

```

public Builder author(Author author_) {
    author = author_;
    return this;
}

public Builder updateText(String updateText_) {
    updateText = updateText_;
    return this;
}

public Update build() {
    return new Update(this);
}
}

```

可用在调用链中返回
Builder的方法

略去hashCode()和
equals()方法

有了这段代码，你就可以创建新的Update对象：

```
Update.Builder ub = new Update.Builder();
Update u = ub.author(myAuthor).updateText("Hello").build();
```

这是一个得到广泛应用的通用模式。实际上，在代码清单4-1和4-2中我们已经使用了不可变对象的特性。

关于不可变对象的最后一点——关键字final仅对其直接指向的对象有用。如图4-6所示，对主对象的引用不能赋值为对象3，但在主对象内部，对1的引用可以改为指向对象2。也就是说final引用可以指向带有非final域的对象。

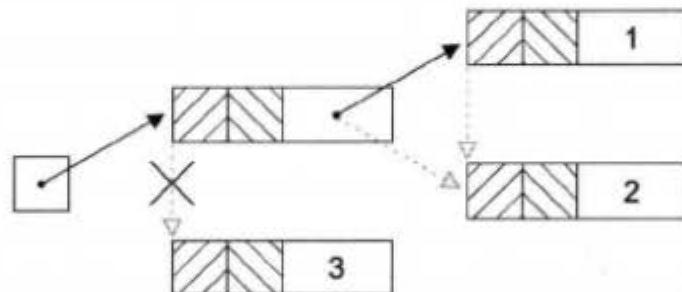


图4-6 值的不可变性与引用

不可变是非常强的技术，用处十分广泛。但有时候只用不可变对象开发效率不行，因为每次修改对象状态就需要构建一个新对象。所以可变对象很有必要保留。

我们马上就要开始讨论本章最重要的主题——java.util.concurrent中更加现代化、概念更简单的并发API。看看怎么用它们写代码。

4.3 现代并发应用程序的构件

随着Java 5的到来，Java对并发的重新思考也浮出了水面。这些新思想主要体现在java.util.concurrent包上，其中包含了大量用来编写多线程代码的新工具。在后续版本中，这些工具不断得到改进，但其工作方式却依然保持不变，并且直到今天还是对开发人员很有帮助。

我们马上快速过一下java.util.concurrent中主要的类及相关包，比如atomic和locks包。我们会向你介绍这些类及其适用的情景。你也应该读一下它们的Javadoc，并尝试熟悉整个包——它们使编写并发类容易多了。

代码迁移

如果你还有基于（Java 5之前的）老办法编写的多线程代码，建议你用java.util.concurrent重构。按我们的经验，几乎在所有案例中，如果你特意把代码迁移到新的API中，代码就会得以改进。你的努力付出将使代码在清晰性和可靠性上得到极大提升。

请把这次讨论当做并发编程的启动工具，而不是一次研讨会。想要充分利用好java.util.concurrent，你还需要知道更多的知识。

4.3.1 原子类：java.util.concurrent.atomic

java.util.concurrent.atomic中有几个名字以Atomic打头的类。它们的语义基本上和volatile一样，只是封装在一个API里了，这个API包含为操作提供的适当的原子（要么不做，要做就全做）方法。对于开发人员来说，这是非常简单的避免在共享数据上出现竞争危害^①的办法。

在编写这些实现时利用了现代处理器的特性，所以如果能从硬件和操作系统上得到适当的支持，它们可以是非阻塞（无需线程锁）的，而大多数现代系统都能提供这种支持。常见的用法是实现序列号机制，在AtomicInteger或AtomicLong上用原子操作getAndIncrement()方法。

要做序列号，该类应该有个nextId()方法，每次调用时肯定能返回一个唯一并且完全增长的数值。这和数据库里序列号的概念很像（所以这个变量叫这个名字）。

来看一段产生序列号的代码：

```
private final AtomicLong sequenceNumber = new AtomicLong(0);

public long nextId() {
    return sequenceNumber.getAndIncrement();
}
```

注意 原子类不是从有相似名称的类继承而来的，所以AtomicBoolean不能当Boolean用，AtomicInteger也不是Integer，虽然它确实扩展了Number。

接下来，我们会检查一下java.util.concurrent如何对同步模型的核心建模——Lock接口。

4.3.2 线程锁：java.util.concurrent.locks

块结构同步方式基于锁这样一个简单的概念。这种方式有几个缺点。

- 锁只有一种类型。
- 对被锁住对象的所有同步操作都是一样的作用。

^① 竞争危害（race hazard）又名竞态条件（race condition）。一个系统或进程的输出，依赖于不受控制事件的出现顺序或时机。例如两个进程都试图修改一个共享内存的内容。在没有并发控制的情况下，最后的结果取决于两个进程的执行顺序与时机，如果发生了并发访问冲突，最后的结果是不正确的。——译者注

- 在同步代码块或方法开始时取得线程锁。
- 在同步代码块或方法结束时释放线程锁。
- 线程或者得到锁，或者阻塞——没有其他可能。

如果我们要重构对线程锁的支持，有几处可以得到提升。

- 添加不同类型的锁，比如读取锁和写入锁。
- 对锁的阻塞没有限制，即允许在一个方法中上锁，在另一个方法中解锁。
- 如果线程得不到锁，比如锁由另外一个线程持有，就允许该线程后退或继续执行，或者做点别的事情——运用tryLock()方法。
- 允许线程尝试取锁，并可以在超过等待时间后放弃。

能实现以上这些的关键就是java.util.concurrent.locks中的Lock接口。还有它的两个实现类。

- ReentrantLock——本质上跟用在同步块上那种锁是一样的，但它要稍微灵活点儿。
- ReentrantReadWriteLock——在需要读取很多线程而写入很少线程时，用它性能会更好。

块结构并发能实现的所有功能都可以用Lock接口实现。下面是用ReentrantLock重写的那个死锁的例子。

代码清单4-4 用ReentrantLock重写死锁

```
private final Lock lock = new ReentrantLock();

public void propagateUpdate(Update upd_, MicroBlogNode backup_) {
    lock.lock();                                ← 每个线程都先锁住自己的锁
    try {
        System.out.println(ident+": recvd: "+
            ↪ upd_.getUpdateText() +" ; backup: "+
            ↪ backup_.getIdent());
        backup_.confirmUpdate(this, upd_);          ← 调用confirmUpdate()
    } finally {                                    知悉其他线程
        lock.unlock();
    }
}

public void confirmUpdate(MicroBlogNode other_, Update upd_) {
    lock.lock();                                ← 尝试锁住其他线程 ①
    try{
        System.out.println(iden+": recvd confirm: "+
            ↪ upd_.getUpdateText() +" from "+ other_.getIdentifier());
    } finally {
        lock.unlock();
    }
}
```

锁住其他线程的尝试①通常都会失败，因为它已经被锁住了（如图4-3所示）。这就是导致死锁出现的原因。

用锁时带上try...finally

把lock()放在try...finally块中（释放也在那里）的模式是另外一个好用的小工具。在跟块结构并发相似的情景中它同样很好用。而另一方面，如果需要传递Lock对象，比如从一个方法中返回，则不能用这个模式。

使用Lock对象可能要比块结构方式强大得多，但有时用它们很难设计出完善的锁定策略。

对付死锁的策略有很多，但你应该特别注意一个不起任何作用的策略。请看下面这段代码中新版的propagateUpdate()方法（假定confirmUpdate()也做出了同样的修改）。在这个例子中，我们用带有超时机制的tryLock()替换了无条件的锁。通过这种办法可以为其他线程提供得到线程锁的机会，从而去除死锁。

代码清单4-5 一次有缺陷的解决死锁问题的尝试

```
public void propagateUpdate(Update upd_, MicroBlogNode backup_) {
    boolean acquired = false;

    while (!acquired) {
        try {
            int wait = (int)(Math.random() * 10);
            acquired = lock.tryLock(wait, TimeUnit.MILLISECONDS); ←
            if (acquired) {
                System.out.println(ident +": recvd: "+ ←
                    upd_.getUpdateText() +" ; backup: "+backup_.getIdent()); ←
                backup_.confirmUpdate(this, update_); ←
            } else {
                Thread.sleep(wait);
            }
        } catch (InterruptedException e) {
        } finally {
            if (acquired) lock.unlock(); ←
        }
    }
}
```

尝试与锁定，
超时时长随机

在其他线程
上确认

仅在锁定
时解锁

如果运行代码清单4-5中的代码，你会发现它有时候还是不能解决死锁问题。你能看到“received confirm of update”，但它并不会一直出现，时有时无。

实际上，死锁问题并没有真正解决，因为如果线程取得了第一个锁（在propagateUpdate()中），它才会调用confirmUpdate()，并且在完成之前绝不会释放第一个锁。即使两个线程都能在彼此调用confirmUpdate()之前取得第一个线程锁，它们还是会产生成死锁。

如果取得第二个锁的尝试失败，能真正解决问题的办法是让线程释放其持有的第一个锁，再次从头开始等待，从而使其他线程有机会得到完整的锁集合，能走完全程。代码如下所示。

代码清单4-6 修正死锁

```
public void propagateUpdate(Update upd_, MicroBlogNode backup_) {
    boolean acquired = false;
    boolean done = false;
```

```

while (!done) {
    int wait = (int)(Math.random() * 10);
    try {
        acquired = lock.tryLock(wait, TimeUnit.MILLISECONDS);
        if (acquired) {
            System.out.println(ident +": recvd: "+ 
                ↪ upd_.getUpdateText() +" ; backup: "+backup_.getIdent());
            done = backupNode_.tryConfirmUpdate(this, update_);
        }
    } catch (InterruptedException e) {
    } finally {
        if (acquired) lock.unlock();
    }
    if (!done) try {
        Thread.sleep(wait);
    } catch (InterruptedException e) { }
}
}

public boolean tryConfirmUpdate(MicroBlogNode other_, Update upd_) {
    boolean acquired = false;
    try {
        int wait = (int)(Math.random() * 10);
        acquired = lock.tryLock(wait, TimeUnit.MILLISECONDS);

        if (acquired) {
            long elapsed = System.currentTimeMillis() - startTime;
            System.out.println(ident +": recvd confirm: "+ 
                ↪ upd_.getUpdateText() +" from "+other_.getIdent()
                ↪ +" - took "+ elapsed +" millis");
            return true;
        }
    } catch (InterruptedException e) {
    } finally {
        if (acquired) lock.unlock();
    }
    return false;
}

```

如果done为false，
释放锁并等待

检查
tryConfirmUpdate()
的返回值

这一版会检查tryConfirmUpdate()的返回码。如果为false，最初的锁被释放。该线程会暂停一段时间，让其他线程有机会获取锁。

把这段代码运行几次，你会发现这两个线程基本上总能走完全程——死锁问题已经被你解决了。你也许想试验试验之前版本中那段代码的不同形式，诸如最原始的、有缺陷的或被改正的。通过对这些代码的演练，你能对锁机制有更深刻的理解，并且开始渐渐地凭直觉避免死锁问题的出现。

为什么那个有缺陷的版本有时候能奏效？

你已经看到了，死锁仍然存在，那是什么原因导致这个版本中的代码有时可以成功呢？代码中附加的复杂性是罪魁祸首。它影响JVM的线程调度器，让它变得更加难以预测。这意味着它有时候能让某个线程（通常是第一个）在其他线程运行之前进入confirmUpdate()方法并取得第二个锁。这种情况也会发生在原始代码中，只是可能性更低罢了。

我们只是揭开了Lock各种可能性的面纱——有很多种方法可以产生更加复杂的锁定结构。接下来我们就来讨论其中一个概念——锁存器。

4.3.3 CountDownLatch

CountDownLatch是一种简单的同步模式，这种模式允许线程在通过同步屏障之前做些少量的准备工作。

为了达到这种效果，在构建新的CountDownLatch实例时要给它提供一个int值（计数器）。此外，还有两个用来控制锁存器的方法：countDown()和await()。前者对计数器减1，而后者让调用线程在计数器到0之前一直等待。如果计数器已经为0或更小，则它什么也不做。这个简单的机制使得这种所需准备最少的模式非常容易部署。

在下面的代码中，同一进程内的一组更新处理线程至少必须有一半线程正确初始化（假定更新处理线程的初始化要占用一定时间）之后，才能开始接受系统发送给它们中的任何一个线程的更新。

代码清单4-7 用锁存器辅助初始化

```
public static class ProcessingThread extends Thread {
    private final String ident;
    private final CountDownLatch latch;

    public ProcessingThread(String ident_, CountDownLatch cdl_) {
        ident = ident_;
        latch = cdl_;
    }
    public String getIdentifier() {
        return identifier;
    }
    public void initialize() { ← 节点初始化
        latch.countDown();
    }
    public void run() {
        initialize();
    }
}

final int quorum = 1 + (int)(MAX_THREADS / 2);
final CountDownLatch cdl = new CountDownLatch(quorum);

final Set<ProcessingThread> nodes = new HashSet<>();
try {
    for (int i=0; i<MAX_THREADS; i++) {
        ProcessingThread local = new ProcessingThread("localhost:"+
            (9000 + i), cdl);
        nodes.add(local);
        local.start();
    }
    cdl.await(); ← 达到quorum, 开始
} catch (InterruptedException e) { 发送更新
} finally {
}
```

这段代码把锁存器的值设置为quorum。一旦被初始化的线程达到这个数量，就可以开始处理更新了。每个线程完成初始化后都会马上调用countDown()，所以主线程只需等待quorum的到来，然后启动（并派发更新，尽管我们没给出那部分代码）。

我们接下来要讨论的是对多线程开发人员来说最有用的类之一：java.util.concurrent中的ConcurrentHashMap。

4.3.4 ConcurrentHashMap

ConcurrentHashMap类是标准HashMap的并发版本。它改进了Collections类中提供的synchronizedMap()功能，因为那些方法返回的集合中包含的锁要比需要的多。

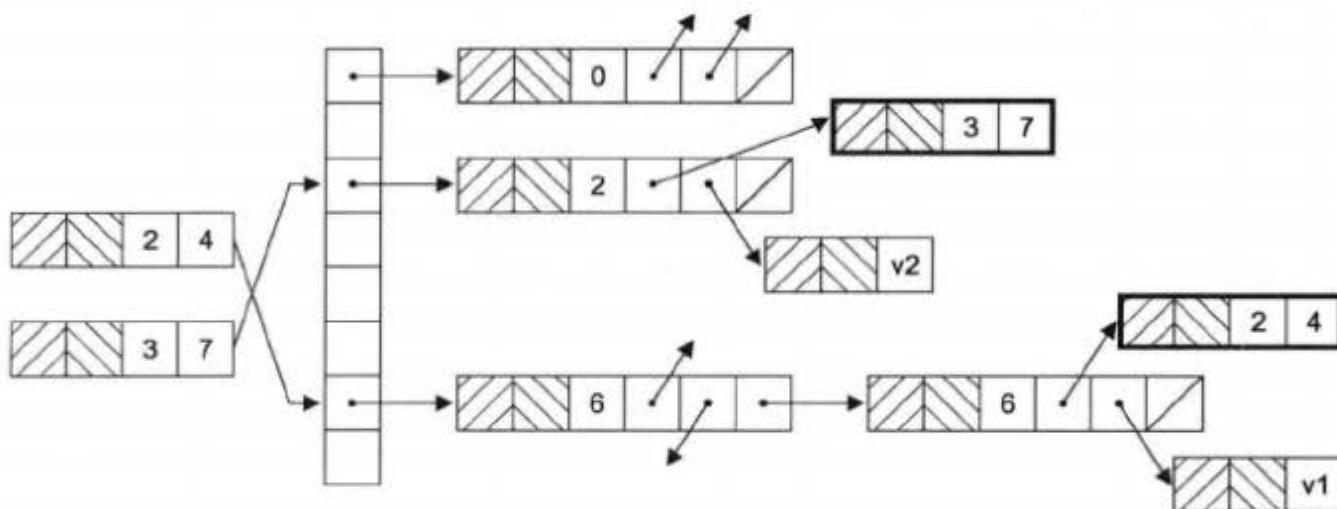


图4-7 HashMap的经典视图

如图4-7所示，传统的HashMap用hash函数来确定存放键/值对的“桶”，这是该类名字中“Hash”的由来。这意味着多线程处理可以更加简单直接——修改HashMap时并不需要把整个结构都锁住，只要锁住即将修改的桶就行了。

提示 好的并发HashMap实现在读取时不用锁，写入时只需锁住要修改的桶。Java基本上能达到这个标准，但这里还有一些大多数开发人员都无需过多关注的底层细节。

ConcurrentHashMap类还实现了ConcurrentMap接口，有些提供了原子操作的新方法。

- putIfAbsent()——如果还没有对应键，则把键/值对添加到HashMap中。
- remove()——如果对应键存在，且值也与当前状态相等(equal)，则用原子方式移除键值对。
- replace()——API为HashMap中原子替换的操作方法提供了两种不同的形式。

比如说，如果你把代码清单4-1中的私有final域arrivalTime的类型从HashMap改成ConcurrentHashMap，那就可以把synchronized方法替换成常规的非同步访问。注意代码清单4-8中锁的缺失——根本就没有显式的同步。

代码清单4-8 使用ConcurrentHashMap

```
public class ExampleMicroBlogTimingNode implements SimpleMicroBlogNode {
    ...
    private final Map<Update, Long> arrivalTime =
        new ConcurrentHashMap<>();
    ...
    public void propagateUpdate(Update upd_) {
        arrivalTime.putIfAbsent(upd_, System.currentTimeMillis());
    }
    public boolean confirmUpdateReceived(Update upd_) {
        return arrivalTime.get(upd_) != null;
    }
}
```

ConcurrentHashMap是java.util.concurrent包中最有用的类之一。它不仅提供了多线程的安全性，并且性能更优，在日常使用中没有严重的缺陷。接下来我们会讨论它的最佳拍档，用于List的CopyOnWriteArrayList。

4.3.5 CopyOnWriteArrayList

从名字就能看出来，CopyOnWriteArrayList是标准ArrayList的替代品。CopyOnWriteArrayList通过增加写时复制（copy-on-write）语义来实现线程安全性，也就是说修改列表的任何操作都会创建一个列表底层数组的新副本（如图4-8所示）。这就意味着所有成形的迭代器^①都不用担心它们会碰到意料之外的修改。

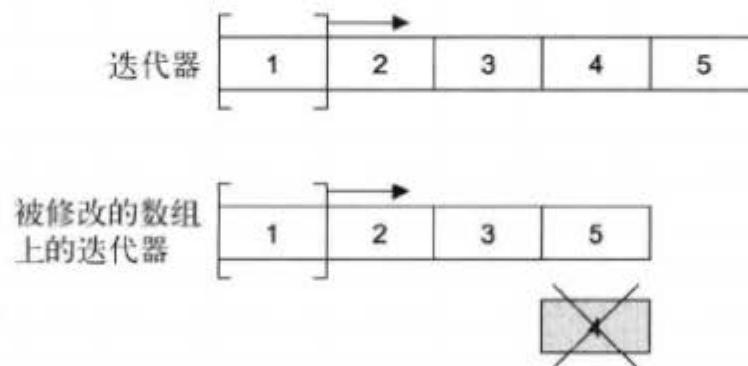


图4-8 写时复制数组

当快速、一致的数据快照（不同的读取器读到的数据偶尔可能会不一样）比完美的同步以及性能上的突破更重要时，这种共享数据的方法非常理想，并经常出现在非关键任务中。

我们来看一个写时复制的案例。假设有个微博的时间线更新，这是一个典型的非关键任务的例子。每个读取器的性能、自身一致性的快照要比全局的一致性更受欢迎。代码清单4-9表示每个用户时间线视图的持有者类。我们将会在代码清单4-10中用它来演示写时复制操作是如何进行的。

^① 迭代器（iterator）是一个对象，它的工作是遍历并选择序列中的对象，而客户端程序员不必知道或关心该序列底层的结构（也就是不同容器的类型）。——译者注

代码清单4-9 写时复制案例

```

public class MicroBlogTimeline {
    private final CopyOnWriteArrayList<Update> updates;
    private final ReentrantLock lock;
    private final String name;
    private Iterator<Update> it;
    public void addUpdate(Update update_) {
        updates.add(update_);
    }
    public void prep() {
        it = updates.iterator();
    }
    public void printTimeline() {
        lock.lock();
        try {
            if (it != null) {
                System.out.print(name+ ": ");
                while (it.hasNext()) {
                    Update s = it.next();
                    System.out.print(s+ ", ");
                }
                System.out.println();
            }
        } finally {
            lock.unlock();
        }
    }
}

```

我们专门设计了这个类来阐明在写时复制语义下的迭代器行为。你需要在输出方法中锁定，以防止输出在两个线程间乱掉，此外你也能看到两个线程各自的状态。

你可以从下面的代码中调用MicroBlogTimeline类。

代码清单4-10 揭示写时复制行为

```

final CountDownLatch firstLatch = new CountDownLatch(1);
final CountDownLatch secondLatch = new CountDownLatch(1);
final Update.Builder ub = new Update.Builder();

final List<Update> l = new CopyOnWriteArrayList<>();
l.add(ub.author(new Author("Ben")).updateText("I like pie").build());
l.add(ub.author(new Author("Charles")).updateText(
    "I like ham on rye").build());

ReentrantLock lock = new ReentrantLock();
final MicroBlogTimeline t1 = new MicroBlogTimeline("TL1", l, lock);
final MicroBlogTimeline t2 = new MicroBlogTimeline("TL2", l, lock);

Thread t1 = new Thread() {
    public void run() {
        l.add(ub.author(new Author("Jeffrey")).updateText(
            "I like a lot of things").build());
        t1.prep();
    }
}

```

```

firstLatch.countDown();
try { secondLatch.await(); }
  ↪ catch (InterruptedException e) { }
t1.printTimeline();
}

};

Thread t2 = new Thread(){
    public void run(){
        try {
            firstLatch.await();
            l.add(ub.author(new Author("Gavin")).updateText(
                "I like otters").build());
            t12.prep();
            secondLatch.countDown();
        } catch (InterruptedException e) { }
        t12.printTimeline();
    }
};
t1.start();
t2.start();

```

4

用锁存器严格限制事件的顺序

用锁存器严格限制事件的顺序

这段代码里有很多辅助的测试代码。但也有很多值得注意的地方：

- CountDownLatch用来严格控制两个线程之间发生的事情。
- 如果用普通的List代替CopyOnWriteArrayList，结果会导致出现ConcurrentModificationException异常。
- 这也是在两个线程之间共享一个Lock对象以控制对共享资源（即STDOUT）访问的例子。
如果用块结构方式写这段代码，会显得更加杂乱。

这段代码的输出如下：

```

TL2: Update [author=Author [name=Ben], updateText=I like pie, createTime=0],
      Update [author=Author [name=Charles], updateText=I like ham on rye,
      createTime=0], Update [author=Author [name=Jeffrey], updateText=I like a
      lot of things, createTime=0], Update [author=Author [name=Gavin],
      updateText=I like otters, createTime=0],

TL1: Update [author=Author [name=Ben], updateText=I like pie, createTime=0],
      Update [author=Author [name=Charles], updateText=I like ham on rye,
      createTime=0], Update [author=Author [name=Jeffrey], updateText=I like a
      lot of things, createTime=0],

```

第二行输出（标签为TL1）漏掉了最后一个更新，就是提到了水獭的那个，尽管按锁存器的意思在列表被修改后t11^①是可以访问的。这说明了t11中所包含的迭代器被t12复制，并且最后一个更新对t11是不可见的。这就是我们想要展示的写时复制特性。

CopyOnWriteArrayList的性能

使用CopyOnWriteArrayList类要比使用ConcurrentHashMap多花点心思，它是HashMap的即用型并发替代品。这是因为性能问题——写时复制特性意味着如果列表在被读取

^① 原文为mbex1，下文同。——译者注

或遍历时做了修改，那就必须复制整个数组。

也就是说如果对列表的修改次数跟读取次数相差不多，这种方式未必能达到较好的性能。但就像我们在第6章一再提到的那样，得到性能优异的代码的唯一可靠的方法就是测试，再测试，并衡量结果。

下一个在并发代码中常用的构件是java.util.concurrent中的Queue。它用于在线程之间切换工作元素，并且还是很多灵活可靠的多线程设计的基础。

4.3.6 Queue

队列是一个非常美妙的抽象概念。不，之所以这么说并不是因为我们生活在伦敦这个世界排队之都。为把处理资源分发给工作单位（或者把工作单元分配给处理资源，这取决于你看待问题的方式），队列提供了一种简单又可靠的方式。

Java中有些多线程编程模式在很大程度上都依赖于Queue实现的线程安全性，所以很有必要充分认识它。Queue接口被放在了java.util包中，因为即便在单线程编程中它也是一个重要的模式，但我们的重点是多线程编程，并且假定你已经熟悉队列的基本用法了。

队列经常用来在线程之间传递工作单元，这个模式通常适合用Queue最简单的并发扩展BlockingQueue来实现。接下来我们就会重点介绍它。

1. BlockingQueue

BlockingQueue还有两个特性。

- 在向队列中put()时，如果队列已满，它会让放入线程等待队列腾出空间。
- 在从队列中take()时，如果队列为空，会导致取出线程阻塞。

这两个特性非常有用，因为如果一个线程（或线程池）的能力超过了其他线程，比较快的线程就会被强制等待，因此可以对整个系统起到调节作用，如图4-9所示。

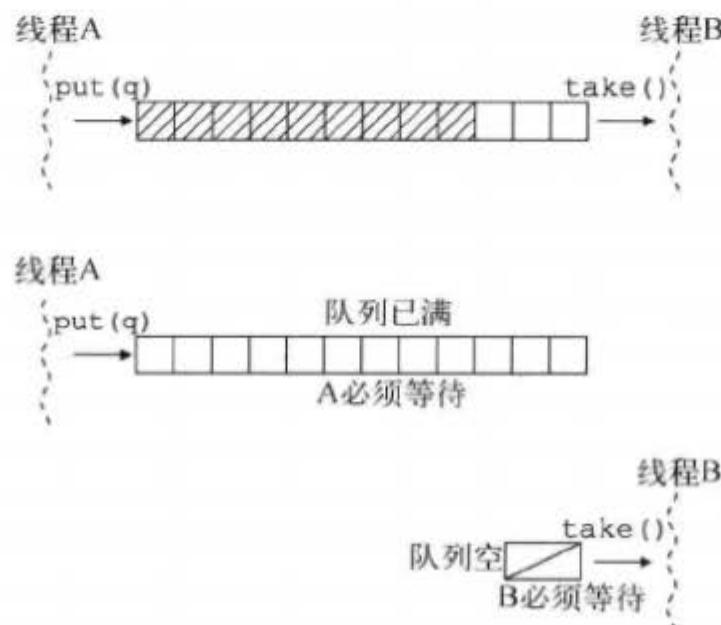


图4-9 BlockingQueue

BlockingQueue的两个实现

Java 提供了 BlockingQueue 接口的两个基本实现： LinkedBlockingQueue 和 ArrayBlockingQueue。它们的特性稍有不同；比如说，在已知队列的大小而能确定合适的边界时，用 ArrayBlockingQueue 非常高效，而 LinkedBlockingQueue 在某些情况下则会快一点儿。

2. 使用工作单元

Queue 接口全都是泛型的——它们是 Queue<E>， BlockingQueue<E>，等等依此类推。尽管看起来奇怪，但有时候利用这一点把工作项封装在一个人工容器类内却是明智之举。

比如说，你有一个表示工作单元的 MyAwesomeClass 类，想要用多线程方式处理，与其用 BlockingQueue<MyAwesomeClass> 不如用 BlockingQueue<WorkUnit<MyAwesomeClass>>。其中 WorkUnit（或 QueueObject，或随你怎么命名这个容器类）是像下面这样的包装接口或类：

```
public class WorkUnit<T> {
    private final T workUnit;
    public T getWork() { return workUnit; }
    public WorkUnit(T workUnit_) {
        workUnit = workUnit_;
    }
}
```

有了这层间接引用，不用牺牲所包含类型（在此即 MyAwesomeClass）在概念上的完整性就可以在这里添加额外的元数据了。

这特别有用。能用上额外元数据的用例很多，下面举几个例子：

- 测试（比如展示一个对象的修改历史）
- 性能指标（比如到达时间或服务质量）
- 运行时系统信息（比如 MyAwesomeClass 实例是如何被排到队列中的）

以后再在这种间接引用里增加元数据可能会非常困难。如果你发现在某些情况下需要更多的元数据，那么要把它们加入到间接引用中可能需要大量的重构工作，而加在 WorkUnit 类中就只是个简单的修改。

3. 一个 BlockingQueue 的例子

我们用一个简单的例子——等着看医生的宠物们——来看看如何使用 BlockingQueue。这个例子中有一个等着让医生给做检查的宠物集合。

代码清单4-11 在Java中对宠物建模

```
public abstract class Pet {
    protected final String name;
    public Pet(String name) {
        this.name = name;
    }
}
```

```

    public abstract void examine();
}

public class Cat extends Pet {
    public Cat(String name) {
        super(name);
    }
    public void examine() {
        System.out.println("Meow!");
    }
}

public class Dog extends Pet {
    public Dog(String name) {
        super(name);
    }
    public void examine() {
        System.out.println("Woof!");
    }
}

public class Appointment<T> {
    private final T toBeSeen;

    public T getPatient(){ return toBeSeen; }

    public Appointment(T incoming) {
        toBeSeen = incoming;
    }
}

```

在这个简单的例子中，我们用`LinkedBlockingQueue<Appointment<Pet>>`表示兽医的候诊队列，`Appointment`起到了`WorkUnit`的作用。

兽医对象是由一个队列和一个暂停时间构建的，其中队列是由一个代表接待员的对象提供的预约队列，暂停时间表示兽医在预约之间的停工时间。

我们可以在下面这段代码中建立兽医的模型。在线程运行时，它在一个无限循环中重复调用`seePatient()`。当然，现实世界中的兽医不可能这样，因为他们晚上和周末要回家，不能一直在办公室等着生病的小动物上门就医。

代码清单4-12 对兽医建模

```

public class Veterinarian extends Thread {
    protected final BlockingQueue<Appointment<Pet>> appts;
    protected String text = "";
    protected final int restTime;
    private boolean shutdown = false;

    public Veterinarian(BlockingQueue<Appointment<Pet>> lbq, int pause) {
        appts = lbq;
        restTime = pause;
    }

    public synchronized void shutdown(){
        shutdown = true;
    }
}

```

```

@Override
public void run() {
    while (!shutdown) {
        seePatient();
        try {
            Thread.sleep(restTime);
        } catch (InterruptedException e) {
            shutdown = true;
        }
    }
}

public void seePatient() {
    try {
        Appointment<Pet> ap = appts.take(); ← 阻塞take
        Pet patient = ap.getPatient();
        patient.examine();
    } catch (InterruptedException e) {
        shutdown = true;
    }
}
}

```

在seePatient()方法中，线程会从队列中取出预约，并挨个检查对应的宠物，如果当前队列中没有预约等待，则会阻塞。

4. BlockingQueue的细粒度控制

除了简单的take()和offer()API，BlockingQueue还提供了另外一种与队列交互的方式，这种方式对队列的控制力度更大，但稍微有点复杂。这就是带有超时的放入或取出操作，它允许线程在遇到问题时可以从与队列的交互中退出来，转而做点儿其他的事情。

