21 Параллельное выполнение

До настоящего момента мы занимались последовательным программированием, при котором все действия в программе выполняются одно за другим.

Последовательное программирование способно решить многие задачи. Однако в некоторых областях бывает удобно (и даже необходимо) выполнять несколько частей программы параллельно, чтобы создать иллюзию их параллельного выполнения — или, если компьютер оснащен несколькими процессорами, действительно выполнять их одновременно.

Параллельное программирование способно серьезно улучшить скорость выполнения и/или предоставить упрощенную модель проектирования для некоторых видов программ. Тем не менее освоение теории и основных приемов параллельного программирования выходит за рамки всего, что вы узнали в этой книге; эта тема лежит где-то между промежуточным и высоким уровнем. Данная глава содержит вводный материал, и вы не можете считать себя хорошим специалистом по параллельному программированию, даже если полностью поймете весь материал.

Как вы вскоре увидите, настоящие проблемы с параллельным выполнением начинаются тогда, когда одновременно выполняемые задачи начинают вмешиваться в дела друг друга. Такое вмешательство бывает настолько тонким и неочевидным, что параллельное выполнение можно с вескими основаниями считать условно-недетерминированным. Другими словами, теоретически возможно написать параллельные программы, которые благодаря внимательности программиста и анализу кода всегда работают корректно. Однако на практике гораздо проще писать параллельные программы, которые на первый взгляд работают корректно, но при некоторых условиях порождают ошибки. Возможно, эти условия никогда не возникнут или же будут возникать так редко, что никогда не встретятся при тестировании. Более того, нет гарантий, что вы сможете написать тестовый код, порождающий сбои в вашей параллельной программе. Ошибки будут происходить в очень редких случаях и в результате обернутся жалобами пользователей. Это одна из самых веских причин для изучения параллельного программирования: если вы не будете уделять ему внимания, то, скорее всего, сами же от этого и пострадаете.

Итак, параллельное программирование сопряжено с неизбежным риском. Если вас это немного пугает, пожалуй, это даже хорошо. Хотя в Java SE5 в области параллельности были достигнуты значительные усовершенствования, до сих пор не существует никакой «страховки» вроде проверки во время компиляции или контролируемых исключений, сообщающих о допущенных ошибках. В области параллельного программирования вы сами отвечаете за все, и только повышенная подозрительность в сочетании с настойчивостью позволит вам писать надежный многопоточный код на Java.

Некоторые люди считают, что параллельное программирование — слишком сложная тема для включения в книгу, которая только знакомит читателя с языком. Они полагают, что эта тема должна излагаться отдельно, а для тех редких случаев, когда она проявляется в повседневном программировании (например, построении графических интерфейсов), можно подобрать специальные идиомы. Зачем излагать такую сложную тему, если ее можно обойти?

Если бы так... К сожалению, вы не властны над тем, где и как в ваших программах Java проявятся программные потоки. Даже если вы никогда не запускаете поток самостоятельно, это не значит, что вам удастся избежать написания многопоточного кода. Например, веб-системы — одна из самых распространенных областей для применения Java, а базовый класс *сервлета* изначально является многопоточным — и это неизбежно, потому что веб-серверы часто оснащаются несколькими процессорами, а параллельность является идеальным способом эффективного использования дополнительных процессоров. Каким бы простым ни казался сервлет, для правильного использования сервлетов необходимо понимать проблемы параллельного программирования. Это относится и к программированию графического интерфейса, как вы увидите в главе 22. Хотя в библиотеках Swing и SWT предусмотрены механизмы потоковой безопасности, вряд ли кто-то сможет правильно пользоваться ими без понимания параллельности.

Java является многопоточным языком, и проблемы параллельного программирования присутствуют независимо от того, знаете вы о них или нет. В результате сейчас используется много Java-приложений, которые либо работают по случайности, либо работают большую часть времени, а иногда непостижимым образом «ломаются» от скрытых дефектов, связанных с параллельным выполнением. Иногда такие поломки приводят к потере ценных данных, и если вы не будете хотя бы в общих чертах представлять проблемы параллельности, то можете прийти к ошибочному выводу, что проблема кроется не в вашей программе, а где-то еще. Такие проблемы также могут выявляться или усиливаться при перемещении программы в многопроцессорную систему. По сути, разбираясь в параллелизме, вы будете знать, что даже правильная на первый взгляд программа может проявлять неправильное поведение.

