第二章 构建线程安全应用程序

第二章 构建线程安全应用程序 1
2.1. 什么是线程安全性 2
2.2. Servlet的线程安全性 5
2.3. 同步与互斥 9
2.3.1 线程干扰 9
2.3.2 同步 11
2.4. 同步与volatile 13
2.5. 活性 14
2.6. ThreadLocal变量 15
2.7. 高级并发对象 19
参考文献 20

2.1. 什么是线程安全性

当对一个复杂对象进行某种操作时，从操作开始到操作结束，被操作的对象往往会经历若干非法的中间状态。这跟外科医生做手术有点像，尽管手术的目的是改善患者的健康，但医生把手术过程分成了几个步骤，每个步骤如果不是完全结束的话，都会严重损害患者的健康。想想看，如果一个医生切开患者的胸腔后要休三周假会怎么样？与此类似，调用一个函数（假设该函数是正确的）操作某对象常常会使该对象暂时陷入**不可用的状态**（通常称为不稳定状态），等到操作完全结束，该对象才会重新回到完全可用的状态。**如果其他线程企图访问一个处于不可用状态的对象，该对象将不能正确响应从而产生无法预料的结果，如何避免这种情况发生是线程安全性的核心问题**。单线程的程序中是不存在这种问题的，因为在一个线程更新某对象的时候不会有其他线程也去操作同一个对象。(除非其中有异常，异常是可能导致上述问题的。当一个正在更新某对象的线程因异常而中断更新过程后，再去访问没有完全更新的对象，会出现同样的问题）

给线程安全下定义是比较困难的。很多正式的定义都比较复杂。如，有这样的定义：“**一个类在可以被多个线程安全调用时就是线程安全的**”。但是它不能帮助我们区分一个线程安全的类与一个线程不安全的类。

实际上，所有线程安全的定义都有某种程序的循环，因为它必须符合类的规格说明 —— 这是对类的功能、其副作用、哪些状态是有效和无效的、不可变量、前置条件、后置条件等等的一种非正式的松散描述(由规格说明给出的对象状态约束只应用于外部可见的状态，即那些可以通过调用其公共方法和访问其公共字段看到的状态，而不应用于其私有字段中表示的内部状态)[1]。

类要成为线程安全的，首先必须在单线程环境中有正确的行为。如果一个类实现正确(这是说它符合规格说明的另一种方式)，那么没有一种对这个类的对象的操作序列(读或者写公共字段以及调用公共方法)可以让对象处于无效状态，观察到对象处于无效状态、或者违反类的任何不可变量、前置条件或者后置条件的情况。

此外，一个类要成为线程安全的，在被多个线程访问时，不管运行时环境执行这些线程有什么样的时序安排或者交错，它必须仍然有如上所述的正确行为，并且在调用的代码中没有任何额外的同步。其效果就是，在所有线程看来，对于线程安全对象的操作是以固定的、全局一致的顺序发生的。

正确性与线程安全性之间的关系非常类似于在描述 ACID(原子性、一致性、独立性和持久性)事务时使用的一致性与独立性之间的关系：从特定线程的角度看，由不同线程所执行的对象操作是先后(虽然顺序不定)而不是并行执行的。

考虑下面的代码片段，它迭代一个 Vector 中的元素。尽管 Vector 的所有方法都是同步的，但是在多线程的环境中不做额外的同步就使用这段代码仍然是不安全的，因为如果另一个线程恰好在错误的时间里删除了一个元素，则 get() 会抛出一个 ArrayIndexOutOfBoundsException 。

```
 Vector v = new Vector();
// contains race conditions -- may require external synchronization
for (int i=0; i<v.size(); i++) {
  doSomething(v.get(i));
}
```

这里发生的事情是：get(index) 的规格说明里有一条前置条件要求 index 必须是非负的并且小于 size() 。但是，在多线程环境中，没有办法可以知道上一次查到的 size() 值是否仍然有效，因而不能确定 i<size() ，除非在上一次调用了 size() 后独占地锁定 Vector 。

更明确地说，这一问题是由 get() 的前置条件是以 size() 的结果来定义的这一事实所带来的。只要看到这种必须使用一种方法的结果作为另一种讲法的输入条件的样式，它就是一个**状态依赖**，就必须保证至少在调用这两种方法期间元素的状态没有改变。一般来说，做到这一点的唯一方法在调用第一个方法之前是独占性地锁定对象，一直到调用了后一种方法以后。在上面的迭代 Vector 元素的例子中，您需要在迭代过程中同步 Vector 对象。

如上面的例子所示，线程安全性不是一个非真即假的命题。Vector 的方法都是同步的，并且 Vector 明确地设计为在多线程环境中工作。但是它的线程安全性是有限制的，即在某些方法之间有状态依赖(类似地，如果在迭代过程中 Vector 被其他线程修改，那么由 Vector.iterator() 返回的 iterator 会抛出 ConcurrentModificationException)。

对于 Java 类中常见的线程安全性级别，没有一种分类系统可被广泛接受，不过重要的是在编写类时尽量记录下它们的线程安全行为。

Bloch 给出了描述五类线程安全性的分类方法：不可变、线程安全、有条件线程安全、线程兼容和线程对立。只要明确地记录下线程安全特性，那么您是否使用这种系统都没关系。这种系统有其局限性——各类之间的界线不是百分之百地明确，而且有些情况它没照顾到，但是这套系统是一个很好的起点。这种分类系统的核心是调用者是否可以或者必须用外部同步包围操作(或者一系列操作)。下面分别描述了线程安全性的这五种类别。

1) 不可变

不可变的对象一定是线程安全的，并且永远也不需要额外的同步。因为一个不可变的对象只要构建正确，其外部可见状态永远也不会改变，永远也不会看到它处于不一致的状态。Java 类库中大多数基本数值类如 Integer、String 和 BigInteger 都是不可变的。

2) 线程安全

由类的规格说明所规定的约束在对象被多个线程访问时仍然有效，不管运行时环境如何排列，线程都不需要任何额外的同步。这种线程安全性保证是很严格的——许多类，如 Hashtable 或者 Vector 都不能满足这种严格的定义。

3) 有条件的线程安全

有条件的线程安全类对于单独的操作可以是线程安全的，但是**某些操作序列可能需要外部同步**。条件线程安全的最常见的例子是遍历由 Hashtable 或者 Vector 或者返回的迭代器——由这些类返回的 fail-fast 迭代器假定在迭代器进行遍历的时候底层集合不会有变化。为了保证其他线程不会在遍历的时候改变集合，进行迭代的线程应该确保它是独占性地访问集合以实现遍历的完整性。通常，独占性的访问是由对锁的同步保证的——并且类的文档应该说明是哪个锁(通常是对象的内部监视器(intrinsic monitor))。

如果对一个有条件线程安全类进行记录，那么您应该不仅要记录它是有条件线程安全的，而且还要记录必须防止哪些操作序列的并发访问。用户可以合理地假设其他操作序列不需要任何额外的同步。

4) 线程兼容

线程兼容类不是线程安全的，但是可以通过正确使用同步而在并发环境中安全地使用。这可能意味着用一个 synchronized 块包围每一个方法调用，或者创建一个包装器对象，其中每一个方法都是同步的(就像 Collections.synchronizedList() 一样)。也可能意味着用 synchronized 块包围某些操作序列。为了最大程度地利用线程兼容类，如果所有调用都使用同一个块，那么就不应该要求调用者对该块同步。这样做会使线程兼容的对象作为变量实例包含在其他线程安全的对象中，从而可以利用其所有者对象的同步。

许多常见的类是线程兼容的，如集合类 ArrayList 和 HashMap 、java.text.SimpleDateFormat 、或者 JDBC 类 Connection 和 ResultSet 。

5) 线程对立

线程对立类是那些不管是否调用了外部同步都不能在并发使用时安全地呈现的类。线程对立很少见，当类修改静态数据，而静态数据会影响在其他线程中执行的其他类的行为，这

时通常会出现线程对立。线程对立类的一个例子是调用 System.setOut() 的类。

线程安全类(以及线程安全性程度更低的的类) 可以允许或者不允许调用者锁定对象以进行独占性访问。Hashtable 类对所有的同步使用对象的内部监视器，但是 ConcurrentHashMap 类不是这样，事实上没有办法锁定一个 ConcurrentHashMap 对象以进行独占性访问。除了记录线程安全程序，还应该记录是否某些锁——如对象的内部锁——对类的行为有特殊的意义。

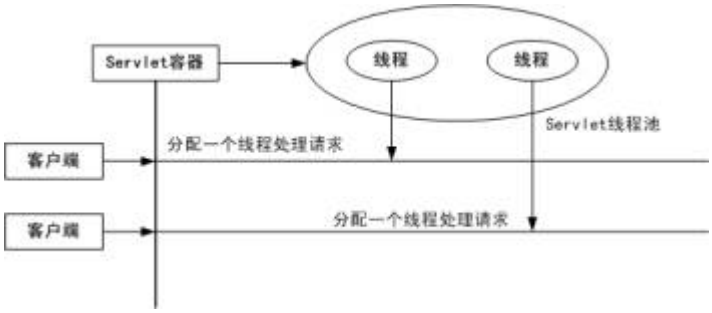
通过将类记录为线程安全的(假设它确实是线程安全的)，您就提供了两种有价值的服务：您告知类的维护者不要进行会影响其线程安全性的修改或者扩展，您还告知类的用户使用它时可以不使用外部同步。通过将类记录为线程兼容或者有条件线程安全的，您就告知了用户这个类可以通过正确使用同步而安全地在多线程中使用。通过将类记录为线程对立的，您就告知用户即使使用了外部同步，他们也不能在多线程中安全地使用这个类。不管是哪种情况，您都在潜在的严重问题出现之前防止了它们，而要查找和修复这些问题是很昂贵的。

一个类的线程安全行为是其规格说明中的固有部分，应该成为其文档的一部分。因为还没有描述类的线程安全行为的声明式方式，所以必须用文字描述。虽然 Bloch 的描述类的线程安全程度的五层系统没有涵盖所有可能的情况，但是它是一个很好的起点。如果每一个类都将这种线程行为的程度加入到其 Javadoc 中，那么可以肯定的是我们大家都会受益。

2.2. Servlet 的线程安全性

Servlet/JSP 默认是以多线程模式执行的，所以，在编写代码时需要非常细致地考虑多线程的安全性问题。然而，很多人编写 Servlet/JSP 程序时并没有注意到多线程安全性的问题，这往往造成编写的程序在少量用户访问时没有任何问题，而在并发用户上升到一定值时，就会经常出现一些莫明其妙的问题。

Servlet 体系结构是建立在 Java 多线程机制之上的，它的生命周期是由 Web 容器负责的。当客户端第一次请求某个 Servlet 时，Servlet 容器将会根据 web.xml 配置文件实例化这个 Servlet 类。当有新的客户端请求该 Servlet 时，一般不会再实例化该 Servlet 类，也就是有多个线程在使用这个实例。Servlet 容器会自动使用线程池等技术来支持系统的运行。



这样，当两个或多个线程同时访问同一个 Servlet 时，可能会发生多个线程同时访问同一资源的情况，数据可能会变得不一致。所以在用 Servlet 构建的 Web 应用时如果不注意线程安全的问题，会使所写的 Servlet 程序有难以发现的错误。

1. 无状态 Servlet

下面是一个无状态的 Servlet，它从 Request 中解包数据，然后将这两个数据进行相乘，最后把结果封装在 Response 中。

```
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
public class ConcurrentServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;
    public ConcurrentServlet() {
        super();
    }
    protected void doGet(HttpServletRequest request,
            HttpServletResponse response) throws ServletException,
IOException {
        String s1 = request.getParameter("num1");
        String s2 = request.getParameter("num2");
        int result = 0;
        if (s1 != null && s1 != null) {
            result = Integer.parseInt(s1) * Integer.parseInt(s2);
        }
        PrintWriter out = response.getWriter();
        out.print(result);
        out.close();
    }
}
```

这个 Servlet 是无状态的，它不包含域，也没有引用其它类的域，一次特定计算的瞬时状

态，会唯一的存储在本地变量中，这些本地变量存在线程的栈中，只有执行线程才能访问，一个执行该 Servlet 的线程不会影响访问同一个 Servlet 的其它线程的计算结果，因为两个线程不共享状态，他们如同在访问不同的实例。

因为线程访问无状态对象的行为，不会影响其它线程访问对象时的正确性，所以无状态对象是线程安全的。

2 有状态 Servlet

对上面的 Servlet 进行修改，把 result 变量提升为类的实例变量。那么这个 Servlet 就有状态了。有状态的 Servlet 在多线程访问时，有可能发生线程不安全性。请看下面的代码。

```
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
public class StatefulServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;
    int result = 0;
    public StatefulServlet() {
        super();
    }
    protected void doGet(HttpServletRequest request,
            HttpServletResponse response) throws ServletException,
IOException {
        String s1 = request.getParameter("num1");
        String s2 = request.getParameter("num2");
        if (s1 != null && s1 != null) {
            result = Integer.parseInt(s1) * Integer.parseInt(s2);
        }
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        PrintWriter out = response.getWriter();
        out.print(result);
        out.close();
    }
}
```

在 Servlet 中定义了一个实例变量 result，Servlet 把它的值进行输出。当只有一个用户访问该 Servlet 时，程序会正常的运行，但当多个用户并发访问时，就可能会出现其它用户的信

息显示在另外一些用户的浏览器上的问题 。这是一个严重的问题。

为了突出并发问题，便于测试、观察，我们在回显用户信息时执行了一个延时的操作。

打开两个浏览器窗口，分别输入：

http://localhost:8080/test/StatefulServlet?num1=5&num2=80

http://localhost:8080/test/StatefulServlet?num1=5&num2=70。

相隔 5000 毫秒之内执行这两个请求，产生的结果如下图：

从运行结果可以看出，两个请求显示了相同的计算结果，也就是说，因为两个线程访问了共同的有状态的 Servlet，其中一个线程的计算结果覆盖了另外一个线程的计算结果。从程序分析可以看出第一个线程在输出 result 时，暂停了一段时间，那么它的值就被第二个线程的计算结果所覆盖，两个请求输出了相同的结果。这就是潜在的线程不安全性。

要解决线程不安全性，其中一个主要的方法就是取消 Servlet 的实例变量，变成无状态的 Servlet。另外一种方法是对共享数据进行同步操作。使用 synchronized 关键字能保证一次只有一个线程可以访问被保护的区段，同步后的 Servlet 如下：

```
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
public class StatefulServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;
    int result = 0;
    public StatefulServlet() {
        super();
    }
    protected void doGet(HttpServletRequest request,
            HttpServletResponse response) throws ServletException,
```

```
IOException {
        String s1 = request.getParameter("num1");
        String s2 = request.getParameter("num2");
        synchronized (this) {
            if (s1 != null && s1 != null) {
                result = Integer.parseInt(s1) * Integer.parseInt(s2);
            }
            try {
                Thread.sleep(5000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            PrintWriter out = response.getWriter();
            out.print(result);
            out.close();
        }
    }
}
```

Servlet 的线程安全问题只有在大量的并发访问时才会显现出来，并且很难发现，因此在编写 Servlet 程序时要特别注意。线程安全问题主要是由实例变量造成的，因此在 Servlet 中应避免使用实例变量。如果应用程序设计无法避免使用实例变量，那么使用同步来保护要使用的实例变量，但为保证系统的最佳性能，应该同步可用性最小的代码路径。

2.3. 同步与互斥

线程通信主要通过共享访问字段或者字段引用的对象完成的，但是有可能出现两种错误：线程干扰（thread interference）和内存一致性错误(memory consistency)。用来防止这些错误的工具是同步(synchronization)。

2.3.1 线程干扰

为了便于说明线程干扰的问题，定义一个银行帐户类 BankAccount，有两个方法：取款的方法 withdraw 和存款的方法 deposit。并定义不同的线程进行存款和取款。存款线程每次存 100 元，取款线程每次取 100 元。各运行 10 万次。

```
package sync;
public class BankAccount {
    private int number;
```

```
    private int balance;
    public BankAccount(int number, int balance) {
        this.number = number;
        this.balance = balance;
    }
    public int getBalance() {
        return balance;
    }
    public void deposit(int amount) {
        balance = balance + amount;
    }
    public void withdraw(int amount) {
        balance = balance - amount;
    }
    public static void main(String[] args) throws InterruptedException {
        BankAccount a = new BankAccount(1, 1000);
        Thread t1 = new Thread(new Depositor(a, 100), "depositor");
        Thread t2 = new Thread(new Withdrawer(a, 100), "withdraw");
        t1.start();
        t2.start();
         t1.join();
         t2.join();
        System.out.println(a.getBalance());
    }
    static class Depositor implements Runnable {
        BankAccount account;
        int amount;
        public Depositor(BankAccount account, int amount) {
            this.account = account;
            this.amount = amount;
        }
        @Override
        public void run() {
            for (int i = 0; i < 100000; i++)
                account.deposit(amount);
        }
    }
    static class Withdrawer implements Runnable {
        BankAccount account;
        int amount;
        public Withdrawer(BankAccount account, int amount) {
            this.account = account;
            this.amount = amount;
        }
```

```
        @Override
        public void run() {
            for (int i = 0; i < 100000; i++)
                account.withdraw(amount);
        }
    }
}
```

帐户的初始余额为 1000 元。取款线程和存款线程各运行 10 万次后，程序的运行结果如下：

```
<terminated> BankAccount [Java Application] D:\Java\Jdk6\bin\javaw.exe (2008-12-29 上午11:10:25)
1000
```

再运行一次，程序的运行结果如下：

```
<terminated> BankAccount [Java Application] D:\Java\Jdk6\bin\javaw.exe (2008-12-29 上午11:11:43)
10001000
```

分析运行结果，第一个结果 “1000” 符合我们的预期，第二个结果“10001000”不符合我们的预期，但是它确实是一个程序的最终运行结果。这就是因为线程之间的干扰导致的预期之外的结果。

当运行在不同线程中的两个操作对相同数据进行操作时，就会出现干扰，就是说，两个操作有多个步骤组成，并且操作步骤的序列重叠了。

BankAccount 中的操作似乎不可能重叠，他们都是单一的简单语句，但是即使单一的语句也可能被虚拟机转换为多个步骤。

“balance = balance - amount;”一般可能会分解成 3 个步骤：1）取出 balance 的值，2）执行减法，3）计算结果赋值给 balance。

假设线程 t1 执行 deposit 操作时，线程 t2 几乎同时执行 withdraw 操作，帐户的初始值为 1000，那么当存款的初始化值为 1000 时，取款的初始值也为 1000，存款操作的结果可能覆盖取款操作的结果，balance 变为 1100。10 万次操作后，就会形成比较严重的误差。

2.3.2 同步

当两个线程需要使用同一个对象时，存在交叉操作而破坏数据的可能性。这种潜在的干扰动作在术语上被称作临界区（critical section）。通过同步（Synchronize）对临界区的访问可以避免这种线程干扰。

某些动作操作对象之前，必须先获得这个对象的锁。获取待操作对象上的锁可以阻止其

他对象获取这个锁，直至这个锁的持有者释放它为止。这样，多线程就不会同时执行那些会互相干扰的动作。

同步是围绕被称为**内在锁**（intrinsic lock）或者监视器锁（monitor lock）的内部实体构建的，强制对对象状态的独占访问，以及建立可见性所需的发生前关系。

每个对象都具有与其关联的内在锁，按照约定，需要对对象的字段进行独占和一致性访问的线程，在进行访问之前，必须获得这个对象的内在锁，访问操作完成之后必须释放内在锁。在从获得锁到释放锁的时间段内，线程被称为拥有内在锁。只要有线程拥有内在锁，其他线程就不能获得同一个锁，试图获得锁的其他线程将被阻塞。

Java 提供了 synchronized 关键字来支持内在锁。Synchronized 关键字可以放在方法的前面、对象的前面、类的前面。

1. 同步方法中的锁

当线程调用同步方法时，它自动获得这个方法所在对象的内在锁，并且方法返回时释放锁，如果发生未捕获的异常，也会释放锁。

当调用静态同步方法时，因为静态方法和类相关联，线程获得和这个类关联的 Class 对象的内在锁。

使用内在锁后，把 deposit 方法和 withdraw 方法修改为同步方法，就可以避免线程干扰。

```
public synchronized void deposit(int amount) {
    balance = balance + amount;
}
public synchronized void withdraw(int amount) {
    balance = balance - amount;
}
```

2. 同步语句

同步语句必须指定提供内在锁的对象，其基本用法如下：

synchronized（提供锁的对象）{

    临界代码

}

用同步语句修改 BankAccount 类中的方法如下：

```
public void deposit(int amount) {
    synchronized (this) {
        balance = balance + amount;
    }
}
```

```
    public void withdraw(int amount) {
        synchronized (this) {
            balance = balance - amount;
        }
    }
```

3. 同步类

把 synchronized 关键字放在类的前面，这个类中的所有方法都是同步方法。

4. 可重入同步

线程可以获得他已经拥有的锁，运行线程多次获得同一个锁，就是可以重入（reentrant）同步。这种情况通常是同步代码直接或者间接的调用也包含了同步代码的方法，并且两个代码集都使用同一个锁。如果没有可重入同步，那么，同步代码就必须采取很多额外的预防措施避免线程阻塞自己。

2.4. 同步与 volatile

任何被不同线程所共享的可变值应该总是被同步的访问以防止发生干扰，然而同步是需要代价的。Java 可以保证对任何变量的读写都是原子性的，原子（atomic）操作是必须同时完成的操作，这样变量就只会持有某个线程写入的值，而绝不会持有两个不同线程写入的部分交叉混合的值。这意味着原子变量只能有一个线程来写，多个线程来读，因此不需要对他的访问进行同步以防止数据被破坏，因为这些访问之间不存在互相干扰的可能性。但这对“获取-修改-设置”（如++操作）没有任何帮助，这种操作需要同步。

需要注意的是，原子访问并不能保证线程总是会读取变量最近的写入值，如果没有同步，**一个线程的写入值对另一个线程可能永远都不会是可见的**。有很多因为会影响一个线程写入的变量何时会对另一个线程变为可见的。当缺乏同步机制时，不同线程发现被更新变量的顺序也可以完全不同。 **在确定内存访问如何排序以及合适，可以确保他们可见时所使用的规则被称为 Java 编程语言的内存模型**。

线程所读取的所有变量的值都是由内存模型决来决定的，因为内存模型定义了变量被读取时允许返回的值集合。从程序员的角度看，这个值集合应该只包含单一的值，即由某个线程最近写入的值。然而在缺乏同步时，实际获得的值集合可能包含许多不同的值。

假设 BankAccount 中的字段 balancek 可以被一个线程不断的显示，并且可以由其他线程使用非同步的方法对其进行修改。

```
    //更新
```

```
    public void updateBalance() {
        balance = (int) (Math.random() * 100);
    }
    //显示
    public void showValue() throws InterruptedException {
        balance = 10;
        for (;;) {
            showBalance(balance);
            Thread.sleep(1000);
        }
    }
```

当第一次进行循环时，balance 唯一可能的值是 10，由于没有使用线程同步，所以每当由线程调用 updateBalance 时，都会有新值被添加到所要读取的可能值集合中。当在循环中读取 balance 时，可能值也许已经包含了 10，20，25，35 和 78，其中任何一个值都可以通过读取操作返回，因为根据内存模型的规则，任何被某个线程写入的值都可以通过读取操作返回。实际上，如果 showValue 无法改变 balance 的值，那么编译器就会假设他可以认为 balance 在循环体内未发生改变，从而在每次调用 showValue 时直接使用常量 10 来表示 balance。这种策略和内存模型是一致的。内存模型没有控制要返回哪一个值。

为了让程序能像我们所描述的那样运行，我们必须使得在写入 balance 时，写入值可以成为内存模型唯一允许读取的值。要做到这一点，必须对写入和读取的操作进行同步。

Java 提供了一种同步机制，它不提供对锁的独占访问，但同样可以确保对变量的每一个读取操作都返回最近写入的值，这种机制就是只用 **volatile 变量**。字段变量可以用修饰符 volatile 来声明，volatile 变量的写入操作将与随后所有这个变量的读取操作进行同步。如果 balance 被声明为 volatile，那么我们所给出的示例代码就会被正确的同步，并且总是会显示最新的值。volatile 变量并没有提供可以跨多个动作的原子性，经常被用作简单的标记以表示发生了某个事件，或者被用来编写无锁算法（lock-free）。

将变量设置为 volatile 所产生的另一个效果就是可以确保读写操作都是原子性的。

2.5. 活性

并发应用程序按照及时方式执行的能力称为活性（liveness）[2]。一般包括三种类型的问题死锁、饿死和活锁。

1. 死锁

线程死锁是并发程序设计中可能遇到的主要问题之一。他是指程序运行中，多个线程竞

争共享资源时可能出现的一种系统状态，每个线程都被阻塞，都不会结束，进入一种永久等待状态。

可能发生死锁的最典型的例子是哲学家用餐问题：五个哲学家围坐在一圆桌旁，每人的两边放着一支筷子，共 5 支筷子。大家边讨论问题边用餐，并规定如下条件：1）每个人只有拿起位于自己两边的筷子，合成一双才可以用餐；2）用餐后，每人必须将两支筷子放回原处。

可以想想，如果每个哲学家都彬彬有礼，并且高谈阔论，轮流吃饭，则这种融洽的气氛可以长久的保持下去。但是可能出现这样一种情景：当每个人都拿起自己左手边的筷子，并同时去拿自己右手边的筷子时，5 个人每人拿着一根筷子，盯着自己右手边那位哲学家手里的筷子，处于僵持状态。这就发生了线程死锁。

另一个线程死锁的例子是两个朋友 A 和 B 鞠躬，都非常讲礼貌，礼貌的一个严格规则是，当你向朋友鞠躬时，必须保持鞠躬状态，知道你的朋友向你还礼。

两个朋友可能同时向对方鞠躬，当朋友 A 和朋友 B 同时向对方鞠躬时，都在等待对方起身，进入阻塞状态。发生线程死锁。

2. 饿死

饿死（starvation）描述这样的情况：一个线程不能获得对共享资源的常规访问，并且不能继续工作，当共享资源被贪婪线程长期占有而不可用时，就会发生这样的情况。

3. 活锁

一个线程经常对另一个线程的操作作出响应，如果另一个线程的操作也对这个线程的操作作出响应，那么就可能导致活锁（livelock）。和死锁类似，发生活锁的线程不能进行进一步操作。但是，线程没有被锁定，它只是忙于相互响应，以致不能恢复工作。

活锁可以比喻为两人在走廊中相遇。A 避让的自己的左边让 B 通过，而 B 同时避让到自己的右边让 A 通过。发现他们仍然挡住了对方，A 就避让到自己的右边，而 B 同时避让到了自己的左边，他们还是挡住了对方，所以就没完没了。

2.6. ThreadLocal 变量

早在 JDK 1.2 的版本中就提供 java.lang.ThreadLocal，为解决多线程程序的并发问题提供了一种新的思路。使用这个工具类可以很简洁地编写出优美的多线程程序。

ThreadLocal 很容易让人望文生义，想当然地认为是一个“本地线程”。其实，ThreadLocal 并不是一个 Thread，而是 Thread 的局部变量，也许把它命名为 ThreadLocalVariable 更容易让

人理解一些。当使用 ThreadLocal 维护变量时，ThreadLocal 为每个使用该变量的线程提供独立的变量副本，所以每一个线程都可以独立地改变自己的副本，而不会影响其它线程所对应的副本。

从线程的角度看，目标变量就是线程的本地变量，这也是类名中“Local”所要表达的意思。线程局部变量并不是 Java 的新发明，很多语言（如 IBM XL FORTRAN）在语法层面就提供线程局部变量。在 Java 中没有提供语言级支持，而是变相地通过 ThreadLocal 的类提供支持。

JDK 5 以后提供了泛型支持，ThreadLocal 被定义为支持泛型：

public class ThreadLocal<T> extends Object

T 为线程局部变量的类型。该类定义了 4 个方法：

1） protected T initialValue()：返回此线程局部变量的当前线程的“初始值”。线程第一次使用 get() 方法访问变量时将调用此方法，但如果线程之前调用了 set(T) 方法，则不会对该线程再调用 initialValue 方法。通常，此方法对每个线程最多调用一次，但如果在调用 get() 后又调用了 remove()，则可能再次调用此方法。

该实现返回 null；如果程序员希望线程局部变量具有 null 以外的值，则必须为 ThreadLocal 创建子类，并重写此方法。通常将使用匿名内部类完成此操作。

2）public T get()：返回此线程局部变量的当前线程副本中的值。如果变量没有用于当前线程的值，则先将其初始化为调用 initialValue() 方法返回的值。

3）public void set(T value)：将此线程局部变量的当前线程副本中的值设置为指定值。大部分子类不需要重写此方法，它们只依靠 initialValue() 方法来设置线程局部变量的值。

4）public void remove()：移除此线程局部变量当前线程的值。如果此线程局部变量随后被当前线程读取，且这期间当前线程没有设置其值，则将调用其 initialValue() 方法重新初始化其值。这将导致在当前线程多次调用 initialValue 方法。

下面是一个使用 ThreadLocal 的例子，每个线程产生自己独立的序列号。就是使用 ThreadLocal 存储每个线程独立的序列号复本，线程之间互不干扰。

```
package sync;
public class SequenceNumber {
    // 定义匿名子类创建ThreadLocal的变量
    private static ThreadLocal<Integer> seqNum = new
ThreadLocal<Integer>() {
        // 覆盖初始化方法
        public Integer initialValue() {
```

```
            return 0;
        }
    };
    // 下一个序列号
    public int getNextNum() {
        seqNum.set(seqNum.get() + 1);
        return seqNum.get();
    }
    private static class TestClient extends Thread {
        private SequenceNumber sn;
        public TestClient(SequenceNumber sn) {
            this.sn = sn;
        }
        // 线程产生序列号
        public void run() {
            for (int i = 0; i < 3; i++) {
                System.out.println("thread[" +
Thread.currentThread().getName()
                        + "] sn[" + sn.getNextNum() + "]");
            }
        }
    }
    /**
     * @param args
     */
    public static void main(String[] args) {
        SequenceNumber sn = new SequenceNumber();
        // 三个线程产生各自的序列号
        TestClient t1 = new TestClient(sn);
        TestClient t2 = new TestClient(sn);
        TestClient t3 = new TestClient(sn);
        t1.start();
        t2.start();
        t3.start();
    }
}
```

程序的运行结果如下：

```
thread[Thread-1] sn[1]
thread[Thread-1] sn[2]
thread[Thread-1] sn[3]
thread[Thread-2] sn[1]
thread[Thread-2] sn[2]
thread[Thread-2] sn[3]
thread[Thread-0] sn[1]
```

```
thread[Thread-0] sn[2]
thread[Thread-0] sn[3]
```

从运行结果可以看出，使用了 ThreadLocal 后，每个线程产生了独立的序列号，没有相互干扰。通常我们通过匿名内部类的方式定义 ThreadLocal 的子类，提供初始的变量值。

ThreadLocal 和线程同步机制相比有什么优势呢？**ThreadLocal 和线程同步机制都是为了解决多线程中相同变量的访问冲突问题。**

在同步机制中，通过对象的锁机制保证同一时间只有一个线程访问变量。这时该变量是多个线程共享的，使用同步机制要求程序慎密地分析什么时候对变量进行读写，什么时候需要锁定某个对象，什么时候释放对象锁等繁杂的问题，程序设计和编写难度相对较大。

而 ThreadLocal 则从另一个角度来解决多线程的并发访问。ThreadLocal 会为每一个线程提供一个独立的变量副本，从而隔离了多个线程对数据的访问冲突。因为每一个线程都拥有自己的变量副本，从而也就没有必要对该变量进行同步了。ThreadLocal 提供了线程安全的共享对象，在编写多线程代码时，可以把不安全的变量封装进 ThreadLocal。

概括起来说，对于多线程资源共享的问题，同步机制采用了“以时间换空间”的方式，而 ThreadLocal 采用了“以空间换时间”的方式。前者仅提供一份变量，让不同的线程排队访问，而后者为每一个线程都提供了一份变量，因此可以同时访问而互不影响。

需要注意的是 ThreadLocal 对象是一个本质上存在风险的工具，应该在完全理解将要使用的线程模型之后，再去使用 ThreadLocal 对象。这就引出了线程池（thread pooling）的问题，线程池是一种线程重用技术，有了线程池就不必为每个任务创建新的线程，一个线程可能会多次使用，用于这种环境的任何 ThreadLocal 对象包含的都是最后使用该线程的代码所设置的状态，而不是在开始执行新线程时所具有的未被初始化的状态。

那么 ThreadLocal 是如何实现为每个线程保存独立的变量的副本的呢？通过查看它的源代码，我们会发现，是通过把当前“线程对象”当作键，变量作为值存储在一个 Map 中。

```
private T setInitialValue() {
       T value = initialValue();
       Thread t = Thread.currentThread();
       ThreadLocalMap map = getMap(t);
       if (map != null)
           map.set(this, value);
       else
           createMap(t, value);
       return value;
}
```

2.7. 高级并发对象

本章重点介绍的是低级别的 API，都是 Java 平台最基本的组成部分，这些都足以胜任基本的任务，但是更加高级的任务需要更高级别的 API，对应充分利用现代多处理器和多核心系统功能的大规模并发应用程序来说，这尤其重要。

JDK5.0 以后的版本都引入了高级并发特性，并且新的版本在不断的补充和完善。大多数的特性在 java.util.concurrent 包中实现，Java 集合框架中也有新的并发数据结构。

主要增加的高级并发对象有：Lock 对象，执行器，并发集合、原子变量和同步器。具体用法请参考第三章。

1）Lock 对象

前面介绍的同步代码依靠简单类型的可重入锁，即内部锁（隐式锁）。这种类型的锁易于使用，但是有很多局限性。新的 Lock 对象支持更加复杂的锁定语法。

和隐式锁类似，每一时刻只有一个线程能够拥有 Lock 对象，通过与其相关联的 Condition 对象，Lock 对象也支持 wait 和 notify 机制。Lock 对象的最大优势在于能够阻挡获得锁的企图。如果锁不能立即可用或者在超时时间到期之前可用，tryLock 方法就会阻挡，如果另一个线程在获得锁之前发送中断，lockInterruptibly 方法就会阻挡。

2）执行器

前面例子，线程完成的任务（Runnable 对象）和线程对象（Thread）之间紧密相连。适用于小型程序，在大型应用程序中，把线程管理和创建工作与应用程序的其余部分分离开更有意义。封装线程管理和创建的对象被称为执行器（Executor）。

JDK 中定义了 3 个执行器接口：Executor，ExecutorService 和 ScheduledExecutorService。

3）并发集合

并发集合是原有集合框架的补充，为多线程并发程序提供了支持。主要有：BlockingQueue，ConcurrentMap，ConcurrentNavigableMap。

4）原子变量

定义了支持对单一变量执行原子操作的类。所有类都有 get 和 set 方法，工作方法和对 volatile 变量的读取和写入一样。

5）同步器

提供了一些帮助在线程间协调的类，包括 semaphores, mutexes, barriers, latches, exchangers 等。

参考文献

[1] 描述线程安全性. http://www.ibm.com/developerworks/cn/java/j-jtp09263/

[2] (美)扎克雷尔等. Java 教程，第四版.人民邮电出版社 2007.9

[3] Brian Goetz, Tim Peierls, Joshua Bloch. Java Concurrency in Practice. Addison Wesley Professional,2006.9

第三章 使用 JDK 并发包构建程序

第三章 使用JDK并发包构建程序 1
3.1 java.util.concurrent概述 2
3.2 原子量 2
3.2.1 锁同步法 3
3.2.2 比较并交换 4
3.2.3 原子变量类 6
3.2.4 使用原子量实现银行取款 8
3.3 并发集合 12
3.3.1 队列Queue与BlockingQueue 12
3.3.2 使用 ConcurrentMap 实现类 19
3.3.3 CopyOnWriteArrayList和CopyOnWriteArraySet 20
3.4 同步器 21
3.4.1 Semaphore 21
3.4.2 Barrier 24
3.4.3 CountDownLatch 27
3.4.4 Exchanger 29
3.4.5 Future和FutureTask 31
3.5 显示锁 33
3.5.1 ReentrantLock 33
3.5.1.1 ReentrantLock的特性 34
3.5.1.2 ReentrantLock性能测试 38
3.5.2 ReadWriteLock 42
3.6 Fork-Join框架 46
3.6.1 应用Fork-Join 47
3.6.2 应用ParallelArray 51
参考文献 52

3.1 java.util.concurrent 概述

JDK5.0 以后的版本都引入了高级并发特性，大多数的特性在 java.util.concurrent 包中，是专门用于多线并发编程的，充分利用了现代多处理器和多核心系统的功能以编写大规模并发应用程序。主要包含原子量、并发集合、同步器、可重入锁，并对线程池的构造提供了强力的支持。

原子量是定义了支持对单一变量执行原子操作的类。所有类都有 get 和 set 方法，工作方法和对 volatile 变量的读取和写入一样。

并发集合是原有集合框架的补充，为多线程并发程序提供了支持。主要有：BlockingQueue，ConcurrentMap，ConcurrentNavigableMap。

同步器提供了一些帮助在线程间协调的类，包括 semaphores,barriers,latches,exchangers 等。

一般同步代码依靠内部锁（隐式锁），这种锁易于使用，但是有很多局限性。新的 Lock 对象支持更加复杂的锁定语法。和隐式锁类似，每一时刻只有一个线程能够拥有 Lock 对象，通过与其相关联的 Condition 对象，Lock 对象也支持 wait 和 notify 机制。

