive`.

Правило прерывания. Ситуация, когда поток вызывает прерывание на другом потоке, *происходит перед* тем, как прерванный поток обнаружит прерывание (в результате выдачи исключения `InterruptedException` либо активации `isInterrupted` или `interrupted`).

Правило финализатора. Завершение конструктора объекта *происходит перед* началом финализатора этого объекта.

Транзитивность. Если *A происходит перед B* и *B происходит перед C*, то *A* происходит перед *C*.

Хотя действия упорядочены только частично, действия синхронизации — замковое приобретение и освобождение, и чтение и запись волатильных переменных `volatile` — упорядочены полностью. Это дает смысл описы-

---

[1] Операции `lock` и `unlock` над явными замковыми объектаами `Lock` имеют ту же семантику памяти, что и внутренние замки.

[2] Чтения и записи атомарных переменных имеют ту же семантику памяти, что и волатильные переменные.

вать отношение *происходит перед* в терминах «последующих» замковых приобретений и чтений волатильных переменных.

На рис. 16.2 показано отношение *происходит перед*, когда два потока синхронизуются с помощью общего замка. Все действия внутри потока упорядочены по правилу программного порядка, как и действия в потоке *B*. Поскольку *A* освобождает замок *M* и *B* впоследствии приобретает *M*, все действия в *A* перед освобождением замка, следовательно, упорядочены перед действиями в *B* после приобретения замка. Когда два потока синхронизуются на *разных* замках, мы ничего не можем сказать об упорядоченности действий между ними — между действиями в двух потоках нет отношения *происходит перед*.

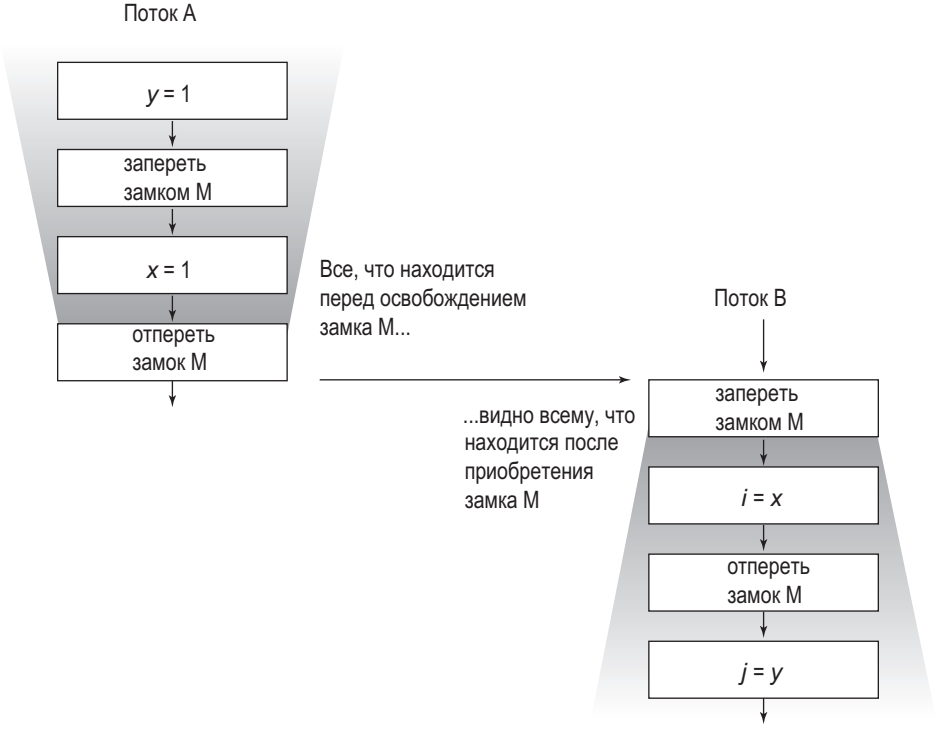

Рис. 16.2. Иллюстрация отношения «происходит перед» в модели памяти Java

16.1.4. Совмещение за счет синхронизации

Из-за прочности упорядочения *происходит перед* вы можете иногда использовать свойства видимости в существующей синхронизации. Это влечет за собой комбинирование правила программного порядка для упорядочивания *происходит перед* с одним из других правил упорядочивания (обычно с правилом мониторного замка или волатильной переменной) с целью упорядочивания обращения к переменной, которая в иных случаях защищена замком. Этот метод очень чувствителен к порядку, в котором происходят инструкции, и поэтому довольно хрупок; данное продвинутое техническое решение следует резервировать для выжимания последней капли производительности из наиболее критичных для производительности классов, таких как `ReentrantLock`.

