Java 多线程文章系列

【枫叶 KR 整理】

Java 多线程文章系列

Java 多线程编程详解

一：理解多线程

多线程是这样一种机制，它允许在程序中并发执行多个指令流，每个指令流都称为一个线程，彼此间互相独立。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远较进程简单。

多个线程的执行是并发的，也就是在逻辑上“同时”，而不管是否是物理上的“同时”。如果系统只有一个 CPU，那么真正的“同时”是不可能的，但是由于 CPU 的速度非常快，用户感觉不到其中的区别，因此我们也不用关心它，只需要设想各个线程是同时执行即可。多线程和传统的单线程在程序设计上最大的区别在于，由于各个线程的控制流彼此独立，使得各个线程之间的代码是乱序执行的，由此带来的线程调度，同步等问题，将在以后探讨。

二：在 Java 中实现多线程

我们不妨设想，为了创建一个新的线程，我们需要做些什么？很显然，我们必须指明这个线程所要执行的代码，而这就是在 Java 中实现多线程我们所需要做的一切！

真是神奇！Java 是如何做到这一点的？通过类！作为一个完全面向对象的语言，Java 提供了类 java.lang.Thread 来方便多线程编程，这个类提供了大量的方法来方便我们控制自己的各个线程，我们以后的讨论都将围绕这个类进行。

那么如何提供给 Java 我们要线程执行的代码呢？让我们来看一看 Thread 类。Thread 类最重要的方法是 run() ，它为 Thread 类的方法 start() 所调用，提供我们的线程所要执行的代码。为了指定我们自己的代码，只需要覆盖它！

方法一：继承 Thread 类，覆盖方法 run()，我们在创建的 Thread 类的子类中重写 run() ，加入线程所要执行的代码即可。下面是一个例子：

```
public class MyThread extends Thread {
int count= 1, number;
public MyThread(int num) {
number = num;
System.out.println("创建线程 " + number);
}
public void run() {
while(true) {
System.out.println("线程 " + number + ":计数 " + count);
if(++count== 6) return;
```

```
}
}
public static void main(String args[]) {
for(int i = 0; i < 5; i++) new MyThread(i+1).start();
}
}
```

这种方法简单明了，符合大家的习惯，但是，它也有一个很大的缺点，那就是如果我们的类已经从一个类继承（如小程序必须继承自 Applet 类），则无法再继承 Thread 类，这时如果我们又不想建立一个新的类，应该怎么办呢？

我们不妨来探索一种新的方法：我们不创建 Thread 类的子类，而是直接使用它，那么我们只能将我们的方法作为参数传递给 Thread 类的实例，有点类似回调函数。但是 Java 没有指针，我们只能传递一个包含这个方法的类的实例。那么如何限制这个类必须包含这一方法呢？当然是使用接口！（虽然抽象类也可满足，但是需要继承，而我们之所以要采用这种新方法，不就是为了避免继承带来的限制吗？）

Java 提供了接口 java.lang.Runnable 来支持这种方法。

方法二：实现 Runnable 接口

Runnable 接口只有一个方法 run()，我们声明自己的类实现 Runnable 接口并提供这一方法，将我们的线程代码写入其中，就完成了这一部分的任务。但是 Runnable 接口并没有任何对线程的支持，我们还必须创建 Thread 类的实例，这一点通过 Thread 类的构造函数 public Thread(Runnable target);来实现。下面是一个例子：

```
public class MyThread implements Runnable {
int count= 1, number;
public MyThread(int num) {
number = num;
System.out.println("创建线程 " + number);
}
public void run() {
while(true) {
System.out.println("线程 " + number + ":计数 " + count);
if(++count== 6) return;
}
}
public static void main(String args[]) {
for(int i = 0; i < 5; i++) new Thread(new MyThread(i+1)).start();
}
}
```

严格地说，创建 Thread 子类的实例也是可行的，但是必须注意的是，该子类必须没有覆盖 Thread 类的 run 方法，否则该线程执行的将是子类的 run 方法，而不是我们用以实现 Runnable 接口的类的 run 方法，对此大家不妨试验一下。

使用 Runnable 接口来实现多线程使得我们能够在一个类中包容所有的代码，有利于封装，它的缺点在于，我们只能使用一套代码，若想创建多个线程并使各个线程执行不同的代码，则仍必须额外创建类，如果这样的话，在大多数情况下也许还不如直接用多个类分别继承 Thread 来得紧凑。

综上所述，两种方法各有千秋，大家可以灵活运用。

下面让我们一起来研究一下多线程使用中的一些问题。

三：线程的四种状态

1. 新状态：线程已被创建但尚未执行（start() 尚未被调用）。
2. 可执行状态：线程可以执行，虽然不一定正在执行。CPU 时间随时可能被分配给该线程，从而使得它执行。
3. 死亡状态：正常情况下 run() 返回使得线程死亡。调用 stop()或 destroy() 亦有同样效果，但是不被推荐，前者会产生异常，后者是强制终止，不会释放锁。
4. 阻塞状态：线程不会被分配 CPU 时间，无法执行。

四：线程的优先级

线程的优先级代表该线程的重要程度，当有多个线程同时处于可执行状态并等待获得 CPU 时间时，线程调度系统根据各个线程的优先级来决定给谁分配 CPU 时间，优先级高的线程有更大的机会获得 CPU 时间，优先级低的线程也不是没有机会，只是机会要小一些罢了。

你可以调用 Thread 类的方法 getPriority() 和 setPriority()来存取线程的优先级，线程的优先级界于 1(MIN_PRIORITY) 和 10(MAX_PRIORITY) 之间，缺省是 5(NORM_PRIORITY)。

五：线程的同步

由于同一进程的多个线程共享同一片存储空间，在带来方便的同时，也带来了访问冲突这个严重的问题。Java 语言提供了专门机制以解决这种冲突，有效避免了同一个数据对象被多个线程同时访问。

由于我们可以通过 private 关键字来保证数据对象只能被方法访问，所以我们只需针对方法提出一套机制，这套机制就是 synchronized 关键字，它包括两种用法：synchronized 方法和 synchronized 块。

1. synchronized 方法：通过在方法声明中加入 synchronized 关键字来声明 synchronized 方法。如：

```
public synchronized void accessVal(int newVal);
```

synchronized 方法控制对类成员变量的访问：每个类实例对应一把锁，每个 synchronized 方法都必须获得调用该方法的类实例的锁方能执行，否则所属线程阻塞，方法一旦执行，就独占该锁，直到从该方法返回时才将锁释放，此后被阻塞的线程方能获得该锁，重新进入可执行状态。这种机制确保了同一时刻对于每一个类实例，其所有声明为 synchronized 的成员函数中至多只有一个处于可执行状态（因为至多只有一个能够获得该类实例对应的锁），从而有效避免了类成员变量的访问冲突（只要所有可能访问类成员变量的方法均被声明为 synchronized）。

在 Java 中，不光是类实例，每一个类也对应一把锁，这样我们也可将类的静态成员函数声明为 synchronized ，以控制其对类的静态成员变量的访问。
synchronized 方法的缺陷：若将一个大的方法声明为 synchronized 将会大大影响效率，典型地，若将线程类的方法 run() 声明为 synchronized ，由于在线程的整个生命期内它一直在运行，因此将导致它对本类任何 synchronized 方法的调用都永远不会成功。当然我们可以通过将访问类成员变量的代码放到专门的方法中，将其声明为 synchronized ，并在主方法中调用来解决这一问题，但是 Java 为我们提供了更好的解决办法，那就是 synchronized 块。
2. synchronized 块：通过 synchronized 关键字来声明 synchronized 块。语法如下：

```
synchronized(syncObject) {
//允许访问控制的代码
}
```

synchronized 块是这样一个代码块，其中的代码必须获得对象 syncObject （如前所述，可以是类实例或类）的锁方能执行，具体机制同前所述。由于可以针对任意代码块，且可任意指定上锁的对象，故灵活性较高。

六：线程的阻塞

为了解决对共享存储区的访问冲突，Java 引入了同步机制，现在让我们来考察多个线程对共享资源的访问，显然同步机制已经不够了，因为在任意时刻所要求的资源不一定已经准备好了被访问，反过来，同一时刻准备好了的资源也可能不止一个。为了解决这种情况下的访问控制问题，Java 引入了对阻塞机制的支持。

阻塞指的是暂停一个线程的执行以等待某个条件发生（如某资源就绪），学过操作系统的同学对它一定已经很熟悉了。Java 提供了大量方法来支持阻塞，下面让我们逐一分析。
1. sleep() 方法：sleep() 允许 指定以毫秒为单位的一段时间作为参数，它使得线程在指定的时间内进入阻塞状态，不能得到 CPU 时间，指定的时间一过，线程重新进入可执行状态。典型地，sleep() 被用在等待某个资源就绪的情形：测试发现条件不满足后，让线程阻塞一段时间后重新测试，直到条件满足为止。
2. suspend() 和 resume() 方法：两个方法配套使用，suspend()使得线程进入阻塞状态，并且不会自动恢复，必须其对应的 resume() 被调用，才能使得线程重新进入可执行状态。典型地，suspend() 和 resume() 被用在等待另一个线程产生的结果的情形：测试发现结果还没有产生后，让线程阻塞，另一个线程产生了结果后，调用 resume() 使其恢复。
3. yield() 方法：yield() 使得线程放弃当前分得的 CPU 时间，但是不使线程阻塞，即线程仍处于可执行状态，随时可能再次分得 CPU 时间。调用 yield() 的效果等价于调度程序认为该线程已执行了足够的时间从而转到另一个线程。
4. wait() 和 notify() 方法：两个方法配套使用，wait() 使得线程进入阻塞状态，它有两种形式，一种允许指定以毫秒为单位的一段时间作为参数，另一种没有参数，前者当对应的 notify() 被调用或者超出指定时间时线程重新进入可执行状态，后者则必须对应的 notify() 被调用。

初看起来它们与 suspend() 和 resume() 方法对没有什么分别，但是事实上它们是截然不同的。区别的核心在于，前面叙述的所有方法，阻塞时都不会释放占用的锁（如果占用了的话），而这一对方法则相反。
上述的核心区别导致了一系列的细节上的区别。

首先，前面叙述的所有方法都隶属于 Thread 类，但是这一对却直接隶属于 Object 类，也就是说，所有对象都拥有这一对方法。初看起来这十分不可思议，但是实际上却是很自然的，因为这一对方法阻塞时要释放占用的锁，而锁是任何对象都具有的，调用任意对象的 wait() 方法导致线程阻塞，并且该对象上的锁被释放。而调用 任意对象的 notify()方法则导致因调用该对象的 wait() 方法而阻塞的线程中随机选择的一个解除阻塞（但要等到获得锁后才真正可执行）。

其次，前面叙述的所有方法都可在任何位置调用，但是这一对方法却必须在 synchronized 方法或块中调用，理由也很简单，只有在 synchronized 方法或块中当前线程才占有锁，才有锁可以释放。同样的道理，调用这一对方法的对象上的锁必须为当前线程所拥有，这样才有锁可以释放。因此，这一对方法调用必须放置在这样的 synchronized 方法或块中，该方法或块的上锁对象就是调用这一对方法的对象。若不满足这一条件，则程序虽然仍能编译，但在运行时会出现 IllegalMonitorStateException 异常。

wait() 和 notify() 方法的上述特性决定了它们经常和 synchronized 方法或块一起使用，将它们和操作系统的进程间通信机制作一个比较就会发现它们的相似性：synchronized 方法或块提供了类似于操作系统原语的功能，它们的执行不会受到多线程机制的干扰，而这一对方法则相当于 block 和 wakeup 原语（这一对方法均声明为 synchronized）。它们的结合使得我们可以实现操作系统上一系列精妙的进程间通信的算法（如信号量算法），并用于解决各种复杂的线程间通信问题。

关于 wait() 和 notify() 方法最后再说明两点：

第一：调用 notify() 方法导致解除阻塞的线程是从因调用该对象的 wait() 方法而阻塞的线程中随机选取的，我们无法预料哪一个线程将会被选择，所以编程时要特别小心，避免因这种不确定性而产生问题。

第二：除了 notify()，还有一个方法 notifyAll() 也可起到类似作用，唯一的区别在于，调用 notifyAll() 方法将把因调用该对象的 wait() 方法而阻塞的所有线程一次性全部解除阻塞。当然，只有获得锁的那一个线程才能进入可执行状态。

谈到阻塞，就不能不谈一谈死锁，略一分析就能发现，suspend() 方法和不指定超时期限的 wait() 方法的调用都可能产生死锁。遗憾的是，Java 并不在语言级别上支持死锁的避免，我们在编程中必须小心地避免死锁。

以上我们对 Java 中实现线程阻塞的各种方法作了一番分析，我们重点分析了 wait() 和 notify() 方法，因为它们的功能最强大，使用也最灵活，但是这也导致了它们的效率较低，较容易出错。实际使用中我们应该灵活使用各种方法，以便更好地达到我们的目的。

七：守护线程

守护线程是一类特殊的线程，它和普通线程的区别在于它并不是应用程序的核心部分，当一个应用程序的所有非守护线程终止运行时，即使仍然有守护线程在运行，应用程序也将终止，反之，只要有一个非守护线程在运行，应用程序就不会终止。守护线程一般被用于在后台为其它线程提供服务。

可以通过调用方法 isDaemon() 来判断一个线程是否是守护线程，也可以调用方法 setDaemon() 来将一个线程设为守护线程。

八：线程组

线程组是一个 Java 特有的概念，在 Java 中，线程组是类 ThreadGroup 的对象，每个线程都隶属于唯一一个线程组，这个线程组在线程创建时指定并在线程的整个生命期内都不能更改。你可以通过调用包含 ThreadGroup 类型参数的 Thread 类构造函数来指定线程属的线程组，若没有指定，则线程缺省地隶属于名为 system 的系统线程组。

在 Java 中，除了预建的系统线程组外，所有线程组都必须显式创建。在 Java 中，除系统线程组外的每个线程组又隶属于另一个线程组，你可以在创建线程组时指定其所隶属的线程组，若没有指定，则缺省地隶属于系统线程组。这样，所有线程组组成了一棵以系统线程组为根的树。

Java 允许我们对一个线程组中的所有线程同时进行操作，比如我们可以通过调用线程组的相应方法来设置其中所有线程的优先级，也可以启动或阻塞其中的所有线程。

Java 的线程组机制的另一个重要作用是线程安全。线程组机制允许我们通过分组来区分有不同安全特性的线程，对不同组的线程进行不同的处理，还可以通过线程组的分层结构来支持不对等安全措施的采用。Java 的 ThreadGroup 类提供了大量的方法来方便我们对线程组树中的每一个线程组以及线程组中的每一个线程进行操作。