实际上，这个功能并不常用，但它偶尔能派上大用场，所以我们要介绍一下。下面的例子还是来自微博。

代码清单4-13 BlockingQueue行为的例子

```

public abstract class MicroBlogExampleThread extends Thread {
    protected final BlockingQueue<Update> updates;
    protected String text = "";
    protected final int pauseTime;
    private boolean shutdown = false;

    public MicroBlogExampleThread(BlockingQueue<Update> lbq_, int pause_) {
        updates = lbq_;
        pauseTime = pause_;
    }

    public synchronized void shutdown() {
        shutdown = true; ← 使线程可以彻底地结束
    }

    @Override
    public void run() {
        while (!shutdown) {
            doAction();
        }
    }
}

```

```

try {
    Thread.sleep(pauseTime);
} catch (InterruptedException e) {
    shutdown = true;
}
}

public abstract void doAction();
}

final Update.Builder ub = new Update.Builder();
final BlockingQueue<Update> lbq = new LinkedBlockingQueue<>(100);

MicroBlogExampleThread t1 = new MicroBlogExampleThread(lbq, 10) {
    public void doAction(){
        text = text + "X";
        Update u = ub.author(new Author("Tallulah")).updateText(text).build();
        boolean handed = false;
        try {
            handed = updates.offer(u, 100, TimeUnit.MILLISECONDS);
        } catch (InterruptedException e) {
        }
        if (!handed) System.out.println(
            "Unable to hand off Update to Queue due to timeout");
    }
};

MicroBlogExampleThread t2 = new MicroBlogExampleThread(lbq, 1000) {
    public void doAction(){
        Update u = null;
        try {
            u = updates.take();
        } catch (InterruptedException e) {
            return;
        }
    }
};

t1.start();
t2.start();

```

运行这段代码展示了填充队列的速度有多么快，也表明供给线程的速度超过了提取线程的速度。很快，“Unable to hand off Update to Queue due to timeout”消息就出现了。

这是“相连线程池”中的一种典型的极端状况，当上游的线程池比下游的快，这种情况就会发生。“相连线程池”可能会引发一些问题，比如会导致`LinkedBlockingQueue`溢出。另外，如果消费者比生产者多，队列会因此而经常空着。好在Java 7在`BlockingQueue`上有了解决办法——`TransferQueue`。

5. TransferQueue——Java 7中的新贵

Java 7引入了`TransferQueue`。它本质上是多了一项`transfer()`操作的`BlockingQueue`。如果接收线程处于等待状态，该操作会马上把工作项传给它。否则就会阻塞直到取走工作项的线程出现。你可以把这看做“挂号信”选项，即正在处理工作项的线程在交付当前工作项之前不会开始其他工作项的处理工作。这样系统就可以调控上游线程池获取新工作项的速度。

用限定大小的阻塞队列也能达到这种调控效果，但TransferQueue接口更灵活。此外，用TransferQueue取代BlockingQueue的代码性能表现可能会更好。这是因为编写TransferQueue的实现时已经将现代编译器和处理器的特性考虑在内，执行起来效率更高。聊了这么久性能，不能空口无凭，必须给出测量结果并能证明才行。另外你也应该意识到，Java 7只给出了TransferQueue的一种实现形式——链表版。

在下面的例子中，你会发现用TransferQueue代替BlockingQueue是多么简单。只要对清单4-13中的代码做些简单修改，就可以升级成TransferQueue，请看这里。

代码清单4-14 用TransferQueue代替BlockingQueue

```
public abstract class MicroBlogExampleThread extends Thread {
    protected final TransferQueue<Update> updates;
    ...
    public MicroBlogExampleThread(TransferQueue<Update> lbq_, int pause_) {
        updates = lbq_;
        pauseTime = pause_;
    }
    ...
}
final TransferQueue<Update> lbq = new LinkedTransferQueue<Update>(100);
MicroBlogExampleThread t1 = new MicroBlogExampleThread(lbq, 10) {
    public void doAction() {
        ...
        try {
            handed = updates.tryTransfer(u, 100, TimeUnit.MILLISECONDS);
        } catch (InterruptedException e) {
        }
        ...
    }
};
```

到此为止，用来开发多线程应用的主要构件我们都见识过了。接下来该把它们整合到驱动并发代码的引擎（执行器框架）上了。用它们可以对任务进行调度和控制，可以组合高效的并发流处理工作项，从而构建大型多线程应用程序。

4.4 控制执行

我们在前面的讨论中一直把工作任务当成抽象的单元。然而有个细节需要注意，我们一直没有提到的是这些单元要比Thread小——它们提供的方法把计算任务包含在一个工作单元中，无需为每个单元启动新的线程。这样处理多线程代码通常效率更高，因为免除了为每个单元启动Thread的开销。执行代码的线程是重用的，处理完一个任务后会继续处理新的工作单元。

虽然复杂一些，但你可以实现线程池、工人与管理者模式和执行者等开发人员最常用的模式。我们接下来要密切关注可以对任务（Callable、Future和FutureTask）和执行者建模的类和接口，特别是ScheduledThreadPoolExecutor。

4.4.1 任务建模

我们的终极目标是不用为调度每个任务或工作单元而启动新线程。归根结底，就是要把它们做成可以调用（通常由执行者调用）的代码，而不是直接可运行的线程。

我们来看对任务建模的三种办法——`Callable`和`Future`接口以及`FutureTask`类。

1. Callable接口

`Callable`接口是一个非常常见的概念，代表了一段可以调用并返回结果的代码。尽管这种做法很直接，但实际上它的作用微妙而又强大，用它可以创建出一些特别实用的模式。

`Callable`的典型用法是匿名实现类。这段代码的最后一行把`s`赋值为`out.toString()`：

```
final MyObject out = getSampleObject();

Callable<String> cb = new Callable<String>() {
    public String call() throws Exception {
        return out.toString();
    }
};
String s = cb.call();
```

可以把`Callable`的匿名实现类当做对单一抽象方法`call()`的递延调用，该实现必须提供这个方法。

`Callable`是SAM类型（“单一抽象方法”的缩写，有时会这样称呼它）的示例——这是Java 7把函数作为一等类型最可行的办法。在后续章节讨论非Java语言时还会遇到它们，那时我们还会进一步讨论把函数作为值或一等类型的概念。

2. Future接口

`Future`接口用来表示异步任务，是还没有完成的任务给出的未来结果。我们在第2章介绍NIO.2和异步I/O时提过。

下面是`Future`中的主要方法。

- `get()`——用来获取结果。如果结果还没准备好，`get()`会被阻塞直到它能取得结果。还有一个可以设置超时的版本，这个版本永远不会阻塞。
- `cancel()`——在运算结束前取消。
- `isDone()`——调用者用它来判断运算是否结束。

下面这段代码（找素数）展示了`Future`的用法：

```
Future<Long> fut = getNthPrime(1_000_000_000);

Long result = null;
while (result == null) {
    try {
        result = fut.get(60, TimeUnit.SECONDS);
    } catch (TimeoutException tox) { }
    System.out.println("Still not found the billionth prime!");
}
System.out.println("Found it: " + result.longValue());
```

在这段代码中，你应该想象一下返回Future的getNthPrime()在某个后台线程或多个线程上运行的情景，也有可能是在执行者框架上运行。即便使用先进的硬件，这种运算可能也需要很长时间——你最后还是要用Future的cancel()方法。

3. FutureTask类

FutureTask是Future接口的常用实现类，它也实现了Runnable接口。这意味着FutureTask可以由执行者调度，这一点很关键。它对外提供的方法基本上就是Future和Runnable接口的组合：get()、cancel()、isDone()、isCancelled()和run()，最后一个方法通常都是由执行者调用，你基本不需要直接调用它。

FutureTask还提供了两个很方便的构造器：一个以Callable为参数，另一个以Runnable为参数。这些类之间的关联表明对于任务建模的办法非常灵活，允许你基于FutureTask的Runnable特性（因为它实现了Runnable接口），把任务写成Callable，然后封装进一个由执行者调度并在必要时可以取消的FutureTask。

4.4.2 ScheduledThreadPoolExecutor

ScheduledThreadPoolExecutor（以下简称STPE）是线程池类中的重中之重——它功能多样，广受欢迎。STPE接收任务，并把它们安排给线程池里的线程。

- 线程池的大小可以预定义，也可自适应。
- 所安排的任务可以定期执行，也可只运行一次。
- STPE扩展了ThreadPoolExecutor类（很相似，但不具备定期调度能力）。

和java.util.concurrent中的工具类相结合的STPE线程池是大中型多线程应用程序最常见的模式之一，这些工具类包括我们在前面已经见过的ConcurrentHashMap、CopyOnWriteArrayList和BlockingQueue等。

STPE不过是通过Executors类的工厂方法轻易获取的众多执行者之一。使用这些工厂方法很方便，开发人员通过它们可以轻易获取典型配置，需要时还可以开放完整的接口方法。

下面的代码是一个定期读取的例子。这是newScheduledThreadPool()的常见用法：msgReader对象被安排poll()一个队列，从队列中的WorkUnit对象里取得工作项，然后输出。

代码清单4-15 STPE定期读取

```
private ScheduledExecutorService stpe;
private ScheduledFuture<?> hndl;                                取消时需要

private BlockingQueue<WorkUnit<String>> lbq = new LinkedBlockingQueue<>();

private void run(){
    stpe = Executors.newScheduledThreadPool(2);                  执行者的工厂方法
    final Runnable msgReader = new Runnable(){
        public void run(){
            String nextMsg = lbq.poll().getWork();
            if (nextMsg != null) System.out.println("Msg recv'd: "+ nextMsg);
        }
    };
    hndl = stpe.scheduleAtFixedRate(msgReader, 0, 1, TimeUnit.SECONDS);
}
```

```

    }
};

hndl = stpe.scheduleAtFixedRate(msgReader, 10, 10,
    TimeUnit.MILLISECONDS);
}

public void cancel() {
    final ScheduledFuture<?> myHndl = hndl;
    stpe.schedule(new Runnable() {
        public void run() { myHndl.cancel(true); }           ← 取消时需要
    }, 10, TimeUnit.MILLISECONDS);
}

```

在这个例子中，STPE每隔10毫秒就唤醒一个线程，让它尝试poll()一个队列。如果读取返回null（因为队列当前为空），则什么也不会发生，线程回去继续睡大觉。如果收到了一个工作单元，则线程会输出该工作单元的内容。

用Callable调用的代表性问题

形式简单的Callable、FutureTask及相关类存在几个问题——尤其在涉及类型系统时。要明白这一点，可以想想怎么才能满足一个未知方法可能出现的所有方法签名。Callable只能用于没有参数的方法。要满足所有可能性，你需要Callable的不同变体。

在Java中，你可以通过指定模型系统内的方法签名来解决这个问题。但你在本书第三部分会见到，动态语言不能用这种静态视图来约束。我们将会返回来重点讨论这种类型系统之间的不匹配。现在你只要注意到，虽然Callable很有用，但要用它构建一个通用框架来对线程执行进行建模还是有点儿限制得太死了。

现在我们要转向Java 7重点突出的框架之一——用于轻量级并发的分支/合并（fork/join）框架。这个框架比我们在本节中见到的执行者在处理并发问题方面更加高效，要达到这点绝非易事。

4.5 分支/合并框架

就像第6章要讨论的一样，处理器的速度（或者更准确地说是CPU上的晶体管数量）最近几年增长迅猛。由此产生的副作用就是处理器等待I/O操作变成了家常便饭。这表明我们能够更好地利用计算机的处理能力。分支/合并框架就可以解决这个问题——这也是Java 7对并发领域新做出的最大贡献。

分支/合并框架完全是为了实现线程池中任务的自动调度，并且这种调度对用户来说是透明的。为了达到这种效果，必须按用户指定的方式对任务进行分解。在很多应用程序中，对于分支/合并框架来说都可以很自然地把其中的任务分成“小型”和“大型”任务。

我们来快速浏览一些与分支/合并框架相关的重要事实和基本原理。

- 引入了一种新的执行者服务，称为ForkJoinPool。
- ForkJoinPool服务处理一种比线程“更小”的并发单元ForkJoinTask。

- ForkJoinTask是一种由ForkJoinPool以更轻量化的方式所调度的抽象。
- 通常使用两种任务（尽管两种都表示为ForkJoinTask实例）：
 - “小型”任务是那种无需处理器耗费太多时间就可以直接执行的任务。
 - “大型”任务是那种需要在直接执行前进行分解（还可能不止一次）的任务。
- 提供了支持大型任务分解的基本方法，它还有自动调度和重新调度的能力。

这个框架的关键特性之一就是这些轻量的任务都能够生成新的ForkJoinTask实例，而这些实例将仍由执行它们父任务的线程池来安排调度。这就是分而治之。

我们会通过一个简单的例子告诉你如何使用分支/合并框架，然后简短介绍“工作窃取”这个特性，最后讨论一下那些适用于并行处理技术的特性。使用分支/合并框架最好的办法就是从例子入手。

4.5.1 一个简单的分支/合并例子

我们为说明分支/合并框架而设定了这样的应用场景：有一个数组，里面存放不同时间到达的微博更新，我们想按到达时间对它们排序，以便为用户生成时间线，就像在代码清单4-9中生成的那个一样。

我们会用MergeSort的变体实现多线程排序。代码清单4-16中用到了ForkJoinTask的特定子类RecursiveAction。因为它明显可以独立完成任务（对这些更新的排序能当即完成），而且具备递归处理能力（递归特别适合做排序），所以用RecursiveAction会比用通用的ForkJoinTask更简单。

MicroBlogUpdateSorter类用Update对象的compareTo()方法对更新列表排序。compute()方法（超类RecursiveAction中的抽象方法，必须实现）基本上是按创建时间对微博更新数组排序。

代码清单4-16 用RecursiveAction排序

```
public class MicroBlogUpdateSorter extends RecursiveAction {
    private static final int SMALL_ENOUGH = 32;
    private final Update[] updates;
    private final int start, end;
    private final Update[] result;

    public MicroBlogUpdateSorter(Update[] updates_) {
        this(updates_, 0, updates_.length);
    }

    public MicroBlogUpdateSorter(Update[] upds_,
        int startPos_, int endPos_) {
        start = startPos_;
        end = endPos_;
        updates = upds_;
        result = new Update[updates.length];
    }

    private void merge(MicroBlogUpdateSorter left_,
        MicroBlogUpdateSorter right_, int start_, int end_)
```



串行排序项只有32个或更少

```

    ↪ MicroBlogUpdateSorter right_) {
        int i = 0;
        int lCt = 0;
        int rCt = 0;
        while (lCt < left_.size() && rCt < right_.size()) {
            result[i++] = (left_.result[lCt].compareTo(right_.result[rCt]) < 0)
                ? left_.result[lCt++]
                : right_.result[rCt++];
        }
        while (lCt < left_.size()) result[i++] = left_.result[lCt++];
        while (rCt < right_.size()) result[i++] = right_.result[rCt++];
    }

    public int size() {
        return end - start;
    }

    public Update[] getResult() {
        return result;
    }

    @Override
    protected void compute() {
        if (size() < SMALL_ENOUGH) {
            System.arraycopy(updates, start, result, 0, size());
            Arrays.sort(result, 0, size());
        } else {
            int mid = size() / 2;
            MicroBlogUpdateSorter left = new MicroBlogUpdateSorter(
                ↪ updates, start, start + mid);
            MicroBlogUpdateSorter right = new MicroBlogUpdateSorter(
                ↪ updates, start + mid, end);
            invokeAll(left, right);
            merge(left, right)
        }
    }
}

```

RecursiveAction
中声明的方法

要使用这个排序器，你可以用下面这样的代码驱动它，生成一些包含由X组成的字符串的更新，并打乱它们的顺序，之后再传给排序器。最终得到重新排序后的更新。

代码清单4-17 使用递归排序器

```

List<Update> lu = new ArrayList<Update>();
String text = "";
final Update.Builder ub = new Update.Builder();
final Author a = new Author("Tallulah");

for (int i=0; i<256; i++) {
    text = text + "X";
    long now = System.currentTimeMillis();
    lu.add(ub.author(a).updateText(text).createTime(now).build());
    try {
        Thread.sleep(1);
    }
}

```

```

    } catch (InterruptedException e) {
    }
}
Collections.shuffle(lu);
Update[] updates = lu.toArray(new Update[0]);
MicroBlogUpdateSorter sorter = new MicroBlogUpdateSorter(updates);
ForkJoinPool pool = new ForkJoinPool(4);
pool.invoke(sorter);

for (Update u: sorter.getResult()) {
    System.out.println(u);
}

```

传入空数组，
省掉空间分配

TimSort

随着Java 7的到来，默认的数组排序算法已经变了。以前是以QuickSort的形式，但到了Java 7时代则变成了“TimSort”——MergeSort和插入排序的混合体。TimSort最初是Tim Peters为Python开发的，而且从2.3版（2002）开始就是Python中的默认排序算法了。

如果想看看TimSort在Java 7中存在的证据，可以给清单4-16中的代码传入一个null数组。对数组排序时，由于数组尺寸太小，会调用Array.sort()方法，该方法会抛出空指针异常，在输出的异常信息里就能看到TimSort类。

4

4.5.2 ForkJoinTask 与工作窃取

ForkJoinTask是RecursiveAction的超类。它是从动作中返回结果的泛型类型，所以RecursiveAction扩展了ForkJoinTask<Void>。这使得ForkJoinTask非常适合用MapReduce^①方式返回数据集中归结出的结果。

ForkJoinTasks由ForkJoinPool调度安排，ForkJoinPool是专为这些轻量任务设计的新型执行者服务。该服务维护每个线程的任务列表，并且当某个任务完成时，它能把挂在满负荷线程上的任务重新安排到空闲线程上去。

采用这种“工作窃取”的算法是为了解决大小不同的任务所导致的调度问题。大小不同的任务所需的运行时间通常也会有很大差别。比如说，某个线程的运行队列中都是小任务，而另外一个全是大任务。如果小任务的运行速度比大任务快五倍，只处理小任务的线程很可能在处理大任务的线程完成之前就处于空闲状态了。

Java 7实现的工作窃取机制精准地解决了这个问题，并且在分支/合并框架工作的整个生命周期中使线程池中的所有线程都有用武之地。工作窃取完全是自动的，你什么也不用做就能享受到它带来的好处。不需要手工干预，而是由运行环境承担更多工作帮助开发人员管理并发，这在Java 7中已经不是什么新鲜事了。

^① MapReduce是Google提出的一个软件架构，用于大规模数据集（大于1TB）的并行运算。当前的软件实现是指定一个Map（映射）函数，用来把一组键值对映射成一组新的键值对，指定并发的Reduce（化简）函数，用来保证所有映射的键值对中的每一个元素都共享相同的键组。——译者注

4.5.3 并行问题

分支/合并框架的确对我们的帮助很大，但在实际中，并不是每个问题都能像4.5.1节中那样轻易地简化成多线程MergeSort。

这里是一些可以用分支/合并方法解决的问题：

- 模拟大量简单对象的运动，比如粒子效果；
- 日志文件分析；
- 从输入中计数的数据操作，比如mapreduce操作。

从另一个角度来说，图4-10中这个被分解的问题正是分支/合并框架可以解决的。

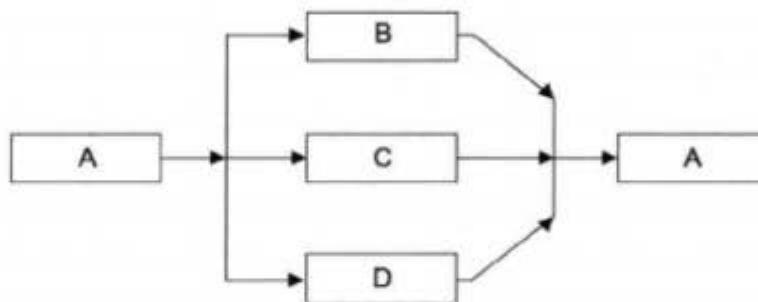


图4-10 分支与合并

用下面这个列表检查问题及其子任务是一个切实有效的方法，它可以确定是否能用分支/合并来解决这个问题。

- 问题的子任务是否无需与其他子任务有显式的协作或同步也可以工作？
- 子任务是不是不会对数据进行修改，只是经过计算得出些结果（它们是不是函数程序员称为“纯粹的”函数的函数）？
- 对于子任务来说，分而治之是不是很自然的事？子任务是不是会创建更多的子任务，而且它们要比派生出它们的任务粒度更细？

对于前面这些提问，如果答案是肯定的，或者“大体如此，但有临界情况”，那你解决问题很可能适合用分支/合并的方式解决。反过来，如果答案是“可能吧”或者“算不上”，你就会发现分支/合并帮不上什么忙，可能用其他的同步方式更合适。

注意 前面的检查列表是测试某个问题（比如在Hadoop和NoSQL数据库中常见的那种）能否很好地用分支/合并方式解决的有效方法。

想设计出优秀的多线程算法并不容易，分支/合并方法也不能面面俱到。在适用的领域，它的用处很广。其实归根结底，你必须要确定你的问题是否适合这个框架，如果不适合，你只能在性能卓越的java.util.concurrent工具箱上构建自己的解决方案。

在下一节中，我们会详细讨论经常被误解的Java内存模型（Java Memory Model，JMM）。很多Java程序员都知道JMM，并且在没有经过正式介绍的情况下按自己的理解写代码。如果你觉得

这是在说你，那么接下来的内容会帮助你重新认识JMM，并且帮你打下扎实的基础。JMM这个话题相当有深度，所以如果你急着进入下一章，可以先跳过它。

4.6 Java 内存模型

Java语言规范（JLS）在第17.4节中介绍了JMM。其中的描述非常正式，用同步动作和被称为偏序^①的数学结构描述JMM。这对于语言理论学家和Java规范的实现者（编译器和虚拟机的制造者）来说非常棒，但对于需要理解多线程代码如何执行的应用开发人员来说，这种描述会让他们头昏脑胀。

我们在这里不重复规范里的正式描述，而是用两个基本概念列出最重要的规则，这两个概念是代码块之间的之前发生（Happens-Before）和同步约束（Synchronizes-With）关系。

□ 之前发生——这种关系表明一段代码块在其他代码开始之前就已经全部完成了。

□ 同步约束——这意味着动作继续执行之前必须把它的对象视图与主内存进行同步。

如果你认真研究过OO编程，应该听到过面向对象构件的Has-A和Is-A这两种表述方式。一些开发人员发现，用之前发生和同步约束来描述基本的概念构件对理解Java并发很有帮助。这和Has-A与Is-A的道理一样，但这两组概念在技术上没有直接关联。

图4-11中是一个易失性写入与后续的读取访问（用于`println`）之间同步约束的例子。

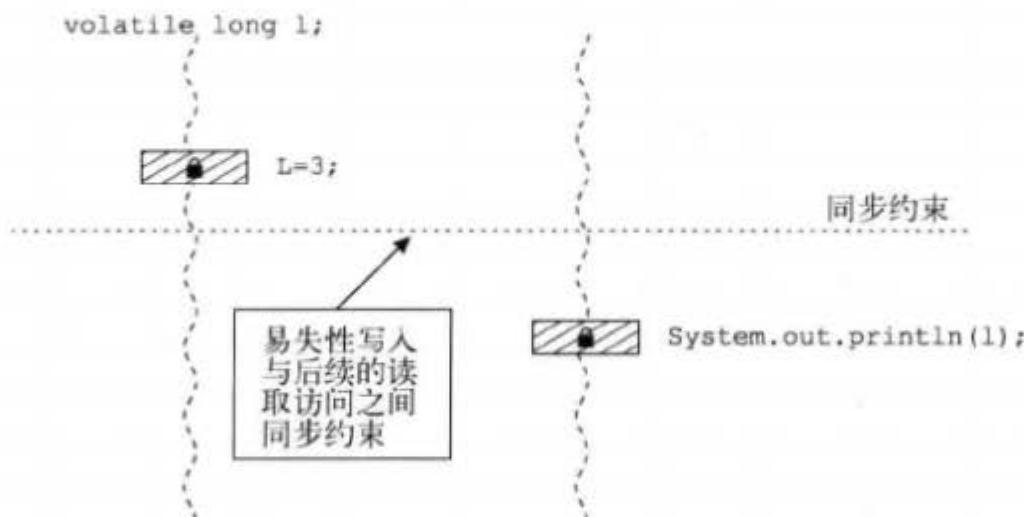


图4-11 同步约束的例子

JMM的主要规则如下。

- 在监测对象上的解锁操作与后续的锁操作之间存在同步约束关系。
- 对易失性（volatile）变量的写入与后续对该变量的读取之间存在同步约束关系。
- 如果动作A受到动作B的同步约束，则A在B之前发生。
- 如果在程序中的线程内A出现在B之前，则A在B之前发生。

^① 设A是一个非空集，P是A上的一个关系，若关系P是自反的（对任意的 $a \in A$, $(a, a) \in P$ ）、反对称的（若 $(a, b) \in P$ 且 $(b, a) \in P$, 则 $a=b$ ）和传递的（若 $(a, b) \in P$, $(b, c) \in P$, 则 $(a, c) \in P$ ），则称P是集合A上的偏序关系。比如实数集上的小于等于关系（ $a \leq a$; $a \leq b$, $b \leq a$, 则 $a=b$; $a \leq b$, $b \leq c$, 则 $a \leq c$ ）。——译者注

前两个规则通俗来说就是“先放后取”。换句话说，一个线程在写入时持有的锁要在其他操作（包括读取）能够获取锁之前被释放掉。

这里还有些规则，实际上是关于敏感行为的。

- 构造方法要在那个对象的终结器开始运行之前完成（一个对象被终结之前必须已经构造完整）。
- 开始一个线程的动作受到这个新线程的第一个动作的同步制约。
- `Thread.join()`受到被合并的线程的最后一个（和其他全部）动作的同步制约。
- 如果X在Y之前发生，并且Y在Z之前发生，则X在Z之前发生（传递性）。

这些简单的规则定义了内存和同步如何工作的全平台视图。图4-12展示了传递性规则。

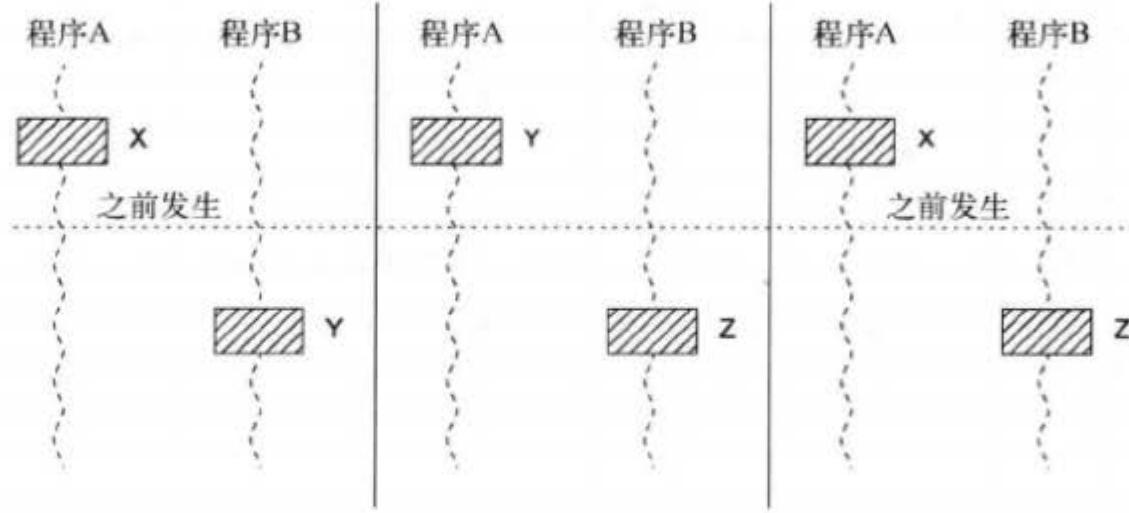


图4-12 之前发生的传递性

注意 实际上，这些规则是JMM做出的最低保证。真正的JVM实际上可能表现得更好。对于开发人员来说，这可能是个陷阱，因为某个特定JVM中的行为实际上是个隐藏在底层并发中的诡异bug，却很容易给人造成错觉，以为是它提供的安全特性。

从这些最低保证中，很容易可以看出不可变性成为Java并发编程中的一个重要概念的原因。如果对象不可改变，确保改变对所有线程可见的相关问题就不会出现。

4.7 小结

并发是Java平台最重要的特性之一，扎实的并发编程知识对于一个优秀的开发人员来说日益重要。我们回顾了Java并发的基础和多线程系统的设计原则，并讨论了Java内存模型和Java平台如何实现并发的底层细节。

更重要的是我们解释了`java.util.concurrent`中的那些类和接口，现代Java开发人员在编写新的多线程代码时，更喜欢用到这些工具。我们还向你详细介绍了Java 7中一些新的类，如`LinkedTransferQueue`和分支/合并框架。

希望我们已经为你打好了基础，使你能够用java.util.concurrent中的类编写代码。这是本章内容的重中之重。尽管我们也探讨了一些核心理论，但最重要的还是实际样例。哪怕你刚开始使用ConcurrentHashMap和Atomic类，也能马上见识到这些经过严格测试的类所带来的好处。

时间到了，我们马上就要进入下一主题——一个能让你从Java开发者中脱颖而出的重要主题。在下一章，你会在Java平台的另一个基础领域（类加载和字节码）打下坚实的基础。这一领域是很多讨论平台安全和性能特性内容的核心，并巩固了生态系统内的很多先进技术。所以对于想鹤立鸡群的开发人员来说，这是个绝佳的研究课题。

本章内容

- 类加载
- 方法句柄
- 解剖类文件
- JVM字节码以及它的重要性
- 新的操作码invokedynamic

要成为优秀的Java开发人员，需要深入理解Java平台的工作方式。其中就包括类加载和JVM字节码这样的核心特性。

假设有一个大量使用依赖注入（DI）技术的应用程序，比如Spring，它在启动时出了问题，报了一个神秘的错误信息。如果不是简单的配置错误问题，你就需要了解如何实现DI框架才能跟踪问题来源。也就是说你得明白类加载机制。

或者假定跟你合作的开发商跑路了，只给你留下了一堆编译过的代码，没有源码，文档也乱七八糟的。你怎么能知道编译过的代码包含了什么呢？

最常见的程序启动失败错误就是ClassNotFoundException或NoClassDefFoundError，但很多开发人员都不知道它们是什么，有什么区别以及为什么会出现。

本章重点就是这些与开发相关的平台特性。此外还会讨论一些高级特性——它们是为Java的粉丝准备的，如果你时间有限，可以跳过那部分内容。

我们会从类加载的概览开始，这是VM为运行中的程序定位和激活新类型的过程。其核心是在VM中表示类型的class对象。接下来我们会介绍一下新的方法句柄API，并和Java 6中已有的技术（比如反射）进行比较。

之后我们会讨论检查和分析类文件的工具。用Oracle JDK提供的javap作为参考工具。上过类文件的解剖课后，我们会转而讨论字节码，其中涉及JVM操作码的主要体系以及运行时的底层操作。

在你用字节码的知识把自己武装起来之后，我们会深入探讨invokedynamic操作码，它是Java 7新引入进来的，为的是让非Java语言能充分利用JVM的平台特性。

我们先从类加载开始吧，这是一个将新的类合并到正在运行着的JVM进程中的过程。

5.1 类加载和类对象

一个.class文件定义了JVM中的一个类型，包括域、方法、继承信息、注解和其他元数据。规范中对类文件的格式有详细描述，任何想在JVM上运行的语言都必须遵守。

类是平台能加载的最小程序代码单元。要将新的类加入到JVM的当前运行态中，有几步操作必须执行。首先，类文件必须被加载进来并连接，而且必须进行大量的验证。之后会提供一个代表该类型的新Class对象给正在运行的系统，并可以创建新的实例。