线程完成的任务（Runnable 对象）和线程对象（Thread）之间紧密相连。适用于小型程序，在大型应用程序中，把线程管理和创建工作与应用程序的其余部分分离开更有意义。线程池封装线程管理和创建线程对象。线程池在第一章已经讲过，不再赘述。

3.2 原子量

近来关于并发算法的研究主要焦点是无锁算法（nonblocking algorithms），这些无锁算法使用低层原子化的机器指令，例如使用compare-and-swap（CAS）代替锁保证并发情况下数据的完整性。无锁算法广泛应用于操作系统与JVM中，比如线程和进程的调度、垃圾收集、实现锁和其他并发数据结构。

在 JDK5.0 之前，如果不使用本机代码，就不能用 Java 语言编写无等待、无锁定的算法。在 java.util.concurrent 中添加原子变量类之后，这种情况发生了变化。本节了解这些新类开发高度可伸缩的无阻塞算法。

要使用多处理器系统的功能，通常需要使用多线程构造应用程序。但是正如任何编写并发应用程序的人可以告诉你的那样，要获得好的硬件利用率，只是简单地在多个线程中分割工作是不够的，还必须确保线程确实大部分时间都在工作，而不是在等待更多的工作，或等待锁定共享数据结构。

如果线程之间不需要协调，那么几乎没有任务可以真正地并行。以线程池为例，其中执行的任务通常相互独立。如果线程池利用公共工作队列，则从工作队列中删除元素或向工作队列添加元素的过程必须是线程安全的，并且这意味着要协调对头、尾或节点间链接指针所进行的访问。正是这种协调导致了所有问题。

3.2.1 锁同步法

在 Java 语言中，协调对共享字段访问的传统方法是使用**同步**，确保完成对共享字段的所有访问，同时具有适当的锁定。通过同步，可以确定（假设类编写正确）具有保护一组访问变量的所有线程都将拥有对这些变量的独占访问权，并且以后其他线程获得该锁定时，将可以看到对这些变量进行的更改。弊端是如果**锁定**竞争太厉害（线程常常在其他线程具有锁定时要求获得该锁定），会损害吞吐量，因为竞争的同步非常昂贵。对于现代 JVM 而言，无竞争的同步现在非常便宜。

基于锁的算法的另一个问题是：如果延迟具有锁的线程（因为页面错误、计划延迟或其他意料之外的延迟），则没有要求获的锁的线程可以继续运行。

还可以使用 **volatile** 变量来以比同步更低的成本存储共享变量，但它们有局限性。虽然可以保证其他变量可以立即看到对 **volatile** 变量的写入，但无法呈现原子操作的读-修改-写顺序，这意味着 **volatile** 变量无法用来可靠地实现互斥（互斥锁定）或计数器。

下面以实现一个计数器为例。通常情况下一个计数器要保证计数器的增加，减少等操作需要保持原子性，使类成为线程安全的类，从而确保没有任何更新信息丢失，所有线程都看到计数器的最新值。使用内部锁实现的同步代码一般如下：

```
package jdkapidemo;
public class SynchronizedCounter {
    private int value;
    public synchronized int getValue() {
        return value;
    }
    public synchronized int increment() {
```

```
        return ++value;
    }
    public synchronized int decrement() {
        return --value;
    }
}
```

increment() 和 decrement() 操作是原子的读-修改-写操作，为了安全实现计数器，必须使用当前值，并为其添加一个值，或写出新值，所有这些均视为一项操作，其他线程不能打断它。否则，如果两个线程试图同时执行增加，操作的不幸交叉将导致计数器只被实现了一次，而不是被实现两次。(注意，通过使值变量成为 volatile 变量并不能可靠地完成这项操作。)

计数器类可以可靠地工作，在竞争很小或没有竞争时都可以很好地执行。然而，在竞争激烈时，这将大大损害性能，因为 JVM 用了更多的时间来调度线程，管理竞争和等待线程队列，而实际工作（如增加计数器）的时间却很少。

使用锁，如果一个线程试图获取其他线程已经具有的锁，那么该线程将被阻塞，直到该锁可用。此方法具有一些明显的缺点，其中包括当线程被阻塞来等待锁时，它无法进行其他任何操作。如果阻塞的线程是高优先级的任务，那么该方案可能造成非常不好的结果（称为优先级倒置的危险）。

使用锁还有一些其他危险，如死锁（当以不一致的顺序获得多个锁时会发生死锁）。甚至没有这种危险，锁也仅是相对的粗粒度协调机制，同样非常适合管理简单操作，如增加计数器或更新互斥拥有者。如果有更细粒度的机制来可靠管理对单独变量的并发更新，则会更好一些；在大多数现代处理器都有这种机制。

3.2.2 比较并交换

大多数现代处理器都包含对多处理的支持。当然这种支持包括多处理器可以共享外部设备和主内存，同时它通常还包括对指令系统的增加来支持多处理的特殊要求。特别是，几乎每个现代处理器都有通过可以检测或阻止其他处理器的并发访问的方式来更新共享变量的**指令**。

现在的处理器（包括 Intel 和 Sparc 处理器）使用的最通用的方法是实现名为“比较并交换（Compare And Swap）”或 CAS 的原语。（在 Intel 处理器中，比较并交换通过 cmpxchg 系列指令实现。PowerPC 处理器有一对名为“加载并保留”和“条件存储”的指令，它们实现相同的目地；MIPS 与 PowerPC 处理器相似，除了第一个指令称为“加载链

接”。）

CAS 操作包含三个操作数 —— 内存位置（V）、预期原值（A）和新值(B)。如果内存位置的值与预期原值相匹配，那么处理器会自动将该位置值更新为新值。否则，处理器不做任何操作。无论哪种情况，它都会在 CAS 指令之前返回该位置的值。（在 CAS 的一些特殊情况下将仅返回 CAS 是否成功，而不提取当前值。）CAS 有效地说明了“我认为位置 V 应该包含值 A；如果包含该值，则将 B 放到这个位置；否则，不要更改该位置，只告诉我这个位置现在的值即可。”

通常将 CAS 用于同步的方式是从地址 V 读取值 A，执行多步计算来获得新值 B，然后使用 CAS 将 V 的值从 A 改为 B。如果 V 处的值尚未同时更改，则 CAS 操作成功。

类似于 CAS 的指令允许算法执行读-修改-写操作，而无需害怕其他线程同时修改变量，因为如果其他线程修改变量，那么 CAS 会检测它（并失败），算法可以对该操作重新计算。下面的程序说明了 CAS 操作的行为（而不是性能特征），但是 CAS 的价值是它可以在硬件中实现，并且是极轻量级的（在大多数处理器中）。后面我们分析 Java 的源代码可以知道，JDK 在实现的时候使用了本地代码。下面的代码说明 CAS 的工作原理（为了便于说明，用同步语法表示）。

```
package jdkapidemo;
public class SimulatedCAS {
    private int value;
    public synchronized int getValue() {
        return value;
    }
    public synchronized int compareAndSwap(int expectedValue, int
newValue) {
        if (value == expectedValue)
            value = newValue;
        return value;
    }
}
```

基于 CAS 的并发算法称为“无锁定算法”，因为线程不必再等待锁定（有时称为互斥或关键部分，这取决于线程平台的术语）。无论 CAS 操作成功还是失败，在任何一种情况中，它都在可预知的时间内完成。如果 CAS 失败，调用者可以重试 CAS 操作或采取其他适合的操作。下面的代码显示了重新编写的计数器类来使用 CAS 替代锁定：

```
package jdkapidemo;
public class CasCounter {
```

```
    private SimulatedCAS value;
    public int getValue() {
        return value.getValue();
    }
    public int increment() {
        int oldValue = value.getValue();
        while (value.compareAndSwap(oldValue, oldValue + 1) !=
oldValue)
            oldValue = value.getValue();
        return oldValue + 1;
    }
}
```

如果每个线程在其他线程任意延迟（或甚至失败）时都将持续进行操作，就可以说该算法是 **“无等待”**的。**“无锁定算法”**要求某个线程总是执行操作。（无等待的另一种定义是保证每个线程在其有限的步骤中正确计算自己的操作，而不管其他线程的操作、计时、交叉或速度。这一限制可以是系统中线程数的函数；例如，如果有 10 个线程，每个线程都执行一次 CasCounter.increment() 操作，最坏的情况下，每个线程将必须重试最多九次，才能完成增加。）

再过去的 15 年里，人们已经对无等待且无锁算法（也称为**无阻塞算法**）进行了大量研究，许多人通用数据结构已经发现了无阻塞算法。无阻塞算法被广泛用于操作系统和 JVM 级别，进行诸如线程和进程调度等任务。虽然它们的实现比较复杂，但相对于基于锁的备选算法，它们有许多优点：可以避免优先级倒置和死锁等危险，竞争比较便宜，协调发生在更细的粒度级别，允许更高程度的并行机制等等。

3.2.3 原子变量类

java.util.concurrent.atomic 包中添加原子变量类。所有原子变量类都公开**“比较并设置”原语**（与比较并交换类似），这些原语都是使用平台上可用的最快本机结构（比较并交换、加载链接/条件存储，最坏的情况下是旋转锁）来实现的。 java.util.concurrent.atomic 包中提供了原子变量的 9 种风格（AtomicInteger、AtomicLong、AtomicReference、AtomicBoolean、原子整型、长型、引用、及原子标记引用和戳记引用类的数组形式，其原子地更新一对值）。

原子变量类可以认为是 volatile 变量的泛化，它扩展了 volatile 变量的概念，来支持原子条件的比较并设置更新。读取和写入原子变量与读取和写入对 volatile 变量的访问具有相同的存取语义。

虽然原子变量类表面看起来与 SynchronizedCounter 例子一样，但相似仅是表面的。在表面之下，原子变量的操作会变为平台提供的用于并发访问的硬件原语，比如比较并交换。

调整具有竞争的并发应用程序的可伸缩性的通用技术是降低使用的锁对象的粒度，希望更多的锁请求从竞争变为不竞争。从锁转换为原子变量可以获得相同的结果，通过切换为更细粒度的协调机制，竞争的操作就更少，从而提高了吞吐量。

下面的程序是使用原子变量后的计数器：

```
package jdkapidemo;
import java.util.concurrent.atomic.AtomicInteger;
public class AtomicCounter {
    private AtomicInteger value = new AtomicInteger();
    public int getValue() {
        return value.get();
    }
    public int increment() {
        return value.incrementAndGet();
    }
    public int increment(int i) {
        return value.addAndGet(i);
    }
    public int decrement() {
        return value.decrementAndGet();
    }
    public int decrement(int i) {
        return value.addAndGet(-i);
    }
}
```

下面写一个测试类：

```
package jdkapidemo;
public class AtomicCounterTest extends Thread {
    AtomicCounter counter;
    public AtomicCounterTest(AtomicCounter counter) {
        this.counter = counter;
    }
    @Override
    public void run() {
        int i = counter.increment();
        System.out.println("generated number:" + i);
    }
    public static void main(String[] args) {
        AtomicCounter counter = new AtomicCounter();
        for (int i = 0; i < 10; i++) {//10个线程
```

```
            new AtomicCounterTest(counter).start();
        }
    }
}
```

运行结果如下：

```
generated number:1
generated number:2
generated number:3
generated number:4
generated number:5
generated number:7
generated number:6
generated number:9
generated number:10
generated number:8
```

会发现 10 个线程运行中，没有重复的数字，原子量类使用本机 CAS 实现了值修改的原子性。

3.2.4 使用原子量实现银行取款

下面再看一个银行取款的例子，下面定义了一个帐户类 Account，重点关注其中的取款方法 withdraw()，取款前先判断余额是否足够支付，然后把余额减去取款额，为了更好的模拟线程并发的情况，在其中增了一个休眠语句。

```
package jdkapidemo.bank;
public class Account {
    private double balance;
    public Account(double money) {
        balance = money;
        System.out.println("Totle Money: " + balance);
    }
    public void deposit(double money) {
        balance = balance + money;
    }
    public void withdraw(double money, int delay) {
        if (balance >= money) {
            try {
                Thread.sleep(delay);
                balance = balance - money;
                System.out.println(Thread.currentThread().getName()
                        + " withdraw  " + money + " successful!" +
balance);
            } catch (InterruptedException e) {
```

```
            }
        } else
            System.out.println(Thread.currentThread().getName()
                    + " balance is not enough, withdraw failed!" +
balance);
    }
}
```

为了测试帐户类，定义一个测试类

```
package jdkapidemo.bank;
public class AccountThread extends Thread {
    Account account;
    int delay;
    public AccountThread(Account acount, int delay) {
        this.account = acount;
        this.delay = delay;
    }
    public void run() {
        account.withdraw(100, delay);
    }
    public static void main(String[] args) {
        Account acount = new Account(100);
        AccountThread acountThread1 = new AccountThread(acount, 1000);
        AccountThread acountThread2 = new AccountThread(acount, 0);
        acountThread1.start();
        acountThread2.start();
    }
}
```

运行结果如下：

```
Totle Money: 100.0
Thread-1 withdraw    100.0 successful!0.0
Thread-0 withdraw    100.0 successful!-100.0
```

从运行结果可以看出，总额 100 元，使用两个线程同时取钱，都成功，最后帐户余额为 -100 元，表现为透支，这样破坏了数据的完整性。

从程序可以看出 withdrawal 方法包含了余额判断语句，为什么还会发生数据的一致性被破坏呢？因多线程并发，当执行“`balance = balance – money`”这条语句时，`balance` 的实际值已经不是先前的值。

按照正确的业务逻辑，需要保证在一个取款操作结束时，不能执行另一个取款操作，需要把 withdraw 同步起来，我们可以使用 synchronized 关键字。修改如下：

```
public synchronized void withdraw(double money, int delay)
```

运行修改后的程序,结果如下：

```
Totle Money: 100.0
Thread-1 withdraw  100.0 successful!0.0
Thread-0balance is not enough, withdraw failed!0.0
```

前面我们讲过了原子量的使用，现在修改 balance 为原子量。用原子量的特性实现取款操作的原子性。

把 Account 类修为 AtomicAccount，把 balance 定义为 AtomicLong 类型，然后修改 withdraw 方法，把原来方法的修改语句“balance = balance – money”修改为“**balance.compareAndSet(oldvalue, oldvalue - money)**”，这个方法在执行的时候是原子化的，首先比较所读取的值是否和被修改的值一致，如果一致则执行原子化修改，否则失败。如果帐余额在读取之后，被修改了，则 **compareAndSet** 会返回 FALSE，则余额修改失败，不能完成取款操作。

```
package jdkapidemo.bank;
import java.util.concurrent.atomic.AtomicLong;
public class AtomicAccount {
    AtomicLong balance;
    public AtomicAccount(long money) {
        balance = new AtomicLong(money);
        System.out.println("Totle Money: " + balance);
    }
    public void deposit(long money) {
        balance.addAndGet(money);
    }
    public void withdraw(long money, int delay) {
        long oldvalue = balance.get();
        if (oldvalue >= money) {
            try {
                Thread.sleep(delay);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            if (balance.compareAndSet(oldvalue, oldvalue - money)) {
                System.out.println(Thread.currentThread().getName()
                        + " withdraw  " + money + " successful!" +
balance);
            } else {
                System.out.println(Thread.currentThread().getName()
                        + "thread concurrent, withdraw failed!" +
balance);
```

```
            }
        } else {
            System.out.println(Thread.currentThread().getName()
                    + " balance is not enough,withdraw failed!" +
balance);
        }
    }
    public long get() {
        return balance.get();
    }
}
```

重新定义测试类：

```
package jdkapidemo.bank;
public class AtomicAccountTest extends Thread {
    AtomicAccount account;
    int delay;
    public AtomicAccountTest(AtomicAccount account, int delay) {
        this.account = account;
        this.delay = delay;
    }
    public void run() {
        account.withdraw(100, delay);
    }
    public static void main(String[] args) {
        AtomicAccount account = new AtomicAccount(100);
        AtomicAccountTest accountThread1 = new
AtomicAccountTest(account, 1000);
        AtomicAccountTest accountThread2 = new
AtomicAccountTest(account, 0);
        accountThread1.start();
        accountThread2.start();
    }
}
```

运行结果如下：

```
    Totle Money: 100
    Thread-1 withdraw  100 successful!0
    Thread-0 thread concurrent, withdraw failed!0
```

从运行结果可以看出，两个线程在执行 withdraw 方法时，开始余额比较都是成功的，随后在更新余额是我们使用了 balance.compareAndSet(oldvalue, oldvalue - money)原子方法，这个方法在修改余额值之前还要**比较所读取的值是否和被修改的值一致**，如果一致则修改，

如果不一致则修改失败，返回 false。并且保证在修改的过程是原子性的，不会被中断。

大多数用户都不太可能自己使用原子变量开发无阻塞算法,他们更可能使用 java.util.concurrent 中提供的版本，如 ConcurrentLinkedQueue。但是万一您想知道对比以前 JDK 中的相类似的功能，这些类的性能是如何改进的，可以使用通过原子变量类公开的细粒度、硬件级别的并发原语。

开发人员可以直接将原子变量用作共享计数器、序号生成器和其他独立共享变量的高性能替代，否则必须通过同步保护这些变量。

通过内部公开新的低级协调原语，和提供一组公共原子变量类，现在用 Java 语言开发无等待、无锁定算法首次变为可行。然后， java.util.concurrent 中的类基于这些低级原子变量工具构建，为它们提供比以前执行相似功能的类更显著的可伸缩性优点。虽然您可能永远不会直接使用原子变量，还是应该为它们的存在而欢呼。

3.3 并发集合

我们将探讨集合框架中新的 Queue 接口、这个接口的非并发和并发实现、并发 Map 实现和专用于读操作大大超过写操作这种情况的并发 List 和 Set 实现。

3.3.1 队列 Queue 与 BlockingQueue

java.util 包为集合提供了一个新的基本接口：java.util.Queue。虽然肯定可以在相对应的两端进行添加和删除而将 java.util.List 作为队列对待，但是这个新的 Queue 接口提供了支持添加、删除和检查集合的更多方法。

1) boolean add(Object e)：将指定的元素插入此队列（如果立即可行且不会违反容量限制），在成功时返回 true，如果当前没有可用的空间，则抛出 IllegalStateException。

2) public boolean offer(Object element)：将指定的元素插入此队列（如果立即可行且不会违反容量限制），当使用有容量限制的队列时，此方法通常要优于 add(E)，后者可能无法插入元素，而只是抛出一个异常。

3) public Object remove()：获取并移除此队列的头。

4) public Object poll()；获取并移除此队列的头，如果此队列为空，则返回 null。

5) public Object element()；获取但是不移除此队列的头。此队列为空时将抛出一个异常。

6) public Object peek()；获取但不移除此队列的头；如果此队列为空，则返回 null。

基本上，一个队列就是一个先入先出（FIFO）的数据结构。一些队列有大小限制，因此如果想在一个满的队列中加入一个新项，多出的项就会被拒绝。这时新的 offer 方法就可以起作用了。它不是对调用 add()方法抛出一个 unchecked 异常，而只是得到由 **offer()方法**返回的 false。 remove() 和 poll() 方法都是从队列中删除第一个元素（head）。 remove() 的行为与 Collection 接口的版本相似，但是新的 **poll()方法**在用空集合调用时不是抛出异常，只是返回 null。因此新的方法更适合容易出现异常条件的情况。后两个方法 element() 和 peek()用于在队列的头部查询元素。与 remove() 方法类似，在队列为空时， element() 抛出一个异常，而 peek() 返回 null。

在 JDK 中有两组 Queue 实现：实现了新 BlockingQueue 接口的和没有实现这个接口的。我将首先分析那些没有实现的。

在最简单的情况下，原来有的 java.util.LinkedList 实现已经改造成不仅实现 java.util.List 接口，而且还实现 java.util.Queue 接口。可以将 LinkedList 集合看成这两者中的任何一种。下面的程序将显示把 LinkedList 作为 Queue 的使用方法：

```
package queuedemo;
import java.util.LinkedList;
import java.util.Queue;
public class QueueTest {
    public static void main(String[] args) {
        Queue queue = new LinkedList();
        queue.offer("One");
        queue.offer("Two");
        queue.offer("Three");
        queue.offer("Four");
        System.out.println("Head of queue is: " + queue.poll());
    }
}
```

输出结果为：

```
Head of queue is: One
```

PriorityQueue 和 ConcurrentLinkedQueue 类在 Collection Framework 中加入两个具体集合实现。PriorityQueue 类实质上维护了一个有序列表。加入到 Queue 中的元素根据它们的天然排序（通过其 java.util.Comparable 实现）或者根据传递给构造函数的 java.util.Comparator 实现来定位。将上面程序中的 LinkedList 改变为 PriorityQueue 将会打印出 Four 而不是 One，因为按字母排列，即字符串的天然顺序，Four 是第一个。 ConcurrentLinkedQueue 是

基于链接节点的、线程安全的队列。并发访问不需要同步。因为它在队列的尾部添加元素并从头部删除它们，所以只要不需要知道队列的大小， ConcurrentLinkedQueue 对公共集合的共享访问就可以工作得很好。收集关于队列大小的信息会很慢，需要遍历队列。

```
package queuedemo;
import java.util.PriorityQueue;
import java.util.Queue;
public class PriorityQueueDemo {
    public static void main(String[] args) {
        Queue<String> queue = new PriorityQueue<String>();
        queue.offer("One");
        queue.offer("Two");
        queue.offer("Three");
        queue.offer("Four");
        System.out.println("Head of queue is: " + queue.poll());
    }
}
```

输出结果如下：

```
Head of queue is: Four
```

新的 java.util.concurrent 包可用的具体集合类中加入了 BlockingQueue 接口和五个阻塞队列类。阻塞队列实质上就是一种带有一点扭曲的 FIFO 数据结构，不是立即从队列中添加或者删除元素，线程执行操作被阻塞，直到有空间或者元素可用。 BlockingQueue 接口的 Javadoc 给出了阻塞队列的基本用法，生产者中的 put() 操作会在没有空间可用时阻塞，而消费者的 take() 操作会在队列中没有任何东西时阻塞。

五个队列所提供的各有不同：

ArrayBlockingQueue ： 一个由数组支持的有界队列。

LinkedBlockingQueue ： 一个由链接节点支持的可选有界队列。

PriorityBlockingQueue ： 一个由优先级堆支持的无界优先级队列。

DelayQueue ： 一个由优先级堆支持的、基于时间的调度队列。

SynchronousQueue ： 一个利用 BlockingQueue 接口的简单聚集（rendezvous）机制。

下面以 `ArrayBlockingQueue` 为例写一个程序，表示生产者-消费者问题。生产者向阻塞队列中放入字符，消费者从阻塞队列中移除字符。

```
package queuedemo;
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;
public class BlockingQueueDemo {
```

```
        public static void main(String[] args) {
            BlockingQueue<String> queue = new ArrayBlockingQueue<String>(5);
            Producer p = new Producer(queue);
            Consumer c1 = new Consumer(queue);
            Consumer c2 = new Consumer(queue);
            new Thread(p).start();
            new Thread(c1).start();
            new Thread(c2).start();
        }
    }
    class Producer implements Runnable {
        private final BlockingQueue<String> queue;
        Producer(BlockingQueue<String> q) {
            queue = q;
        }
        public void run() {
            try {
                for (int i = 0; i < 100; i++) {
                    queue.put(produce());
                }
            } catch (InterruptedException ex) {
                ex.printStackTrace();
            }
        }
        String produce() {
            String temp = "" + (char) ('A' + (int) (Math.random() * 26));
            System.out.println("produce " + temp);
            return temp;
        }
    }
    class Consumer implements Runnable {
        private final BlockingQueue<String> queue;
        Consumer(BlockingQueue<String> q) {
            queue = q;
        }
        public void run() {
            try {
                for (int i = 0; i < 100; i++) {
                    consume(queue.take());
                }
            } catch (InterruptedException ex) {
                ex.printStackTrace();
            }
        }
```

```
        void consume(String x) {
            System.out.println("consume " + x);
        }
    }
```

输出结果如下：

```
produce W
produce S
produce D
produce Q
consume S
consume W
consume Q
consume D
produce V
produce J
produce P
produce A
consume V
consume P
produce I
consume J
consume I
produce C
...
```

前两个类 ArrayBlockingQueue 和 LinkedBlockingQueue 几乎相同，只是在后备存储器方面有所不同，LinkedBlockingQueue 并不总是有容量界限。无大小界限的 LinkedBlockingQueue 类在添加元素时永远不会有阻塞队列的等待（至少在其中有 Integer.MAX_VALUE 元素之前不会）。

PriorityBlockingQueue 是具有无界限容量的队列，它利用所包含元素的 Comparable 排序顺序来以逻辑顺序维护元素。可以将它看作 TreeSet 的可能替代物。例如，在队列中加入字符串 One、Two、Three 和 Four 会导致 Four 被第一个取出来。对于没有天然顺序的元素，可以为构造函数提供一个 Comparator 。不过对 PriorityBlockingQueue 有一个技巧。从 iterator() 返回的 Iterator 实例不需要以优先级顺序返回元素。如果必须以优先级顺序遍历所有元素，那么让它们都通过 toArray() 方法并自己对它们排序，像 Arrays.sort(pq.toArray()) 。

新的 DelayQueue 实现可能是其中最有意思（也是最复杂）的一个。加入到队列中的元素必须实现新的 Delayed 接口（只有一个方法 long getDelay(java.util.concurrent.TimeUnit

unit))。因为队列的大小没有界限，使得添加可以立即返回，但是在延迟时间过去之前，不能从队列中取出元素。如果多个元素完成了延迟，那么最早失效/失效时间最长的元素将第一个取出。实际上没有听上去这样复杂。下面的程序演示了这种新的阻塞队列集合的使用：

```
package queuedemo;
import java.util.Random;
import java.util.concurrent.DelayQueue;
import java.util.concurrent.Delayed;
import java.util.concurrent.TimeUnit;
public class DelayQueueDemo {
    static class NanoDelay implements Delayed {
        long trigger;
        NanoDelay(long i) {
            trigger = System.nanoTime() + i;
        }
        public boolean equals(Object other) {
            return ((NanoDelay) other).trigger == trigger;
        }
        public boolean equals(NanoDelay other) {
            return ((NanoDelay) other).trigger == trigger;
        }
        public long getDelay(TimeUnit unit) {
            long n = trigger - System.nanoTime();
            return unit.convert(n, TimeUnit.NANOSECONDS);
        }
        public long getTriggerTime() {
            return trigger;
        }
        public String toString() {
            return String.valueOf(trigger);
        }
        @Override
        public int compareTo(Delayed o) {
            long i = trigger;
            long j = ((NanoDelay) o).trigger;
            if (i < j)
                return -1;
            if (i > j)
                return 1;
            return 0;
        }
    }
    public    static    void    main(String    args[])    throws
```

```
InterruptedException {
          Random random = new Random();
          DelayQueue<NanoDelay> queue = new DelayQueue<NanoDelay>();
          for (int i = 0; i < 5; i++) {
              queue.add(new NanoDelay(random.nextInt(1000)));
          }
          long last = 0;
          for (int i = 0; i < 5; i++) {
              NanoDelay delay = (NanoDelay) (queue.take());
              long tt = delay.getTriggerTime();
              System.out.println("Trigger time: " + tt);
              if (i != 0) {
                  System.out.println("Delta: " + (tt - last));
              }
              last = tt;
          }
      }
  }
```

运行结果如下：

```
Trigger time: 5629057839457
Trigger time: 5629057894502
Delta: 55045
Trigger time: 5629057925948
Delta: 31446
Trigger time: 5629057938107
Delta: 12159
Trigger time: 5629057948783
Delta: 10676
```

这个例子首先是一个内部类 NanoDelay ，它实质上将暂停任意纳秒（nanosecond）数，这里利用了 System 的新 nanoTime() 方法。然后 main() 方法只是将 NanoDelay 对象放到队列中并再次将它们取出来。如果希望队列项做一些其他事情，就需要在 Delayed 对象的实现中加入方法，并在从队列中取出后调用这个新方法。（请随意扩展 NanoDelay 以试验加入其他方法做一些有趣的事情。）显示从队列中取出元素的两次调用之间的时间差。如果时间差是负数，可以视为一个错误，因为永远不会在延迟时间结束时，在一个更早的触发时间从队列中取得项。

SynchronousQueue 类是最简单的。它没有内部容量。它就像线程之间的手递手机制。在队列中加入一个元素的生产者会等待另一个线程的消费者。当这个消费者出现时，这个元素就直接在消费者和生产者之间传递，永远不会加入到阻塞队列中。

3.3.2 使用 ConcurrentMap 实现类

java.util.concurrent.ConcurrentMap 接口和 ConcurrentHashMap 实现类只能在键不存在时将元素加入到 map 中，只有在键存在并映射到特定值时才能从 map 中删除一个元素。主要定义了下面几个方法（K 表示键的类型，V 表示值的类型）：

V putIfAbsent(K key,V value)：如果指定键已经不再与某个值相关联，则将它与给定值关联。

boolean remove(Object key,Object value)：只有目前将键的条目映射到给定值时，才移除该键的条目。

boolean replace(K key,V oldValue,V newValue)：只有目前将键的条目映射到给定值时，才替换该键的条目。

V replace(K key,V value)：只有目前将键的条目映射到某一值时，才替换该键的条目。

putIfAbsent() 方法用于在 map 中进行添加。这个方法以要添加到 ConcurrentMap 中的键的值为参数，就像普通的 put() 方法，但是只有在 map 不包含这个键时，才能将键加入到 map 中。如果 map 已经包含这个键，那么这个键的现有值就会保留。 putIfAbsent() 方法是原子的。等价于下面的代码（除了原子地执行此操作之外）：

```
if (!map.containsKey(key))
      return map.put(key, value);
  else
    return map.get(key);
```

像 putIfAbsent() 方法一样，重载后的 remove() 方法有两个参数：键和值。在调用时，只有当键映射到指定的值时才从 map 中删除这个键。如果不匹配，那么就不删除这个键，并返回 false。如果值匹配键的当前映射内容，那么就删除这个键，**这个方法是原子性的**。这种操作的等价源代码（除了原子地执行此操作之外）：

```
if (map.containsKey(key) && map.get(key).equals(value)) {
      map.remove(key);
      return true;
} else return false;
```

总之，ConcurrentMap 中定义的方法是原子性的。

3.3.3 CopyOnWriteArrayList 和 CopyOnWriteArraySet

这两个集合对对 copy-on-write 模式作了比较好的支持。这个模式说明了，为了维护对象的一致性快照，要依靠不可变性（immutability）来消除在协调读取不同的但是相关的属性时需要的同步。

对于集合，这意味着如果有大量的读（即 get() ）和迭代，不必进行同步操作以照顾偶尔的写（即 add() ）调用。对于新的 CopyOnWriteArrayList 和 CopyOnWriteArraySet 类，所有可变的（mutable）操作都首先取得后台数组的副本，对副本进行更改，然后替换副本。这种做法保证了在遍历自身可更改的集合时，永远不会抛出 ConcurrentModificationException。遍历集合会用原来的集合完成，而在以后的操作中使用更新后的集合。

这些新的集合最适合于读操作通常大大超过写操作的情况。集合的使用与它们的非 copy-on-write 替代物完全一样。只是创建集合并在其中加入或者删除元素。即使对象加入到了集合中，原来的 Iterator 也可以进行，继续遍历原来集合中的项。

下面是使用 copy-on-write 集合和一般类型集合进行遍历的例子：

```
package copyonwrite;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.ConcurrentModificationException;
import java.util.Iterator;
import java.util.List;
import java.util.concurrent.CopyOnWriteArrayList;
public class CopyOnWriteDemo {
    @SuppressWarnings("unchecked")
    public static void main(String args[]) {
        String[] ss = { "aa", "bb", "cc" };
        List list1 = new CopyOnWriteArrayList(Arrays.asList(ss));
        List list2 = new ArrayList(Arrays.asList(ss));
        Iterator itor1 = list1.iterator();
        Iterator itor2 = list2.iterator();
        list1.add("New");
        list2.add("New");
        try {
            printAll(itor1);
        } catch (ConcurrentModificationException e) {
```

```
            System.err.println("Shouldn't get here");
        }
        try {
            printAll(itor2);
        } catch (ConcurrentModificationException e) {
            System.err
                    .println("Will get
here.ConcurrentModificationException occurs!");
        }
    }
    @SuppressWarnings("unchecked")
    private static void printAll(Iterator itor) {
        while (itor.hasNext()) {
            System.out.println(itor.next());
        }
    }
}
```

运行结果如下：

```
Will get here.ConcurrentModificationException occurs!
aa
bb
cc
```

这个示例程序创建 CopyOnWriteArrayList 和 ArrayList 这两个实例。在得到每一个实例的 Iterator 后，分别在其中加入一个元素。当 ArrayList 迭代因一个 ConcurrentModificationException 问题而立即停止时，CopyOnWriteArrayList 迭代可以继续，不会抛出异常，因为原来的集合是在得到 iterator 之后改变的。如果这种行为（比如通知原来一组事件监听器中的所有元素）是您需要的，那么最好使用 copy-on-write 集合。如果不使用的话，就还用原来的，并保证在出现异常时对它进行处理。

3.4 同步器

3.4.1 Semaphore

类 java.util.concurrent.Semaphore 提供了一个计数信号量，从概念上讲，信号量维护了一个许可集。如有必要，在许可可用前会阻塞每一个 acquire()，然后再获取该许可。每个 release() 添加一个许可，从而可能释放一个正在阻塞的获取者。但是，不使用实际的许可对象，

Semaphore 只对可用许可的号码进行计数，并采取相应的行动。

Semaphore 通常用于限制可以访问某些资源（物理或逻辑的）的线程数目。一般操作系统的进程调度中使用了 PV 原语，需要设置一个信号量表示可用资源的数量，P 原语就相当于 acquire()，V 原语就相当于 release()。

例如，下面的类使用信号量控制对内容池的访问，内容池的大小作为 Semaphore 的构造参数传递初始化*许可*的数目，每个线程获取数据之前必须获得许可，这样就限制了访问内容池的线程数目：

```
package synchronizer;
import java.util.concurrent.Semaphore;
class PoolSemaphoreDemo {
    private static final int MAX_AVAILABLE = 5;
    private final Semaphore available = new Semaphore(MAX_AVAILABLE, true);
    public static void main(String args[]) {
        final PoolSemaphoreDemo pool = new PoolSemaphoreDemo();
        Runnable runner = new Runnable() {
            @Override
            public void run() {
                try {
                    Object o;
                    o = pool.getItem();
                    System.out.println(Thread.currentThread().getName()
                            + " acquire " + o);
                    Thread.sleep(1000);
                    pool.putItem(o);
                    System.out.println(Thread.currentThread().getName()
                            + " release " + o);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        };
        for (int i = 0; i < 10; i++)// 构造 10 个线程
        {
            Thread t = new Thread(runner, "t" + i);
            t.start();
        }
    }
    //获取数据，需要得到许可
    public Object getItem() throws InterruptedException {
        available.acquire();
        return getNextAvailableItem();
```

```
        }
        //放回数据，释放许可
        public void putItem(Object x) {
            if (markAsUnused(x))
                available.release();
        }
        protected Object[] items = { "AAA", "BBB", "CCC", "DDD", "EEE" };
        protected boolean[] used = new boolean[MAX_AVAILABLE];
        protected synchronized Object getNextAvailableItem() {
            for (int i = 0; i < MAX_AVAILABLE; ++i) {
                if (!used[i]) {
                    used[i] = true;
                    return items[i];
                }
            }
            return null;
        }
        protected synchronized boolean markAsUnused(Object item) {
            for (int i = 0; i < MAX_AVAILABLE; ++i) {
                if (item == items[i]) {
                    if (used[i]) {
                        used[i] = false;
                        return true;
                    } else
                        return false;
                }
            }
            return false;
        }
    }
```

运行结果如下：

```
t0 acquire AAA
t1 acquire BBB
t4 acquire EEE
t5 acquire DDD
t2 acquire CCC
t0 release AAA
t3 acquire AAA
t8 acquire BBB
t1 release BBB
t5 release DDD
t6 acquire DDD
t4 release EEE
t7 acquire EEE
```

```
t2 release CCC
t9 acquire CCC
t3 release AAA
t8 release BBB
t6 release DDD
t7 release EEE
t9 release CCC
```

获得一项前，每个线程必须从信号量获取许可，从而保证可以使用该项。该线程结束后，将项返回到池中并将许可返回到该信号量，从而允许其他线程获取该项。

从程序的运行结果，我们可以看出，池的大小是 5，先前有 5 个线程可以使用池中的内容，后面的线程调用 acquire()获得池的许可时，被阻塞。直到前面的线程释放已经获得的许可，后面的线程才可以使用池中的内容。

注意，调用 acquire()时无法保持同步锁，因为这会阻止将数据项返回到池中。信号量封装所需的同步，以限制对池的访问，这同维持该池本身一致性所需的同步是分开的。

将信号量初始化为 1，使得它在使用时最多只有一个可用的许可，从而可用作一个相互排斥的锁。这通常也称为二进制信号量，因为它只能有两种状态：一个可用的许可，或零个可用的许可。按此方式使用时，二进制信号量具有某种属性（与很多 Lock 实现不同），即可以由线程释放“锁”，而不是由所有者（因为信号量没有所有权的概念）。在某些专门的上下文（如死锁恢复）中这会很有用。

3.4.2 Barrier

在实际应用中，有时候需要多个线程同时工作以完成同一件事情，而且在完成过程中，往往会等所有线程都到达某一个阶段后再统一执行。

比如有几个旅行团需要途经深圳、广州、最后到达重庆。旅行团中有自驾游的、有徒步的、有乘坐旅游大巴的；这些旅行团同时出发，并且每到一个目的地，都要等待其他旅行团到达此地后再同时出发，直到都到达终点站重庆。