九：总结

在这一讲中，我们一起学习了 Java 多线程编程的方方面面，包括创建线程，以及对多个线程进行调度、管理。我们深刻认识到了多线程编程的复杂性，以及线程切换开销带来的多线程程序的低效性，这也促使我们认真地思考一个问题：我们是否需要多线程？何时需要多线程？

多线程的核心在于多个代码块并发执行，本质特点在于各代码块之间的代码是乱序执行的。我们的程序是否需要多线程，就是要看这是否也是它的内在特点。

假如我们的程序根本不要求多个代码块并发执行，那自然不需要使用多线程；假如我们的程序虽然要求多个代码块并发执行，但是却不要求乱序，则我们完全可以用一个循环来简单高效地实现，也不需要使用多线程；只有当它完全符合多线程的特点时，多线程机制对线程间通信和线程管理的强大支持才能有用武之地，这时使用多线程才是值得的。

解析 Java 中的多线程机制

一、进程与应用程序的区别

进程（Process）是最初定义在 Unix 等多用户、多任务操作系统环境下用于表示应用程序在内存环境中基本执行单元的概念。以 Unix 操作系统为例，进程是 Unix 操作系统环境中的基本成分、是系统资源分配的基本单位。Unix 操作系统中完成的几乎所有用户管理和资

源分配等工作都是通过操作系统对应用程序进程的控制来实现的。

C、C++、Java 等语言编写的源程序经相应的编译器编译成可执行文件后，提交给计算机处理器运行。这时，处在可执行状态中的应用程序称为进程。从用户角度来看，进程是应用程序的一个执行过程。从操作系统核心角度来看，进程代表的是操作系统分配的内存、CPU 时间片等资源的基本单位，是为正在运行的程序提供的运行环境。进程与应用程序的区别在于应用程序作为一个静态文件存储在计算机系统的硬盘等存储空间中，而进程则是处于动态条件下由操作系统维护的系统资源管理实体。多任务环境下应用程序进程的主要特点包括：

●进程在执行过程中有内存单元的初始入口点，并且进程存活过程中始终拥有独立的内存地址空间；

●进程的生存期状态包括创建、就绪、运行、阻塞和死亡等类型；

●从应用程序进程在执行过程中向 CPU 发出的运行指令形式不同，可以将进程的状态分为用户态和核心态。处于用户态下的进程执行的是应用程序指令、处于核心态下的应用程序进程执行的是操作系统指令。

在 Unix 操作系统启动过程中，系统自动创建 swapper、init 等系统进程，用于管理内存资源以及对用户进程进行调度等。在 Unix 环境下无论是由操作系统创建的进程还要由应用程序执行创建的进程，均拥有唯一的进程标识（PID）。

二、进程与 Java 线程的区别

应用程序在执行过程中存在一个内存空间的初始入口点地址、一个程序执行过程中的代码执行序列以及用于标识进程结束的内存出口点地址，在进程执行过程中的每一时间点均有唯一的处理器指令与内存单元地址相对应。

Java 语言中定义的线程（Thread）同样包括一个内存入口点地址、一个出口点地址以及能够顺序执行的代码序列。但是进程与线程的重要区别在于线程不能够单独执行，它必须运行在处于活动状态的应用程序进程中，因此可以定义线程是程序内部的具有并发性的顺序代码流。

Unix 操作系统和 Microsoft Windows 操作系统支持多用户、多进程的并发执行，而 Java 语言支持应用程序进程内部的多个执行线程的并发执行。多线程的意义在于一个应用程序的多个逻辑单元可以并发地执行。但是多线程并不意味着多个用户进程在执行，操作系统也不把每个线程作为独立的进程来分配独立的系统资源。进程可以创建其子进程，子进程与父进程拥有不同的可执行代码和数据内存空间。而在用于代表应用程序的进程中多个线程共享数据内存空间，但保持每个线程拥有独立的执行堆栈和程序执行上下文（Context）。

基于上述区别，线程也可以称为轻型进程 (Light Weight Process，LWP)。不同线程间允许任务协作和数据交换，使得在计算机系统资源消耗等方面非常廉价。

线程需要操作系统的支持，不是所有类型的计算机都支持多线程应用程序。Java 程序设计语言将线程支持与语言运行环境结合在一起，提供了多任务并发执行的能力。这就好比一个人在处理家务的过程中，将衣服放到洗衣机中自动洗涤后将大米放在电饭锅里，然后开始做菜。等菜做好了，饭熟了同时衣服也洗好了。

需要注意的是：在应用程序中使用多线程不会增加 CPU 的数据处理能力。只有在多 CPU 的计算机或者在网络计算体系结构下，将 Java 程序划分为多个并发执行线程后，同时启动多个线程运行，使不同的线程运行在基于不同处理器的 Java 虚拟机中，才能提高应用程序的执行效率。

另外，如果应用程序必须等待网络连接或数据库连接等数据吞吐速度相对较慢的资源时，多线程应用程序是非常有利的。基于 Internet 的应用程序有必要是多线程类型的，例如，当开发要支持大量客户机的服务器端应用程序时，可以将应用程序创建成多线程形式来响应客户端的连接请求，使每个连接用户独占一个客户端连接线程。这样，用户感觉服务器只为连接用户自己服务，从而缩短了服务器的客户端响应时间。

三、Java 语言的多线程程序设计方法

利用 Java 语言实现多线程应用程序的方法很简单。根据多线程应用程序继承或实现对象的不同可以采用两种方式：一种是应用程序的并发运行对象直接继承 Java 的线程类 Thread；另外一种方式是定义并发执行对象实现 Runnable 接口。

继承 Thread 类的多线程程序设计方法

Thread 类是 JDK 中定义的用于控制线程对象的类，在该类中封装了用于进行线程控制的方法。见下面的示例代码：

[code]//Consumer.java
```
import java.util.*;
class Consumer extends Thread
{
  int nTime;
  String strConsumer;
  public Consumer(int nTime, String strConsumer)
  {
  this.nTime = nTime;
  this.strConsumer = strConsumer;
```

```
  }
  public void run()
  {
while(true)
{
  try
{
  System.out.println("Consumer name:"+strConsumer+"\n");
  Thread.sleep(nTime);
  }
catch(Exception e)
{
  e.printStackTrace();
  }
}
  }
static public void main(String args[])
{
  Consumer aConsumer = new Consumer (1000, "aConsumer");
  aConsumer.start();
  Consumer bConsumer = new Consumer (2000, "bConsumer");
  bConsumer.start();
  Consumer cConsumer = new Consumer (3000, "cConsumer ");
  cConsumer.start();
}
} [/code]
```

从上面的程序代码可以看出：多线程执行地下 Consumer 继承 Java 语言中的线程类 Thread 并且在 main 方法中创建了三个 Consumer 对象的实例。当调用对象实例的 start 方法时，自动调用 Consumer 类中定义的 run 方法启动对象线程运行。线程运行的结果是每间隔 nTime 时间打印出对象实例中的字符串成员变量 strConsumer 的内容。

可以总结出继承 Thread 类的多线程程序设计方法是使应用程序类继承 Thread 类并且在该类的 run 方法中实现并发性处理过程。

实现 Runnable 接口的多线程程序设计方法

Java 语言中提供的另外一种实现多线程应用程序的方法是多线程对象实现 Runnable 接口并且在该类中定义用于启动线程的 run 方法。这种定义方式的好处在于多线程应用对象可以继承其它对象而不是必须继承 Thread 类，从而能够增加类定义的逻辑性。

实现 Runnable 接口的多线程应用程序框架代码如下所示：

```
//Consumer.java
import java.util.*;
class Consumer implements Runnable
{
  … …
public Consumer(int nTime, String strConsumer){… …}
public void run(){… …}
static public void main(String args[])
{
Thread aConsumer = new Thread(new Consumer(1000, "aConsumer"));
aConsumer.start();
//其它对象实例的运行线程
  //… …
  }
}
```

从上述代码可以看出：该类实现了 Runnable 接口并且在该类中定义了 run 方法。这种多线程应用程序的实现方式与继承 Thread 类的多线程应用程序的重要区别在于启动多线程对象的方法设计方法不同。在上述代码中，通过创建 Thread 对象实例并且将应用对象作为创建 Thread 类实例的参数。

四、线程间的同步

Java 应用程序的多个线程共享同一进程的数据资源，多个用户线程在并发运行过程中可能同时访问具有敏感性的内容。在 Java 中定义了线程同步的概念，实现对共享资源的一致性维护。下面以笔者最近开发的移动通信计费系统中线程间同步控制方法，说明 Java 语言中多线程同步方式的实现过程。

在没有多线程同步控制策略条件下的客户账户类定义框架代码如下所示：

```
public class RegisterAccount
{
float fBalance;
//客户缴费方法
public void deposit(float fFees){ fBalance += fFees; }
//通话计费方法
public void withdraw(float fFees){ fBalance -= fFees; }
… …
}
```

读者也许会认为：上述程序代码完全能够满足计费系统实际的需要。确实，在单线程环境下该程序确实是可靠的。但是，多进程并发运行的情况是怎样的呢？假设发生这种情况：客户在客户服务中心进行缴费的同时正在利用移动通信设备仅此通话，客户通话结束时计费系统启动计费进程，而同时服务中心的工作人员也提交缴费进程运行。读者可以看到如果发生这种情况，对客户账户的处理是不严肃的。

如何解决这种问题呢？很简单，在 RegisterAccount 类方法定义中加上用于标识同步方法的关键字 synchronized。这样，在同步方法执行过程中该方法涉及的共享资源（在上述代码中为 fBalance 成员变量）将被加上共享锁，以确保在方法运行期间只有该方法能够对共享资源进行访问，直到该方法的线程运行结束打开共享锁，其它线程才能够访问这些共享资源。在共享锁没有打开的时候其它访问共享资源的线程处于阻塞状态。

进行线程同步策略控制后的 RegisterAccount 类定义如下面代码所示：

```
public class RegisterAccount
{
float fBalance;
public synchronized void deposit(float fFees){ fBalance += fFees; }
public synchronized void withdraw(float fFees){ fBalance -= fFees; }
… …
}
```

从经过线程同步机制定义后的代码形式可以看出：在对共享资源进行访问的方法访问属性关键字（public）后附加同步定义关键字 synchronized，使得同步方法在对共享资源访问的时候，为这些敏感资源附加共享锁来控制方法执行期间的资源独占性，实现了应用系统数据资源的一致性管理和维护。

五、 Java 线程的管理

线程的状态控制

在这里需要明确的是：无论采用继承 Thread 类还是实现 Runnable 接口来实现应用程序的多线程能力，都需要在该类中定义用于完成实际功能的 run 方法，这个 run 方法称为线程体（Thread Body）。按照线程体在计算机系统内存中的状态不同，可以将线程分为创建、就绪、运行、睡眠、挂起和死亡等类型。这些线程状态类型下线程的特征为：

创建状态：当利用 new 关键字创建线程对象实例后，它仅仅作为一个对象实例存在，JVM 没有为其分配 CPU 时间片等线程运行资源；

就绪状态：在处于创建状态的线程中调用 start 方法将线程的状态转换为就绪状态。这时，线程已经得到除 CPU 时间之外的其它系统资源，只等 JVM 的线程调度器按照线程的优先级对该线程进行调度，从而使该线程拥有能够获得 CPU 时间片的机会。

睡眠状态：在线程运行过程中可以调用 sleep 方法并在方法参数中指定线程的睡眠时间将线程状态转换为睡眠状态。这时，该线程在不释放占用资源的情况下停止运行指定的睡眠时间。时间到达后，线程重新由 JVM 线程调度器进行调度和管理。

挂起状态：可以通过调用 suspend 方法将线程的状态转换为挂起状态。这时，线程将释放占用的所有资源，由 JVM 调度转入临时存储空间，直至应用程序调用 resume 方法恢复线程运行。

死亡状态：当线程体运行结束或者调用线程对象的 stop 方法后线程将终止运行，由 JVM 收回线程占用的资源。

在 Java 线程类中分别定义了相应的方法，用于在应用程序中对线程状态进行控制和管理。

线程的调度

线程调用的意义在于 JVM 应对运行的多个线程进行系统级的协调，以避免多个线程争用有限资源而导致应用系统死机或者崩溃。

为了线程对于操作系统和用户的重要性区分开，Java 定义了线程的优先级策略。Java 将线程的优先级分为 10 个等级，分别用 1-10 之间的数字表示。数字越大表明线程的级别越高。相应地，在 Thread 类中定义了表示线程最低、最高和普通优先级的成员变量 MIN_PRIORITY、MAX_PRIORITY 和 NORMAL_PRIORITY，代表的优先级等级分别为 1、10 和 5。当一个线程对象被创建时，其默认的线程优先级是 5。

为了控制线程的运行策略，Java 定义了线程调度器来监控系统中处于就绪状态的所有线程。线程调度器按照线程的优先级决定那个线程投入处理器运行。在多个线程处于就绪状态的条件下，具有高优先级的线程会在低优先级线程之前得到执行。线程调度器同样采用"抢占式"策略来调度线程执行，即当前线程执行过程中有较高优先级的线程进入就绪状态，则高优先级的线程立即被调度执行。具有相同优先级的所有线程采用轮转的方式来共同分配 CPU 时间片。

在应用程序中设置线程优先级的方法很简单，在创建线程对象之后可以调用线程对象的 setPriority 方法改变该线程的运行优先级，同样可以调用 getPriority 方法获取当前线程的优先级。

在 Java 中比较特殊的线程是被称为守护（Daemon）线程的低级别线程。这个线程具有最低的优先级，用于为系统中的其它对象和线程提供服务。将一个用户线程设置为守护线程的方式是在线程对象创建之前调用线程对象的 setDaemon 方法。典型的守护线程例子是 JVM 中的系统资源自动回收线程，它始终在低级别的状态中运行，用于实时监控和管理系统中的可回收资源。

线程分组管理

Java 定义了在多线程运行系统中的线程组（ThreadGroup）对象，用于实现按照特定功能对线程进行集中式分组管理。用户创建的每个线程均属于某线程组，这个线程组可以在线程创建时指定，也可以不指定线程组以使该线程处于默认的线程组之中。但是，一旦线程加入某线程组，该线程就一直存在于该线程组中直至线程死亡，不能在中途改变线程所属的线程组。