本节会讨论所有这些步骤，并介绍一下类加载器，也就是控制整个过程的那些类。我们先来看看加载和连接。

5.1.1 加载和连接概览

JVM的目的是使用类文件并执行其中的字节码。要实现这个目的，JVM必须以字节数据流的方式取出类文件中的内容，并将其转换成可用的格式加入运行态中。这个分两步走的过程被称为加载和连接，但连接又会被分解为几个子阶段。

加载

这个过程首先要读取构成类文件的字节数据流并给类的表现形式解冻。该过程一开始是创建一个字节数组，其内容通常是从文件系统中读取的，然后产生与所加载的类对应的Class对象。在这个过程中会对类做一些基本检查，但在加载过程结束时，Class对象还不成熟，所以类也不可用。

连接

加载完成之后，类必须被连接起来。这一步骤分为三个子阶段——验证、准备和解析。验证阶段证实类文件符合预期，不会引起系统的运行时错误或其他问题。之后是类的准备阶段，在类文件中引用的其他类型全部都要定位到，以确保该类已准备就绪。

连接步骤中各子阶段之间的相互关系如图5-1所示。

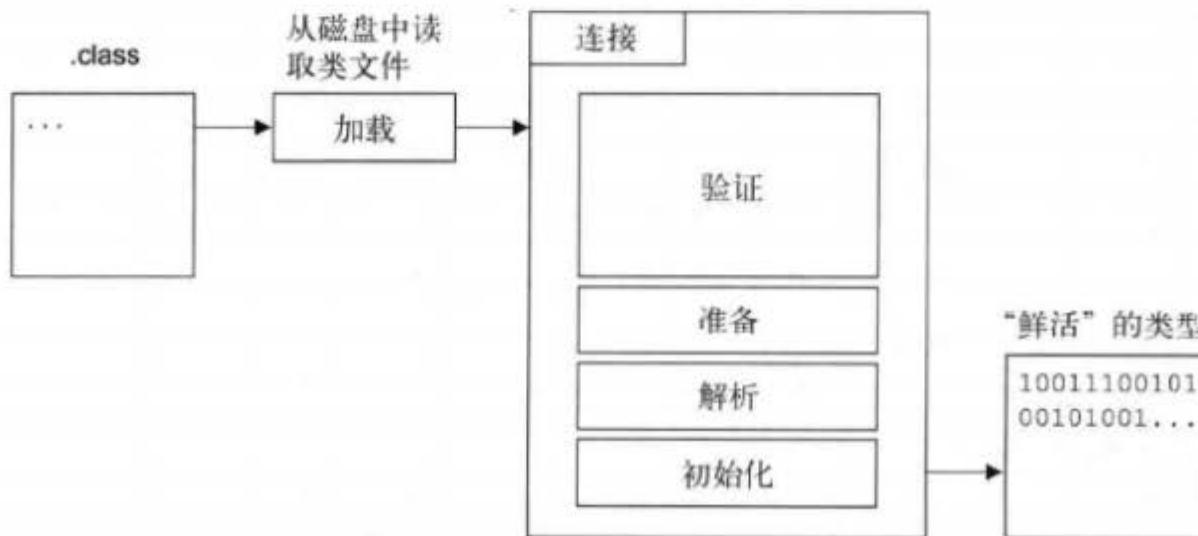


图5-1 加载与连接（及连接的子阶段）

5.1.2 验证

验证是一个非常复杂的过程，它分为几个步骤。

首先是完整性检查。这实际上是加载过程中的一部分，会确保类文件格式良好，可以连接。

接着是检查常量池（详情参见5.3.3节）中的符号信息是自相一致的，并要遵守常量的基本行为准则。其他不涉及代码的静态检查也要在这一阶段完成，比如检查final方法有没有被重写。

之后是验证中最复杂的部分——方法的字节码检查。要检查字节码行为良好，并且不会试图摆脱VM的环境控制。下面是一些主要检查。

- 是否所有方法都遵守访问控制关键字的限定。
- 方法调用的参数个数和静态类型是否正确。
- 确保字节码不会试图滥用堆栈。
- 确保变量使用之前被正确初始化了。
- 检查变量是否仅被赋予恰当类型的值。

做这些检查是出于性能方面的考虑，这样可以加快解释码的运行速度，运行时就不用再做这些检查了。同时还简化了运行时把字节码编译为机器码的过程（即时编译，详情参见6.6节）。

准备

类的准备包括分配内存和准备好初始化类中的静态变量，但不会现在初始化变量，也不会执行任何VM字节码。

解析

解析会促使VM检查类文件中所引用的类型是不是都是已知的类型。如果有运行时未知的类型，那它们也需要被加载进来。这些可见的未知类型会再次引发类加载过程。

一旦需要加载的其他类型全被定位并解析完成，VM就可以初始化那个最初要加载的类。这时所有静态变量都可以被初始化，所有静态初始化代码块都会运行。现在你运行的字节码就是来自新加载进来的类里的。这一步完成之后，类的加载就已全部完成，类也就可以使用了。

5.1.3 Class 对象

连接和加载过程的最终结果是一个Class对象，用于表示加载并连接起来的新类型。尽管出于性能方面的考虑，Class对象只是在要求的地方做了初始化，但现在它在VM中完全生效了。代码可以继续执行了，它可以使用新类型并创建新实例。此外，Class对象提供了一些不错的方法，比如getSuperClass()，可以用它返回Class对象的父类。

Class对象可以和反射API一起实现对方法、域、构造方法等类成员的间接访问。Class对象中有对类成员Method和Field对象的引用。反射API可以用这些对象实现对它们的间接访问。图5-2是这种结构的高层视图。

运行时的哪个部分会定位并连接字节流以生成新的加载类？在下一个主题中，我们会讨论这个问题，即能够完成这些工作的类加载器，它是由抽象类ClassLoader的子类们组成的。

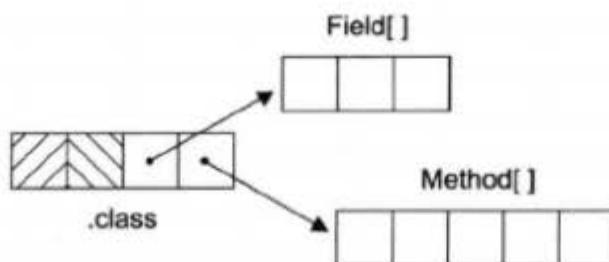


图5-2 Class对象与Method引用

5.1.4 类加载器

Java平台里有几个经典的类加载器，它们在平台的启动和常规操作过程中承担不同的任务：

- **根（或引导）类加载器**——通常在VM启动后不久实例化，一般用本地代码实现。最好把它看做VM的一部分。它的作用通常是负责加载系统的基础JAR（主要是rt.jar），而且它不做验证工作。
- **扩展类加载器**——用来加载安装时自带的标准扩展。一般包括安全性扩展。
- **应用（或系统）类加载器**——这是应用最广泛的类加载器。它负责加载应用类。在大多数SE（Java标准版）的环境中，主要工作都是由它来完成。
- **定制类加载器**——在更复杂的环境中，比如EE（Java企业版）或比较复杂的SE框架，通常会有些附加（即定制）的类加载器。有些团队甚至为他们的某个应用程序编写了特定的类加载器。

除了核心任务，类加载器还经常要从JAR文件或classpath中加载资源（不是类文件，比如图片或配置文件）。

例子——工具类加载器

在EMMA测试覆盖工具（<http://emma.sourceforge.net/>）中使用的一个类加载器可以作为加载时转化的例子。

当为了加上额外的测试辅助信息而加载类时，EMMA的类加载器会修改字节码。当在这些代码上运行测试用例时，EMMA会记录测试用例实际测试了哪些方法和代码分支。从这些记录中，开发人员能看出对一个类的单元测试是否全面。关于测试和覆盖，在11和12章还有更多的相关讨论。

有些框架和代码还经常会使用带有额外属性的专用（甚至用户自定义的）类加载器。这些类加载器经常会在加载时对字节码进行转换，我们在第1章有提到过。

图5-3中是类加载器的继承层级以及不同加载器之间的相互关系。

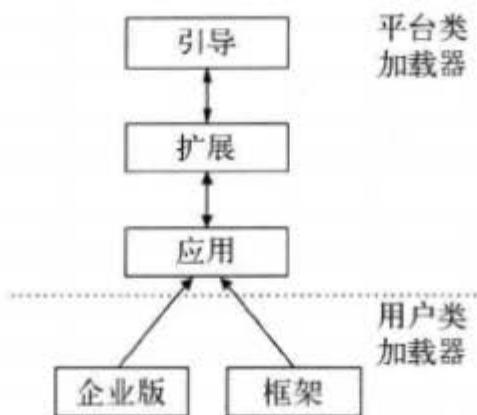


图5-3 类加载器层级

我们先来看一个专用类加载器的例子——如何用类加载实现依赖注入。

5.1.5 示例：依赖注入中的类加载器

DI有两个核心思想。

- 系统内的功能单元要靠依赖项和配置信息才能正常发挥作用。
- 在对象自身的上下文里，很难表示依赖项，即使可以，也很复杂难懂。

你脑海中应该浮现出一幅包含了行为，配置和依赖项信息（它们处在对象之外）的画面。这部分通常被称为对象的运行时路线。

第3章以Guice框架为例介绍了DI。本节中我们会讨论利用类加载器实现DI的方式，但这种方式与Guice不同，实际上它更像简化版的Spring框架。

在这个想象出来的DI框架下，我们像这样启动应用程序：

```
java -cp <CLASSPATH> org.framework.DIMain /path/to/config.xml
```

CLASSPATH中必须包含DI框架的JAR文件以及在config.xml文件中引用的所有类（以及它们的所有依赖项）的JAR文件。

我们改写了前面一个类似的例子——代码清单3-7中的服务，结果如清单5-1所示。

代码清单5-1 HollywoodService——不同的DI风格

```

public class HollywoodServiceDI {
    private AgentFinder finder = null;
    public HollywoodServiceDI() {}                                | 空的构造方法
    public void setFinder(AgentFinder finder) {                  | ↓
        this.finder = finder;                                     | setter方法
    }
    public List<Agent> getFriendlyAgents() {                      | ↓
        ...
    }
    public List<Agent> filterAgents(List<Agent> agents, String agentType) {
        ...
    }
}

```

同代码清单3-7

同代码清单3-7

为了将它置于DI框架的管理之下，还需要一个配置文件：

```
<beans>
    <bean id="agentFinder" class="wgjd.ch03.WebServiceAgentFinder"
        ...
    </bean>
    <bean id="hwService" class="wgjd.ch05.HollywoodServiceDI"
        p:finder-ref="agentFinder"/>
</beans>
```

在这种方式中，DI框架利用配置文件来确定要构造的对象。这个例子需要hwService和agentFinder两个bean，框架会为每个bean调用空构造方法，之后是setter方法（比如为HollywoodServiceDI的依赖项AgentFinder调用setFinder()）。

这说明类加载分为两个阶段。第一阶段由应用类加载器完成，负责加载DIMain及其引用的类。然后DIMain开始运行，并在main()的参数中得到配置文件的位置。

这时候，框架已经在JVM中运行起来了，但config.xml中指定的用户类还碰都没碰呢。实际上，在DIMain检查配置文件之前，框架不可能知道要加载什么类。

要启动config.xml中指定的应用配置，需要类加载的第二阶段。这要用到定制的类加载器。首先，要检查config.xml文件的一致性，确保它没有错误。然后，如果毫无差错，定制的类加载器就会试图从CLASSPATH中加载指定类型。如果任何一步失败了，整个过程就会被中止。

如果成功了，DI框架可以继续创建所需的实例，并调用实例上的setter方法。如果这些都顺利完成了，程序上下文就开始运行了。

我们简单介绍了一下Spring风格的DI方式，其中大量使用了类加载。在Java技术中，还有很多要用到类加载器及其相关技术的领域。下面是一些众所周知的例子：

- 插件架构；
- 厂商提供的或自主研发的框架；
- 从非正常位置（非文件系统或URL）获取类文件；
- Java EE；
- 任何需要在JVM进程已经启动后加入新的、未知代码的情况下。

我们对类加载的讨论就到此为止。让我们进入下一节，探讨Java 7为满足反射等需求而提供的新API。

5.2 使用方法句柄

如果你不熟悉Java的反射API（Class、Method、Field和它们的朋友），可以大致浏览一下（甚至跳过）这一节的内容。可如果你的代码库中有很多反射代码，那么你一定要认真读一读，因为它介绍了Java 7中取得相同效果的新办法，而且所用的代码更简洁。

Java 7为间接调用引入了新的API。其中的关键是java.lang.invoke包，即方法句柄。你可以把它看做反射的现代化方式，但它不像反射API那样有时会显得冗长、繁重和粗糙。

取代反射代码

反射中有很多套路化的代码。如果你写过一些反射代码，就不会忘记必须一次又一次地用Class[]指向内省方法的参数类型，并把该方法的参数都封装成Object[]，还要捕捉各种讨厌的异常以防出错，而且反射代码看起来也很不直观。

通过将反射代码转移到方法句柄，可以去掉套路化的代码，提高代码的可读性，这是大势所趋。

方法句柄是将invokedynamic（详情参见5.5节）引入JVM项目中的一部分。但其作用不仅限于invokedynamic的应用案例，在框架和常规用户代码中也有用武之地。接下来我们会先介绍方法句柄的基本技术；之后会给出一个例子与现有的各种方式进行比较，并总结出其中的差异。

5.2.1 MethodHandle

什么是MethodHandle？它是对可直接执行的方法（或域、构造方法等）的类型化引用，这是标准答案。还有一种说法：方法句柄是一个有能力安全调用方法的对象。

下面我们要获取一个带有两个参数的方法（但我们可能连这个方法的名字都不知道）的方法句柄，之后调用对象obj上的句柄，传入参数arg0和arg1：

```
MethodHandle mh = getTwoArgMH();
MyType ret;
try {
    ret = mh.invokeExact(obj, arg0, arg1);
} catch (Throwable e) {
    e.printStackTrace();
}
```

这种能力有些像反射，还有些像4.4节介绍的Callable接口。实际上，Callable是对方法调用能力建模的早期尝试。但它只适用于不带参数的方法。为了满足现实情况中不同参数组合和调用的可能，我们需要编写带有特定参数组合的其他接口。

Java 6中有很多这种代码，接口四处蔓延，让开发人员万分苦恼（比如耗光保存类信息的PermGen内存——见第6章）。相比较而言，方法句柄则适用于任何方法签名，不需要产生那么多小类。这要归功于新引入的MethodType类。

5.2.2 MethodType

MethodType是表示方法签名类型的不可变对象。每个方法句柄都有一个MethodType实例，用来指明方法的返回类型和参数类型。但它没有方法的名字和“接收者类型”，即调用的实例方法的类型。

用MethodType类中的工厂方法可以得到MethodType实例。这里有几个例子：

```
MethodType mtToString = MethodType.methodType(String.class);
MethodType mtSetter = MethodType.methodType(void.class, Object.class);
```

```
MethodType mtStringComparator = MethodType.methodType(int.class,
String.class, String.class);
```

这些 MethodType 实例分别表示 `toString()`，setter 方法（`Object` 类的成员）和 `Comparator<String>` 定义的 `compareTo()` 方法的类型签名。MethodType 实例一般都遵循相同的模式，第一个传入的参数是方法的返回类型，随后的参数是方法参数的类型（跟 Class 对象一样），如下所示：

```
MethodType.methodType(RetType.class, Arg0Type.class, Arg1Type.class, ...);
```

你看，现在可以用普通对象来表示不同的方法签名了，不再逐一为它们定义新类型。这也在最大程度上保证了类型安全性，而且办法还很简单。如果你想知道某个方法句柄能否用特定的参数集调用，可以检查该句柄的 MethodType。

现在你应该明白 MethodType 是如何解决接口泛滥的问题了，接下来就去看看怎么得到指向类中方法的方法句柄吧。

5

5.2.3 查找方法句柄

下面的代码展示了如何得到指向当前类中 `toString()` 方法的方法句柄。注意，`mtToString` 和 `toString()` 的签名完全一致，返回类型为 `String`，没有参数。也就是说相应的 MethodType 实例是 `MethodType.methodType(String.class)`。

代码清单5-2 查找方法句柄

```
public MethodHandle getToStringMH() {
    MethodHandle mh;
    MethodType mt = MethodType.methodType(String.class);
    MethodHandles.Lookup lk = MethodHandles.lookup();           ← 获取上下文

    try {
        mh = lk.findVirtual(getClass(), "toString", mt);
    } catch (NoSuchMethodException | IllegalAccessException mhx) { ← 从上下文中查
        throw (AssertionError) new AssertionError().initCause(mhx);
    }
    return mh;
}
```

取得新的方法句柄要用 `lookup` 对象，比如代码清单 5-2 中的 `lk`。这个对象可以提供其所在环境中任何可见方法的方法句柄。

要从 `lookup` 对象中得到方法句柄，你需要给出持有所需方法的类、方法的名称，以及跟你所需的方法签名相匹配的 `MethodType`。

注意 在查找上下文（`lookup context`）中可以得到任何类型（包括系统类型）中的方法句柄。

当然，如果要从没有关联的类中取得句柄，查找上下文中只能看到或取得 `public` 方法的句柄。就是说方法句柄总是在安全管理之下安全使用——没有反射中 `setAccessible()` 那种破解方法。

现在你已经拿到了方法句柄，接下来自然是执行它。方法句柄API为此提供了两个方法：`invokeExact()`和`invoke()`。`invokeExact()`方法要求其参数类型与底层方法所期望的参数类型完全匹配。`invoke()`方法会在参数类型不太正确时做些修改，以使其与底层方法参数相匹配（比如在需要时进行装箱或拆箱）。

接下来我们会给出一个长一点儿的例子，说明如何使用方法句柄取代过去的技术，比如反射和小型代理类。

5.2.4 示例：反射、代理与方法句柄

如果你曾经处理过满是反射的代码库，就会深知反射代码所带来的痛苦了。在本节中，我们要向你证明方法句柄可以取代很多套路化的反射代码，会让你的编码生涯更轻松。

代码清单5-3是改编自前面章节的例子。`ThreadPoolManager`负责将新任务分配给线程池，和代码清单4-15稍有不同。它还能取消正在运行的任务，但是个私有方法。

为了阐明方法句柄和其他技术之间的差别，我们给出了从外部访问类的私有方法`cancel()`的三种办法：`makeReflective`、`makeProxy`和`makeMh`。我们还展示了两种Java 6技术：反射和代理类。并且和基于`MethodHandle`的方式进行了比较。我们用到了一个读取队列的任务`QueueReaderTask`（实现了`Runnable`接口）。你可以在本章源码中找到`QueueReaderTask`实现。

代码清单5-3 三种访问方式

```
public class ThreadPoolManager {
    private final ScheduledExecutorService stpe =
        Executors.newScheduledThreadPool(2);
    private final BlockingQueue<WorkUnit<String>> lbq;
    public ThreadPoolManager(BlockingQueue<WorkUnit<String>> lbq_) {
        lbq = lbq_;
    }
    public ScheduledFuture<?> run(QueueReaderTask msgReader) {
        msgReader.setQueue(lbq);
        return stpe.scheduleAtFixedRate(msgReader, 10, 10,
            TimeUnit.MILLISECONDS);
    }
    private void cancel(final ScheduledFuture<?> hndl) { ←
        stpe.schedule(new Runnable() {
            public void run() { hndl.cancel(true); }
        }, 10, TimeUnit.MILLISECONDS);
    }
    public Method makeReflective() {
        Method meth = null;
        try {
            Class<?>[] argTypes = new Class[] { ScheduledFuture.class };
            meth = ThreadPoolManager.class.getDeclaredMethod("cancel",
                argTypes);
        } catch (Exception e) {
        }
    }
}
```

要访问的私有方法

```

        meth.setAccessible(true);
    } catch (IllegalArgumentException | NoSuchMethodException
| SecurityException e) {
    e.printStackTrace();
}

return meth;
}

public static class CancelProxy {
    private CancelProxy() { }

    public void invoke(ThreadPoolManager mae_, ScheduledFuture<?> hndl_) {
        mae_.cancel(hndl_);
    }
}

public CancelProxy makeProxy() {
    return new CancelProxy();
}

public MethodHandle makeMh() {
    MethodHandle mh;
    MethodType desc = MethodType.methodType(void.class,
ScheduledFuture.class);

    try {
        mh = MethodHandles.lookup()
.findVirtual(ThreadPoolManager.class, "cancel", desc);
    } catch (NoSuchMethodException | IllegalAccessException e) {
        throw (AssertionError)new AssertionError().initCause(e);
    }

    return mh;
}
}

```

5

这个类提供了三个访问私有方法cancel()的方法。实际上，一般实现时只会用一个，我们是为了讨论它们之间的差别才全都列了出来。

下面是使用这些方法的例子。

代码清单5-4 使用这些访问方法

```

private void cancelUsingReflection(ScheduledFuture<?> hndl) {
    Method meth = manager.makeReflective();

    try {
        System.out.println("With Reflection");
        meth.invoke(hndl);
    } catch (IllegalAccessException | IllegalArgumentException
| InvocationTargetException e) {
        e.printStackTrace();
    }
}

```

```

private void cancelUsingProxy(ScheduledFuture<?> hndl) {
    CancelProxy proxy = manager.makeProxy();
    System.out.println("With Proxy");
    proxy.invoke(manager, hndl);
}

private void cancelUsingMH(ScheduledFuture<?> hndl) {
    MethodHandle mh = manager.makeMh();
    try {
        System.out.println("With Method Handle");
        mh.invokeExact(manager, hndl);
    } catch (Throwable e) {
        e.printStackTrace();
    }
}

BlockingQueue<WorkUnit<String>> lbq = new LinkedBlockingQueue<>();
manager = new ThreadPoolManager(lbq);

final QueueReaderTask msgReader = new QueueReaderTask(100) {
    @Override
    public void doAction(String msg_) {
        if (msg_ != null) System.out.println("Msg recv'd: "+ msg_);
    }
};
hndl = manager.run(msgReader);

```

通过代理调用
是静态类型的

方法签名必须
完全一致

必须捕捉
Throwable

然后用**hndl**
取消任务

这几个cancelUsing方法都有一个ScheduledFuture参数，所以你可以用前面的代码试验不同的取消方法。实际上，作为API的使用者，你可以不用去管这是如何实现的。

在下一节中，我们会告诉你API或框架开发人员用方法句柄取代其他方式的原因。

5.2.5 为什么选择 **MethodHandle**

在上一节中我们看了一个把方法句柄用在Java 6中使用反射和代理的地方的例子。这引出了一个问题：为什么要用方法句柄取代过去的方式？

从表5-1可以看出，反射最大的优势就是人们熟悉它。代理对于简单用例可能更容易理解，但我们认为方法句柄在这两方面做得都是最棒的。我们强烈推荐你使用方法句柄。

表5-1 Java的方法间接访问技术比较

特 性	反 射	代 理	方法句柄
访问控制	必须使用setAccessible()。内部类可以访问受限方法会被安全管理器禁止	在恰当的上下文中对所有方法都有完整的访问权限。和安全管理器没有冲突	
类型纪律 (Type discipline)	没有。不匹配就抛出异常	静态的。过于严格。为了存储全部的代理类，可能需要很多PermGen	在运行时是类型安全的。不占用PermGen
性 能	跟其他的比算慢的	跟其他方法调用一样快	力求跟其他方法调用一样快

方法句柄还有一个特性，可以从静态上下文中确定当前类。如果你曾经编写过这样的日志代码（比如log4j）：

```
Logger lgr = LoggerFactory.getLogger(MyClass.class);
```

你应该知道这样的代码很脆弱。如果它被重构进超类或子类中，显式声明的类名就会有问题。然而在Java 7中，你可以这样写：

```
Logger lgr = LoggerFactory.getLogger(MethodHandles.lookup().lookupClass());
```

在这行代码中，可以把`lookupClass()`看成用在静态上下文中的`getClass()`。这在处理日志框架之类的情况下特别有用，因为通常每个用例都有自己的logger。

带着新掌握的方法句柄技术，我们去检查一下类文件的底层细节和使其变得有意义的工具。

5.3 检查类文件

5

类文件是二进制块，所以想直接和它打交道不太容易。但有很多时候你会发现必须和类文件交手。

比如说，为了在运行时更好地监控（比如通过JMX）应用程序，你需要加上额外的公共方法。重新编译和再次部署看起来顺利完成了，但检查管理API时却发现没有那些方法。又进行了几次构建和部署还是没有发现。

为了找出部署问题，你需要检查一下javac产生的类文件是不是你想要的那个。还有时候你需要研究那些没有源码的类文件，以验证文档中是不是真有你所怀疑的错误。

对于类似的任务，你必须用工具检查类文件的内容。好在标准的Oracle JVM中有javap这个工具，用它来探视类文件内部和反汇编类文件非常得心应手。

我们一开始会先介绍javap，以及为检查类文件而设置的各种基本参数。接下来会讨论方法名称和类型在JVM内部的一些表示方式。然后看一下常量池，它是JVM的“藏宝箱”，对于理解字节码如何工作非常重要。

5.3.1 介绍 javap

javap的用处很多，既能看类声明了什么方法，又能输出字节码。我们来看一下javap最简单的用途，在第4章讨论的微博Update上试一下。

```
$ javap wgjd/ch04/Update.class
Compiled from "Update.java"
public class wgjd.ch04.Update extends java.lang.Object {
    public wgjd.ch04.Author getAuthor();
    public java.lang.String getUpdateText();
    public int hashCode();
    public boolean equals(java.lang.Object);
    public java.lang.String toString();
    wgjd.ch04.Update(wgjd.ch04.Update$Builder, wgjd.ch04.Update);
}
```

默认情况下，javap会显示访问权限为public、protected和默认（即包级protected）级别的方法。加上-p选项后还可以显示private方法和域。

5.3.2 方法签名的内部形式

JVM内部用的方法签名和javap显示出来供人阅读的形式不太一样。随着我们对JVM的不断深入，这些内部名称出现将更加频繁。如果你赶时间，可以跳过这一节。但请记住它，因为你可能还要回来参考这些内容。

在紧凑形式中，类型名称是经过压缩的。比如int是用I表示的。这些紧凑形式有时被称为类型描述符。表5-2中是类型描述符的完整列表。

表5-2 类型描述符

描述符	类 型
B	byte
C	char (16位Unicode字符)
D	double
F	float
I	int
J	Long
L<类型名称>	引用类型（比如Ljava/lang/String；用于字符串）
S	short
Z	boolean
[array-of

某些情况下，类型描述符可能比类型名称还要长（比如Ljava/lang/Object就比Object长），但类型描述符是完全限定的，所以可以直接解析。

javap还有一个有用的选项-s，可以输出签名的类型描述符，所以你没必要用那个表自己做转换。你可以使用javap高级一些的方法来显示我们之前看过的一些方法的签名：

```
$ javap -s wgjd/ch04/Update.class
Compiled from "Update.java"
public class wgjd.ch04.Update extends java.lang.Object {
    public wgjd.ch04.Author getAuthor();
        Signature: ()Lwgjd/ch04/Author;

    public java.lang.String getUpdateText();
        Signature: ()Ljava/lang/String;

    public int compareTo(wgjd.ch04.Update);
        Signature: (Lwgjd/ch04/Update;)I

    public int hashCode();
        Signature: ()I

    ...
}
```

如你所见，方法签名中的所有类型都是用类型描述符表示的。

在下一节中你会看到类型描述符的另一个用途。它会出现在类文件中非常重要的部分——常量池。

5.3.3 常量池

常量池是为类文件中的其他（常量）元素提供快捷访问方式的区域。如果你研究过C或Perl之类语言，应该知道符号表，对于JVM来说，常量池就类似于符号表。但和其他语言不同，Java没有完全开放对常量池中信息的访问。

为了不纠缠于过多的细节，我们用一个非常简单的例子来演示线程池。下面是一个简单的“游戏围栏”或者叫“演算本”类。我们在这个类的run()里面写一点代码，就可以快速测试Java的语法特性或类库。

代码清单5-5 游戏围栏样例类

```
package wgjd.ch04;

public class ScratchImpl {
    private static ScratchImpl inst = null;

    private ScratchImpl() {
    }

    private void run() {
    }

    public static void main(String[] args) {
        inst = new ScratchImpl();
        inst.run();
    }
}
```

要查看常量池中的信息，可以用javap -v。这个命令还会输出很多其他信息，不过我们只关注常量池中的条目。

如下所示：

```
#1 = Class           #2      // wgjd/ch04/ScratchImpl
#2 = Utf8          wgjd/ch04/ScratchImpl
#3 = Class           #4      // java/lang/Object
#4 = Utf8          java/lang/Object
#5 = Utf8          inst
#6 = Utf8          Lwgjd/ch04/ScratchImpl;
#7 = Utf8          <clinit>
#8 = Utf8          ()V
#9 = Utf8          Code
#10 = Fieldref     #1.#11
⇒ // wgjd/ch04/ScratchImpl.inst:Lwgjd/ch04/ScratchImpl;
#11 = NameAndType   #5:#6      // instance:Lwgjd/ch04/ScratchImpl;
#12 = Utf8          LineNumberTable
#13 = Utf8          LocalVariableTable
#14 = Utf8          <init>
#15 = Methodref     #3.#16      // java/lang/Object."<init>":()V
#16 = NameAndType   #14:#8      // "<init>":()V
#17 = Utf8          this
#18 = Utf8          run
#19 = Utf8          ([Ljava/lang/String;)V
#20 = Methodref     #1.#21      // wgjd/ch04/ScratchImpl.run:()V
```

```

#21 = NameAndType #18:#8    // run:()V
#22 = Utf8          args
#23 = Utf8          [Ljava/lang/String;
#24 = Utf8          main
#25 = Methodref    #1.#16   // wgjd/ch04/ScratchImpl.<init>:()V
#26 = Methodref    #1.#27
// wgjd/ch04/ScratchImpl.run:([Ljava/lang/String;)V
#27 = NameAndType #18:#19   // run:([Ljava/lang/String;)V
#28 = Utf8          SourceFile
#29 = Utf8          ScratchImpl.java
/* // wgjd/ch04/ScratchImpl.run:([Ljava/lang/String;)V
#27 = NameAndType #18:#19   // run:([Ljava/lang/String;)V
#28 = Utf8          SourceFile
#29 = Utf8          ScratchImpl.java

```

如你所见，常量池中的条目是带有类型的。它们还会相互引用，比如说，一个类型为Class的条目会引用类型为Utf8的条目。而Utf8的条目是个字符串，所以Class条目引用的Utf8条目应该是类的名称。

表5-3是可能出现在常量池中的条目集。在讨论常量池中的条目时，有时会用CONSTANT_前缀，比如CONSTANT_Class。

表5-3 常量池条目

名 称	描 述
Class	类常量。引用类的名称 (Utf8 条目)
Fieldref	定义域。引用该域的Class 和 NameAndType
Methodref	定义方法。引用该方法的Class和NameAndType
InterfaceMethodref	定义接口方法。引用该方法的Class 和 NameAndType
String	字符串常量。引用保存字符的Utf8常量
Integer	整型常量 (4字节)
Float	浮点常量 (4字节)
Long	长整型常量 (8字节)
Double	双精度浮点型常量 (8字节)
NameAndType	描述名称和类型对。类型引用一个保存类型描述符的Utf8条目
Utf8	一个表示以Utf8编码的字符的二进制字节流
InvokeDynamic	(Java 7中新引入的) 见5.5节
MethodHandle	(Java 7中新引入的) 描述MethodHandle常量
MethodType	(Java 7中新引入的) 描述MethodType常量