Параллельное программирование напоминает изучение нового языка — или по крайней мере новых языковых концепций. Понять логику параллельного программирования так же сложно, как понять логику объектно-ориентированного программирования. С некоторыми усилиями можно понять базовый механизм, но для подлинного понимания вопроса требуется углубленное изучение. Эта глава закладывает прочный фундамент в области основ, чтобы вы понимали концепции и могли писать несложные многопоточные программы. Тем не менее для написания любой сколько-нибудь нетривиальной программы вам придется изучать специализированную литературу по теме.

Многогранная параллельность

Главная причина сложностей с изучением параллельного программирования заключается в том, что оно применяется для решения самых разнообразных задач и реализуется разными методами, поэтому между разными случаями не существует четкого соответствия. В результате вам приходится разбираться во всех нюансах и частных случаях, чтобы эффективно использовать параллельное программирование.

Задачи, решаемые при помощи параллельного программирования, можно условно разделить на категории «скорости» и «управляемости структуры».

Ускорение выполнения

Со скоростью на первый взгляд дело обстоит просто: если вы хотите, чтобы программа выполнялась быстрее, разбейте ее на части и запустите каждую часть на отдельном процессоре. Параллельность является основным инструментом многопроцессорного программирования. В наши дни, когда закон Мура постепенно перестает действовать (по крайней мере для традиционных микросхем), ускорение проявляется в форме многоядерных процессоров, нежели в ускорении отдельных чипов. Чтобы ваша программа работала быстрее, вы должны научиться эффективно использовать дополнительные процессоры, и это одна из возможностей, которую параллельное программирование может вам предоставить.

На многопроцессорном компьютере задачи могут распределяться по разным процессорам, что приводит к радикальному возрастанию скорости. Это явление характерно для мощных многопроцессорных веб-серверов, которые могут распределять большое количество пользовательских запросов по разным процессорам в программе, назначающей отдельный поток для каждого запроса.

Однако параллельность часто повышает производительность программ, выполняющихся на *одном* процессоре.

На первый взгляд это нелогично. Если задуматься, выполнение многопоточной программы на одном процессоре неизбежно должно сопровождаться *повышенными* затратами ресурсов по сравнению с последовательным выполнением всех частей программы, из-за так называемых переключений контекста (перехода от одной задачи к другой). Казалось бы, эффективнее выполнить все части программы как одну задачу и избавиться от затрат на переключение контекста.

Однако при этом необходимо учитывать проблему *блокирования*. Если одна задача в программе не может продолжать выполнение из-за какого-то условия, неподконтрольного программе (обычно ввода-вывода), говорят, что эта задача или программный поток *блокируется* (blocks). Традиционная программа останавливается до изменения внешнего условия. Однако в программе, написанной с учетом параллельности, во время блокировки одной задачи могут продолжать выполняться другие задачи, так что программа не будет простаивать. Итак, с точки зрения производительности параллельность на однопроцессорной машине имеет смысл только в том случае, если некоторые задачи могут блокироваться.

Очень распространенным примером повышения быстродействия на однопроцессорной системе является модель *событийного программирования*. Возьмем программу, которая выполняет некоторую продолжительную операцию и перестает реагировать на действия пользователя. Конечно, можно предусмотреть кнопку прерывания операции, но опрашивать ее состояние в каждом написанном вами фрагменте кода явно нежелательно. Такие проверки загромождают код, вдобавок ничего не гарантирует, что программист не забудет выполнить проверку. Без применения параллельности реакцию пользовательского интерфейса можно обеспечить только одним способом: периодической проверкой ввода во всех задачах. Создание отдельного потока выполнения для реакции на действия пользователя (даже при том, что этот поток большую часть времени будет оставаться заблокированным) гарантирует определенную способность программы к реагированию на внешние воздействия.