当 Java 的 Application 应用程序运行时，JVM 创建名称为 main 的线程组。除非单独指定，在该应用程序中创建的线程均属于 main 线程组。在 main 线程组中可以创建其它名称的线程组并将其它线程加入到该线程组中，依此类推，构成线程和线程组之间的树型管理和继承关系。

与线程类似，可以针对线程组对象进行线程组的调度、状态管理以及优先级设置等。在对线程组进行管理过程中，加入到某线程组中的所有线程均被看作统一的对象。

六、小结：

本文针对 Java 平台中线程的性质和应用程序的多线程策略进行了分析和讲解。

与其它操作系统环境不同，Java 运行环境中的线程类似于多用户、多任务操作系统环境下的进程，但在进程和线程的运行及创建方式等方面，进程与 Java 线程具有明显区别。

Unix 操作系统环境下，应用程序可以利用 fork 函数创建子进程，但子进程与该应用程序进程拥有独立的地址空间、系统资源和代码执行单元，并且进程的调度是由操作系统来完成的，使得在应用进程之间进行通信和线程协调相对复杂。而 Java 应用程序中的多线程则是共享同一应用系统资源的多个并行代码执行体，线程之间的通信和协调方法相对简单。

可以说：Java 语言对应用程序多线程能力的支持增强了 Java 作为网络程序设计语言的优势，为实现分布式应用系统中多客户端的并发访问以及提高服务器的响应效率奠定坚实基础。

Java 多线程程序设计初步

在 Java 语言产生前，传统的程序设计语言的程序同一时刻只能单任务操作，效率非常低，例如程序往往在接收数据输入时发生阻塞，只有等到程序获得数据后才 能继续运行。 随着 Internet 的迅猛发展，这种状况越来越不能让人们忍受：如果网络接收数据阻塞，后台程序就处于等待状态而不继续任何操作，而这种阻塞是经常会碰 到的，此时 CPU 资源被白白的闲置起来。如果在后台程序中能够同时处理多个任务，该多好啊！应 Internet 技术而生的 Java 语言解决了这个问题，多 线程程序是 Java 语言的一个很重要的特点。在一个 Java 程序中，我们可以同时并行运行多个相对独立的线程，例如，我们如果创建一个线程来进行数据输入输出，而创建另一个线程在后台进行其它的数据处理，如果输入输出线程在接收数据时阻塞，而处理数据的线程仍然在运行。多线程程序设计大大提高了程序执行效 率和处理能力。

一、线程的创建

我们知道 Java 是面向对象的程序语言，用 Java 进行程序设计就是设计和使用类，Java 为我们提供了线程类 Thread 来创建线程，创建线程与创建普通的类的对象的操作是一样的，而线程就是 Thread 类或其子类的实例对象。下面是一个创建启动一个线程的语句：

Thread thread1=new Thread(); file://声明一个对象实例，即创建一个线程；

Thread1.run(); file://用 Thread 类中的 run()方法启动线程；

从这个例子，我们可以通过 Thread()构造方法创建一个线程，并启动该线程。事实上，启动线程，也就是启动线程的 run()方法，而 Thread 类 中的 run()方法没有任何操作语句，所以这个线程没有任何操作。要使线程实现预定功能，必须定义自己的 run()方法。Java 中通常有两种方式定义 run()方法：

通过定义一个 Thread 类的子类，在该子类中重写 run()方法。Thread 子类的实例对象就是一个线程，显然，该线程有我们自己设计的线程体 run()方法，启动线程就启动了子类中重写的 run()方法。

通过 Runnable 接口，在该接口中定义 run()方法的接口。所谓接口跟类非常类似，主要用来实现特殊功能，如复杂关系的多重继承功能。在此，我们 定义一个实现 Runnable() 接口的类，在该类中定义自己的 run()方法，然后以该类的实例对象为参数调用 Thread 类的构造方法来创建一个线程。

线程被实际创建后处于待命状态，激活（启动）线程就是启动线程的 run()方法，这是通过调用线程的 start()方法来实现的。

下面一个例子实践了如何通过上述两种方法创建线程并启动它们：

```
// 通过 Thread 类的子类创建的线程；
   class thread1 extends Thread
      { file://自定义线程的 run()方法；
         public void run()
            {
               System.out.println("Thread1 is running…");
            }
         }
   file://通过 Runnable 接口创建的另外一个线程；
class thread2 implements Runnable
   { file://自定义线程的 run()方法；
      public void run()
      {
         System.out.println("Thread2 is running…");
      }
   }
   file://程序的主类'
   class Multi_Thread file://声明主类；
      {
         plubic static void mail(String args[]) file://声明主方法；
            {
               thread1 threadone=new thread1(); file://用 Thread 类的子类创建线程；
```

```
        Thread threadtwo=new Thread(new thread2()); file://用 Runnable 接口类的对象创建线程；
        threadone.start(); threadtwo.start(); file://strat()方法启动线程；
    }
}
```

运行该程序就可以看出，线程 threadone 和 threadtwo 交替占用 CPU，处于并行运行状态。可以看出，启动线程的 run()方法是通过调用 线程的 start()方法来实现的(见上例中主类)，调用 start()方法启动线程的 run()方法不同于一般的调用方法，调用一般方法时，必须等到一般方法执行完毕才能够返回 start()方法，而启动线程的 run()方法后，start()告诉系统该线程准备就绪可以启动 run()方法后，就返回 start()方法执行调用 start()方法语句下面的语句，这时 run()方法可能还在运行，这样，线程的启动和运行并行进行，实现了多任务操作。

二、线程的优先级

对于多线程程序，每个线程的重要程度是不尽相同，如多个线程在等待获得 CPU 时间时，往往我们需要优先级高的线程优先抢占到 CPU 时间得以执行；又如多个线程交替执行时，优先级决定了级别高的线程得到 CPU 的次数多一些且时间多长一 些；这样，高优先级的线程处理的任务效率就高一些。

Java 中线程的优先级从低到高以整数 1~10 表示，共分为 10 级，设置优先级是通过调用线程对象的 setPriority()方法，如上例中，设置优先级的语句为：

```
thread1 threadone=new thread1(); file://用 Thread 类的子类创建线程；
Thread threadtwo=new Thread(new thread2()); file://用 Runnable 接口类的对象创建线程；
threadone.setPriority(6); file://设置 threadone 的优先级 6；
threadtwo.setPriority(3); file://设置 threadtwo 的优先级 3；
threadone.start(); threadtwo.start(); file://strat()方法启动线程；
```

这样，线程 threadone 将会优先于线程 threadtwo 执行，并将占有更多的 CPU 时间。该例中，优先级设置放在线程启动前，也可以在启动后进行设置，以满足不同的优先级需求。

三、线程的（同步）控制

一个 Java 程序的多线程之间可以共享数据。当线程以异步方式访问共享数据时，有时候是不安全 的或者不和逻辑的。比如，同一时刻一个线程在读取数据，另外一个线程在处理数据，当处理数据的线程没有等到读取数据的线程读取完毕就去处理数据，必然得到 错误的处理结果。这和我们前面提到的读取数据和处理数据并行多任务并不矛盾，这儿指的是处理数据的线程不能处理当前还没有读取结束的数据，但是可以处理其 它的数据。

如果我们采用多线程同步控制机制，等到第一个线程读取完数据，第二个线程才能处理该数据，就会避免错误。可见，线程同步是多线程编程的一个相当重要的技术。

在讲线程的同步控制前我们需要交代如下概念：

1 **用 Java 关键字 synchonized 同步对共享数据操作的方法**

在一个对象中，用 synchonized 声明的方法为同步方法。Java 中有一个同步模型-监视器，负责管理线程对对象中的同步方法的访问，它的原理 是：赋予该对象唯一一把'钥匙'，当多个线程进入对象，只有取得该对象钥匙的线程才可以访问同步方法，其它线程在该对象中等待，直到该线程用 wait() 方法放弃这把钥匙，其它等待的线程抢占该钥匙，抢占到钥匙的线程后才可得以执行，而没有取得钥匙的线程仍被阻塞在该对象中等待。

file://声明同步的一种方式：将方法声明同步

```
class store
  {
    public synchonized void store_in()
    {
        ….
    }
    public synchonized void store_out(){
        ….}
    }
```

2 **利用 wait()、notify()及 notifyAll()方法发送消息实现线程间的相互联系**

Java 程序中多个线程通过消息来实现互动联系的，这几种方法实现了线程间的消息发送。例如定义一个对象的 synchonized 方法，同一时刻只能够有一个线程访问该对象中的同步方法，其它线程被阻塞。通常可以用 notify()或 notifyAll()方法唤醒其它一个或所有线 程。而使用 wait()方法来使该线程处于阻塞状态，等待其它的线程用 notify()唤醒。

一个实际的例子就是生产和销售，生产单元将产品生产出来放在仓库中，销售单元则从仓库中提走产品，在这个过程中，销售单元必须在仓库中有产品时才能提货；如果仓库中没有产品，则销售单元必须等待。

程序中，假如我们定义一个仓库类 store，该类的实例对象就相当于仓库，在 store 类中定义两个成员方法：store_in()，用来模拟产品制造 者往仓库中添加产品；strore_out()方法则用来模拟销售者从仓库中取走产品。然后定义两个线程类：customer 类，其中的 run()方法通 过调用仓库类中的 store_out()从仓库中取走产品，模拟销售者；另外一个线程类 producer 中的 run()方法通过调用仓库类中的 store_in()方法向仓库添加产品，模拟产品制造者。在主类中创建并启动线程，实现向仓库中添加产品或取走产品。

如果仓库类中的 store_in() 和 store_out()方法不声明同步，这就是个一般的多线程，我们知道，一个程序中的多线程是交替执行的，运行也是无序的，这样，就可能存在这样的问题：

仓库中没有产品了，销售者还在不断光顾，而且还不停的在'取'产品，这在现实中是不可思义的,在程序中就表现为负值;如果将仓库类中的 stroe_in()和 store_out()方法声明同步，如上例所示：就控制了同一时刻只能有一个线程访问仓库对象中的同步方法；即一个生产类线程访 问被声明为同步的 store_in()方法时，其它线程将不能够访问对象中的 store_out()同步方法，当然也不能访问 store_in()方法。 必须等到该线程调用 wait()方法放弃钥匙，其它线程才有机会访问同步方法。

这个原理实际中也很好理解，当生产者 （producer）取得仓库唯一的钥匙，就向仓库中添放产品，此时其它的销售者（customer，可以是一个或多个）不可能取得钥匙，只有当生产者添 放产品结束，交还钥匙并且通知销售者，不同的销售者根据取得钥匙的先后与否决定是否可以进入仓库中提走产品。

彻底明白 Java 的多线程-线程间的通信

一、实现多线程

1. 虚假的多线程

例 1：

```
publicclassTestThread
{
inti=0, j=0;
publicvoidgo(intflag){
while(true){
try{
java/lang/Thread.java.html" target="_blank">
Thread
.sleep(100);
}
catch(java/lang/InterruptedException.java.html" target="_blank">
InterruptedException
e){
java/lang/System.java.html" target="_blank">
System
.out.println("Interrupted");
}
if(flag==0)
i++;
java/lang/System.java.html" target="_blank">
System
.out.println("i=" + i);
}
else{
j++;
java/lang/System.java.html" target="_blank">
System
.out.println("j=" + j);
}
}
}
publicstaticvoidmain(java/lang/String.java.html" target="_blank">
String
[] args){
newTestThread().go(0);
newTestThread().go(1);
}
}
```

上面程序的运行结果为：

i=1

i=2

i=3

。。。

结果将一直打印出 I 的值。我们的意图是当在 while 循环中调用 sleep()时，另一个线程就将起动，打印出 j 的值，但结果却并不是这样。关于 sleep()为什么不会出现我们预想的结果，在下面将讲到。

2. 实现多线程

通过继承 class　Thread 或实现 Runnable 接口，我们可以实现多线程

2.1 通过继承 class　Thread 实现多线程

class　Thread 中有两个最重要的函数 run()和 start()。

1) run()函数必须进行覆写，把要在多个线程中并行处理的代码放到这个函数中。

2) 虽然 run()函数实现了多个线程的并行处理，但我们不能直接调用 run()函数，而是通过调用 start()函数来调用 run()函数。在调用 start()的时候，start()函数会首先进行与多线程相关的初始化（这也是为什么不能直接调用 run()函数的原因），然后再调用 run()函 数。

例 2：

```
publicclassTestThread                    extendsjava/lang/Thread.java.html"
target="_blank">
Thread
{
privatestaticintthreadCount = 0;
privateintthreadNum = ++threadCount;
privateinti = 5;
publicvoidrun(){
while(true){
try{
java/lang/Thread.java.html" target="_blank">
Thread
.sleep(100);
}
catch(java/lang/InterruptedException.java.html" target="_blank">
InterruptedException
e){
java/lang/System.java.html" target="_blank">
System
.out.println("Interrupted");
}
java/lang/System.java.html" target="_blank">
System
.out.println("Thread " + threadNum + " = " + i);
if(--i==0) return;
}
}
>publicstaticvoidmain(java/lang/String.java.html" target="_blank">
String
```

```
[] args){
for(inti=0; i<5; i++)
newTestThread().start();
}
}
```
运行结果为：
Thread 1 = 5
Thread 2 = 5
Thread 3 = 5
Thread 4 = 5
Thread 5 = 5
Thread 1 = 4
Thread 2 = 4
Thread 3 = 4
Thread 4 = 4
Thread 1 = 3
Thread 2 = 3
Thread 5 = 4
Thread 3 = 3
Thread 4 = 3
Thread 1 = 2
Thread 2 = 2
Thread 5 = 3
Thread 3 = 2
Thread 4 = 2
Thread 1 = 1
Thread 2 = 1
Thread 5 = 2
Thread 3 = 1
Thread 4 = 1
Thread 5 = 1
从结果可见，例 2 能实现多线程的并行处理。
＊＊： 在上面的例子中，我们只用 new 产生 Thread 对象，并没有用 reference 来记录所产生的 Thread 对象。根据垃圾回收机制，当一个对象没有被 reference 引用时，它将被回收。但是垃圾回收机制对 Thread 对象“不成立”。因为每一个 Thread 都会进行注册动作，所以即使我们在产生 Thread 对象时没有指定一个 reference 指向这个对象，实际上也会在某个地方有个指向该对象的 reference，所以垃圾回收器无法回收它 们。
3) 通过 Thread 的子类产生的线程对象是不同对象的线程

```
classTestSynchronized                extendsjava/lang/Thread.java.html″
target=″_blank″>
Thread
{
publicTestSynchronized(java/lang/String.java.html″ target=″_blank″>
```

```
String
name){
super(name);
}
publicsynchronizedstaticvoidprt(){
for(inti=10; i<20; i++){
java/lang/System.java.html" target="_blank">
System
.out.println(java/lang/Thread.java.html" target="_blank">
Thread
.currentThread().getName() + " : " + i);
try{
java/lang/Thread.java.html" target="_blank">
Thread
.sleep(100);
}
catch(java/lang/InterruptedException.java.html" target="_blank">
InterruptedException
e){
java/lang/System.java.html" target="_blank">
System
.out.println("Interrupted");
}
}
}
publicsynchronizedvoidrun(){
for(inti=0; i<3; i++){
java/lang/System.java.html" target="_blank">
System
.out.println(java/lang/Thread.java.html" target="_blank">
Thread
.currentThread().getName() + " : " + i);
try{
java/lang/Thread.java.html" target="_blank">
Thread
.sleep(100);
}
catch(java/lang/InterruptedException.java.html" target="_blank">
InterruptedException
e){
java/lang/System.java.html" target="_blank">
System
.out.println("Interrupted");
}
```

```
}
}
}
publicclassTestThread{
publicstaticvoidmain(java/lang/String.java.html" target="_blank">
String
[] args){
TestSynchronized t1 = newTestSynchronized("t1");
TestSynchronized t2 = newTestSynchronized("t2");
t1.start();
t1.start();  //（1）
//t2.start();  （2）
}
}
```