你可以用这个表格从演算类的常量池中看到常量解析的例子。比如条目#10中的Fieldref。要解析一个域，你需要名称、类型，还有它所在的类：#10的值是#1.#11，这就是说常量#11来自类#1。在输出中可以很容易看出#1确实是一个Class类型的常量，并且#11是NameAndType。#1指向ScratchImpl类本身，#11是#5:#6——一个名称为inst的ScratchImpl变量。所以综合来看，#10指向ScratchImpl类内部的自身静态变量inst(你可能已经从清单5-6的输出中猜出来了)。

在类加载过程中的验证环节，有一步是检查类文件中的静态信息是否一致的。前面的例子是运行时在加载新类时要做的完整性检查。

对于类文件的基本结构，我们已经讨论的差不多了。接下来要进入下一话题——字节码。理解源码如何变成字节码会对你理解代码如何运行有很大的帮助。在学习第6章以及后面的章节时，还能引导你更加深入地了解平台的能力。

5.4 字节码

到目前为止，在我们的讨论中，字节码一直有点幕后工作者的意思。我们先来回顾一下对它已经有了哪些了解，然后再对它进行详细介绍：

- 字节码是程序的中间表示形式：介于人类可读的源码和机器码之间。
- 字节码是由源码文件中的javac产生的。
- 某些高层语言特性在编译时已经从字节码中去掉了。比如说Java的循环结构(`for`、`while`等)在字节码中就被转换成了分支指令。
- 每个操作码都由一个字节表示(因此被叫做字节码)。
- 字节码是一种抽象表示法，不是“某种虚拟CPU的机器码”。
- 字节码可以进一步编译成机器码，通常是“即时编译”。

5

字节码解释起来有点像先有鸡还是先有蛋的问题。要彻底搞清楚状况，你既要懂字节码，又要明白执行它的运行时环境。

这是一个循环依赖，为了解决这个问题，我们先来探索一个相对简单的例子。即使这一次你不太明白，也可以在后续章节读到更多字节码相关内容时再回来看看。

在例子之后，我们会给出一些与运行时环境相关的上下文和JVM操作码的目录(其中包括用于数学计算、调用、快捷形式之类的字节码)。最后，我们会用另外一个基于字符串拼接的例子来结束。现在就先去看看如何检查.class文件的字节码吧。

5.4.1 示例：反编译类

用带有-c选项的javap可以对类进行反编译。我们会以代码清单5-5中的演算类为例，主要检查方法之内的字节码。我们还会加上-p选项，以便能见到私有方法内的字节码。

我们一节一节的来——javap输出的每一部分都有很多信息，很容易让人不堪重负。首先，让我们先看头部。这里没什么特别出人意料或让人喜出望外的：

```
$ javap -c -p wgjd/ch04/ScratchImpl.class
Compiled from "ScratchImpl.java"
public class wgjd.ch04.ScratchImpl extends java.lang.Object {
    private static wgjd.ch04.ScratchImpl inst;
```

接下来是静态块。变量的初始化就放在这里，所以这表示inst被初始化为null了。看起来putstatic可能是一个把值放到静态域中的字节码。

```

static {};
Code:
 0: aconst_null
 1: putstatic    #10   // Field inst:Lwgjd/ch04/ScratchImpl;
 4: return

```

代码前面的数字表示从方法开始算起的字节码偏移量。所以字节1是putstatic操作码，字节2和3表示一个16位的常量池索引，这个16位索引在这里的值是10，表示该值（此处为null）会存在常量池的条目#10所指明的域中。从字节码流开始的第4个字节是return操作符，表明这个代码块结束了。

接下来是构造方法。

```

private wgjd.ch04.ScratchImpl();
Code:
 0: aload_0
 1: invokespecial #15   // Method java/lang/Object."<init>":()V
 4: return

```

在Java中，void构造方法总会隐式调用超类中的构造方法。这从上面的字节码里就能看出来invokespecial指令。一般来说，任何方法调用都会转换成VM的某一调用指令。

在run()方法中没有代码，因为这只是一个空白的演算类。

```

private void run();
Code:
 0: return

```

在main方法中，你初始化了inst，还做了点对象创建。这说明了辨识通用字节码的基本模式：

```

public static void main(java.lang.String[]);
Code:
 0: new           #1   // class wgjd/ch04/ScratchImpl
 3: dup
 4: invokespecial #21   // Method "<init>":()V

```

这种3个字节码指令的模式——new、dup和一个<init>的invokespecial——都表示创建新实例。

操作码new只为新实例分配内存。dup复制栈顶上的元素。要完整创建该对象，你需要调用构造方法的代码块。<init>方法中包含构造方法的代码，所以可以用invokespecial调用那段代码。我们继续看main方法中其余的字节码：

```

7: putstatic    #10   // Field inst:Lwgjd/ch04/ScratchImpl;
10: getstatic    #10   // Field inst:Lwgjd/ch04/ScratchImpl;
13: invokespecial #22   // Method run:()V
16: return
}

```

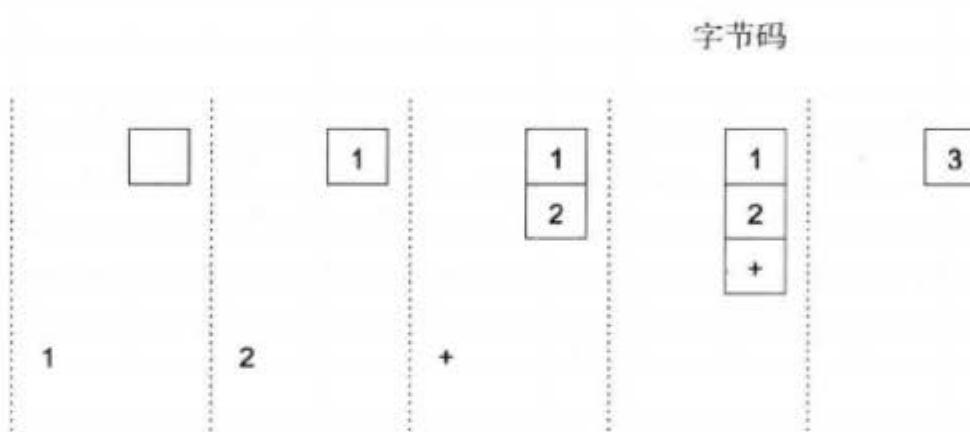
指令7保存刚刚创建的单例实例。指令10把它放回到栈顶上，这样指令13就可以调用它上面的方法了。注意，因为调用的run()是私有方法，所以13是invokespecial。私有方法不能重写，所以不能用Java的标准虚拟查询。大多数方法调用都会转换成invokevirtual指令。

注意 通常来说，javac产生的字节码没有经过特别优化，是非常简单的表示形式。基本策略是由JIT编译器来完成大部分的优化工作，所以简单直白的起点对它们是很有帮助的。VM实现者表示，“字节码就应该傻傻的”，这是他们对从源语言产生的字节码的总体感觉。

接下来我们要讨论字节码所需的运行时环境，之后会介绍用来描述字节码指令主要“家庭成员”的表格，其中包括加载/存储、数学计算、执行控制、方法调用和平台操作。然后我们会讨论操作码可能的快捷形式，最后会再给出一个例子。

5.4.2 运行时环境

因为JVM使用堆栈机，所以理解堆栈机的操作对理解字节码至关重要。



5

图5-4 将栈用于数学运算

JVM与硬件CPU（比如x64或ARM芯片）最显著的差别在于它没有处理器寄存器，而是用栈完成所有的计算和操作。有时候也这也被称为操作数栈（或计算堆栈）。图5-4展示了如何用操作数栈完成两个int数值的相加运算。

正如我们前面讨论过的，当一个类被链接进运行时环境时，它的字节码会受到检查，并且其中很多验证都可以归结为对栈中类型模式的分析。

注意 栈中的值只有类型正确时对它的处理才能生效。比如，如果我们把对一个对象的引用压入栈，然后试图将其作为int型进行数学计算，就可能会发生未定义或糟糕的事情。类加载过程中的验证阶段会进行广泛的检查，以确保新加载的类中不会有滥用栈的方法。这样做能够防止系统接受了损坏（或恶意）的类并引发问题。

方法在运行时需要一块内存区域作为计算堆栈来计算新值。另外，每个运行的线程都需要一个调用堆栈（栈跟踪中会报告的那个栈）来记录当前正在执行的方法。在某些情况下，这两个栈会有交互。看下面这行代码：

```
return 3 + petRecords.getNumberOfPets("Ben");
```

要计算出这行代码的结果，需要把3压入操作数栈。然后调用方法计算Ben有多少只宠物。为此，你需要把接收对象（方法属主，即petRecords）压入计算堆栈，要传入的所有参数尾随其后。

然后invoke操作符会调用方法getNumberOfPets()，把控制权移交给被调用的方法，刚刚进入的方法会出现在调用堆栈中。但进入新方法后，需要启用不同的操作数栈，所以已经在调用者的操作数栈中的值不可能影响被调用方法的计算结果。

在getNumberOfPets()完成时，返回结果会被放到调用者的操作数栈中，进程中与getNumberOfPets()相关的部分也会从调用堆栈中移走。然后相加运算可以得到两个值并把它们加在一起。

现在我们开始审视字节码。这是个大课题，而且有很多特殊情况，所以我们即将呈现的只是主要特性的概览，而不是完整的介绍。

5.4.3 操作码介绍

JVM字节码由操作码(opcode)序列构成，每个指令后面可能会跟着一些参数。操作码希望看到栈处于指定状态中，然后它对栈进行转换，把参数移走，放入结果。

每个操作码都由一个单字节值表示，所有最多只能有255个操作码。当前仅用了200个左右。对我们来说，把它们全列出来有点儿太多了，好在大多数操作码都可以归为几大族系。我们会逐一讨论这些族系，帮助你理解它们。还有一些操作码不好界定应该归为哪一族系，但好在你不会经常遇见它们。

注意 JVM不是纯粹的面向对象运行时环境——它支持原始类型。这在某些操作码族系中有所体现——其中一些基本操作码类型（比如存储和相加）要有一些变体，在处理原始类型时会有所不同。

操作码表有四列：

- 名称：这是操作码类型的通用名称。大多数情况下，都会有几个相关的操作码在做类似的事情。
- 参数：操作码的参数。以i打头的参数是用来作为常量池或局部变量中的查询索引的几个字节。如果有更多的此类参数，它们会合并在一起，所以i1, i2表示“从这两个字节中生成一个16位的索引”。如果参数出现在括号里，就表明不是所有形式的操作码都会使用它。
- 堆栈布局：它展示了栈在操作码执行前后的状态。括号中的元素表明不是所有形式的操作码都使用它们，或者这些元素是可选的（比如调用操作码）。
- 描述：操作码的用处。

我们从表5-4中拿过来一行代码做例子，检查一下操作码getfield的条目。这个操作码用于从对象的域中读出一个值。

getfield i1, i2 [obj] → [val] 从栈顶端对象的常量池中取出指定位置的域。

第一列给出了操作码的名字：getfield。后面一列说明在字节码流中有两个参数跟在操作码后面。这些参数合在一起构成一个16位的值，可以用来从常量池里找到想要的域（记住常量池的索引总是16位的）。

堆栈布局那一列表明在找到栈顶端对象的类的常量池中的索引位置之后，该对象被移除，它的位置被那个域所替代。

这种把移走对象作为操作一部分的模式是一种让字节码变得紧凑的办法，没有繁琐的清理工作，也不用记着要挪走处理完的对象实例。

5.4.4 加载和储存操作码

加载和储存操作码这个族系负责将值加载到栈或检索值。表5-4给出了加载/储存族系的主要操作。

表5-4 加载和储存操作码

名 称	参 数	堆栈布局	描 述
load	(i1)	[] • [val]	从局部变量加载值（原始型或引用型）到栈上。有快捷形式，并且有针对不同类型的变体
ldc	i1	[] • [val]	从池中加载常量到栈上，针对不同类型有不同的变体，并且范围广泛
store	(i1)	[val] • []	把值（原始型或引用型）从进程的栈中移走，存到局部变量中。有快捷形式，有针对不同类型的变体
dup		[val] • [val, val]	复制栈顶部的值，有不同形式的变体
getfield	i1, i2	[obj] • [val]	从栈顶部对象的常量池中得到指定位置的域
putfield	i1, i2	[obj, val] • []	把值放入对象在常量池中指定位置的域上

前面提过，加载和储存指令有很多不同形式的变体。比如用来把双精度数从局部变量加载到栈上的dload操作码，以及用来把对象引用从栈弹出到局部变量中的astore操作码。

5.4.5 数学运算操作码

这些操作符在栈上执行数学运算。它们从栈顶端取出参数并进行计算。这些参数（总是原始型）必须完全匹配，但平台提供了很多对原始型进行类型转换的操作码。表5-5给出了基本的数学运算操作码。

类型转换（cast）操作码的名称非常短，比如i2d是把int转为double的操作码。需要特别说明的是，类型转换操作码中并没有cast，所以在表5-5中用括号把它括了起来。

表5-5 数学运算操作码

名 称	参 数	堆栈布局	描 述
add		[val1, val2] • [res]	把栈顶端的两个值相加（必须是相同的原始类型），并把结果存在栈中。有快捷形式，有针对不同类型的变体
sub		[val1, val2] • [res]	把栈顶端的两个值相减（必须是相同的原始类型），并把结果存在栈中。有快捷形式，有针对不同类型的变体

(续)

名 称	参 数	堆栈布局	描 述
div		[val1, val2] • [res]	把栈顶端的两个值相除（必须是相同的原始类型），并把结果存在栈中。有快捷形式，有针对不同类型的变体
mul		[val1, val2] • [res]	把栈顶端的两个值相乘（必须是相同的原始类型），并把结果存在栈中。有快捷形式，有针对不同类型的变体
(cast)		[value] • [res]	把值从一种原始类型转换为另外一种。每一种可能的类型转换都有对应的形式

5.4.6 执行控制操作码

如前所述，高级语言的控制结构在JVM字节码中没有出现。相反，流程控制是由很少的几个原始指令完成的，如表5-6所示。

表5-6 流程控制操作码

名 称	参 数	堆栈布局	描 述
if	b1, b2	[val1, val2] • []或[val1] • []	如果符合特定条件，则跳转到特定分支的偏移处
goto	b1, b2	[] • []	无条件地跳转到分支偏移处。有宽大形式
jsr	b1, b2	[] • [ret]	跳到本地子流程中，并把返回地址（下一个操作码的偏移地址）放到栈中。有宽大形式
ret	索引	[] • []	返回到索引的局部变量所指向的偏移地址
tableswitch	{依情况而定}	[index] • []	用于实现switch
lookupswitch	{依情况而定}	[key] • []	用于实现switch

就像用于查找常量的索引字节，参数b1、b2用于构造方法内部的字节码跳转地址。jsr指令用于访问主流程之外一个自成体系的字节码区域（偏移地址可能在方法的主字节码之外）。在某些情况下，比如在异常处理块中，可能会用到它。

goto和jsr指令的宽大形式要用4个字节的参数，并且所构造的偏移量大于64 KB。但这并不常用。

5.4.7 调用操作码

调用操作码中有四个操作码可以处理普通的方法调用，还有一个Java 7中新出的特别操作码invokedynamic（5.5节有更多细节）。这五个方法调用操作码如表5-7所示。

表5-7 调用操作码

名 称	参 数	堆栈布局	描 述
invokestatic	i1, i2	[(val1, ...)] • []	调用一个静态方法
invokevirtual	i1, i2	[obj, (val1, ...)] • []	调用一个“常规”的实例方法

(续)

名称	参数	堆栈布局	描述
invokeinterface	i1, i2, count, 0	[obj, (val1, ...)] * []	调用一个接口方法
invokespecial	i1, i2	[obj, (val1, ...)] * []	调用一个“特殊”的实例方法
invokedynamic	i1, i2, 0, 0	[val1, ...] * []	动态调用, 见5.5节

在调用操作码中, 有两个地方需要注意。第一个是invokeinterface中多出来的参数。这些参数基于历史原因和向后兼容而产生, 但现在已经用不到了。在invokedynamic的参数中多出来的两个0是基于前向兼容而产生的。

另外一个是常规和特别实例方法调用之间的差别。常规调用是虚拟的。这就是说被调用的方法是在运行时按照标准的Java方法重写规则查找的。特殊调用不考虑重写。在两种情况下这很重要, 即私有方法和超类方法的调用。在这两种情况下, 你不想触发重写规则, 所以需要不同的调用操作码处理这种情况。

5

5.4.8 平台操作操作码

平台操作族系的操作码包括new, 用于分配新的对象实例, 还有与线程相关的操作码, 比如monitorenter和monitorexit。详细内容请参见表5-8。

平台操作码用来控制对象生命周期, 比如创建新对象并锁住它们。一定要注意, new操作码只分配存储空间。对象构建的高层概念还包括运行构造方法内的代码。

表5-8 平台操作码

名称	参数	堆栈布局	描述
new	i1, i2	[] * [obj]	为新对象分配内存, 类型由指定位置的常量确定
monitorenter		[obj] * []	锁住对象
monitorexit		[obj] * []	解锁对象

在字节码这一级, 构造方法被转换成带有特殊名称<init>的方法。这不能由用户代码调用, 但可以由字节码调用。这便形成了一个与对象创建直接相关的不同字节码模式: new之后跟着一个dup, 然后是一个调用<init>方法的invokespecial。

5.4.9 操作码的快捷形式

为了节省字节, 很多字节码都有快捷形式。通常对某些局部变量的访问要比其他的访问更加频繁, 所以用特殊的操作码来表示“在局部变量上直接执行常见操作”便很有价值。因此加载/存储族系中出现了aload_0和dstore_2这种操作码。

我们来检查一下其中的理论, 再来看一个例子。

5.4.10 示例: 字符串拼接

我们给演算类中加点料, 来阐明几个稍微高级点的字节码, 下面的例子会涉及字节码主要族

系中的大多数。

别忘了，Java中的字符串是不可变的。那在用+运算符把两个字符串拼在一起时发生了什么？你必须创建一个新字符串，但实际上可能不止这么简单。

看一下修改了run()方法之后的演算类：

```
private void run(String[] args) {
    String str = "foo";
    if (args.length > 0) str = args[0];
    System.out.println("this is my string: " + str);
}
```

这个简单方法对应的字节码为：

```
$ javap -c -p wgjd/ch04/ScratchImpl.class
Compiled from "ScratchImpl.java"

private void run(java.lang.String[]);
  Code:
    0: ldc           #17                  // String foo
    2: astore_2
    3: aload_1
    4: arraylength
    5: ifle          12 #A
```

如果传入数组尺寸小于等于0，跳到指令12。

```
    8:  aload_1
    9:  iconst_0
   10:  aaload
   11:  astore_2
   12:  getstatic   #19
=> // Field java/lang/System.out:Ljava/io/PrintStream;
```

上面这行是访问System.out的字节码。

```
15: new            #25                  // class java/lang/StringBuilder
18: dup
19: ldc           #27                  // String this is my string:
21: invokespecial #29
=> // Method java/lang/StringBuilder."<init>":(Ljava/lang/String;)V
24:  aload_2
25:  invokevirtual #32
=> // Method java/lang/StringBuilder.append
  (Ljava/lang/String;)Ljava/lang/StringBuilder;
28:  invokevirtual #36
=> // Method java/lang/StringBuilder.toString:()Ljava/lang/String;
```

这些指令展示了拼接字符串的创建过程。特别是15~23表示对象创建（new、dup和invokespecial）的指令，但在这个例子中dup之后还有一个ldc（加载常量）。这种模式表明字节码调用的是一个非空构造方法，在此是StringBuilder(String)。

这个结果一开始可能有些出乎你的意料。你只是想把一些字符串拼在一起，但到了底层突然变成了创建额外的StringBuilder对象，并调用append()，然后又调用toString()。这是因为java中的字符串是不可变的。你不能通过拼接修改字符串对象，所以必须创建新的对象。StringBuilder是完成这个任务的便捷方法。

最后是调用相应的方法输出结果：

```
31: invokevirtual #40
→ // Method java/io/PrintStream.println: (Ljava/lang/String;)V
34: return
```

最终，输出字符串拼好了，你可以调用println()方法。因为此时栈顶部的两个元素是[`System.out`, <output string>]，所以这是在`System.out`之上调用的。就跟你在看表5-7（定义了有效的invokevirtual的堆栈布局）时所预期的一样。

要成为一名真正优秀的Java开发人员，你应该找几个自己写的类用javap运行一下，并学会识别通用的字节码模式。现在，让我们带着对字节码的简单了解，进入下一主题——Java 7中重要的新特性invokedynamic。

5.5 invokedynamic

5

本节主要针对Java 7中最复杂的新特性之一。尽管这个特性十分强大，但它并不是给所有开发人员准备的，它只会出现在非常高级的用例中。目前来看，这个特性是为框架开发人员和非Java语言准备的。

也就是说如果你对平台底层如何运转不感兴趣，对新的字节码细节毫不关心，请跳到小结部分或直接进入下一章，没关系的。

如果你还在，很好。接下来我们可以向你介绍invokedynamic的出现是多么不同寻常。Java 7引入了一个崭新的字节码，这在Java世界中可是从来没有过的大事件。这个字节码新秀就是invokedynamic，一种新的调用指令，是用来做方法调用的。它可以用来告诉VM必须延迟确定要调用哪个方法。也就是说VM不用像往常一样在编译或连接时就敲定所有细节。

相反，需要什么方法在运行时决定。通过调用一个辅助方法来确定应该调用哪个方法。

javac不会产生invokedynamic

在Java 7中，Java语言还不能直接支持invokedynamic，没有哪个Java表达式会被javac直接编译成invokedynamic。人们希望Java 8会增加更多的语言结构（比如默认方法）来使用这些动态能力。

invokedynamic是为非java语言准备的。添加它是为了让动态语言能够利用Java 7 VM，不过有些聪明的Java框架也找到了让invokedynamic为它们服务的办法。

我们在本节中会给出invokedynamic的工作细节，还会给出一个详细的例子——反编译一个利用新字节码的调用点。注意，要使用那些用到invokedynamic的语言和框架不一定要完全搞清楚这些内容。

5.5.1 invokedynamic 如何工作

为了支持invokedynamic，Java 7又新增加了几条常量池定义。这些是在Java 6技术中无法

提供的支持。

给invokedynamic指令的索引必须指向类型为CONSTANT_InvokeDynamic的常量。这个常量上是两个16位的索引（也就是4字节）。第一个索引指向方法表（用来确定要调用什么）。它们被称为引导方法（有时简写为BSM），并且必须是静态的，还要有确定的参数签名。第二个索引指向CONSTANT_NameAndType。

从中可以看出CONSTANT_InvokeDynamic和普通的CONSTANT_MethodRef差不多，只是CONSTANT_MethodRef指明在哪个类的常量池里找寻方法，而invokedynamic调用则通过引导方法来寻找答案。

引导方法会返回一个CallSite实例，用它来接收与调用点相关的信息，并连接动态调用。调用点中有一个MethodHandle，调用点在这里起一个代理的作用，对它的所有调用实际上就是对MethodHandle的调用。^①

invokedynamic一开始并没有目标方法（还没连接）。在第一次调用时，该点的引导方法被调用。引导方法返回一个CallSite，它被连接到invokedynamic指令上。该过程如图5-5所示。

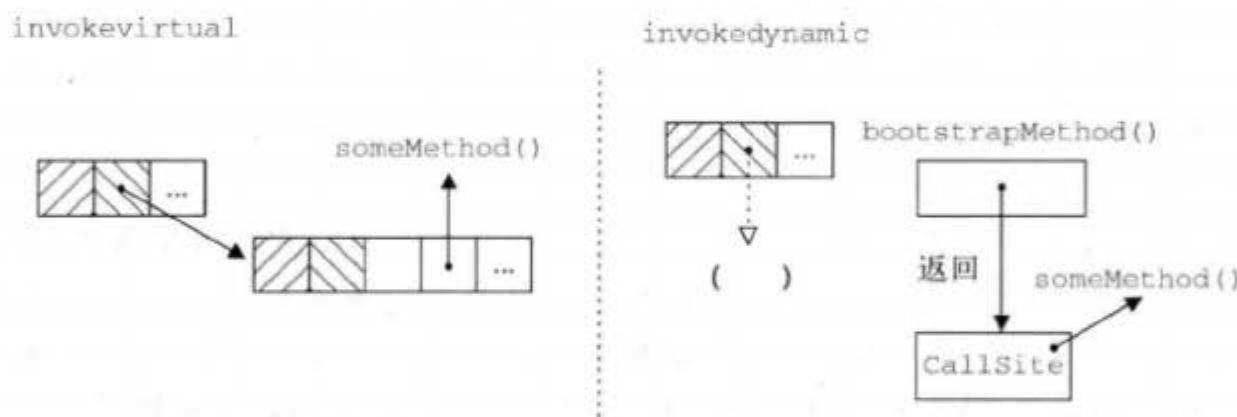


图5-5 虚拟与动态调用

连接上CallSite后，就可以调用真正的方法了，即CallSite持有的MethodHandle所指向的方法。这种设定表明JIT编译器可以像优化invokevirtual调用那样优化invokedynamic调用。下一章会讨论更多有关优化的内容。

还有一点值得注意，某些CallSite对象是可以重连的（在它们的生命期内指向不同的目标方法）。一些动态语言会大量使用这一特性。

下一节会给出一个简单的例子，我们可以看到invokedynamic调用在字节码中如何表示。

5.5.2 示例：反编译 `invokedynamic` 调用

如前所述，Java 7中没有支持invokedynamic的Java语法。要得到带有动态调用指令的.class文件，你只能向字节码处理类库求助。ASM类库（<http://asm.ow2.org/>）就是一个不错的选择——它是一个工业级类库，在Java框架中得到了广泛应用。

^① 详情请参见CallSite的Javadoc：<http://cr.openjdk.java.net/~jrose/pres/indy-javadoc-m1vm/java/lang/Invoke/CallSite.html>。

——译者注

我们可以用这个类库构造一个包含invokedynamic指令的类，然后将其转换为字节流。这既可以写到磁盘里，也可以交给类加载器插入到运行的VM中。

一个简单的例子是让ASM产生的类包含一种invokedynamic指令的静态方法。这个方法可以由普通的Java代码调用——它封装（或隐藏）了真正调用的动态本质。作为invokedynamic开发工作的一部分，Remi Forax和ASM团队提供了一个简单的工具来产生这样的测试类。ASM是第一批完全支持新字节码的工具之一。

让我们来看一下这种封装方法的字节码：

```
public static java.math.BigDecimal invokedynamic();
Code:
  0: invokedynamic #22,  0
  ↳ // InvokeDynamic #0:_:()Ljava/math/BigDecimal;
  5: areturn
```

到目前为止还没什么看头，因为复杂性主要体现在常量池中。我们来看看和动态调用相关的常量池条目：

```
BootstrapMethods:
  0: #17 invokestatic test/invdyn/DynamicIndyMakerMain.bsm:
    ↳ (Ljava/lang/invoke/MethodHandles$Lookup;Ljava/lang/String;
    ↳ Ljava/lang/invoke/MethodType;Ljava/lang/Object;)
    ↳ Ljava/lang/invoke/CallSite;
  Method arguments:
    #19 1234567890.1234567890
#10 = Utf8           ()Ljava/math/BigDecimal;
#18 = Utf8           1234567890.1234567890
#19 = String         #18   // 1234567890.1234567890
#20 = Utf8
#21 = NameAndType   #20:#10 // _:()Ljava/math/BigDecimal;
#22 = InvokeDynamic #0:#21 // #0:_:()Ljava/math/BigDecimal;
```

要想完全搞清楚确实得花点心思琢磨琢磨。我们逐一来看一下。

- invokedynamic操作码在条目#22中。它指向引导方法#0和NameAndType#21。
- 在#0的BSM是类DynamicIndyMakerMain中的普通静态方法bsm()。它有BSM的正确签名。
- 条目#21给出了这个动态连接点的名称“_”，还有返回类型BigDecimal（保存在#10）。
- 条目#19是传入引导方法的静态参数。

如你所见，这里需要做很多基础工作来保证类型安全。但在运行时出错的方式仍然还有很多，但这种机制做了很大贡献，它在保留了灵活性的同时提供了安全性。

注意 BootstrapMethods方法指向方法句柄而不是直接指向方法，这提供了额外的间接性，或者说灵活性。在前面的讨论中我们并没有涉及，因为它可能会混淆正在发生的事情，对于理解这种机制如何工作并没有实质性的帮助。

到此为止，我们已经结束了对invokedynamic和字节码及类加载内部工作机制的讨论。

5.6 小结

在本章中，我们快速浏览了字节码和类加载，还解剖了类文件，并简单介绍了JVM提供的运行时环境。随着对平台内部更深入地了解，我们相信你会成为更厉害的开发者。

希望你从本章中学到如下知识：

- 类文件格式和类加载是JVM操作的核心。它们对于任何想在VM上运行的语言来说十分重要；
- 类加载的各个阶段同时保证了运行时的安全和性能特性；
- 方法句柄是Java 7主要新API之一，它是反射之外的一个可选方案；
- JVM按相关功能分为不同的族系；
- Java 7引入了invokedynamic：一种调用方法的新办法。

现在是时候进入下一个大主题了。通过阅读下一章，你会在性能分析方面打下坚实的基础。你将学会如何评估和优化性能以及如何充分发挥JVM核心技术的能力（比如JIT编译器，它会把字节码转换成超快的机器码）。

理解性能调优

6

本章内容

- 性能的重要性
- 新的垃圾收集器G1
- VisualVM：内存可视化工具
- 即时编译

6

糟糕的性能会“杀死”你的应用程序，使你名声扫地，在客户中的信誉大受影响。除非你具有绝对垄断地位，否则你的客户将夺门而出，直奔你的竞争对手而去。要让糟糕的性能不再糟蹋你的项目，你需要理解性能分析，还要知道如何利用它。

性能分析与调优是个非常庞大的课题，但是现在有太多处理方式都在误人子弟。所以我们准备把性能调优的秘诀透漏给你。

秘诀来了——性能调优唯一的惊天秘诀就是：你必须量体裁衣。没有评测，就没有合适的调优。

原因：人们总是猜不对系统变慢的是哪里。所有人都猜不对。你，我，甚至是James Gosling大神——我们总会心生偏见，并倾向于那些可能根本不存在的模式。

实际上，“我的哪些Java代码需要优化？”这个问题的答案经常是“哪个也不用，都挺好的”。

假设有一个经典（相当保守）的电子商务Web应用为注册客户提供服务。它有一个SQL数据库，一个面向Java应用服务器的Apache Web服务器，以及连接这一切的标准网络配置。系统真正的瓶颈经常是非Java部分（数据库、文件系统、网络），但不经过评测，Java开发人员永远都不会知道问题出在哪里。开发人员不去解决真正的问题，而是把时间浪费在对于改进系统性能毫无意义的代码微调上。

