第 5 章 数据冲突及诊断工具 MTRAT

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。

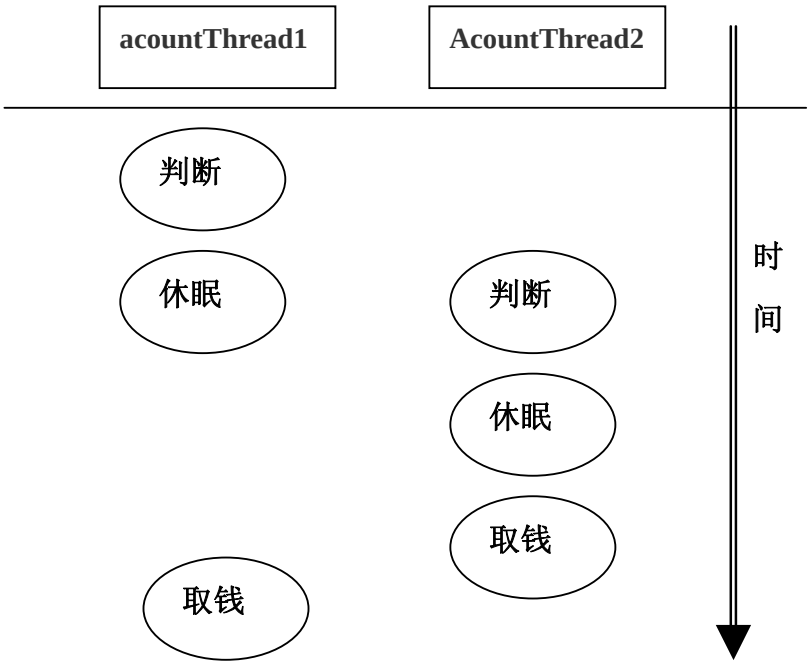

图 5-1 一种可能的线程运行过程

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。

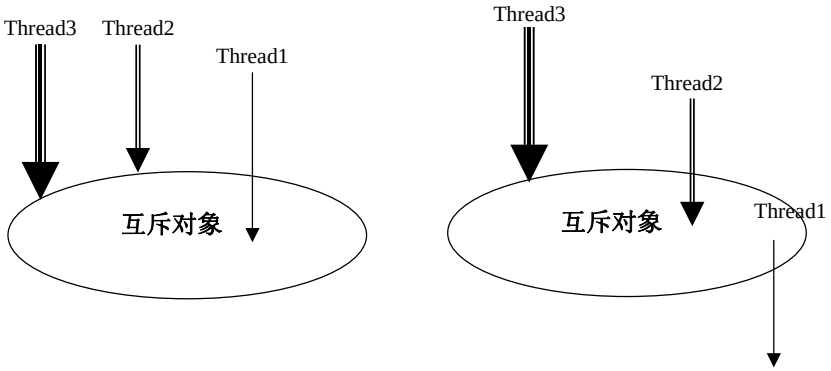