Реализация защищенных методов абстрактного класса `AbstractQueuedSynchronizer` в `FutureTask` иллюстрирует выезд за счет синхронизации. AQS поддерживает целое число состояния синхронизатора, которое `FutureTask` использует для хранения состояния задачи: работает, завершилась или отменена. Но `FutureTask` также поддерживает дополнительные переменные, такие как результат вычисления. Когда один поток вызывает метод `set`, для того чтобы сохранить результат, и другой поток вызывает метод `get`, для того чтобы его извлечь, обоим лучше быть упорядоченными отношением *происходит перед*. Это можно сделать, создав ссылку на результат `volatile`, однако для того, чтобы достичь того же самого результата за меньшую стоимость, можно задействовать существующую синхронизацию.

Класс `FutureTask` тщательно продуман с целью обеспечения того, чтобы успешный вызов метода `tryReleaseShared` всегда *происходил перед* последующим вызовом метода `tryAcquireShared`; метод `tryReleaseShared` всегда пишет в волатильную переменную, которая читается методом `tryAcquireShared`. В листинге 16.2 показаны методы `innerSet` и `innerGet`, которые вызываются, когда результат сохраняется и извлекается; поскольку `innerSet` пишет `result` перед вызовом метода `releaseShared` (который вызывает `tryReleaseShared`) и `innerGet` читает `result` после вызова метода `acquireShared` (который вызывает `tryAcquireShared`), правило программного порядка комбинируется с правилом волатильной переменной с целью обеспечения того, чтобы запись результата `result` в `innerGet` происходила перед чтением результата `result` в `innerGet`.

Листинг 16.2. Внутренний класс FutureTask, иллюстрирующий двустороннюю синхронизацию

```
// Внутренний для FutureTask класс
private final class Sync extends AbstractQueuedSynchronizer {
    private static final int RUNNING = 1, RAN = 2, CANCELLED = 4;
    private V result;
    private Exception exception;

    void innerSet(V v) {
        while (true) {
            int s = getState();
            if (ranOrCancelled(s))
                return;
            if (compareAndSetState(s, RAN))
                break;
        }
        result = v;
        releaseShared(0);
        done();
    }

    V innerGet() throws InterruptedException, ExecutionException {
        acquireSharedInterruptibly(0);
        if (getState() == CANCELLED)
            throw new CancellationException();
        if (exception != null)
            throw new ExecutionException(exception);
        return result;
    }
}
```

Мы называем это техническое решение «совмещением за счет синхронизации», потому что оно использует существующую упорядоченность *происходит перед*, которая была создана по какой-то другой причине с целью обеспечения видимости объекта *X*, вместо создания упорядоченности *происходит перед* специально для публикации *X*.

Такого рода выезд, эксплуатируемый классом `FutureTask`, является довольно хрупким и должен предприниматься с осторожностью. Тем не менее, в некоторых случаях выезд за счет синхронизации является вполне

разумным, например когда класс придерживается упорядоченности *происходит перед* между методами в рамках своей спецификации. Например, безопасная публикация с использованием `BlockingQueue` является формой такого выезда. Один поток, размещающий объект в очередь, а другой поток, впоследствии извлекающий его, представляют собой безопасную публикацию, поскольку в реализации `BlockingQueue` гарантированно имеется достаточно внутренней синхронизации, которая обеспечивает, чтобы размещение в очередь *происходило перед* удалением из очереди.

Другие упорядоченности *происходит перед*, гарантированные библиотекой классов, включают в себя следующее:

- Размещение элемента в потокобезопасной коллекции происходит *перед тем*, как другой поток извлекает этот элемент из коллекции.
- Обратный отсчет на защелке `CountDownLatch` происходит *перед тем*, как поток возвращается из `await` на этой защелке.
- Освобождение разрешения назад семафору `Semaphore` *происходит перед* получением разрешения от того же семафора.
- Действия, предпринимаемые задачей, представленной `Future`, *происходят перед* тем, как другой поток успешно вернется из метода `Future.get`.
- Предоставление `Runnable` или `Callable` исполнителю `Executor` *происходит перед* началом выполнения задачи.
- Прибытие потока к `CyclicBarrier` или `Exchanger` *происходит перед* тем, как другие потоки освобождаются из того же барьера или точки обмена. Если `CyclicBarrier` использует барьерное действие, то прибытие к барьеру происходит перед барьерным действием, которое в свою очередь происходит перед тем, как потоки освобождаются из барьера.

16.2. Публикация

В главе 3 рассматриваются варианты того, как объект может безопасно или нет публиковаться. Описанные там технические решения безопасной публикации основаны на гарантиях, предоставляемых JMM-моделью; риски ненадлежащей публикации являются последствиями отсутствия упорядоченности *происходит перед* между публикацией совместного объекта и обращением к нему из другого потока.

16.2.1. Небезопасная публикация

Возможность переупорядочивания при отсутствии отношения *происходит перед* объясняет, почему публикация объекта без надлежащей синхронизации может позволить другому потоку увидеть *частично сконструированный объект* (см. раздел 3.5). Инициализация нового объекта предусматривает запись в переменные — поля нового объекта. Схожим образом публикация ссылки предусматривает запись в еще одну переменную — ссылку на новый объект.