你希望能够回答如下几类基本问题。

- 如果你们搞了次促销，客户突然暴增十倍，系统有足够的内存来应付这种局面吗？
- 客户从应用程序中看到的平均响应时间是多长？
- 跟竞争对手比起来怎么样？

要做性能调优，你就不能猜测导致系统变慢的原因。你必须知道并且确保你真正实现性能调优的唯一办法就是性能评测。

你还需要明白性能调优不是：

- 一堆技巧和窍门；
- 秘密武器；
- 你在项目结束时撒一把的仙粉。

对“技巧和窍门”要特别小心。JVM是一个非常复杂，并经过高度优化的环境，如果脱离了上下文，这些技巧基本都没什么用，而且可能还会带来麻烦。随着JVM在代码优化方面越来越智能，它们也很快就会过时。

性能分析实际上是种试验性的科学。你可以把代码看成是某种科学试验，有输入，会产生“输出”——性能指标表明系统执行任务的效率。性能工程师的工作是研究这些输出，并找出其中的模式。所以性能调优是统计应用的一个分支，而不是一群老太婆的闲言碎语。

本章将是你的新起点，我们会向你介绍Java性能调优实战。但这是个大课题，由于篇幅有限，我们只能把你领进门，帮助你分析其中的重要原理和标志性内容。我们也会尽量解答大部分基本问题。

- 性能为什么这么重要？
- 性能分析为什么这么难？
- JVM的哪些方面会让调优变得复杂？
- 应该如何考虑和完成性能调优？
- 哪些是导致系统迟缓的最常见原因？

我们还会介绍JVM中与性能相关的两个最重要子系统：

- 垃圾收集子系统
- JIT编译器

有了这些，你就可以开始着手解决编码时遇到的实际问题了。

我们先来快速浏览一些基本词汇，以便你可以表达并框定自己的性能问题和目标。

6.1 性能术语

为了让你充分理解本章所讨论的内容，我们会给出正规的性能概念定义。下面是性能工程师词典里最重要的一些术语：

- 等待时间 (Latency)
- 吞吐量 (Throughput)
- 利用率 (Utilization)
- 效率 (Efficiency)
- 容量 (Capacity)
- 扩展性 (Scalability)
- 退化 (Degradation)

Doug Lea讨论这些术语时都是放在多线程代码的上下文中，但我们要考虑的范围更广：从一个多线程处理器到整个集群服务器平台。

6.1.1 等待时间

等待时间是在给定工作量下处理一个任务单元所消耗的时长。通常，都是在工作量“正常”的情况下提到等待时间的。但有价值的性能评测一般都是用一张图形来显示在工作量不断增加的情况下等待时间随之改变的函数关系。

图6-1显示了在工作量增加时，某一性能指标（比如等待时间）出现了一个突发的非线性退化。这通常被称为性能肘。

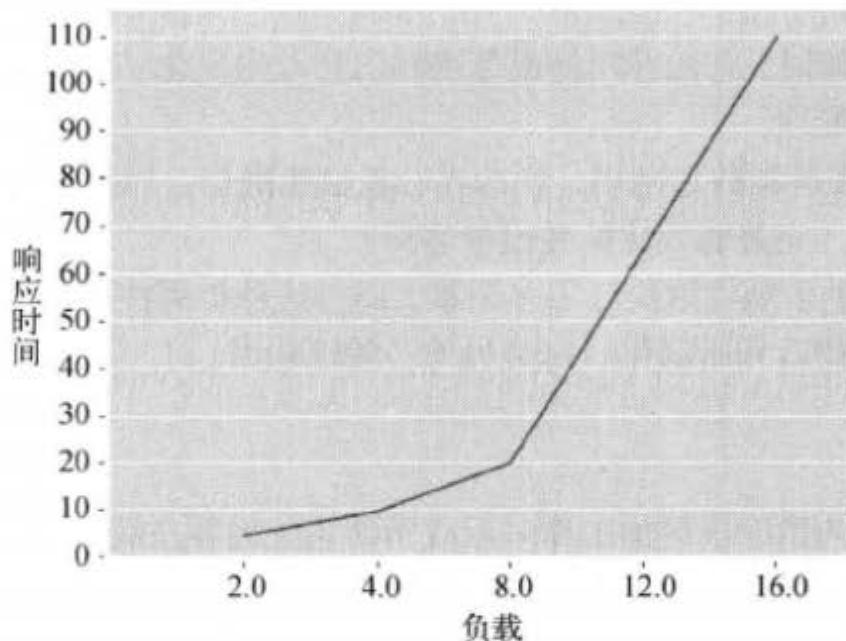


图6-1 性能肘

6.1.2 吞吐量

吞吐量是系统在限定资源、限定期长内能完成的单位工作量。用的最多的是在某一参考平台（比如指明了硬件配置、操作系统和软件环境的特定品牌服务器）上的每秒事务处理数。

6.1.3 利用率

利用率表示可用资源中用来处理工作单元（而不是清理任务或处于空闲状态）的资源百分比。人们通常会说服务器的利用率是10%，这其实是说在正常处理时间内处理工作单元的CPU百分比。注意，不同资源的利用率水平可能有非常大的差异，比如CPU和内存之间。

6.1.4 效率

系统的效率等于吞吐量除以所用资源。一个用更多资源产生相同吞吐量的系统效率更差。

比如比较两个集群方案。如果为了达到相同的吞吐量，方案A需要的服务器数量是方案B的两倍，则方案A的效率是B的一半。

别忘了，资源也可以用成本来衡量——如果方案X的成本是方案Y的两倍或需要两倍的员工运行生产环境，则方案X的效率是Y的一半。

6.1.5 容量

容量是任一时刻能通过系统的工作单元（比如事务）数量。也就是在特定的等待时间或吞吐量下，能够得到同步处理的工作单元数量。

6.1.6 扩展性

当系统得到更多资源时，它的吞吐量或等待时间会发生变化。这种发生在吞吐量或等待时间上的变化就是系统的扩展性。

如果方案A可用的服务器数量翻倍，它的吞吐量也能翻倍，那我们就说它实现了完美的线性扩展。在大多数情况下，完美的线性扩展很难达到。

还应该注意，系统的扩展性取决于很多因素，而且这种扩展性还是变化的。系统能以线性方式向上扩展到某一点，然后开始退化。这是另外一种性能肘。

6.1.7 退化

如果在不增加资源的情况下增加工作单元或网络系统的客户端，一般等待时间或吞吐量都会发生变化。这是系统在负载增加时出现的退化。

正面退化与负面退化

在正常情况下，退化是负面的。也就是说给系统增加工作单元会对性能产生负面影响，比如导致处理等待时间变长。但某些情况下退化也有可能是正面的。

比如说，如果过重的负载导致系统某些部分超过了阈值，迫使系统切换到高性能模式，这会让系统工作效率更高，缩短处理时间，尽管还要完成更多工作。JVM是个动态性非常强的运行时系统，并且有几部分可以达成这种效果。

前面这些术语是最常用的性能指标，当然还有其他一些重要的指标，但这些是指导系统性能调优的基本统计数据。在下一节中，我们会给出一个以密切关注这些数值为基础，并尽可能定量的性能调优方法。

6.2 务实的性能分析法

许多开发人员在接到性能分析任务时，脑子里都不清楚他们要通过分析得到什么。所有开发人员或经理在开始做这件事时经常只是模模糊糊地感觉代码“应该跑得更快”。

但这是彻底的倒退。要进行真正有效的性能调优，在开始做任何技术类工作之前，你应该先认真考虑下面这些问题并找出答案。

- 你正在测量的代码有哪些可观测的环节?
- 如何测量那些可观测环节?
- 这些可观测环节的目标是什么?
- 你怎么判断性能调优是否做好了?
- 性能调优可接受的最大支出是多少(按开发人员投入的时间和增加的代码复杂度计算)?
- 在优化的过程中,哪些东西是你不能舍弃的?

最重要的,也是我们要反复强调的,就是你必须测量。你至少得测量一个可观测环节,才算得上是在做性能分析。

当你开始测量代码,便经常会发现事情并非你想的那样。很多性能问题的根源可能是一个丢失的数据库索引,或者有争议的文件系统锁。在优化代码时,你应该时刻牢记代码很可能不是问题的关键。为了定量分析问题,你首先需要知道自己在测量什么。

6.2.1 知道你在测量什么

做性能调优必须测量一些东西。如果你没有测量可观测环节,就不能算做性能调优。坐在那里盯着代码,希望脑子里蹦出一个可以更快解决问题的方法,这可不是性能分析。

6

提示 要成为优秀的性能工程师,你必须知道平均数、中位数、模式、方差、百分位数、标准差、样本大小、正态分布等这样一些术语。如果还不熟悉这些概念,最好现在就到网上搜搜,如果有必要的话,认真看看搜出来的内容。

做性能分析最重要的是知道哪个可观测环节(上节介绍的)最重要。你应该总是把测量结果、目标和结论跟一个或多个基本可观测环节结合起来。

这里有些常见的可观测项,都是性能调优的好对象。

- 方法handleRequest()运行所需的平均时间(启动完成之后)。
- 并发客户端数量为10时,系统等待时间的第90个百分位数。
- 把并发用户数从1增长到1000时,响应时间的退化。

以上这些都是工程师想要测量的代表性数值,并很有可能需要优化。想得到准确又有用的数值,必须掌握基本的统计学知识。

知道你要测量什么,对数值的准确性有信心是性能调优的第一步。但模糊或随意的目标通常没什么好结果,性能调优也是如此。

6.2.2 知道怎么测量

要精确确定一个方法或其他代码片段运行需要多长时间,只有两种方法:

- 直接测量,在类源码中插入测量代码;
- 在类加载时把类转换成受测类。

大多数简单直接的性能测量技术都依赖于以上其中一种或全部技术。

还应该提一下JVM工具接口（JVMTI），用它可以创建非常复杂的分析器，但它也有缺陷。它需要性能工程师编写本地代码，并且它产生的分析数值本质上是统计平均值，而不是直接测量结果。

直接测量

直接测量是最容易理解的技术，但它是侵入式的。最简单的是像下面这种形式：

```
long t0 = System.currentTimeMillis();
methodToBeMeasured();
long t1 = System.currentTimeMillis();
long elapsed = t1 - t0;
System.out.println("methodToBeMeasured took " + elapsed + " millis");
```

这段代码会输出methodToBeMeasured()精确到毫秒的运行时长。很不方便的是，你要到处添加这种代码，而且随着测量结果不断增多，代码很容易被数据淹没。

除此之外还有其他问题，如果methodToBeMeasured()运行时长不足一毫秒会出现什么情况？稍后我们就会看到，此外还值得注意的是冷启动效果——后运行的方法可能比先运行的快。

通过类加载自动测量

我们在第1章和第5章讨论过如何把类编译成可执行程序。其中一个关键步骤是在加载字节码时进行转换。这个特性非常强大，是很多现代Java平台的核心技术。其中一个简单的例子就是方法的自动测量。

在这种方法中，特殊的类加载器加载methodToBeMeasured()所属类，在方法开始和结束的地方加上记录方法进入和退出时间的字节码。这些时间通常会被写入共享的数据结构，由其他线程访问。这些线程一般会将数据写入日志文件，或者通过网络交给负责处理原始数据的服务器。

很多高端的性能监测工具（比如OpTier CoreFirst）都是以这项技术为核心的。但在编写本书时，这个市场上似乎还没有开源工具。

注意 我们会在后面讨论到，Java方法开始时需要进行解释，然后才切换到编译模式。要得到真正的性能指标结果，你必须去掉解释模式占用的时间，因为它们会严重扭曲真实结果。后面还会给出更多细节，告诉你如何确定方法切换为编译模式的时间。

你可以用这两项技术（其一或全部）找出某一方法执行所需的时长。下一个问题，完成调优之后，你想得到什么样的数值？

6.2.3 知道性能目标是什么

清晰的目标能让人注意力集中，所以了解和传达优化的最终目标（知道要测量什么）至关重要。大多数情况下，这个目标简单而明确，比如：

- 将10个并发用户的端到端等待时间的第90个百分位数减少20%；
- 将handleRequest()的平均等待时间减少40%，方差减少25%。

在一些更复杂的情况下，目标可能由几个相关的性能目标共同构成。你要知道，你所测量和想要优化的独立可观测项越多，调优工作就会变得越复杂。优化一个性能目标可能会对其他性能目标产生负面影响。

有时，在设定目标之前你很有必要做些初步分析，比如在确定要让方法运行得更快这一目标之前，应该先确定哪些方法最重要。这很好，但经过初步探索后，你最好停下来再确认一下目标，然后再达成它们。开发人员非常爱犯只顾低头拉车，不顾抬头看路的错误。

6.2.4 知道什么时候停止优化

理论上来说，知道什么时候停止优化并不难——达成目标之时就是任务完成之日。然而实际中人们很容易陷入性能调优的泥淖。如果事情进展顺利，你肯定想要继续前进并做得更好。而如果不太顺利，你为了达成目标就会不断尝试新策略。

要想知道什么时候停止优化，你需要对目标有清醒的认识并理解它们的价值。能达成性能目标的90%通常就足够了，你还可以利用节省下来的时间去做些别的事。

还要考虑一点，你要看看有多少工作投入到了极少用到的代码路径上。通过优化代码来减少程序运行时长的1%（甚至更少）完全是在浪费时间，但奇怪的是做这种事儿的开发人员数量惊人。

至于该优化什么，这里有一组非常简单的指导规则。你可能需要根据自身情况进行调整，但它们的适用范围很广泛：

- 优化那些重要，而不是最容易的代码。
- 首先优化那些最重要（通常是调用最频繁）的方法。
- 在遇到那些唾手可得的优化时，把它办了，但要清楚代码的调用频率。

最后再做一轮测量工作。如果还没达成性能目标，你就需要清查一下，看看离命中目标还有多大差距，以及取得的成绩是不是已经对整体性能产生了你所期望的影响。

6.2.5 知道高性能的成本

所有性能调整都贴着价签。

- 分析和优化代码要占用的时间（在任何软件项目中，开发人员的时间基本都是最大的开支）。
- 所做的调整可能会引入额外的技术复杂度（也有简化代码的性能优化，但它们不是主流）。
- 为了让主处理线程运行得更快，可能会引入额外的线程来执行辅助任务，但这些线程可能会在负载较高时对系统整体产生不可预料的影响。

不管是什高价签，你都要重视，并尽量在完成第一轮优化之前找到它们。

这有助于你了解提高性能的最大可接受成本。这个成本可能是设定开发人员调优的时间限制，额外的类数或代码行数。比如说，开发人员决定花在优化上的时间不能超过一个星期，或者因优化而生的类增长不应该超过100%（即大小变成原来的两倍）。

6.2.6 知道过早优化的危险

关于优化，Donald Knuth有段著名的评论：

程序员浪费了大量时间考虑，或担心程序中无关紧要部分的速度，并且那些尝试改进效率的行为实际上有很强的负面影响……过早优化是万恶之源。^①

这段话在业内引起了广泛争论，而且人们通常只记住了最后一句。这之所以令人感到遗憾，有如下原因。

- 在评论的前段，Knuth含蓄地提醒我们要测量，没有测量就不能确定程序的关键部分。
- 我们再次提醒你，可能不是代码导致等待时间过长——环境中的其他部分也会产生等待时间。
- 在完整的评论中，很容易看出Knuth是在谈论那些有意识的、齐心协力的优化。
- 这段评论的简短版让它变成了不良设计或糟糕执行选择的相当巧合的借口。

有些优化体现在良好的编码风格上：

- 不要分配不需要的对象。
- 如果再也不需要调试日志，就去掉它。

我们在下面的代码中加了一个检查，看日志对象是否处理调试日志。这种检查被称为日志守卫。如果日志子系统被设置为不处理调试日志，这段代码就不会构造日志消息，省掉了为了日志消息而调用currentTimeMillis()和构造StringBuilder对象的开销。

```
if (log.isDebugEnabled()) log.debug("Useless log at: " +
    System.currentTimeMillis());
```

但如果调试日志真的没有用，我们可以把这段代码一并去掉，就能再节省两个处理器周期(日志守卫的开销)。

性能调优的工作之一就是从一开始就写出质地优良、高效运行的代码。更好地认识Java平台，知道它的底层运行机制（比如理解在合并两个字符串时隐含的对象分配），并在编码时考虑到性能问题，才能写出更好的代码。

现在我们有了框定性能问题和目标的基本词汇，还有如何解决问题的方法大纲。但我们还没解释为什么这是软件工程师会遇到的问题，以及这种需求来自哪里。要弄懂这个，我们有必要简单了解一下硬件的世界。

6.3 哪里出错了？我们担心的原因

在几年前，性能问题看起来并不重要。时钟速度不断上升，所有软件工程师只要多等几个月，哪怕是写得很烂的代码，也能借助节节攀升的CPU速度表现出优异性能。

^① Donald E. Knuth，“带go to语句的结构化编程”，计算调查，6，no.4（1974年12月）。http://pplab.snu.ac.kr/courses/adv_pl05/papers/p261-knuth.pdf。

那么，事情怎么会错得这么离谱？为什么时钟速度的提升不再那么快了？更让人担忧的是，为什么有3GHz芯片的电脑看起来比2GHz芯片的快不了多少？行业中软件工程师需要考虑性能问题的这种趋势是从哪里来的？

我们会在本节讨论引导这股趋势的力量，以及连最纯粹的软件开发人员也需要了解一点硬件知识的原因。我们会为本章剩下的内容打好基础，让你理解JIT编译的概念和一些有深度的例子。

你可能听说过“摩尔定律”。很多开发人员都知道这个定律与提高计算机速度有关，但对于具体细节不甚了了。我们来解释一下它到底是什么意思，以及它在不久的将来可能带来的影响。

6.3.1 摩尔定律——过去和未来的性能趋势

摩尔定律是Gordon Moore提出来的，他是Intel的创始人之一。该定律最常见的形式之一是：晶片上的晶体管数量每两年翻一番是合算的。

这个定律实际上是对计算机处理器（CPU）发展趋势的一种看法，基于Gordon Moore在1965年发表的一篇论文，最初是对未来10年进行预测——也就是直到1975年。而这一预测直到现在仍然能够应验（据预测其在2015年以前都有效），实在值得称道。

我们在图6-2中绘制了Intel x86家族从1980年发展到2010年的i7整个历程中的一些真实CPU。图中显示了不同时期发布的芯片所集成的晶体管数量。

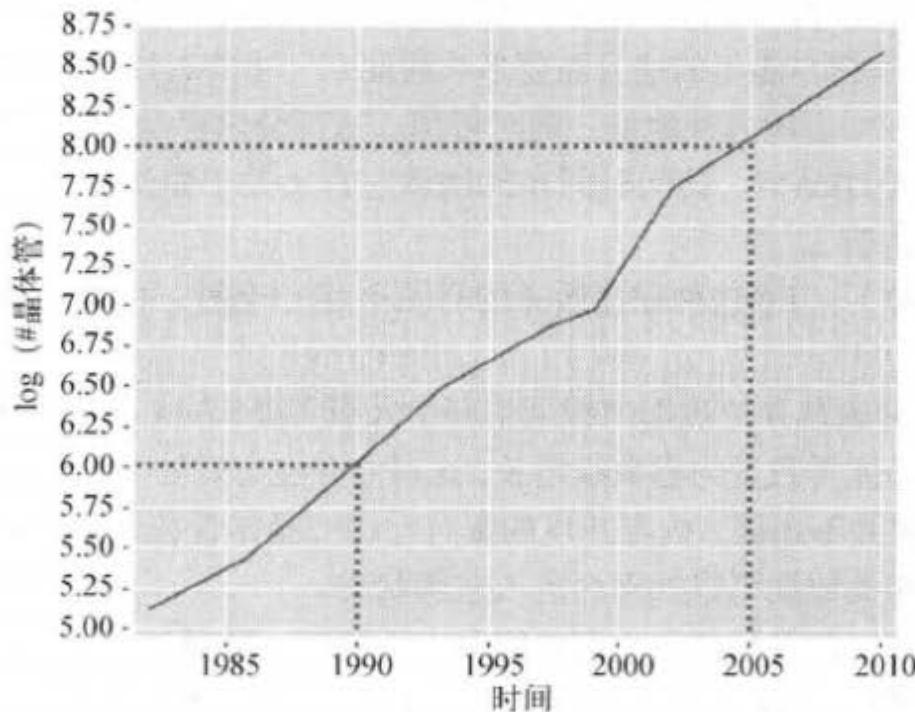


图6-2 随时间变化的晶体管数量对数线性图

这是一个对数线性图，所以y轴上的每次增长都是前一个值的10倍。如你所见，这条线基本是直的，而且每隔六或七年就会穿过一个垂直层级。这证明了摩尔定律的准确性，因为六或七年增长十倍和每隔两年翻一番基本是一致的。

图中y轴的刻度是对数，这就是说Intel在2005年生产的主流新品大概有一亿个晶体管。这是1990年生产的芯片上的晶体管数量的100倍。

一定要注意摩尔定律特别谈到了晶体管数量。要知道，单凭摩尔定律不足以让软件工程师继续从硬件工程师那里得到好处，理解这一点是最基本的要求。

注意 晶体管数量和时钟速度不是一回事儿，人们一般会认为更高的时钟速度就意味着更好的性能，其实这是一种过于草率的想法。

摩尔定律在过去很有指导意义，并且在未来一段时间内应该仍然准确（有不同的估算，但在2015年前看起来是合理的）。但摩尔定律计算的是晶体管数量，用这个数值来指导开发人员提高代码性能越来越不靠谱。实际上，你会发现情况要更复杂。

在实际操作中，性能是由一系列因素共同决定的，而且每个因素都很重要。然而如果硬让我们从里面挑一个，应该是定位指令与数据运行速度的相关性。这个概念对性能非常重要，我们会深入了解。

6.3.2 理解内存延迟层级

计算机处理器要处理数据。如果它要处理的数据到不了，CPU时钟多快都没用——它只能等着，执行空操作（NOP），在数据到来之前基本就处于停转状态。

也就是说在解决延迟时，最重要的两个问题是，“CPU核心要处理的数据的最近的副本在哪里？”，还有“把它送到核心能用的地方需要多长时间？”主要有以下几种答案。

- **寄存器**：这是CPU上的内存地址，随时可用。这部分内存是指令直接操作的。
- **主存**：这一般是DRAM。访问时间在50纳秒左右（关于如何使用处理器缓存避免这段延迟请参见后续内容）。
- **固态磁盘（SSD）**：访问这种磁盘所需的时间不足0.1毫秒，但跟传统硬盘比起来，它们要便宜一些。
- **硬盘**：访问这种磁盘并把数据加载到主存中大概需要5毫秒。

摩尔定律预测了晶体管数量的指数级增长，这对内存也有好处——内存访问速度也能以指数级增长。但这两种指数并不相同。内存速度的提升比CPU晶体管数量的增长要慢得多，这意味着核心迟早会因为没有相关数据可以处理而落入空闲状态。

为了解决这个问题，在寄存器和主存之间引入了缓存。缓存是少量更快的内存（SRAM，而不是DRAM）。这种更快的内存成本要比DRAM高很多（无论是金钱还是晶体管），所以计算机不全用SRAM作为内存。

缓存分为一级缓存（L1）和二级缓存（L2）（某些机器还有L3），数值表明缓存到CPU的距离（越近越快）。我们会在6.6节（在JIT编译上）详细讨论缓存，并给出一个例子来表明L1缓存对运行代码的重要影响。图6-3展示了L1和L2缓存比主存快多少。之后，我们还会给出一个例子来阐明这些速度差异对运行代码的性能有什么影响。



图6-3 寄存器、处理器缓存和主存的相对访问时间

除了增加缓存，20世纪90年代到21世纪早期大量使用了另外一种技术解决内存延迟的问题，就是增加处理器的功能，这使得处理器越来越复杂。即便CPU处理能力和内存延迟之间的差距越来越大，也仍然采用复杂的硬件技术来保证CPU有数据可以处理，比如指令级并行（ILP）和芯片多线程（CMT）。

这些技术的出现消耗了CPU晶体管预算中的大部分，并且它们使真实性能收益递减。这一趋势导致了新观点的出现，即在未来设计带有多个（或很多）核心的CPU芯片。

这意味着未来的性能和并发密切相关——主要办法之一就是通过拥有更多核心让系统整体性能得到提升。那样，即便有一个核心在等待数据，其他核心也可以继续工作。这种关系十分重要，所以我们要一再提起。

- 将来的CPU是多核的。
- 性能和并发绑在一起变成了相同的关注点。

Java程序员除了要关注硬件，还要注意JVM特性带来了额外的复杂性。下一节我们就来看一下这些内容。

6.3.3 为什么 Java 性能调优存在困难

在JVM或其他任何受控运行时环境上做性能调优天生就比在非受控环境下做调优困难。这是因为C/C++程序员几乎所有事情都要自己做。OS只提供很少的服务，比如基本的线程调度。

在受控系统中，基本观点是让运行时来控制环境，不用开发人员自己处理所有细节。这能提高程序员的生产率，但要放弃某些控制权。另外一种选择是放弃受控运行时提供的所有便利，但和性能调优所做的工作相比，这个代价实在太高了。

造成调优困难的平台特性主要是：

- 线程调度；
- 垃圾收集（GC）；
- 即时（JIT）编译。

这些特性能以很巧妙的方式交互。例如，编译子系统用计时器来决定编译哪个方法。也就是说等待编译的候选方法集可能会受到调度和GC等特性的影响。每次运行时所编译的方法可能都不同。

正如你在本节中看到的，准确测量是性能分析决策过程的关键。如果你决定认真对待性能调优，那么理解Java平台中处理时间的细节和限制就非常有用。

6.4 一个来自于硬件的时间问题

你有没有想过计算机里的时间存在哪里以及在哪里处理？我们都知道硬件最终负责跟踪时间，但事实可能不像你想的那么简单。

为了进行性能调优，你需要对时间如何工作有深刻的认识。为此我们先从底层硬件开始讨论，然后探讨Java如何与这些子系统集成，最后介绍**nanoTime()**方法的复杂性。

6.4.1 硬件时钟

在基于x64的机器里有四种不同的硬件时间源：RTC、8254、TSC以及HPET。

实时时钟（RTC）基本上和便宜的电子表（基于石英晶体）里找到的电子器件一样，在系统断电时由主板上的电池供电。系统在启动时就是从它那里得到时间的，不过很多机器在OS启动过程中会通过网络时间协议（Network Time Protocol，NTP）跟网络上的时间服务器同步。

所有古董都曾是新东西

实时时钟这个名字现在看来十分不恰当——在20世纪80年代它刚出现时确实被认为是实时的，但现在它的准确度对于关键应用来说已经不够用了。以“新”或“快”命名的创新经常是这种结局，比如巴黎的Pont Neuf（“新桥”）。它建于1607年，现在已经是巴黎市内最古老的桥了。

8254是可编程计时芯片，也是始祖级的东西。它的时钟源是一个119.318kHz的晶体，这个频率是NTSC彩色副载波频率的三分之一，这也是它返回到CGA图形系统的原因。它曾经为OS调度器提供定期时点（用于时间片），但现在已经有其他时间源（或者不再需要）了。

下面介绍应用最广泛的现代计时器——时间戳计时器（TSC）。基本上，这是一个跟踪CPU运行了多少个周期的CPU计数器。乍看起来它似乎很适合做时钟。但这个计数器是跟CPU的，并且在运行时可能会受到节能或其他因素的影响。也就是说，不同的CPU会互相偏离，也不能跟钟表时间保持一致。

最后还有高精度事件计时器（HPET）。这种计时器是最近几年才出现的，有助于人们用较老的时钟硬件更好地计时。HPET使用至少10MHz的计时器，所以其精度至少应该是 $1\mu\text{s}$ ——但它并不是在所有硬件上都可用，也不是所有操作系统都支持。

如果这些内容看起来有点乱，那是因为它们本来就乱。好在Java平台提供了可以使用它们的工具——它把对硬件和OS支持的依赖隐藏到特定的机器配置里。然而试图隐藏依赖项的做法并没有完全成功。

6.4.2 麻烦的 **nanoTime()**

Java中有两个获取时间的方法：`System.currentTimeMillis()` 和 `System.nanoTime()`，后面一个用于测量比毫秒更精确的时间。表6-1总结了它们两个的主要差异。

表6-1 Java内置时间获取方法的比较

currentTimeMillis()	nanoTime()
解析度为毫秒级	纳秒级引用
几乎所有情况下都跟钟表时间相符	可能偏离钟表时间

如果表6-1中对nanoTime()的描述让它看起来有点像计时器，那就对了，因为如今在大多数操作系统上，它的时间源都是CPU计数钟——TSC。

nanoTime()的输出是相对于某个固定时间的。也就是说必须用它记录间隔期，用nanoTime()的返回结果减去之前调用得到的返回结果。下面这段代码来自后面的一个研究案例，恰好表明了这种情况：

```
long t0 = System.nanoTime();
doLoop1();
long t1 = System.nanoTime();
...
long el = t1 - t0;
```

el是doLoop1()执行所用的时间(以纳秒为单位)。

要在性能调优中正确使用这些方法，必须对nanoTime()的行为有所了解。代码清单6-1输出了毫秒计时器和纳秒计时器(通常由TSC提供)之间的最大偏移。

6

代码清单6-1 时间偏移

```
private static void runWithSpin(String[] args) {
    long nowNanos = 0, startNanos = 0;
    long startMillis = System.currentTimeMillis();
    long nowMillis = startMillis;
    while (startMillis == nowMillis) {           ← 将startNanos在
        startNanos = System.nanoTime();          毫秒边界上对齐
        nowMillis = System.currentTimeMillis();
    }
    startMillis = nowMillis;
    double maxDrift = 0;
    long lastMillis;
    while (true) {
        lastMillis = nowMillis;
        while (nowMillis - lastMillis < 1000) {
            nowNanos = System.nanoTime();
            nowMillis = System.currentTimeMillis();
        }
        long durationMillis = nowMillis - startMillis;
        double driftNanos = 1000000 *
            ((double)(nowNanos - startNanos)) / 1000000 - durationMillis;
        if (Math.abs(driftNanos) > maxDrift) {
            System.out.println("Now - Start = " + durationMillis
                + " driftNanos = " + driftNanos);
            maxDrift = Math.abs(driftNanos);
        }
    }
}
```

这段代码会输出可观测到的最大偏移，并且证明其表现与操作系统的相关度很高。下面是Linux上的一段输出：

```
Now - Start = 1000 driftNanos = 14.99999996212864
Now - Start = 3000 driftNanos = -86.99999989403295
Now - Start = 8000 driftNanos = -89.00000011635711
Now - Start = 50000 driftNanos = -92.00000204145908
Now - Start = 67000 driftNanos = -96.0000033956021
Now - Start = 113000 driftNanos = -98.00000407267362
Now - Start = 136000 driftNanos = -98.99999713525176
Now - Start = 150000 driftNanos = -101.0000123642385
Now - Start = 497000 driftNanos = -2035.000012256205
Now - Start = 1006000 driftNanos = 20149.99999664724
Now - Start = 1219000 driftNanos = 44614.00001309812
```

注意driftNanos
从-2035到20149
出现了一个非常
大的跳跃

这里还有一个装在相同硬件上的老Solaris上的输出结果：

```
Now - Start = 1000 driftNanos = 65961.0000000157
Now - Start = 2000 driftNanos = 130928.0000000399
Now - Start = 3000 driftNanos = 197020.999999497
Now - Start = 4000 driftNanos = 261826.99999981196
Now - Start = 5000 driftNanos = 328105.999999343
Now - Start = 6000 driftNanos = 393130.99999981205
Now - Start = 7000 driftNanos = 458913.9999998224
Now - Start = 8000 driftNanos = 524811.9999996561
Now - Start = 9000 driftNanos = 590093.9999992261
Now - Start = 10000 driftNanos = 656146.9999996916
Now - Start = 11000 driftNanos = 721020.0000008626
Now - Start = 12000 driftNanos = 786994.000000497
```