图 5-2 互斥对象及其等待队列

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。

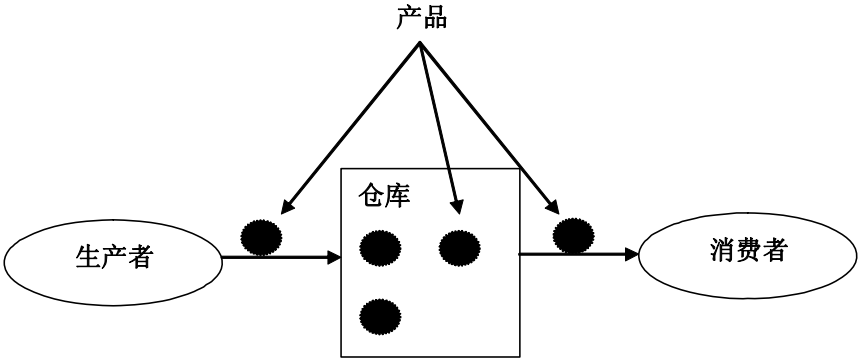

图 5-3 生产-消费者模型

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。

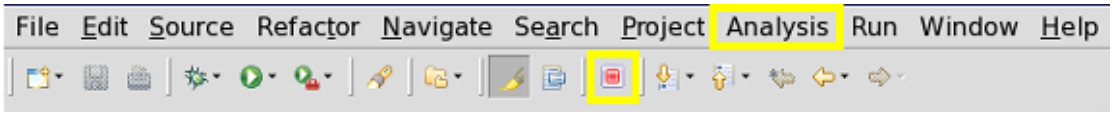

图 5-5MtratEclipse 插件