Если вы не обеспечиваете того, чтобы публикация совместной ссылки *происходила перед* тем, как другой поток загрузит эту совместную ссылку, то запись ссылки в новый объект может быть переупорядочена (с точки зрения потока, потребляющего объект) с записями в его поля. В этом случае другой поток мог бы видеть актуальное значение ссылки на объект, но *устаревшие значения некоторых или всех состояний этого объекта* — частично сконструированный объект.

Небезопасная публикация может произойти в результате неправильной ленивой инициализации, как показано в листинге 16.3. На первый взгляд, единственной проблемой здесь является гоночное условие, описанное в разделе 2.2.2. При определенных обстоятельствах, например когда все экземпляры ресурса `Resource` идентичны, вы, возможно, захотите ими пренебречь (наряду с неэффективностью возможного создания объекта `Resource` более одного раза). К сожалению, даже если этими дефектами пренебречь, класс `UnsafeLazyInitialization` по-прежнему небезопасен, так как другой поток может наблюдать ссылку на частично сконструированный объект `Resource`.

Листинг 16.3. Небезопасная ленивая инициализация. *Так делать не следует*

```
@NotThreadSafe
public class UnsafeLazyInitialization {
    private static Resource resource;

    public static Resource getInstance() {
        if (resource == null)
```

```
            resource = new Resource(); // небезопасная публикация
        return resource;
    }
}
```

Предположим, что поток *A* первый активировал метод `getInstance`. Он видит, что поле `resource` равно `null`, создает новый ресурс, инстанцировав `Resource`, и назначает полю `resource` ссылку на него. Когда поток *B* позже вызывает `getInstance`, он может увидеть, что поле `resource` уже имеет значение не `null` и просто использует уже сконструированный объект `Resource`. Поначалу это может показаться безвредным, но *между записью ресурса в A и чтением ресурса в B отсутствует упорядоченность «происходит перед»*. Для публикации объекта была использована гонка данных, и поэтому *B* гарантированно не увидит правильное состояние объекта `Resource`.

Конструктор `Resource` меняет поля свежевыделенного объекта `Resource` со значений по умолчанию (записываемых конструктором `Object`) на их исходные значения. Поскольку ни один из потоков не использовал синхронизацию, *B* может видеть действия *A* в порядке, отличном от того, в котором их выполнял *A*. Таким образом, даже если *A* перед установкой поля `resource` инициализировал объект *Resource* ссылкой на него, *B* может видеть, что запись в поле `resource` происходила *перед* записями в поля объекта `Resource`. *B*, следовательно, может видеть частично сконструированный объект `Resource`, который вполне может находиться в недопустимом состоянии и состояние которого может неожиданно измениться позже.

> За исключением немутируемых объектов не является небезопасным использовать объект, инициализированный другим потоком, только если публикация не *происходит перед* тем, как потребляющий поток его использует.

16.2.2. Безопасная публикация

Идиомы безопасной публикации, описанные в главе 3, обеспечивают то, чтобы опубликованный объект был виден другим потокам, поскольку они обеспечивают, чтобы публикация *происходила перед* тем, как потребляющий поток загрузит ссылку на опубликованный объект.

Если поток *A* помещает объект *X* в очередь `BlockingQueue` (и никакой поток впоследствии его не модифицирует), и поток *B* извлекает его из очереди, то *B* гарантированно видит *X* в том состоянии, в каком его оставил *A*. Это обусловлено тем, что реализации очереди `BlockingQueue` имеют достаточную внутреннюю синхронизацию для обеспечения того, чтобы операция `put` *происходила перед* операцией `take`. Схожим образом использование совместной переменной, защищенной замком, либо совместной волатильной переменной обеспечивает то, чтобы операции чтения и записи этой переменной были упорядочены отношением «*происходит перед*».

Такая гарантия упорядоченности *происходит перед* фактически представляет собой более сильное обещание видимости и упорядоченности, чем те, которые даются безопасной публикацией. Когда *X* безопасно публикуется из *A* в *B*, безопасная публикация гарантирует видимость состояния *X*, но не состояния других переменных *A*, которые могли быть затронуты. Но если тот факт, что *A* помещает *X* в очередь, *происходит перед* тем, как *B* доставляет *X* из этой очереди, то не только *B* видит *X* в том состоянии, в котором *A* его оставил (при условии что *X* не было впоследствии изменено *A* или кем-то еще), но *B* видит *все*, что *A* делал до эстафетной передачи (опять же, с такой же оговоркой)[1].

Почему мы так сильно сосредоточились на `@GuardedBy` и безопасной публикации, когда JMM уже предоставляет нам более мощные средства обеспечения упорядоченности *происходит перед*? Мышление с точки зрения эстафетной передачи владения объектом и публикации лучше вписывается в большинство программных проектов, чем мышление с точки зрения видимости отдельных записей памяти. Упорядоченность *происходит перед* работает на уровне отдельных обращений к памяти — это своего рода «сборочный язык конкурентности». Безопасная публикация же работает на уровне, близком к проекту вашей программы.