运行结果为：

t1 : 0
t1 : 1
t1 : 2
t1 : 0
t1 : 1
t1 : 2

由于是同一个对象启动的不同线程，所以 run()函数实现了 synchronized。如果去掉（2）的注释，把代码（1）注释掉，结果将变为：

t1 : 0
t2 : 0
t1 : 1
t2 : 1
t1 : 2
t2 : 2

由于 t1 和 t2 是两个对象，所以它们所启动的线程可同时访问 run()函数。

2.2 通过实现 Runnable 接口实现多线程

如果有一个类，它已继承了某个类，又想实现多线程，那就可以通过实现 Runnable 接口来实现。

1) Runnable 接口只有一个 run()函数。

2) 把一个实现了 Runnable 接口的对象作为参数产生一个 Thread 对象，再调用 Thread 对象的 start()函数就可执行并行操作。如果在产生一 个 Thread 对象时以一个 Runnable 接口的实现类的对象作为参数，那么在调用 start()函数时，start()会调用 Runnable 接口的 实现类中的 run()函数。

例 3.1：

```
publicclassTestThread          implementsjava/lang/Runnable.java.html"
target="_blank">
Runnable
{
```

```
privatestaticintthreadCount = 0;
privateintthreadNum = ++threadCount;
privateinti = 5;
publicvoidrun(){
while(true){
try{
java/lang/Thread.java.html" target="_blank">
Thread
.sleep(100);
}
catch(java/lang/InterruptedException.java.html" target="_blank">
InterruptedException
e){
java/lang/System.java.html" target="_blank">
System
.out.println("Interrupted");
}
java/lang/System.java.html" target="_blank">
System
.out.println("Thread " + threadNum + " = " + i);
if(--i==0) return;
}
}
publicstaticvoidmain(java/lang/String.java.html" target="_blank">
String
[] args){
for(inti=0; i<5; i++)
newjava/lang/Thread.java.html" target="_blank">
Thread
(newTestThread()).start();  // (1)
}
}
```

运行结果为：

Thread 1 = 5
Thread 2 = 5
Thread 3 = 5
Thread 4 = 5
Thread 5 = 5
Thread 1 = 4
Thread 2 = 4
Thread 3 = 4
Thread 4 = 4
Thread 4 = 3
Thread 5 = 4

Thread 1 = 3
Thread 2 = 3
Thread 3 = 3
Thread 4 = 2
Thread 5 = 3
Thread 1 = 2
Thread 2 = 2
Thread 3 = 2
Thread 4 = 1
Thread 5 = 2
Thread 1 = 1
Thread 2 = 1
Thread 3 = 1
Thread 5 = 1

例 3 是对例 2 的修改，它通过实现 Runnable 接口来实现并行处理。代码（1）处可见，要调用 TestThread 中的并行操作部分，要把一个 TestThread 对象作为参数来产生 Thread 对象，再调用 Thread 对象的 start()函数。

3) 同一个实现了 Runnable 接口的对象作为参数产生的所有 Thread 对象是同一对象下的线程。

例 3.2：

```
packagemypackage1;
publicclassTestThread          implementsjava/lang/Runnable.java.html"
target="_blank">
Runnable
{
publicsynchronizedvoidrun(){
for(inti=0; i<5; i++){
java/lang/System.java.html" target="_blank">
System
.out.println(java/lang/Thread.java.html" target="_blank">
Thread
.currentThread().getName() + " : " + i);
try{
java/lang/Thread.java.html" target="_blank">
Thread
.sleep(100);
}
catch(java/lang/InterruptedException.java.html" target="_blank">
InterruptedException
e){
java/lang/System.java.html" target="_blank">
System
.out.println("Interrupted");
```

```
}
}
}
publicstaticvoidmain(java/lang/String.java.html" target="_blank">
String
[] args){
TestThread testThread = newTestThread();
for(inti=0; i<5; i++)
//new Thread(testThread, "t" + i).start();   (1)
newjava/lang/Thread.java.html" target="_blank">
Thread
(newTestThread(), "t" + i).start();   (2)
}
}
```

运行结果为：

```
t0 : 0
t1 : 0
t2 : 0
t3 : 0
t4 : 0
t0 : 1
t1 : 1
t2 : 1
t3 : 1
t4 : 1
t0 : 2
t1 : 2
t2 : 2
t3 : 2
t4 : 2
t0 : 3
t1 : 3
t2 : 3
t3 : 3
t4 : 3
t0 : 4
t1 : 4
t2 : 4
t3 : 4
t4 : 4
```

由于代码（2）每次都是用一个新的 TestThread 对象来产生 Thread 对象的，所以产生出来的 Thread 对象是不同对象的线程，所以所有 Thread 对象都可同时访问 run()函数。如果注释掉代码（2），并去掉代码（1）的注释，结果为：

```
t0 : 0
```

t0 : 1
t0 : 2
t0 : 3
t0 : 4
t1 : 0
t1 : 1
t1 : 2
t1 : 3
t1 : 4
t2 : 0
t2 : 1
t2 : 2
t2 : 3
t2 : 4
t3 : 0
t3 : 1
t3 : 2
t3 : 3
t3 : 4
t4 : 0
t4 : 1
t4 : 2
t4 : 3
t4 : 4

由于代码（1）中每次都是用同一个 TestThread 对象来产生 Thread 对象的，所以产生出来的 Thread 对象是同一个对象的线程，所以实现 run()函数的同步。

二、共享资源的同步

1. 同步的必要性

例 4：

```
classSeq{
privatestaticintnumber = 0;
privatestaticSeq seq = newSeq();
privateSeq() {}
publicstaticSeq getInstance(){
returnseq;
}
publicintget(){
number++;   // (a)
returnnumber;  // (b)
```

```
}
}
publicclassTestThread{
publicstaticvoidmain(java/lang/String.java.html" target="_blank">
String
[] args){
Seq.getInstance().get();  // (1)
Seq.getInstance().get();  // (2)
}
}
```

上面是一个取得序列号的单例模式的例子，但调用 get()时，可能会产生两个相同的序列号：

当代码（1）和（2）都试图调用 get()取得一个唯一的序列。当代码（1）执行完代码（a），正要执行代码（b）时，它被中断了并开始执行代码（2）。一旦当代码（2）执行完（a）而代码（1）还未执行代码（b），那么代码（1）和代码（2）就将得到相同的值。

2. 通过 synchronized 实现资源同步

2.1 锁标志

2.1.1 每个对象都有一个标志锁。当对象的一个线程访问了对象的某个 synchronized 数据（包括函数）时，这个对象就将被“上锁”，所以被声明为 synchronized 的数据（包括函数）都不能被调用（因为当前线程取走了对象的“锁标志”）。只有当前线程访问完它要访问的 synchronized 数据，释放“锁标志”后，同一个对象的其它线程才能访问 synchronized 数据。

2.1.2 每个 class 也有一个“锁标志”。对于 synchronized static 数据（包括函数）可以在整个 class 下进行锁定，避免 static 数据的同时访问。

例 5：

```
classSeq{
privatestaticintnumber = 0;
privatestaticSeq seq = newSeq();
privateSeq() {}
publicstaticSeq getInstance(){
returnseq;
}
publicsynchronizedintget(){ // (1)
number++;
returnnumber;
}
}
```

例 5 在例 4 的基础上，把 get()函数声明为 synchronized，那么在同一个对象中，就只能有一个线程调用 get()函数，所以每个线程取得的 number 值就是唯一的了。

例 6：

```
classSeq{
```

```
privatestaticintnumber = 0;
privatestaticSeq seq = null;
privateSeq() {}
synchronizedpublicstaticSeq getInstance(){ // (1)
if(seq==null) seq = newSeq();
returnseq;
}
publicsynchronizedintget(){
number++;
returnnumber;
}
}
```

例 6 把 getInstance()函数声明为 synchronized，那样就保证通过 getInstance()得到的是同一个 seq 对象。

2.2 non-static 的 synchronized 数据只能在同一个对象的纯种实现同步访问，不同对象的线程仍可同时访问。

例 7：

```
classTestSynchronized      implementsjava/lang/Runnable.java.html"
target="_blank">
Runnable
{
publicsynchronizedvoidrun(){  // (1)
for(inti=0; i<10; i++){
java/lang/System.java.html" target="_blank">
System
.out.println(java/lang/Thread.java.html" target="_blank">
Thread
.currentThread().getName() + " : " + i);
/* (2) */
try{
java/lang/Thread.java.html" target="_blank">
Thread
.sleep(100);
}
catch(java/lang/InterruptedException.java.html" target="_blank">
InterruptedException
e){
java/lang/System.java.html" target="_blank">
System
.out.println("Interrupted");
}
}
}
```

```
}
publicclassTestThread{
publicstaticvoidmain(java/lang/String.java.html" target="_blank">
String
[] args){
TestSynchronized r1 = newTestSynchronized();
TestSynchronized r2 = newTestSynchronized();
java/lang/Thread.java.html" target="_blank">
Thread
t1 = newjava/lang/Thread.java.html" target="_blank">
Thread
(r1, "t1");
java/lang/Thread.java.html" target="_blank">
Thread
t2 = newjava/lang/Thread.java.html" target="_blank">
Thread
(r2, "t2");  //(3)
//Thread t2 = new Thread(r1, "t2");   (4)
t1.start();
t2.start();
}
}
```

运行结果为：

```
t1 : 0
t2 : 0
t1 : 1
t2 : 1
t1 : 2
t2 : 2
t1 : 3
t2 : 3
t1 : 4
t2 : 4
t1 : 5
t2 : 5
t1 : 6
t2 : 6
t1 : 7
t2 : 7
t1 : 8
t2 : 8
t1 : 9
t2 : 9
```

虽 然我们在代码（1）中把 run()函数声明为 synchronized，但由于 t1、t2 是两个对象（r1、

r2）的线程，而 run()函数是 non- static 的 synchronized 数据，所以仍可被同时访问（代码（2）中的 sleep()函数由于在暂停时不会释放“标志锁”，因为线程中的循环 很难被中断去执行另一个线程，所以代码（2）只是为了显示结果）。

如果把例 7 中的代码（3）注释掉，并去年代码（4）的注释，运行结果将为：

t1 : 0
t1 : 1
t1 : 2
t1 : 3
t1 : 4
t1 : 5
t1 : 6
t1 : 7
t1 : 8
t1 : 9
t2 : 0
t2 : 1
t2 : 2
t2 : 3
t2 : 4
t2 : 5
t2 : 6
t2 : 7
t2 : 8
t2 : 9

修改后的 t1、t2 是同一个对象（r1）的线程，所以只有当一个线程（t1 或 t2 中的一个）执行 run()函数，另一个线程才能执行。

2.3 对象的“锁标志”和 class 的“锁标志”是相互独立的。

例 8：

```
classTestSynchronized extendsjava/lang/Thread.java.html"
target="_blank">
Thread
{
publicTestSynchronized(java/lang/String.java.html"
target="_blank">
String
name){
super(name);
}
publicsynchronizedstaticvoidprt(){
for(inti=10; i<20; i++){
java/lang/System.java.html" target="_blank">
System
.out.println(java/lang/Thread.java.html" target="_blank">
```

```
Thread
.currentThread().getName() + " : " + i);
try{
java/lang/Thread.java.html" target="_blank">
Thread
.sleep(100);
}
catch(java/lang/InterruptedException.java.html" target="_blank">
InterruptedException
e){
java/lang/System.java.html" target="_blank">
System
.out.println("Interrupted");
}
}
}
publicsynchronizedvoidrun(){
for(inti=0; i<10; i++){
java/lang/System.java.html" target="_blank">
System
.out.println(java/lang/Thread.java.html" target="_blank">
Thread
.currentThread().getName() + " : " + i);
try{
java/lang/Thread.java.html" target="_blank">
Thread
.sleep(100);
}
catch(java/lang/InterruptedException.java.html" target="_blank">
InterruptedException
e){
java/lang/System.java.html" target="_blank">
System
.out.println("Interrupted");
}
}
}
}
publicclassTestThread{
publicstaticvoidmain(java/lang/String.java.html" target="_blank">
String
[] args){
TestSynchronized t1 = newTestSynchronized("t1");
TestSynchronized t2 = newTestSynchronized("t2");
```

```
t1.start();
t1.prt();  //（1）
t2.prt();  //（2）
}
}
```

运行结果为：

```
main : 10
t1 : 0
main : 11
t1 : 1
main : 12
t1 : 2
main : 13
t1 : 3
main : 14
t1 : 4
main : 15
t1 : 5
main : 16
t1 : 6
main : 17
t1 : 7
main : 18
t1 : 8
main : 19
t1 : 9
main : 10
main : 11
main : 12
main : 13
main : 14
main : 15
main : 16
main : 17
main : 18
main : 19
```