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

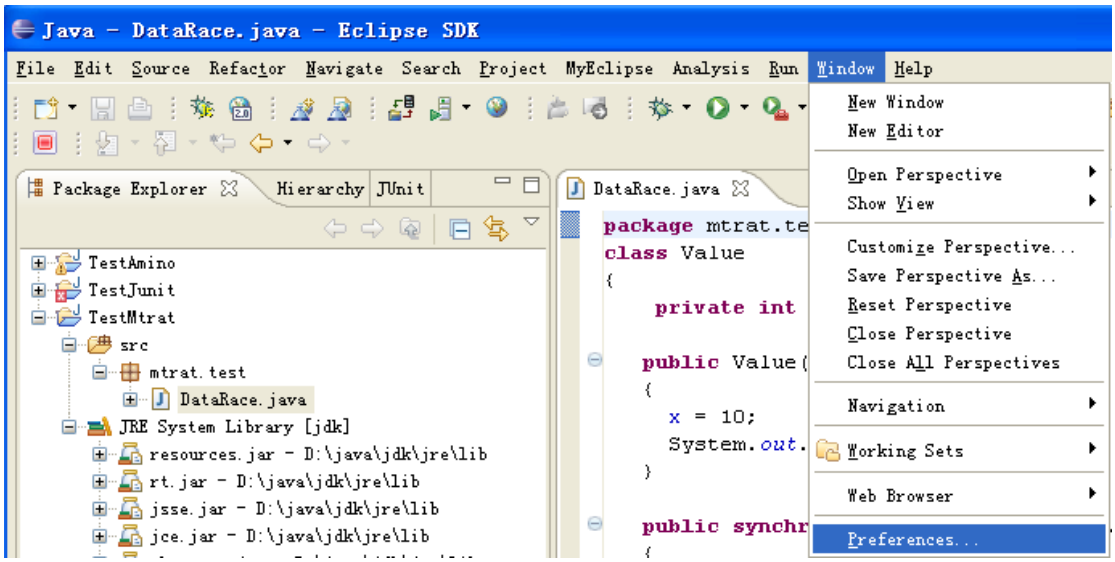

图 5-3EclipsePerences 界面

2）选择命令行软件安装目录 D:\Mtrat,如图 5-6。

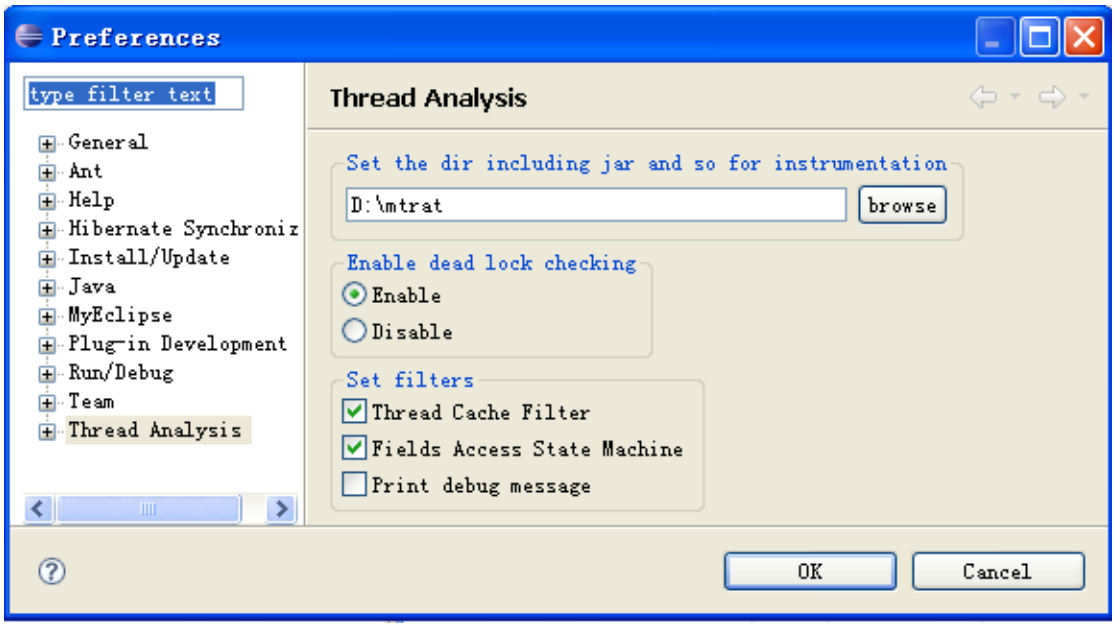

图 5-6Perences 界面下配置 Mtrat