16.2.3. Идиомы безопасной инициализации

Иногда имеет смысл отложить инициализацию объектов, чья инициализация обходится дорого, до тех пор пока они действительно не понадобятся,

[1] JMM гарантирует, что *B* видит значение по крайней мере таким же актуальным, как и значение, записанное *A*; последующие записи могут быть или не быть видимыми.

но мы видели, как неправильное использование ленивой инициализации может привести к проблемам. Класс `UnsafeLazylnitialization` может быть исправлен, если сделать метод `getResource` синхронизированным, как показано в листинге 16.4. Поскольку ветвь кода через метод `getInstance` является довольно короткой (тестовая и предсказываемая ветвь), если `geInstance` не вызывается многими потоками часто, то существует достаточно малый конфликт за замок `SafeLazyInitialization`, и поэтому такой подход обеспечивает достаточную производительность.

Работа со статическими полями с помощью инициализаторов (или полей, значение которых инициализируется в блоке статической инициализации [JPL 2.2.1 и 2.5.3]) является несколько особенной и предлагает дополнительные гарантии потокобезопасности. Статические инициализаторы запускаются JVM-машиной во время инициализации класса, после загрузки класса, но перед использованием класса любым потоком. Поскольку JVM приобретает замок во время инициализации [JLS 12.4.2] и этот замок приобретается каждым потоком по крайней мере один раз с целью обеспечения загрузки класса, записи в память, осуществляемые во время статической инициализации, автоматически видны всем потокам. Таким образом, статически инициализированные объекты не требуют явной синхронизации ни во время конструирования, ни во время ссылки на них. Однако это относится только к состоянию «в том виде, в каком он сконструирован» — если объект является мутируемым, то синхронизация по-прежнему требуется и читателю, и писателю, для того чтобы сделать последующие модификации видимыми и избежать повреждения данных.

Листинг 16.4. Потокобезопасная ленивая инициализация

```
@ThreadSafe
public class SafeLazyInitialization {
    private static Resource resource;

    public synchronized static Resource getInstance() {
        if (resource == null)
            resource = new Resource();
        return resource;
    }
}
```

Листинг 16.5. Нетерпеливая инициализация

```
@ThreadSafe
public class EagerInitialization {
    private static Resource resource = new Resource();

    public static Resource getResource() { return resource; }
}
```

Использование упреждающей (`earger`) инициализации, показанной в листинге 16.5, устраняет стоимость синхронизации при каждом вызове метода `getInstance` в `SafeLazyInitialization`. Это техническое решение может быть объединено с ленивой загрузкой класса JVM-машиной с целью создания технического решения ленивой инициализации, которое не требует синхронизации на общей ветви кода. Идиома *держателя с ленивой инициализацией* [EJ пункт 48] в листинге 16.6 использует класс, единственной целью которого является инициализация объекта `Resource`. JVM откладывает инициализацию класса `ResourceHolder` до его фактического использования [JLS 12.4.1], и поскольку `Resource` инициализируется статическим инициализатором, дополнительная синхронизация не требуется. Первый вызов метода `getResource` любым потоком приводит к загрузке и инициализации класса `ResourceHolder`, и в это самое время инициализация объекта `Resource` происходит через статический инициализатор.

Листинг 16.6. Идиома класса-держателя с ленивой инициализацией

```
@ThreadSafe
public class ResourceFactory {
    private static class ResourceHolder {
        public static Resource resource = new Resource();
    }

    public static Resource getResource() {
        return ResourceHolder.resource;
    }
}
```

16.2.4. Блокировка с двойной проверкой

Ни одна книга по конкурентности не будет полной без обсуждения печально известного антипаттерна блокировки с двойной проверкой (double-checked locking, DCL), показанного в листинге 16.7. В очень ранних JVM

синхронизация, даже неоспариваемая, имела значительную стоимость для производительности. В результате было изобретено много умных (или, по крайней мере, похожих на такие) хитростей с целью уменьшить влияние синхронизации — некоторые хорошие, некоторые плохие и некоторые уродливые. DCL попадает в категорию «уродливых».

Листинг 16.7. Антипаттерн блокировки с двойной проверкой. *Так делать не следует*

```
@NotThreadSafe
public class DoubleCheckedLocking {
    private static Resource resource;

    public static Resource getInstance() {
        if (resource == null) {
            synchronized (DoubleCheckedLocking.class) {
                if (resource == null)
                    resource = new Resource();
            }
        }
        return resource;
    }
}
```

Опять же, поскольку производительность ранних JVM оставляла желать лучшего, ленивая инициализация часто использовалась для того, чтобы избежать потенциально ненужных дорогостоящих операций либо сократить время запуска приложения. Надлежаще написанный метод ленивой инициализации требует синхронизации. Но в то время синхронизация была медленной и, что более важно, не совсем понятной: аспекты исключения были достаточно хорошо поняты, но аспекты видимости — нет.