间隔很平滑

注意看最大值的增长，在Solaris上很稳定，而在Linux上相当一段时间内看起来都OK，然后出现了大的跳跃。我们在选择示例代码时相当认真，尽量避免创建额外的线程，甚至对象，以将平台的干预降到最低（比如说，没有对象的创建就意味着不会做垃圾收集），但即便如此，我们还是能看到JVM的影响。

最终证实Linux时序上出现的跳跃是由不同CPU上的TSC计数器之间的差异造成的。JVM会定期挂起正在运行的Java线程，并将它迁移到不同核心上。所以程序代码会见到不同CPU计数器上的差异。

这就是说对于间隔较长的时间，`nanoTime()`基本上是不可信的。只能用它测量较短的时间间隔，较长（宏观）的时间间隔应该用`currentTimeMillis()`重新校准。

要充分掌握性能调优，即要有扎实的测量理论，还需要知道实现细节。

6.4.3 时间在性能调优中的作用

要做好性能调优，你必须知道该如何解读代码运行期间得到的测量记录，也就是说你必须明白在Java平台上得到的时间测量结果的局限性。

精确度

时间的量通常被冠以与其最接近的某一刻度单位。这被称为测量的精确度。比如说，测量时间经常精确到毫秒。如果重复围绕同一数值进行测量，其结果范围很小，则该计时器是精确的。

精确度是对给定测量中所包含的随机噪音量的度量。我们假定对一段代码的测量结果是正态分布的。那么精确度通常是宽度为95%的置信区间。

准确度

测量（指测量时间）的准确度是取得接近真实值的测量结果的能力。实际上，真实值一般不可知，所以准确度可能比精确度更难确定。

准确度是对测量中的系统性错误的度量。可能存在准确但不太精确的测量结果（所以基本读数是正确的，但有随机的环境噪音）。也可能存在精确但不准确的结果。

理解测量结果

一个精确到纳秒的时间间隔测量结果是用准确度为1微秒的计时器测量的，其值为5945纳秒，那真实值应该介于3945~7945纳秒之间（95%的可能性）。当心那些看起来过于精确的数据，必须随时对测量结果的精确度和准确度进行检查。

粒度

系统真正的粒度是最快计时器的频率——很可能是10纳秒范围内的中断计时器。这有时被称为可辨别能力，可以肯定“几乎一起发生，但时间不同”是两个事件最短的发生间隔。

在我们跨过操作系统、虚拟机和类库代码的不同层面时，已经不太可能辨别这些极短的时间间隔。在大多数情况下，应用程序开发人员是得不到这些特别短的时间间隔的。

分布式网络计时

我们对性能调优的大部分讨论都是以单机上的系统为中心的。但你应该知道，当涉及网络上的系统调优时，会有一些特别的问题。网络上的同步和计时并不容易，而且不仅仅是在互联网上，即便是以太网也会出现这些问题。

详细讲解分布式网络计时超出了本书的范围，但你应该知道，通常来说，很难得到用于跨越几台机器的工作流的准确时序。另外，即便NTP这样的标准协议对于高精度工作来说准确度也不够。

在开始讨论垃圾收集之前，我们先看一个前面提到过的例子——缓存对代码性能的影响。

6.4.4 案例研究：理解缓存未命中

对于很多吞吐量较高的代码来说，影响性能的一个主要因素就是一级缓存未命中的数量。

代码清单6-2中的代码操作1MB的数组，并输出执行两个循环中之一所用的时间。在第一个循环中，每隔16个条目对int数组中的元素加1。一级缓存的一个缓存行中通常有64个字节（在32位JVM上，Java的int是4个字节），所以这意味着每次会读取一个缓存行（ $64=16*4$ ）。

代码清单6-2 理解缓存未命中

```
public class CacheTester {
    private final int ARR_SIZE = 1 * 1024 * 1024;
    private final int[] arr = new int[ARR_SIZE];
    private void doLoop2() {
        for (int i=0; i<arr.length; i++) arr[i]++;
    }
}
```

处理每个条目

```

private void doLoop1() {
    for (int i=0; i<arr.length; i += 16) arr[i]++;
}

private void run() {
    for (int i=0; i<10000; i++) {
        doLoop1();
        doLoop2();
    }
    for (int i=0; i<100; i++) {
        long t0 = System.nanoTime();
        doLoop1();
        long t1 = System.nanoTime();
        doLoop2();
        long t2 = System.nanoTime();
        long el = t1 - t0;
        long el2 = t2 - t1;
        System.out.println("Loop1: " + el + " nanos ; Loop2: " + el2);
    }
}

public static void main(String[] args) {
    CacheTester ct = new CacheTester();
    ct.run();
}
}

```

处理每个缓存行
代码热身

注意，在你得到准确结果之前应该让代码热热身，以便让JVM对你感兴趣的方法进行编译。我们会在6.6节讨论更多与代码热身相关的内容。

第二个循环，doLoop2()给数组中的每个元素加1，所以看起来它做的工作是doLoop1()的16倍。下面是在笔记本上运行这段代码得到的结果：

```

Loop1: 634000 nanos ; Loop2: 868000
Loop1: 801000 nanos ; Loop2: 952000
Loop1: 676000 nanos ; Loop2: 930000
Loop1: 762000 nanos ; Loop2: 869000
Loop1: 706000 nanos ; Loop2: 798000

```

计时子系统的疑难杂症

结果中的所有纳秒值都很整齐，全是一千的整数倍。这表明底层系统调用（System.nanoTime()最终所调用的）仅仅返回了一个微秒整数值——一微秒是1000纳秒。因为这个结果是在Mac笔记本上得到的，所以我们猜测在OS X的底层系统调用只有微秒级的精度，实际上，它调用的是gettimeofday()。

从这个结果来看，doLoop2()所用的时长不是doLoop1()的16倍。这表明内存访问在总体性能配置中占有支配性地位。doLoop1()和doLoop2()读取缓存行的次数相同，而修改数据所用的CPU周期只占整体时间的一小部分。

我们先来回顾下Java时间系统的要点。

- 大多数系统内部都有几个不同的时钟。
- 毫秒计时器是安全可靠的。
- 更高精度的时间需要仔细处理以防止出现偏离。
- 你需要知道计时测量的精确度和准确度。

我们下一个将要讨论的是Java平台的垃圾收集子系统。这是性能的决定性因素中非常重要的一部分，并且它有很多可调节的部分，对于做性能分析的开发人员来说都可以成为非常重要的工具。

6.5 垃圾收集

内存自动管理是Java平台最重要的组成部分之一。在出现Java和.NET这样的托管平台之前，开发人员把大部分时间都用在追踪不完善的内存处理引发的bug上了。

然而近年来，内存自动分配技术发展的如此先进可靠，已经变得让人无法察觉了，因此大部分Java开发人员不知道Java平台的内存管理是如何完成的，不知道可以使用哪些选项，也不知道如何在框架限定内进行优化。

这说明Java的做法取得了成功。大多数开发者不知道内存和GC系统的细节是因为他们没必要知道。虚拟机在这方面做得非常棒，在处理大多数应用时都不用特别调整，所有大多数应用从没调整过。

本节我们将讨论在确实需要做些调整的情况下你能做什么。我们会给出基本原理，解释为了运行Java进程该如何处理内存，并探索标记和清除集合的基础，再讨论两个工具——jmap和VisualVM。最后介绍两个收集器——并发标记清除（Concurrent Mark-Sweep，简称CMS）和新的垃圾优先（Garbage First，简称G1）收集器。

也许你有个服务器端程序耗光了内存，或者承受着长时间中断的痛苦。在6.5.3节讨论jmap时，我们将会告诉你一个查看类是否占用大量内存的简单办法。我们还会教你使用控制虚拟机内存配置的选项开关。

先从基本算法开始吧。

6.5.1 基本算法

标准的Java进程既有栈又有堆。栈保存原始型局部变量（引用型局部变量会指向以堆方式分配的内存）。堆保存要创建的对象。图6-4展示了各种类型变量存储的位置。

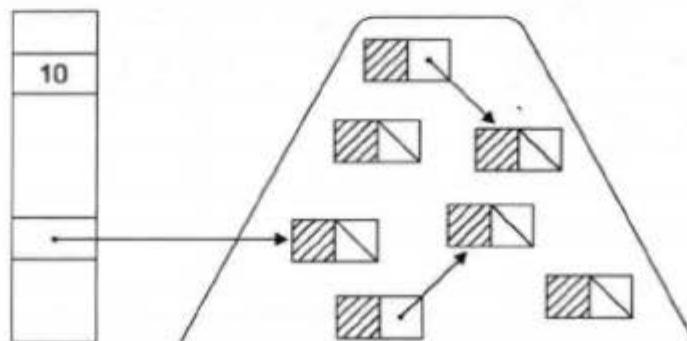


图6-4 堆和栈中的变量

注意，对象的原始型域仍然分配在堆内的地址上。Java平台对堆内存回收和再利用的基本算法被称为标记和清除，应用程序中代码已经不再使用它了。

6.5.2 标记和清除

标记和清除是最简单、也是出现最早的垃圾收集算法。业内还有其他内存自动管理技术，比如Perl和PHP等语言采用的引用计数^①，有人说它更简单，但它是不需做垃圾收集的方案。

最简单的标记和清除算法会暂停所有正在运行的线程，并从一组“活”对象——在任何用户线程的任何堆栈帧中存在引用（不管是局部变量、方法参数、临时变量，还是某些非常少见的情况）的对象——开始遍历其引用树，标记出遍历路径上的所有活对象。遍历完成后，所有没被标记的都被当做垃圾，可以回收（清除）。注意，被清除的内存不会还给操作系统，而是还给JVM。

Java平台对基本的标记清除方法进行了改进，采用“分代式垃圾收集”。在这种方法中，会根据Java对象的生命周期将堆内存划分为不同的区域。在对象的生存期内，对它的引用可能指向内存中几个不同区域（如图6-5所示）。在垃圾收集过程中，可能会将对象移动到不同区域。

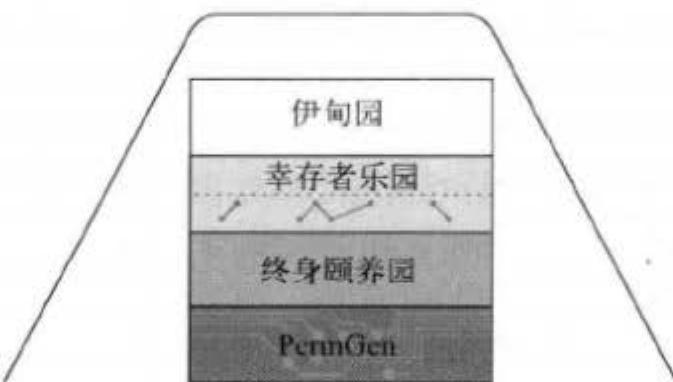


图6-5 内存中的伊甸园、幸存者乐园、终身颐养园和PermGen区

这样做是因为根据对系统运行时期的研究，发现对象的生存期或者较短，或者很长。Java平台把堆内存划分为不同区域可以充分利用对象生命周期的这种特点。

出现时长不确定的暂停怎么办？

Java和.NET经常受到这样的批评：标记和清除式的垃圾收集不可避免地会导致世界停转（所有用户线程都必须停止），而且这种暂停的时长是不确定的。

其实这个问题被夸大了。对于服务器端软件来说，应用程序不会在意垃圾收集引起的暂停。为了避免暂停或完全收集而精心制作解决方案完全是凭空想象——除非经过认真分析，发现全内存收集时间真的存在问题，才应该避免。

^① 引用计数就是为每个内存对象维护一个引用数值，当有新的引用指向该对象时则将其引用计数加一，销毁时则减一。当引用计数为零时就收回该对象占用的内存资源。这种方式虽然简单，但存在两个问题：每次内存对象被引用或引用被销毁时必须修改引用计数，造成整体性能消耗；出现循环引用时难以处理。——译者注

1. 内存区域

JVM为存储不同生命周期阶段的对象将内存分成了几个不同区域。

- 伊甸园——伊甸园是对象最初降生的堆区域，并且对大多数对象来说，这里是它们唯一存在过的区域。
- 幸存者乐园——这里通常有两个空间（或者也可以认为是被分成两半的一个空间）。从伊甸园幸存下来的对象会被挪到这里。它们有时候被称为从何而来和到哪里去。除非正在执行垃圾收集，否则总有一个幸存者空间是空的，原因会在后面给出。
- 终身颐养园——终身制空间（即老一代）是那些“足够老”的幸存对象的归宿（从幸存者空间挪过来的）。在年轻代收集过程中是不会碰终身制内存的。
- PermGen——这是为内部结构分配的内存，比如类定义。PermGen不是严格的堆内存，并且普通的对象最后不会在这里结束。

就像前面提到的，这些内存区域的垃圾收集方式也不尽相同。具体来说有两种方式：年轻代收集和完全收集。

2. 年轻代收集

年轻代收集只会清理“年轻的”空间（伊甸园和幸存者乐园）。其过程相当简单。

- 在标记阶段发现的所有仍然存活的年轻对象都会被挪走：
 - 那些足够老的对象（从次数足够多的GC中幸存下来的）进入终身颐养园；
 - 剩下那些年轻的存活对象进入幸存者乐园里空着的空间。
- 最后，伊甸园和最近腾空的幸存者乐园就可以重用了，因为它们里面已经全是垃圾了。

当伊甸园满了的时候就会触发一次年轻代收集。注意，标记阶段必须遍历整个生存对象图。也就是说如果有年轻对象被一个终身对象引用了，终身对象所持有的引用也必须被扫描到并标记上。否则只被终身对象引用的伊甸园对象可能会出问题。如果标记阶段不是全遍历，这个伊甸园对象就再也看不到了，而且不可能对它做出正确处理。

3. 完全收集

当年轻代收集不能把对象放进终身颐养园时（空间不够了），就会触发一次完全收集。根据老年代所用的收集器，这可能会牵涉到老年代对象的内部迁移。这样做是为了确保必要时能从老年代对象所占的内存中给大的对象腾出足够的空间。这被称为压缩。

4. 安全点

要想做垃圾收集，至少得让所有应用线程暂停一会儿。但是线程不可能为了GC说停就停。所以它们给执行GC留出了特定的位置——安全点。常见的安全点是方法被调用的地方（“调用点”），不过也有其他安全点。为了执行垃圾收集，所有应用程序线程都必须停在安全点上。

我们暂停一下，先介绍一个简单的工具——jmap，它能帮你弄清楚程序运行时的内存使用情况，以及所有内存都用在哪里。我们后续还会介绍一个更先进的GUIL工具，但既然很多问题都可以用非常简单的命令解决，你最好应该知道如何使用，而不是直接就使用GUIL工具。

6.5.3 jmap

Oracle JVM自带了一些简单的工具，可以帮你了解运行中的进程。jmap是其中最简单的一个，用来显示Java进程的内存映射（它也能分析Java核心文件^①，甚至能连到远程调试服务器上）。让我们回到电子商务服务器端应用程序的例子上，用jmap对它进行一番探索。

1. 默认视图

jmap最简单的用法是查看连接到进程里的本地类库。除非你的应用程序里有很多JNI代码，否则这种用法通常没什么用，但我们还是会演示一下，以免你忘了指定jmap选项时被它搞糊涂：

```
$ jmap 19306
Attaching to process ID 19306, please wait...
Debugger attached successfully.
Server compiler detected.
JVM version is 20.0-b11
0x08048000 46K      /usr/local/java/sunjdk/1.6.0_25/bin/java
0x55555000 108K     /lib/ld-2.3.4.so
... some entries omitted
0x563e8000 535K     /lib/libnss_db.so.2.0.0
0x7ed18000 94K      /usr/local/java/sunjdk/1.6.0_25/jre/lib/i386/libnet.so
0x80cf3000 2102K    /usr/local/kerberos/mitkrb5/1.4.4/lib/
    libgss_all.so.3.1
0x80dcf000 1440K    /usr/local/kerberos/mitkrb5/1.4.4/lib/libkrb5.so.3.2
```

一般用得比较多的是-heap和-histo选项，下面我们就来讨论这两个选项。

2. 堆视图

使用-heap选项时，jmap会抓取进程当前的堆快照。在输出结果中能看到构成Java进程堆内存的基本参数。

堆的大小是年轻代、老年代加上PermGen区的总和。但在年轻代内部有伊甸园和幸存者乐园，并且我们还没告诉你这些区域的大小之间有什么关系。这些区域的相对大小是由一个叫做幸存比例的数值决定的。

我们来看一些输出样例。你能在其中看到伊甸园、幸存者乐园（标签为From和To）、终身颐养园（old Generation）以及一些相关信息：

```
$ jmap -heap 22186
Attaching to process ID 22186, please wait...
Debugger attached successfully.
Server compiler detected.
JVM version is 20.0-b11
using thread-local object allocation.
Parallel GC with 13 thread(s)
```

^① Java核心文件（Java core file）主要保存各应用线程在某一时刻的运行位置，即JVM执行到哪个类、哪个方法及哪一行上。它是一个文本文件，打开后可以看到每一个线程的执行栈，以及stack trace的显示。一般Java程序遇到致命问题，在JVM死掉之前会产生两个文件，其中就有Java核心文件，另一个是HeapDump文件。有时为了调试或查找性能问题也会手工生成这两个文件。——译者注

```

Heap Configuration:
  MinHeapFreeRatio = 40
  MaxHeapFreeRatio = 70
  MaxHeapSize      = 536870912 (512.0MB)
  NewSize          = 1048576 (1.0MB)
  MaxNewSize       = 4294901760 (4095.9375MB)
  OldSize          = 4194304 (4.0MB)
  NewRatio         = 2
  SurvivorRatio    = 8
  PermSize         = 16777216 (16.0MB)
  MaxPermSize     = 67108864 (64.0MB)

Heap Usage:
PS Young Generation
Eden Space:
  capacity = 163774464 (156.1875MB)
  used     = 58652576 (55.935455322265625MB)
  free     = 105121888 (100.25204467773438MB)
  35.81301661289516% used
From Space:
  capacity = 7012352 (6.6875MB)
  used     = 4144688 (3.9526824951171875MB)
  free     = 2867664 (2.7348175048828125MB)
  59.10553263726636% used
To Space:
  capacity = 7274496 (6.9375MB)
  used     = 0 (0.0MB)
  free     = 7274496 (6.9375MB)
  0.0% used
PS Old Generation
  capacity = 89522176 (85.375MB)
  used     = 6158272 (5.87298583984375MB)
  free     = 83363904 (79.50201416015625MB)
  6.87904637170571% used
PS Perm Generation
  capacity = 30146560 (28.75MB)
  used     = 30086280 (28.69251251220703MB)
  free     = 60280 (0.05748748779296875MB)
  99.80004352072011% used

```

伊甸园= (From+To) *
幸存比例

伊甸园= (From+To) *
幸存比例

To空间当前为空

尽管空间的基本构成可能会非常有用，但在这副图里看不到堆里面有什么。如果能看到是哪些对象占用了内存中的空间，你就知道内存都到哪里去了。jmap恰好提供了一个柱状图模式，可以让你看到这些数据的简单统计结果。

3. 柱状视图

柱状视图显示了系统中每个类型的实例（还有一些内部实体）占用的内存量。各个类型按使用内存多少排列，这样就比较容易看到最大的内存猪。

当然，如果所有内存都交给了框架和平台类，这里可能就没你什么事了。但如果真有一个你的类，有了这些信息便能更好地干预它的内存占用。

小小的警告：jmap使用类型内部名称。比如字符数组会写成[C，类对象的数组会显示。

```
$ jmap -histo 22186 | head -30
num  #instances   #bytes  class name
-----
1:    452779  31712472  [C
2:     76877  14924304  [B
3:    20817  12188728  [Ljava.lang.Object;
4:     2520  10547976  com.company.cache.Cache$AccountInfo
5:    439499  9145560  java.lang.String
6:    64466  7519800  [I
7:    64466  5677912  <constMethodKlass>
8:    96840  4333424  <methodKlass>
9:    6990  3384504  <symbolKlass>
10:   6990  2944272  <constantPoolKlass>
11:   4991  1855272  <instanceKlassKlass>
12:   25980  1247040  <constantPoolCacheKlass>
13:   17250  1209984  java.nio.HeapCharBuffer
14:   13515  1173568  [Ljava.util.HashMap$Entry;
15:   9733  778640  java.lang.reflect.Method
16:   17842  713680  java.nio.HeapByteBuffer
17:   7433  713568  java.lang.Class
18:   10771  678664  [S
19:   1543  489368  <methodDataKlass>
20:   10620  456136  [[I
21:   18285  438840  java.util.HashMap$Entry
22:   9985  399400  java.util.HashMap
23:   13725  329400  java.util.Hashtable$Entry
24:   9839  314848  java.util.LinkedHashMap$Entry
25:   9793  249272  [Ljava.lang.String;
26:   11927  241192  [Ljava.lang.Class;
27:    6903  220896  java.lang.ref.SoftReference
```

VM 内部对象
和类型信息

因为在柱状图模式下输出的数据很多，所以上面只显示了输出内容的一部分。你可能要用 grep或其他工具来查看柱状图视图，找到感兴趣的细节。

输出中有很多占用内存的 [C 实体。字符数组数据经常出现在 String 对象里（字符串的内容就存在那里），所以这不奇怪——大多数 Java 程序里都有很多字符串。但从柱状图中还能看出其他有趣的事情。先来看看下面两个。

前几个实体里唯一一个应用类是 Cache\$AccountInfo——其他全是平台或框架类型——所以它们是开发人员可以完整控制的最重要的类型。 AccountInfo 对象占了很多空间——大概 2 500 个实体占了 10.5 MB（或者每个账号占 4 KB）。对于账号细节来说这实在是很多。

这个信息真的非常有用。你已经知道代码里什么占内存最多了。假如老板现在过来告诉你，因为大规模促销，一个月内系统客户数可能要暴增 10 倍。你知道这可能会给系统增加很多压力—— AccountInfo 对象可是个凶猛的家伙。虽然你有点担心，但至少你已经开始分析这个问题了。

jmap 输出的信息可以作为潜在问题处理决策流程的辅助输入。你是不是应该把账号缓存分开，减少该类型保存的信息项，或者买更多的内存给服务器装上。在做出任何决定之前，你还要做很多分析工作，但已经有个起点了。

柱状图模式下还能看到其他有意思的事情，这次指定-histo:live选项。这是告诉jmap只处理存活对象，而不是整个堆（jmap默认会处理所有对象，也包括还没被收集的垃圾）。让我们看看这次输出什么：

```
$ jmap -histo:live 22186 | head -7
num      #instances      #bytes  class name
-----
1:          2520       10547976  com.company.cache.Cache$AccountInfo
2:          32796        4919800  [I
3:          5392        4237628  [Ljava.lang.Object;
4:         141491        2187368  [C
```

注意输出的变化——字符数据已经从31MB降到了2MB左右了，证明你第一次看到的String对象里有将近三分之二都是等待回收的垃圾。然而账号对象全是活的，进一步证明了它们是消耗内存的主要力量。

使用jmap时应该稍微谨慎点。进行该操作时JVM还在运行（如果你不走运，还有可能在读取快照期间做了垃圾回收），所以你应该多运行几次，特别是在你看到任何奇怪或太好的结果时。

产生离线导出文件

jmap能创建导出文件，像这样：

```
jmap -dump:live,format=b,file=heap.hprof 19306
```

导出结果可以用来做离线分析，可以留给jmap以后自己用，也可以留给Oracle的jhat（Java堆分析工具）做高级分析。可惜我们没办法在这里全面讨论。

使用jmap可以看到一些基本设置和程序的内存占用。然而要做性能调优，一般需要对GC子系统有更多控制，其标准方式是通过命令行参数，我们来看一些控制JVM的参数，用它们使JVM的行为更适用于你的应用程序。

6.5.4 与 GC 相关的 JVM 参数

JVM的参数非常多（最少上百个），用来定制JVM运行时的行为。本节我们会讨论一些跟垃圾收集有关的选项，后续章节中还会讨论其他选项。

非标准的JVM选项

以-X:开头的选项不是标准选项，在其他JVM上可能不可用。

以-XX:开头的是扩展选项，不要随便使用。很多与性能相关的选项都是扩展选项。

有些选项相当于布尔型的参数，并且前面有+或-作为它的开关。还有带参数的选项，比如-XX:CompileThreshold=1000（这个方法会在调用次数达到1000之后才被JIT编译）。还有一些参数（包括很多标准参数）既没有开关也不能带参数。

表6-2中是基本的GC选项，还有这些选项的默认值（如果存在）。

表6-2 基本垃圾收集选项

选 项	效 果
-Xms<几MB>m	堆的初始大小 (默认2 MB)
-Xmx<几MB>m	堆的最大大小 (默认64 MB)
-Xmn<几MB>m	堆中年轻代的大小
-XX:-DisableExplicitGC	让调用System.gc()不产生任何作用

一个常用的小技巧是把-Xms和-Xmx的大小设成一样的。这样进程就会用恰当的堆尺寸运行，没必要在执行过程中调整大小（可能会引发意想不到的降速）。

表中最后一个选项输出GC的标准信息到日志中，我们在下一节会讨论如何解释这些信息。

6.5.5 读懂 GC 日志

为了充分利用垃圾收集，你需要经常看看子系统在做什么。除了基本的verbose:gc标记，还有很多可以控制输出信息的选项。

别拿GC日志不当回事儿，你可能时不时地就会发现自己被输出信息淹没了。下一节讨论VisualVM时你会发现，有一个可视化工具可以帮你看到VM的行为，这个工具非常有用。不管怎样，会读日志以及了解影响GC的基本选项非常重要，因为有时候你可能没法用GUI工具。最常用的GC日志选项如表6-3所示。

表6-3 用于扩展日志的额外选项

选 项	效 果
-XX:+PrintGCDetails	关于GC更详细的细节
-XX:+PrintGCDateStamps	GC操作的时间戳
-XX:+PrintGCApplicationConcurrentTime	在应用线程仍然运行的情况下用在GC上的时间

这些选项组合在一起时，会产生下面这种日志：

```
6.580: [GC [PSYoungGen: 486784K->7667K(499648K)]
1292752K->813636K(1400768K), 0.0244970 secs]
```

我们把它分解，看看每一部分是什么意思：

```
<time>: [GC [<collector name>: <occupancy at start>
=> <occupancy at end>(<total size>)] <full heap occupancy at start>
=> <full heap occupancy at end>(<total heap size>), <pause time> secs
```

第一块是GC的发生时间，从JVM启动开始算，到发生时的秒数。然后是用来收集年轻代的收集器名称（PSYoungGen）。接着是年轻代收集前后占用的内存，以及年轻代的总大小。接着是反映完全收集情况的相同部分。

除了GC日志选项，还有一个选项如果不经解释可能会引起误解。用选项-XX:+PrintGCApplicationStoppedTime产生的日志是这样的：

```

Application time: 0.9279047 seconds
Total time for which application threads were stopped: 0.0007529 seconds
Application time: 0.0085059 seconds
Total time for which application threads were stopped: 0.0002074 seconds
Application time: 0.0021318 seconds

```

这些并不一定指GC用了多长时间，而是指在一个从安全点开始的操作中，线程停了多长时间。这包括GC操作，但也包括其他安全点操作（比如Java 6中的偏向锁操作^①），所以没有十足把握说这是指GC时长。

所有这些信息对记录日志和事后分析都有用，但不容易做可视化处理。而很多开发人员在做初始分析时都喜欢使用GUI工具。好在HotSpot VM（标准的Oracle VM，稍后讨论）自带了一个非常实用的工具。

6.5.6 用 VisualVM 查看内存使用情况

VisualVM是Oracle JVM自带的可视化工具。它是插件架构，采用标准配置，比JConsole用起来更方便。

图6-6是标准的VisualVM汇总界面。启动VisualVM并把它连接到本地运行的程序上，就能看到这样的界面。（VisualVM也能连接到远程应用上，但有些功能通过网络不可用。）这个界面中VisualVM连接的是MacBook Pro上运行的Eclipse，你可以看到我们用来编写本书代码的Eclipse的设置。图6-6如下所示。

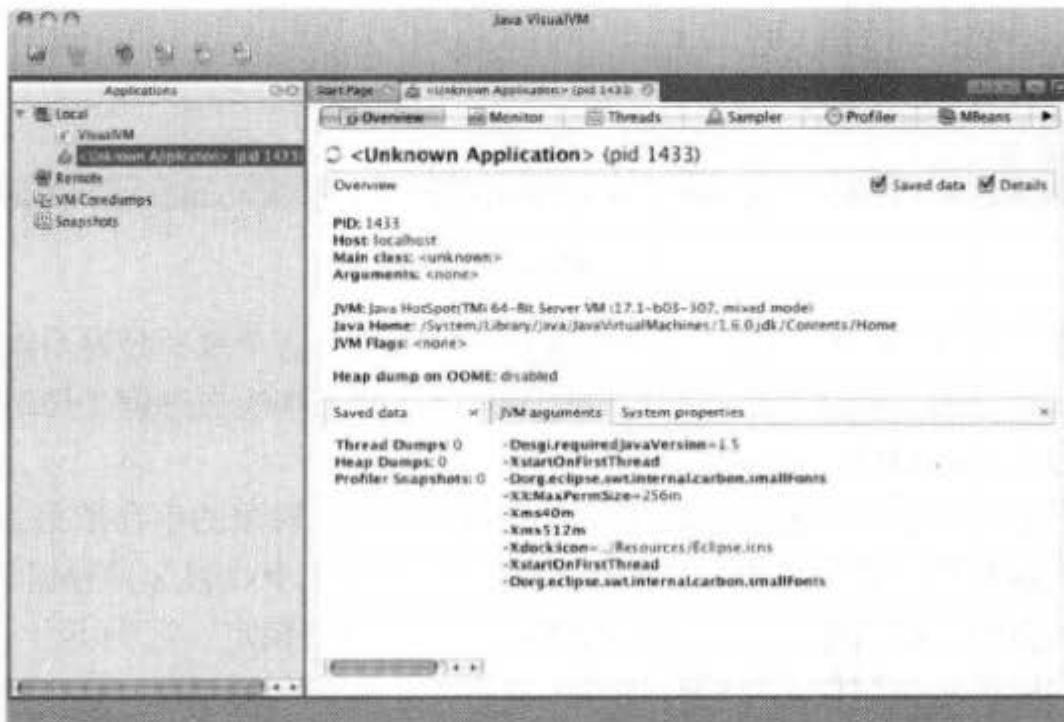


图6-6 VisualVM汇总界面

^① 在Java 6之前，加锁会导致一次原子CAS（Compare-And-Set）操作。对于没有争用的资源，该操作会造成无谓的开销。为解决这一问题，Java 6中引入了偏向锁技术，即偏向于第一个加锁的线程，该线程后续加锁操作不需要同步。其基本实现方式为：锁最初为NEUTRAL状态，当第一个线程加锁时，将该锁的状态修改为BIASED，并记录线程ID，这一线程在后续加锁时若发现状态是BIASED并且线程ID是当前线程ID，则只设置一下加锁标志，不需要进行CAS操作。其他线程若要加这个锁，需要使用CAS操作将状态替换为REVOKE，并等待加锁标志清零，以后该锁的状态就变成DEFAULT。这一功能可用-XX:-UseBiasedLocking命令禁止。——译者注