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。

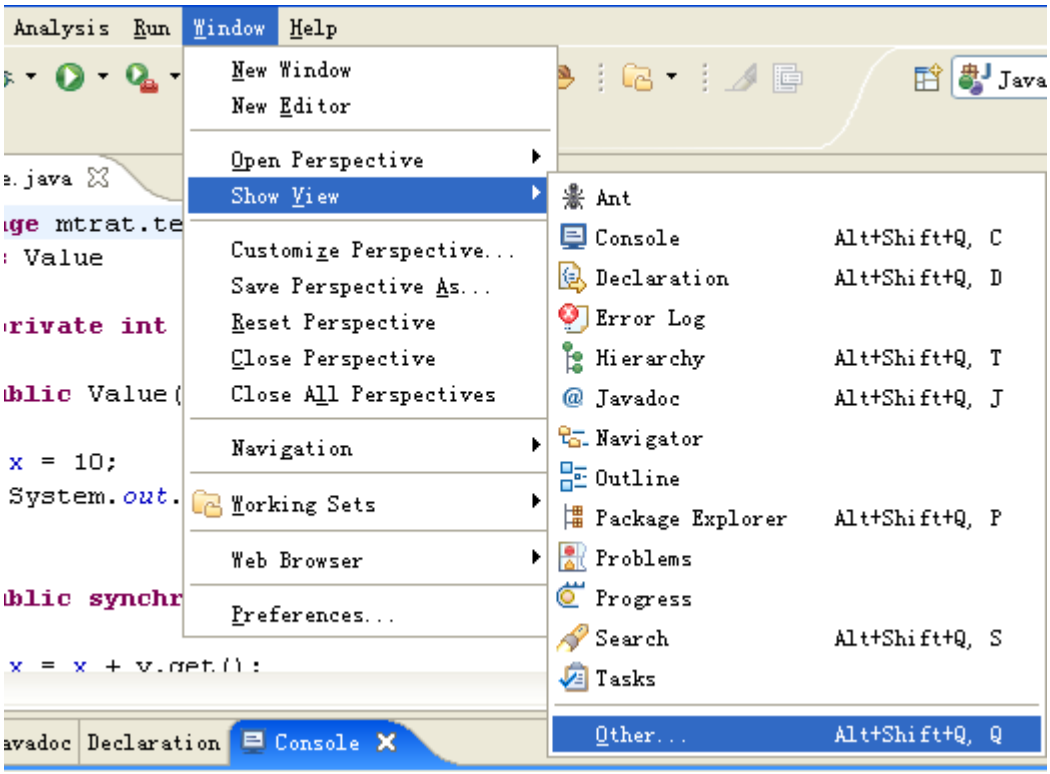

图 5-7Eclipse 窗口 ShowView 界面

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

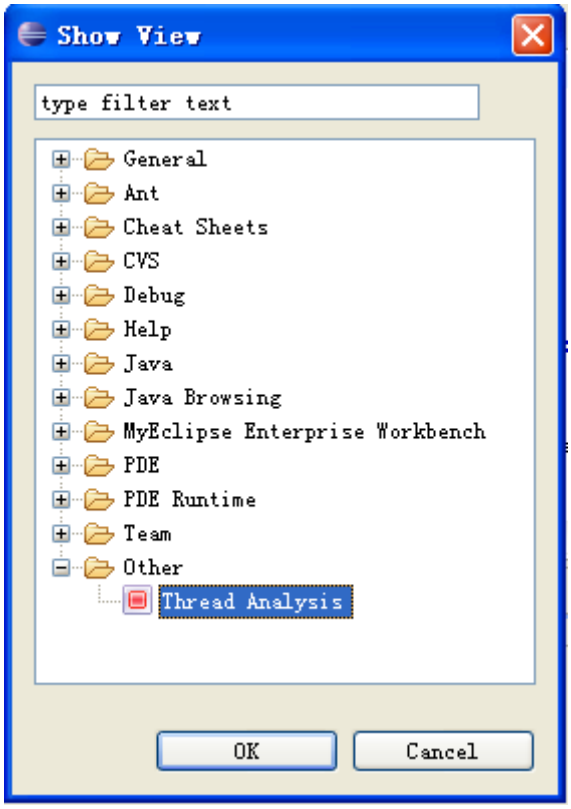

图 5-8ShowView 界面的选项