DCL претендовал на то, чтобы предложить лучшее из обоих миров — ленивую инициализацию без уплаты штрафа за синхронизацию на общей ветви кода. Блокировка с двойной проверкой работала следующим образом. Сначала нужно было проверить необходимость инициализации без синхронизации, и, если ссылка на `resource` не равна `null`, использовать ее. В противном случае синхронизировать и проверить еще раз факт инициализации объекта `Resource` с целью обеспечения того, чтобы совместный

объект `Resource` фактически инициализировался только одним потоком. Общая ветвь кода — доставка ссылки на уже сконструированный объект `Resource` — не использует синхронизацию. И вот здесь как раз проблема: как описано в разделе 16.2.1, поток может видеть частично сконструированный объект `Resource`.

Реальная проблема с DCL заключается в принятом допущении, что худшее, что может произойти при чтении ссылки на совместный объект без синхронизации, — это ошибочно увидеть устаревшее значение (в данном случае `null`); в этом случае идиома DCL компенсирует этот риск, повторяя попытку с занятым замком. Но наихудший случай на самом деле значительно хуже — можно увидеть текущее значение ссылки, но устаревшие значения состояния объекта, имея в виду, что объект может наблюдаться в недопустимом или неправильном состоянии.

Последующие изменения в JMM (Java 5.0 и позже) позволили DCL работать, *если* `resource` сделан волатильным, и влияние на производительность этого подхода является небольшим, так как волатильные чтения обычно только слегка дороже, чем неволатильные.

Полезность этой идиомы в значительной степени сошла на нет — силы, которые ее мотивировали (медленная неоспариваемая синхронизация, медленный запуск JVM), больше не играют роли, делая ее менее эффективной в качестве оптимизации. Идиома держателя ленивой инициализации предлагает те же преимущества и легче для понимания.

16.3. Безопасность инициализации

Гарантия *безопасности инициализации* позволяет надлежаще сконструированным немутируемым объектам безопасно совместно использоваться потоками без синхронизации, независимо от того, как они публикуются — даже если они публикуются с использованием гонки данных. (Это означает, что `UnsafeLazyInitialization` фактически является безопасным, *если* объект `Resource` является немутируемым.)

Без безопасности инициализации предположительно немутируемые объекты, такие как `String`, могут внешне изменять свое значение, если синхронизация не используется как публикующими, так и потребляющими потоками. Архитектура безопасности опирается на немутируемость

объекта `String`; отсутствие безопасности инициализации может создать уязвимости в безопасности, которые позволяют вредоносному коду обходить проверки безопасности.

> Безопасность инициализации гарантирует, что в случае *надлежаще сконструированных* объектов все потоки будут видеть правильные значения финальных полей, которые были заданы конструктором, независимо от способа публикации объекта. Кроме того, любые переменные, *достижимые* через финальное поле надлежаще сконструированного объекта (например, элементы финального массива или содержимое хеш-массива `HashMap`, на которое ссылается финальное поле), также гарантированно будут видны другим потокам[1].

Для объектов с финальными полями безопасность инициализации запрещает переупорядочивание любой части конструкции с начальной загрузкой ссылки на этот объект. Все записи в финальные поля, осуществляемые конструктором, а также в любые переменные, достижимые через эти поля, становятся «замороженными», когда конструктор завершает свою работу, и любой поток, который получает ссылку на этот объект, гарантированно видит значение, которое так же актуально, как и замороженное значение. Записи, которые инициализируют переменные, достижимые через финальные поля, не переупорядочиваются операциями, следующими за постконструкционной заморозкой.

Безопасность инициализации означает, что класс `SafeStates` в листинге 16.8 можно безопасно публиковать даже с помощью небезопасной ленивой инициализации или припрятав ссылку на `SafeStates` в публичном статическом поле без синхронизации, даже если он не использует синхронизацию и полагается на непотокобезопасное хеш-множество `HashSet`.

Листинг 16.8. Безопасность инициализации для немутируемых объектов.

```
@ThreadSafe
public class SafeStates {
    private final Map<String, String> states;
```

продолжение ⇗

[1] Это относится лишь к объектам, достижимым только через финальные поля конструируемого объекта.

Листинг 16.8 *(продолжение)*

```
    public SafeStates() {
        states = new HashMap<String, String>();
        states.put("alaska", "AK");
        states.put("alabama", "AL");
        ...
        states.put("wyoming", "WY");
    }

    public String getAbbreviation(String s) {
        return states.get(s);
    }
}
```

Ряд небольших изменений в классе `SafeStates` повредит его потокобезопасности. Если бы поле `states` не было финальным или если бы какой-либо другой метод, кроме конструктора, смодифицировал его содержимое, то безопасность инициализации не была бы достаточно сильной, чтобы безопасно обратиться к `SafeStates` без синхронизации. Если бы класс `SafeStates` имел другие финальные поля, то другие потоки могли бы по-прежнему видеть неправильные значения этих полей. И разрешение объекту ускользнуть во время конструирования делает недействительной гарантию безопасности инициализации.