在代码（1）中，虽然是通过对象 t1 来调用 prt()函数的，但由于 prt()是静态的，所以调用它时不用经过任何对象，它所属的线程为 main 线程。

由 于调用 run()函数取走的是对象锁，而调用 prt()函数取走的是 class 锁，所以同一个线程 t1（由上面可知实际上是不同线程）调用 run()函数 且还没完成 run()函数时，它就能调用 prt()函数。但 prt()函数只能被一个线程调用，如代码（1）和代码（2），即使是两个不同的对象也不能同 时调用 prt()。

3. 同步的优化

1) synchronized block

语法为：synchronized(reference){ do this }

reference 用来指定“以某个对象的锁标志”对“大括号内的代码”实施同步控制。

例 9：

```
classTestSynchronized implementsjava/lang/Runnable.java.html" target="_blank">
Runnable
{
staticintj = 0;
publicsynchronizedvoidrun(){
for(inti=0; i<5; i++){
// (1)
java/lang/System.java.html" target="_blank">
System
.out.println(java/lang/Thread.java.html" target="_blank">
Thread
.currentThread().getName() + " : " + j++);
try{
java/lang/Thread.java.html" target="_blank">
Thread
.sleep(100);
}
catch(java/lang/InterruptedException.java.html" target="_blank">
InterruptedException
e){
java/lang/System.java.html" target="_blank">
System
.out.println("Interrupted");
}
}
}
}
publicclassTestThread{
publicstaticvoidmain(java/lang/String.java.html" target="_blank">
String
[] args){
TestSynchronized r1 = newTestSynchronized();
TestSynchronized r2 = newTestSynchronized();
java/lang/Thread.java.html" target="_blank">
Thread
```

```
t1 = newjava/lang/Thread.java.html" target="_blank">
Thread
(r1, "t1");
java/lang/Thread.java.html" target="_blank">
Thread
t2 = newjava/lang/Thread.java.html" target="_blank">
Thread
(r1, "t2");
t1.start();
t2.start();
}
}
```

运行结果为：

```
t1 : 0
t1 : 1
t1 : 2
t1 : 3
t1 : 4
t2 : 5
t2 : 6
t2 : 7
t2 : 8
t2 : 9
```

上面的代码的 run()函数实现了同步，使每次打印出来的 j 总是不相同的。但实际上在整个 run()函数中，我们只关心 j 的同步，而其余代码同步与否我们是不关心的，所以可以对它进行以下修改：

```
classTestSynchronized      implementsjava/lang/Runnable.java.html"
target="_blank">
Runnable
{
staticintj = 0;
publicvoidrun(){
for(inti=0; i<5; i++){
// (1)
synchronized(this){
java/lang/System.java.html" target="_blank">
System
.out.println(java/lang/Thread.java.html" target="_blank">
Thread
.currentThread().getName() + " : " + j++);
}
try{
java/lang/Thread.java.html" target="_blank">
```

```
Thread
.sleep(100);
}
catch(java/lang/InterruptedException.java.html" target="_blank">
InterruptedException
e){
java/lang/System.java.html" target="_blank">
System
.out.println("Interrupted");
}
}
}
}
publicclassTestThread{
publicstaticvoidmain(java/lang/String.java.html" target="_blank">
String
[] args){
TestSynchronized r1 = newTestSynchronized();
TestSynchronized r2 = newTestSynchronized();
java/lang/Thread.java.html" target="_blank">
Thread
t1 = newjava/lang/Thread.java.html" target="_blank">
Thread
(r1, "t1");
java/lang/Thread.java.html" target="_blank">
Thread
t2 = newjava/lang/Thread.java.html" target="_blank">
Thread
(r1, "t2");
t1.start();
t2.start();
}
}
```

运行结果为：

```
t1 : 0
t2 : 1
t1 : 2
t2 : 3
t1 : 4
t2 : 5
t1 : 6
t2 : 7
t1 : 8
t2 : 9
```

由于进行同步的范围缩小了，所以程序的效率将提高。同时，代码（1）指出，当对大括号内的 println()语句进行同步控制时，会取走当前对象的“锁标志”，即对当前对象“上锁”，不让当前对象下的其它线程执行当前对象的其它 synchronized 数据。

三． 线程间的通信

1. 线程的几种状态

线程有四种状态，任何一个线程肯定处于这四种状态中的一种：

1) 产生（New）：线程对象已经产生，但尚未被启动，所以无法执行。如通过 new 产生了一个线程对象后没对它调用 start()函数之前。

2) 可执行（Runnable）：每个支持多线程的系统都有一个排程器，排程器会从线程池中选择一个线程并启动它。当一个线程处于可执行状态时，表示它可能正 处于线程池中等待排排程器启动它；也可能它已正在执行。如执行了一个线程对象的 start()方法后，线程就处于可执行状态，但显而易见的是此时线程不一 定正在执行中。

3) 死亡（Dead）：当一个线程正常结束，它便处于死亡状态。如一个线程的 run()函数执行完毕后线程就进入死亡状态。

4) 停滞（Blocked）：当一个线程处于停滞状态时，系统排程器就会忽略它，不对它进行排程。当处于停滞状态的线程重新回到可执行状态时，它有可能重新执 行。如通过对一个线程调用 wait()函数后，线程就进入停滞状态，只有当两次对该线程调用 notify 或 notifyAll 后它才能两次回到可执行状 态。

2. class Thread 下的常用函数函数

2.1 suspend()、resume()

1) 通过 suspend()函数，可使线程进入停滞状态。通过 suspend()使线程进入停滞状态后，除非收到 resume()消息，否则该线程不会变回可执行状态。

2) 当调用 suspend()函数后，线程不会释放它的“锁标志”。

例 11：

```
class TestThreadMethod extends Thread{
public static int shareVar = 0;
public TestThreadMethod(String name){
super(name);
}
public synchronized void run(){
if(shareVar==0){
for(int i=0; i<5; i++){
shareVar++;
if(shareVar==5){
this.suspend();  // (1)
}
}
```

```
}
else{
System.out.print(Thread.currentThread().getName());
System.out.println(" shareVar = " + shareVar);
this.resume();  //（2）
}
}
}
public class TestThread{
public static void main(String[] args){
TestThreadMethod t1 = new TestThreadMethod("t1");
TestThreadMethod t2 = new TestThreadMethod("t2");
t1.start();  //（5）
//t1.start();  //（3）
t2.start();  //（4）
}
}
```

运行结果为：

t2 shareVar = 5

i. 当代码（5）的 t1 所产生的线程运行到代码（1）处时，该线程进入停滞状态。然后排程器从线程池中唤起代码（4）的 t2 所产生的线程，此时 shareVar 值不为 0，所以执行 else 中的语句。

ii. 也许你会问，那执行代码（2）后为什么不会使 t1 进入可执行状态呢？正如前面所说，t1 和 t2 是两个不同对象的线程，而代码（1）和（2）都只对当前对象 进行操作，所以 t1 所产生的线程执行代码（1）的结果是对象 t1 的当前线程进入停滞状态；而 t2 所产生的线程执行代码（2）的结果是把对象 t2 中的所有处 于停滞状态的线程调回到可执行状态。

iii. 那现在把代码（4）注释掉，并去掉代码（3）的注释，是不是就能使 t1 重新回到可执行状态呢？运行结果是什么也不输出。为什么会这样呢？也许你会认为，当 代码（5）所产生的线程执行到代码（1）时，它进入停滞状态；而代码（3）所产生的线程和代码（5）所产生的线程是属于同一个对象的，那么就当代码（3） 所产生的线程执行到代码（2）时，就可使代码（5）所产生的线程执行回到可执行状态。但是要清楚，suspend()函数只是让当前线程进入停滞状态，但 并不释放当前线程所获得的“锁标志”。所以当代码（5）所产生的线程进入停滞状态时，代码（3）所产生的线程仍不能启动，因为当前对象的“锁标志”仍被代 码（5）所产生的线程占有。

2.2 sleep()

1) sleep ()函数有一个参数，通过参数可使线程在指定的时间内进入停滞状态，当指定的时间过后，线程则自动进入可执行状态。

2) 当调用 sleep ()函数后，线程不会释放它的“锁标志”。

例 12：

```
class TestThreadMethod extends Thread{
class TestThreadMethod extends Thread{
public static int shareVar = 0;
public TestThreadMethod(String name){
super(name);
```

```
}
public synchronized void run(){
for(int i=0; i<3; i++){
System.out.print(Thread.currentThread().getName());
System.out.println(" : " + i);
try{
Thread.sleep(100);  // (4)
}
catch(InterruptedException e){
System.out.println("Interrupted");
}
}
}
}
public class TestThread{
public static void main(String[] args){
TestThreadMethod t1 = new TestThreadMethod("t1");
TestThreadMethod t2 = new TestThreadMethod("t2");
t1.start();  (1)
t1.start();  (2)
//t2.start();  (3)
}
}
```

运行结果为：

t1 : 0
t1 : 1
t1 : 2
t1 : 0
t1 : 1
t1 : 2

由结果可证明，虽然在 run()中执行了 sleep()，但是它不会释放对象的“锁标志”，所以除非代码(1)的线程执行完 run()函数并释放对象的“锁标志”，否则代码（2）的线程永远不会执行。

如果把代码（2）注释掉，并去掉代码（3）的注释，结果将变为：

t1 : 0
t2 : 0
t1 : 1
t2 : 1
t1 : 2
t2 : 2

由于 t1 和 t2 是两个对象的线程，所以当线程 t1 通过 sleep()进入停滞时，排程器会从线程池中调用其它的可执行线程，从而 t2 线程被启动。

例 13：

```
class TestThreadMethod extends Thread{
```

```
public static int shareVar = 0;
public TestThreadMethod(String name){
super(name);
}
public synchronized void run(){
for(int i=0; i<5; i++){
System.out.print(Thread.currentThread().getName());
System.out.println(" : " + i);
try{
if(Thread.currentThread().getName().equals("t1"))
Thread.sleep(200);
else
Thread.sleep(100);
}
catch(InterruptedException e){
System.out.println("Interrupted");
}
}
}
}
public class TestThread{
public static void main(String[] args){
TestThreadMethod t1 = new TestThreadMethod("t1");
TestThreadMethod t2 = new TestThreadMethod("t2");
t1.start();
//t1.start();
t2.start();
}
}
```

运行结果为：

t1 : 0
t2 : 0
t2 : 1
t1 : 1
t2 : 2
t2 : 3
t1 : 2
t2 : 4
t1 : 3
t1 : 4

由于线程 t1 调用了 sleep(200)，而线程 t2 调用了 sleep(100)，所以线程 t2 处于停滞状态的时间是线程 t1 的一半，从从结果反映出来的就是线程 t2 打印两倍次线程 t1 才打印一次。

2.3 yield()

1) 通过 yield ()函数，可使线程进入可执行状态，排程器从可执行状态的线程中重新进

行排程。所以调用了 yield()的函数也有可能马上被执行。

2) 当调用 yield ()函数后，线程不会释放它的“锁标志”。

例 14：

```
class TestThreadMethod extends Thread{
public static int shareVar = 0;
public TestThreadMethod(String name){
super(name);
}
public synchronized void run(){
for(int i=0; i<4; i++){
System.out.print(Thread.currentThread().getName());
System.out.println(" : " + i);
Thread.yield();
}
}
}
public class TestThread{
public static void main(String[] args){
TestThreadMethod t1 = new TestThreadMethod("t1");
TestThreadMethod t2 = new TestThreadMethod("t2");
t1.start();
t1.start();  // (1)
//t2.start();   (2)
}
}
```

运行结果为：

```
t1 : 0
t1 : 1
t1 : 2
t1 : 3
t1 : 0
t1 : 1
t1 : 2
t1 : 3
```

从结果可知调用 yield()时并不会释放对象的“锁标志”。

如果把代码（1）注释掉，并去掉代码（2）的注释，结果为：

```
t1 : 0
t1 : 1
t2 : 0
t1 : 2
t2 : 1
t1 : 3
t2 : 2
t2 : 3
```

从结果可知，虽然 t1 线程调用了 yield()，但它马上又被执行了。

2.4 sleep()和 yield()的区别

1) sleep()使当前线程进入停滞状态，所以执行 sleep()的线程在指定的时间内肯定不会执行；yield()只是使当前线程重新回到可执行状态，所以执行 yield()的线程有可能在进入到可执行状态后马上又被执行。

2) sleep()可使优先级低的线程得到执行的机会，当然也可以让同优先级和高优先级的线程有执行的机会；yield()只能使同优先级的线程有执行的机会。

例 15：

```
class TestThreadMethod extends Thread{
public static int shareVar = 0;
public TestThreadMethod(String name){
super(name);
}
public void run(){
for(int i=0; i<4; i++){
System.out.print(Thread.currentThread().getName());
System.out.println(" : " + i);
//Thread.yield();  (1)
/* (2) */
try{
Thread.sleep(3000);
}
catch(InterruptedException e){
System.out.println("Interrupted");
}

}
}
}
public class TestThread{
public static void main(String[] args){
TestThreadMethod t1 = new TestThreadMethod("t1");
TestThreadMethod t2 = new TestThreadMethod("t2");
t1.setPriority(Thread.MAX_PRIORITY);
t2.setPriority(Thread.MIN_PRIORITY);
t1.start();
t2.start();
}
}
```

运行结果为：

```
t1 : 0
t1 : 1
t2 : 0
t1 : 2
```

t2 : 1
t1 : 3
t2 : 2
t2 : 3

由结果可见，通过 sleep()可使优先级较低的线程有执行的机会。注释掉代码（2），并去掉代码（1）的注释，结果为：

t1 : 0
t1 : 1
t1 : 2
t1 : 3
t2 : 0
t2 : 1
t2 : 2
t2 : 3

可见，调用 yield()，不同优先级的线程永远不会得到执行机会。

2.5 join()

使调用 join()的线程执行完毕后才能执行其它线程，在一定意义上，它可以实现同步的功能。

例 16：

```
class TestThreadMethod extends Thread{
public static int shareVar = 0;
public TestThreadMethod(String name){
super(name);
}
public void run(){
for(int i=0; i<4; i++){
System.out.println(" " + i);
try{
Thread.sleep(3000);
}
catch(InterruptedException e){
System.out.println("Interrupted");
}
}
}
}
public class TestThread{
public static void main(String[] args){
TestThreadMethod t1 = new TestThreadMethod("t1");
t1.start();
try{
t1.join();
}
catch(InterruptedException e){}
```

```
t1.start();
}
}
```

运行结果为：

```
0
1
2
3
0
1
2
3
```

3. class　Object 下常用的线程函数

wait()、notify()和 notifyAll()这三个函数由 java.lang.Object 类提供，用于协调多个线程对共享数据的存取。

3.1 wait()、notify()和 notifyAll()