这时候 **java.util.concurrent.CyclicBarrier** 就可以派上用场。一个同步辅助类，它允许一组线程互相等待，直到到达某个公共屏障点 (common barrier point)。在涉及一组固定大小的线程的程序中，这些线程必须不时地互相等待，此时 CyclicBarrier 很有用。

因为该 barrier 在释放等待线程后可以重用，所以称它为循环的 barrier。CyclicBarrier 最重要的属性就是参与者个数，另外最要方法是 await()。当所有线程都调用了 await()后，就

表示这些线程都可以继续执行，否则就会等待。

CyclicBarrier 支持一个可选的 Runnable 命令，在一组线程中的最后一个线程到达之后（但在释放所有线程之前），该命令只在每个屏障点运行一次。若在继续所有参与线程之前更新共享状态，此屏障操作有用。

上面提到的旅行团问题，可以用下面的程序实现，在程序中，某一个旅行团先到达某一个中转站后，调用 await()方法等待其他旅行团，都到齐后，执行 Runnable。

```
    package synchronizer;
    import java.text.SimpleDateFormat;
    import java.util.Date;
    import java.util.concurrent.BrokenBarrierException;
    import java.util.concurrent.CyclicBarrier;
    import java.util.concurrent.ExecutorService;
    import java.util.concurrent.Executors;
    public class CyclicBarrierDemo {
        // 徒步需要的时间: Shenzhen, Guangzhou, Chongqing
        private static int[] timeForWalk = { 5, 8, 15 };
        // 自驾游
        private static int[] timeForSelf = { 1, 3, 4 };
        // 旅游大巴
        private static int[] timeForBus = { 2, 4, 6 };
        static String nowTime() {//时间格式化
            SimpleDateFormat sdf = new SimpleDateFormat("HH:mm:ss");
            return sdf.format(new Date()) + ": ";
        }
        static class Tour implements Runnable {
            private int[] timeForUse;
            private CyclicBarrier barrier;
            private String tourName;
            public Tour(CyclicBarrier barrier, String tourName, int[] timeForUse)
{
                this.timeForUse = timeForUse;
                this.tourName = tourName;
                this.barrier = barrier;
            }
            public void run() {
                try {
                    Thread.sleep(timeForUse[0] * 1000);
                    System.out.println(nowTime()  +  tourName  +  "  Reached
Shenzhen");
                    barrier.await();//到达中转站后等待其他旅行团
                    Thread.sleep(timeForUse[1] * 1000);
```

```
                    System.out.println(nowTime() + tourName + " Reached
Guangzhou");
                    barrier.await();//到达中转站后等待其他旅行团
                    Thread.sleep(timeForUse[2] * 1000);
                    System.out.println(nowTime() + tourName + " Reached
Chongqing");
                    barrier.await();//到达中转站后等待其他旅行团
                } catch (InterruptedException e) {
                } catch (BrokenBarrierException e) {
                }
            }
        }
        public static void main(String[] args) {
            // 三个旅行团都到到达某一个站点后，执行下面的操作，表示都到齐了。
            Runnable runner = new Runnable() {
                @Override
                public void run() {
                    System.out.println("we all are here.");
                }
            };
            CyclicBarrier barrier = new CyclicBarrier(3, runner);
            //使用线程池
            ExecutorService exec = Executors.newFixedThreadPool(3);
            exec.submit(new Tour(barrier, "WalkTour", timeForWalk));
            exec.submit(new Tour(barrier, "SelfTour", timeForSelf));
            exec.submit(new Tour(barrier, "BusTour", timeForBus));
            exec.shutdown();
        }
    }
```

运行结果如下：

```
17:13:18: SelfTour Reached Shenzhen
17:13:19: BusTour Reached Shenzhen
17:13:22: WalkTour Reached Shenzhen
we all are here.
17:13:25: SelfTour Reached Guangzhou
17:13:26: BusTour Reached Guangzhou
17:13:30: WalkTour Reached Guangzhou
we all are here.
17:13:34: SelfTour Reached Chongqing
17:13:36: BusTour Reached Chongqing
17:13:45: WalkTour Reached Chongqing
we all are here.
```

3.4.3 CountDownLatch

类 **java.util.concurrent.CountDownLatch** 是一个同步辅助类，在完成一组正在其他线程中执行的操作之前，它允许一个或多个线程一直等待。

用给定的数字作为计数器初始化 CountDownLatch。一个线程调用 await()方法后，在当前计数到达零之前，会一直受阻塞。其他线程调用 countDown() 方法，会使计数器递减，所以，计数器的值为 0 后，会释放所有等待的线程。其他后续的 await 调用都将立即返回。这种现象只出现一次，因为计数无法被重置。如果需要重置计数，请考虑使用 CyclicBarrier。

CountDownLatch 作为一个通用同步工具，有很多用途。使用“1”初始化的 CountDownLatch 用作一个简单的开/关锁存器，或入口：在通过调用 countDown() 的线程打开入口前，所有调用 await 的线程都一直在入口处等待。用 N 初始化的 CountDownLatch 可以使一个线程在 N 个线程完成某项操作之前一直等待，或者使其在某项操作完成 N 次之前一直等待。

下面给出了两个类，其中一组 worker 线程使用了两个倒计数 **CountDownLatch**：

第一个类是一个启动信号，在 driver 为继续执行 worker 做好准备之前，它会阻止所有的 worker 继续执行。

第二个类是一个完成信号，它允许 driver 在完成所有 worker 之前一直等待。

```
package synchronizer;
import java.util.concurrent.CountDownLatch;
public class LatchDriverDemo {
    public static final int N = 5;
    public static void main(String[] args) throws InterruptedException
{
        // 用于向工作线程发送启动信号
        CountDownLatch startSignal = new CountDownLatch(1);
        // 用于等待工作线程的结束信号
        CountDownLatch doneSignal = new CountDownLatch(N);
        for (int i = 0; i < N; ++i)
            // 创建启动线程
            new Thread(new LatchWorker(startSignal, doneSignal), "t"
+ i)
                    .start();
        // 得到线程开始工作的时间
        long start = System.nanoTime();
        // 主线程，递减开始计数器，让所有线程开始工作
        startSignal.countDown();
```

```
        // 主线程阻塞，等待所有线程完成
        doneSignal.await();
        long end = System.nanoTime();
        System.out.println("all worker take time（ms）:" + (end - start)
                / 1000000);
    }
}
class LatchWorker implements Runnable {
    // 用于等待启动信号
    private final CountDownLatch startSignal;
    // 用于发送结束信号
    private final CountDownLatch doneSignal;
    LatchWorker(CountDownLatch startSignal, CountDownLatch
doneSignal) {
        this.startSignal = startSignal;
        this.doneSignal = doneSignal;
    }
    public void run() {
        try {
            startSignal.await();// 阻塞，等待开始新信号
            doWork();
            doneSignal.countDown();// 发送完成信号
        } catch (InterruptedException ex) {
        }
    }
    void doWork() {
        System.out.println(Thread.currentThread().getName() + " is
working...");
        int sum = 0;
        for (int i = 0; i < 10000000; i++) {
            sum += i;
        }
    }
}
```

运行结果如下：

```
t0 is working...
t4 is working...
t1 is working...
t3 is working...
t2 is working...
all worker take time（ms）:65
```

另一种典型用法是，将一个问题分成 N 个部分，用执行每个部分并让 **CountDownLatch**

倒计数的 Runnable 来描述每个部分，然后将所有 Runnable 加入到 Executor 队列。当所有的子部分完成后，协调线程就能够通过 await。（当线程必须用这种方法反复倒计数时，可改为使用 CyclicBarrier。）

这个做法请大家在实验中完成。

3.4.4 Exchanger

类 java.util.concurrent.Exchanger 提供了一个同步点，在这个同步点，一对线程可以交换数据。每个线程通过 exchange()方法的入口提供数据给他的伙伴线程，并接收他的伙伴线程提供的数据，并返回。

当在运行不对成的活动时很有用，比如当一个线程填充了 buffer，另一个线程从 buffer 中消费数据；这些线程可以用 Exchanger 来交换数据。当两个线程通过 Exchanger 交互了对象，这个交换对于两个线程来说都是安全的。

下面给出了两个线程：一个生产者生产数据，通过 Exchanger 与另外一个消费者交换数据。

```
package synchronizer;
import java.util.ArrayList;
import java.util.Date;
import java.util.Iterator;
import java.util.List;
import java.util.concurrent.Exchanger;
public class ExchangerDemo {
    private static final Exchanger ex = new Exchanger();
    class DataProducer implements Runnable {
        private List list = new ArrayList();
        public void run() {
            for (int i = 0; i < 5; i++) {
                System.out.println("生产了一个数据，耗时1秒");
                list.add(new Date());
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            try {
```

```
                // 将数据准备用于交换，并返回消费者的数据
                list = (List) ex.exchange(list);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            for (Iterator iterator = list.iterator();
iterator.hasNext();) {
                System.out.println("Producer " + iterator.next());
            }
        }
    }
    class DataConsumer implements Runnable {
        private List list = new ArrayList();
        public void run() {
            for (int i = 0; i < 5; i++) {
                // 消费者产生数据，后面交换的时候给生产者
                list.add("这是一个收条。");
            }
            try {
                // 进行交换数据，返回生产者的数据
                list = (List) ex.exchange(list);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            for (Iterator iterator = list.iterator();
iterator.hasNext();) {
                Date d = (Date) iterator.next();
                System.out.println("consumer:" + d);
            }
        }
    }
    public static void main(String args[]) {
        ExchangerDemo et = new ExchangerDemo();
        new Thread(et.new DataProducer()).start();
        new Thread(et.new DataConsumer()).start();
    }
}
```

运行结果如下：

```
生产了一个数据，耗时 1 秒
生产了一个数据，耗时 1 秒
生产了一个数据，耗时 1 秒
生产了一个数据，耗时 1 秒
生产了一个数据，耗时 1 秒
Producer 这是一个收条。
```

```
Producer 这是一个收条。
Producer 这是一个收条。
Producer 这是一个收条。
Producer 这是一个收条。
consumer:Wed Feb 25 12:08:10 CST 2009
consumer:Wed Feb 25 12:08:11 CST 2009
consumer:Wed Feb 25 12:08:12 CST 2009
consumer:Wed Feb 25 12:08:13 CST 2009
consumer:Wed Feb 25 12:08:14 CST 2009
```

从运行结果可以看出，使用 Exchanger 完成了两个线程的数据交换。

3.4.5 Future 和 FutureTask

接口 public interface Future<V> 表示异步计算的结果。它提供了检查计算是否完成的方法，以等待计算的完成，并获取计算的结果。计算完成后只能使用 get() 方法来获取结果，如有必要，计算完成前可以**阻塞**此方法。取消则由 cancel 方法来执行。还提供了其他方法，以确定任务是正常完成还是被取消了。一旦计算完成，就不能再取消计算。如果为了可取消性而使用 Future 但又不提供可用的结果，则可以声明 Future<?> 形式类型、并返回 null 作为底层任务的结果。

Future 主要定义了 5 个方法：

1) boolean cancel(boolean mayInterruptIfRunning)：试图取消对此任务的执行。如果任务已完成、或已取消，或者由于某些其他原因而无法取消，则此尝试将失败。当调用 cancel 时，如果调用成功，而此任务尚未启动，则此任务将永不运行。如果任务已经启动，则 mayInterruptIfRunning 参数确定是否应该以试图停止任务的方式来中断执行此任务的线程。

2) boolean isCancelled()：如果在任务正常完成前将其取消，则返回 true。

3) boolean isDone()：如果任务已完成，则返回 true。 可能由于正常终止、异常或取消而完成，在所有这些情况中，此方法都将返回 true。

4) V get() throws InterruptedException,ExecutionException：如有必要，等待计算完成，然后获取其结果。

5) V get(long timeout, TimeUnit unit) throws InterruptedException, ExecutionException, TimeoutException：如有必要，最多等待为使计算完成所给定的时间之后，获取其结果（如果结果可用）。

FutureTask 类是 Future 的一个实现，并实现了Runnable，所以可通过Executor(线程池）来执行。也可传递给Thread对象执行。

如果在主线程中需要执行比较耗时的操作时，但又不想阻塞主线程时，可以把这些作业交给 Future 对象在后台完成，当主线程将来需要时，就可以通过 Future 对象获得后台作业的计算结果或者执行状态。

下面的例子模拟一个会计算账的过程，主线程已经获得其他帐户的总额了，为了不让主线程等待 PrivateAccount 类的计算结果的返回而启用新的线程去处理，并使用 FutureTask 对象来监控，这样，主线程还可以继续做其他事情，最后需要计算总额的时候再尝试去获得 PrivateAccount 的信息。

```
import java.util.Random;
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.FutureTask;
public class FutureTaskDemo {
    public static void main(String[] args) {
        // 初始化一个Callable对象和FutureTask对象;
        Callable pAccount = new PrivateAccount();
        FutureTask futureTask = new FutureTask(pAccount);
        // 使用FutureTask创建一个线程
        Thread pAccountThread = new Thread(futureTask);
        System.out.println("future task starts at " +
System.nanoTime());
        // 启动线程
        pAccountThread.start();
        // 主线程执行自己的任务
        System.out.println("main thread doing something else here. ");
        // 从其他帐户获取总金额
        int totalMoney = new Random().nextInt(100000);
        System.out.println(" You have  " + totalMoney
                + "  in your other Accounts. ");
        System.out.println(" Waiting for data from Private Account ");
        // 测试后台的就计算线程是否完成，如果未完成，等待
        while (!futureTask.isDone()) {
            try {
                Thread.sleep(5);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        System.out.println("future task ends at " +
```

```
System.nanoTime());
        Integer privataAccountMoney = null;
        // 如果后台的FutureTask计算完成，则返回计算结果
        try {
            privataAccountMoney = (Integer) futureTask.get();
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }
        System.out.println(" The total moeny you have is  "
                + (totalMoney + privataAccountMoney.intValue()));
    }
}
// 创建一个Callable类，模拟计算一个私有帐户中的金额
class PrivateAccount implements Callable {
    Integer totalMoney;
    @Override
    public Integer call() throws Exception {
        // 为了延长计算时间，这里暂停几秒
        Thread.sleep(5000);
        totalMoney = new Integer(new Random().nextInt(10000));
        System.out.println(" You have  " + totalMoney
                + "  in your private Account. ");
        return totalMoney;
    }
}
```

运行结果如下：

```
future task starts at 8081043630405
main thread doing something else here.
 You have   10802   in your other Accounts.
 Waiting for data from Private Account
 You have   6771   in your private Account.
future task ends at 8086046077923
 The total moeny you have is   17573
```

从运行结果可以看出，使用 FutureTask 后，主线程可以获得异步线程的计算结果了。

3.5 显示锁

3.5.1 ReentrantLock

java.util.concurrent.lock 中的类 ReentrantLock 被作为 Java 语言中 synchronized 功能

的替代，它具有相同的内存语义、相同的锁定，但在争用条件下却有更好的性能，此外，它还有 synchronized 没有提供的其他特性。

Java 是第一个直接把跨平台线程模型和正规的内存模型集成到语言中的主流语言。核心类库包含一个 Thread 类，可以用它来构建、启动和操纵线程，Java 语言包括了跨线程传达并发性约束的构造 —— synchronized 和 volatile 。在简化与平台无关的并发类的开发的同时，它决没有使并发类的编写工作变得更繁琐，只是使它变得更容易了。

把代码块声明为 synchronized，有两个重要后果，通常是指该代码具有原子性（atomicity）和可见性（visibility）。原子性意味着一次只能有一个线程执行一个指定监控对象（lock）保护的代码，从而防止多个线程在更新共享状态时相互冲突。可见性则更为微妙；它要对付内存缓存和编译器优化的各种反常行为。一般来说，线程以某种不必让其他线程立即可以看到的方式（不管这些线程在寄存器中、在处理器特定的缓存中，还是通过指令重排或者其他编译器优化）不受缓存变量值的约束，但是如果开发人员使用了同步，如下面的代码所示，那么运行库将确保某一线程对变量所做的更新先于对现有 synchronized 块所进行的更新，当进入由同一监控器（lock）保护的另一个 synchronized 块时，将立刻可以看到这些对变量所做的更新。类似的规则也存在于 volatile 变量上。

使用 synchronized 进行同步的典型方法如下：

```
synchronized (lockObject) {
   //更新对象状态
}
```

实现同步操作需要考虑安全更新多个共享变量所需的一切，不能有争用条件，不能破坏数据（假设同步的边界位置正确），而且要保证正确同步的其他线程可以看到这些变量的最新值。通过定义一个清晰的、跨平台的内存模型，通过遵守下面这个简单规则，构建“一次编写，随处运行”的并发类是有可能的：不论什么时候，只要您将编写的变量接下来可能被另一个线程读取，或者您将读取的变量最后是被另一个线程写入的，那么您必须进行同步。

Synchronized 虽然能够实现同步，但是他有一些限制，比如：它无法中断一个正在等候获得锁的线程，也无法通过投票得到锁，如果不想等下去，也就没法得到锁。

3.5.1.1ReentrantLock 的特性

java.util.concurrent.lock 中的 Lock 框架是锁定的一个抽象，它允许把锁定的实现作为

Java 类，而不是作为语言的特性来实现。这就为 Lock 的多种实现留下了空间，各种实现可能有不同的调度算法、性能特性或者锁定语义。 ReentrantLock 类实现了 Lock ，它拥有与 synchronized 相同的并发性和内存语义，但是添加了类似锁投票、定时锁等候和可中断锁等候的一些特性。此外，它还提供了在激烈争用情况下更佳的性能。（换句话说，当许多线程都想访问共享资源时，JVM 可以花更少的时候来调度线程，把更多时间用在执行线程上。）

ReentrantLock（可重入锁）有一个与锁相关的获取计数器，如果拥有锁的某个线程再次得到锁，那么获取计数器就加 1，然后锁需要被释放两次才能获得真正释放。这模仿了 synchronized 的语义；如果线程进入由线程已经拥有的监控器保护的 synchronized 块，就允许线程继续进行，当线程退出第二个（或者后续）synchronized 块的时候，不释放锁，只有线程退出它进入的监控器保护的第一个 synchronized 块时，才释放锁。

ReentrantLock 锁的使用方法如下：

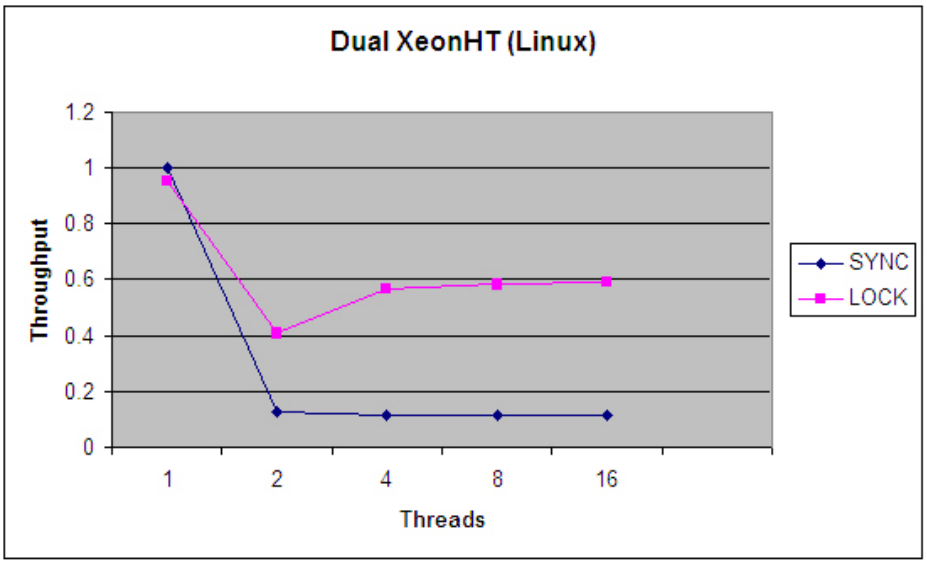
```
Lock lock = new ReentrantLock();
lock.lock();
try {
   // 更新对象状态
}
finally {
   lock.unlock();
}
```

Lock 和 synchronized 有一点明显的区别 —— lock 必须在 finally 块中释放。否则，如果受保护的代码将抛出异常，锁就有可能永远得不到释放。这一点区别看起来可能没什么，但是实际上，它极为重要。忘记在 finally 块中释放锁，可能会在程序中留下一个定时炸弹，当有一天炸弹爆炸时，您要花费很大力气才有找到源头在哪。而使用 synchronized 同步，JVM 将确保锁会获得自动释放。

除此之外，与目前的 synchronized 实现相比，争用下的 ReentrantLock 实现更具可伸缩性。

国外学者 Tim Peierls 用一个简单的线性全等伪随机数生成器（PRNG）构建了一个简单的评测，用它来测量 synchronized 和 Lock 之间相对的可伸缩性。这个示例很好，因为每次调用 nextRandom() 时，PRNG 都确实在做一些工作，所以这个基准程序实际上是在测量一个合理的、真实的 synchronized 和 Lock 应用程序，而不是测试纯粹纸上谈兵或者什么也不做的代码（就像许多所谓的基准程序一样。）

在这个基准程序中，有一个 PseudoRandom 的接口，它只有一个方法 nextRandom(int bound) 。该接口与 java.util.Random 类的功能非常类似。因为在生成下一个随机数时，PRNG 用最新生成的数字作为输入，而且把最后生成的数字作为一个实例变量来维护，其重点在于让更新这个状态的代码段不被其他线程抢占，所以要用某种形式的锁来确保这一点。（ java.util.Random 类也可以做到这点。）为 PseudoRandom 构建了两个实现；一个使用 synchronized，另一个使用 java.util.concurrent.ReentrantLock 。驱动程序生成了大量线程，每个线程都疯狂地争夺时间片，然后计算不同版本每秒能执行多少轮。下面的图总结了不同线程数量的结果。这个评测并不完美，而且只在两个系统上运行了（一个是双 Xeon 运行超线程 Linux，另一个是单处理器 Windows 系统），但是，应当足以表现 synchronized 与 ReentrantLock 相比所具有的伸缩性优势了。



根类 Object 包含某些特殊的方法，如：wait() 、 notify() 和 notifyAll() 在线程之间进行通信。这些是高级的并发性特性，许多开发人员从来没有用过它们 —— 这可能是件好事，因为它们相当微妙，很容易使用不当。幸运的是，随着 JDK 5.0 中引入 java.util.concurrent ，开发人员几乎更加没有什么地方需要使用这些方法了。

通知与锁定之间有一个交互 —— 为了在对象上 wait 或 notify ，您必须持有该对象的锁。就像 Lock 是同步的概括一样， Lock 框架包含了对 wait 和 notify 的概括，这个概括叫做条件（Condition）。Lock 对象则充当绑定到这个锁的条件变量的工厂对象，与标准的 wait 和 notify 方法不同，对于指定的 Lock ，可以有不止一个条件变量与它关联。这样就简化了许多并发算法的开发。例如， 条件（Condition） 的 Javadoc 显示了一个有界缓冲区实现的示例，该示例使用了两个条件变量，“not full”和“not empty”，它比每个 lock

只用一个 wait 设置的实现方式可读性要好一些(而且更有效)。Condition 的方法与 wait 、notify 和 notifyAll 方法类似，分别命名为 await 、 signal 和 signalAll ，因为它们不能覆盖 Object 上的对应方法。

ReentrantLock 构造器的一个参数是 boolean 值，它允许选择想要一个公平（fair）锁，还是一个不公平（unfair）锁。公平锁使线程按照请求锁的顺序依次获得锁；而不公平锁则允许讨价还价，在这种情况下，线程有时可以比先请求锁的其他线程先得到锁。

为什么我们不让所有的锁都公平呢？毕竟，公平是好事，不公平是不好的，不是吗？在现实中，公平保证了锁是非常健壮的锁，有很大的性能成本。要确保公平所需要的记帐（bookkeeping）和同步，就意味着被争夺的公平锁要比不公平锁的吞吐率更低。作为默认设置，应当把公平设置为 false ，除非公平对您的算法至关重要，需要严格按照线程排队的顺序对其进行服务。

那么同步又如何呢？内置的监控器锁是公平的吗？答案令许多人感到大吃一惊，它们是不公平的，而且永远都是不公平的。但是没有人抱怨过线程饥渴，因为 JVM 保证了所有线程最终都会得到它们所等候的锁。确保统计上的公平性，对多数情况来说，这就已经足够了，而这花费的成本则要比绝对的公平保证的低得多。所以，默认情况下 ReentrantLock 是“不公平”的，这一事实只是把同步中一直不公平的东西表面化而已。如果您在同步的时候并不介意这一点，那么在 ReentrantLock 时也不必为它担心。

虽然 ReentrantLock 是个非常动人的实现，相对 synchronized 来说，它有一些重要的优势，但是急于把 synchronized 视若敝屣，绝对是个严重的错误。 java.util.concurrent.lock 中的锁定类是用于高级用户和高级情况的工具 。一般来说，除非您对 Lock 的某个高级特性有明确的需要，或者有明确的证据（而不是仅仅是怀疑）表明在特定情况下，同步已经成为可伸缩性的瓶颈，否则还是应当继续使用 synchronized。

为什么我在一个显然“更好的”实现的使用上主张保守呢？因为对于 java.util.concurrent.lock 中的锁定类来说，synchronized 仍然有一些优势。比如，在使用 显示锁的时候，可能忘记用 finally 块释放锁，这对程序非常有害。您的程序能够通过测试，但会在实际工作中出现死锁，那时会很难指出原因（这也是为什么根本不让初级开发人员使用 Lock 的一个好理由。）但在退出 synchronized 块时，JVM 会为您做这件事。

另一个原因是因为，当 JVM 用 synchronized 管理锁定请求和释放时，JVM 在生成线程转储时能够包括锁定信息。这些对调试非常有价值，因为它们能标识死锁或者其他异常行为的来源。 Lock 类只是普通的类，JVM 不知道具体哪个线程拥有 Lock 对象。而且，几

乎每个开发人员都熟悉 synchronized，它可以在 JVM 的所有版本中工作。在 JDK 5.0 成为标准（从现在开始可能需要两年）之前，使用 Lock 类将意味着要利用的特性不是每个 JVM 都有的，而且不是每个开发人员都熟悉的。

既然如此，我们什么时候才应该使用 ReentrantLock 呢？答案非常简单 —— 在确实需要一些 synchronized 所没有的特性的时候，比如时间锁等候、可中断锁等候、无块结构锁、多个条件变量或者锁投票。 ReentrantLock 还具有可伸缩性的好处，应当在高度争用的情况下使用它，但是请记住，大多数 synchronized 块几乎从来没有出现过争用，所以可以把高度争用放在一边。我建议用 synchronized 开发，直到确实证明 synchronized 不合适，而不要仅仅是假设如果使用 ReentrantLock “性能会更好”。请记住，这些是供高级用户使用的高级工具。（而且，真正的高级用户喜欢选择能够找到的最简单工具，直到他们认为简单的工具不适用为止。）。一如既往，首先要把事情做好，然后再考虑是不是有必要做得更快。

3.5.1.2ReentrantLock 性能测试

下面的例子是一个计数器，启动 N 个线程对计数器 Counter 进行递增操作，显然，这个递增操作需要同步以保证原子性，采用不同的锁来实现同步，然后查看结果。实验环境是 Windows XP with SP2，双核酷睿处理器。通过查看输出结果可以比较一下不同锁的性能。

计数器接口：

```
package locks;
public interface Counter {
    public long getValue();
    public void increment();
}
```

内部锁：

```
package locks;
public class SynchronizedBenchmarkDemo implements Counter {
    private long count = 0;
    public long getValue() {
        return count;
    }
    public synchronized void increment() {
        count++;
    }
}
```

不公平重入锁

```java
package locks;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
public class ReentrantLockUnfairBeanchmarkDemo implements Counter {
    private volatile long count = 0;
    private Lock lock;
    public ReentrantLockUnfairBeanchmarkDemo() {
        // 使用非公平锁，true就是公平锁
        lock = new ReentrantLock(false);
    }
    public long getValue() {
        return count;
    }
    public void increment() {
        lock.lock();
        try {
            count++;
        } finally {
            lock.unlock();
        }
    }
}
```

公平重入锁

```
package locks;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
public class ReentrantLockFairBeanchmarkDemo implements Counter {
    private volatile long count = 0;
    private Lock lock;
    public ReentrantLockFairBeanchmarkDemo() {
        // true 就是公平锁
        lock = new ReentrantLock(true);
    }
    public long getValue() {
        return count;
    }
    public void increment() {
        lock.lock();
        try {
            count++;
        } finally {
            lock.unlock();
        }
    }
```

```
    }
```

总测试程序

```
package locks;
import java.util.concurrent.CyclicBarrier;
public class BenchmarkTest {
    private Counter counter;
    private CyclicBarrier barrier;
    private int threadNum;
    private int loopNum;
    private String testName;
    public BenchmarkTest(Counter counter, int threadNum, int loopNum,
            String testName) {
        this.counter = counter;
        barrier = new CyclicBarrier(threadNum + 1); // 关卡计数=线程数
+1
        this.threadNum = threadNum;
        this.loopNum = loopNum;
        this.testName = testName;
    }
    public static void main(String args[]) throws Exception {
        int threadNum = 5000;
        int loopNum = 100;
        new BenchmarkTest(new SynchronizedBenchmarkDemo(),
threadNum, loopNum,
                "内部锁").test();
        new BenchmarkTest(new ReentrantLockUnfairBeanchmarkDemo(),
threadNum,
                loopNum, "不公平重入锁").test();
        new BenchmarkTest(new ReentrantLockFairBeanchmarkDemo(),
threadNum,
                loopNum, "公平重入锁").test();
    }
    public void test() throws Exception {
        try {
            for (int i = 0; i < threadNum; i++) {
                new TestThread(counter, loopNum).start();
            }
            long start = System.currentTimeMillis();
            barrier.await(); // 等待所有任务线程创建,然后通过关卡，统一执行
所有线程
            barrier.await(); // 等待所有任务计算完成
            long end = System.currentTimeMillis();
            System.out.println(this.testName + " count value:"
                    + counter.getValue());
```

```
            System.out.println(this.testName + " 花费时间:" + (end -
start) + "毫秒");
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
    class TestThread extends Thread {
        int loopNum = 100;
        private Counter counter;
        public TestThread(final Counter counter, int loopNum) {
            this.counter = counter;
            this.loopNum = loopNum;
        }
        public void run() {
            try {
                barrier.await();// 等待所有的线程开始
                for (int i = 0; i < this.loopNum; i++)
                    counter.increment();
                barrier.await();// 等待所有的线程完成
            } catch (Exception e) {
                throw new RuntimeException(e);
            }
        }
    }
}
```

从程序中可以看出两路 threadNum 和 loopNum 的值分别为 5000 和 100，就是创建 5000 个线程，每个线程循环 100 次。运行结果如下：

```
内部锁  count value:500000
内部锁  花费时间:1406 毫秒
不公平重入锁  count value:500000
不公平重入锁  花费时间:704 毫秒
公平重入锁  count value:500000
公平重入锁  花费时间:22796 毫秒
```

可以看出不公平重入锁需要的时间小于内部锁，公平重入锁需要的时间最多。

把 threadNum 修改为 500，loopNum=100；

运行结果如下：

```
内部锁  count value:50000
内部锁  花费时间:47 毫秒
不公平重入锁  count value:50000
不公平重入锁  花费时间:47 毫秒
公平重入锁  count value:50000
```

```
公平重入锁 花费时间:953 毫秒
```

threadNum=2000, loopNum=100；运行结果如下

```
内部锁 count value:200000
内部锁 花费时间:484 毫秒
不公平重入锁 count value:200000
不公平重入锁 花费时间:125 毫秒
公平重入锁 count value:200000
公平重入锁 花费时间:7500 毫秒
```

threadNum=2000, loopNum=1000；运行结果如下

```
内部锁 count value:2000000
内部锁 花费时间:921 毫秒
不公平重入锁 count value:2000000
不公平重入锁 花费时间:750 毫秒
公平重入锁 count value:2000000
公平重入锁 花费时间:57813 毫秒
```

从上面的运行结果可以看出，非公平重入锁的性能最好，公平重入锁的性能最差。在线程数比较少的情况下，内部锁和非公平重入锁的性能相当。

ReentrantLock 还有两个比较重要的方法是：**tryLock**()和 tryLock(long timeout, TimeUnit unit)。tryLock()仅在调用时锁未被另一个线程保持的情况下，才获取该锁。后者如果锁在给定等待时间内没有被另一个线程持有，且当前线程未被中断，则获取该锁。其他方法详细看 JDK 文档。

3.5.2 ReadWriteLock

ReadWriteLock 维护了一对相关的锁，一个用于只读操作，另一个用于写入操作。只要没有 writer，读取锁可以由多个 reader 线程同时保持。写入锁是独占的。

所有 ReadWriteLock 实现都必须保证 writeLock 操作的内存同步效果也要保持与相关 readLock 的联系。也就是说，成功获取读锁的线程会看到写入锁之前版本所做的所有更新。

与互斥锁相比，读-写锁允许对共享数据进行更高级别的并发访问。虽然一次只有一个线程（writer 线程）可以修改共享数据，但在许多情况下，任何数量的线程可以同时读取共享数据（reader 线程），读-写锁利用了这一点。从理论上讲，与互斥锁相比，使用读-写锁所允许的并发性增强将带来更大的性能提高。在实践中，只有在多处理器上并且只在访问模式适用于共享数据时，才能完全实现并发性增强。

与互斥锁相比，使用读-写锁能否提升性能则取决于读写操作期间读取数据相对于修改数据的频率，以及数据的争用——即在同一时间试图对该数据执行读取或写入操作的线程数。例如，某个最初用数据填充并且之后不经常对其进行修改的 collection，因为经常对其进行搜索（比如搜索某种目录），所以这样的 collection 是使用读-写锁的理想候选者。但是，如果数据更新变得频繁，数据在大部分时间都被独占锁，这时，就算存在并发性增强，也是微不足道的。更进一步地说，如果读取操作所用时间太短，则读-写锁实现（它本身就比互斥锁复杂）的开销将成为主要的执行成本，在许多读-写锁实现仍然通过一小段代码将所有线程序列化时更是如此。最终，只有通过分析和测量，才能确定应用程序是否适合使用读-写锁。

下面是一个使用读写锁的例子，创建几个写线程和读线程对 HashMap 中数据进行操作。读线程的个数多于写线程，也就是说读取数据的频率高于修改数据的频率。使用读写锁比合适。

```
package locks.readwritelock;
import java.util.Calendar;
import java.util.Map;
import java.util.TreeMap;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantReadWriteLock;
public class ReadWriteLockDemo {
    // 可重入读写锁
    private ReentrantReadWriteLock lock = null;
    private Lock readLock = null;// 读锁
    private Lock writeLock = null;// 写锁
    public int key = 100;
    public int index = 100;
    public Map<Integer, String> dataMap = null;// 线程共享数据
    public ReadWriteLockDemo() {
        lock = new ReentrantReadWriteLock(true);
        readLock = lock.readLock();
        writeLock = lock.writeLock();
        dataMap = new TreeMap<Integer, String>();
    }
    public static void main(String[] args) {
        ReadWriteLockDemo tester = new ReadWriteLockDemo();
        // 第一次获取锁
        tester.writeLock.lock();
        System.out
                .println(Thread.currentThread().getName() + " get writeLock.");
```

```
        // 第二次获取锁，应为是可重入锁
        tester.writeLock.lock();
        System.out
                .println(Thread.currentThread().getName() + " get writeLock.");
        tester.readLock.lock();
        System.out.println(Thread.currentThread().getName() + " get readLock");
        tester.readLock.lock();
        System.out.println(Thread.currentThread().getName() + " get readLock");
        tester.readLock.unlock();
        tester.readLock.unlock();
        tester.writeLock.unlock();
        tester.writeLock.unlock();
        tester.test();
    }
    public void test() {
        // 读线程比写线程多
        for (int i = 0; i < 10; i++) {
            new Thread(new reader(this)).start();
        }
        for (int i = 0; i < 3; i++) {
            new Thread(new writer(this)).start();
        }
    }
    public void read() {
        // 获取锁
        readLock.lock();
        try {
            if (dataMap.isEmpty()) {
                Calendar now = Calendar.getInstance();
                System.out.println(now.getTime().getTime() + " R "
                        + Thread.currentThread().getName()
                        + " get key, but map is empty.");
            }
            String value = dataMap.get(index);
            Calendar now = Calendar.getInstance();
            System.out.println(now.getTime().getTime() + " R "
                    + Thread.currentThread().getName() + " key = " + index
                    + " value = " + value + " map size = " + dataMap.size());
            if (value != null) {
                index++;
            }
        } finally {
            // 释放锁
            readLock.unlock();
```

```
            }
            try {
                Thread.sleep(3000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        public void write() {
            writeLock.lock();
            try {
                String value = "value" + key;
                dataMap.put(new Integer(key), value);
                Calendar now = Calendar.getInstance();
                System.out.println(now.getTime().getTime() + " W "
                        + Thread.currentThread().getName() + " key = " + key
                        + " value = " + value + " map size = " + dataMap.size());
                key++;
                try {
                    Thread.sleep(500);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            } finally {
                writeLock.unlock();
            }
        }
    }
    class reader implements Runnable {
        private ReadWriteLockDemo tester = null;
        public reader(ReadWriteLockDemo tester) {
            this.tester = tester;
        }
        public void run() {
            Calendar now = Calendar.getInstance();
            System.out.println(now.getTime().getTime() + " R "
                    + Thread.currentThread().getName() + " started");
            for (int i = 0; i < 10; i++) {
                tester.read();
            }
        }
    }
    class writer implements Runnable {
        private ReadWriteLockDemo tester = null;
        public writer(ReadWriteLockDemo tester) {
```

```
            this.tester = tester;
        }
        public void run() {
            Calendar now = Calendar.getInstance();
            System.out.println(now.getTime().getTime() + " W "
                    + Thread.currentThread().getName() + " started");
            for (int i = 0; i < 10; i++) {
                tester.write();
            }
        }
    }
```