> Безопасность инициализации гарантирует видимость только тех значений, которые достижимы через финальные поля на момент завершения работы конструктора. В случае значений, достижимых через нефинальные поля, или значений, которые могут измениться после конструирования, вы должны использовать синхронизацию с целью обеспечения видимости.

Итоги

Модель памяти Java определяет время, когда действия одного потока на память гарантированно будут видны другому. Их особенности включают в себя обеспечение того, чтобы операции частично упорядочивались отношением *происходит перед*. Данная упорядоченность задается на уровне отдельных операций с памятью и операций синхронизации. Если достаточная синхронизация отсутствует, то в то время, когда потоки об-

ращаются к совместным данным, могут происходить очень странные вещи. Однако правила более высокого уровня, предлагаемые в главах 2 и 3, такие как `@GuardedBy` и безопасная публикация, могут использоваться с целью обеспечения потокобезопасности, чтобы не прибегать к низкоуровневым деталям упорядочивания *происходит перед*.

Приложение А.
Аннотации для конкурентности

Мы использовали аннотации, такие как `@GuardedBy` и `@ThreadSafe`, для того чтобы показать, как можно документировать обещания потокобезопасности и политики синхронизации. Данное приложение документирует эти аннотации; их исходный код можно загрузить с веб-сайта этой книги. (Разумеется, существуют дополнительные обещания потокобезопасности и детали реализации, которые должны быть задокументированы, но которые не охвачены этим минимальным набором аннотаций.)

А.1. Аннотации классов

Для описания обещанного уровня потокобезопасности класса мы используем три аннотации: `©Immutable`, `©ThreadSafe` и `©NotThreadSafe`. `©Immutable` означает, конечно, что класс является немутируемым и из него вытекает `©ThreadSafe`. `©NotThreadSafe` не является обязательным — если класс не аннотируется как потокобезопасный, то следует предположить, что он не является потокобезопасным, но если вы хотите сделать его более понятным, используйте `©NotThreadSafe`.

Эти аннотации относительно ненавязчивы и полезны как для пользователей, так и для сопроводителей. Пользователи могут сразу же увидеть, является ли класс потокобезопасным, а сопроводители могут сразу же выяснить, нужно ли сохранять гарантии потокобезопасности. Аннотации также полезны для третьей группы: инструменты. Инструменты статического анализа кода могут быть в состоянии верифицировать, что код соответствует контракту, обозначенному аннотацией, например, верифицировать, что класс, аннотированный `©Immutable`, фактически является немутируемым.

А.2. Аннотации полей и методов

Приведенные выше аннотации уровня класса являются частью публичной документации для класса. Другие аспекты стратегии потокобезопасности класса предназначены исключительно для сопроводителей и не являются частью его публичной документации.

Классы, использующие замковую защиту, должны документировать, какие переменные состояния защищены замками и какие замки используются для защиты этих переменных. Общий источник неумышленной непотокобезопасности проистекает в ситуациях, когда потокобезопасный класс непротиворечиво использует замковую защиту для защиты своего состояния, но позже модифицируется, для того чтобы добавить новые переменные состояния, которые неадекватно защищены замком, либо новые методы, которые не используют замковую защиту надлежаще для того, чтобы защитить существующие переменные состояния. Документирование того, какие переменные защищены замками, может помочь предотвратить оба типа упущений.

`@GuardedBy(lock)` документирует, что к полю или методу следует обращаться только с определенным занятым замком. Аргумент `lock` идентифицирует замок, которым следует владеть во время обращения к аннотированному полю или методу. Ниже приведены возможные значения для аргумента `lock`:

- `@GuardedBy("this")` имеет в виду наложение внутреннего замка на содержащий объект (объект, членом которого является метод или поле);
- `@GuardedBy("имяПоля")` имеет в виду замок, ассоциированный с объектом, на который ссылается именованное поле, либо внутренний замок (для полей, которые не ссылаются на `Lock`), либо явный замковый объект `Lock` (для полей, которые ссылаются на `Lock`);
- `@GuardedBy("ИмяКласса.имяПоля")`, например `@GuardedBy("имяПоля")`, но ссылающийся на замковый объект, находящийся в статическом поле другого класса;
- `@GuardedBy("имяМетода")` имеет в виду замковый объект, возвращаемый при вызове именованного метода;
- `@GuardedBy("ИмяКласса.класс")` имеет в виду буквальный объект класса для именованного класса.

Использование `@GuardedBy` для идентификации каждой переменной состояния, которая нуждается в замковой защите, и того, какой замок его защищает, может помочь в сопровождении и ревизии кода, а также помочь автоматизированным инструментам анализа выявлять потенциальные ошибки потокобезопасности.

Библиография

Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language*, Fourth Edition. Addison–Wesley, 2005.