1) wait()函数有两种形式：第一种形式接受一个毫秒值，用于在指定时间长度内暂停线程，使线程进入停滞状态。第二种形式为不带参数，代表 waite()在 notify()或 notifyAll()之前会持续停滞。

2) 当对一个对象执行 notify()时，会从线程等待池中移走该任意一个线程，并把它放到锁标志等待池中；当对一个对象执行 notifyAll()时，会从线程等待池中移走所有该对象的所有线程，并把它们放到锁标志等待池中。

3) 当调用 wait()后，线程会释放掉它所占有的“锁标志”，从而使线程所在对象中的其它 synchronized 数据可被别的线程使用。

例 17：

下面，我们将对例 11 中的例子进行修改

```
class TestThreadMethod extends Thread{
public static int shareVar = 0;
public TestThreadMethod(String name){
super(name);
}
public synchronized void run(){
if(shareVar==0){
for(int i=0; i<10; i++){
shareVar++;
if(shareVar==5){
try{
this.wait();  // (4)
}
catch(InterruptedException e){}
```

```
}
}
}
if(shareVar!=0){
System.out.print(Thread.currentThread().getName());
System.out.println(" shareVar = " + shareVar);
this.notify();  //（5）
}
}
}
public class TestThread{
public static void main(String[] args){
TestThreadMethod t1 = new TestThreadMethod("t1");
TestThreadMethod t2 = new TestThreadMethod("t2");
t1.start();  //（1）
//t1.start();   （2）
t2.start();  //（3）
}
}
```

运行结果为：

t2 shareVar = 5

因为 t1 和 t2 是两个不同对象，所以线程 t2 调用代码（5）不能唤起线程 t1。如果去掉代码（2）的注释，并注释掉代码（3），结果为：

t1 shareVar = 5

t1 shareVar = 10

这 是因为，当代码（1）的线程执行到代码（4）时，它进入停滞状态，并释放对象的锁状态。接着，代码（2）的线程执行 run()，由于此时 shareVar 值为 5，所以执行打印语句并调用代码（5）使代码（1）的线程进入可执行状态，然后代码（2）的线程结束。当代码（1）的线程重新执行后，它接着执行 for()循环一直到 shareVar=10，然后打印 shareVar。

3.2 wait()、notify()和 synchronized

waite() 和 notify()因为会对对象的“锁标志”进行操作，所以它们必须在 synchronized 函数或 synchronized block 中进行调用。 如果在 non-synchronized 函数或 non-synchronized block 中进行调用，虽然能编译通过，但在运行时会发生 IllegalMonitorStateException 的异常。

例 18：

```
class TestThreadMethod extends Thread{
public int shareVar = 0;
public TestThreadMethod(String name){
super(name);
new Notifier(this);
}
public synchronized void run(){
if(shareVar==0){
for(int i=0; i<5; i++){
```

```
shareVar++;
System.out.println("i = " + shareVar);
try{
System.out.println("wait......");
this.wait();
}
catch(InterruptedException e){}
}
}
}
}
class Notifier extends Thread{
private TestThreadMethod ttm;
Notifier(TestThreadMethod t){
ttm = t;
start();
}
public void run(){
while(true){
try{
sleep(2000);
}
catch(InterruptedException e){}
/*1 要同步的不是当前对象的做法 */
synchronized(ttm){
System.out.println("notify......");
ttm.notify();
}
}
}
}
public class TestThread{
public static void main(String[] args){
TestThreadMethod t1 = new TestThreadMethod("t1");
t1.start();
}
}
```

运行结果为：

```
i = 1
wait......
notify......
i = 2
wait......
notify......
```

```
i = 3
wait......
notify......
i = 4
wait......
notify......
i = 5
wait......
notify......
```

4. wait()、notify()、notifyAll()和 suspend()、resume()、sleep()的讨论

4.1 这两组函数的区别

1) wait()使当前线程进入停滞状态时，还会释放当前线程所占有的“锁标志”，从而使线程对象中的 synchronized 资源可被对象中别的线程使用；而 suspend()和 sleep()使当前线程进入停滞状态时不会释放当前线程所占有的“锁标志”。

2) 前一组函数必须在 synchronized 函数或 synchronized block 中调用，否则在运行时会产生错误；而后一组函数可以 non-synchronized 函数和 synchronized block 中调用。

4.2 这两组函数的取舍

Java2 已不建议使用后一组函数。因为在调用 wait()时不会释放当前线程所取得的“锁标志”，这样很容易造成“死锁”。

Java 多线程学习笔记

一、线程类

Java 是通过 Java.lang.Thread 类来实现多线程的，第个 Thread 对象描述了一个单独的线程。要产生一个线程，有两种方法：

1 、需要从 Java.lang.Thread 类继承一个新的线程类，重载它的 run()方法；

2 、通过 Runnalbe 接口实现一个从非线程类继承来类的多线程，重载 Runnalbe 接口的 run()方法。运行一个新的线程，只需要调用它的 start()方法即可。如：

```
/**================================================================
===

* 文件：ThreadDemo_01.java

* 描述：产生一个新的线程

*
================================================================
=

*/

class ThreadDemo extends Thread{

  Threads()

  {

  }


  Threads(String szName)

  {

    super(szName);

  }


  // 重载 run 函数

  public void run()

  {

     for (int count = 1,row = 1; row < 20; row++,count++)

     {

        for (int i = 0; i < count; i++)

        {
```

```
            System.out.print('*');
        }
        System.out.println();
    }
  }
}
class ThreadMain{
  public static void main(String argv[]){
    ThreadDemo th = new ThreadDemo();
    // 调用 start()方法执行一个新的线程
    th.start();
  }
}
```

线程类的一些常用方法：

sleep(): 强迫一个线程睡眠N毫秒。

isAlive(): 判断一个线程是否存活。

join(): 等待线程终止。

activeCount(): 程序中活跃的线程数。

enumerate(): 枚举程序中的线程。

currentThread(): 得到当前线程。

isDaemon(): 一个线程是否为守护线程。

setDaemon(): 设置一个线程为守护线程。(用户线程和守护线程的区别在于，是否等待主线程依赖于主线程结束而结束)

setName(): 为线程设置一个名称。

wait(): 强迫一个线程等待。

notify(): 通知一个线程继续运行。

setPriority(): 设置一个线程的优先级。

二、等待一个线程的结束

有些时候我们需要等待一个线程终止后再运行我们的另一个线程，这时我们应该怎么办呢？请看下面的例子：

```
/**================================================================
===

* 文件：ThreadDemo_02.java

* 描述：等待一个线程的结束

*
==========================================================================
=

*/

class ThreadDemo extends Thread{

  Threads()

  {

  }



  Threads(String szName)

  {

    super(szName);

  }



  // 重载 run 函数
```

```
  public void run()
  {
     for (int count = 1,row = 1; row < 20; row++,count++)
     {
        for (int i = 0; i < count; i++)
        {
           System.out.print('*');
        }
        System.out.println();
     }
  }
}
class ThreadMain{
  public static void main(String argv[]){
    //产生两个同样的线程
    ThreadDemo th1 = new ThreadDemo();
    ThreadDemo th2 = new ThreadDemo();
   // 我们的目的是先运行第一个线程，再运行第二个线程
   th1.start();
   th2.start();
  }
}
```

这里我们的目标是要先运行第一个线程，等第一个线程终止后再运行第二个线程，而实际运行的结果是如何的呢？实际上我们运行的结果并不是两个我们想要

的直角三角形，而是一些乱七八糟的＊号行，有的长，有的短。为什么会这样呢？因为线程并没有按照我们的调用顺序来执行，而是产生了线程赛跑现象。实际上 Java 并不能按我们的调用顺序来执行线程，这也说明了线程是并行执行的单独代码。如果要想得到我们预期的结果，这里我们就需要判断第一个线程是否已经终 止，如果已经终止，再来调用第二个线程。代码如下：

```
/**=================================================================
===

* 文件：ThreadDemo_03.java

* 描述：等待一个线程的结束的两种方法

*
=================================================================
=

*/

class ThreadDemo extends Thread{

  Threads()

  {

  }



  Threads(String szName)

  {

    super(szName);

  }



  // 重载 run 函数

  public void run()

  {

     for (int count = 1,row = 1; row < 20; row++,count++)
```

```
      {
         for (int i = 0; i < count; i++)
         {
            System.out.print('*');
         }
         System.out.println();
      }
   }
}
class ThreadMain{
   public static void main(String argv[]){
      ThreadMain test = new ThreadMain();
      test.Method1();
      // test.Method2();
   }
   // 第一种方法：不断查询第一个线程是否已经终止，如果没有，则让主线程睡眠一直到它终止为止
   // 即：while/isAlive/sleep
   public void Method1(){
      ThreadDemo th1 = new ThreadDemo();
      ThreadDemo th2 = new ThreadDemo();
      // 执行第一个线程
      th1.start();
      // 不断查询第一个线程的状态
```

```
    while(th1.isAlive()){
      try{
         Thread.sleep(100);
      }catch(InterruptedException e){
      }
    }
    //第一个线程终止，运行第二个线程
    th2.start();
  }

  // 第二种方法：join()
  public void Method2(){
    ThreadDemo th1 = new ThreadDemo();
    ThreadDemo th2 = new ThreadDemo();
    // 执行第一个线程
    th1.start();
    try{
      th1.join();
    }catch(InterruptedException e){
    }
    // 执行第二个线程
    th2.start();
}
```

三、线程的同步问题

有些时候，我们需要很多个线程共享一段代码，比如一个私有成员或一个类中的静态成员，但是由于线程赛跑的问题，所以我们得到的常常不是正确的输出结果，而相反常常是张冠李戴，与我们预期的结果大不一样。看下面的例子：

```
/**================================================================
===========

 * 文件：ThreadDemo_04.java

 * 描述：多线程不同步的原因

 *
==========================================================================
========

 */

// 共享一个静态数据对象

class ShareData{

 public static String szData = "";

}

class ThreadDemo extends Thread{


  private ShareData oShare;

  ThreadDemo(){

  }

  ThreadDemo(String szName,ShareData oShare){

    super(szName);

    this.oShare = oShare;

  }
```

```
  public void run(){
    // 为了更清楚地看到不正确的结果，这里放一个大的循环
    for (int i = 0; i < 50; i++){
       if (this.getName().equals("Thread1")){
         oShare.szData = "这是第 1 个线程";
         // 为了演示产生的问题，这里设置一次睡眠
         try{
           Thread.sleep((int)Math.random() * 100);
         catch(InterruptedException e){
         }
         // 输出结果
         System.out.println(this.getName() + ":" + oShare.szData);
       }else if (this.getName().equals("Thread2")){
         oShare.szData = "这是第 1 个线程";
         // 为了演示产生的问题，这里设置一次睡眠
         try{
           Thread.sleep((int)Math.random() * 100);
         catch(InterruptedException e){
         }
         // 输出结果
         System.out.println(this.getName() + ":" + oShare.szData);
       }
    }
}
```

```
class ThreadMain{

  public static void main(String argv[]){

    ShareData oShare = new ShareData();

    ThreadDemo th1 = new ThreadDemo("Thread1",oShare);

    ThreadDemo th2 = new ThreadDemo("Thread2",oShare);

    th1.start();

    th2.start();

  }

}
```

由于线程的赛跑问题，所以输出的结果往往是 Thread1 对应“这是第 2 个线程”，这样与我们要输出的结果是不同的。为了解决这种问题（错误），Java 为我们提供了“锁”的机制来实现线程的同步。锁的机制要求每个线程在进入共享代码之前都要取得锁，否则不能进入，而退出共享代码之前则释放该锁，这样就防止了几个或多个线程竞争共享代码的情况，从而解决了线程的不同步的问题。 可以这样说，在运行共享代码时则是最多只有一个线程进入，也就是和我们说的垄断。锁机制的实现方法，则是在共享代码之前加入 synchronized 段，把共享代码包含在 synchronized 段中。上述问题的解决方法为：

```
/**================================================================
===========

 * 文件：ThreadDemo_05.java

 * 描述：多线程不同步的解决方法--锁

 *
================================================================
========

 */

// 共享一个静态数据对象

class ShareData{

 public static String szData = "";
```

```
}
class ThreadDemo extends Thread{

  private ShareData oShare;
  ThreadDemo(){
  }
  ThreadDemo(String szName,ShareData oShare){
    super(szName);
    this.oShare = oShare;
  }
  public void run(){
    // 为了更清楚地看到不正确的结果，这里放一个大的循环
    for (int i = 0; i < 50; i++){
       if (this.getName().equals("Thread1")){
         // 锁定 oShare 共享对象
         synchronized (oShare){
           oShare.szData = "这是第 1 个线程";
           // 为了演示产生的问题，这里设置一次睡眠
           try{
             Thread.sleep((int)Math.random() * 100);
           catch(InterruptedException e){
           }
           // 输出结果
           System.out.println(this.getName() + ":" + oShare.szData);
```

```
        }
      }else if (this.getName().equals("Thread2")){
        // 锁定共享对象
        synchronized (oShare){
          oShare.szData = "这是第 1 个线程";
          // 为了演示产生的问题，这里设置一次睡眠
          try{
            Thread.sleep((int)Math.random() * 100);
          catch(InterruptedException e){
          }
          // 输出结果
          System.out.println(this.getName() + ":" + oShare.szData);
        }
      }
   }
}
class ThreadMain{
  public static void main(String argv[]){
    ShareData oShare = new ShareData();
    ThreadDemo th1 = new ThreadDemo("Thread1",oShare);
    ThreadDemo th2 = new ThreadDemo("Thread2",oShare);
    th1.start();
    th2.start();
  }
```

}

由于过多的 synchronized 段将会影响程序的运行效率，因此引入了同步方法，同步方法的实现则是将共享代码单独写在一个方法里，在方法前加上 synchronized 关键字即可。

在线程同步时的两个需要注意的问题：

1、无同步问题：即由于两个或多个线程在进入共享代码前，得到了不同的锁而都进入共享代码而造成。

2、死锁问题：即由于两个或多个线程都无法得到相应的锁而造成的两个线程都等待的现象。这种现象主要是因为相互嵌套的 synchronized 代码段而造成，因此，在程序中尽可能少用嵌套的 synchronized 代码段是防止线程死锁的好方法。