运行结果如下：

```
main get writeLock.
main get writeLock.
main get readLock
main get readLock
1235978402187 R Thread-3 started
1235978402187 W Thread-12 started
1235978402187 R Thread-8 started
1235978402187 R Thread-7 started
1235978402187 R Thread-5 started
1235978402187 R Thread-4 started
1235978402187 W Thread-10 started
1235978402187 R Thread-2 started
1235978402187 R Thread-6 started
1235978402187 R Thread-1 started
1235978402187 R Thread-0 started
1235978402187 W Thread-11 started
1235978402187 R Thread-9 started
1235978402187 R Thread-3 get key, but map is empty.
1235978402187 R Thread-3 key = 100 value = null map size = 0
1235978402187 W Thread-12 key = 100 value = value100 map size = 1
1235978402687 R Thread-5 key = 100 value = value100 map size = 1
1235978402687 R Thread-4 key = 100 value = value100 map size = 1
1235978402687 R Thread-7 key = 100 value = value100 map size = 1
1235978402687 R Thread-8 key = 102 value = value100 map size = 1
1235978402687 W Thread-10 key = 101 value = value101 map size = 2
.......
```

3.6 Fork-Join 框架

在 JDK 7 中，java.util.concurrent 包的新增功能之一是一个 fork-join 风格的并行分解框架。fork-join 概念提供了一种分解多个算法的自然机制，可以有效地应用硬件并行性。[12,13]

JDK7 中还未正式发布，目前提供的开发版本中还为包含相关 API，是 JSR166y 的一部分。主要参考 IBM DWs 上面的文章，相关代码仅供参考。

3.6.1 应用 Fork-Join

语言、库和框架形成了我们编写程序的方式。Alonzo Church 早在 1934 年就曾表明，所有已知的计算性框架对于它们所能表示的程序集都是等价的，程序员实际编写的程序集是由特定语言形成的，而编程模型（由语言、库和框架驱动）可以简化这些语言的表达。

另一方面，一个时代的主流硬件平台形成了我们创建语言、库和框架的方法。Java 语言从一开始就能够支持线程和并发性；该语言包括像 synchronized 和 volatile 这样的同步原语，而类库包含像 Thread 这样的类。然而，1995 年流行的并发原语反映了当时的硬件现状：大多数商用系统根本没有提供并行性，甚至最昂贵的系统也只提供了有限的并行性。当时，线程主要用来表示异步，而不是并发，而这些机制已足够满足当时的需求了。

随着多处理器系统价格降低，更多的应用程序需要使用这些系统提供的硬件并行性。而且程序员们发现，使用 Java 语言提供的低级原语和类库编写并发程序非常困难且容易出错。在 Java 5 中，java.util.concurrent 包被添加到 Java 平台，它提供了一组可用于构建并发应用程序的组件：并发集合、队列、信号量、锁存器（latch）、线程池等等。这些机制非常适合用于粗任务粒度的程序；应用程序只需对工作进行划分，使并发任务的数量不会持续少于可用的处理器数量。通过将对单个请求的处理用作 Web 服务器、邮件服务器或数据库服务器的工作单元，应用程序通常能满足这种需求，因此这些机制能够确保充分利用并行硬件。

技术继续发展，硬件的趋势非常清晰；摩尔定律表明不会出现更高的时钟频率，但是每个芯片上会集成更多的内核。很容易想象让十几个处理器繁忙地处理一个粗粒度的任务范围，比如一个用户请求，但是这项技术不会扩大到数千个处理器。在很短一段时间内流量可能会呈指数级增长，但最终硬件趋势将会占上风。当跨入多内核时代时，我们需要找到更细粒度的并行性，否则将面临处理器处于空闲的风险，即使还有许多工作需要处理。如果希望

跟上技术发展的脚步，软件平台也必须配合主流硬件平台的转变。最终，Java 7 将会包含一种框架，用于表示某种更细粒度并行算法的类：fork-join 框架。

如今，大多数服务器应用程序将用户请求-响应处理作为一个工作单元。服务器应用程序通常会运行比可用的处理器数量多很多的并发线程或请求。这是因为在大多数服务器应用程序中，对请求的处理包含大量 I/O，这些 I/O 不会占用太多的处理器（所有网络服务器应用程序都会处理许多的套接字 I/O，因为请求是通过套接字接收的；也会处理大量磁盘（或数据库）I/O）。如果每个任务的 90% 的时间用来等待 I/O 完成，您将需要 10 倍于处理器数量的并发任务，才能充分利用所有的处理器。随着处理器数量增加，可能没有足够的并发请求保持所有处理器处于繁忙状态。但是，仍有可能使用并行性来改进另一种性能度量：用户等待获取响应的时间。

一个典型网络服务器应用程序的例子是，考虑一个数据库服务器。当一个请求到达数据库服务器时，需要经过一连串的处理步骤。首先，解析和验证 SQL 语句。然后必须选择一个查询计划；对于复杂查询，数据库服务器将会评估许多不同的候选计划，以最小化预期的 I/O 操作数量。搜索查询计划是一种 CPU 密集型任务；在某种情况下，考虑过多的候选计划将会产生负面影响，但是如果候选计划太少，所需的 I/O 操作肯定比实际数量要多。从磁盘检索到数据之后，也许需要对结果数据集进行更多的处理；查询可能包含聚合操作，比如 SUM、AVERAGE，或者需要对数据集进行排序。然后必须对结果进行编码并返回到请求程序。

就像大多数服务器请求一样，处理 SQL 查询涉及到计算和 I/O。虽然添加额外的 CPU 不会减少完成 I/O 的时间（但是可以使用额外的内存，通过缓存以前的 I/O 操作结果来减少 I/O 数量），但是可以通过并行化来缩短请求处理的 CPU 密集型部分（比如计划评估和排序）的处理时间。在评估候选的查询计划时，可以并行评估不同的计划；在排序数据集时，可以将大数据集分解成更小的数据集，分别进行排序然后再合并。这样做会使用户觉得性能得到了提升，因为会更快收到结果（即使总体上可能需要更多工作来服务请求）。

合并排序是分治（ divide-and-conquer） 算法的一个例子，在这种算法中将一个问题递归分解成子问题，再将子问题的解决方案组合得到最终结果。并行分解方法常常称作 fork-join，因为执行一个任务将首先分解（fork）为多个子任务，然后再合并（join）（完成后）。

fork-join 框架支持几种风格的 ForkJoinTasks，包括那些需要显式完成的，以及需要循环执行的。下面程序是一个从大型数组中选择最大值的问题，使用的 RecursiveAction 类直

接支持 non-result-bearing 任务的并行递归分解风格；RecursiveTask 类解决 result-bearing 任务的相同问题（其他 fork-join 任务类包括 CyclicAction、AsyncAction 和 LinkedAsyncAction；要获得关于如何使用它们的更多细节，请查阅 Javadoc）。

下面的程序仅供参考，不一定能运行。

```
package forkjoin;
public class SelectMaxProblem {
    private final int[] numbers;
    private final int start;
    private final int end;
    public final int size = 1000;
    public SelectMaxProblem(int[] numbers2, int i, int j) {
        this.numbers = numbers2;
        this.start = i;
        this.end = j;
    }
    public int solveSequentially() {
        int max = Integer.MIN_VALUE;
        for (int i = start; i < end; i++) {
            int n = numbers[i];
            if (n > max)
                max = n;
        }
        return max;
    }
    public SelectMaxProblem subproblem(int subStart, int subEnd)
{
        return new SelectMaxProblem(numbers, start + subStart, start
+ subEnd);
    }
}
```

```
package forkjoin;
import jsr166y.ForkJoinPool;
public class MaxWithFJ {
    private final int threshold;
    private final SelectMaxProblem problem;
    public int result;
    public MaxWithFJ(SelectMaxProblem problem, int threshold) {
        this.problem = problem;
        this.threshold = threshold;
    }
    protected void compute() {
```

```
        if (problem.size < threshold)
            result = problem.solveSequentially();
        else {
            int midpoint = problem.size / 2;
            MaxWithFJ left = new MaxWithFJ(problem.subproblem(0,
midpoint),
                    threshold);
            MaxWithFJ right = new
MaxWithFJ(problem.subproblem(midpoint + 1,
                    problem.size), threshold);
            coInvoke(left, right);
            result = Math.max(left.result, right.result);
        }
    }
    public static void main(String[] args) {
        SelectMaxProblem problem = ….;
        int threshold = 500;
        int nThreads = 10;
        MaxWithFJ mfj = new MaxWithFJ(problem, threshold);
        ForkJoinExecutor fjPool = new ForkJoinPool(nThreads);
        fjPool.invoke(mfj);
        int result = mfj.result;
    }
}
```

使用传统的线程池来实现 fork-join 具有挑战性，因为 fork-join 任务将线程生命周期的大部分时间花费在等待其他任务上。这种行为会造成线程饥饿死锁（thread starvation deadlock），除非小心选择参数以限制创建的任务数量，或者池本身非常大。传统的线程池是为相互独立的任务设计的，而且设计中也考虑了潜在的阻塞、粗粒度任务。 fork-join 解决方案不会产生这两种情况。对于传统线程池的细粒度任务，也存在所有工作线程共享的任务队列发生争用的情况。

fork-join 框架通过一种称作工作窃取（work stealing） 的技术减少了工作队列的争用情况。每个工作线程都有自己的工作队列，这是使用双端队列（或者叫做 deque）来实现的（Java 6 在类库中添加了几种 deque 实现，包括 ArrayDeque 和 LinkedBlockingDeque）。当一个任务划分一个新线程时，它将自己推到 deque 的头部。当一个任务执行与另一个未完成任务的合并操作时，它会将另一个任务推到队列头部并执行，而不会休眠以等待另一任务完成（像 Thread.join() 的操作一样）。当线程的任务队列为空，它将尝试从另一个线程的 deque 的尾部 窃取另一个任务。

fork-join 方法提供了一种表示可并行化算法的简单方式，而不用提前了解目标系统将提供多大程度的并行性。所有的排序、搜索和数字算法都可以进行并行分解（以后，像 Arrays.sort() 这样的标准库机制将会使用 fork-join 框架，允许应用程序免费享有并行分解的益处）。随着处理器数量的增长，我们将需要在程序内部使用更多的并行性，以有效利用这些处理器；对计算密集型操作（比如排序）进行并行分解，使程序能够更容易利用未来的硬件。

3.6.2 应用 ParallelArray

随着处理器数量的增加，为了有效利用可用的硬件，我们需要识别并利用程序中更细粒度的并行性。最近几年中，选择粗粒度的任务边界（例如在 Web 应用程序中处理单一请求）和在线程池中执行任务，通常能够提供足够的并行性，实现可接受的硬件利用效率。但是如果要再进一步，就必须深入挖掘更多的并行性，以让硬件全速运转。一个成熟的并行领域就是大数据集中的排序和搜索。用 fork-join 可以很容易地表示这类问题。但是由于这些问题非常普遍，所以该类库提供了一种更简单的方法 — ParallelArray。

在主流服务器应用程序中，最适合更细粒度并行性的地方是数据集的排序、搜索、选择和汇总。其中的每个问题都可以用 divide-and-conquer 轻松地并行化，并能轻松地表示为 fork-join 任务。例如，要将对大数据集求平均值的操作并行化，可以递归地将大数据集分解成更小的数据集 — 就像在合并排序中做的那样 — 对子集求均值，然后在合并步骤中求出各子集的平均值的加权平均值。

对于排序和搜索问题，fork-join 库提供了一种表示可以并行化的数据集操作的非常简单的途径：ParallelArray 类。其思路是：用 ParallelArray 表示一组结构上类似的数据项，用 ParallelArray 上的方法创建一个对分解数据的具体方法的描述。然后用该描述并行地执行数组操作（幕后使用的是 fork-join 框架）。这种方法支持声明性地指定数据选择、转换和后处理操作，允许框架计算出合理的并行执行计划，就像数据库系统允许用 SQL 指定数据操作并隐藏操作的实现机制一样。ParallelArray 的一些实现可用于不同的数据类型和大小，包括对象数组和各种原语组成的数组。

ParallelArray 支持以下基本操作：

1） 筛选：选择计算过程中包含的元素子集。

2） 应用：将一个过程应用到每个选中的元素。

3） 映射：将选中的元素转换为另一种形式（例如从元素中提取数据字段）。

4） 替换：将每个元素替换为由它派生的另一个元素，创建新的并行数组。

此技术与映射类似，但是形成新的 ParallelArray，可以在其上执行进一步查询。替换的一种情况是排序，将元素替换为不同的元素，从而对其进行排序（内置的 sort() 方法可用于此操作）。另一种特殊情况是 cumulate() 方法，该方法根据指定的组合操作用累积值替换每个元素。替换操作也可用于组合多个 ParallelArray，例如创建一个 ParallelArray，其元素为对并行数组 a 和 b 执行 a[i]+b[i] 操作得到的值。

5） 汇总：将所有值组合为一个值，例如计算总和、平均值、最小值或最大值。

ParallelArray 并不是一种通用的内存中数据库，也不是一种指定数据转换和提取的通用机制；它只是用于简化特定范围的数据选择和转换操作的表达方式，以将这些操作轻松、自动地并行化。所以，它存在一些局限性；例如，必须在映射操作之前指定筛选操作。（允许多个筛选操作，但是将它们组合成一个复合筛选操作通常会更有效）。它的主要目的是使开发人员不用思考如何将工作并行化；如果能够用 ParallelArray 提供的操作表示转换，那么就能轻松实现并行化。

ParallelArray 提供了一种不错的方法，可用于声明性地指定数据集上的筛选、处理和聚合操作，还方便自动并行化。但是，尽管它的语法比使用原始的 for-join 库更容易表达，但还是有些麻烦；每个筛选器、映射器、reducer 通常被指定为内部类。Java 7 可能会在 Java 语言中加入闭包；支持闭包的一种说法是：闭包使得小段代码 — 例如 ParallelArray 中的筛选器、映射器、reducer —的表示更加紧凑。

随着可用的处理器数量增加，我们需要发现程序中更细粒度的并行性来源。最有吸引力候选方案之一是聚合数据操作——排序、搜索和汇总。JDK 7 中将引入的 fork-join 库提供了一种 “轻松表示” 某类可并行化算法的途径，从而让程序能够在一些硬件平台上有效运行。通过声明性地描述想要执行的操作，然后让 ParallelArray 确定具体的执行方法，fork-join 库的 ParallelArray 组件使并行聚合操作的表示变得更加简单。

由于 JDK 还未发布，没有编写能够实际运行的程序。

参考文献

1. Brian Goetz ， Java 理 论 与 实 践 : 流 行 的 原 子 ，

http://www-128.ibm.com/developerworks/cn/java/j-jtp11234/index.html
2. sun.misc.Unsafe源代码，http://docjar.org/html/api/sun/misc/Unsafe.java.html
3. java并发集合，http://www.ibm.com/developerworks/cn/java/j-tiger06164/index.html
4. http://blog.csdn.net/xiaojunjava/archive/2007/05/24/1624122.aspx
5. Simple Thread Control With Java's CountDownLatch ，http://www.developer.com/java/article.php/3713031
6. http://jncz.javaeye.com/blog/151729
7. http://www.ibm.com/developerworks/cn/java/j-jtp10264/index.html
8. http://blog.csdn.net/blackartanan/archive/2009/01/20/3839013.aspx
9. http://blog.csdn.net/doudou_bb_08/archive/2008/06/01/2400941.aspx
10. http://soft.zdnet.com.cn/software_zone/2007/1015/556305.shtml
11. 应用 fork-join 框架（第一部份）http://www.ibm.com/developerworks/cn/java/j-jtp11137.html
12. 应用 fork-join 框架（第二部份），http://www.ibm.com/developerworks/cn/ java /j-jtp03048.html

第 4 章 使用开源软件 Amino 构建并发应用程序

第 4 章 使用开源软件构建并发应用程序 1
4.1 开源软件Amino介绍 2
4.2 无锁（Lock-Free）数据结构 3
4.3 应用Amino提供的数据结构 6
4.3.1 简单集合 6
4.3.2 树 11
4.3.3 图 13
4.4 Amino使用的模式和调度算法 14
4.5 Amino的简单使用 17
参考资料： 20

在实际的并发线程应用程序中，常常会用到数组、树、图、集合等数据结构，而这些结构也涉及到并发线程所遇到的安全问题。采用 Amino 组件可以很方便地实现线程安全的数据结构。本章将介绍 Amino 组件在 Java 多线程中的使用。

4.1 开源软件 Amino 介绍

Amino是Apache旗下的开源软件。读者可以访问http://amino-cbbs.sourceforge.net/得到其最新版本。面向并发编程，它有以下特点：

1） 可操作性和良好的伸缩性

2） 跨平台性

3） 无论在 Java、C++或其他流行语言中，编程风格一致

4） 适用于多核的各种操作系统

5） 可以进行并发编程正确性的测试

本章将介绍 Amino 的 Java 版。Amino Java 类库将提供优化后的并发线程组件，适用于 JDK6.0 及其以后的版本。

Amino Java 类库将涉及下面四个方面的内容：

1） 数据结构

该组件将提供一套免锁的集合类。因为这些数据结构采用免锁的运算法则来生成，所以，它们将拥有基本的免锁组件的特性，如可以避免不同类型的死锁，不同类型的线程初始化顺序等。

2） 并行模式

Amino 将为应用程序提供一个或几个大家熟知的并行计算模式。采用这些并行模式可以使开发者起到事半功倍的效果，这些模式包括 Master-Worker、Map-reduce、Divide and conquer, Pipeline 等，线程调度程序可以与这些模式类协同工作，提供了开发效率。

3） 并行计算中的一般功能

Amino 将为应用程序提供并行计算中常用的方法，例如：

a. String、Sequence 和 Array 的处理方面。如 Sort、Search、Merge、Rank、Compare、Reverse、 Shuffle、Rotate 和 Median 等

b. 处理树和图的方法：如组件连接，树生成，最短路径，图的着色等

4）原子和 STM（软件事务内存模型）

下面的程序可以简单地演示使用 Amino 的例子：

```
// LogServerGood.java
package    org.amino.logserver;

import    org.amino.ds.lockfree.LockFreeQueue;
public    class    LogServerGood {
   /*Standard Queue interface*/
   private    Queue<String> queue;
   public    LogServerGood() throws    IOException {
/*Amino components are compatible with standard interface whenever    possible*/
     queue = new    LockFreeQueue<String>();
     }
}
```

4.2 无锁（Lock-Free）数据结构

我们知道，在传统的多线程环境下，我们需要共享某些数据，但为了避免竞争条件引致数据出现不一致的情况，某些代码段需要变成基于锁（Lock based）的原子操作去执行。加锁可以让某一线程可以独占共享数据，避免竞争条件，确保数据一致性。从好的一面来说，只要互斥体是在锁状态，就可以放心地进行任何操作，不用担心其它线程会闯进来搞坏你的共享数据。

然而，正是这种在互斥体的锁状态下可以为所欲为的机制同样也带来了很大的问题。例如，可以在锁期间读键盘或进行某些耗时较长的 I/O 操作，这种阻塞意味着其它想要占用正占用着的互斥体的线程只能被搁在一旁等着。更糟的是有可能引起死锁。基于锁（Lock based）的多线程设计更可能引发死锁、优先级倒置、饥饿等情况，令到一些线程无法继续其进度。

在 Amino 类库中，主要算法将使用锁无关的（Lock-Free）的数据结构。

锁无关（Lock-Free）算法，顾名思义，即不牵涉锁的使用。这类算法可以在不使用锁的情况下同步各个线程。对比基于锁的多线程设计，锁无关算法有以下优势：

- 对死锁、优先级倒置等问题免疫：它属于非阻塞性同步，因为它不使用锁来协调各个线程，所以对死锁、优先级倒置等由锁引起的问题免疫；
- 保证程序的整体进度：由于锁无关算法避免了死锁等情况出现，所以它能确保线程是在运行当中，从而确保程序的整体进度；
- 性能理想：因为不涉及使用锁，所以在普遍的负载环境下，使用锁无关算法可以

得到理想的性能提升。

自 JDK5 推出之后，包 java.util.concurrent.atomic 中的一组类为实现锁无关算法提供了重要的基础。锁无关数据结构是线程安全的，在使用时无需再编写额外代码去确保竞争条件不会出现。

而在锁无关多线程编程的世界里，几乎任何操作都是无法原子地完成的。只有很小一集操作可以被原子地进行，这一限制使得锁无关编程的难度大大地增加了。锁无关编程带来的好处是在线程进展和线程交互方面，借助于锁无关编程，你能够对线程的进展和交互提供更好的保证。

经过十几年的发展，锁无关的数据结构已经非常成熟，性能并不逊色于传统的实现方式。虽然编写锁无关算法十分困难的，但因为数据结构是经常被重用的部分，开发者可以使用现成的 API（如 Amino）轻易让程序进入锁无关的世界。

首先，一个“等待无关”的程序可以在有限步之内结束，而不管其它线程的相对速度如何。

一个“锁无关”的程序能够确保执行它的所有线程中至少有一个能够继续往下执行。这便意味着有些线程可能会被任意地延迟，然而在每一步都至少有一个线程能够往下执行。尽管有些线程的进度可能不如其它线程来得快，但系统作为一个整体总是在“前进”的。而基于锁的程序则无法提供上述的任何保证。一旦任何线程持有了某个互斥体并处于等待状态，那么其它任何想要获取同一互斥体的线程就只好站着干瞪眼；一般来说，基于锁的算法无法摆脱“死锁”或“活锁”的阴影，前者如两个线程互相等待另一个所占有的互斥体，后者如两个线程都试图去避开另一个线程的锁行为，就像两个在狭窄桥面上撞了个照面的家伙，都试图去给对方让路，结果像跳舞似的摆来摆去最终还是面对面走不过去。

2003 年，Maurice Herlihy因他在 1991 年发表的开创性论文“Wait-Free Synchronization”（ http://www.podc.org/dijkstra/2003.html）而获得了分布式编程的Edsger W. Dijkstra奖。在论文中，Herlihy证明了哪些原语对于构造锁无关数据结构来说是好的，哪些则是不好的。他证明了一些简单的结构就足以实现出任何针对任意数目的线程的锁无关算法。例如，Herlihy证明了原语Compare-and-swap（CAS） 是实现锁无关数据结构的通用原语。CAS 可以原子地比较一个内存位置的内容及一个期望值，如果两者相同，则用一个指定值取替这个内存位置里的内容，并且提供结果指示这个操作是否成功。很多现代的处理器已经提供了 CAS 的硬件实现，例如在 x86 架构下的 CMPXCHG8 指令。而在 Java 下，位于 java.util.concurrent.atomic 内的 AtomicReference<V> 类亦提供了 CAS 原语的实现，并且有很多其他的扩展功能。

下面我们来简单的了解一下硬件同步指令的工作方式：

在进行多处理时，现代 CPU 都可以通过检测或者阻止其他处理器的并发访问来更新共享内存，最通用的方法是实现 CAS（比较并转换）指令，例如在 Intel 处理器中 CAS 是通过 cmpxchg 指令实现的。CAS 操作过程是：当处理器要更新一个内存位置的值的时候，它首先将目前内存位置的值与它所知道的修改前的值进行对比（要知道在多处理的时候，你要更新的内存位置上的值有可能被其他处理更新过，而你全然不知），如果内存位置目前的值与期望的原值相同（说明没有被其他处理更新过），那么就将新的值写入内存位置；而如果不同（说明有其他处理在我不知情的情况下改过这的值咯），那么就什么也不做，不写入新的值（现在最新的做法是定义内存值的版本号，根据版本号的改变来判断内存值是否被修改，一般情况下，比较内存值的做法已经满足要求了）。CAS 的价值所在就在于它是在硬件级别实现的，速度那是相当的快。JDK5.0 中的原子类就是利用了现代处理器中的这个特性，可以在不进行锁定的情况下，进行共享属性访问的同步。

下面我们将举例说明锁无关栈（Stack）的实现方法。

栈能以数组或者链表作为底下的储存结构，虽然采取链表为基础的实现方式会占用多一点空间去储存代表元素的节点，但却可避免处理数组溢出的问题。故此我们将以链表作为栈的基础，本文不打算展开对栈数据结构的论述，仅给出相应的实现代码：

```
// 锁无关的栈实现
import java.util.concurrent.atomic.*;
 class Node<T> {
     Node<T> next;
     T value;
     public Node(T value, Node<T> next) {
          this.next = next;
          this.value = value;
     }
 }
 public class Stack<T> {
     AtomicReference<Node<T>> top = new AtomicReference<Node<T>>();
     public void push(T value) {
          boolean sucessful = false;
          while (!sucessful) {
               Node<T> oldTop = top.get();
               Node<T> newTop = new Node<T>(value, oldTop);
               sucessful = top.compareAndSet(oldTop, newTop);
          };
     }
```

```
        public T peek() {
            return top.get().value;
        }
        public T pop() {
            boolean sucessful = false;
            Node<T> newTop = null;
            Node<T> oldTop = null;
            while (!sucessful) {
                oldTop = top.get();
                newTop = oldTop.next;
                sucessful = top.compareAndSet(oldTop, newTop);
            }
            return oldTop.value;
        }
    }
```

成员数据 top 的类型为 AtomicReference<Node<T>> ， AtomicReference<V> 这个类可以对 top 数据成员施加 CAS 操作，亦即是可以允许 top 原子地和一个期望值比较，两者相同的话便用一个指定值取代。

4.3 应用 Amino 提供的数据结构

Amino Java 并发类库提供了应用程序常用的一些数据结构，如集合、树和图等，下面将分别举例说明。

4.3.1 简单集合

在 Amino 并发类库提供了 List,Queque,Set,Vector,Dirctionary,Stack,Deque 等数据结构，采用 Lock-Free 数据结构，可以确保线程安全。

1、 List

在 java.util.*包中，List 接口继承了 Collection 并声明了类集的新特性。使用一个基于零的下标，元素可以通过它们在列表中的位置被插入和访问。一个列表可以包含重复元素。Collection 接口是构造集合框架的基础，它声明所有类集合都将拥有的核心方法。

下面的例子中将采用 Amino 提供的线程安全的 `LockFreeList` 集合类。

【例 4.1】 采用并发线程的方式，构造共享 List 集合

```
// ListTest.java
    package org.amino.test;
  import java.util.List;
```

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicInteger;
import org.amino.ds.lockfree.LockFreeList; //Amino 提供的无锁数据结构

public class ListTest {
    private static final int ELEMENT_NUM = 80;
    public static void main(String[] argvs) {
        ExecutorService exec = Executors.newCachedThreadPool();
        final List<String> listStr = new LockFreeList<String>();
        for (int i = 0; i < ELEMENT_NUM; ++i) {
            exec.submit(new ListInsTask(listStr));
        }
        exec.shutdown();
        try {
            exec.awaitTermination(500, TimeUnit.SECONDS);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Size of list is " + listStr.size());
        for (int i = 1; i <= ELEMENT_NUM; ++i) {
            if (!listStr.contains(i)) {
                System.out.println("didn't find " + i);
            }
        }
    }
}
class ListInsTask implements Runnable {
    private static AtomicInteger count = new AtomicInteger();
    List list;
    public ListInsTask(List l) {
        list = l;
    }
    public void run() {
        if ( list.add(count.incrementAndGet())) {
            System.out.println("List Size= " + list.size() );
        }else{
                System.out.println("did not insert " + count.get());
           }
    }
}
```

程序运行结果可能（根据计算机具体情况而变化）如下：

```
List Size= 1
List Size= 2
List Size= 3
……
List Size= 33
List Size= 34
List Size= 35
List Size= 80
List Size= 79
List Size= 78
……
List Size= 38
List Size= 37
List Size= 36
Size of list is 80
```

该程序在线程池中运行，可以看出，线程的调度是并发和抢先式的。线程的结束和创建的顺序是不一样的，但依然保证了结果的正确性。

对上面的程序进行简单的修改，使用 Amino 提供的 LockFreeOrderedList 类，就可以实现有序的线程安全的 List 集合。

【例 4.2】 采用并发线程的方式，实现无锁结构的有序 List 集合

```
// OrderedListTest.java
package org.amino.test;
import java.util.*;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicInteger;
import org.amino.ds.lockfree.LockFreeOrderedList;
public class OrderedListTest{
    private static final int ELEMENT_NUM = 80;
    public static void main(String[] argvs) throws Exception {
        ExecutorService exec = Executors.newCachedThreadPool();
        final List<Integer> listStr = new LockFreeOrderedList<Integer>();
        for (int i = 0; i < ELEMENT_NUM; ++i) {
            exec.submit(new ListInsTask1(listStr));
        }
        exec.shutdown();
        try {
            exec.awaitTermination(500, TimeUnit.SECONDS);
        } catch (InterruptedException e) {
            e.printStackTrace();
```

```
            }

            System.out.println("Size of list is " + listStr.size());
            Thread.sleep(600L);
            Iterator iterator = listStr.iterator();
            int nn=1;
            while (iterator.hasNext()) {
                System.out.println( "After order:"+nn+"="+(Integer) iterator.next() );
nn++;
            }
        }
    }

    class ListInsTask1 implements Runnable {
        private static AtomicInteger count = new AtomicInteger();
        List list;
        public ListInsTask1(List l) {
            list = l;
        }
        public void run() {
            int rom;
            rom=(int)(1000 * java.lang.Math.random());
            System.out.println("rom="+rom);
            if ( list.add(count. addAndGet ( rom ) ) ){
                System.out.println("List Size= " + list.size() );
            }else{
                System.out.println("did not insert " + count.get());
            }
        }
    }
```

该程序将得到一个有序的共享集合序列 List。部分结果如下：

```
。。。。。。
After order:1=324
After order:2=852
After order:3=1291
After order:4=1640
After order:5=1754
After order:6=2560
After order:7=3377
After order:8=3594
。。。。。。  // 省略余下的部分。
```

2. Set

Set 接口定义了一个集合。它继承了 Collection 并说明了不允许重复元素的类集的特性。因此，如果试图将重复元素加到集合中时，add( )方法将返回 false。下面例子将使用 Amino 提供的无数锁的线程安全的 LockFreeSet.

【例 4.3】 采用并发线程的方式，无锁结构的 Set 集合

```
// SetTest.java
package org.amino.test;
import java.util.Iterator;
import java.util.Set;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicInteger;
import org.amino.ds.lockfree.LockFreeSet;
public class SetTest {
 private static final int ELEMENT_NUM = 80;
 public static void main(String[] argvs) throws Exception {
 ExecutorService exec = Executors.newCachedThreadPool();
 final Set<Integer> setStr = new LockFreeSet<Integer>();
 Future[] results = new Future[ELEMENT_NUM];
 for (int i = 0; i < ELEMENT_NUM; ++i) {
        results[i] = exec.submit(new SetInsTask(setStr));
 }
 try {
                for (int i = 0; i < ELEMENT_NUM; ++i) {
                        results[i].get();
                }
        } catch (InterruptedException e) {
                e.printStackTrace();
        } catch (ExecutionException e) {
                e.printStackTrace();
        }
        exec.shutdown();
        try {
                exec.awaitTermination(60, TimeUnit.SECONDS);
        } catch (InterruptedException e) {
                e.printStackTrace();
        }
        System.out.println("Size of set is " + setStr.size());
        for (int i = 1; i <= ELEMENT_NUM; ++i) {
                if (!setStr.contains(i)) {
```

```
                    System.out.println("didn't find " + i);
                }
            }
        Thread.sleep(600L);
        Iterator iterator = setStr.iterator();
            int nn=1;
            while (iterator.hasNext()) {
                System.out.println( "After insert:"+nn+"="+(Integer) iterator.next() ); nn++;
            }
        }
    }
    class SetInsTask implements Runnable {
        private static AtomicInteger count = new AtomicInteger();
        Set set;
        public SetInsTask(Set q) {
            set = q;
        }
        public void run() {
            if (!set.add(count.incrementAndGet())) {
                System.out.println("did not insert " + count.get());
            }
        }
    }
```

程序运行结果可能（根据计算机具体情况而变化）如下：

```
Size of set is 80
After insert:1=68
After insert:2=21
After insert:3=42
After insert:4=63
……
After insert:78=71
After insert:79=41
After insert:80=60
```

4.3.2 树

在树的种类中，二叉树是一种常见的数据结构。从二叉树的递归定义可知，一棵非空的二叉树由根结点及左、右子树这三个基本部分组成。下面的例子将使用 Amino 提供的无锁线程安全的二叉树

【例 4.4】无锁结构的二叉树

```
    // TreeTest.java
package org.amino.test;
package org.amino.examples;
import java.util.Queue;
```

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicInteger;
import org.amino.ds.lockfree.LockFreeQueue;
import org.amino.mcas.LockFreeBSTree;

public class TreeTest {
 private static final int ELEMENT_NUM = 1000;
 public static void main(String[] argvs) {
 ExecutorService exec = Executors.newCachedThreadPool();
final LockFreeBSTree<Integer, String> bstree = new LockFreeBSTree<Integer, String>();
        for (int i = 0; i < ELEMENT_NUM; ++i) {
            exec.submit(new InsertTask(bstree));
        }
        exec.shutdown();
        try {
            exec.awaitTermination(60, TimeUnit.SECONDS);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        Boolean result = true;
        for (int i = 1; i < ELEMENT_NUM; ++i) {
            if (bstree.find(i) == null) {
                System.out.println("didn't find " + i);
                result = false;
            }
        }

        if(result) {
            System.out.println("Test successfully!");
        }
       }
}

class InsertTask implements Runnable {
 private static AtomicInteger count = new AtomicInteger();
 LockFreeBSTree tree;

 public InsertTask(LockFreeBSTree tr) {
        tree = tr;
 }
 public void run() {
```

```
            int c = count.incrementAndGet();
            if(tree.update(c, c) != null) {
                System.out.println("did not insert " + c);
            }
        }
    }
```

程序运行结果如下：

```
Test successfully!
```

结果说明该集合建立成功。

4.3.3 图

一般的图算法涉及对图属性和类型、图搜索、有向图、最小生成树、最短路径以及网络流等研究。图形数据结构在工程设计、地理空间信息、模式识别等多方面有广泛的用途。Amino组件中实现了线程并发情况下线程安全的无锁数据结构Graph接口。Graph接口的内容如下：

```
package org.amino.ds.graph;
import java.util.Collection;
public interface Graph<E> extends Collection<E>, Cloneable {
    Collection<Node<E>> getNodes(E e);
    Collection<Node<E>> getAllNodes();
    Collection<Edge<E>> getEdges(Node<E> start, Node<E> end);
    Collection<Edge<E>> getEdges(E start, E end);
    Node<E> addNode(E e);
    Node<E> addNode(Node<E> node);
    boolean addAllNodes(Collection<Node<E>> nodes);
    boolean addEdge(E start, E end, double weight);
    boolean addEdge(Node<E> start, Node<E> end, double weight);
    boolean addEdge(Edge<E> edge);
    Collection<AdjacentNode<E>> getLinkedNodes(Node<E> node);
    Collection<Edge<E>> getLinkedEdges(Node<E> node);
    boolean removeEdge(Node<E> start, Node<E> end);
    boolean removeEdge(Edge<E> edge);
    boolean removeEdge(E start, E end);
    boolean removeNode(Node<E> node);
    boolean containsEdge(E start, E end);
    Graph<E> clone() throws CloneNotSupportedException;
    boolean containsNode(Node<E> start);
}
```

从上面的接口中可以看出，在Graph中实现了对节点、边的操作。由于Gragh的操作涉及太多的代码行，本章中没有给出响应的实例。有兴趣的读者可以参考amino-cbbs-0.3.1.jar和它的原代码，以及网上的案例，其网址是

http://amino-cbbs.sourceforge.net/qs_cpp_examples.html。（作者完成本书时，Amino组件正处于0.3.1版的阶段，很多功能还没有开发出来。）

4.4 Amino 使用的模式和调度算法

在Amino并发库中，将使用的模式和调度算法有：Master-Worker, Map-reduce, Divide and conquer, Pipeline等几种。本节将对Master-Worker作简单介绍。

Master－Worker 是一类典型的并行计算。在这类应用中，存在一个 Master，由它将一个大问题进行分割，分割后的各个小问题送给各个 Worker 进行计算，最后由 Master 将所有 Worker 计算结果进行汇总。在这一类的应用中，Master 只进行少量的计算，而主要的计算工作由各个 Worker 进行。

Master－Worker 并行计算模式分为静态模式和动态模式。在静态模式中，计算过程在不同的进度中进行。首先，所有的被分割后的各个小问题同时被分派给 Worker，然后 Worker 开始紧张的计算。在动态模式中，分派问题和计算小问题是同时动态进行的。