David F. Bacon, Ravi B. Konuru, ChetMurthy, andMauricio J. Serrano. Thin Locks: Featherweight Synchronization for Java. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 258–268, 1998. URL http://citeseer.ist.psu.edu/bacon98thin.html.

Joshua Bloch. *Effective Java Programming Language Guide*. Addison–Wesley, 2001.

Joshua Bloch and Neal Gafter. *Java Puzzlers*. Addison–Wesley, 2005.

Hans Boehm. Destructors, Finalizers, and Synchronization. In *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 262–272. ACM Press, 2003. URL http://doi.acm.org/10. 1145/604131.604153.

Hans Boehm. Finalization, Threads, and the Java Memory Model. JavaOne presentation, 2005. URL http://developers.sun.com/learning/javaoneonline/2005/coreplatform/TS-3281.pdf.

Joseph Bowbeer. The Last Word in Swing Threads, 2005. URL http://java.sun.com/products/jfc/tsc/articles/threads/threads3.html.

Cliff Click. Performance Myths Exposed. JavaOne presentation, 2003.

Cliff Click. Performance Myths Revisited. JavaOne presentation, 2005. URL http://developers.sun.com/learning/javaoneonline/2005/coreplatform/ TS-3268.pdf.

Martin Fowler. Presentation Model, 2005. URL http://www.martinfowler.com/eaaDev/PresentationModel.html.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison–Wesley, 1995.

Martin Gardner. The fantastic combinations of John Conway's new solitaire game 'Life'. *Scientific American*, October 1970.

James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*, Third Edition. Addison–Wesley, 2005.

Tim Harris and Keir Fraser. Language Support for Lightweight Transactions. In *OOPSLA '03: Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 388–402. ACM Press, 2003. URL http://doi.acm.org/10.1145/949305.949340.

Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable Memory Transactions. In *PPoPP '05: Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 48–60. ACM Press, 2005. URL http://doi.acm.org/10.1145/1065944.1065952.

Maurice Herlihy. Wait-Free Synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, 1991. URL http://doi.acm.org/10.1145/114005.102808.

Maurice Herlihy and Nir Shavit. *Multiprocessor Synchronization and Concurrent Data Structures*. Morgan-Kaufman, 2006.

C. A. R. Hoare. Monitors: An Operating System Structuring Concept. *Communications of the ACM*, 17(10):549–557, 1974. URL http://doi.acm.org/10.1145/355620.361161.

David Hovemeyer and William Pugh. Finding Bugs is Easy. *SIGPLAN Notices*, 39 (12):92–106, 2004. URL http://doi.acm.org/10.1145/1052883.1052895.

Ramnivas Laddad. *AspectJ in Action*. Manning, 2003.

Doug Lea. *Concurrent Programming in Java*, Second Edition. Addison–Wesley, 2000.

Doug Lea. JSR-133 Cookbook for Compiler Writers. URL http://gee.cs.oswego.edu/dl/jmm/cookbook.html.

J. D. C. Little. A proof of the Queueing Formula $L = \lambda W$». *Operations Research*, 9:383–387, 1961.

Jeremy Manson, William Pugh, and Sarita V. Adve. The Java Memory Model. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 378–391. ACM Press, 2005. URL http://doi.acm.org/10.1145/1040305.1040336.

George Marsaglia. XorShift RNGs. *Journal of Statistical Software*, 8(13), 2003. URL http://www.jstatsoft.org/v08/i14.

Maged M. Michael and Michael L. Scott. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *Symposium on Principles of Distributed Computing*, pages 267–275, 1996. URL http://citeseer.ist.psu.edu/michael96simple.html.

Mark Moir and Nir Shavit. *Concurrent Data Structures*, In *Handbook of Data Structures and Applications*, chapter 47. CRC Press, 2004.

William Pugh and Jeremy Manson. Java Memory Model and Thread Specification, 2004. URL http://www.cs.umd.edu/~pugh/java/memoryModel/jsr133.pdf.

M. Raynal. *Algorithms for Mutual Exclusion*. MIT Press, 1986.

William N. Scherer, Doug Lea, and Michael L. Scott. Scalable Synchronous Queues. In *11th ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming (PPoPP)*, 2006.

R. K. Treiber. Systems Programming: Coping with Parallelism. Technical Report RJ 5118, IBM Almaden Research Center, April 1986.

Andrew Wellings. *Concurrent and Real-Time Programming in Java*. John Wiley & Sons, 2004.

