第二章 构建线程安全应用程序

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。

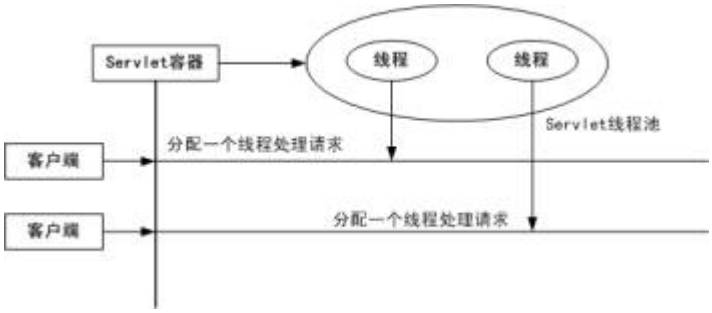
通过将类记录为线程安全的(假设它确实是线程安全的)，您就提供了两种有价值的服务：您告知类的维护者不要进行会影响其线程安全性的修改或者扩展，您还告知类的用户使用它时可以不使用外部同步。通过将类记录为线程兼容或者有条件线程安全的，您就告知了用户这个类可以通过正确使用同步而安全地在多线程中使用。通过将类记录为线程对立的，您就告知用户即使使用了外部同步，他们也不能在多线程中安全地使用这个类。不管是哪种情况，您都在潜在的严重问题出现之前防止了它们，而要查找和修复这些问题是很昂贵的。

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