在Amino开源代码中提供了Master－Worker工厂摸式，代码如下：

```
package org.amino.pattern.internal;

/**
 * Classes for a MasterWorker Factory.
 * @author blainey
 *
 */
public class MasterWorkerFactory {
 /**
  *
  * @param <X> input type
  * @param <Y> result type
  * @param r work item
  * @return    StaticMasterWorker
  */
 public static <X,Y> MasterWorker<X,Y> newStatic(Doable<X,Y> r) {
       return new StaticMasterWorker<X,Y>(r);
 }

 /**
  *
  * @param <X> input type
  * @param <Y> result type
```

```
     * @param r work item
     * @param numWorkers number of workers (threads)
     * @return StaticMasterWorker
     */
    public static <X,Y> MasterWorker<X,Y> newStatic(Doable<X,Y> r, int numWorkers) {
        return new StaticMasterWorker<X,Y>(r,numWorkers);
    }

    /**
     *
     * @param <X> input type
     * @param <Y> result type
     * @param r work item
     * @return DynamicMasterWorker
     */
    public static <X,Y> MasterWorker<X,Y> newDynamic(DynamicWorker<X,Y> r) {
        return new DynamicMasterWorker<X,Y>(r);
    }

    /**
     *
     * @param <X> input type
     * @param <Y> result type
     * @param r work item
     * @param numWorkers number of workers
     * @return DynamicMasterWorker
     */
    public static <X,Y> MasterWorker<X,Y> newDynamic(DynamicWorker<X,Y> r, int
numWorkers) {
        return new DynamicMasterWorker<X,Y>(r,numWorkers);
    }
   }
```

从上面的代码中，我们可以看出Amino提供了StaticMasterWorker、DynamicMasterWorker两种底层的Master-Worker算法。

AbstractMasterWorker.java提供了Master-Worker算法的中基本的实现，由于代码太长，请读者参阅Amino的提供的源代码

对于StaticMasterWorker算法，他继承了AbstractMasterWorker类，其实现如下：

```
package org.amino.pattern.internal;
import java.util.Queue;
import java.util.concurrent.ConcurrentLinkedQueue;
import org.amino.scheduler.internal.AbstractScheduler;
```

```
    /**
     * Classes for a static MasterWorker, where upper bound of master workers is fixed once work
     * is initiated.
     *
     * @param <S> input type.
     * @param <T> result type.
     */
    class StaticMasterWorker<S,T> extends AbstractMasterWorker<S,T> {
     protected Queue<WorkItem> workQ = new ConcurrentLinkedQueue<WorkItem>();

     /**
      * @author ganzhi
      *
      */
     private class WorkWrapper implements Runnable {
          private Doable<S,T> w;

          public void run() {
               while (true) {
                    /*
                    // Go wait in the staff lounge
                    if (!waitInLounge()) break;
                    */

                    workerPool.startWork();
                    try {
                         while(true) {
                              final WorkItem input = workQ.poll();
                              if (input == null) break;

                              final T output = w.run(input.value());
                              resultMap.put(input.key(),output);
                         }
                    } finally {
                         workerPool.complete();
                         break;
                    }
               }
          }

          /**
           *
           * @param w work item
```

```
             */
            public WorkWrapper (Doable<S,T> w) {
                  this.w = w;
            }
      }

      /**
       *
       * @param r work item
       */
      public StaticMasterWorker(Doable<S,T> r) {
            this(r,AbstractScheduler.defaultNumberOfWorkers());
      }
      /**
       *
       * @param r work item
       * @param numWorkers size of worker pool.
       */
      public StaticMasterWorker(Doable<S,T> r, int numWorkers) {
            super(numWorkers);

            Runnable run = new WorkWrapper(r);
            for (int i=0; i<numWorkers; i++) workerPool.createWorker(i, run);
      }

      public boolean isStatic() { return true; }

      /**
         * {@inheritDoc}
         */
      public ResultKey submit(S w) {
            if (workerPool.anyStarted()) return null;

            ResultKey key = new ResultKeyImpl();
            boolean added = workQ.offer(new WorkItem(w,key));

            // The queue is unbounded so should fail to add new entries only in out of memory
situations
            assert added;

            return key;
      }
     }
```

4.5 Amino 的简单使用

下面讲解一个使用的Amino的集合的简单使用。在本例中，我们将采用字符串作为集合对象的值，然后对其进行排序及搜索等操作。

【例4.5】 多线程状态下`LockFreeVector`的使用及排序

```
package sample.amino;
import java.util.*;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicInteger;

import org.amino.ds.lockfree.*;
import org.amino.alg.sort.*;

public class StringDealSort {
    private static final int ELEMENT_NUM = 80;

    public static void main(String[] argvs) throws Exception {
        ExecutorService exec = Executors.newCachedThreadPool();

final LockFreeVector<String> vectorStr = new LockFreeVector<String>();

        Future[] results = new Future[ELEMENT_NUM];
        for (int i = 0; i < ELEMENT_NUM; ++i) {
            results[i] = exec.submit(new VectorInsTaskSort(vectorStr));
        }

        try {
            for (int i = 0; i < ELEMENT_NUM; ++i) {
                results[i].get();
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }

        exec.shutdown();
        try {
```

```
                    exec.awaitTermination(60, TimeUnit.SECONDS);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }

                System.out.println("Size of set is " + vectorStr.size());

                for (int i = 1; i <= ELEMENT_NUM; ++i) {
                    if (!vectorStr.contains(new Integer(i).toString())) {
                        System.out.println("didn't find " + i);
                    }
                }

                Thread.sleep(10L);
                QuickSorter qs=new QuickSorter();
                qs.sort(vectorStr);

                Iterator iterator = vectorStr.iterator();
                int nn=1;
                while (iterator.hasNext()) {
                    System.out.println(  "After  insert:"+nn+"="+(String)  iterator.next()  );
nn++;
                }
            }

        }

        class VectorInsTaskSort implements Runnable {
            private static AtomicInteger count = new AtomicInteger();
            LockFreeVector vector;

            public VectorInsTaskSort(LockFreeVector q) {
                vector = q;
            }

            public void run() {
                int rom=0;
                rom=(int)(26 * java.lang.Math.random());
                String romstr=new Integer(rom).toString();
       if( vector.contains(romstr) ) {
        if( vector.contains( new Integer(rom+26).toString() ) ) {
            if( vector.contains( new Integer(rom+26*2).toString() ) ) {
                if( vector.contains( new Integer(rom+26*3).toString() ) ) {
                                vector.add( new Integer(rom+26*4).toString() );
```

```
                                }else{
                                    vector.add( new Integer(rom+26*3).toString() );
                    }
                    }else{
                    vector.add( new Integer(rom+26*2).toString() );
                 }
                }else{
                 vector.add( new Integer(rom+26).toString() );
                }
         }else{
        vector.add(romstr);
         }
        }
        }
```

在本例中，我们在每个线程中向vector添加随机字符串对象，均采用了比较，如果存在，则在原来的字符值上再加上26，变成字符对象后然后再行添加，如此算法重复4次后，结果基本上没有重复的。

从本例中可以看出，vector的操作是线程安全的。

参考资料：

1) http://www.ibm.com/developerworks/cn/java/j-lo-lockfree/index.html
2) http://blog.csdn.net/chinajxw/archive/2006/03/08/618865.aspx
3) http://amino-cbbs.sourceforge.net/

第 5 章 数据冲突及诊断工具 MTRAT

第 5 章 数据冲突及诊断工具MTRAT.....1
5.1 如何避免数据冲突.....2
5.1.1 数据冲突与竞争条件.....2
5.1.2 锁与数据冲突.....4
5.1.3 采用原子性操作避免数据冲突.....9
5.1.4 采用Volatile避免数据冲突.....11
5.1.5ThreadLocal 14
5.2 使用阻塞队列的生产者-消费者模式.....15
5.3 MTRAT介绍.....19
5.3.1 有潜在数据冲突的例子.....20
5.3.2 MTRAT软件介绍.....22
5.3.3 Mtrat软件测试案例.....25
5.3.4 Mtrat软件的其他选项.....27
5.4 使用MTRAT诊断数据冲突.....28
参考文献：.....32

在前面的章节中，我们已经了解了线程安全和数据冲突的概念，在本章中我们将讲解如何避免数据冲突，以及如何进行诊断。由于并行程序的不确定性造成并行程序的错误很难查找，重现和调试，IBM 提供的 MTRAT 工具 可以收集程序的运行时信息，实时分析程序中所有可能的并行程序错误(如死锁、数据冲突)。

5.1 如何避免数据冲突

在前面的章节中我们已经了解了数据冲突。当线程之间共享数据引起了并发执行程序中的同步问题就是数据冲突。

Java 的数据有两种基本类型内存分配模式（不算虚拟机内部类型，详细内容参见虚拟机规范）：运行时栈和堆两种。由于运行时栈是线程所私有的，它主要用来保存局部变量和中间运算结果，因此它们的数据是不可能被线程之间所共享的。内存堆是创建类对象和数组地方，它们是被虚拟机内各个线程所共享的，因此如果一个线程能获得某个堆对象的引用，那么就称这个对象是对该线程可见的。

编写线程安全的代码，本质上就是管理对状态（state）的访问，而且通常这些状态都是共享的、可变的。一个对象的状态就是它的数据，存储在状态变量（state variables）中，比如实例域或静态域。对象的状态还包括了其他附属对象的域。

例如，在 Web 网站中，我们为统计系统的点击数设计了一个计数器。由于计数器是被多用户共享的，每个用户访问时都涉及“读-改-写”等操作，由于这些操作都不是原子的，计数器有可能出现问题。

两个线程在缺乏同步的条件下，试图同时更新一个计数器时。假设计数器的初始值为 19，在某些特殊的分时里，每个线程都将读它的值，并看到值是 19，然后同时加 1，最后都将 counter 设置为 20。很显然，这不是我们期望发生的事情：一次递增操作凭空取消了，一次命中计数被永久地取消了。在基于 Web 的服务中，如果计数器出现这种问题，可能问题不大，但已经导致严重的数据完整性问题和错误。如各在其他环境中，如银行帐号管理，那就不可原谅。

在并发编程环境中，这种问题有一个专用的名称叫竞争条件。

5.1.1 数据冲突与竞争条件

程序中如果存在数个竞争条件，将可能导致不正确的结果。当计算的正确性依赖于运行时中相关的时序或者多线程的交替时，会产生竞争条件。换句话说，想得到正确的答案，要依赖于“幸运”的时序。最常见的一种竞争条件是“检查再运行（check-then-act）”，使用一个潜在的过期值作为决定下一步操作的依据。

在现实生活中，我们也常常会遇到竞争条件。请看下面的从银行取钱的例子。在本例中，类 Account 代表一个银行账户。其中变量 balance 是该账户的余额。

【例 5-1】从银行账号取钱的例子

```
//Account.java
class Account
{
    double balance;
    public Account(double money)
    {
        balance = money;
        System.out.println("Totle Money: " + balance);
    }
}
```

下面我们定义一个线程，该线程的主要任务是从 Account 中取出一定数目的钱。

```
    //AccountThread.java
public class AccountThread extends Thread
{
    Account Account;
    int delay;
    public AccountThread(Account Account, int delay)
    {
        this.Account = Account;
        this.delay = delay;
    }
    public void run()
    {
        if (Account.balance >= 100) {
            try {
                sleep(delay);
                Account.balance = Account.balance - 100;
                System.out.println("withdraw    100 successful!");
            } catch (InterruptedException e) {
            }
        } else
            System.out.println("withdraw failed!");
    }
```

```
    public static void main(String[] args)
    {
        Account Account = new Account(100);
        AccountThread AccountThread1 = new AccountThread(Account, 1000);
        AccountThread AccountThread2 = new AccountThread(Account, 0);
        AccountThread1.start();
        AccountThread2.start();
    }
}
```

程序运行结果为：

```
Totle Money: 100.0
withdraw    100 successful!
withdraw    100 successful!
```

该结果非常奇怪，因为尽管账面上只有 100 元，但是两个取钱线程都取得了 100 元钱，也就是总共得到了 200 元钱。出错的原因在哪里呢？图 5-1 给出了一种导致这种结果的线程运行过程。

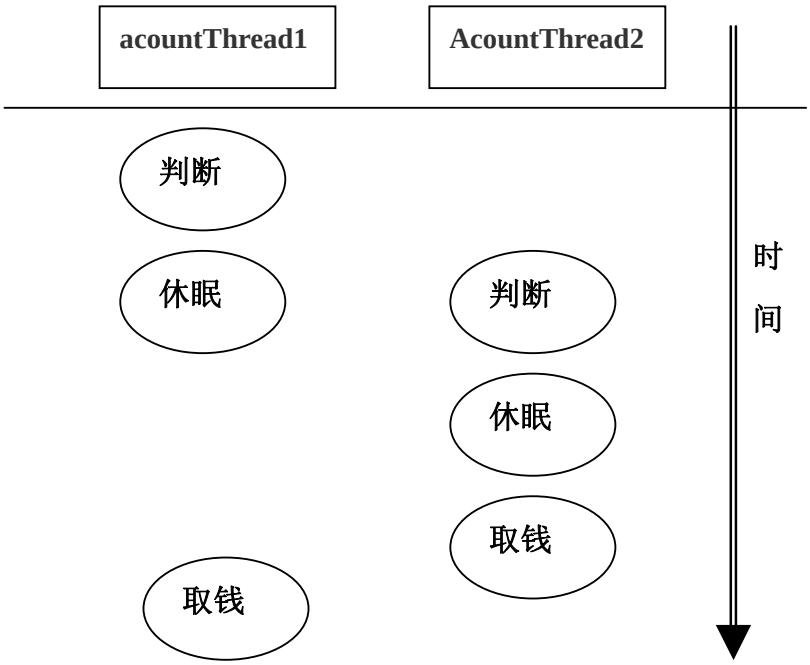


图 5-1 一种可能的线程运行过程

可以看出，由于线程 1 在判断满足取钱的条件后，被线程 2 打断，还没有来得及修改余额。因此线程 2 也满足取钱的条件，并完成了取钱动作。从而使共享数据 balance 的完整性被破坏。

上例中就出现了竞争条件，它使用了一个潜在的过期值作为决定下一步操作的依据。导致了数据冲突。在现实生活中，如果出现本例中的错误，那将无法容忍的。

5.1.2 锁与数据冲突

上面的问题，我们可以采用互斥锁的方式来解决（也可以采用其他方式来解决）。

在并发程序设计中，对多线程共享的资源或数据成为临界资源，而把每个线（进）程中访问临界资源的那一段代码段成为临界代码段。通过为临界代码段设置信号灯，就可以保证资源的完整性，从而安全地访问共享资源。

为了实现这种机制，Java 语言提供以下两方面的支持：

1 为每个对象设置了一个“互斥锁”标记。该标记保证在每一个时刻，只能有一个线程拥有该互斥锁，其它线程如果需要获得该互斥锁，必须等待当前拥有该锁的线程将其释放。该对象成为互斥对象。

2 为了配合使用对象的互斥锁，Java 语言提供了保留字 synchronized.其基本用法如下：

```
synchronized(互斥对象){
    临界代码段
}
```

当一个线程执行到该代码段时，首先检测该互斥对象的互斥锁。如果该互斥锁没有被别的线程所拥有，则该线程获得该互斥锁，并执行临界代码段，直到执行完毕并释放互斥锁；如果该互斥锁已被其它线程占用，则该线程自动进入该互斥对象的等候队列，等待其它线程释放该互斥锁。如图 5-2 所示，左边的图形表示，一个线程获得了对象的互斥锁，等待队列中有两个线程；右边的图形表示线程 1 释放互斥锁后，线程 2 获得互斥锁。

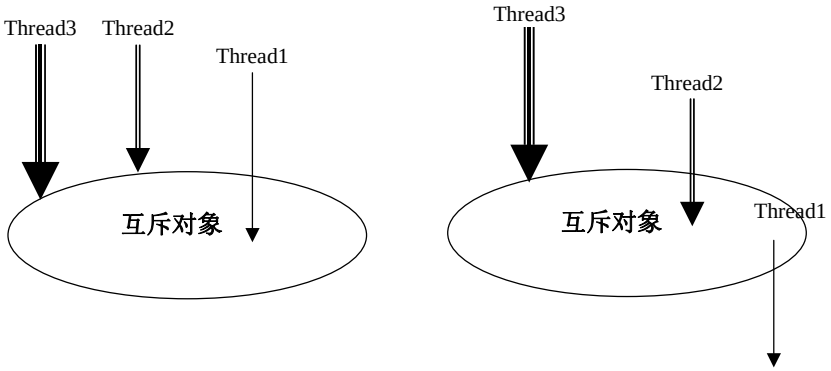


图 5-2 互斥对象及其等待队列

可以看出，任意一个对象都可以作为信号灯，从而解决上面存在的问题。我们首先定义一个互斥对象类，作为信号灯。由于该对象只作为信号量使用，所以我们并不需要为它定义其它的方法。因此该类的定义极其简单。

【例 5-2】使用互斥锁改写例 5-1

首先定义一个类，利用其对象作为互斥信号灯

```
// AccountThread2.java
class Semaphore{}

//我们可以对上面的程序进行修改，形成新的线程。
public class AccountThread2 extends Thread {
    Account account;
```

```
        int delay;
        Semaphore semaphore;
        public AccountThread2(Account account,int delay,Semaphore semaphore) {
        this.account =account;
        this.delay = delay;
        this.semaphore = semaphore;
      }
        public void run(){
        synchronized (semaphore) {
          if (account.balance >= 100) {
            try {
              sleep(delay);
              account.balance = account.balance - 100;
              System.out.println("withdraw    100 successful!");
            }
            catch (InterruptedException e) {
            }
          }
          else
            System.out.println("withdraw failed!");
        }
       }
       public static void main(String[] args) {
        Account account = new Account(100);
        Semaphore semaphore = new Semaphore();
        AccountThread2 accountThread1 = new
AccountThread2(account,1000,semaphore);
        AccountThread2 accountThread2 = new AccountThread2(account,0,semaphore);
        accountThread1.start();
        accountThread2.start();
      }
    }
```

运行该程序，其结果为：

```
Totle Money: 100.0
withdraw    100 successful!
withdraw failed!
```

在上面的程序中，对于临界资源 Account 的访问代码位于线程中。按照面向对象中封装对象的思想，我们应该将对资源的访问通过对象的方法来提供；另外，对象 Account 本身就是一个互斥对象，因此就可以作为信号灯。综合这两条，我们对 Account 对象进行修改如下：

【例 5-3】

```
// Account2.java
```

```
public class Account2
{
    double balance;
    public Account2(double money)
    {
        balance = money;
        System.out.println("Totle Money: " + balance);
    }
    public void withdraw(double money)
    {
        synchronized (this) {
            if (balance >= money) {
                balance = balance - money;
                System.out.println("withdraw 100 success");
            } else
                System.out.println("withdraw 100 failed!");
        }
    }
}
```

这样修改后，线程部分的代码变得很简单。

```
//AccountThread3.java
public class AccountThread3 extends Thread
{
    Account2 account;
    public AccountThread3(Account2 account)
    {
        this.account = account;
    }
    public void run()
    {
        account.withdraw(100);
    }
    public static void main(String[] args)
    {
        Account2 account = new Account2(100);
        AccountThread3 accountThread31 = new AccountThread3(account);
        AccountThread3 accountThread32 = new AccountThread3(account);
        accountThread31.start();
        accountThread32.start();
    }
}
```

其运行结果与上面相同。需要指出的是，在类 Account2 中，由于方法 withdraw 的所有

代码都为临界代码，所以也可以将关键字 synchronized 加在该方法的声明前面，如例 10-7 所示。它表示以方法所在的对象为互斥对象，因此不需要明确指出互斥对象，并且该方法的所有代码都作为临界代码。因此与 Account2 完全相同。

【例 5-4】

```
 //Account3
public class Account3 extends Thread
{
    double balance;
    public Account3(double money)
    {
        balance = money;
        System.out.println("Totle Money: " + balance);
    }
    public synchronized void withdraw(double money)
    {
        if (balance >= money) {
            balance = balance - money;
            System.out.println("withdraw 100 success");
        } else
            System.out.println("withdraw 100 failed!");
    }
}
```

也可以将关键字 synchronized 加在类的声明前面，表示该类的所有方法为临界代码（同步方法），该类的对象为互斥对象。

从上面的例子中，我们可以看出，Java 提供了强制原子性的内置锁机制：synchronized 块。一个 synchronized 块有两部分：锁对象的引用，以及这个锁保护的代码块。synchronized 方法是对跨越了整个方法体的 synchronized 块的简短描述，至于 synchronized 方法的锁，就是该方法所在的对象本身。（静态的 synchronized 方法从 Class 对象上获取锁。）

```
synchronized (lock) {
  // 访问或修改被锁保护的共享状态
}
```

每个 Java 对象都可以隐式地扮演一个用于同步的锁的角色；这些内置的锁被称作内部锁（intrinsic locks）或监视器锁（monitor locks）。执行线程进入 synchronized 块之前会自动获得锁；而无论通过正常控制路径退出，还是从块中抛出异常，线程都在放弃对 synchronized 块的控制时自动释放锁。获得内部锁的唯一途径是：进入这个内部锁保护的同步块或方法。

内部锁在 Java 中扮演了互斥锁（mutual exclusion lock，也称作 mutex）的角色，意味着

至多只有一个线程可以拥有锁，当线程 A 尝试请求一个被线程 B 占有的锁时，线程 A 必须等待或者阻塞，直到 B 释放它。如果 B 永远不释放锁，A 将永远等下去。

同一时间，只能有一个线程可以运行特定锁保护的代码块，因此，由同一个锁保护的 synchronized 块会各自原子地执行，不会相互干扰。在并发的上下文中，原子性的含义与它在事务性应用中相同——一组语句（statements）作为单独的，不可分割的单元运行。

执行 synchronized 块的线程，不可能看到会有其他线程能同时执行由同一个锁保护的 synchronized 块。

由于采用锁的机制可能出现等待或者阻塞，在多核系统中，效率是一个必须考虑的问题。

5.1.3 采用原子性操作避免数据冲突

通常在一个多线程环境下，我们需要共享某些数据，但为了避免竞争条件引致数据出现不一致的情况，某些代码段需要变成原子操作去执行。这时，我们便需要利用各种同步机制如互斥（Mutex）去为这些代码段加锁，让某一线程可以独占共享数据，避免竞争条件，确保数据一致性。但可惜的是，这属于阻塞性同步，所有其他线程唯一可以做的就是等待。基于锁（Lock based）的多线程设计更可能引发死锁、优先级倒置、饥饿等情况，令到一些线程无法继续其进度。

锁无关（Lock free）算法，顾名思义，即不牵涉锁的使用。这类算法可以在不使用锁的情况下同步各个线程。

自 JDK 1.5 推出之后，当中的 java.util.concurrent.atomic 的一组类为实现锁无关算法提供了重要的基础。下面我们采用原子操作改写例 4-1。

再过去的十多年里，人们已经对无等待且无锁定算法（也称为 无阻塞算法）进行了大量研究，许多人通用数据结构已经发现了无阻塞算法。无阻塞算法被广泛用于操作系统和 JVM 级别，进行诸如线程和进程调度等任务。虽然它们的实现比较复杂，但相对于基于锁定的备选算法，它们有许多优点：可以避免优先级倒置和死锁等危险，竞争比较便宜，协调发生在更细的粒度级别，允许更高程度的并行机制等等。

在 JDK 5.0 之前，如果不使用本机代码，就不能用 Java 语言编写无等待、无锁定的算法。在 java.util.concurrent.atomic 包中添加原子变量类之后，这种情况才发生了改变。所有原子变量类都公开比较并设置原语（与比较并交换类似），这些原语都是使用平台上可用的最快本机结构（比较并交换、加载链接/条件存储，最坏的情况下是旋转锁）来实现的。

java.util.concurrent.atomic 包中提供了原子变量的 9 种风格（ AtomicInteger； AtomicLong；AtomicReference； AtomicBoolean；原子整型；长型；引用；及原子标记引用和戳记引用类的数组形式；其原子地更新一对值）。

对上节例 5-1 的案例进行修改，采用原子性操作来修改，如下例 5-5。

【例 5-5】AtomicAccountTest.java

```
package test.race;

import java.util.concurrent.atomic.AtomicLong;
class AtomicAccount {
    AtomicLong balance;
    public AtomicAccount(long money) {
        balance = new AtomicLong(money);
        System.out.println("Totle Money: " + balance);
    }
    public void deposit(long money) {
         balance.addAndGet(money);
    }
    public void withdraw(long money, int delay) {
    long oldvalue = balance.get();
    if (oldvalue >= money) {
         try {
                Thread.sleep(delay);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            if (balance.compareAndSet(oldvalue, oldvalue - money)) {
                System.out.println(Thread.currentThread().getName()
                        + " withdraw    " + money + " successful!" + balance);
            } else {
                System.out.println(Thread.currentThread().getName()
                        + "thread concurrent, withdraw failed!" + balance);
            }
        } else {
            System.out.println(Thread.currentThread().getName()
                    + " balance is not enough,withdraw failed!" + balance);
        }
    }
    public long get() {
        return balance.get();
    }
}
```

```
public class AtomicAccountTest extends Thread {
    AtomicAccount account;
    int delay;

     public AtomicAccountTest(AtomicAccount account, int delay) {
         this.account = account;
         this.delay = delay;
     }
     public void run() {
         account.withdraw(100, delay);
     }
     public static void main(String[] args) {
         AtomicAccount account = new AtomicAccount(100);
         AtomicAccountTest accountThread1 = new AtomicAccountTest(account, 1000);
         AtomicAccountTest accountThread2 = new AtomicAccountTest(account, 0);
         accountThread1.start();
         accountThread2.start();
     }
}
```

运行结果如下:

```
Totle Money: 100
Thread-1 withdraw  100 successful!0
Thread-0thread concurrent, withdraw failed!0
```

在上一章中我们我们讲过了原子量的使用，现在修改 balance 为原子量。用原子量的特性实现取款操作的原子性。

把 Account 类修为 AtomicAccount，把 balance 定义为 AtomicLong 类型，然后修改 withdraw 方法，把原来方法的修改语句“balance = balance – money”修改为“balance.compareAndSet(oldvalue, oldvalue - money)”，这个方法在执行的时候是原子化的，首先比较所读取的值是否和被修改的值一致，如果一致则执行原子化修改，否则失败。如果帐余额在读取之后，被修改了，则 compareAndSet 会返回 FALSE，则余额修改失败，不能完成取款操作

5.1.4 采用 Volatile 避免数据冲突

Java 语言包含两种内在的同步机制：同步块（或方法）和 volatile 变量。这两种机制的提出都是为了实现代码线程的安全性。其中 Volatile 变量的同步性较差（但有时它更简单并且开销更低），而且其使用也更容易出错。volatile 变量可以被看作是一种“程度较轻的 synchronized”；与 synchronized 块相比，volatile 变量所需的编码较少，并且运行时开销也较

少，但是它所能实现的功能也仅是 synchronized 的一部分锁提供了两种主要特性：互斥（mutualexclusion）和可见性（visibility）。互斥即一次只允许一个线程持有某个特定的锁，因此可使用该特性实现对共享数据的协调访问协议，这样，一次就只有一个线程能够使用该共享数据。可见性要更加复杂一些，它必须确保释放锁之前对共享数据做出的更改对于随后获得该锁的另一个线程是可见的——如果没有同步机制提供的这种可见性保证，线程看到的共享变量可能是修改前的值或不一致的值，这将引发许多严重问题。

Volatile 变量具有 synchronized 的可见性特性，但是不具备原子特性。这就是说线程能够自动发现 volatile 变量的最新值。Volatile 变量可用于提供线程安全，但是只能应用于非常有限的一组用例：多个变量之间或者某个变量的当前值与修改后值之间没有约束。因此，单独使用 volatile 还不足以实现计数器、互斥锁或任何具有与多个变量相关的不变式（Invariants）的类。

在有限的一些情形下可以使用 volatile 变量替代锁。要使 volatile 变量提供理想的线程安全，必须同时满足下面两个条件：

1）对变量的写操作不依赖于当前值。

2）该变量没有包含在具有其他变量的不变式中

这些条件表明，可以被写入 volatile 变量的这些有效值独立于任何程序的状态，包括变量的当前状态。

第一个条件的限制使 volatile 变量不能用作线程安全计数器。虽然增量操作（x++）看上去类似一个单独操作，实际上它是一个由读取－修改－写入操作序列组成的组合操作，必须以原子方式执行，而 volatile 不能提供必须的原子特性。实现正确的操作需要使 x 的值在操作期间保持不变，而 volatile 变量无法实现这点。

大多数编程情形都会与这两个条件的其中之一冲突，使得 volatile 变量不能像 synchronized 那样普遍适用于实现线程安全。例 5-6 显示了一个非线程安全的数值范围类。它包含了一个不变式--下界总是小于或等于上界。

【例 5-6】//非线程安全的类

```
    @NotThreadSafe
publicclassNumberRange{
privateintlower,upper;
    publicintgetLower(){returnlower;}
publicintgetUpper(){returnupper;}
    publicvoidsetLower(intvalue){
if(value>upper)
```

```
thrownewIllegalArgumentException();
lower=value;
}
      publicvoidsetUpper(intvalue){
if(value<lower)
thrownewIllegalArgumentException();
upper=value;
}
}
```

这种方式限制了范围的状态变量，因此将 lower 和 upper 字段定义为 volatile 类型不能够充分实现类的线程安全；从而仍然需要使用同步。否则，如果凑巧两个线程在同一时间使用不一致的值执行 setLower 和 setUpper 的话，则会使范围处于不一致的状态。例如，如果初始状态是(0,5)，同一时间内，线程 A 调用 setLower(4)并且线程 B 调用 setUpper(3)，显然这两个操作交叉存入的值是不符合条件的，那么两个线程都会通过用于保护不变式的检查，使得最后的范围值是(4,3)——一个无效值。至于针对范围的其他操作，我们需要使 setLower()和 setUpper()操作原子化——而将字段定义为 volatile 类型是无法实现这一目的的。

volatile 操作不会像锁一样造成阻塞，因此，在能够安全使用 volatile 的情况下，volatile 可以提供一些优于锁的可伸缩特性。如果读操作的次数要远远超过写操作，与锁相比，volatile 变量通常能够减少同步的性能开销。

很多并发性专家事实上往往引导用户远离 volatile 变量，因为使用它们要比使用锁更加容易出错。然而，如果谨慎地遵循一些良好定义的模式，就能够在很多场合内安全地使用 volatile 变量。要始终牢记使用 volatile 的限制——只有在状态真正独立于程序内其他内容时才能使用 volatile——这条规则能够避免将这些模式扩展到不安全的用例。

例如：volatile 变量用于多个独立观察结果的发布

```
   publicclassUserManager{
   publicvolatileStringlastUser;
   publicbooleanauthenticate(Stringuser,Stringpassword){
   booleanvalid=passwordIsValid(user,password);
    if(valid){
        Useru=newUser();
        activeUsers.add(u);
        lastUser=user;
   }
   returnvalid;
   }
}
```

将某个值发布以在程序内的其他地方使用，但是与一次性事件的发布不同，这是一系列独立事件。这个情况要求被发布的值是有效不可变的——即值的状态在发布后不会更改。使用该值的代码需要清楚该值可能随时发生变化

Volatile 变量还可以用于下面的情况：状态标志、一次性安全发布、多个独立观察结果的发布、“volatilebean”模式、开销较低的读－写锁策略等。

例如：结合使用 volatile 和 synchronized 实现“开销较低的读－写锁”

```
    @ThreadSafe
publicclassCheesyCounter{
//Employsthecheapread-writelocktrick
//AllmutativeoperationsMUSTbedonewiththethislockheld
@LizhxedBy(this)
      privatevolatileintvalue;
      publicintgetValue(){returnvalue;}
      publicsynchronizedintincrement(){
            returnvalue++;
        }
      }
```

之所以将这种技术称之为“开销较低的读－写锁”是因为您使用了不同的同步机制进行读写操作。因为本例中的写操作违反了使用 volatile 的第一个条件，因此不能使用 volatile 安全地实现计数器——必须使用锁。然而，可以在读操作中使用 volatile 确保当前值的可见性，因此可以使用锁进行所有变化的操作，使用 volatile 进行只读操作。其中，锁一次只允许一个线程访问值，volatile 允许多个线程执行读操作，因此当使用 volatile 保证读代码路径时，要比使用锁执行全部代码路径获得更高的共享度——就像读－写操作一样。

与锁相比，Volatile 变量是一种非常简单但同时又非常脆弱的同步机制，它在某些情况下将提供优于锁的性能和伸缩性。如果严格遵循 volatile 的使用条件——即变量真正独立于其他变量和自己以前的值——在某些情况下可以使用 volatile 代替 synchronized 来简化代码。然而，使用 volatile 的代码往往比使用锁的代码更加容易出错

从本节上面的叙述中，我们可以得出下面的结论：解决数据冲突的规则有：锁规则，共享变量规则，线程启动和终止规则等。合理使用这些规则，可以避免数据冲突的发生。

5.1.5ThreadLocal

另外，还必须提到 ThreadLocal。因为 ThreadLocal 可以很好地解决 Sping 框架、Hibernate 框架中出现的多线程问题。

早在 JDK1.2 的版本中就提供 java.lang.ThreadLocal，ThreadLocal 为解决多线程程序的并发问题提供了一种新的思路。使用这个工具类可以很简洁地编写出优美的多线程程序。

ThreadLocal 很容易让人望文生义，想当然地认为是一个“本地线程”。其实，ThreadLocal 并不是一个 Thread，而是 Thread 的局部变量。当使用 ThreadLocal 维护变量时，ThreadLocal 为每个使用该变量的线程提供独立的变量副本，所以每一个线程都可以独立地改变自己的副本，而不会影响其它线程所对应的副本。

从线程的角度看，目标变量就象是线程的本地变量，这也是类名中“Local”所要表达的意思。

ThreadLocal 和线程同步机制相比有什么优势呢？ThreadLocal 和线程同步机制都是为了解决多线程中相同变量的访问冲突问题。

在同步机制中，通过对象的锁机制保证同一时间只有一个线程访问变量。这时该变量是多个线程共享的，使用同步机制要求程序慎密地分析什么时候对变量进行读写，什么时候需要锁定某个对象，什么时候释放对象锁等繁杂的问题，程序设计和编写难度相对较大。

而 ThreadLocal 则从另一个角度来解决多线程的并发访问。ThreadLocal 会为每一个线程提供一个独立的变量副本，从而隔离了多个线程对数据的访问冲突。因为每一个线程都拥有自己的变量副本，从而也就没有必要对该变量进行同步了。ThreadLocal 提供了线程安全的共享对象，在编写多线程代码时，可以把不安全的变量封装进 ThreadLocal。

概括起来说，对于多线程资源共享的问题，同步机制采用了“以时间换空间”的方式，而 ThreadLocal 采用了“以空间换时间”的方式。前者仅提供一份变量，让不同的线程排队访问，而后者为每一个线程都提供了一份变量，因此可以同时访问而互不影响。

ThreadLocal 是解决线程安全问题一个很好的思路，它通过为每个线程提供一个独立的变量副本解决了变量并发访问的冲突问题。在很多情况下，ThreadLocal 比直接使用 synchronized 同步机制解决线程安全问题更简单，更方便，且结果程序拥有更高的并发性。但仅限于数据结构中不涉及数据冲突的情况。

5.2 使用阻塞队列的生产者-消费者模式

在前面我们研究了共享资源的访问问题。在实际应用中，多个线程之间不仅需要互斥机制来保证对共享数据的完整性，而且有时需要多个线程之间互相协作，按照某种既定的步骤来共同完成任务。一个典型的应用是称之为生产-消费者模型。该模型可抽象为如图 5-3。其

约束条件为：

1) 生产者负责产品，并将其保存到仓库中；
2) 消费者从仓库中取得产品。
3) 由于库房容量有限，因此只有当库房还有空间时，生产者才可以将产品加入库房；否则只能等待。
4) 只有库房中存在满足数量的产品时，消费者才能取走产品，否则只能等待。

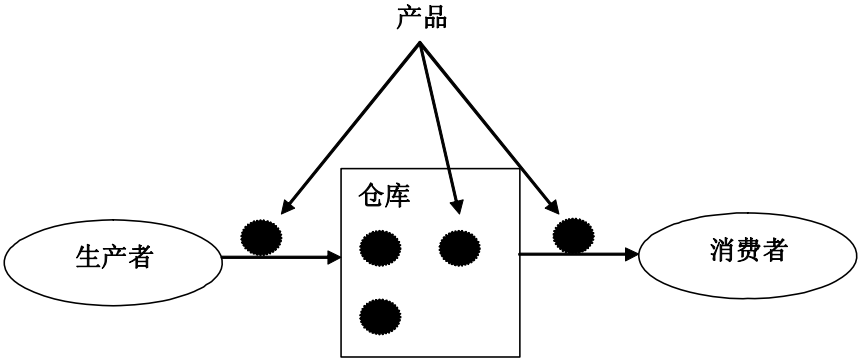


图 5-3 生产-消费者模型

实际应用中的许多例子都可以归结为该模型。如在操作系统中的打印机调度问题，库房的管理问题等。为了研究该问题，我们仍然以前面的存款与取款问题作为例子，假设存在一个账户对象(仓库)及两个线程：存款线程(生产者)和取款线程(消费者)，并对其进行如下的限制；

- 只有当账户上的余额 balance=0 时，存款线程才可以存进 100 元；否则只能等待；
- 只有当账户上的余额 balance=100 时，取款线程才可以取走 100 元；否则只能等待。

根据生产-消费者模型，我们应该得到一个运行序列：存款 100、取款 100、存款 100、取款 100…。很明显，使用我们前面的互斥对象，已无法完成这两个线程的同步问题。为此，Java 语言为互斥对象提供了两个方法，一个是 wait(),一个是 notify()，用于对两个线程进行同步。需要注意的事，这两个方法虽然用于线程同步，但却不是作为 Thread 类的方法提供，是不是有点奇怪？原因我们后面再讲。

wait()方法的语义是：当一个线程执行了该方法，则该线程进入阻塞状态，同时让出同步对象的互斥锁，并自动进入互斥对象的等待队列。

notify()方法的语义是:当一个线程执行了该方法，则拥有该方法的互斥对象的等待队列中的第一个线程被唤醒，同时自动获得该互斥对象的互斥锁，并进入就绪状态等待调度。