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：

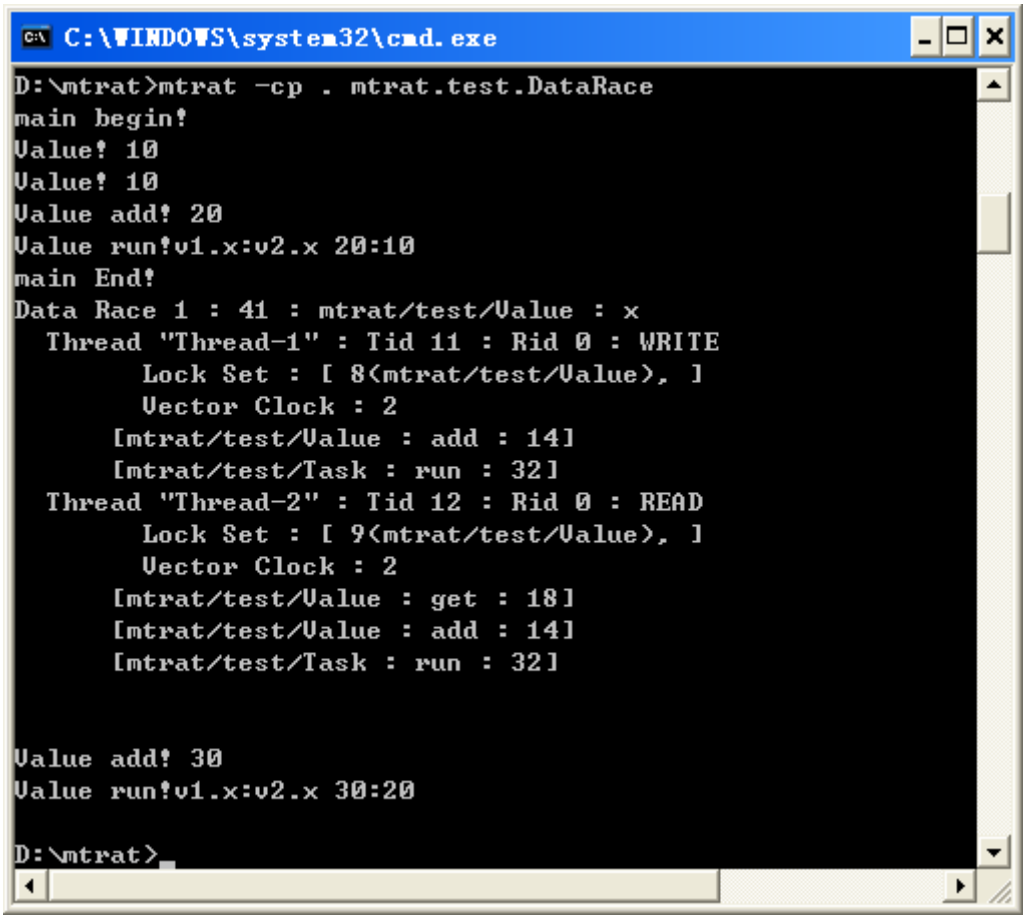

图 5-10Mtrat 分析案例命令行窗口

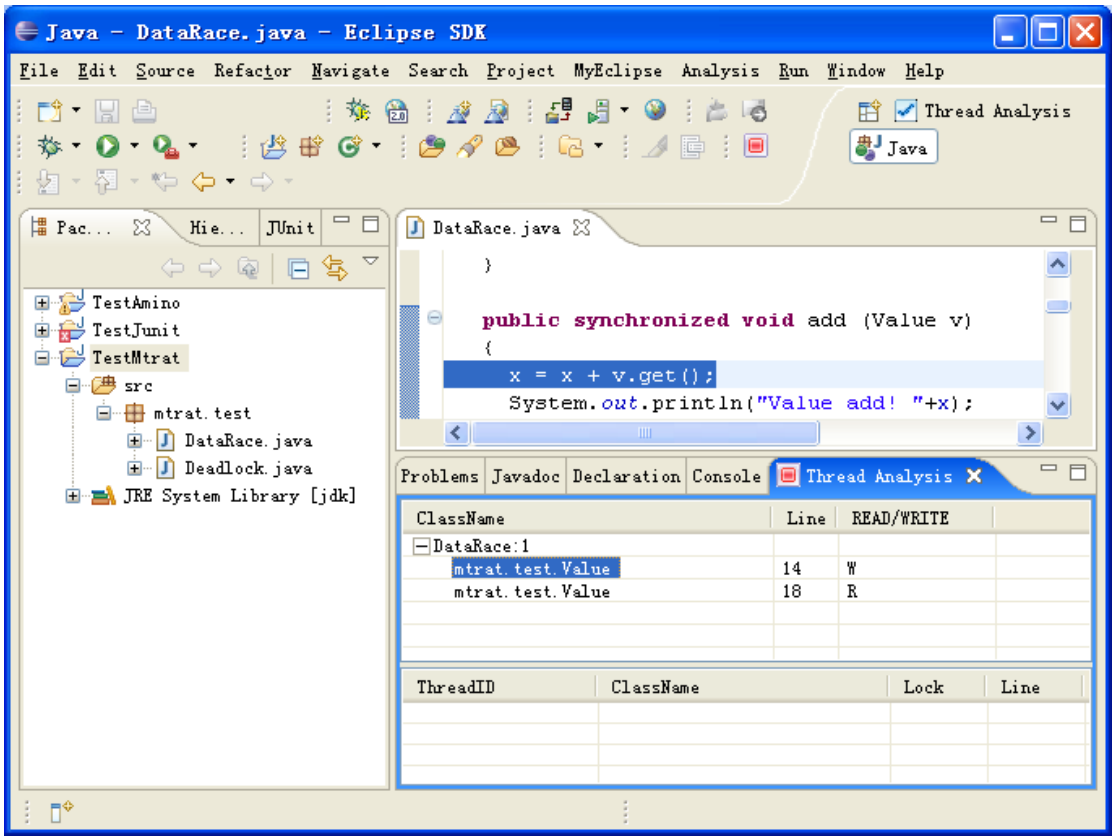

图 5-11Mtrat 分析案例 Eclipse 窗口

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。