在写上面的代码遇到的一个没能解决的问题，在这里拿出来，希望大家讨论是什么原因。

```
/**================================================================
===========

 * 文件：ThreadDemo_06.java

 * 描述：为什么造成线程的不同步。

 *
================================================================
========

 */

class ThreadDemo extends Thread{

  //共享一个静态数据成员

  private static String  szShareData = "";

  ThreadDemo(){

  }

  ThreadDemo(String szName){
```

```
    super(szName);
  }
  public void run(){
    // 为了更清楚地看到不正确的结果，这里放一个大的循环
    for (int i = 0; i < 50; i++){
      if (this.getName().equals("Thread1")){
        synchronized(szShareData){
          szShareData = "这是第 1 个线程";
          // 为了演示产生的问题，这里设置一次睡眠
          try{
            Thread.sleep((int)Math.random() * 100);
          catch(InterruptedException e){
          }
          // 输出结果
          System.out.println(this.getName() + ":" + szShareData);
        }
      }else if (this.getName().equals("Thread2")){
        synchronized(szShareData){
          szShareData = "这是第 1 个线程";
          // 为了演示产生的问题，这里设置一次睡眠
          try{
            Thread.sleep((int)Math.random() * 100);
          catch(InterruptedException e){
          }
```

```
            // 输出结果
            System.out.println(this.getName() + ":" + szShareData);
          }
        }
    }
}
class ThreadMain{
  public static void main(String argv[]){
    ThreadDemo th1 = new ThreadDemo("Thread1");
    ThreadDemo th2 = new ThreadDemo("Thread2");
    th1.start();
    th2.start();
  }
}
```

这段代码的共享成员是一个类中的静态成员，按理说，这里进入共享代码时得到的锁应该是同样的锁，而实际上以上程序的输入却是不同步的，为什么呢？？

四、Java 的等待通知机制

在有些时候，我们需要在几个或多个线程中按照一定的秩序来共享一定的资源。例如生产者－－消费者的关系，在这一对关系中实际情况总是先有生产者生产了产 品后，消费者才有可能消费；又如在父－－子关系中，总是先有父亲，然后才能有儿子。然而在没有引入等待通知机制前，我们得到的情况却常常是错误的。这里我 引入《用线程获得强大的功能》一文中的生产者－－消费者的例子：

```
/*
================================================================================
```

```
 * 文件：ThreadDemo07.java
 * 描述：生产者－－消费者
 * 注：其中的一些注释是我根据自己的理解加注的
 *
==========================================================================
=============
 */
// 共享的数据对象
 class ShareData{
  private char c;

  public void setShareChar(char c){
   this.c = c;
  }

  public char getShareChar(){
   return this.c;
  }
 }

 // 生产者线程
 class Producer extends Thread{

  private ShareData s;
```

```
 Producer(ShareData s){
  this.s = s;
 }

 public void run(){
  for (char ch = 'A'; ch <= 'Z'; ch++){
   try{
    Thread.sleep((int)Math.random() * 4000);
   }catch(InterruptedException e){}

   // 生产
   s.setShareChar(ch);
   System.out.println(ch + " producer by producer.");
  }
 }
}

// 消费者线程
class Consumer extends Thread{

 private ShareData s;

 Consumer(ShareData s){
```

```
    this.s = s;
  }

  public void run(){
   char ch;

   do{
    try{
     Thread.sleep((int)Math.random() * 4000);
    }catch(InterruptedException e){}
    // 消费
    ch = s.getShareChar();
    System.out.println(ch + " consumer by consumer.");
   }while(ch != 'Z');
  }
 }
class Test{
 public static void main(String argv[]){
  ShareData  s = new ShareData();
  new Consumer(s).start();
  new Producer(s).start();
 }
}
```

在以上的程序中，模拟了生产者和消费者的关系，生产者在一个循环中不断生产了从Ａ－Ｚ的共享数据，而消费者则不断地消费生产者生产的Ａ－Ｚ的共享数据。我们开始已经说过，在这一对关系中，必须先有生产者生产，才能有消费者消费。但如果运行我们上面这个程序，结果却出现了在生产者没有生产之前，消 费都就已经开始消费了或者是生产者生产了却未能被消费者消费这种反常现象。为了解决这一问题，引入了等待通知（wait/notify)机制如下：

１、在生产者没有生产之前，通知消费者等待；在生产者生产之后，马上通知消费者消费。

２、在消费者消费了之后，通知生产者已经消费完，需要生产。

下面修改以上的例子（源自《用线程获得强大的功能》一文）：

```
/*
================================================================================
=============

 * 文件：ThreadDemo08.java

 * 描述：生产者－－消费者

 * 注：其中的一些注释是我根据自己的理解加注的

 *
================================================================================
=============

 */

class ShareData{



 private char c;

 // 通知变量

 private boolean writeable = true;

 //
--------------------------------------------------------------------------------
----

 // 需要注意的是：在调用 wait()方法时，需要把它放到一个同步段里，否则将会出现
```

```
 // "java.lang.IllegalMonitorStateException: current thread not owner"
的异常。

 //
-------------------------------------------------------------------------
----

 public synchronized void setShareChar(char c){

  if (!writeable){

   try{

    // 未消费等待

    wait();

   }catch(InterruptedException e){}

  }



  this.c = c;

  // 标记已经生产

  writeable = false;

  // 通知消费者已经生产，可以消费

  notify();

 }



 public synchronized char getShareChar(){

  if (writeable){

   try{

    // 未生产等待

    wait();
```

```
    }catch(InterruptedException e){}
   }
   // 标记已经消费
   writeable = true;
   // 通知需要生产
   notify();
   return this.c;
  }
}
// 生产者线程
class Producer extends Thread{

 private ShareData s;

 Producer(ShareData s){
  this.s = s;
 }

 public void run(){
  for (char ch = 'A'; ch <= 'Z'; ch++){
   try{
    Thread.sleep((int)Math.random() * 400);
   }catch(InterruptedException e){}
```

```
    s.setShareChar(ch);
    System.out.println(ch + " producer by producer.");
   }
  }
}
// 消费者线程
class Consumer extends Thread{

 private ShareData s;

 Consumer(ShareData s){
  this.s = s;
 }

 public void run(){
  char ch;

  do{
   try{
    Thread.sleep((int)Math.random() * 400);
   }catch(InterruptedException e){}

   ch = s.getShareChar();
   System.out.println(ch + " consumer by consumer.**");
```

```
  }while (ch != 'Z');
 }
}
class Test{
 public static void main(String argv[]){
  ShareData  s = new ShareData();
  new Consumer(s).start();
  new Producer(s).start();
 }
}
```

在以上程序中，设置了一个通知变量，每次在生产者生产和消费者消费之前，都测试通知变量，检查是否可以生产或消费。最开始设置通知变量为 true，表示还未生产，在这时候，消费者需要消费，于时修改了通知变量，调用 notify()发出通知。这时由于生产者得到通知，生产出第一个产品，修 改通知变量，向消费者发出通知。这时如果生产者想要继续生产，但因为检测到通知变量为 false，得知消费者还没有生产，所以调用 wait()进入等待状 态。因此，最后的结果，是生产者每生产一个，就通知消费者消费一个；消费者每消费一个，就通知生产者生产一个，所以不会出现未生产就消费或生产过剩的情 况。

五、线程的中断

在很多时候，我们需要在一个线程中调控另一个线程，这时我们就要用到线程的中断。用最简单的话也许可以说它就相当于播放 机中的暂停一样，当第一次按下暂停时，播放器停止播放，再一次按下暂停时，继续从刚才暂停的地方开始重新播放。而在 Java 中，这个暂停按钮就是 Interrupt()方法。在第一次调用 interrupt()方法时，线程中断；当再一次调用 interrupt()方法时，线程继续运行直到终止。 这里依然引用《用线程获得强大功能》一文中的程序片断，但为了更方便看到中断的过程，我在原程序的基础上作了些改进，程序如下：

```
/*
================================================================================
```

```
 * 文件：ThreadDemo09.java
```

```
 * 描述：线程的中断
 *
==========================================================================
==============
 */
class ThreadA extends Thread{

 private Thread thdOther;

 ThreadA(Thread thdOther){
  this.thdOther = thdOther;
 }

 public void run(){

  System.out.println(getName() + " 运行...");

  int sleepTime = (int)(Math.random() * 10000);
  System.out.println(getName() + " 睡眠 " + sleepTime
   + " 毫秒。");

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

```
  System.out.println(getName() + " 觉醒，即将中断线程 B。");
  // 中断线程B,线程B暂停运行
  thdOther.interrupt();
 }
}
class ThreadB extends Thread{
 int count = 0;

 public void run(){

  System.out.println(getName() + " 运行...");

  while (!this.isInterrupted()){
   System.out.println(getName() + " 运行中 " + count++);

   try{
    Thread.sleep(10);
   }catch(InterruptedException e){
    int sleepTime = (int)(Math.random() * 10000);
    System.out.println(getName() + " 睡眠" + sleepTime
     + " 毫秒。觉醒后立即运行直到终止。");

    try{
```

```
      Thread.sleep(sleepTime);
     }catch(InterruptedException m){}

     System.out.println(getName() + " 已经觉醒，运行终止...");
     // 重新设置标记，继续运行
     this.interrupt();
    }
   }

   System.out.println(getName() + " 终止。");
  }
}
class Test{
 public static void main(String argv[]){
   ThreadB thdb = new ThreadB();
   thdb.setName("ThreadB");

   ThreadA thda = new ThreadA(thdb);
   thda.setName("ThreadA");

   thdb.start();
   thda.start();
  }
}
```

运行以上程序，你可以清楚地看到中断的过程。首先线程 B 开始运行，接着运行线程 A，在线程 A 睡眠一段时间觉醒后，调用 interrupt()方 法中断线程 B，此是可能 B 正在睡眠，觉醒后掏出一个 InterruptedException 异常，执行其中的语句，为了更清楚地看到线程的中断恢复，我 在 InterruptedException 异常后增加了一次睡眠，当睡眠结束后，线程 B 调用自身的 interrupt()方法恢复中断，这时测试 isInterrupt()返回 true,线程退出。

更多的线程组及相关的更详细的信息，请参考《用线程获得强大功能》一文及《Thinking in Java》。

创建 Java 中的线程池

线程是 Java 的一大特性，它可以是给定的指令序列、给定的方法中定义的变量或者一些共享数据(类一级的变量)。在 Java 中每个线程有自己的堆栈和程序 计数器（PC），其中堆栈是用来跟踪线程的上下文（上下文是当线程执行到某处时，当前的局部变量的值），而程序计数器则用来跟踪当前线程正在执行的指令。

在通常情况下，一个线程不能访问另外一个线程的堆栈变量，而且这个线程必须处于如下状态之一：

1.排队状态 （Ready），在用户创建了一个线程以后，这个线程不会立即运行。当线程中的方法 start()被调用时，这个线程就会进行排队状态，等待调度程序将它 转入运行状态（Running）。当一个进程被执行后它也可以进行排队状态。如果调度程序允许的话，通过调用方法 yield()就可以将进程放入排队状 态。

2.运行状态(Running)，当调度程序将 CPU 的运行时间分配给一个线程，这个线程就进入了运行状态开始运行。

3.等待状态（Waiting），很多原因都可以导致线程处于等待状态，例如线程执行过程中被暂停，或者是等待 I/O 请求的完成而进入等待状态。

在 Java 中不同的线程具有不同的优先级，高优先级的线程可以安排在低优先级线程之前完成。如果多个线程具有相同的优先级，Java 会在不同的线程之间切换 运行。一个应用程序可以通过使用线程中的方法 setPriority()来设置线程的优先级，使用方法 getPriority()来获得一个线程的优先 级。

一、线程的生命周期

一个线程的的生命周期可以分成两阶段：生存（Alive）周期和 死亡（Dead）周期，其中生存周期又包括运行状态（Running）和等待状态（Waiting）。当创建一个新线程后，这个线程就进入了排队状态 （Ready），当线程中的方法 start()被调用时，线程就进入生存周期，这时它的方法 isAlive()始终返回真值，直至线程进入死亡状态。

二、线程的实现

有两种方法可以实现线程，一种是扩展 java.lang.Thread 类，另一种是通过 java.lang.Runnable 接口。

Thread 类封装了线程的行为。要创建一个线程，必须创建一个从 Thread 类扩展出的新类。由于在 Thread 类中方法 run()没有提供任何的操作，因此，在创 建线程时用户必须覆盖方法 run()来完成有用的工作。当线程中的方法 start()被调用时，方法 run()再被调用。下面的代码就是通过扩展 Thread 类来实现线程：

```
import java.awt.*;
class Sample1{
public static void main(String[] args){
  Mythread test1=new Mythread(1);
  Mythread test2=new Mythread(2);
  test1.start();
  test2.start();
}
}
class Mythread  extends Thread {
int id;
Mythread(int i)
{ id=i;}
public void run() {
  int i=0;
  while(id+i==1){
    try {sleep(1000);
  } catch(InterruptedException e) {}
}
System.out.println(“The id  is ”+id);
}
```

通 常当用户希望一个类能运行在自己的线程中，同时也扩展其它某些类的特性时，就需要

借助运行 Runnable 接口来实现。Runnable 接口只有一个方法 run()。不论什么时候创建了一个使用 Runnable 接口的类，都必须在类中编写 run()方法来覆盖接口中的 run()方法。例如下面的代码就是通 过 Runnable 接口实现的线程：

```
import java.awt.*;
import java.applet.Applet;
public class Bounce extends Applet implements Runnable{
static int r=30;
static int x=100;
static int y=30;
Thread t;
public void init()
{
 t = new Thread(this);
 t.start();
}
public void run()
{
 int y1=+1;
 int i=1;
 int sleeptime=10;
 while(true)
 {
  y+=(i*y);
  if(y-r<i ||y+r>getSize().height)
  y1*=-1;
  try{
  t.sleep(sleeptime);
  }catch(InterruptedException e){ }
 }
}
}
```

三、为什么要使用线程池

在 Java 中，如果每当一个请求到达就创建一个新线程，开销是相当大的。在实际使用中，每个请求创建新线程的服务器在创建和销毁线程上花费的时间和消耗的系 统资源，甚至可能要比花在处理实际的用户请求的时间和资源要多得多。除了创建和销毁线程的开销之外，

活动的线程也需要消耗系统资源。如果在一个 JVM 里创 建太多的线程，可能会导致系统由于过度消耗内存或“切换过度”而导致系统资源不足。为了防止资源不足，服务器应用程序需要一些办法来限制任何给定时刻处理 的请求数目，尽可能减少创建和销毁线程的次数，特别是一些资源耗费比较大的线程的创建和销毁，尽量利用已有对象来进行服务，这就是“池化资源”技术产生的 原因。

线程池主要用来解决线程生命周期开销问题和资源不足问题。通过对多个任务重用线程，线程创建的开销就被分摊到了多个任务上了， 而且由于在请求到达时线程已经存在，所以消除了线程创建所带来的延迟。这样，就可以立即为请求服务，使应用程序响应更快。另外，通过适当地调整线程池中的 线程数目可以防止出现资源不足的情况。

四、创建一个线程池

一个比较简单的线程池至少应包 含线程池管理器、工作线程、任务队列、任务接口等部分。其中线程池管理器（ThreadPool Manager）的作用是创建、销毁并管理线程池，将工作线程放入线程池中；工作线程是一个可以循环执行任务的线程，在没有任务时进行等待；任务队列的作 用是提供一种缓冲机制，将没有处理的任务放在任务队列中；任务接口是每个任务必须实现的接口，主要用来规定任务的入口、任务执行完后的收尾工作、任务的执 行状态等，工作线程通过该接口调度任务的执行。下面的代码实现了创建一个线程池，以及从线程池中取出线程的操作：