利用这两个方法，请看下面的程序。

【例 5-6】两个线程之间的同步

```
//Account4.java
    publicclassAccount4
    {
        doublebalance;

        publicAccount4()
        {
            balance=0;
            System.out.println("TotleMoney:"+balance);
        }

        publicsynchronizedvoidwithdraw(doublemoney)
        {
            if(balance==0)
                try{
                    wait();
                }catch(InterruptedExceptione){
                }
            balance=balance-money;
            System.out.println("withdraw100success");
            notify();
        }

        publicsynchronizedvoiddeposite(doublemoney)
        {
            if(balance!=0)
                try{
                    wait();
                }catch(InterruptedExceptione){
                }
            balance=balance+money;
            System.out.println("deposite100success");
            notify();
        }
    }


//WithdrawThread.java

publicclassWithdrawThreadextendsThread

{

   Account4account;
```

```
    publicWithdrawThread(Account4acount)
    {
        this.account=acount;
    }
    publicvoidrun()
    {
        for(inti=0;i<5;i++)
            account.withdraw(100);
    }
}
//DepositeThread.java
    publicclassDepositThreadextendsThread
    {
        Account4account;
        publicDepositThread(Account4account)
        {
            this.account=account;
        }
        publicvoidrun()
        {
            for(inti=0;i<5;i++)
                account.deposite(100);
        }
    }

//TestProCon.java
    publicclassTestProCon
    {
        publicstaticvoidmain(String[]args)
        {
            Account4acount=newAccount4();
            WithdrawThreadwithdraw=newWithdrawThread(acount);
            DepositThreaddeposite=newDepositThread(acount);
            withdraw.start();
            deposite.start();
        }
    }
```

运行程序，其执行结果如下：

```
    TotleMoney:0.0
```

```
deposite100success
withdraw100success
deposite100success
withdraw100success
deposite100success
withdraw100success
deposite100success
withdraw100success
deposite100success
withdraw100success
```

可见，该运行结果满足要求。关于这两个方法的使用，需要注意如下问题：

1）wait()和 notify()这两个方法必须位于临界代码段中。也就是说，执行该方法的线程必须已获得了互斥对象的互斥锁。这是因为这两个方法实际上也是在操作互斥对象的互斥锁：当一个线程调用 wait 方法进入阻塞状态，同时会释放互斥对象的互斥锁；只有当另一个线程调用互斥对象的 notify 方法被调用时，该互斥对象等待队列中的第一个线程才能进入就绪状态。这也就是为什么这两个方法是作为互斥对象的方法来实现，而不是作为 Thread 类的方法实现的原因。前面我们讲过，sleep 是作为 Thread 类的方法实现的，当一个线程通过调用 sleep 方法进入阻塞状态时，它并不操作互斥对象的互斥锁，也就是说该线程可能仍然拥有互斥对象的互斥锁。

2）wait()和 notify()方法必须配对使用。当某个线程由于调用某个互斥对象的 wait()方法进入阻塞状态，只有另一个线程调用该互斥对象的 notify()方法才能唤醒该线程，使其进入就绪状态，否则该线程将永远处于阻塞状态。

3）在某些情况下，可以根据需要使用 notifyall()方法。该方法也是互斥对象的方法，与 notify 方法功能相同，当该方法将会唤醒互斥对象等待对列中所有处于阻塞状态的线程，使其进入就绪状态。

5.3 MTRAT 介绍

处理器技术正在发生着重大的改革，超线程技术使得新型 CPU 具有真正能同时执行多个线程的能力。支持多线程的多核处理器将变成主流，但是在这样的硬件环境中，移植旧的程序或者编写新的程序是十分困难，并且容易出错。因为程序员不得不考虑和编写并行程序，不得不去担心程序中各个线程的通信，同步，负载平衡，数据竞争，死锁以及不确定的行为等等。

多线程运行时分析工具（Multi-ThreadRun-timeAnalysisTool，MTRAT）是由 IBM 开发

的一个即高效又准确的动态分析工具，它可以查找出多线程程序中的潜在的数据竞争和死锁。该工具通过修改程序的字节码，来收集为了检查死锁和数据竞争的程序的运行时信息。被修改过的程序在执行的过程中，会产生一些事件，这些事件会被 MTRAT 精巧设计的数据竞争检测和死锁检测算法分析。

MTRAT 把不同的技术集成到了一个单一的开发工具中，避免了用户使用的复杂性，使得 MTRAT 方便使用。MTRAT 主要由以下部分组成，

- 简单的命令行界面和 Eclipse 插件。输出 MTRAT 检查到的并行错误。
- 动态的 Java 字节码修改引擎。可以在 Java 类文件被 Java 虚拟机加载的时候，修改 Java 类。
- 程序运行时信息收集器。收集程序的动态信息，比如内存访问，线程同步，创建和结束。
- 高效的运行时分析引擎。收集到的运行时信息会被在线分析，如果发现潜在的并行错误，将会通过界面报告给用户。

Mtrat软件下载网址是http://www.alphaworks.ibm.com/tech/mtrat。

5.3.1 有潜在数据冲突的例子

在并行程序中，当两个并行的线程，在没有任何约束的情况下，访问一个共享变量或者共享对象的一个域，而且至少要有一个操作是写操作，就会发生数据竞争错误。MTRAT 最强大的功能就是发现并行程序中潜在的数据竞争错误。下边例子中就隐藏了一个潜在的数据竞争错误。

【例 5-7】数据竞争

```
//DataRace.java
packagemtrat.test;
classValue
{
    privateintx;

publicValue()
{
x=10;
System.out.println("Value!"+x);
}

publicsynchronizedvoidadd(Valuev)
```

```
{
x=x+v.get();
System.out.println("Valueadd!"+x);
}

publicintget(){returnx;}
}
classTaskextendsThread
{
Valuev1,v2;

publicTask(Valuev1,Valuev2)
{
this.v1=v1;
this.v2=v2;
}

publicvoidrun()
{
    v1.add(v2);
    System.out.println("Valuerun!v1.x:v2.x"+v1.get()+":"+v2.get());
    }
}
publicclassDataRace
{
publicstaticvoidmain(String[]args)throwsInterruptedException
{
    System.out.println("mainbegin!");
    Valuev1=newValue();
Valuev2=newValue();
Threadt1=newTask(v1,v2);
Threadt2=newTask(v2,v1);
t1.start();
t2.start();
System.out.println("mainEnd!");
}
}
```

本例的运行结果如下：

```
mainbegin!
Value!10
Value!10
mainEnd!
Valueadd!20
Valuerun!v1.x:v2.x20:10
```

```
Valueadd!30
Valuerun!v1.x:v2.x30:20
```

上面程序虽然能运行，但该程序隐藏了一个潜在的数据竞争错误。类 Value 声明一个整形域 x，一个同步方法 add 修改这个域，和一个方法 get 返回域 x 的值。线程 t1、t2 在调用 add 方法时可能涉及 v1、v2 两个对象等待对方的数据的冲突问题。

5.3.2 MTRAT 软件介绍

Mtrat 软件运行的环境如下：

1） JavaSE6.0

2） 正确设置 JAVA_HOME

3） Eclipse3.2 或 Eclipse3.3(附带 MDTOCLExample 插件)。

1、命令行部分软件安装

按照 Mtrat 软件的使用说明书，现下载最新的 Windows 版本的 mtrat-instrument-analysis-[date].zip（要求下载 2008 年 12 月 08 日及以后的文档），然后解压得到 Mtrat 软件。将 Mtrat 软件解压至 D:\Mtrat（读者可自定）。然后从网上下载 asm-3.0-bin.zip，从中拷贝 asm-all-3.0.jar 至 D:\Mtrat。然后运行其中的 install.bat，其内容如下：

echo"Generatetarget.jar..."

java-cpasm-all-3.0.jar;class.jarcom.ibm.threadanalysis.dynamic.tool.InstrumentClass.

如果运行成功将在当前目录找到 target.jar，如图 5-4。

图 5-4Mtrat 命令行软件组成

2、Eclipse 插件安装

下载 com.ibm.threadanalysis.dynamic.feature.zip,解压后，将插件拷贝至 Eclipse 相应的目录，重新启动 Eclipse。安装成功后的 Eclipse 界面如图 5-5。

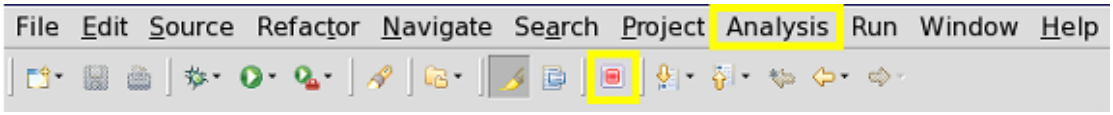


图 5-5MtratEclipse 插件

3、配置 Mtrat

在命令行软件和 Eclipse 插件安装好后，为了让它们能共同工作，下面进行配置。配置过程如下：

1）点击菜单"Window|Preference..."：如图 5-6

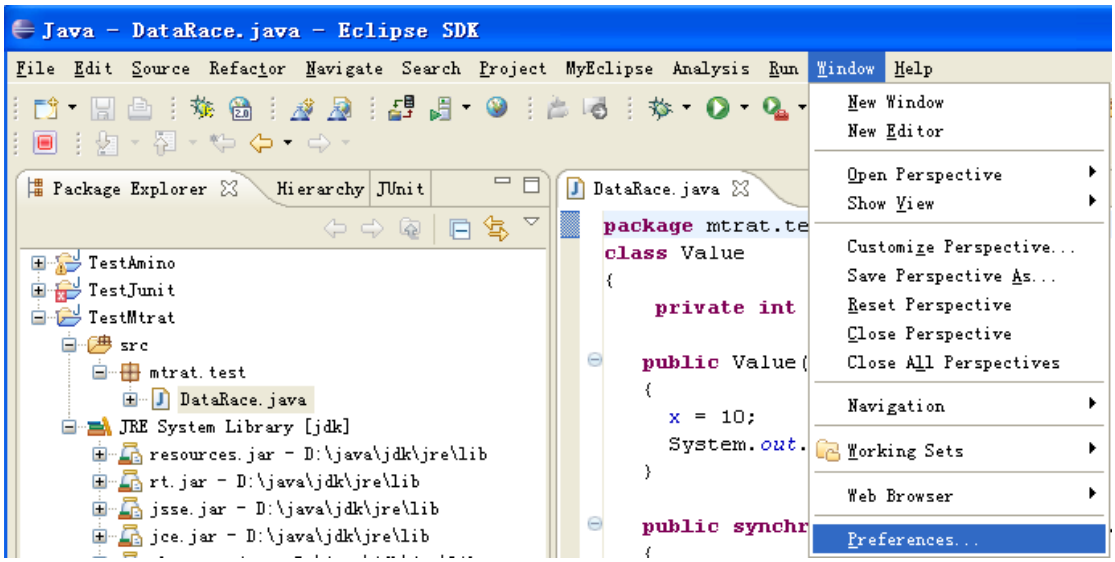


图 5-3EclipsePerences 界面

2）选择命令行软件安装目录 D:\Mtrat,如图 5-6。

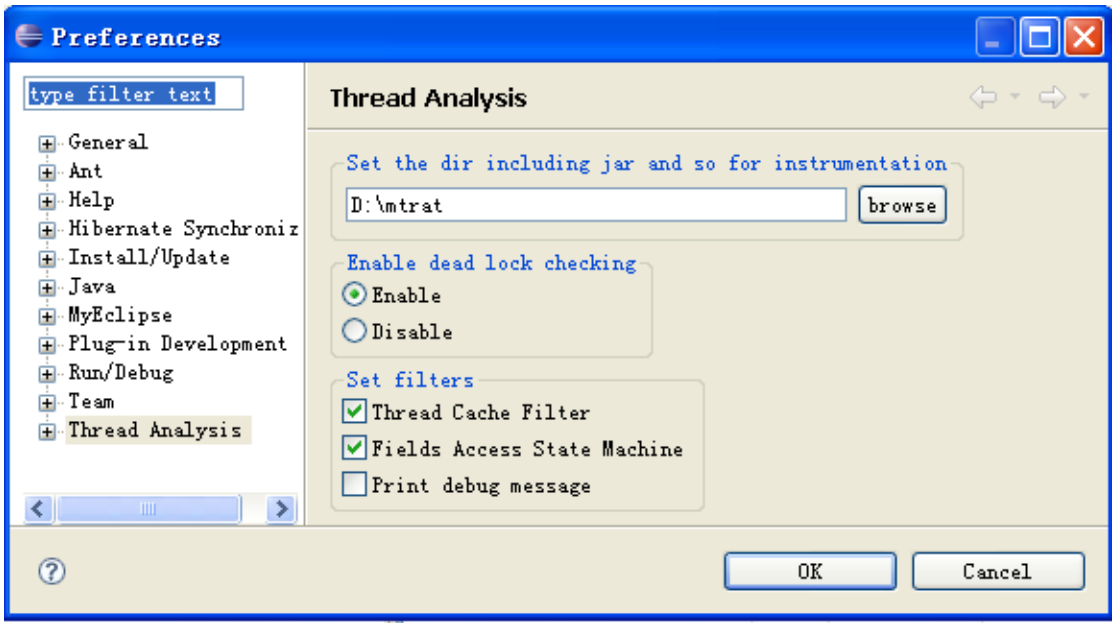


图 5-6Perences 界面下配置 Mtrat

要求 D:\Mtrat 目录中的必须具有下面的文档：asm-all-3.0.jar、instrument.jar、class.jar、target.jar、runtime.jar 和 libjvmtiagent.dll。

3）点击菜单"Window|ShowView",并点击 other：如图 5-7。

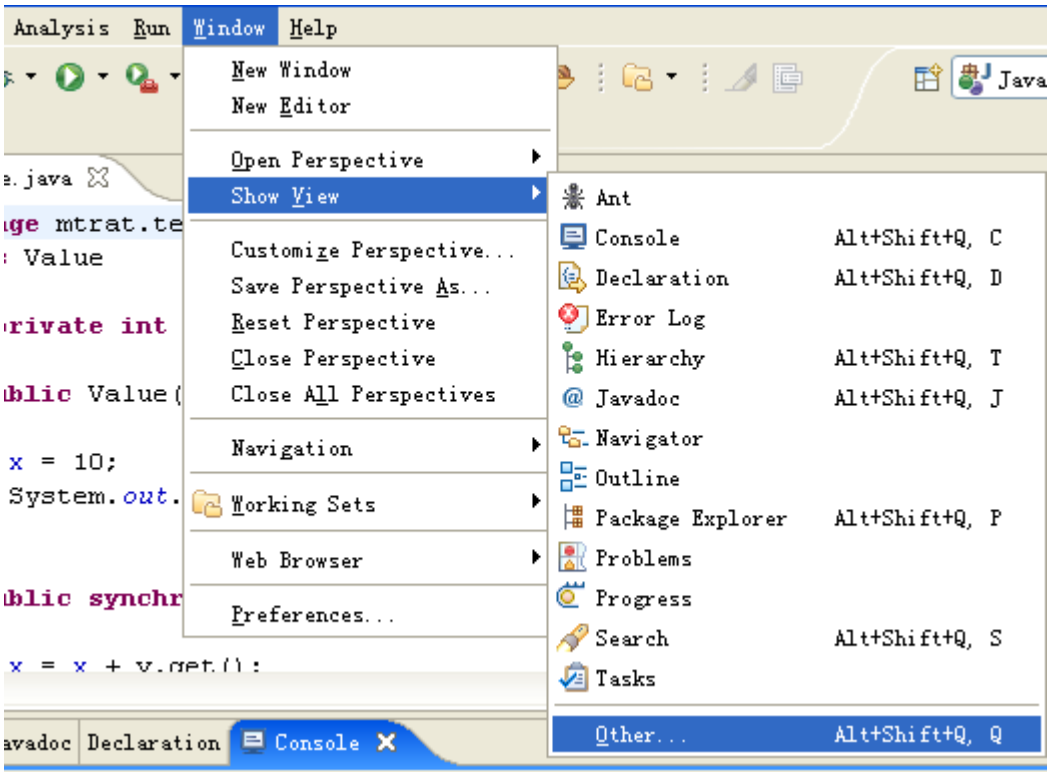


图 5-7Eclipse 窗口 ShowView 界面

4)然后选择 other 的子项 ThreadAnalysis,如图 5-8 和图 5-9

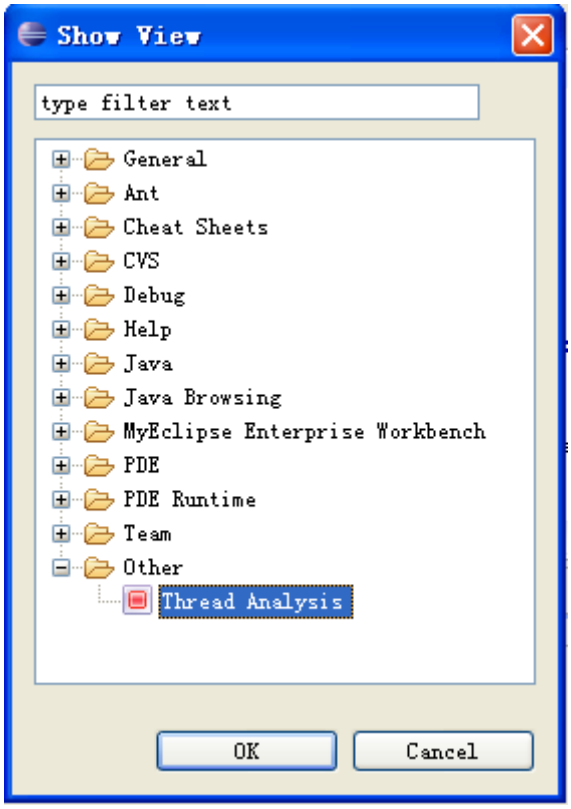


图 5-8ShowView 界面的选项

图 5-9MtratThreadAnalysis 窗口

5.3.3 Mtrat 软件测试案例

对例 5-1 的案例进行测试，结果如图 5-10：

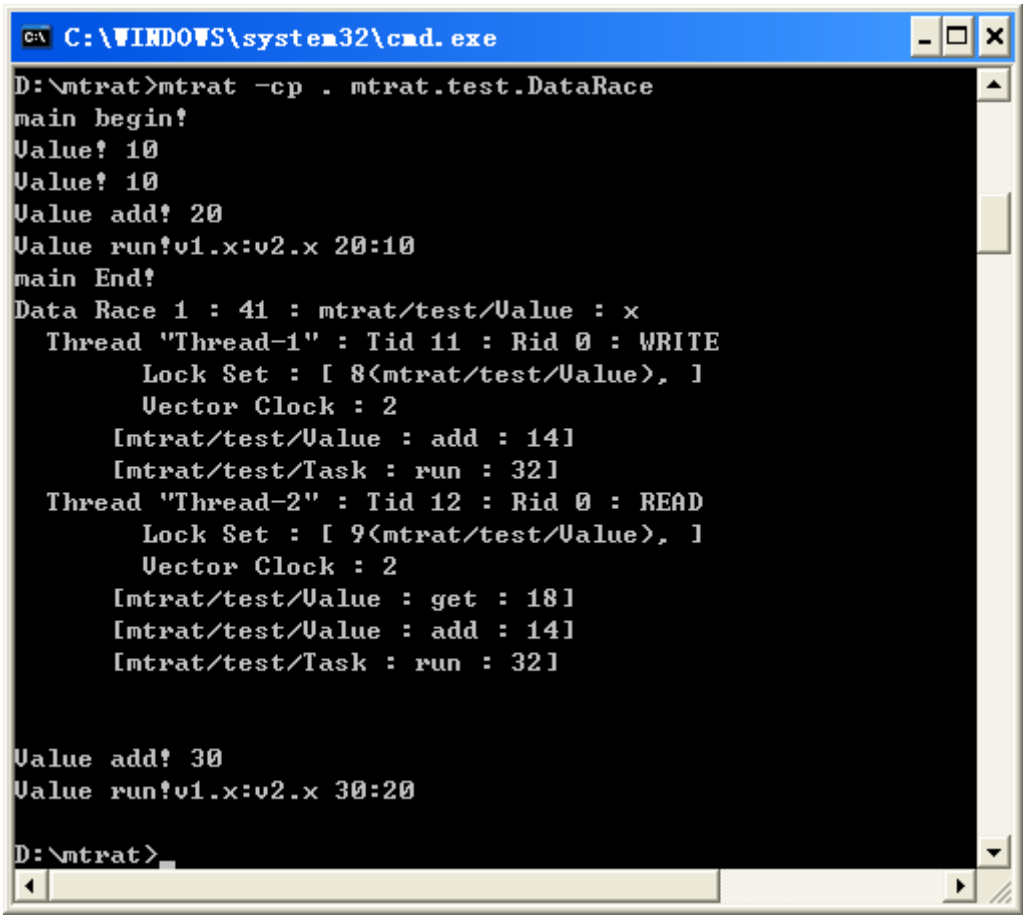


图 5-10Mtrat 分析案例命令行窗口

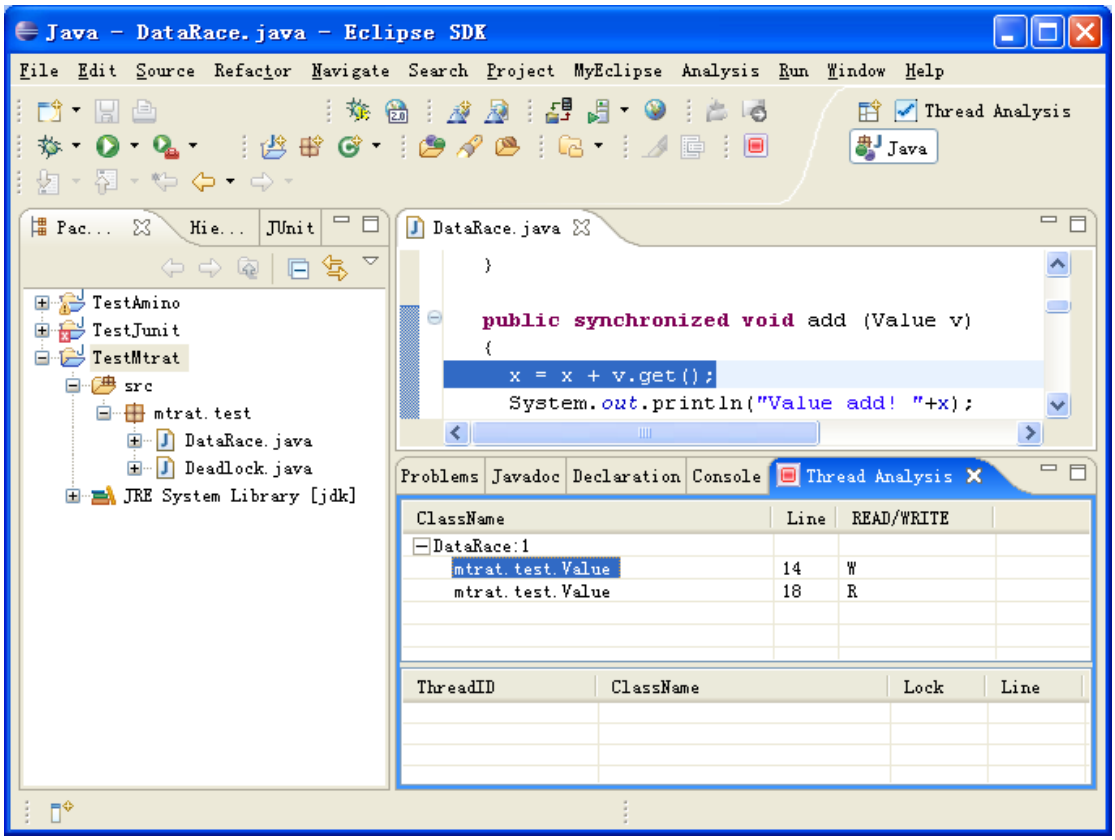


图 5-11Mtrat 分析案例 Eclipse 窗口

从上面的案例上来看，MTRAT 报告出了一个数据竞争错误，如图 5-11，因为类 Task 的两个实例会访问类 Value 的对象，然而这个共享的对象却没有被一个共同的锁保护。例如，在并行程序执行过程中，可能存在这样的时刻，一个线程执行方法 get 读域 x 的值，而另外一个线程执行执行方法 add 写域 x。

根据检查结果，MTRAT 发现了一个数据竞争错误，在类 sample/Value 域 x。程序员在得到这个数据竞争错误后，很容易就能发现程序中存在两个线程并发访问同一个对象域的可能。如果两个线程可以顺序访问这个对象域，这两个数据竞争问题就可以被消除了。

本例的 main 方法可以按下面的方式修改。

```
publicstaticvoidmain(String[]args)throwsInterruptedException
{
    System.out.println("mainbegin!");
    Valuev1=newValue();
Valuev2=newValue();
Threadt1=newTask(v1,v2);
Threadt2=newTask(v2,v1);
t1.start();
t1.join();//添加内容
t2.start();
t2.join();//添加内容
System.out.println("mainEnd!");
}
}
```

同时 Mtrat 还可以对死锁进行检查。Mtrat 对死锁检查的案例将在下一章进行讲述。

5.3.4 Mtrat 软件的其他选项

在命令行窗口的情况下，Mtrat 还有下面的表中的一些选项。如表 5-1

表 5-1Mtrat 命令行选项

选项	格式	解释
-x	-xpack1.class1:pack2.*	分析器不装载的类
-i	-ipack1.class1:pack2.*	分析器要装载的类
-Dcom.ibm.mtrat.instrument.include	-Dcom.ibm.mtrat.instrument.include=pack1.class1:pack2.*…	分析器内存存取事件所涉及的类
-Dcom.ibm.mtrat.instrument.noinit	-Dcom.ibm.mtrat.instrument.noinit=true|false	配置分析器存取事件的开关
-Dcom.ibm.mtrat.dbg.cl	-Dcom.ibm.mtrat.dbg.cl=false|true	分析器状态输出的开关
-Dcom.ibm.mtrat.deadlock	-Dcom.ibm.mtrat.dbg.cl=false|true	死锁检查开关

-Dcom.ibm.mtrat.threadcache	-Dcom.ibm.mtrat.threadcache=true|false	同地址的内存存取事件的优化
-Dcom.ibm.mtrat.osm	-Dcom.ibm.mtrat.osm=false|true	对象状态机制过滤器的优化

另外，在 Eclipse 插件的状态下，上面的一些选项也可以在面板中配置出来，如图 5-4 和图 5-12。

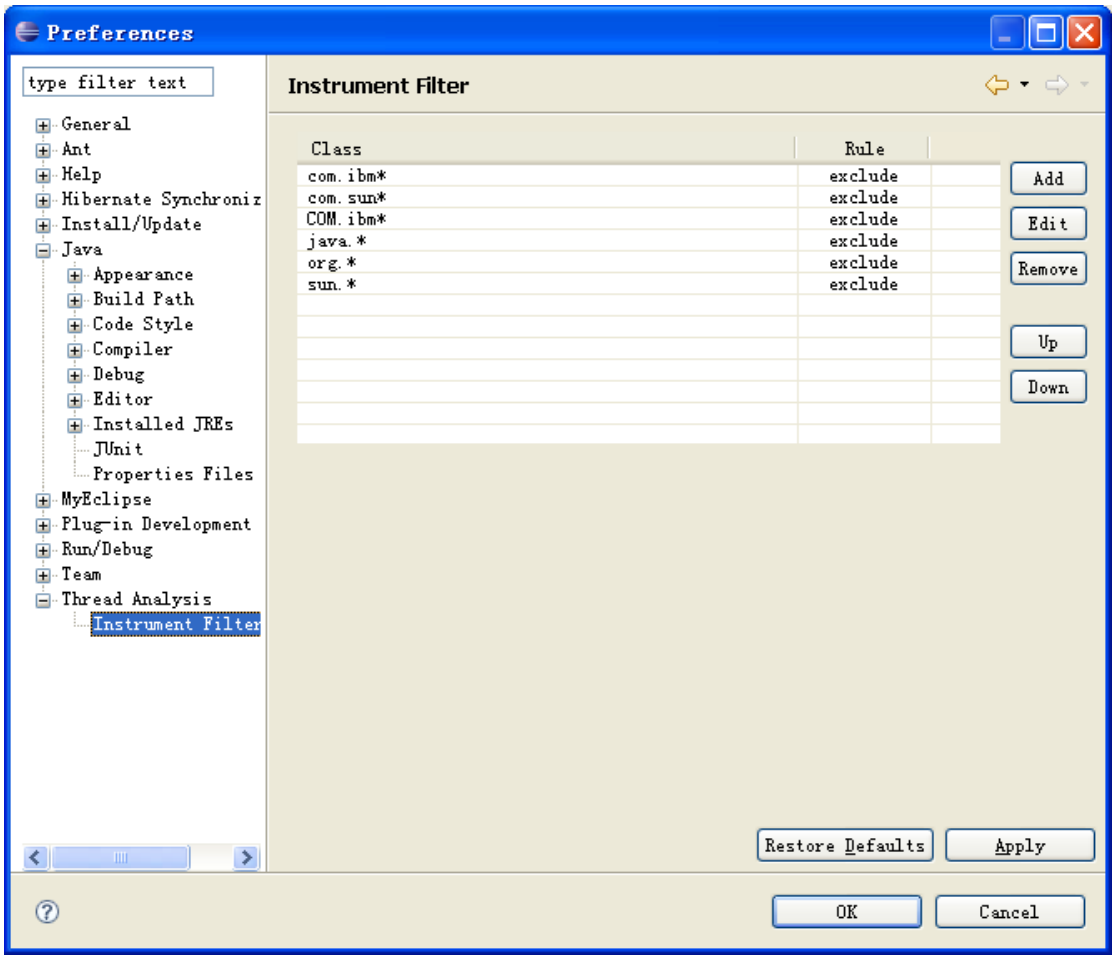


图 5-12Mtrat 分析器排出在外的类

在图 5-4 中，除了配置 Mtrat 的工作目录外，还有死锁检测开关和一些过滤器的设置。

5.4 使用 MTRAT 诊断数据冲突

在上一章中，我们使用 Amino 组件实现了高效的并发线程编程，例如使用 LockFreeList 实现线程安全的 List 集合。案例如下：

程序的代码如下：

```
//ListTest.java
packageorg.amino.test;
    importjava.util.List;
    importjava.util.concurrent.ExecutorService;
```

```
importjava.util.concurrent.Executors;
importjava.util.concurrent.TimeUnit;
importjava.util.concurrent.atomic.AtomicInteger;
importorg.amino.ds.lockfree.LockFreeList;
publicclassListTest{
    privatestaticfinalintELEMENT_NUM=80;
    publicstaticvoidmain(String[]argvs){
        ExecutorServiceexec=Executors.newCachedThreadPool();
        finalList<String>listStr=newLockFreeList<String>();
        for(inti=0;i<ELEMENT_NUM;++i){
            exec.submit(newListInsTask(listStr));
        }
        exec.shutdown();
        try{
            exec.awaitTermination(500,TimeUnit.SECONDS);
        }catch(InterruptedExceptione){
            e.printStackTrace();
        }
        System.out.println("Sizeoflistis"+listStr.size());
        for(inti=1;i<=ELEMENT_NUM;++i){
            if(!listStr.contains(i)){
                System.out.println("didn'tfind"+i);
            }
        }
    }

}
classListInsTaskimplementsRunnable{
    privatestaticAtomicIntegercount=newAtomicInteger();
    Listlist;

    publicListInsTask(Listl){
        list=l;
    }

    publicvoidrun(){
        if(list.add(count.incrementAndGet())){
            System.out.println("ListSize="+list.size());
        }else{
            System.out.println("didnotinsert"+count.get());
        }
    }
}
```

图 5-13 是 EclipseMtrat 插件分析的结果，可以看出没有出现数据冲突。

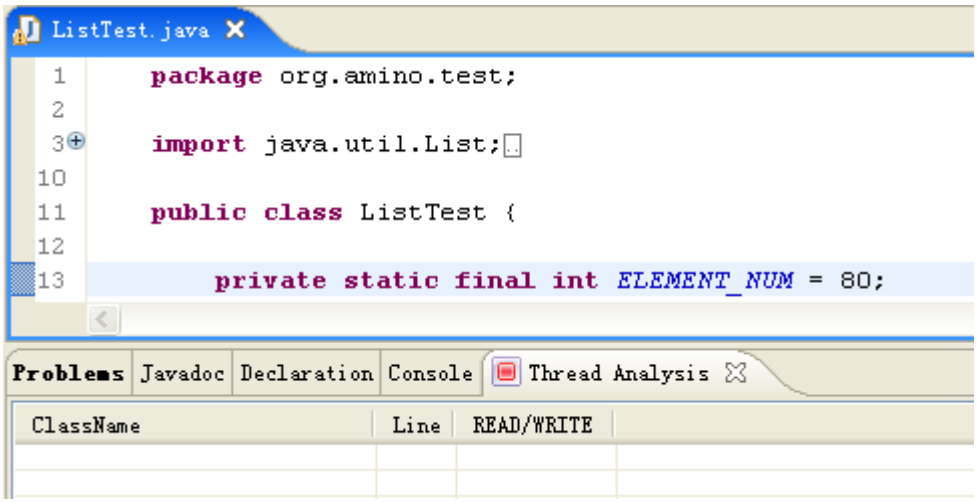


图 5-13Eclipse 插件分析结果

采用命令行分析的结果如下：

```
D:\mtrat>mtrat-cp.;d:\mtrat\amino-cbbs-0.3.1.jarorg.amino.test.ListTest
DataRace1:42:java/util/jar/JarFile:manRef
Thread"main":Tid1:Rid0:WRITE
LockSet:[]
VectorClock:1
[java/util/jar/JarFile:getManifestFromReference:0]
[java/util/jar/JarFile:getManifest:0]
[sun/misc/URLClassPath$JarLoader$2:getManifest:0]
[java/net/URLClassLoader$1:run:0]
[org/amino/test/ListTest:main:0]
Thread"pool-1-thread-1":Tid11:Rid209:READ
LockSet:[]
VectorClock:1
[java/util/jar/JarFile:getManifestFromReference:0]
[java/util/jar/JarFile:getManifest:0]
[sun/misc/URLClassPath$JarLoader$2:getManifest:0]
[java/net/URLClassLoader$1:run:0]
[org/amino/ds/lockfree/LockFreeList:add:0]
[org/amino/test/ListInsTask:run:51]

DataRace2:73:java/util/jar/Attributes$Name:hashCode
Thread"main":Tid1:Rid0:WRITE
LockSet:[]
VectorClock:1
[java/util/jar/Attributes$Name:hashCode:0]
[java/util/jar/Attributes:get:0]
[java/util/jar/Attributes:getValue:0]
[java/net/URLClassLoader$1:run:0]
[org/amino/test/ListTest:main:0]
```

```
Thread"pool-1-thread-1":Tid11:Rid209:READ
LockSet:[]
VectorClock:1
[java/util/jar/Attributes$Name:hashCode:0]
[java/util/jar/Attributes:get:0]
[java/util/jar/Attributes:getValue:0]
[java/net/URLClassLoader$1:run:0]
[org/amino/ds/lockfree/LockFreeList:add:0]
[org/amino/test/ListInsTask:run:51]

DataRace3:42:java/util/jar/JarFile:jv
Thread"main":Tid1:Rid0:WRITE
LockSet:[]
VectorClock:1
[java/util/jar/JarFile:getManifestFromReference:0]
[java/util/jar/JarFile:getManifest:0]
[sun/misc/URLClassPath$JarLoader$2:getManifest:0]
[java/net/URLClassLoader$1:run:0]
[org/amino/test/ListTest:main:0]
Thread"pool-1-thread-1":Tid11:Rid209:READ
LockSet:[17(sun/misc/URLClassPath$JarLoader$2),6(java/util/jar/JarFile),]
VectorClock:1
[java/util/jar/JarFile:maybeInstantiateVerifier:0]
[java/util/jar/JarFile:getInputStream:0]
[sun/misc/URLClassPath$JarLoader$2:getInputStream:0]
[sun/misc/Resource:cachedInputStream:0]
[sun/misc/Resource:getByteBuffer:0]
[java/net/URLClassLoader$1:run:0]
[org/amino/ds/lockfree/LockFreeList:add:0]
[org/amino/test/ListInsTask:run:51]

DataRace4:42:java/util/jar/JarFile:verify
Thread"main":Tid1:Rid0:WRITE
LockSet:[10(sun/misc/URLClassPath$JarLoader$2),6(java/util/jar/JarFile),]
VectorClock:1
[java/util/jar/JarFile:initializeVerifier:0]
[java/util/jar/JarFile:getInputStream:0]
[sun/misc/URLClassPath$JarLoader$2:getInputStream:0]
[sun/misc/Resource:cachedInputStream:0]
[sun/misc/Resource:getByteBuffer:0]
[java/net/URLClassLoader$1:run:0]
[org/amino/test/ListTest:main:0]
Thread"pool-1-thread-1":Tid11:Rid209:READ
LockSet:[]
```

```
VectorClock:1
[java/util/jar/JarFile:maybeInstantiateVerifier:0]
[java/util/jar/JarFile:access$000:0]
[java/util/jar/JarFile$JarFileEntry:getCodeSigners:0]
[sun/misc/URLClassPath$JarLoader$2:getCodeSigners:0]
[java/net/URLClassLoader$1:run:0]
[org/amino/ds/lockfree/LockFreeList:add:0]
[org/amino/test/ListInsTask:run:51]

ListSize=10
ListSize=11
……
ListSize=80
Sizeoflistis80
```