右侧面板顶部有很多标签。其中有扩展（Extension）、样例（Sampler）、JConsole、MBeans 和VisualVM插件。VisualVM插件为掌握Java运行时的动态情况提供了非常棒的工具。建议你在用 VisualVM 做任何实际工作前把这些插件都装上。

图6-7展示了内存占用的“锯齿”模式。这绝对是Java平台中内存占用情况的经典表现。它表示对象被分配在伊甸园中，使用，然后在年轻代中被回收。

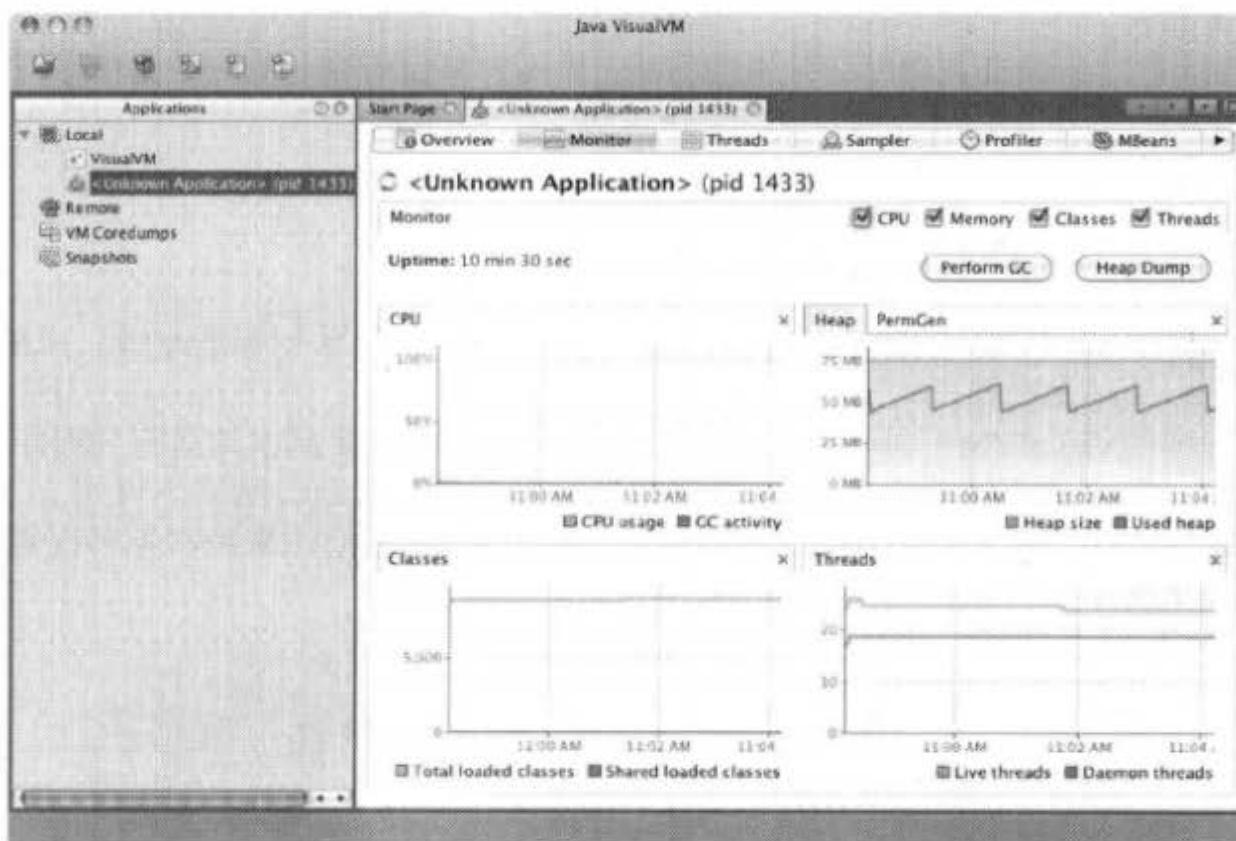


图6-7 VisualVM总览界面

每次年轻代收集之后，被占用的内存量回落到基线水平。这个水平是终身制对象和幸存者对象合起来的用量，可以用它来确定Java进程的健康状况。如果基线在进程工作时保持稳定或者逐渐递减，则表明内存的使用情况非常健康。

如果基线水平上升，也不一定就是出错了，可能只是有些对象的生存期很长，长到足够转入终身颐养园中。在这种情况下，最终会进行一次完全收集。完全收集会导致锯齿模式再次出现，从而使内存占用回落到基线水平。如果完全收集基线持续保持稳定，进程不会耗光内存。

锯齿上斜坡的陡度是进程使用年轻代内存（通常是伊甸园）的频率，这个概念很重要。降低年轻代收集的频率基本上就是降低锯齿的陡度。

内存使用情况的另外一种可视化方式如图6-8所示。你能看到伊甸园、幸存者乐园（S0和S1）、终身颐养园及PermGen区。在程序运行时，你能看到各个空间的大小变化。特别是在年轻代收集之后，可以看到伊甸园变小，幸存者乐园中两个空间的角色也互相转换了。

探索内存系统和运行时环境有助于你理解代码如何运行。相应地，这也表明VM提供的服务对性能影响很大，所以你绝对应该花时间研究一下VisualVM，尤其要结合Xmx和Xms这些选项试一下。

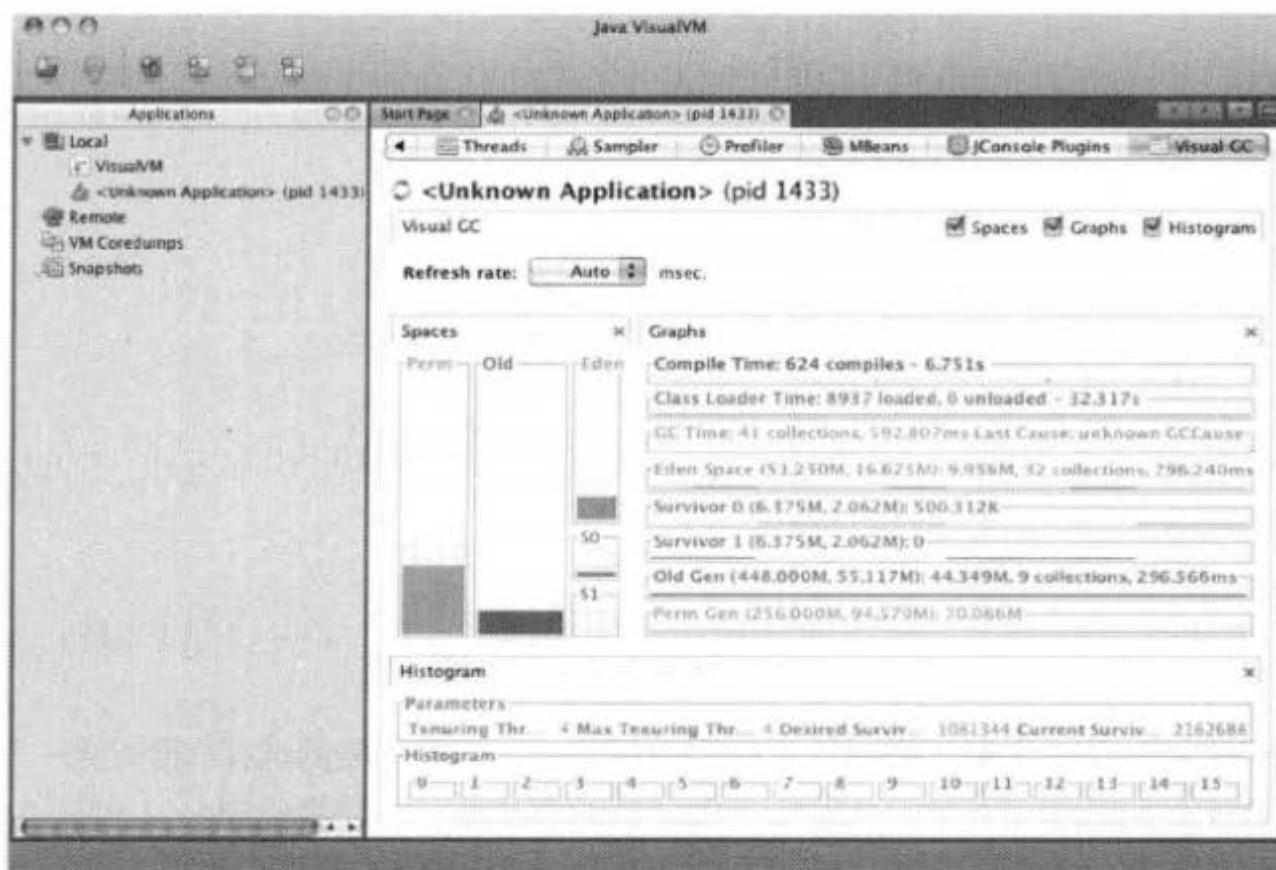


图6-8 VisualVM的可视化GC插件

下一节中，我们将要讨论JVM中的一项新技术，这项技术会在执行过程中自动降低堆内存的占用量。

6.5.7 逸出分析

本节介绍了JVM最近的一项修改，内容仅供参考。程序员不能直接控制或影响这项修改，并且在最近发布的Java中，这项优化是默认的。因此本节中没有太多关于这项修改的信息或例子。所以如果你想了解一下JVM提升自身性能的技巧，请继续。如果没兴趣，可以跳到6.5.8节去研究并发的垃圾收集。

逸出分析乍一看是个相当出人意料的想法。其基本思路是分析方法并确认其中哪个局部变量（的引用类型）只用在方法内部，以及哪些变量不会传入其他方法或从当前方法中返回。

这样JVM就可以在当前方法的栈框架内部创建这个对象，而不再使用堆内存。这会减少程序年轻代收集的次数，从而提高性能。请参见图6-9。

这就是说可以避免堆分配，因为在当前方法返回时，被局部变量占用的内存就自动释放了。用这种不牵扯堆分配的方式分配变量空间不会产生垃圾，当然就不需要收集垃圾。

逸出分析是减少JVM垃圾收集的新办法。它能对线程的年轻代收集次数产生显著影响。经实践证明，它通常能对总体性能产生百分之几的影响。虽然影响不是特别大，但也很有价值，特别是在进程的垃圾收集次数比较多的时候。

从Java 6u23往后，逸出分析是默认打开的，所以新版Java的速度免费提升了。

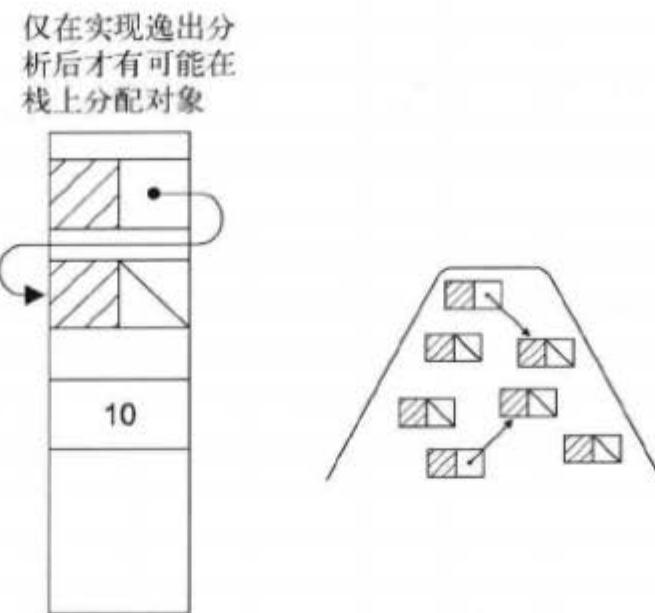


图6-9 逸出分析避免了对象的堆分配

现在我们去看另外一个对代码有巨大影响的环节——收集策略的选择。我们从一个经典的高性能选择（并发标记清除）开始，然后看一看最新的收集器——垃圾优先。

选择高性能收集器有很多原因。应用程序可能会从较短的GC暂停中受益，并且也愿意运行更多线程（占用CPU资源）来加快速度。或者你想控制GC暂停的频度。除了基本的收集器，你还可以用选项迫使平台采用不同的收集策略。在接下来的两节中，我们会介绍两个把这种可能性变成现实的收集器。

6.5.8 并发标记清除

并发标记清除（CMS）收集器是Java 5推荐的高性能收集器，在Java 6中仍然保持了旺盛的生命力。可以通过下面几个选项激活它，如表6-4所示。

表6-4 用于CMS收集器的选项

选 项	效 果
<code>-XX:+UseConcMarkSweepGC</code>	打开CMS收集
<code>-XX:+CMSIncrementalMode</code>	增量模式（一般都需要）
<code>-XX:+CMSIncrementalPacing</code>	配合增量模式，根据应用程序的行为自动调整每次执行的垃圾回收任务的幅度（一般都需要）
<code>-XX:+UseParNewGC</code>	并发收集年轻代
<code>-XX:ParallelGCThreads=<N></code>	GC使用的线程数

这些选项会覆盖垃圾收集的默认设置，为GC配置有N个并行线程的CMS垃圾收集器。这些线程会尽可能地在并发模式下完成GC工作。

这种并发方式是如何工作的呢？下面是与标记清除相关的三个重要事实：

- 某种世界停转（简称STW）的暂停是不可避免的；
- GC子系统绝对不能漏掉存活对象，这样做会导致JVM垮掉（或者更糟）；

□ 只有所有应用线程都为整体收集暂停下来，才能保证收集所有的垃圾。

CMS利用了最后一点。它制造两个非常短暂的STW暂停，并且在GC周期的剩余时间和应用程序的线程一起运行。这表明它愿意跟“伪阴性”妥协，由于竞争危害而无法标识某些垃圾（被漏掉的垃圾会在下一个GC周期中得到收集）。

CMS还要在运行时做复杂的记账工作，记录哪些是垃圾，哪些不是。这些额外的开销是为了在不停止应用线程的情况下运行GC所付出的代价。CMS在有更多CPU核心的机器上会表现得更好，并且会制造更频繁的短暂暂停。它的日志输出如下所示：

```
2010-11-17T15:47:45.692+0000: 90434.570: [GC 90434.570:  
[ParNew: 14777K->14777K(14784K), 0.0000595 secs] 90434.570:  
[CMS: 114688K->114688K(114688K), 0.9083496 secs] 129465K->117349K(129472K),  
[CMS Perm : 49636K->49634K(65536K)] icms_dc=100 , 0.9086004 secs]  
[Times: user=0.91 sys=0.00, real=0.91 secs]
```

这些日志和6.4.4节中基本的GC日志差不多，但增加了CMS和CMS Perm收集器部分。

最近几年，CMS作为最佳高性能收集器的地位受到了挑战，挑战者是垃圾优先（G1）收集器。我们来看看这颗冉冉升起的新星，了解一下它的新颖方法，以及它能够突破所有现存的Java收集器的原因。

6

6.5.9 新的收集器：G1

G1是Java平台中崭新的收集器。本来想把它和Java 7一起发布，但后来作为预发布版本跟Java 6一起发布了，到Java 7时就是成品了。它在Java 6中并没有得到广泛的应用，但随着Java 7逐渐普及，有望让G1成为高性能应用（也可能是所有应用）的默认选择。

G1的核心思想是暂停目标（pause goal），也就是程序在执行时能为GC暂停多长时间（比如每5分钟20ms）。G1会竭尽所能达成暂停目标。它和我们原来遇到的收集器完全不同，并且开发人员对GC如何执行有更多控制权。

G1不是真正的分代式垃圾收集器（尽管它仍然使用标记清除法）。相反，G1把堆分成大小相同的区域（比如每个1 MB），不区分年轻区和年老区。暂停时，对象被撤到其他区域（就像伊甸园对象被挪到幸存者乐园一样），清空的区域被放回到（空白区的）自由列表上。这种将堆划分为大小相同区域的做法如图6-10所示。

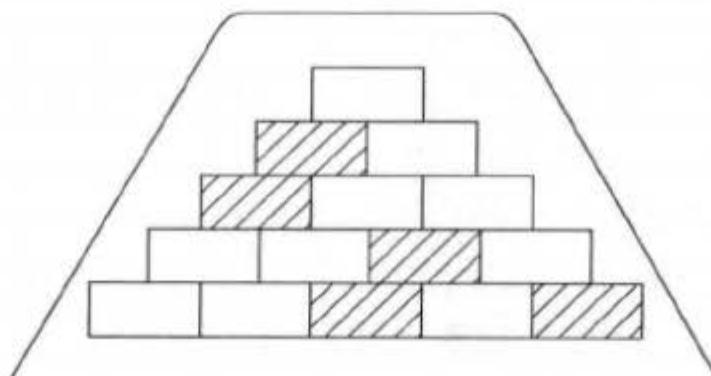


图6-10 G1如何划分堆空间

这个新的收集策略让Java平台可以统计收集单个区域需用的平均时长。这样你就可以在合理范围内指定一个暂停目标。G1只会在有限的时间内收集尽可能多的区域（尽管在收集最后一个区域时所用的时间可能比预期的长）。

要打开G1，需要用到表6-5中的选项。

表6-5 G1收集器的选项

选 项	效 果
-XX:+UseG1GC	打开G1收集
-XX:MaxGCPauseMillis=50	告诉G1它在一次收集中暂停的时间应该尽量保持在50ms以内
-XX:GCPauseIntervalMillis=200	告诉G1它将两次收集的时间间隔尽量保持在200ms以上

这些选项可以组合，比如设置最大暂停目标是50 ms，暂停间隔不能少于200 ms。当然，GC系统所能承受的压力是有限的。必须有充足的暂停时间把垃圾取出来。每隔100年1ms的暂停目标肯定是不现实的。

G1可以支持的负载和应用类型范围很广。如果你的应用程序已经到了需要对GC调优的地步，G1会是一个不错的选择。

在下一节中，我们会介绍JIT编译。对于很多或大多数程序来说，这是唯一一个可以为产生高性能代码做出最大贡献的因素。我们会学习JIT编译的基础知识，最后解释一下如何打开JIT编译的日志，让你能够判断正在编译哪个方法。

6.6 HotSpot 的 JIT 编译

正如我们在第1章所讲，Java是一种“动态编译”语言。也就是说在程序运行时，其中的类还会再进行一次编译，然后转换成机器码。

这个过程称为即时编译或JITing，并且通常是一次处理一个方法。要在庞大的代码库中找出其中的重要部分，理解这个过程是关键。

下面是一些与JIT编译有关的基本事实。

- 几乎所有现代JVM中都有某种JIT编译器。
- 相比较而言，纯粹解释型的VM要慢得多。
- 编译过的方法在运行速度上要比解释型的代码快很多，非常多。
- 先编译用得最多的方法，这是有道理的。
- 在做JIT编译时，先处理唾手可得的编译很重要。

按照最后一点，我们应该先研究编译过的代码，因为在正常情况下，所有仍然处于解释状态下的方法都没有已经编译过的方法运行频繁。偶尔会有无法编译的方法，但非常罕见。

方法一开始都是以字节码形态存在的，有调用时JVM只会对字节码进行解释并执行，同时记录方法被调用的次数及其他一些统计数据。当被调用次数达到某个阈值（默认10 000次）后，如

果它是合格的方法，就会有个JVM线程在后台把它的字节码编译成机器码。如果编译成功，以后所有对该方法的调用都会用它的编译结果，除非出现了某些导致检验失效的情况，或者出现了逆优化^①。

根据实际情况，方法编译后产生的机器码运行速度可能比解释模式下的字节码快100倍。改善性能通常都要先弄明白程序中哪些方法比较重要，以及哪些重要的方法被编译了。

为什么要动态编译？

有时人们会问，Java平台为什么要费心去做动态编译——为什么不提前编译好（像C++一样）。第一个答案通常都是：因为用平台无关的东西（.jar和.class文件）作为基本部署单位要比为每个目标平台做一份不同的编译好的二进制文件更轻松。

另外一种答案是动态编译会给编译器提供更多信息。具体地说，提前（AOT）编译的语言得不到运行时的任何信息——比如某个指令是否可用，其他的硬件细节以及代码运行情况的统计数据。这些变量让事情变得很有趣，使得Java这样的动态编译语言实际上可能会比提前编译的语言运行得更快。

在接下来对JITing机制的讨论中，我们所说的JVM特指HotSpot。后续讨论中很多通用内容也适用于其他VM，但在具体细节上可能会有很大出入。

我们会先介绍一下HotSpot提供的几个JIT编译器，然后解释HotSpot中最有力的两项优化技术（内联和独占派发）。在本节的结尾，我们会告诉你如何打开方法编译日志，以便你可以看到被编译的确切方法。下面有请HotSpot。

6.6.1 介绍 HotSpot

Oracle收购Sun时拿到了HotSpot VM（原来收购BEA时还拿到一个JRockit）。HotSpot是OpenJDK的基础。它有两种运行模式——客户端模式和服务器端模式。可以在启动JVM时指定-client或-server选项来选择不同的模式。（必须是命令行中的第一个选项。）每种模式都有各自适用的应用程序。

1. 客户端编译器

客户端编译器主要用于GUI应用程序。在这个领域中，操作的一致性至关重要，所以客户端编译器（有时叫C1）在编译时所做的决定往往更保守。也就是说它不能因为要取消一个经证实不正确或基于错误假设的优化决定而意外暂停。

2. 服务器端编译器

相反，服务器端编译器（C2）在编译时会大胆假设。为了确保代码正确运行，C2会快速地

^① JVM的动态优化技术可能会基于一些大胆（甚至不安全）的假设来编译字节码。比如假定要处理的数据都属于某一类，而在编译结果中只保留处理该类数据的程序分支。如果假设不成立，则JVM只能放弃编译结果，回去解释并执行原来的字节码，这一过程被称为逆优化。——译者注

做一次运行时检查（通常被称为警戒条件），以确保假设有效。如果假设无效，它会取消这次编译，并尝试别的编译。这种大胆假设的方式比保守的客户端编译器产生的编译结果性能好很多。

3. 实时Java

近年来出现了一种实时Java平台，有些开发人员好奇为什么那些需要表现出高性能的代码不直接用这个平台（它是独立的JVM，不是HotSpot选件）。那是因为实时系统不一定是最快的。

实时编程的关注点实际上是承诺能否兑现。从统计角度讲，实时系统是为了让执行操作的时间尽量保持一致，并且为了达成这个目的，它可能会牺牲一些平均等待时间。为了让运行状况保持一致，整体性能是可以受到轻微影响的。

图6-11中有两组代表等待时间的点阵。系列2（上面那组点阵）的平均等待时间在增长（因为它的等待时间刻度更高），但方差在减小，因为这些点比系列1中的点更靠近自己的平均值，系列1的点阵相较而言分布更加广泛。

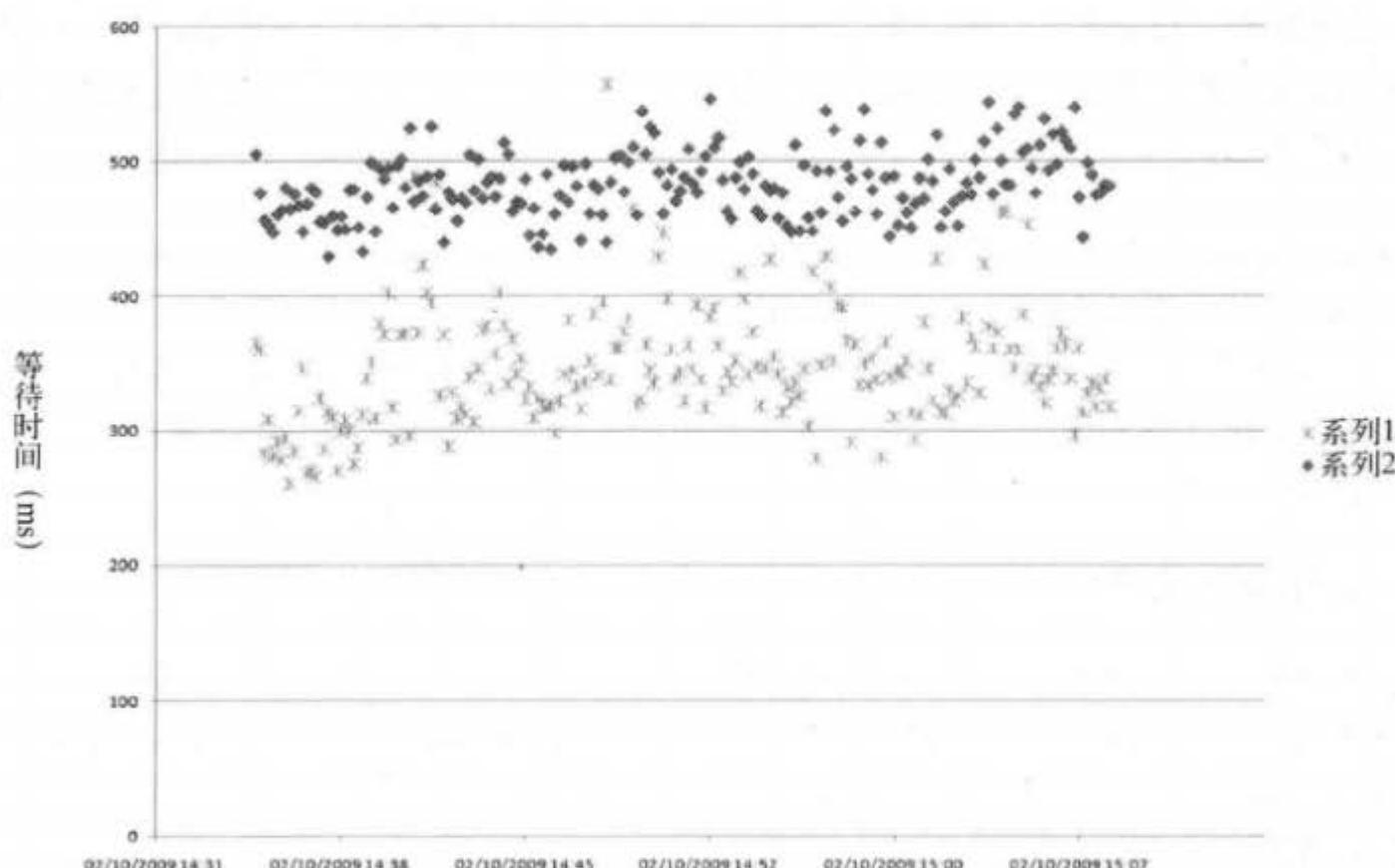


图6-11 方差和均值的变化

但希望实现高性能表现的团队想要的是更低的平均等待时间，即便以更高的方差为代价，所以他们通常会选择服务器端编译器的大胆优化策略（对应系列1）。

接下来我们会讨论所有运行时（服务器端、客户端和实时）广泛采用的技术，这项技术使它们表现得更好。

6.6.2 内联方法

内联是HotSpot的最大卖点之一。内联的方法不再是被调用，而是将调用方法的代码直接放

到调用者内部。

平台有这方面的优势，编译器可以根据运行时的统计数据（方法的调用频率）和其他因素（比如会不会因为调用者方法太多而对代码缓存产生影响）来决定如何处理内联。也就是说HotSpot编译器所做的内联决策比提前编译的编译器更智能。

方法的内联是全自动的，并且默认参数值几乎适用于任何情况。但也有选项可以用来控制内联方法大小，以及方法在成为内联候选之前的调用频率要达到多高。对于好奇的程序员来说，这些选项对于深入了解内联如何工作很有帮助。通常它们对于生产环境下的代码用处不大，并且应该作为性能调优的最后选择，因为它们对运行时系统的性能可能存在不可预测的影响。

访问器方法怎么处理？

有些开发人员错误地认为访问器方法（访问私有变量的公共方法）不能由HotSpot内联。他们认为变量是私有的，方法调用不能因为优化而去掉，不能在类外访问这个变量。这种想法不对。HotSpot把方法编译成机器码时能够并且会忽略访问控制，不用访问器方法直接访问私有域。这并不违背Java的安全模型，因为所有访问控制都在类加载和连接阶段检查过。

6

如果你还不信，可以做个练习，写一个跟代码清单6-2类似的测试类，对比一下预热过的访问器方法的速度和直接访问公共域的速度。

6.6.3 动态编译和独占调用

独占调用就是这种大胆优化的例子之一。它是基于大量观察做出的优化，像下面这种对象上的方法调用：

```
MyActualClassNotInterface obj = getInstance();
obj.callMyMethod();
```

只会在一种类型的对象上调用。换句话说，就是调用点`obj.callMyMethod()`几乎不会同时碰到一个类和它的子类。这时可以把Java方法查找替换为`callMyMethod()`编译结果的直接调用。

提示 独占派发提供了一个剖析JVM运行时的例子，允许Java平台进行C++这种AOT语言实现不了的优化。

出于非技术的原因，`getInstance()`方法有时不能返回`MyActualClassNotInterface`类型的对象，而其他情况下不能返回一些子类的对象，但实际上这种情况几乎从没发生过。但为了防止这种情况出现，会有一个运行时检查来确保对象的类型是由编译器按预期插入的。如果这个预期被违背，运行时会取消优化，程序甚至都不会注意也不会犯任何错误。

只有服务器端编译器才会做这种大胆的优化。实时和客户端编译器都不会这样做。

6.6.4 读懂编译日志

我们来看一个例子，了解一下如何使用JIT编译日志。依巴谷星表中详细列出了从地球上可以观测到的星星。我们的程序会处理这个目录，产生能在指定夜晚、指定地址看到的星图。

我们来看这个程序输出的一些日志，看看在星图应用运行时编译了哪些方法。我们用的关键选项是`-XX:+PrintCompilation`。我们前面简单讨论过这个扩展选项。把这个选项加到启动JVM的命令里是告诉JIT编译线程输出标准日志。这些日志表明方法超过编译阈值并被转成机器码的时间。

```

1      java.lang.String::hashCode (64 bytes)
2      java.math.BigInteger::mulAdd (81 bytes)
3      java.math.BigInteger::multiplyToLen (219 bytes)
4      java.math.BigInteger::addOne (77 bytes)
5      java.math.BigInteger::squareToLen (172 bytes)
6      java.math.BigInteger::primitiveLeftShift (79 bytes)
7      java.math.BigInteger::montReduce (99 bytes)
8      sun.security.provider.SHA::implCompress (491 bytes)
9      java.lang.String::charAt (33 bytes)
1% !    sun.nio.cs.SingleByteDecoder::decodeArrayLoop @ 129 (308 bytes)
...
39     sun.misc.FloatingDecimal::doubleValue (1289 bytes)
40     org.camelot.hipparcos.DelimitedLine::getNextString (5 bytes)
41 !    org.camelot.hipparcos.Star::parseStar (301 bytes)
...
2% !    org.camelot.CamelotStarter::populateStarStore @ 25 (106 bytes)
65 s    java.lang.StringBuffer::append (8 bytes)

```

这是非常典型的`PrintCompilation`输出。这些日志表明了“热”到可以编译的方法。跟你想的一样，第一个被编译的方法很可能是平台方法（比如`String#hashCode`）。再过一段时间，应用方法（比如`org.camelot.hipparcos.Star#parseStar`方法，在例子中用于分析天文目录里的记录）也会被编译。