Программа должна продолжать свою работу и в то же время она должна вернуть управление пользовательскому интерфейсу для обработки действий пользователя. Однако традиционный механизм выполнения не предоставляет такой возможности. Все выглядит так, словно процессор должен делать два дела одновременно — но именно такую иллюзию создает параллельное программирование (а в многопроцессорной системе это больше чем иллюзия).

Один из простейших механизмов реализации параллельности — *процессы* уровня операционной системы. Процесс представляет собой самостоятельную программу, выполняемую в собственном адресном пространстве. Многозадачная операционная система может одновременно выполнять более одного процесса (программы), переключая процессор между процессами; внешне все выглядит так, словно каждый процесс постепенно двигается вперед без прерывания. Процессы очень удобны, потому что операционная система обычно изолирует их друг от друга; таким образом, один процесс не может помешать выполнению другого процесса, благодаря чему программирование оказывается относительно простым. Напротив, параллельные системы (вроде той, что используется в Java) совместно используют ресурсы — память, ввод-вывод и т. д., поэтому основные трудности при написании многопоточных программ связаны с координацией использования ресурсов между задачами разных потоков, чтобы в любой момент времени ресурс был доступен только для одной задачи.

Рассмотрим простой пример использования процессов операционной системы. Во время работы над книгой я часто создавал несколько резервных копий текущего состояния материалов. Одна копия сохранялась в локальном каталоге, другая на карте памяти, третья на Zip-диске и четвертая на сервере FTP. Чтобы автоматизировать этот процесс, я написал маленькую программу (на Python, но концепции остаются теми же), которая архивирует материалы в файл, имя которого включает номер версии, а затем выполняет копирование. Изначально все копии создавались последовательно, а каждая новая операция копирования начиналась только после завершения предыдущей. Но потом я понял, что операции копирования занимают разное время в зависимости от скорости ввода-вывода носителя. Так как я работаю в многозадачной операционной системе, я мог запустить каждую операцию копирования в отдельном процессе, а затем выполнять их параллельно, чтобы ускорить выполнение всей программы. Если один процесс блокировался, другой мог двигаться вперед.

Это идеальный пример параллельного выполнения. Каждая задача выполняется как самостоятельный процесс в своем адресном пространстве, поэтому любые конфликты

между задачами исключены. Что еще важнее, задачам не нужно взаимодействовать друг с другом — они полностью независимы. Операционная система берет на себя все технические подробности и следит за выполнением копирования. Никакого риска, а выполнение программы ускоряется практически «бесплатно».

Некоторые специалисты доходят до того, что представляют процессы как единственно разумный подход к организации параллельного выполнения[1]. К сожалению, применимость процессов в области параллельного выполнения обычно ограничивается факторами, относящимися к их количеству и непроизводительным затратам.

Некоторые языки программирования спроектированы так, чтобы изолировать параллельно выполняемые задачи друг от друга. Обычно в так называемых *функциональных языках* любой вызов функции не создает побочных эффектов (а следовательно, не может помешать другим функциям), что позволяет выполнять его в независимой задаче. Один из таких языков — Erlang — включает безопасные механизмы «общения» между задачами. Если выясняется, что часть вашей программы должна интенсивно использовать параллельность и у вас возникает слишком много проблем с построением этой части, возможно, ее следует написать на специализированном языке параллельных взаимодействий — таком, как Erlang.

В Java был выбран более традиционный подход добавления поддержки многопоточной модели на базе последовательного языка[2]. Вместо того чтобы порождать внешние процессы в многозадачной операционной системе, модель потоков создает новые задачи *в пределах* одного процесса, представленного выполняемой программой. Одним из преимуществ такого решения была прозрачность операционной системы. Например, версии операционной системы Macintosh, предшествовавшие OS X (достаточно важная область для первых версий Java), не поддерживали многозадачность. Без реализации многопоточности в Java параллельные программы Java не переносились бы на Macintosh и другие платформы, нарушая тем самым требование «написано однажды — работает везде»[3].

Улучшение структуры кода

Программа, использующая многозадачность на машине с одним процессором, в любой момент времени все равно выполняет только одну операцию, поэтому теоретически должно быть возможно написать ту же программу в однозадачном варианте. Однако параллельность предоставляет важные архитектурные преимущества: она способна заметно упростить структуру программы. Некоторые виды задач — например, моделирование — плохо решаются без поддержки параллельности.