可以看出，上面的数据冲突均来至于 java.uti.jar.*,采用下面的命令行可以得到简洁的结果：

```
D:\mtrat>mtrat-cp.;d:\mtrat\amino-cbbs-0.3.1.jar-xjava.util.jar.*org.amino.test.ListTest
test.ListTest
ListSize=5
ListSize=5
……
ListSize=78
ListSize=55
……
ListSize=67
Sizeoflistis80
```

结果没有数据冲突。

参考文献：

1) http://www-128.ibm.com/developerworks/cn/java/j-jtp11234/index.html
2) http://www.zxbc.cn/html/20070802/25611.html
3) http://ec.icxo.com/htmlnews/2007/07/10/1157904.htm

第 6 章 死锁

第 6 章 死锁.....1
6.1 死锁概述.....2
6.2 死锁示例.....3
6.3 避免死锁和死锁诊断.....7
6.4 减小锁的竞争和粒度.....9
6.4.1 缩小锁的范围.....9
6.4.2 减小锁的粒度.....11
6.5 使用MTRAT诊断死锁.....12
6.6 饿死和活锁.....16
参考资料：.....18

多个线程同时被阻塞，它们中的一个或者全部都在等待某个资源被释放。由于线程被无限期地阻塞，因此程序不可能正常终止，这种情况叫死锁。本章将对 Java 多线程编程中可能出现死锁的情况进行详细的讲解，以及如何采用 MTRAT 来检查死锁。

6.1 死锁概述

线程又称为轻量级进程，它和进程一样拥有独立的执行控制，由操作系统负责调度，区别在于线程没有独立的存储空间，而是和所属进程中的其它线程共享一个存储空间，这使得线程间的通信较进程简单。编写多线程序时，必须注意每个线程是否干扰了其他线程的工作。每个进程开始生命周期时都是单一线程，称为“主线程”，在某一时刻主线程会创建一个对等线程。如果主线程停滞则系统就会切换到其对等线程。和一个进程相关的线程此时会组成一个对等线程池，一个线程可以杀死其任意对等线程。

因为每个线程都能读写相同的共享数据。这样就带来了新的麻烦：由于数据共享会带来同步问题，进而会导致死锁的产生。

由多线程带来的性能改善是以可靠性为代价的，主要是因为有可能产生线程死锁。死锁是这样一种情形：多个线程同时被阻塞，它们中的一个或者全部都在等待某个资源被释放。由于线程被无限期地阻塞，因此程序不能正常运行。简单的说就是:线程死锁时，第一个线程等待第二个线程释放资源，而同时第二个线程又在直接或间接等待第一个线程释放资源。这里举一个通俗的例子：如在人行道上两个人迎面相遇，为了给对方让道，两人同时向一侧迈出一步，双方无法通过，又同时向另一侧迈出一步，这样还是无法通过。假设这种情况一直持续下去，这样就会发生死锁现象。

更形象的例子如下： 五个哲学家围坐在一圆桌旁，每人的两边放着一支筷子，共五支筷子。大家边讨论问题边用餐。并规定如下的条件是：

1）每个人只有拿起位于自己两边的筷子，合成一双才可以用餐。

2）用餐后每人必须将两只筷子放回原处。

我们可以想象，如果每个哲学家都彬彬有礼，并且高谈阔论，轮流吃饭，则这种融洽的气氛可以长久地保持下去。但是可能出现这样一种情景：当每个人都拿起自己左手边的筷子，并同时去拿自己右手边的筷子时，会发生什么情况：五个人每人拿着一支筷子，盯着自己右手边的那位哲学手里的一支筷子，处于僵持状态。这就是发生了“线程死锁”。

多个线程竞争共享资源时可能出现的一种系统状态：线程 1 拥有资源 1，并等待资源 2，

而线程 2 拥有资源 2，并等待资源 3,…,以此类推，线程 n 拥有资源 n-1,并等待资源 1。在这种状态下，各个线程互不相让，永远进入一种等待状态。于是出现了死锁的现象。

虽然线程死锁只是系统的一种状态，该状态出现的机会可能会非常小，但简单的测试往往无法发现。遗憾的是 Java 语言也没有有效的方法可以避免或检测死锁。

6.2 死锁示例

下面给出出现死锁的一些案例。

【6-1】由 Mtrat 提供的线程死锁的案例

```
 class T3 extends Thread{
StringBuffer L1;
StringBuffer L2;
public T3(StringBuffer L1, StringBuffer L2) {
   this.L1 = L1;
   this.L2 = L2;
}
public void run() {
   synchronized (L1) {
      synchronized (L2) {
      }
   }
}
 }
public class Deadlock{
void harness2() throws InterruptedException
{
   StringBuffer L1 = new StringBuffer("L1");
   StringBuffer L2 = new StringBuffer("L2");

   Thread t1 = new T3(L1, L2);
   Thread t2 = new T3(L2, L1);
   t1.start();
   t2.start();
   t1.join();
   t2.join();
}
public static void main(String args) throws InterruptedException
{
   Deadlock dlt = new Deadlock();
   dlt.harness2();
}
}
```

在类 Deadlock 的 harness2 方法中，类 Deadlock 的两个实例被创建，作为参数传递到类 T3 的构造函数中。在类 T3 的 run 方法中，线程会依次获得这个两个对象的锁，然后以相反的顺序释放这两个锁。由于两个 StringBuffer 实例以不同的顺序传递给类 T3，两个线程会以不同的顺序获得这两个锁。这样，死锁就出现了。

【6-2】哲学家吃饭的案例

```
// ChopStick.java
public class ChopStick
{
    private String name;
    public ChopStick(String name)
    {
        this.name = name;
    }
    public String getNumber()
    {
        return name;
    }
}

// Philosopher.java
import java.util.*;
public class Philosopher extends Thread
{
    private ChopStick leftChopStick;
    private ChopStick rightChopStick;
    private String name;
    private static Random random = new Random();
    public Philosopher(String name, ChopStick leftChopStick,
            ChopStick rightChopStick)
    {
        this.name = name;
        this.leftChopStick = leftChopStick;
        this.rightChopStick = rightChopStick;
    }
    public String getNumber()
    {
        return name;
    }
    public void run()
    {
        try {
```

```
                sleep(random.nextInt(10));
            } catch (InterruptedException e) {
            }
            synchronized (leftChopStick) {
                System.out.println(this.getNumber() + " has "
                        + leftChopStick.getNumber() + " and wait for "
                        + rightChopStick.getNumber());
                synchronized (rightChopStick) {
                    System.out.println(this.getNumber() + " eating");
                }
            }
        }
        public static void main(String args[])
        {
            // 建立三个筷子对象
            ChopStick chopStick1 = new ChopStick("ChopStick1");
            ChopStick chopStick2 = new ChopStick("ChopStick2");
            ChopStick chopStick3 = new ChopStick("ChopStick3");
    ChopStick chopStick4 = new ChopStick("ChopStick4");
    ChopStick chopStick5 = new ChopStick("ChopStick5");

            // 建立哲学家对象，并在其两边摆放筷子。
            Philosopher philosopher1 = new Philosopher("philosopher1", chopStick1,
                    chopStick2);
            Philosopher philosopher2 = new Philosopher("philosopher2", chopStick2,
                    chopStick3);
            Philosopher philosopher3 = new Philosopher("philosopher3", chopStick3,
                    chopStick4);
            Philosopher philosopher4= new Philosopher("philosopher4", chopStick4,
                    chopStick5);
            Philosopher philosopher5 = new Philosopher("philosopher5", chopStick5,
                    chopStick1);

            // 启动五个线程
            philosopher1.start();
            philosopher2.start();
            philosopher3.start();
```

```
        philosopher4.start();
        philosopher5.start();
            }
        }
```

运行结果如下：

```
philosopher1 has ChopStick1 and wait for ChopStick2
philosopher1 eating
philosopher2 has ChopStick2 and wait for ChopStick3
philosopher2 eating
philosopher5 has ChopStick5 and wait for ChopStick1
philosopher5 eating
philosopher3 has ChopStick3 and wait for ChopStick4
philosopher3 eating
philosopher4 has ChopStick4 and wait for ChopStick5
philosopher4 eating
```

本例中由于采用了干预，避免死锁。如在哲学家问题中，如果规定每个哲学家必须在拿到自己左边的筷子后，才能去拿自己右边的筷子，那么讲很容易形成一个请求环，因此也就可能形成死锁。但如果我们规定其中的某一个哲学家只能在拿到自己右边筷子的前提下，才能去拿左边的筷子，那么就不会形成请求环，从而也不会出现死锁。

【6-3】另一个例子

```
public class AnotherDeadLock {
    public static void main(String[] args) {
        final Object resource1 = "resource1";
        final Object resource2 = "resource2";
        // t1 tries to lock resource1 then resource2
        Thread t1 = new Thread() {
            public void run() {
                // Lock resource 1
                synchronized (resource1) {
                    System.out.println("Thread 1: locked resource 1");

                    try {
                        Thread.sleep(50);
                    } catch (InterruptedException e) {
                    }

                    synchronized (resource2) {
                        System.out.println("Thread 1: locked resource 2");
                    }
                }
            }
        };
```

```
        // t2 tries to lock resource2 then resource1
        Thread t2 = new Thread() {
          public void run() {
            synchronized (resource2) {
              System.out.println("Thread 2: locked resource 2");

              try {
                Thread.sleep(50);
              } catch (InterruptedException e) {
              }
              synchronized (resource1) {
                System.out.println("Thread 2: locked resource 1");
              }
            }
          }
        };

        // If all goes as planned, deadlock will occur,
        // and the program will never exit.
        t1.start();
        t2.start();
      }
    }
```

该例中，对锁作了一个调整，变得更普遍了。本质上和例 6-1 是一致的。程序中存在死锁的问题。

6.3 避免死锁和死锁诊断

一般来说，要出现死锁必须同时具备四个条件。因此，如果能够尽可能地破坏这四个条件中的任意一个，就可以避免死锁的出现。

1） 互斥条件。即至少存在一个资源，不能被多个线程同时共享。如在哲学家问题中，一支筷子一次只能被一个哲学家使用。

2） 至少存在一个线程，它拥有一个资源，并等待获得另一个线程当前所拥有的资源。如在哲学家聚餐问题中，当发生死锁时，至少有一个哲学家拿着一支筷子，并等待取得另一个哲学家拿着的筷子。

3） 线程拥有的资源不能被强行剥夺，只能有线程资源释放。如在哲学家问题中，如果允许一个哲学家之间可以抢夺筷子，则就不会发生死锁问题。

4） 线程对资源的请求形成一个圆环。即：线程 1 拥有资源 1，并等待资源 2，而线程 2 拥有资源 2，并等待资源 3,…,以此类推，最后线程 n 拥有资源 n-1,并等待资源 1，从而构成了一个环。这是构成死锁的一个重要条件。如在哲学家问题中，如果规定每个哲学家必须在拿到自己左边的筷子后，才能去拿自己右边的筷子，那么讲很容易形成一个请求环，因此也就可能形成死锁。但如果我们规定其中的某一个哲学家只能在拿到自己右边筷子的前提下，才能去拿左边的筷子，那么就不会形成请求环，从而也不会出现死锁。

理解了死锁的原因，尤其是产生死锁的四个必要条件，就可以最大可能地避免、预防和解除死锁。所以，在系统设计、进程调度等方面注意如何不让这四个必要条件成立，如何确定资源的合理分配算法，避免进程永久占据系统资源。此外，也要防止线程在处于等待状态的情况下占用资源,在系统运行过程中，对线程发出的每一个系统能够满足的资源申请进行动态检查，并根据检查结果决定是否分配资源，若分配后系统可能发生死锁，则不予分配，否则予以分配 。因此，对资源的分配要给予合理的规划。下面有两种方法可以有效避免死锁。

1）有序资源分配法

这种算法资源按某种规则系统中的所有资源统一编号（例如打印机为 1、磁带机为 2、磁盘为 3、等等），申请时必须以上升的次序。系统要求申请线程：

1、对它所必须使用的而且属于同一类的所有资源，必须一次申请完；

2、在申请不同类资源时，必须按各类设备的编号依次申请。例如：进程 PA，使用资源的顺序是 R1，R2； 进程 PB，使用资源的顺序是 R2，R1；若采用动态分配有可能形成环路条件，造成死锁。

采用有序资源分配法：R1 的编号为 1，R2 的编号为 2；

PA：申请次序应是：R1，R2。

PB：申请次序应是：R1，R2。

这样就破坏了环路条件，避免了死锁的发生。

2）银行算法

避免死锁算法中最有代表性的算法是 Dijkstra E.W 于 1968 年提出的银行家算法：

在系统运行过程中，对线程发出的每一个系统能够满足的资源申请进行动态检查，并根据检查结果决定是否分配资源，若分配后系统可能发生死锁，则不予分配，否则予以分配。

系统安全序列的概念：对于线程序列{P1，…，Pn}是安全的话，如果对于每一个线程

Pi(1≤i≤n)，它以后尚需要的资源量不超过系统当前剩余资源量与所有线程 Pj(j < i)当前占有资源量之和，系统则处于安全状态(安全状态一定是没有死锁发生的)。

银行家算法的中心思想是在安全状态下系统不会进入死锁，不安全状态可能进入死锁。在进行资源分配之前，先计算分配的安全性，判断是否为安全状态。

该算法需要检查申请者对资源的最大需求量，如果系统现存的各类资源可以满足申请者的请求，就满足申请者的请求。

这样申请者就可很快完成其计算，然后释放它占用的资源，从而保证了系统中的所有进程都能完成，所以可避免死锁的发生。

死锁排除的方法

1、撤消陷于死锁的全部线程；

2、逐个撤消陷于死锁的线程，直到死锁不存在；

3、从陷于死锁的线程中逐个强迫放弃所占用的资源，直至死锁消失。

4、从另外一些线程那里强行剥夺足够数量的资源分配给死锁线程，以解除死锁状态

6.4 减小锁的竞争和粒度

竞争性的锁将会导致两种损失：可伸缩性和性能，所以减少锁的竞争能够改进性能和可伸缩性。

访问独占锁守护的资源是串行的——一次只能有一个线程访问它。当然，我们有很好的理由使用锁，比如避免数据过期，但是这样的安全性是用很大的代价换来的。对锁长期的竞争会限制可伸缩性。并发程序中，对可伸缩性首要的威胁是独占的资源锁。

有两个原因影响着锁的竞争性：锁被请求的频率，以及每次持有该锁的时间。如果这两者的乘积足够小，那么大多数请求锁的尝试都是非竞争的，这样竞争性的锁将不会成为可伸缩性巨大的阻碍。但是，如果这个锁的请求量很大，线程将会阻塞以等待锁；在极端的情况下，处理器将会闲置，即使仍有大量工作等着完成。

有 3 种方式来减少锁的竞争：

- 减少持有锁的时间；
- 减少请求锁的频率；
- 用协调机制取代独占锁，从而允许更强的并发性。

6.4.1 缩小锁的范围

减小竞争发生可能性的有效方式是尽可能缩短把持锁的时间。这可以通过把与锁无关的代码移出 synchronized 块来实现，尤其是那些花费“昂贵”的操作，以及那些潜在的阻塞操作，比如 I/O 操作。

我们很容易观察到长时间持有“热门”锁究竟是如何限制可伸缩性的;对于例 6-4 的例子，无论你拥有多少个空闲处理器,如果一个操作持有锁超过 2 毫秒并且每一个操作都需要那个锁，吞吐量不会超过每秒 500 个操作。但是如果减少持有这个锁的时间到 1 毫秒，那将能够把这个与锁相关的吞吐量提高到每秒 1000 个操作。

【6-4】AttributeStore

```
public class AttributeStore {
@GuardedBy("this") private final Map<String, String>
attributes = new HashMap<String, String>();
public synchronized boolean userLocationMatches(String name,
            String regexp) {
  String key = "users." + name + ".location";
  String location = attributes.get(key);
    if (location == null)
    return false;
     else
    return Pattern.matches(regexp, location);
  }
}
```

将例 6-4 的代码改称下面 6-5 的形式

【6-5】BetterAttributeStore

```
public class BetterAttributeStore {
    @GuardedBy("this") private final Map<String, String>
      attributes = new HashMap<String, String>();
     public boolean userLocationMatches(String name, String regexp) {
     String key = "users." + name + ".location";
     String location;
     synchronized (this) {
        location = attributes.get(key);
     }
     if (location == null)
        return false;
      else
     return Pattern.matches(regexp, location);
    }
}
```

缩小 userLocationMatches 方法中锁守护的范围，这大大减少了调用中遇到锁住情况的次数。串行化的代码少了，减少了占有锁的时间。

尽管缩小 synchronized 块能够提高可伸缩性，synchronized 块可以变得极小——需要原子化的操作（比如在限定约束的情况下更新多个变量）必须包含在一个 synchronized 块中。并且因为同步的开销非零，保证正确的情况下，如果把一个 synchronized 块分拆成多个 synchronized 块，在某些时刻反而会对性能产生反作用。

6.4.2 减小锁的粒度

减小持有锁的总体时间比例的另一种方式是让线程减少调用它的频率。可以通过分拆锁（lock splitting）和分离锁（lock striping）来实现，也就是采用相互独立的锁，守卫多个独立的状态变量，在改变之前，它们都是由一个锁守护的。这些技术减小了锁发生时的粒度，潜在实现了更好的可伸缩性。但是使用更多的锁同样会增加死锁的风险。

例如：待分拆锁的候选程序

【6-6】// ServerStatus.java

```
public class ServerStatus {
   @GuardedBy("this") public final Set<String> users;
   @GuardedBy("this") public final Set<String> queries;
   public synchronized void addUser(String u) { users.add(u); }
   public synchronized void addQuery(String q) { queries.add(q); }
   public synchronized void removeUser(String u) {
   users.remove(u);
   }
   public synchronized void removeQuery(String q) {
   queries.remove(q);
   }
}
```

【6-7】拆分后的锁

```
public class ServerStatus {
   @GuardedBy("users") public final Set<String> users;
   @GuardedBy("queries") public final Set<String> queries;
   public void addUser(String u) {
   synchronized (users) {
    users.add(u);
   }
   }
   public void addQuery(String q) {
   synchronized (queries) {
```

```
            queries.add(q);
          }
          }
        }
```

中等竞争强度的锁，能够切实地把它们大部分转化成非竞争的锁，这个结果是性能和可伸缩性都期望得到的。

把一个竞争激烈的锁分拆成两个，很可能形成两个竞争激烈的锁。尽管这可以通过两个线程并发执行，取代一个线程，从而对可伸缩性有一些小的改进，但这仍然不能大幅地提高多个处理器在同一系统中并发性的前景。作为分拆锁的例子，ServerStatus 类并没有提供明显的机会来进行进一步分拆。

分拆锁有时候可以被扩展，分成可大可小加锁块的集合，并且它们归属于相互独立的对象，这样的情况就是分离锁

用于减轻竞争锁带来的影响的第三种技术是提前使用独占锁，这有助于使用更友好的并发方式进行共享状态的管理。这包括使用并发容器、读-写锁、不可变对象，以及原子变量。

读-写锁（ReadWriteLock）实行了一个多读者-单写者（multiple-reader, single-write）加锁规则：只要没有更改，那么多个读者可以并发访问共享资源，但是写者必须独占获得锁。对于多数操作都为读操作的数据结构，ReadWriteLock 与独占的锁相比，可以提供更好的并发性；对于只读的数据结构，不变性可以完全消除加锁的必要。

原子变量（参见第 3 章）提供了能够减少更新“热点域”的方式，如静态计数器、序列发生器、或者对链表数据结构头节点的引用。原子变量类提供了针对整数或对象引用的非常精妙的原子操作（因此更具可伸缩性），并且使用现代处理器提供的低层并发原语，比如比较并交换（compare-and-swap）实现。如果你的类只有少量热点域，并且该类不参与其他变量的不变约束，那么使用原子变量替代它可能会提高可伸缩性。

6.5 使用 MTRAT 诊断死锁

下面为 Mtrat 提供的关于死锁的案例

【6-7】Deadlock.java

```
package mtrat.test;
import java.util.concurrent.atomic.AtomicBoolean;
class T1 extends Thread{
      StringBuffer G;
      StringBuffer L1;
```

```
        StringBuffer L2;
        public T1(StringBuffer G, StringBuffer L1, StringBuffer L2)       {
                this.G = G;
                this.L1 = L1;
                this.L2 = L2;
        }
        public void run()     {
                synchronized (G) {
                        synchronized (L1)   {
                                synchronized (L2)   {
                                        System.out.println("Thread " + Thread.currentThread().getId()
                                                        + " : acquire " + G + ", " + L1 + ", " + L2);
                                }
                        }
                }

                Thread t3 = new T3(L1, L2);
                t3.start();
                System.out.println("Thread " + getId() + " start Thread " + t3.getId());
                try   {
                        t3.join();
                } catch (InterruptedException e){
                        e.printStackTrace();
                }
                System.out.println("Thread " + getId() + " join Thread " + t3.getId());
                synchronized (L2)   {
                        synchronized (L1)   {
                                System.out.println("Thread " + getId() + ": acquire " + L2 + ", " + L1);
                        }
                }
        }
}

class T2 extends Thread{
        StringBuffer G;
        StringBuffer L1;
        StringBuffer L2;
        public T2(StringBuffer G, StringBuffer L1, StringBuffer L2)       {
                this.G = G;
                this.L1 = L1;
                this.L2 = L2;
        }
        public void run()     {
                synchronized (G){
```

```
                synchronized (L2)  {
                    synchronized (L1)  {
                        System.out.println("Thread " + getId() + " : acquire " + G + ", "+
L2 + ", "+ L1);
                    }
                }
            }
        }
    }

    class T3 extends Thread{
        StringBuffer L1;
        StringBuffer L2;
        static AtomicBoolean locked = new AtomicBoolean(false);
        public T3(StringBuffer L1, StringBuffer L2){
            this.L1 = L1;
            this.L2 = L2;
        }

        public void run()    {
            synchronized (L1)  {
                while (!locked.compareAndSet(false, true))
                    ;
                synchronized (L2)  {
                    System.out.println("Thread  "  +  Thread.currentThread().getId()  +  "  :
Acquire " + L1 + " " + L2);
                }
                locked.compareAndSet(true, false);
            }
        }
    }

    public class Deadlock{
        // no deadlock here
        void harness1(){
            StringBuffer G = new StringBuffer("G");
            StringBuffer L1 = new StringBuffer("L1");
            StringBuffer L2 = new StringBuffer("L2");
            Thread t2=new T2(G, L1, L2);
            System.out.println ("T2 starts");
            t2.start();
            try   {
                t2.join();
            } catch (InterruptedException e){
```

```
                    e.printStackTrace();
                }
                System.out.println ("T2 join");
                System.out.println ("T1 starts");
                new T1(G, L1, L2).start();
        }

        void harness2() throws InterruptedException{
                StringBuffer L1 = new StringBuffer("L1");
                StringBuffer L2 = new StringBuffer("L2");
                Thread t1 = new T3(L1, L2);
                Thread t2 = new T3(L2, L1);

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

                t1.join();
                t2.join();
        }
        public static void main(String[] args) throws InterruptedException
        {
                Deadlock dlt = new Deadlock();
                dlt.harness1();
                dlt.harness2();
        }
    }
```

程序运行结果如下：

```
T2 starts
Thread 11 : acquire G, L2, L1
T2 join
T1 starts
Thread 12 : acquire G, L1, L2
Thread 13 : Acquire L1 L2
Thread 12 start Thread 15
Thread 15 : Acquire L1 L2
Thread 12 join Thread 15
Thread 12: acquire L2, L1
Thread 14 : Acquire L2 L1
```

采用 Mtrat 进行分析，在 Eclipse 平台下进行分析，发现了两个潜在的死锁问题。

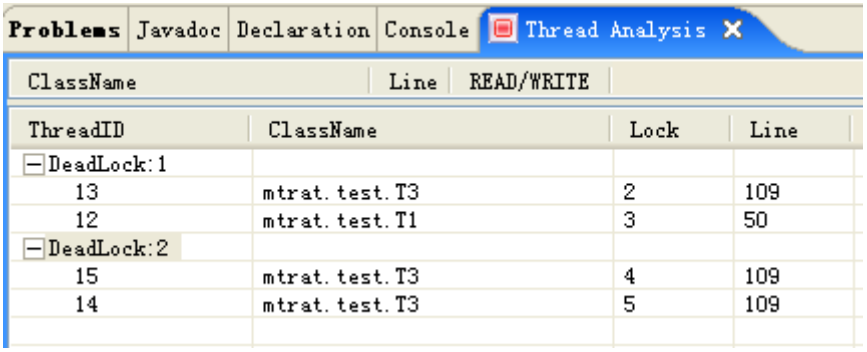


图 6-1 Mtrat 分析死锁的结果

6.6 饿死和活锁

在多线程编程过程，除了可能遇到死锁的情况之外，我们还可能遇到活锁和饿死的情况。

对于死锁来说，由于系统中两个或多个部件的集合发生阻塞，并且每个部件都等待集合中其他部件，从而使计算无法进行；典型的情况下，每个部件是一个被阻塞的线程，它等待集合中其他线程释放所掌握的资源。

什么活锁呢？活锁的产生于循环依赖，当一个线程忙于接受新任务以致它永远没有机会完成任何任务时，就会发生活锁。这个线程最终将超出缓冲区并导致程序崩溃。试想一个秘书需要录入一封信，但她一直在忙于接电话，所以这封信永远不会被录入。如果明智地使用 synchronized 关键字，则完全可以避免内存错误这种气死人的问题。

什么是饿死的概念？饿死的概念和死锁不一样，一些线程不能获得服务，而其他客户端却可以；违反了公平原则。这些不能获得服务的线程即成为饿死的线程。

产生饿死的主要原因是：在一个动态系统中，对于每类系统资源，操作系统需要确定一个分配策略，当多个线程同时申请某类资源时，由分配策略确定资源分配给线程的次序。有时资源分配策略可能是不公平的，即不能保证等待时间上界的存在。在这种情况下，即使系统没有发生死锁，某些线程也可能会长时间等待. 当等待时间给线程推进和响应带来明显影响时，称发生了线程饿死，当饿死到一定程度的线程所赋予的任务即使完成也不再具有实际意义时称该线程被饿死。举个例子，当有多个线程需要打印文件时，如果系统分配打印机的策略是最短文件优先，那么长文件的打印任务将由于短文件的源源不断到来而被无限期推迟，导致最终的饿死甚至饿死。

饿死没有其产生的必要条件，随机性很强。并且饿死可以被消除，因此也将忙式等待时发生的饿死称为活锁。

由于饿死和活锁与资源分配策略有关，因而解决饿死与活锁问题可从资源分配策略的公平性考虑，确保所有线程不被忽视。如时间片轮转算法(RR)。它将 CPU 的处理时间分成一个个时间片，就绪队列中的诸线程轮流运行一个时间片，当时间片结束时，就强迫运行程序让出 CPU，该线程进入就绪队列，等待下一次调度。同时，线程调度又去选择就绪队列中的一个线程，分配给它一个时间片，以投入运行。如此方式轮流调度。这样就可以在不考虑其他系统开销的情况下解决饿死的问题。

最后，我们来比较的看一下死锁与饿死。

死锁与饿死有一定相同点：二者都是由于竞争资源而引起的。但又有明显差别：

(1) 从线程状态考虑，死锁线程都处于等待状态，忙式等待(处于运行或就绪状态)的线程并非处于等待状态，但却可能被饿死；

(2) 死锁线程等待永远不会被释放的资源，饿死线程等待会被释放但却不会分配给自己的资源，表现为等待时限没有上界(排队等待或忙式等待)；

(3) 死锁一定发生了循环等待，而饿死则不然。这也表明通过资源分配图可以检测死锁存在与否，但却不能检测是否有线程饿死；

(4) 死锁一定涉及多个线程，而饿死或被饿死的线程可能只有一个。

(5)在饿死的情形下，系统中有至少一个线程能正常运行，只是饿死线程得不到执行机会。而死锁则可能会最终使整个系统陷入死锁并崩溃。

参考资料：

1) http://www.cqzol.com/programming/Java/200801/83962.html
2) http://book.csdn.net/bookfiles/398/index.html

第七章 显示锁

第七章 显示锁.....1
7.1. Lock和ReentrantLock.....2
7.2. 对性能的考察.....4
7.3 Lock与Condition.....8
7.4. 在内部锁和重入锁之间进行选择.....13
7.5. 读-写锁.....14
参考文献.....21

相对于以前的版本，Java 5.0 引入了新的调节共享对象访问的机制，即重入锁（ReentrantLock）。重入锁可以在内部锁被证明受到局限时，提供可选择的高级特性。它具有与内在锁相同的内存语义、相同的锁定，但在争用条件下却有更好的性能。

同时提供了读写锁，与互斥锁相比，读取数据远大于修改数据的频率时能提升性能。

在第 3 章讲解 JDK 并发 API 时已经介绍过 ReentrantLock，本章做一些提升和补充。

7.1. Lock 和 ReentrantLock

Lock 接口定义了一组抽象的锁定操作。与内部锁定（intrinsic locking）不同，Lock 提供了无条件的、可轮询的、定时的、可中断的锁获取操作，所有加锁和解锁的方法都是显式的。这提供了更加灵活的加锁机制，弥补了内部锁在功能上的一些局限——**不能中断那些正在等待获取锁的线程，并且在请求锁失败的情况下，必须无限等待**。

Lock 接口主要定义了下面的一些方法：

1）void lock()：获取锁。如果锁不可用，出于线程调度目的，将禁用当前线程，并且在获得锁之前，该线程将一直处于休眠状态。

2）void lockInterruptibly() throws InterruptedException：如果当前线程未被中断，则获取锁。如果锁可用，则获取锁，并立即返回。如果当前线程在获取锁时被中断，并且支持对锁获取的中断，则将抛出 InterruptedException，并清除当前线程的已中断状态。

3）boolean tryLock()：如果锁可用，则获取锁，并立即返回值 true。如果锁不可用，则此方法将立即返回值 false。

4）boolean tryLock(long time, TimeUnit unit) throws InterruptedException：如果锁在给定的等待时间内空闲，并且当前线程未被中断，则获取锁。

5）void unlock()：释放锁。

6）Condition newCondition()：返回绑定到此 Lock 实例的新 Condition 实

例。调用 Condition.await() 将在等待前以原子方式释放锁，并在等待返回前重新获取锁。

ReentrantLock 实现了 Lock 接口。获得 ReentrantLock 的锁与进入 synchronized 块具有相同的语义，释放 ReentrantLock 锁与退出 synchronized 块有相同的语义。相比于 synchronized，ReentrantLock 提供了更多的灵活性来处理不可用的锁。下面具体来介绍一下 ReentrantLock 的使用。

1. 实现可轮询的锁请求

在内部锁中，死锁是致命的——唯一的恢复方法是重新启动程序，唯一的预防方法是在构建程序时不要出错。而可轮询的锁获取模式具有更完善的错误恢复机制，可以规避死锁的发生。

如果你不能获得所有需要的锁，那么使用可轮询的获取方式使你能够重新拿到控制权，它会释放你已经获得的这些锁，然后再重新尝试。可轮询的锁获取模式，由 tryLock()方法实现。此方法仅在调用时锁为空闲状态才获取该锁。如果锁可用，则获取锁，并立即返回值 true。如果锁不可用，则此方法将立即返回值 false。此方法的典型使用语句如下：

```
Lock lock = ...;
if (lock.tryLock()) {
    try {
        // manipulate protected state
    } finally {
        lock.unlock();
    }
} else {
    // perform alternative actions
}
```

2. 实现可定时的锁请求

当使用内部锁时，一旦开始请求，锁就不能停止了，所以内部锁给实现具有时限的活动带来了风险。为了解决这一问题，可以使用定时锁。当具有时限的活

动调用了阻塞方法，定时锁能够在时间预算内设定相应的超时。如果活动在期待的时间内没能获得结果，定时锁能使程序提前返回。可定时的锁获取模式，由tryLock(long, TimeUnit)方法实现。

3. 实现可中断的锁获取请求

可中断的锁获取操作允许在可取消的活动中使用。lockInterruptibly()方法能够使你获得锁的时候响应中断。

7.2. 对性能的考察

当 ReentrantLock 被加入到 Java 5.0 中时，它提供的性能要远远优于内部锁。如果有越多的资源花费在锁的管理和调度上，那用留给应用程序的就会越少。更好的实现锁的方法会使用更少的系统调用，发生更少的上下文切换，在共享的内存总线上发起更少的内存同步通信。耗时的操作会占用本应用于程序的资源。Java 6 中使用了经过改善的管理内部锁的算法，类似于 ReentrantLock 使用的算法，从而大大弥补了可伸缩性的不足。因此 ReentrantLock 与内部锁之间的性能差异，会随着 CPU、处理器数量、高速缓存大小、JVM 等因素的发展而改变。

下面具体的构造一个测试程序来具体考察 ReentrantLock 的性能。构造一个计数器 Counter，启动 N 个线程对计数器进行递增操作。显然，这个递增操作需要同步以防止数据冲突和线程干扰，为保证原子性，采用 3 种锁来实现同步，然后查看结果。

测试环境是双核酷睿处理器，内存 3G，JDK6。

第一种是内在锁，第二种是不公平的 ReentrantLock 锁，第三种是公平的 ReentrantLock 锁。

首先定义一个计数器接口。

```
package locks;
public interface Counter {
   public long getValue();
   public void increment();
}
```

下面是使用内在锁的计数器类：

```
package lockbenchmark;
public class SynchronizedCounter implements Counter {
    private long count = 0;
    public long getValue() {
        return count;
    }
    public synchronized void increment() {
        count++;
    }
}
```

下面是使用不公平 ReentrantLock 锁的计数器。

```
package lockbenchmark;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
public class ReentrantUnfairCounterLockCounter implements Counter {
    private volatile long count = 0;
    private Lock lock;
    public ReentrantUnfairCounterLockCounter() {
        // 使用非公平锁，true就是公平锁
        lock = new ReentrantLock(false);
    }
    public long getValue() {
        return count;
    }
    public void increment() {
        lock.lock();
        try {
            count++;
        } finally {
            lock.unlock();
        }
    }
}
```

下面是使用公平的 ReentrantLock 锁的计数器。

```
package lockbenchmark;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
public class ReentrantFairLockCounter implements Counter {
```

```
    private volatile long count = 0;
    private Lock lock;
    public ReentrantFairLockCounter() {
        // true就是公平锁
        lock = new ReentrantLock(true);
    }
    public long getValue() {
        return count;
    }
    public void increment() {
        lock.lock();
        try {
            count++;
        } finally {
            lock.unlock();
        }
    }
}
```

下面是总测试程序。

```
package lockbenchmark;
import java.util.concurrent.CyclicBarrier;
public class BenchmarkTest {
    private Counter counter;
    // 为了统一启动线程，这样好计算多线程并发运行的时间
    private CyclicBarrier barrier;
    private int threadNum;// 线程数
    private int loopNum;// 每个线程的循环次数
    private String testName;
    public BenchmarkTest(Counter counter, int threadNum, int loopNum,
            String testName) {
        this.counter = counter;
        barrier = new CyclicBarrier(threadNum + 1); // 关卡计数=线程数
+1
        this.threadNum = threadNum;
        this.loopNum = loopNum;
        this.testName = testName;
    }
    public static void main(String args[]) throws Exception {
        int threadNum = 2000;
        int loopNum = 1000;
        new BenchmarkTest(new SynchronizedCounter(), threadNum,
loopNum, "内部锁")
                .test();
```

```
        new BenchmarkTest(new ReentrantUnfairCounterLockCounter(),
threadNum,
                loopNum, "不公平重入锁").test();
        new BenchmarkTest(new ReentrantFairLockCounter(), threadNum,
loopNum,
                "公平重入锁").test();
    }
    public void test() throws Exception {
        try {
            for (int i = 0; i < threadNum; i++) {
                new TestThread(counter, loopNum).start();
            }
            long start = System.currentTimeMillis();
            barrier.await(); // 等待所有任务线程创建,然后通过关卡，统一执行
所有线程
            barrier.await(); // 等待所有任务计算完成
            long end = System.currentTimeMillis();
            System.out.println(this.testName + " count value:"
                    + counter.getValue());
            System.out.println(this.testName + " 花费时间:" + (end -
start) + "毫秒");
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
    class TestThread extends Thread {
        int loopNum = 100;
        private Counter counter;
        public TestThread(final Counter counter, int loopNum) {
            this.counter = counter;
            this.loopNum = loopNum;
        }
        public void run() {
            try {
                barrier.await();// 等待所有的线程开始
                for (int i = 0; i < this.loopNum; i++)
                    counter.increment();
                barrier.await();// 等待所有的线程完成
            } catch (Exception e) {
                throw new RuntimeException(e);
            }
        }
    }
}
```

对三种锁分别设置两个不同的参数：不同线程数和每个线程数的循环次数。最后记录每种锁的运行时间（单位：ms），形成下表。

循环次数 1000

循环次数 1000	线程数 200	线程数 500	线程数 1000	线程数 2000
内在锁	62	313	406	875
非公平锁	31	94	250	859
公平锁	4641	17610	44671	57391

循环次数 200

循环次数 200	线程数 200	线程数 500	线程数 1000	线程数 2000
内在锁	47	94	109	265
非公平锁	16	32	125	906
公平锁	781	3031	8671	13625