这些输出中每行都有个数字，表明了这些方法在这次运行中的编译顺序。注意，由于平台的动态性质，这个顺序在每次运行时可能会稍有变化。这里还有一些其他域。

- s——表明该方法是同步的。
- !——表明方法有异常处理。
- %——当前栈替换（OSR）。这个方法被编译了，并且换掉了运行代码中的解释型版本。

注意，OSR方法有它们自己的计数方案，从1开始。

小心僵尸

当查看用服务器端编译器（C2）运行代码的样例日志时，你可能偶尔会看到“变得无法进入”和“变成僵尸”这样的字眼。这表明由于类加载操作（通常情况下），某个已经被编译过的特定方法现在无效了。

逆优化

如果经证实代码优化所基于的假设是不真实的，HotSpot可以对代码进行逆优化操作。在许多情况下，它会重新考虑，尝试不同的优化。因此同一个方法可能会被逆优化和重编译几次。

过了一段时间，你会看到被编译的方法数量趋于稳定。编译好的代码达到了一个稳定的状态，并且大多数代码会保持不变。哪些方法被编译取决于所用的JVM版本和OS平台。并不是所有平台都会产生相同的编译方法集合，并且给定方法编译代码的大小也不会完全一样。就像性能调优里很多其他东西一样，这也应该进行测量，并且结果可能会让人大吃一惊。即便看上去相当简单的Java方法，在Solaris和Linux上经JIT编译生成的机器码也会有五分之一的差异。测量是必不可少的。

6.7 小结

性能调优不是盯着你的代码期待奇迹，或者给代码喝一罐快速修复药水。相反，性能调优需要细致测量，关注细节，还需要你的耐心。你要不断减少测试中出现的错误源，直到引发性能问题的真正凶手出现。

我们先来看看在JVM动态环境中进行性能调优的要点。

- JVM是极为强大的复杂运行时环境。
- JVM的性质使得有时候优化其中的代码很有挑战性。
- 你必须通过测量准确地找到问题的真正所在。
- 要特别注意垃圾收集子系统和JIT编译器。
- 监测还有其他一些工具对你真的很有帮助。
- 学会阅读日志和平台的其他指标——有时不能使用工具。
- 你必须测量并设置目标（这个太重要了，所以我们要一再提起）。

现在你应该具备探索和实验Java平台的高级性能特性所需的基础知识了，并且能够理解性能机制如何影响你的代码。希望你能开放心态，以足够的信心和经验去分析这些数据，并能把这种见解应用于你自己的性能问题。

我们会在下一章看到JVM上除Java语言之外的其他语言，平台的很多性能特性适用范围非常广泛——特别是JIT编译器和GC的相关知识。

Part 3

第三部分

JVM 上的多语言编程

这一部分专门探索 JVM 上的新语言范式和多语言编程。

JVM 是一个迷人的运行时环境：它提供的不仅是性能和能力，还赋予了程序员惊人的灵活性。实际上，JVM 是探索 Java 之外的语言的关口，并且会让你尝试一些不同的编程方式。

如果你只用 Java 写程序，可能想知道学习其他语言会有什么好处。就像我们在第 1 章说的，成为优秀 Java 开发人员的本质就是对 Java 语言、平台和生态系统的方方面面掌握得越来越全面。这包括能够欣赏那些目前刚刚起步，但不久的将来就会变得不可或缺的主题。

未来已经发生，只是分布尚不均匀。

——威廉·吉布森

事实证明，很多未来需要的新想法已经出现在函数式编程等其他 JVM 语言中了。学习新 JVM 语言的过程中，我们可以一瞥另一个世界，我们未来的某些项目很可能就跟它很像。从不同的视角探索问题能帮我们重新审视已有的知识。学习新语言可以开启新的可能性，我们可能会发现自己不知道的新天赋，掌握新技能，而这些东西总有一天会派上用场。

第 7 章会解释一下为什么 Java 不是解决所有问题的理想语言、为什么函数式编程概念有用，以及如何为特定项目选择一种非 Java 语言。

最近，很多书和博客里都提出一种观点，认为函数式编程很快就会成为每个开发人员职业生涯中的重要角色。很多文章都把函数式编程描述得令人生畏，却常常讲不清楚函数式编程怎么在 Java 这样的语言中“发光发热”。

实际上，函数式编程根本算不上一个整体结构。相反，它更像一种风格，开发人员思考方式上的一个过渡。第 8 章会给出一个用 Groovy 语言编写的、稍微带点儿函数式编程味道的例子，就是用一种更清晰的、不太容易出 bug 的风格来处理集合的代码。在第 9 章，我们会用 Scala 语言讨论对象—函数式风格。第 10 章会用 Clojure 语言看一下纯粹的函数式编程（它甚至超过了面向对象）方式。

在第四部分，我们会介绍几个真实案例，针对这些案例，其他语言能够给出更好的解决方案。如果你不信，可以提前看一下第四部分，然后再回来学习应用那些技术所需的语言。

备选JVM语言

本章内容

- 为什么应该使用备选JVM语言
- 语言的类型
- 备选语言的选择标准
- JVM如何处理备选语言

如果你用Java做过大项目，可能已经注意到了，Java有时稍显繁琐和笨拙。你甚至可能希望它不是这样的——总之要再容易点儿。

好在JVM很棒！实际上，它太棒了，Java以外的其他语言也可以很自然地把它当成栖息地。我们在本章里会告诉你为什么要把其他JVM编程语言加入到我们的项目中，以及如何做到这一点。

我们会讨论描述不同语言类型（比如静态与动态）的方式、为什么用备选语言，以及选择它们时有哪些标准。我们还会介绍三种语言：Groovy、Scala和Clojure，并在第三部分和第四部分中更深入地探讨它们。

然而在开始之前，你需要对Java的缺点有更清楚的认识。下一节有一个扩展示例，它突出了Java语言中一些恼人的地方，指出了它未来的发展方向为函数式编程风格。

7.1 Java 太笨？纯粹诽谤

假设你要在一个交易（事务）处理系统中编写一个新组件。这个系统的简化视图如图7-1所示。

在图中可以看到，系统有两个数据源：上游的收单系统（可以通过Web服务查询）和下游的派发数据库。

这是一个很现实的系统，是Java开发人员经常构建的系统。我们在这一节里准备引入一小段代码把两个数据源整合起来。你会看到Java解决这个问题有点笨拙。之后我们会介绍函数式编程的一个核心概念，并展示一下怎么用映射（map）和过滤器（filter）等函数式特性简化很多常见的编程任务。你会看到Java由于缺乏对这些特性的直接支持，编程会困难不少。

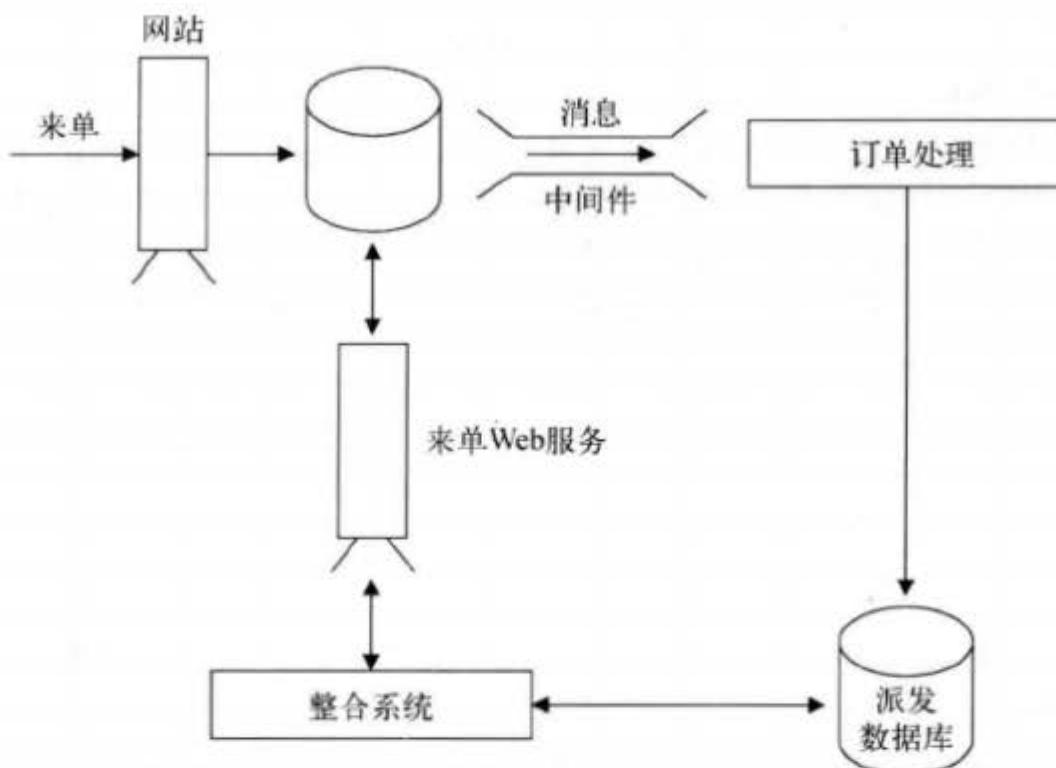


图7-1 交易处理系统的例子

7.1.1 整合系统

7

我们需要一个整合系统来检查数据确实到了数据库。这个系统的核心是reconcile()方法，它有两个参数：sourceData（来自于Web服务的数据，归结到一个Map中）和dbIds。

你需要从sourceData中取出main_ref键值，用它跟数据库记录的主键比较。代码清单7-1是进行比较的代码。

代码清单7-1 整合两个数据源

```

public void reconcile(List<Map<String, String>> sourceData,
Set<String> dbIds) {
    Set<String> seen = new HashSet<String>();
    MAIN: for (Map<String, String> row : sourceData) {
        String pTradeRef = row.get("main_ref");
        if (dbIds.contains(pTradeRef)) {
            System.out.println(pTradeRef + " OK");
            seen.add(pTradeRef);
        } else {
            System.out.println("main_ref: " + pTradeRef + " not present in DB");
        }
    }
    for (String tid : dbIds) {
        if (!seen.contains(tid)) {
            System.out.println("main_ref: " + tid + " seen in DB but not Source");
        }
    }
}

```

假定pTradeRef永远
不会为null

特殊情况

这里主要是检查收单系统中的所有订单是否都出现在派发数据库里。这项检查由打上了MAIN标签的for循环来做。

还有另外一种可能。比如有个实习生通过管理界面做了些测试订单（他没意识到这些订单用的是生产系统）。这样订单数据会出现在派发数据库里，但不会出现在收单系统中。

为了处理这种特殊情况，还需要一个循环。这个循环要检查所见到的集合（同时出现在两个系统中的交易）是否包含了数据库中的全部记录。它还会确认那些遗漏项。下面是这个样例的一部分输出：

```
7172329 OK
1R6GV OK
1R6GW OK
main_ref: 1R6H2 not present in DB
main_ref: 1R6H3 not present in DB
1R6H6 OK
```

哪儿出错了？原来是上游系统不区分大小写而下游系统区分，在派发数据库里表示为1R6H12的记录实际上是1r6h2。

如果你检查一下代码清单7-1，就会发现问题出在contains()方法上。contains()方法会检查其参数是否出现在目标集合中，只有完全匹配时才会返回true。

也就是说其实你应该用containsCaseInsensitive()方法，可这是一个根本就不存在的方法！所以你必须把下面这段代码

```
if (dbIds.contains(pTradeRef)) {
    System.out.println(pTradeRef + " OK");
    seen.add(pTradeRef);
} else {
    System.out.println("main_ref: " + pTradeRef + " not present in DB");
}
```

换成这样的循环：

```
for (String id : dbIds) {
    if (id.equalsIgnoreCase(pTradeRef)) {
        System.out.println(pTradeRef + " OK");
        seen.add(pTradeRef);
        continue MAIN;
    }
}
System.out.println("main_ref: " + pTradeRef + " not present in DB");
```

这看起来比较笨重。只能在集合上执行循环操作，不能把它当成一个整体来处理。代码既不简洁，又似乎很脆弱。

随着应用程序逐渐变大，简洁会变得越来越重要——为了节约脑力，你需要简洁的代码。

7.1.2 函数式编程的基本原理

希望上面的例子中的两个观点引起了你的注意。

- 将集合作为一个整体处理要比循环遍历集合中的内容更简洁，通常也会更好。
- 如果能在对象的现有方法上加一点点逻辑来调整它的行为是不是很棒呢？

如果你遇到过那种基本就是你需要，但又稍微差点儿意思的集合处理方法，你就明白不得不再写一个方法是多么沮丧了，而函数式编程（FP）恰好搔到了这个痒处。

换种说法，简洁（并且安全）的面向对象代码的主要限制就是，不能在现有方法上添加额外的逻辑。这将我们引向了FP的大思路：假定确实有办法向方法中添加自己的代码来调整它的功能。

这意味着什么？要在已经固定的代码中添加自己的处理逻辑，就需要把代码块作为参数传到方法中。下面这种代码才是我们真正想要的（为了突出，我们把这个特殊的contains()方法加粗了）：

```
if (dbIds.contains(pTradeRef, matchFunction)) {
    System.out.println(pTradeRef + " OK");
    seen.add(pTradeRef);
} else {
    System.out.println("main_ref: " + pTradeRef + " not present in DB");
}
```

如果能这样写，contains()方法就能做任何检查，比如匹配区分大小写。这需要能把匹配函数表示成值，即能把一段代码写成“函数字面值”并赋值给一个变量。

函数式编程要把逻辑（一般是方法）表示成值。这是FP的核心思想，我们还会再次讨论，先看一个带点儿FP思想的Java例子。

7.1.3 映射与过滤器

7

我们把例子稍微展开一些，并放在调用reconcile()的上下文中：

```
reconcile(sourceData, new HashSet<String>(extractPrimaryKeys(dbInfos)));
private List<String> extractPrimaryKeys(List<DBInfo> dbInfos) {
    List<String> out = new ArrayList<>();
    for (DBInfo tinfo : dbInfos) {
        out.add(tinfo.primary_key);
    }
    return out;
}
```

extractPrimaryKeys()方法返回从数据库对象中取出的主键值（字符串）列表。FP粉管这叫map()表达式：extractPrimaryKeys()方法按顺序处理List中的每个元素，然后再返回一个List。上面的代码构建并返回了一个新列表。

注意，返回的List中元素的类型（String）可能跟输入的List中元素的类型（DBInfo）不同，并且原始列表不会受到任何影响。

这就是“函数式编程”名称的由来，函数的行为跟数学函数一样。函数 $f(x)=x*x$ 不会改变输入值2，只会返回一个不同的值4。

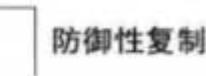
便宜的优化技巧

调用reconcile()时，有个实用但小有难度的技巧：把extractPrimaryKeys()返回的List传入HashSet构造方法中，变成Set。这样可以去掉List中的重复元素，reconcile()方法调用的contains()可以少做一些工作。

`map()`是经典的FP惯用语。它经常和另一个知名模式成对出现：`filter()`形态，请看代码清单7-2。

代码清单7-2 过滤器形态

```
List<Map<String, String>> filterCancels(List<Map<String, String>> in) {
    List<Map<String, String>> out = new ArrayList<>();
    for (Map<String, String> msg : in) {
        if (!msg.get("status").equalsIgnoreCase("CANCELLED")) {
            out.add(msg);
        }
    }
    return out;
}
```



注意其中的防御性复制，它的意思是我们返回了一个新的List实例。这段代码没有修改原有的List (`filter()`的行为跟数学函数一样)。它用一个函数测试每个元素，根据函数返回的boolean值构建新的List。如果测试结果为true，就把这个元素添加到输出List中。

为了使用过滤器，还需要一个函数来判断是否应该把某个元素包括在内。你可以把它想象成一个向每个元素提问问题的函数：“我应该允许你通过过滤器吗？”

这种函数叫做谓词函数 (predicate function)。这里有一个用伪代码 (几乎就是Scala) 编写的方法：

```
(msg) -> { !msg.get("status").equalsIgnoreCase("CANCELLED") }
```

这个函数接受一个参数 (`msg`) 并返回boolean值。如果`msg`被取消了，它会返回`false`，否则返回`true`。用在过滤器中时，它会过滤掉所有被取消的消息。

这就是你想要的。在调用整合代码之前，你需要移除所有被取消的订单，因为被取消的订单不会出现在派发数据库中。

事实上，Java 8准备采用这种写法 (受到了Scala和C#语法的强烈影响)。我们在第14章还会讨论这个主题，但在那之前我们会遇到几次函数字面值 (也称为lambda表达式)。

我们接着往下看，讨论一下其他情况，从JVM上可用的语言类型开始 (有时候我们也把这称为语言生态学)。

7.2 语言生态学

编程语言有多种不同的流派和类型。也就是说，不同的语言有不同的编码风格和编码方式。如果想掌握这些风格并让它们为你所用，你得弄明白这些差异以及如何给语言分类。

注意 这些分类可以帮助思考语言的多样性。尽管某些分法可能更清晰，但没有哪种是完美的。

最近几年，语言在添加新特性时有跨越各种分类的趋势。这就是说，你最好认为某种语言比其他语言“函数化程度更低”，或者“虽然是动态类型，但必要时也有可选的静态类型”。

我们将要讨论的分类是“解释型与编译型”、“动态与静态”、“命令式与函数式”，还有在JVM上重新实现的语言与原生语言。通常这些分类用来明确语言的边界，不要用学院化方式把它们当做完整精确的分类。

Java是运行时编译、静态类型的命令式语言。它强调安全性、代码清晰、性能，并乐于表现出一定程度的繁琐和死板（比如在部署中）。不同的语言可能侧重不同，比如动态类型的语言可能更看重部署速度。

我们先从解释型与编译型分类开始介绍。

7.2.1 解释型与编译型语言

解释型语言是那种源码是什么就执行什么的语言，不会在执行开始之前把整个程序转成机器码。编译型语言则不同，一开始就要用编译器把人类可读的源码变成二进制形式。

这种分别最近变模糊了。80年代和90年代早期这两类语言的边界还相当清晰：C/C++及类似的语言是编译型，Perl和Python是解释型。但Java同时兼具编译型和解释型两种特性，这一点我们已经在第1章讲过了。字节码的出现使这个界限更模糊了。人类肯定读不了字节码，但它也不是真正的机器码。

对于本书中要研究的JVM语言，我们划分的边界是该语言是否会将源码编译为类文件并且执行。不产生类文件的语言会由解释器（可能是用Java写的）逐行执行源码。有些语言既有编译器也有解释器，还有些既有解释器又有产生JVM字节码的即时编译器（JIT）。

7.2.2 动态与静态类型

在动态类型语言中，变量在不同时间可能会有不同的类型。我们以一小段简单的JavaScript代码为例，JavaScript是著名的动态语言。即便你不了解这种语言，也应该很容易理解下面的代码：

```
var answer = 40;
answer = answer + 2;
answer = "What is the answer? " + answer;
```

在这段代码中，变量answer一开始被赋值为40，当然，是个数值。然后给它加上2，变成了42。之后我们给answer赋了个字符串值。这在动态语言中是非常普遍的技术，不会引起语法错误。

JavaScript解释器也能分清两种+操作符的用法。第一个+是数字相加——把2加到40上，而在下一行中，解释器能从上下文中推导出开发人员要做字符串合并。

注意 这里的关键是动态类型语言跟踪变量值的类型（比如数字或字符串）信息，而静态类型语言跟踪变量的类型信息。

静态类型非常适合编译型语言，因为所有类型信息都在变量上，跟变量的值没有关系。这样很容易在编译时推导潜在的类型系统违规行为。

动态类型语言把类型信息放在变量所持有的值上。也就是说很难提前发现类型违规行为，因为推导所需的信息直到运行时才能得到。

7.2.3 命令式与函数式语言

Java 7是典型的命令式语言。命令式语言把程序的运行状态建模为可修改的数据，用一系列指令来改变运行状态。因此，在命令式语言中，程序状态才是核心概念。

命令式语言主要分为两类。一种是过程语言，比如BASIC和FORTRAN。这种语言把代码和数据完全分开，有简单的代码操作数据范式。另外一种是面向对象（OO）语言，数据和代码（以方法的形式）共同封装在对象中。面向对象语言中或多或少地存在元数据（比如类信息）引入的额外结构。

函数式语言不同，它把计算本身当做最重要的概念。函数式语言跟过程语言一样对值进行操作，但它不会修改输入，而是像数学函数一样返回新值。

如图7-2所示，函数被看做“小处理机”，输入值并输出新值。它们没有任何自己的状态，并且把它们和任何外部状态绑在一起也没有任何意义。这就是说一切皆对象的世界观跟函数式语言的自然观点有些分歧。

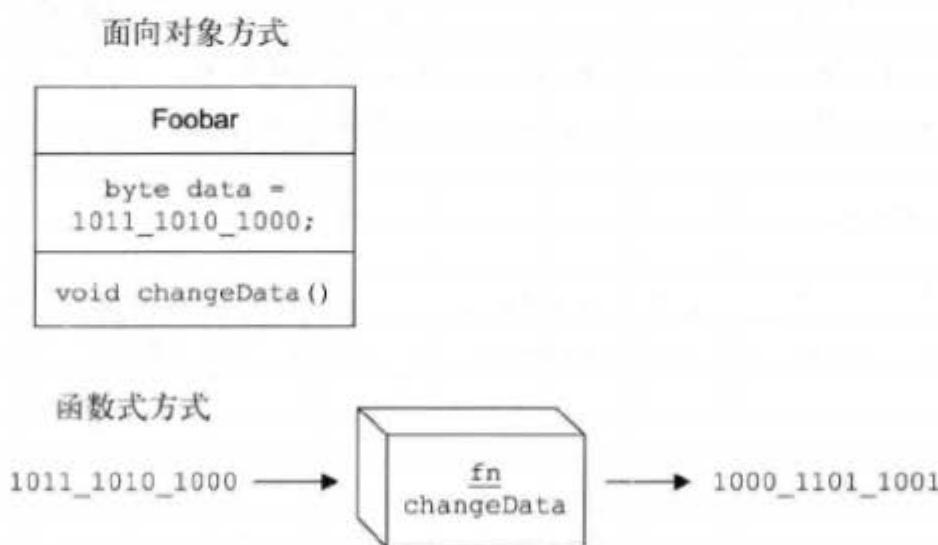


图7-2 命令式和函数式语言

在接下来的三章里，每章重点介绍一种语言，并且都会以前面对函数式编程的讨论为基础。我们会从Groovy开始，它带“一点儿函数式风格”，用我们在7.1节讨论过的方式处理集合；然后是Scala，对FP的利用更加充分；最后是Clojure（纯粹的函数式语言，完全没有面向对象特性）。

7.2.4 重新实现的语言与原生语言

JVM语言之间的另一个重要区别是重新实现已有语言与专门以JVM为目标的划分。通常来说，那些专门以JVM为目标写的语言能把自己的类型系统跟JVM的原生类型结合得更紧密。

下面是三种重新实现已有语言的JVM语言。

- JRuby在JVM上重新实现了Ruby语言。Ruby是一个动态类型的面向对象语言，有些函数式特性。它在JVM上基本算解释型的，但最近发布的版本中有一个运行时JIT编译器，在适当条件下可以生成JVM字节码。
- Jython由Jim Hugunin在1997年发起，当时是为了在Python中使用高性能的Java类库。它在JVM上重新实现了Python，因此是动态的，总体还算面向对象语言。它的运作方式是先在内部生成Python字节码，然后再转换成JVM字节码。这使它能在Python典型的解释模式（看起来像）下工作。通过生成JVM字节码，并把结果类文件保存到硬盘上，它也能在预先（AOT）编译模式下工作。
- Rhino最初是由Netscape开发的，后来转给Mozilla项目。它在JVM上提供了一个JavaScript实现。JavaScript是动态类型的面向对象语言（但和Java实现面向对象的方式截然不同）。Rhino既支持编译模式也支持解释模式，随Java 7一起发布（具体细节请参见com.sun.script.javascript包）。

最早的JVM语言

很难确定最早的JVM语言（除Java之外）是什么。可以肯定的是在1997年前后就出现了Kawa语言，它是一种Lisp语言。在那之后这些语言几乎呈现了爆炸式增长，因此追踪它们的历史太难了。

7

在编写本书时，猜测至少有200种JVM语言应该是合理的。不能说所有语言都很活跃或得到了广泛应用，但这个数字起码能表明JVM是一个非常活跃的语言开发和实现平台。

注意 在随Java 7推出的语言和VM规范里，所有对Java语言的直接引用都从VM规范中去掉了。Java现在只是运行在JVM上的众多语言中的普通一员，它不再享有特权了。

就像我们在第5章中讨论的，能让这么多不同的语言运行在JVM上的关键技术是类文件格式。任何能产生类文件的语言都可以认为是JVM上的编译型语言。

我们接下来会讨论多语言编程怎么变成了让Java程序员感兴趣的领域。我们会解释基本概念，为什么要给我们的项目选择一种备选的JVM语言以及如何操作。

7.3 JVM上的多语言编程

“JVM上的多语言编程”这种说法还挺新颖的。这种说法是为了描述那些以Java代码为核心，但还用了一种或多种其他非Java JVM语言的项目。多语言编程通常是一种关注点分离的形式。如图7-3所示，非Java技术的作用可以分为三个层次。这张图有时被称为多语言编程金字塔，这要归功于Ola Bini。

金字塔中有三个明确的层次：特定领域层、动态层和稳定层。



图7-3 多语言编程金字塔

多语言编程的秘密

多语言编程之所以有意义，是因为不同的代码片段有不同的生存期。银行里的风险引擎可能会持续运行五年以上；而网站上的JSP页面可能只有几个月；最短命的启动代码可能只“活”几天。代码“活”得时间越长，越靠近金字塔的底部。

它代表了不同侧重点的相互折中，比如底部更关注性能和全面测试，而顶部侧重的是灵活性和快速部署能力。

表7-1给出了这三个层次的更多细节。

表7-1 三层多语言编程金字塔

名 称	描 述	例 子
特定领域层	特定领域语言。与应用程序领域的特定部分结合非常紧密	Apache Camel DSL、Drools、Web模板
动态层	开发速度快、生产率高、功能灵活部署	Groovy、Jython、Clojure
稳定层	核心功能、稳定、经过良好测试、性能高	Java、Scala

这些层次中有特定的模式，静态类型语言更倾向于稳定层的任务。相反，能力不是那么强、通用性比较低的技术在金字塔的顶部更容易找到自己的位置。

金字塔中部给动态语言留下了很多位置。这也是最灵活的一层，大多数情况下在动态层内部或者在动态层和相邻层之间有重叠。

我们要对这张图继续深挖，看看Java语言为什么不是金字塔所有层次的最佳选择。接下来我们先来看看为什么要考虑非Java语言，然后给出一些选择非Java语言的重要标准。

7.3.1 为什么要用非Java语言

Java作为一种通用、静态类型的编译型语言有很多优势。这些品质使它成为实现稳定层功能的绝佳选择。但同样的特性放到金字塔上层就会变成负担。比如说：

- 重新编译太费工了；
- 静态类型不够灵活，重构起来时间可能会比较长；

- 部署的动静太大；
- Java的语法天然不适用于生产DSL。

Java项目重新编译的时长迅速攀升到了90秒到2分钟。这个长度足以严重打断开发人员的思路，并且对于只在生产环境中存活几个星期的代码来说，这种开发方式太糟糕了。

比较务实的办法是利用Java丰富的API和类库完成稳定层的繁重工作。

注意 如果你刚开始一个新项目，可能会发现其他稳定层语言（比如Scala）也具备非常重要的特性（比如卓越的并发支持）。然而在大多数情况下，不应该用其他种类的稳定语言重写正在使用的稳定层代码。

这时你可能会纳闷：“每一层都会面临什么样的编程挑战，我该选哪种语言？”一个优秀的Java开发人员知道，根本没有所谓的银弹，但当我们面对选择时，的确有一定的评估标准。我们不可能在这里把每个可能的选项都讨论到，所以在剩下的章节中，我们会集中讨论三种大多数Java开发人员可能都会面临的选择。

7.3.2 崭露头角的语言新星

接下来我们会挑三种，在我们看来可能最有生命力和影响力的语言。这些JVM语言（Groovy、Scala和Clojure）在多语言程序员心目中已经有了相当的分量。这三种语言为什么会得到大家的青睐？且听我们一一道来。

1. Groovy

Groovy语言是James Strachan在2003年发明的。它是动态的编译语言，语法跟Java很像，但更灵活。它被广泛用做脚本语言和快速原型语言，并且经常是开发人员或团队首选的非Java语言调研对象。你可以把Groovy看做是动态层的语言，它以擅长构建DSL著称。第8章主要介绍Groovy。

2. Scala

Scala是一门面向对象的语言，但也支持函数式编程。它的起源可以追溯到2003年，当时Martin Odersky正在用Java做一个与泛型相关的项目，结果却催生了Scala。它和Java一样，是静态类型的编译语言，但和Java不同，它做了大量的类型推断工作。也就是说它经常给人以动态语言的感觉。

Scala从Java中借鉴了很多东西，并且它的设计“修正”了Java中几个长期以来困扰开发人员的问题。Scala可以做稳定层语言，并且有些开发人员认为它可能会取代Java成为“JVM上的下一个大语言”。第9章介绍Scala。

3. Clojure

Clojure是由Rich Hickey设计的，属于Lisp家族的语言。它从Lisp中继承了很多语法特性（包括大量的括号^①）。就像大多数Lisp语言一样，它是动态类型的函数式语言。它是编译型语言，但

^① Lisp的表达式是一个原子（atom）或表（list）：原子（atom）又包含符号（symbol）与数值（number）；表是由零个或多个表达式组成的序列，表达式之间用空格分隔，放入一对括号中。此处的括号应该是指内置的表达式。

——译者注

通常以源码形态发布（稍后解释）。Clojure还向它的Lisp核心中添加了相当可观的新特性（特别是在并发方面）。

Lisp通常被当做专家语言。Clojure在某种程度上来说要比其他Lisp语言容易掌握，然而这并不会影响其强大的力量（也非常适合测试驱动的开发风格）。但它可能还是徘徊在主流之外，只是狂热的爱好者手中的秘密武器，抑或遇到适合它的特殊工作才发光（比如有些金融应用程序发现它的功能组合非常有吸引力）。

Clojure通常被认为是动态层的语言，但由于它的并发支持以及其他一些特性，也能胜任很多稳定层语言的工作。第10章会重点介绍Clojure。

现在我们已经把可选择的一部分语言罗列出来了，接下来该讨论一下决定你做出选择的那些因素了。

7.4 如何挑选称心的非Java语言

一旦决定在项目中实验非Java语言，就要先把项目中的各个工作域分清楚：哪些属于稳定层、哪些属于动态层或特定领域层。表7-2中给出了分属各层的工作。

表7-2 适合稳定层、动态层或特定领域层的项目域

名 称	说 明
特定领域层	构建、持续集成、持续部署
	开发操作
	企业集成模式建模
	业务规则建模
动态层	快速Web开发
	原型
	交互式管理与用户控制台
	脚本
	测试（比如用于测试驱动或行为驱动的开发）
稳定层	并发代码
	应用容器
	核心业务功能

如你所见，这些备选语言的使用范围非常广泛。但确定用备选语言解决哪项工作只是开始，接下来还要评估用备选语言是否合适。下面是帮我们选择技术的一些标准。

- 是否为项目里的低风险区。
- 备选语言跟Java的交互操作是否容易。
- 备选语言是否有工具支持（如IDE支持）。
- 语言学习难度。
- 招聘这门语言的开发人员的难度。