Брайан Гетц, Тим Пайерлс, Джошуа Блох, Джозеф Боубер, Дэвид Холмс, Даг Ли

Java Concurrency на практике

Перевел на русский А. Логунов

Заведующая редакцией	*Ю. Сергиенко*
Ведущий редактор	*К. Тульцева*
Научный редактор	*Н. Мишин*
Литературный редактор	*А. Руденко*
Корректоры	*Н. Сидорова, М. Молчанова*
Верстка	*Е. Неволайнен*

Изготовлено в России. Изготовитель: ООО «Прогресс книга».Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург, Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 02.2020. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 17.01.20. Формат 70x100/16. Бумага офсетная. Усл. п. л. 38,700. Тираж 1200. Заказ

ИЗДАТЕЛЬСКИЙ ДОМ ПИТЕР®
WWW.PITER.COM

ИЗДАТЕЛЬСКИЙ ДОМ «ПИТЕР» предлагает профессиональную, популярную и детскую развивающую литературу

Заказать книги оптом можно в наших представительствах

РОССИЯ

Санкт-Петербург: м. «Выборгская», Б. Сампсониевский пр., д. 29а
тел./факс: (812) 703-73-83, 703-73-72; e-mail: sales@piter.com

Москва: м. «Электрозаводская», Семеновская наб., д. 2/1, стр. 1, 6 этаж
тел./факс: (495) 234-38-15; e-mail: sales@msk.piter.com

Воронеж: тел.: 8 951 861-72-70; e-mail: hitsenko@piter.com

Екатеринбург: ул. Толедова, д. 43а; тел./факс: (343) 378-98-41, 378-98-42;
e-mail: office@ekat.piter.com; skype: ekat.manager2

Нижний Новгород: тел.: 8 930 712-75-13; e-mail: yashny@yandex.ru; skype: yashny1

Ростов-на-Дону: ул. Ульяновская, д. 26
тел./факс: (863) 269-91-22, 269-91-30; e-mail: piter-ug@rostov.piter.com

Самара: ул. Молодогвардейская, д. 33а, офис 223
тел./факс: (846) 277-89-79, 277-89-66; e-mail: pitvolga@mail.ru,
pitvolga@samara-ttk.ru

БЕЛАРУСЬ

Минск: ул. Розы Люксембург, д. 163; тел./факс: +37 517 208-80-01, 208-81-25;
e-mail: og@minsk.piter.com

Издательский дом «Питер» приглашает к сотрудничеству авторов:
тел./факс: (812) 703-73-72, (495) 234-38-15; e-mail: ivanovaa@piter.com
Подробная информация здесь: http://www.piter.com/page/avtoru

Издательский дом «Питер» приглашает к сотрудничеству зарубежных торговых партнеров или посредников, имеющих выход на зарубежный рынок: *тел./факс: (812) 703-73-73; e-mail: sales@piter.com*

Заказ книг для вузов и библиотек:
тел./факс: (812) 703-73-73, доб. 6243; e-mail: uchebnik@piter.com

Заказ книг по почте: *на сайте www.piter.com; тел.: (812) 703-73-74, доб. 6216;*
e-mail: books@piter.com

Вопросы по продаже электронных книг: *тел.: (812) 703-73-74, доб. 6217;*
e-mail: kuznetsov@piter.com

КНИГА-ПОЧТОЙ

ЗАКАЗАТЬ КНИГИ ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР» МОЖНО ЛЮБЫМ УДОБНЫМ ДЛЯ ВАС СПОСОБОМ:

- на нашем сайте: **www.piter.com**
- по электронной почте: **books@piter.com**
- по телефону: **(812) 703-73-74**

ВЫ МОЖЕТЕ ВЫБРАТЬ ЛЮБОЙ УДОБНЫЙ ДЛЯ ВАС СПОСОБ ОПЛАТЫ:

- Наложенным платежом с оплатой при получении в ближайшем почтовом отделении.
- С помощью банковской карты. Во время заказа вы будете перенаправлены на защищенный сервер нашего оператора, где сможете ввести свои данные для оплаты.
- Электронными деньгами. Мы принимаем к оплате Яндекс.Деньги, Webmoney и Qiwi-кошелек.
- В любом банке, распечатав квитанцию, которая формируется автоматически после совершения вами заказа.

ВЫ МОЖЕТЕ ВЫБРАТЬ ЛЮБОЙ УДОБНЫЙ ДЛЯ ВАС СПОСОБ ДОСТАВКИ:

- Посылки отправляются через «Почту России». Отработанная система позволяет нам организовывать доставку ваших покупок максимально быстро. Дату отправления вашей покупки и дату доставки вам сообщат по e-mail.
- Вы можете оформить курьерскую доставку своего заказа (более подробную информацию можно получить на нашем сайте www.piter.com).
- Можно оформить доставку заказа через почтоматы,(адреса почтоматов можно узнать на нашем сайте www.piter.com).

ПРИ ОФОРМЛЕНИИ ЗАКАЗА УКАЖИТЕ:

- фамилию, имя, отчество, телефон, e-mail;
- почтовый индекс, регион, район, населенный пункт, улицу, дом, корпус, квартиру;
- название книги, автора, количество заказываемых экземпляров.

БЕСПЛАТНАЯ ДОСТАВКА:

- курьером по Москве и Санкт-Петербургу при заказе на сумму **от 2000 руб.**
- почтой России при предварительной оплате заказа на сумму **от 2000 руб.**