分析统计结果，在线程数小于 2000 的情况下，非公平可重入锁的性能要优于内部锁。公平可重入锁的性能最差。同时发现内部锁其实也是一个非公平锁。

7.3 Lock 与 Condition

当使用 synchronized 进行同步时，可以在同步代码块中使用 wait 和 notify 等方法。

在使用显示锁的时候，通过将 Condition 对象与任意 Lock 实现组合使用，为每个对象提供多个等待方法。其中，Lock 替代了 synchronized 方法和语句的使用，Condition 替代了 Object 监视器方法的使用。

条件（Condition，也称为条件队列或条件变量）为线程提供了一个含义，以便在某个状态条件现在可能为 true、另一个线程通知它之前，一直挂起该线程（即让其“等待”）。因为访问此共享状态信息发生在不同的线程中，所以它必须受保

护，因此要将某种形式的锁与该条件相关联。等待提供一个条件的主要属性是：以原子方式释放相关的锁，并挂起当前线程，就像 Object.wait 做的那样。

Condition 实例实质上被绑定到一个锁上。要为特定 Lock 实例获得 Condition 实例，请使用其 newCondition() 方法。

下面使用可重入锁与 Condition 替代 synchronized 实现生产者-消费者模式。

生产者-消费者问题一般是，有一个缓冲区，它支持 put 和 take 方法。如果试图在空的缓冲区上执行 take 操作，则在某一个项变得可用之前，线程将一直阻塞；如果试图在满的缓冲区上执行 put 操作，则在有空间变得可用之前，线程将一直阻塞。可以在单独的等待集合中保存 put 线程和 take 线程，这样就可以在缓冲区中的项或空间变得可用时利用最佳规划，一次只通知一个线程。可以使用两个 Condition 实例来做到这一点。

下面是缓冲区类 LockedBuffer，在这个类的 put 和 take 方法中使用了可重入锁与条件变量：

```
package conditionlock;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
public class LockedBuffer {
    // 可重入锁
    final Lock lock = new ReentrantLock();
    // 两个条件对象
    final Condition notFull = lock.newCondition();
    final Condition notEmpty = lock.newCondition();
    // 缓冲区
    final Object[] items = new Object[10];
    int putptr, takeptr, count;// 计数器
    // 放数据操作，生产者调用该方法
    public void put(Object x) throws InterruptedException {
        lock.lock();
        try {
            // 如果缓冲区满了，则线程等待
            while (count == items.length)
                notFull.await();
            items[putptr] = x;
            if (++putptr == items.length)
                putptr = 0;
            ++count;
            // 向消费者线程发送通知
```

```
                notEmpty.signal();
            } finally {
                lock.unlock();
            }
        }
        // 消费者线程调用该方法
        public Object take() throws InterruptedException {
            lock.lock();
            try {
                // 如果缓冲区空，则等待
                while (count == 0)
                    notEmpty.await();
                Object x = items[takeptr];
                if (++takeptr == items.length)
                    takeptr = 0;
                --count;
                // 通知其他生产者线程
                notFull.signal();
                return x;
            } finally {
                lock.unlock();
            }
        }
    }
```

生产者：

```
package conditionlock;
//生产者
class Producer implements Runnable {
    LockedBuffer buffer;
    public Producer(LockedBuffer buf) {
        buffer = buf;
    }
    public void run() {
        char c;
        for (int i = 0; i < 20; i++) {
            c = (char) (Math.random() * 26 + 'A');
            try {
                // 向缓冲区放入数据
                buffer.put(c);
            } catch (InterruptedException e1) {
                e1.printStackTrace();
            }
            System.out.println("Produced: " + c);
```

```
            try {
                Thread.sleep((int) (Math.random() * 100));
            } catch (InterruptedException e) {
            }
        }
    }
}
```

消费者类

```
package conditionlock;
//消费者
class Consumer implements Runnable {
    LockedBuffer buffer;
    public Consumer(LockedBuffer buf) {
        buffer = buf;
    }
    public void run() {
        Object c = null;
        for (int i = 0; i < 20; i++) {
            try {
                // 取数据
                c = buffer.take();
            } catch (InterruptedException e1) {
                e1.printStackTrace();
            }
            System.out.println("Consumed: " + c);
            try {
                Thread.sleep((int) (Math.random() * 1000));
            } catch (InterruptedException e) {
            }
        }
    }
}
```

测试类

```
package conditionlock;
public class LockConditionTest {
    public static void main(String args[]) {
        LockedBuffer stack = new LockedBuffer();
        // 创建生产者，消费者
        int count = 3;
        Producer[] producers = new Producer[count];
        Consumer[] consumers = new Consumer[count];
        for (int i = 0; i < count; i++) {
            producers[i] = new Producer(stack);
```

```
            consumers[i] = new Consumer(stack);
        }
        for (int i = 0; i < count; i++) {
            new Thread(producers[i]).start();
            new Thread(consumers[i]).start();
        }
    }
}
```

程序运行结果如下：

```
Produced: Z
Consumed: Z
Produced: X
Consumed: X
…..
Produced: D
Produced: N
Produced: L
Produced: U
Produced: G
Produced: V
Consumed: Q
Produced: Q
Produced: U
Consumed: M
Produced: I
Consumed: D
….
Consumed: U
Produced: M
Consumed: G
Produced: P
Consumed: V
```

```
Produced: N
Consumed: Q
Produced: J
Consumed: U
Produced: L
……
Produced: Y
Consumed: O
Produced: E
Consumed: M
Produced: I
Consumed: P
```

7.4. 在内部锁和重入锁之间进行选择

重入锁（ReentrantLock）与内部锁在加锁和内存语义上是相同的。从性能上看，重入锁的性能看起来胜过内部锁。在 Java 5.0 中，两者性能之间的差距比较大；而在 Java 6 中，这种差距变得比较小。与重入锁相比，内部锁仍然具有很大的优势，比如内部锁更为人们所熟悉，也更简洁，而且很多现有的程序已经在使用内部锁了。重入锁是很危险的同步工具，程序员在使用重入锁时，容易产生错误。因此，**只有在内部锁不能满足需求，才需要使用重入锁。**

在 Java 5.0 中，内部锁还具有另外一个优点：线程转储能够显示哪些调用框架获得了哪些锁，并能够识别发生了死锁的线程。但 Java 虚拟机并不知道哪个线程持有重入锁，因此在调试使用了重入锁的线程时，无法从中获得帮助信息。这个问题在 Java 6 中得到了解决，它提供了一个管理和调试接口，锁可以使用这个接口进行注册，并通过其他管理和调试接口，从线程转储中得到重入锁的加锁信息。

由于内部锁是内置于 Java 虚拟机中的，它能够进行优化，因此未来的性能改进可能更倾向于内部锁，而不是重入锁。综上所述，除非你的应用程序需要发

布在 Java 5.0 上，或者需要使用重入锁的可伸缩性，否则就应该选择内部锁。

总之，ReentrantLock 锁与 Java 内在锁相比有下面的特点：

1)ReentrantLock 必须在 finally 块中释放锁，而使用 synchronized 同步，JVM 将确保锁会获得自动释放。

2）与目前的 synchronized 实现相比，争用下的 ReentrantLock 实现更具可伸缩性。

3）对于 ReentrantLock ，可以有不止一个条件变量与它关联。

4）允许选择想要一个公平锁，还是一个不公平锁。

5）除非您对 Lock 的某个高级特性有明确的需要，或者有明确的证据表明在特定情况下，同步已经成为可伸缩性的瓶颈，否则还是应当继续使用 synchronized。

6）Lock 类只是普通的类，JVM 不知道具体哪个线程拥有 Lock 对象。而且，几乎每个开发人员都熟悉 synchronized，它可以在 JVM 的所有版本中工作。

7.5. 读-写锁

读-写锁可以提高应用程序的并发度。在很多情况下，数据是“频繁被读取”的——它们是可变的，有的时候会被改变，但多数访问只进行读操作。此时，如果能够允许多个读线程同时访问数据就非常好了。而标准的互斥锁一次最多只允许一个线程能够持有相同的锁，这为保护数据一致性加了很强的约束，因此过分地限制了并发性。互斥是保守的加锁策略，避免了“写/写”和“写/读”的重叠，但是同样避开了“读/读”的重叠。只要每个线程保证能够读到最新的数据，并且在读线程读取数据的时候没有其他线程修改数据，就不会发生问题。读-写锁允许的情况是：一个资源能够被多个读线程访问，或者被一个写线程访问，但两者不能同时进行。读-写锁的定义如表所示。

表 7.1　ReadWriteLock 接口

```
public interface ReadWriteLock{
    Lock readLock();//返回用于读取操作的锁
    Lock writeLock();//返回用于写入操作的锁。
}
```

引入读-写锁是用来进行性能改进的，使得在特定情况下能够有更好的并发性。在实践中，当多处理器系统中频繁的访问主要是读取数据的时候，读-写锁能够改进性能；在其他情况下，运行的情况比互斥锁要稍差一些，这归因于读-写锁更大的复杂性。使用读-写锁究竟能否带来改进，最好通过对系统进行剖析来判断。

读写锁一般可用于缓存设计。

ReadWriteLock 维护了一对相关的锁，一个用于只读操作，另一个用于写入操作。只要没有 writer，读取锁可以由多个 reader 线程同时保持。写入锁是独占的。

在内容管理系统、新闻发布系统等网站的开发设计中，文档的分类（ArticleCategory）一般是极少变化的，并且数据量比较小，读取非常的频繁，可以一次性的把这些文档分类数据从数据库中一次读取出，放入缓存，减少数据库服务器的压力，当数据有变化时，则使缓存失效，然后从新从数据库读取数据。

未使用缓存时，主要包含下面的几个类：

ArticleCategory：表示文档分类的实体类。

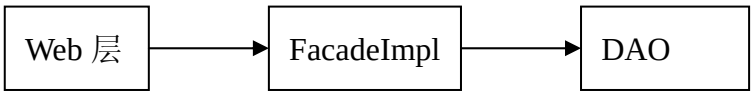
CategoryDao：定义访问数据库操作的接口。

CategoryDaoImpl：具体访问数据库操作的类，实现了 CategoryDao 接口。

Façade：定义了可以使用的业务方法的接口。

FacadeImpl：实现了 Façade 接口中的方法。

基本工作流程是：客户程序调用 Façade 中定义的业务方法 a，业务方法 a 如果需要访问数据库，调用 CategoryDao 中定义的访问数据库的方法，DAO 中定义的是操作 ArticleCategory 实体的方法。



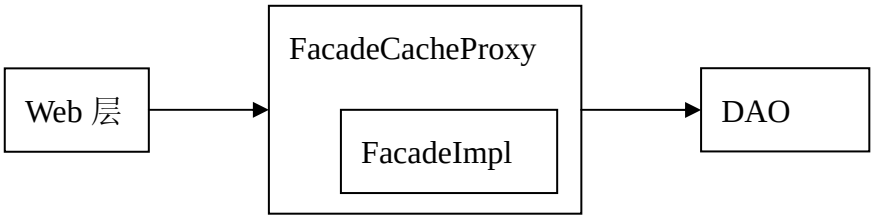
在原有系统基础上进行改造，增加对缓存的支持：

FullCache：缓存类，管理缓存数据。

FacadeCacheProxy：实现了 Façade 接口，其方法的实现又委托给 FacadeImpl 去完成。实现了代理设计模式。

FullCacheTest：缓存程序测试类。

因为 FacadeCacheProxy 也实现了 Façade 接口，使用缓存后，客户调用业务方法的代码无需改变。这样客户程序无需修改。FacadeCacheProxy 中关于读取文档分类的方法直接从缓存读取，执行其他需要更新数据库的方法时，使缓存失效，从新读取数据库更新后的数据填充缓存。



下面是主要类的代码，详细程序请参考光盘上的代码。

下面是 DAO 类，具体负责访问数据库的操作，使用了 Thread.sleep(1)模拟操作数据库的时延。

```
package lockcache;
import java.io.Serializable;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
//DAO 实现类负责具体的访问数据库操作
public class CategoryDaoImpl implements CategoryDao {
    // 这里用内存的一个 HashMap 对象模拟数据库
    private static Map db = new HashMap();
    @Override
    public void create(ArticleCategory category) {
        // 暂停一毫秒模拟数据库的访问时间
        try {
            Thread.sleep(1);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        db.put(category.getId(), category);
    }
    @Override
    public List queryArticleCategories() {
        System.out.println("从数据库读取数据！");
```

```
            try {
                Thread.sleep(1);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            return new ArrayList(db.values());
        }
        @Override
        public ArticleCategory queryArticleCategory(Serializable id) {
            if (db.containsKey((id))) {
                try {
                    Thread.sleep(1);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                return (ArticleCategory) db.get(id);
            }
            return null;
        }
        @Override
        public void update(ArticleCategory category) {
            try {
                Thread.sleep(1);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            db.put(category.getId(), category);
        }
    }
```

管理缓存的类。

```
    package lockcache;
    import java.util.List;
    import java.util.concurrent.locks.Lock;
    import java.util.concurrent.locks.ReadWriteLock;
    import java.util.concurrent.locks.ReentrantReadWriteLock;
    //管理缓存的类
    public abstract class FullCache {
        // 读写锁
        private final ReadWriteLock lock = new ReentrantReadWriteLock();
        private final Lock readLock = lock.readLock(); // 读锁
        private final Lock writeLock = lock.writeLock(); // 写锁
        private List cachedList = null; // 持有缓存的数据，若为 null，表示缓存失
效
        public List getCachedList() {
```

```
            // 获得读锁:
            readLock.lock();
            System.out.println("读取缓存数据！");
            try {
                if (cachedList == null) {
                    // 在获得写锁前，必须先释放读锁:
                    readLock.unlock();
                    writeLock.lock();
                    try {
                        System.out.println("重新填充缓存数据！");
                        cachedList = doGetList(); // 获取真正的数据
                    } finally {
                        // 在释放写锁前，先获得读锁:
                        readLock.lock();
                        writeLock.unlock();
                    }
                }
                return cachedList;
            } finally {
                // 确保读锁在方法返回前被释放:
                readLock.unlock();
            }
        }
        // 子类重写该法，实现具体的获取数据填充缓存的方式
        abstract protected List doGetList();
        public void clearCache() {
            writeLock.lock();
            cachedList = null;
            System.out.println("缓存失效！");
            writeLock.unlock();
        }
    }
```

代理类

```
    package lockcache;
    import java.io.Serializable;
    import java.util.List;
    public class FacadeCacheProxy implements Facade {
        private Facade target;
        public void setFacadeTarget(Facade target) {
            this.target = target;
        }
        private FullCache cache = new FullCache() {
```

```
            // 实现了父类中定义的填充缓存数据的方法
            protected List doGetList() {
                return target.queryArticleCategories();
            }
        };
        public List queryArticleCategories() {
            return cache.getCachedList();
        }
        public void createArticleCategory(ArticleCategory category) {
            target.createArticleCategory(category);
            cache.clearCache();
        }
        public void updateArticleCategory(ArticleCategory category) {
        }
        @Override
        public ArticleCategory queryArticleCategory(Serializable id) {
            return null;
        }
        public void setCategoryDao(CategoryDao categoryDao) {
        }
    }
```

下面是总的缓存测试程序，定义了读取线程和写线程，其中读取频率要远大于写的频率。

```
package lockcache;
//测试缓存的类
public class FullCacheTest {
    // 定义Facade的变量，便于调用业务方法
    static Facade facade = new FacadeImpl();
    // 实现了缓存的Facade
    static Facade proxy = new FacadeCacheProxy();
    static CategoryDao dao = new CategoryDaoImpl();
    public static void main(String[] args) {
        // 设置需要的DAO
        facade.setCategoryDao(dao);
        // 把业务功能委托给FacadeImpl类
        ((FacadeCacheProxy) proxy).setFacadeTarget(facade);
        // 创建更新分类的线程
        Thread t1 = new Thread(new Updater());
        t1.start();
        // 创建读取分类数据的线程
        for (int i = 0; i < 5; i++) {
            new Thread(new Querier(), "t+i").start();
        }
```

```
    }
    // 更新数据的线程体
    static class Updater implements Runnable {
        @Override
        public void run() {
            for (;;) {
                ArticleCategory category = new ArticleCategory();
                double d = Math.random();
                category.setId("" + (int) (d * 10));
                category.setName("name" + d);
                // 创建一个
                proxy.createArticleCategory(category);
                try {
                    Thread.sleep(10);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
    // 读取数据的线程体
    static class Querier implements Runnable {
        @Override
        public void run() {
            for (;;) {
                // 打印读取的数据
                System.out.println(proxy.queryArticleCategories());
                try {
                    Thread.sleep(1);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

下面是程序运行结果的片段：

```
.....
读取缓存数据！
[3/name0.32385907946157355, 2/name0.21873239178259907,
1/name0.10581430507095557, 0/name0.048790763893907685,
```

7/name0.7092783397751578, 6/name0.6450949775582443, 5/name0.5813937994250811, 4/name0.44179702332968607, 9/name0.9571487587444742, 8/name0.8377456665152162]

读取缓存数据!

[3/name0.32385907946157355, 2/name0.21873239178259907, 1/name0.10581430507095557, 0/name0.048790763893907685, 7/name0.7092783397751578, 6/name0.6450949775582443, 5/name0.5813937994250811, 4/name0.44179702332968607, 9/name0.9571487587444742, 8/name0.8377456665152162]

缓存失效!

读取缓存数据!

重新填充缓存数据!

从数据库读取数据!

[3/name0.32385907946157355, 2/name0.21873239178259907, 1/name0.16459410374370664, 0/name0.048790763893907685, 7/name0.7092783397751578, 6/name0.6450949775582443, 5/name0.5813937994250811, 4/name0.44179702332968607, 9/name0.9571487587444742, 8/name0.8377456665152162]

读取缓存数据!

[3/name0.32385907946157355, 2/name0.21873239178259907, 1/name0.16459410374370664, 0/name0.048790763893907685, 7/name0.7092783397751578, 6/name0.6450949775582443, 5/name0.5813937994250811, 4/name0.44179702332968607, 9/name0.9571487587444742, 8/name0.8377456665152162]

……

参考文献

1. JDK 6 API 文档
2. http://www.cnblogs.com/kylindai/archive/2006/01/24/322667.html

第八章 原子变量与非阻塞算法

第八章 原子变量与非阻塞算法.....1
8.1. 锁的劣势.....2
8.2. 原子变量类.....2
8.3. 非阻塞算法.....5
参考文献.....8

本章首先分析锁的劣势，然后分析原子变量类和非阻塞算法的优势。本章内容与第 3 章和第 4 章内容，紧密相关。相关内容情况参考前述章节。

8.1. 锁的劣势

从前面的章节可以看到，使用一致的加锁协议来协调对共享状态的访问，确保当线程持有守护变量的锁时，线程都能独占地访问这些变量，并且保证随后获得同一锁的线程都能看见该线程对变量所作的修改。

Java 虚拟机能够对非竞争锁的获取和释放进行优化，让它们非常高效，但是如果有多个线程同时请求锁，Java 虚拟机就需要向操作系统寻求帮助。倘若了出现这种情况，一些线程将可能被挂起，并稍后恢复运行。从线程开始恢复，到它真正被调度前，可能必须等待其他线程完成它们的调度限额规定的时问。挂起和恢复线程会带来很大的开销，并通常伴有冗长的中断。对于基于锁，并且其操作过度细分的类（比如同步容器类，大多数方法只包含很少的操作），当频繁地发生锁的竞争时，调度与真正用于工作的开销间的比值会很可观。

加锁还有其他的缺点。当一个线程正在等待锁时，它不能做任何其他事情。如果一个线程在持有锁的情况下发生了延迟（原因包括页错误、调度延迟，或者类似情况），那么其他所有需要该锁的线程都不能前进了。如果阻塞的线程是优先级很高的线程，持有锁的线程优先级较低，那么会造成性能风险，被称为优先级倒置（priority inversion）。即虽然更高的优先级占先，但它仍然需要等待锁被释放，这导致它的优先级会降至与优先级较低的线程相同的水平。如果持有锁的线程发生了永久性的阻塞（因为无限循环、死锁、活锁和其他活跃度失败），所有等待该锁的线程都不会前进了。

即使忽略上述的风险，加锁对于小的操作而言，仍然是重量级（heavy weight）的机制，比如自增操作。需要有更好的技术用来管理线程之问的竞争。在 Java 5.0 中，使用原子变量类（atomic variable classes）能够高效地构建非阻塞算法。

8.2. 原子变量类

在 JDK 5.0 之前，如果不使用本机代码，就不能用 Java 语言编写无等待、

无锁定的算法。在 java.util.concurrent 中添加原子变量类之后，这种情况发生了变化。本节了解这些新类开发高度可伸缩的无阻塞算法。

java.util.concurrent.atomic 包中添加原子变量类。所有原子变量类都公开“比较并设置”原语（与比较并交换类似），这些原语都是使用平台上可用的最快本机结构（比较并交换、加载链接/条件存储，最坏的情况下是旋转锁）来实现的。

原子变量类共有 12 个，分成 4 组：计量器、域更新器（field updater）、数组以及复合变量。最常用的原子变量是计量器：AtomicInteger、AtomicLong、AtomicBoolean 以及 AtomicReference。他们都支持 CAS（比较并设置,详细参考第 3 章）；AtomicInteger 和 AtomicLong 还支持算术运算。

原子变量类可以认为是 volatile 变量的泛化，它扩展了 volatile 变量的概念，来支持原子条件的比较并设置更新。读取和写入原子变量与读取和写入对 volatile 变量的访问具有相同的存取语义。

虽然原子变量类表面看起来与 SynchronizedCounter 例子（参考 3.2.1 节）一样，但相似仅是表面的。在表面之下，原子变量的操作会变为平台提供的用于并发访问的硬件原语，比如比较并交换。

调整具有竞争的并发应用程序的可伸缩性的通用技术是降低使用的锁对象的粒度，希望更多的锁请求从竞争变为不竞争。从锁转换为原子变量可以获得相同的结果，通过切换为更细粒度的协调机制，竞争的操作就更少，从而提高了吞吐量。

下面的程序是使用原子变量后的计数器：

```
package jdkapidemo;
import java.util.concurrent.atomic.AtomicInteger;
public class AtomicCounter {
   private AtomicInteger value = new AtomicInteger();
   public int getValue() {
      return value.get();
   }
   public int increment() {
      return value.incrementAndGet();
   }
   public int increment(int i) {
```

```
        return value.addAndGet(i);
    }
    public int decrement() {
        return value.decrementAndGet();
    }
    public int decrement(int i) {
        return value.addAndGet(-i);
    }
}
```

下面写一个测试类：

```
package jdkapidemo;
public class AtomicCounterTest extends Thread {
    AtomicCounter counter;
    public AtomicCounterTest(AtomicCounter counter) {
        this.counter = counter;
    }
    @Override
    public void run() {
        int i = counter.increment();
        System.out.println("generated number:" + i);
    }
    public static void main(String[] args) {
        AtomicCounter counter = new AtomicCounter();
        for (int i = 0; i < 10; i++) {//10个线程
            new AtomicCounterTest(counter).start();
        }
    }
}
```

运行结果如下：

```
generated number:1
generated number:2
generated number:3
generated number:4
generated number:5
```

```
generated number:7
generated number:6
generated number:9
generated number:10
generated number:8
```

会发现 10 个线程运行中，没有重复的数字，原子量类使用本机 CAS 实现了值修改的原子性。

8.3. 非阻塞算法

基于锁的算法会带来一些活跃度失败的风险。譬如，如果线程在持有锁的时候因为阻塞 I/O、页面错误、或其他原因发生延迟，很可能所有线程都不能前进。一个线程的失败或挂起不应该影响其他线程的失败或挂起，这样的算法被称为非阻塞（non-blocking）算法。如果算法的每一步骤中都有一些线程能够继续执行，那么这样的算法称为无锁（lock-free）算法。非阻塞算法对死锁和优先级倒置有“免疫性”。好的非阻塞算法已经应用到多种常见的数据结构上，包括栈、队列、优先级队列、哈希表。

在实现相同功能的前提下，非阻塞算法往往比基于锁的算法更加复杂。创建非阻塞算法的前提是为了维护数据的一致性，解决如何把原子化范围缩小到一个唯一变量。在链式容器类（比如队列）中，有时你可以把它变为对单独链接的修改；进而，你可以使用一个 AtomicReference 来表达每一个必须被原子更新的链接。

非阻塞算法相对于基于锁的算法有几个性能优势。首先，它用硬件的原生形态代替 Java 虚拟机的锁定代码路径，从而在更细的粒度层次上（独立的内存位置）进行同步，失败的线程也可以立即重试，而不会被挂起后重新调度。更细的粒度降低了争用的机会，不用重新调度就能重试的能力也降低了争用的成本。即使有少量失败的 CAS 操作，这种方法仍然会比由于锁竞争造成的重新调度快得多。

下面给出一个非阻塞堆栈的示例代码：

```
public class ConcurrentStack<E> {
```

```
    AtomicReference<Node<E>> head = new
                  AtomicReference<Node<E>>();
    public void push(E item) {
        Node<E> newHead = new Node<E>(item);
        Node<E> oldHead;
        do {
            oldHead = head.get();
            newHead.next = oldHead;
        } while (!head.compareAndSet(oldHead, newHead));
    }
    public E pop() {
       Node<E> oldHead;
       Node<E> newHead;
       do {
           oldHead = head.get();
           if (oldHead == null)
              return null;
              newHead = oldHead.next;
       } while (!head.compareAndSet(oldHead,newHead));
           return oldHead.item;
    }
    static class Node<E> {
        final E item;
        Node<E> next;
        public Node(E item) { this.item = item; }
    }
}
```

对于上面代码中的 ConcurrentStack 中的 push()和 pop()操作，其所做的工作

有些冒险，希望在“提交”工作的时候，底层假设没有失效。push()方法观察当前的栈顶节点，并构建一个新节点放在堆栈上，然后，观察最顶端的节点在初始之后有没有变化，如果没有变化，那么就安装新节点。如果 CAS 失败，意味着另一个线程已经修改了堆栈，那么过程就会重新开始。

所有非阻塞算法的一个基本特征是：有些算法步骤的执行是要冒险的，因为假如 CAS 不成功，可能不得不重做。非阻塞算法通常叫做乐观算法，因为它们继续操作的假设是不会有干扰；假如发现干扰，就会回退并重试。

在轻度到中度的争用情况下，非阻塞算法的性能会超越阻塞算法，因为 CAS 的多数时间都在第一次尝试时就成功，而发生争用时的开销也不涉及线程挂起和上下文切换，只多了几个循环迭代。没有争用的 CAS 要比没有争用的锁便宜得多（这句话肯定是真的，因为没有争用的锁涉及 CAS 加上额外的处理），而争用的 CAS 比争用的锁获取涉及更短的延迟。在高度争用的情况下（即有多个线程不断争用一个内存位置的时候），基于锁的算法开始提供比非阻塞算法更好的吞吐率，因为当线程阻塞时，它就会停止争用，耐心地等候轮到自己，从而避免了进一步争用。但是，这么高的争用程度并不常见，因为多数时候，线程会把线程本地的计算与争用共享数据的操作分开，从而给其他线程使用共享数据的机会。同时，这么高的争用程度也表明需要重新检查算法，朝着更少共享数据的方向努力。

如果深入 Java 虚拟机和操作系统，会发现非阻塞算法无处不在。垃圾收集器使用非阻塞算法加快并发和平行的垃圾搜集；调度器使用非阻塞算法有效地调度线程和进程，实现内在锁。在 Java 6 中，基于锁的 SynchronousQueue 算法被新的非阻塞版本代替。很少有开发人员会直接使用 SynchronousQueue，但是通过 Executors.newCachedThreadPool()工厂构建的线程池用它作为工作队列。比较缓存线程池性能的对比测试显示，新的非阻塞同步队列实现提供了几乎是当前实现 3 倍的速度。在 Java 6 的后续版本中，已经规划了进一步的改进。

非阻塞算法要比基于锁的算法复杂得多。开发非阻塞算法是相当专业的训练，而且要证明算法的正确也极为困难。但是在 Java 版本之间并发性能上的众多改进来自对非阻塞算法的采用，而且随着并发性能变得越来越重要，可以预见在 Java 平台的未来发行版中，会使用更多的非阻塞算法。

参考文献

1. Brian Goetz，Java 理论与实践:流行的原子，
   http://www-128.ibm.com/developerworks/cn/java/j-jtp11234/index.html
2. java 并发集合，
   http://www.ibm.com/developerworks/cn/java/j-tiger06164/index.html

9 Java 内存模型

9 Java内存模型 1
9.1 Java内存模型 2
9.1.1 可见性 3
9.1.2 发生前关系（happen-before） 4
9.2 初始化安全性 5
参考文献 6

Java 平台把线程和多处理技术集成到了语言中，这种集成程度比以前的大多数编程语言都要强很多。该语言对于平台独立的并发及多线程技术的支持是野心勃勃并且是具有开拓性的，或许并不奇怪，这个问题要比 Java 体系结构设计者的原始构想要稍微困难些。关于同步和线程安全的许多底层混淆是 Java 内存模型 (Java Memory Model,JMM)的一些难以直觉到的细微差别。

例如，并不是所有的多处理器系统都表现出缓存一致性（cache coherency）；假如有一个处理器有一个更新了的变量值位于其缓存中，但还没有被存入主存，这样别的处理器就可能会看不到这个更新的值。在缓存缺乏一致性的情况下，两个不同的处理器可以看到在内存中同一位置处有两种不同的值。这听起来不太可能，但是这却是故意的 —— 这是一种获得较高的性能和可伸缩性的方法 —— 但是这加重了开发者和编译器为解决这些问题而编写代码的负担。

9.1 Java 内存模型

内存模型描述的是程序中各变量（实例域、静态域和数组元素）之间的关系，以及在实际计算机系统中将变量存储到内存和从内存取出变量这样的低层细节。对象最终存储在内存中，但编译器、运行库、处理器或缓存可以有特权定时地在变量的指定内存位置存入或取出变量值。

例如，编译器为了优化一个循环索引变量，可能会选择把它存储到一个寄存器中，或者缓存会延迟到一个更适合的时间，才把一个新的变量值存入主存。所有的这些优化是为了帮助实现更高的性能，通常这对于用户来说是透明的，但是对多处理系统来说，这些复杂的事情可能有时会完全显现出来。

JMM 允许编译器和缓存对数据在处理器特定的缓存（或寄存器）和主存之间移动的次序拥有重要的特权，除非程序员已经使用 synchronized 或 final 明确地请求了某些可见性保证。这意味着在缺乏同步的情况下，从不同的线程角度来看，内存的操作是以不同的次序发生的。

许多没有正确同步的程序在某些情况下似乎工作得很好，例如在轻微的负载下、在单处理器系统上，或者在具有比 JMM 所要求的更强的内存模型的处理器上。

“重新排序”(reordering)这个术语用于描述几种对内存操作的真实明显的重新排序的类型：

1) 当编译器不会改变程序的语义时，作为一种优化它可以随意地重新排序某些指令。

2) 在某些情况下，可以允许处理器以颠倒的次序执行一些操作。

3) 通常允许缓存以与程序写入变量时所不相同的次序把变量存入主存。

从另一线程的角度来看，任何这些条件都会引发一些操作以不同于程序指定的次序发生 —— 并且忽略重新排序的源代码时，内存模型认为所有这些条件都是同等的。

9.1.1 可见性

1. 同步与可见性（visibility）

大多数程序员都知道，synchronized 关键字强制实施一个互斥锁（互相排斥），这个互斥锁防止每次有多个线程进入一个给定监控器所保护的同步语句块。但是同步还有另一个方面：正如 JMM 所指定，它强制实施某些**内存可见性规则**。它确保了当存在一个同步块时缓存被更新，当输入一个同步块时缓存失效。因此，在一个由给定监控器保护的同步块期间，一个线程所写入的值对于其余所有的执行由同一监控器所保护的同步块的线程来说是可见的。它也确保了编译器不会把指令从一个同步块的内部移到外部（虽然在某些情况下它会把指令从同步块的外部移到内部）。JMM 在缺乏同步的情况下不会做这种保证 —— 这就是只要有多个线程访问相同的变量时必须使用同步（或者它的同胞，易失性）的原因。

2. 不可变对象的问题

不可变对象似乎可以改变它们的值（这种对象的不变性旨在通过使用 final 关键字来得到保证）。让一个对象的所有字段都为 final 并不一定使得这个对象不可变 —— 所有的字段还必须是原语类型或是对不可变对象的引用。不可变对象（如 String ）被认为不要求同步。但是，因为在将内存写方面的更改从一个线程传播到另一个线程时存在潜在的延迟，所以有可能存在一种竞态条件，即允许一个线程首先看到不可变对象的一个值，一段时间之后看到的是一个不同的值。

3. Volatile 与可见性

可见性——如何知道当线程 A 执行 someVariable=3 时，其他线程是否可以看到线程 A 所写的值 3？有一些原因使其他线程不能立即看到 someVariable 的值 3：可能是因为编译器为了执行效率更高而重新排序了指令，也可能是 someVariable 缓存在寄存器中，或者

它的值写到写处理器的缓存中、但是还没有刷新到主存中，或者在读处理器的缓存中有一个老的（或者无效的）值。内存模型决定什么时候一个线程可以可靠地“看到”由其他线程对变量的写入。特别是，内存模型定义了保证内存操作跨线程的可见性的 volatile 、synchronized 和 final 的语义。

在多个线程访问同一个变量时，必须使用同步或者 volatile。volatile 语义保证 volatile 字段的读写直接在主存而不是寄存器或者本地处理器缓存中进行，并且代表线程对 volatile 变量进行的这些操作是按线程要求的顺序进行的。换句话说，这意味着老的内存模型保证正在读或写的变量的可见性，不保证写入其他变量的可见性。Volatile 变量可以与对非 volatile 变量的读写一起重新排序。

如果当线程 A 写入 volatile 变量 V，而线程 B 读取 V 时，那么在写入 V 时，A 可见的所有变量值现在都可以保证对 B 是可见的。代价是访问 volatile 字段时会对性能产生更大的影响。

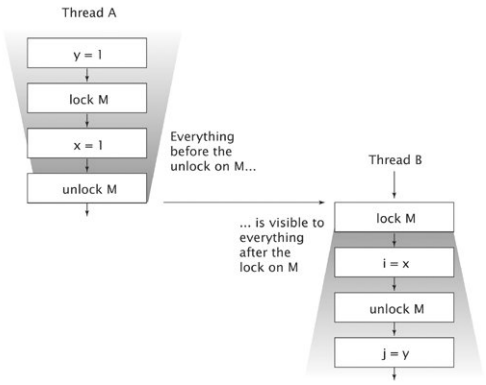
9.1.2 发生前关系（happen-before）

像对变量的读写这样的操作，在线程中是根据所谓的“程序顺序”——程序的语义声明它们应当发生的顺序——排序的。在不同线程中的操作完全不一定要彼此排序——如果启动两个线程并且它们对任何公共监视器都不用同步执行、或者不涉及任何公共 volatile 变量，则完全无法准确地预言一个线程中的操作（或者对第三个线程可见）相对于另一个线程中操作的顺序。

排序保证是在线程启动、一个线程参与另一个线程、一个线程获得或者释放一个监视器（进入或者退出一个同步块）、或者一个线程访问一个 volatile 变量时创建的。JMM 描述了程序使用同步或者 volatile 变量以协调多个线程中的活动时所进行的顺序保证。新的 JMM 非正式地定义了一个名为 happens-before 的排序，它是程序中所有操作的部分顺序。

1) 一个线程中的每个操作“发生之前”于这个线程程序规定的其他后续出现的操作。
2) 对监视器的解锁“发生之前”于同一监视器上的所有后续锁定。
3) 对 volatile 变量的写“发生之前”于同一 volatile 变量 的每一个后续读。
4) 一个线程的 Thread.start()调用“发生之前”于这个启动后的线程的其他操作。
5) 线程中的所有操作“发生之前” 从这个线程的 Thread.join() 成功返回的所有其他线程。

例如，下面是用同步保证线程内存写的可见性。



9.2 初始化安全性

JMM 还寻求提供一种新的初始化安全性保证——只要对象是正确构造的(意即不会在构造函数完成之前发布对这个对象的引用)，然后所有线程都会看到在构造函数中设置的 final 字段的值，不管是否使用同步在线程之间传递这个引用。而且，所有可以通过正确构造的对象的 final 字段可及的变量，如用一个 final 字段引用的对象的 final 字段，也保证对其他线程是可见的。这意味着如果 final 字段包含，比如说对一个 LinkedList 的引用，除了引用的正确的值对于其他线程是可见的外，这个 LinkedList 在构造时的内容在不同步的情况下，对于其他线程也是可见的。

可以不用同步安全地访问这个 final 字段，编译器可以假定 final 字段将不会改变，因而可以优化多次提取。

在构造函数的 final 字段的写与在另一个线程中对这个对象的共享引用的初次装载之间有一个类似于 happens-before 的关系。当构造函数完成任务时，对 final 字段的所有写（以及通过这些 final 字段间接可及的变量）变为“冻结”，所有在冻结之后获得对这个对象的引用的线程都会保证看到所有冻结字段的冻结值。初始化 final 字段的写将不会与构造函数关联的冻结后面的操作一起重新排序。

初始化安全性保证了在多线程共享情况下不可变对象的正确构造。

参考文献

1） 修 Java 内 存 模 型 （ 1 ）， http://www.ibm.com/developerworks/cn/java/j-jtp02244/

2） 修 Java 内 存 模 型 （ 2 ）， http://www.ibm.com/developerworks/cn/java/j-jtp03304/

3） 实 现 一 个 不 受 约 束 的 不 变 性 模 型， http://www.ibm.com/developerworks/cn/java/j-immutability.html

4） 一种新的 Java 存储模型 L2JMM，计算机研究与发展，2006，43（4）