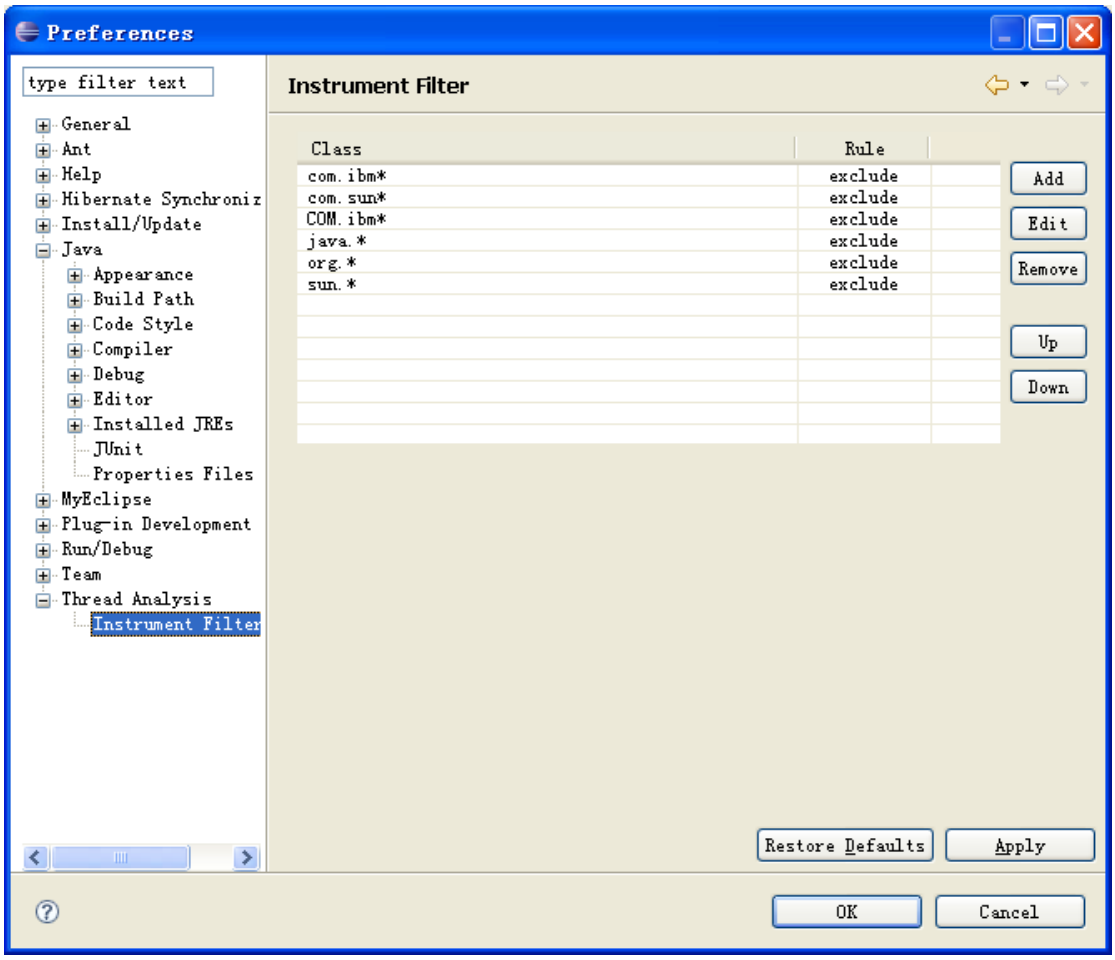

图 5-12Mtrat 分析器排出在外的类

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。

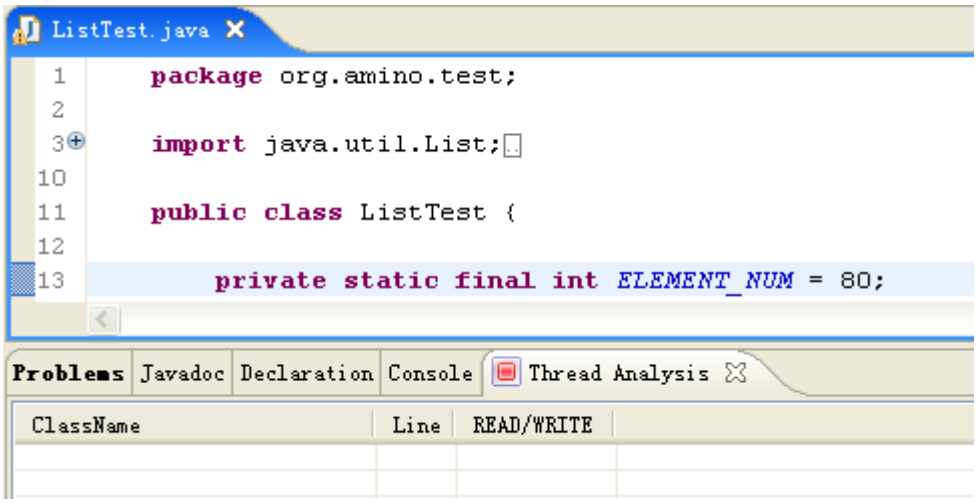

图 5-13Eclipse 插件分析结果

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