---

[1] Например, Эрик Рэймонд настоятельно продвигает эту точку зрения в книге «The Art of UNIX Programming» (Addison-Wesley, 2004).

[2] Иногда высказывается мнение, что попытки «пристегнуть» параллелизм к последовательному языку обречены на неудачу, но вы должны составить собственное мнение.

[3] Вообще-то это требование так и не было реализовано в полной мере, поэтому компания Sun уже не так громко продвигает этот лозунг. Парадоксально, но одна из причин, по которой принцип «написано однажды – работает везде» не сработал в полной мере, была связана с проблемами в системе потокового выполнения, которые, возможно, будут исправлены в Java SE5.

Вероятно, каждый сталкивался с моделированием в компьютерных играх или в компьютерных анимациях в фильмах. В моделировании обычно задействовано много взаимодействующих элементов, каждый из которых обладает собственным «интеллектом». И хотя на однопроцессорной машине каждый элемент модели обрабатывается единственным процессором, с точки зрения программирования намного проще считать, что каждый элемент модели выполняется на отдельном процессоре и является независимой задачей.

В полномасштабной модели может быть задействовано очень большое число задач, поскольку все моделируемые элементы могут действовать независимо друг от друга. В многопоточных системах часто устанавливается относительно невысокое ограничение на количество доступных потоков (порядка десятков или сотен). Это ограничение может быть неподконтрольно программе — оно может зависеть от платформы или в случае Java от версии JVM. В Java в общем случае можно считать, что количество потоков недостаточно для выделения отдельного потока каждому элементу масштабной модели.

Типичное решение проблемы заключается в использовании *кооперативной многопоточности*. В Java используется потоковая модель *с вытеснением*, при которой механизм планирования выделяет временные *кванты* (slices) каждому потоку, периодически прерывая его выполнение и осуществляя переключение контекста на другой поток так, чтобы каждому потоку выделялось разумное время на выполнение его задачи. В кооперативной системе каждая задача добровольно уступает управление, для чего программист должен сознательно вставить в каждую задачу соответствующую команду. Кооперативная система обладает двумя важными преимуществами: во-первых, переключение контекста обычно требует существенно меньших затрат, чем в системе с вытеснением, а во-вторых, количество одновременно выполняемых независимых задач теоретически не ограничивается. В моделях с большим количеством элементов такое решение нередко оказывается идеальным. Однако следует помнить, что некоторые кооперативные системы не предусматривают распределения задач между процессорами, и это ограничение может оказаться очень существенным.

С другой стороны, модель параллельного выполнения очень полезна при работе с современными системами передачи сообщений, в которых задействовано множество независимых компьютеров, распределенных в сети. В этом случае все процессы выполняются полностью независимо друг от друга, без возможности совместного доступа к ресурсам. Однако при этом все равно необходимо синхронизировать передачу информации между процессами, чтобы предотвратить возможную потерю данных или их поступление в неправильное время. Даже если вы не планируете часто использовать параллельное программирование в ближайшем будущем, желательно понимать его, чтобы разобраться в архитектурах передачи сообщений, все чаще применяемых при создании распределенных систем.

Параллельное выполнение сопряжено с определенными затратами, включая возрастание сложности, но обычно они компенсируются улучшением архитектуры программы, балансировкой ресурсов и удобством использования. Как правило, потоки позволяют снизить связанность архитектуры; в противном случае частям вашего кода придется заниматься выполнением операций, которые обычно выполняются потоками автоматически.

Основы построения многопоточных программ

Параллельное программирование позволяет разделить программу на несколько независимых частей. Довольно часто бывает необходимо превратить программу в несколько отдельных, самостоятельно выполняющихся подзадач. Каждая из этих самостоятельных подзадач называется *потоком* (thread). Поток — это выполняемая параллельно в рамках процесса последовательность команд программы. Таким образом, процесс может одновременно содержать несколько выполняющихся потоков, но вы пишете программу так, как будто каждый поток запускается сам по себе и использует процессор монопольно. На самом деле существует некоторый низкоуровневый механизм, который разделяет время процессора, но в основном думать об этом вам не придется.

Модель потоков (и ее поддержка в языке Java) является программным механизмом для упрощения выполнения нескольких операций синхронно в одной и той же программе: процессор вмешивается в происходящее и выделяет каждому потоку некоторый отрезок времени[1]. Каждый поток полагает, что он использует процессор монопольно, но на самом деле время процессора разделяется между всеми существующими в программе потоками (кроме случаев, в которых программа действительно работает на нескольких процессорах). Однако при использовании потоков вам не нужно задумываться об этих тонкостях, ваш код не зависит от того, на скольких процессорах вам придется работать, и это замечательно. Таким образом, потоки предоставляют способ написания программ с прозрачной масштабируемостью — если ваша программа работает слишком медленно, вы в силах легко ускорить ее, добавив на свой компьютер дополнительные процессоры. Многозадачность и многопоточность являются, похоже, наиболее вескими причинами использования многопроцессорных систем.

Определение задач

Поток выполняет некоторую задачу, поэтому вам нужны средства для описания таких задач. Эти средства предоставляются интерфейсом `Runnable`. Чтобы определить задачу, просто реализуйте `Runnable` и напишите метод `run()`, который выполнит нужные действия.

Например, следующая задача `LiftOff` выводит обратный отсчет перед запуском:

```
//: concurrency/LiftOff.java
// Demonstration of the Runnable interface.

public class LiftOff implements Runnable {
  protected int countDown = 10; // Default
  private static int taskCount = 0;
  private final int id = taskCount++;
  public LiftOff() {}
  public LiftOff(int countDown) {
```

продолжение ⇩

[1] Это относится к системам, использующим механизм выделения квантов процессорного времени (например, Windows). В Solaris используется модель FIFO: если в системе не будет активизирован поток с более высоким приоритетом, текущий поток работает до тех пор, пока он не будет заблокирован или завершен. Это означает, что другие потоки с равным приоритетом не будут запущены до тех пор, пока текущий поток не уступит процессор.

```
    this.countDown = countDown;
  }
  public String status() {
    return "#" + id + "(" +
      (countDown > 0 ? countDown : "Liftoff!") + "), ";
  }
  public void run() {
    while(countDown-- > 0) {
      System.out.print(status());
      Thread.yield();
    }
  }
} ///:~
```

Идентификатор `id` различает экземпляры задачи. Он объявлен с ключевым словом `final`, потому что значение не должно изменяться после инициализации.

Метод `run()` практически всегда являет собой некоторый цикл, который выполняется, пока поток еще нужен, поэтому вам придется определить условие выхода из такого цикла (можно просто использовать команду `return`, как сделано в рассматриваемой программе). Зачастую метод `run()` реализуется в форме бесконечного цикла; это значит, что завершение потока осуществляется с помощью какого-либо внешнего фактора или он будет выполняться бесконечно (чуть позже в этой главе вы узнаете, как безопасно сигнализировать потоку, чтобы он «остановился»).

Вызов статического метода `Thread.yield()` внутри `run()` является рекомендацией для планировщика потоков (подсистема механизма потоков Java, которая переключает процессор с одного потока на другой). По сути она означает: «Важная часть моего цикла выполнена, и сейчас было бы неплохо переключиться на другую задачу». Этот вызов полностью необязателен, но он используется здесь, потому что с ним результат получается более интересным: вы с большей вероятностью увидите доказательства переключения потоков.

В следующем примере вызов `run()` не управляется отдельным потоком; метод просто напрямую вызывается в `main()` (поток, конечно, при этом используется: тот, который всегда создается для `main()`):

```
//: concurrency/MainThread.java

public class MainThread {
  public static void main(String[] args) {
    LiftOff launch = new LiftOff();
    launch.run();
  }
} /* Output:
#0(9), #0(8), #0(7), #0(6), #0(5), #0(4), #0(3), #0(2),
#0(1), #0(Liftoff!),
*///:~
```

Класс, производный от `Runnable`, должен содержать метод `run()`, но ничего особенного в этом методе нет — он не обладает никакими «встроенными» потоковыми способностями. Чтобы добиться потокового выполнения, необходимо явно присоединить задачу к потоку.