```
public class ThreadPool
{
private Stack threadpool = new Stack();
private int poolSize;
private int currSize=0;
public void setSize(int n)
{
  poolSize = n;
}
public void run()
{
  for(int i=0;i<poolSize;i++)
  {
    WorkThread workthread=new WorkThread();
    threadpool.push(workthread);
    currSize++;
  }
}
public  synchronized WorkThread  getworker( )
```

```
{
 if (threadpool.empty())
   system.out.println(“stack is empty”);
 else
 try{ return threadpool.pop();
 } catch (EmptyStackException e){}
}
}
```

五、线程池适合应用的场合

当 一个 Web 服务器接受到大量短小线程的请求时，使用线程池技术是非常合适的，它可以大大减少线程的创建和销毁次数，提高服务器的工作效率。但如果线程要求 的运行时间比较长，此时线程的运行时间比创建时间要长得多，单靠减少创建时间对系统效率的提高不明显，此时就不适合应用线程池技术，需要借助其它的技术来 提高服务器的服务效率。

用多线程又有几种常用的编程模型

我这里可以大概给你介绍一下，但对于每一种编程模型要看具体的示例是什么，而且我不可能给你罗列所有的代码，请谅解。
其实我们编程只要尽量站到比较高的层次，很多道理其实你会发现你已经懂了。

就多线程来说，我们开始设想只有两个线程（>2 时是不是算数学归纳法？）那么如果两个独立的线程会发生什么呢？

1。当一个线程进入 moniter（也就是说站用一个 object），另一个线程只有等待或返回，而我们把返回就称为一种模式，这种模式的英文是 Balking。

2。这两个线程可以是有序的执行，而不是让 OS 来调度，这时我们要用一个 object 来调度，这种模式称为 Scheduler。（这个词及其含义其实 OS 中就有）。

3。如果这两个线程同时读一个资源，我们可以让他们执行，但如果同时写的话，你闭着眼睛都会知道可能出现问题，这时我们就要用另一种模式（Read/Write Lock）。

4。如果一个线程是为另一个线程服务的话，比如 IE 中负责数据传输的线程和界面显示的线程，当一个图片没有传完时，另一个线程就无法显示，至少是部分没有传完。那么这时我们

要用一个模式称为生产者和消费者，英文是 Producer-Consumer。

5。两个线程的消亡也可以不是完全又 OS 来控制的，这时我们需要给出一个条件，使得每个线程在符合条件是才消亡，也就是有序的消亡，我们称为 Two-Phase Termination。

那么有这 5 个线程模型，基本上可以用到大多数编程任务中。我需要指出的三点是：

1。从高层次上我们可以再验证是否含盖了所有的情况。

2。其实模式不是完全固定的或者说象定律一样，而模式可以为不同的情况进行适当 的调整和组合，目的是为了简洁和高效。

3。学习模式是为了具备更好的分析问题的能力。

而 似乎这些来自西方的技术，并且是目前的，我们有没有呢？其实我个人有个大胆的推测，我认为我们祖先的《孙子兵法》就是很好设计模式，因为它符合设计模式需 要的基本特征，就是在特定的条件下，用某种特定的方式合理且高效的解决问题。只不过一是用在军事上，二是完备性方面我们还没研究。但我认为我们至少没有很 好的扩展和进行类比式的应用，否则今天可能是我们中国人教外国人什么是设计模式。

类比的方法实际上是发明或发现的常用方法。不知能否让你感觉到其实外国的技术并不是那么的神秘，也许我们从自身的文化当中挖掘出的东西太少了。

用 Java 实现多线程服务器程序

摘要：在 Java 出现之前，编写多线程程序是一件烦琐且伴随许多不安全因素的事情。利用 Java，编写安全高效的多线程程序变得简单，而且利用多线程和 Java 的网络包我们可以方便的实现多线程服务器程序。

---- Java 是伴随 Internet 的大潮产生的，对网络及多线程具有内在的支持，具有网络时代编程语言的一切特点。从 Java 的当前应用看，Java 主要用 于在 Internet 或局域网上的网络编程，而且将 Java 作为主流的网络编程语言的趋势愈来愈明显。实际工作中，我们除了使用商品化的服务器软件外，时 常需要按照实际环境编写自己的服务器软件，以完成特定任务或与特定客户端软件实现交互。在实现服务器程序时，为提高程序运行效率，降低用户等待时间，我们 应用了在 Java Applet 中常见的多线程技术。

一、Java 中的服务器程序与多线程

---- 在 Java 之前，没有一种主流编程语言能够提供对高级网络编程的固有支持。在其他语

言环境中，实现网络程序往往需要深入依赖于操作平台的网络 API 的技术 中去，而 Java 提供了对网络支持的无平台相关性的完整软件包，使程序员没有必要为系统网络支持的细节而烦恼。

---- Java 软件包内在支持的网络协议为 TCP/IP，也是当今最流行的广域网/局域网协议。Java 有关网络的类及接口定义在 java.net 包中。客户端 软件通常使用 java.net 包中的核心类 Socket 与服务器的某个端口建立连接，而服务器程序不同于客户机，它需要初始化一个端口进行监听，遇到连接 呼叫，才与相应的客户机建立连接。Java.net 包的 ServerSocket 类包含了编写服务器系统所需的一切。下面给出 ServerSocket 类 的部分定义。

```
public class ServerSocket {
public ServerSocket(int port)
throws IOException ；
public Socket accept() throws IOException ；
public InetAddress getInetAddress() ；
public int getLocalPort() ；
public void close() throws IOException ；
public synchronized void setSoTimeout
(int timeout) throws SocketException ；
public synchronized int
getSoTimeout() throws IOException ；
}
```

---- ServerSocket 构造器是服务器程序运行的基础，它将参数 port 指定的端口初始化作为该服务器的端口，监听客户机连接请求。Port 的范围是 0 到 65536，但 0 到 1023 是标准 Internet 协议保留端口，而且在 Unix 主机上，这些端口只有 root 用户可以使用。一般自定义的端口号在 8000 到 16000 之间。仅初始化了 ServerSocket 还是远远不够的，它没有同客户机交互的套接字（Socket），因此需要调用该类的 accept 方法接受客户呼叫。Accept()方法直到有连接请求才返回通信套接字(Socket)的实例。通过这个实例的输入、输出流，服务器可以接 收用户指令，并将相应结果回应客户机。ServerSocket 类的 getInetAddress 和 getLocalPort 方法可得到该服务器的 IP 地 址和端口。setSoTimeout 和 getSoTimeout 方法分别是设置和得到服务器超时设置，如果服务器在 timout 设定时间内还未得到 accept 方法返回的套接字实例，则抛出 IOException 的异常。
---- Java 的多线程可谓是 Java 编程的精华之一，运用得当可以极大地改善程序的响应时间，提高程序的并行性。在服务器程序中，由于往往要接收不同客户机的 同时请求或命令，因此可以对每个客户机的请求生成一个命令处理线程，同时对各用户的指令作出反应。在一些较复杂的系统中，我们还可以为每个数据库查询指令 生成单独的线程，并行对数据库进行操作。实践证明，采用多线程设计可以很好的改善系统的响应，并保证用户指令执行的独立性。由于 Java 本身是"线程安 全"的，因此有一条编程原则是能够独立在一个线程中完成的操作就应该开辟一个新的线程。

---- Java 中实现线程的方式有两种，一是生成 Thread 类的子类，并定义该子类自己的 run 方法，线程的操作在方法 run 中实现。但我们定义的类一般是其 他类的子类，而 Java 又不允

许多重继承，因此第二种实现线程的方法是实现 Runnable 接口。通过覆盖 Runnable 接口中的 run 方法实现该线程 的功能。本文例子采用第一种方法实现线程。

二、多线程服务器程序举例

---- 以下是我们在项目中采用的多线程服务器程序的架构，可以在此基础上对命令进行扩充。本例未涉及数据库。如果在线程运行中需要根据用户指令对数据库进行更新 操作，则应注意线程间的同步问题，使同一更新方法一次只能由一个线程调用。这里我们有两个类，receiveServer 包含启动代码（main()）， 并初始化 ServerSocket 的实例，在 accept 方法返回用户请求后，将返回的套接字（Socket）交给生成的线程类 serverThread 的实例，直到该用户结束连接。

```
//类 receiveServer
import java.io.*;
import java.util.*;
import java.net.*;

public class receiveServer{

final int RECEIVE_PORT=9090;
//该服务器的端口号

//receiveServer 的构造器
public receiveServer() {
ServerSocket rServer=null;
//ServerSocket 的实例
Socket request=null; //用户请求的套接字
Thread receiveThread=null;
try{
rServer=new ServerSocket(RECEIVE_PORT);
//初始化 ServerSocket
System.out.println("Welcome to the server!");
System.out.println(new Date());
System.out.println("The server is ready!");
System.out.println("Port: "+RECEIVE_PORT);
while(true){ //等待用户请求
request=rServer.accept();
//接收客户机连接请求
receiveThread=new serverThread(request);
//生成 serverThread 的实例
receiveThread.start();
//启动 serverThread 线程
```

```
}
}catch(IOException e){
System.out.println(e.getMessage());}
}

public static void main(String args[]){
new receiveServer();
} //end of main

} //end of class

//类 serverThread
import java.io.*;
import java.net.*;

class serverThread extends Thread {

Socket clientRequest;
//用户连接的通信套接字
BufferedReader input; //输入流
PrintWriter output; //输出流

public serverThread(Socket s)
{ //serverThread 的构造器
this.clientRequest=s;
//接收 receiveServer 传来的套接字
InputStreamReader reader;
OutputStreamWriter writer;
try{ //初始化输入、输出流
reader=new InputStreamReader
(clientRequest.getInputStream());
writer=new OutputStreamWriter
(clientRequest.getOutputStream());
input=new BufferedReader(reader);
output=new PrintWriter(writer,true);
}catch(IOException e){
System.out.println(e.getMessage());}
output.println("Welcome to the server!");
//客户机连接欢迎词
output.println("Now is:
"+new java.util.Date()+" "+
"Port:"+clientRequest.getLocalPort());
output.println("What can I do for you?");
}
```

```
public void run(){ //线程的执行方法
String command=null; //用户指令
String str=null;
boolean done=false;

while(!done){
try{
str=input.readLine(); //接收客户机指令
}catch(IOException e){
System.out.println(e.getMessage());}
command=str.trim().toUpperCase();
if(str==null || command.equals("QUIT"))
//命令 quit 结束本次连接
done=true;
else if(command.equals("HELP")){
//命令 help 查询本服务器可接受的命令
output.println("query");
output.println("quit");
output.println("help");
}
else if(command.startsWith("QUERY"))
{ //命令 query
output.println("OK to query something!");
}
//else if …….. //在此可加入服务器的其他指令
else if(!command.startsWith("HELP") &&
!command.startsWith("QUIT") &&
!command.startsWith("QUERY")){
output.println("Command not Found!
Please refer to the HELP!");
}
}//end of while
try{
clientRequest.close(); //关闭套接字
}catch(IOException e){
System.out.println(e.getMessage());
}
command=null;
}//end of run
```

---- 启动该服务器程序后，可用 telnet machine port 命令连接，其中 machine 为本机名或地址，port 为程序中指定的端口。也可以编写特定的客户机软件通过 TCP 的 Socket 套接字建立连接。

摘自《计算机世界日报》

关于线程的讲解(出自 Java 原著)

Thread Scheduling

In Java technology,threads are usually preemptive,but not necessarily Time-sliced(the process of giving each thread an equal amount of CPU time).It is common mistake to believe that "preemptive" is a fancy word for "does time-slicing".

For the runtime on a Solaris Operating Environment platform,Java technology does not preempt threads of the same priority.However,the runtime on Microsoft Windows platforms uses time-slicing,so it preempts threads of the same priority and even threads of higher priority.Preemption is not guaranteed;however,most JVM implementations result in behavior that appears to be strictly preemptive.Across JVM implementations,there is no absolute guarantee of preemption or time-slicing.The only guarantees lie in the coder’s use of wait and sleep.

The model of a preemptive scheduler is that many threads might be runnable,but only one thread is actually running.This thread continues to run until it ceases to be runnable or another thread of higher priority becomes runnable.In the latter case,the lower priority thread is preempted by the thread of higher priority,which gets a chance to run instead.

A thread might cease to runnable (that is,because blocked) for a variety of reasons.The thread’s code can execute a Thread.sleep() call,deliberately asking the thread to pause for a fixed period of time.The thread might have to wait to access a resource and cannot continue until that resource become available.

All thread that are runnable are kept in pools according to priority.When a blocked thread becomes runnable,it is placed back into the appropriate runnable pool.Threads from the highest priority nonempty pool are given CPU time.

The last sentence is worded loosed because:
(1) In most JVM implementations,priorities seem to work in a preemptive manner,although there is no guarantee that priorities have any meaning at all;
(2) Microsoft Window’s values affect thread behavior so that it is possible that a Java Priority 4 thread might be running,in spite of the fact that a runnable Java Priority 5 thread is waiting for the CPU.
In reality,many JVMs implement pool as queues,but this is not guaranteed hehavior.

线程调度（试翻译，欢迎指正）
在 java 技术中，线程通常是抢占式的而不需要时间片分配进程（分配给每个线程相等的 cpu 时间的进程）。一个经常犯的错误是认为“抢占”就是“分配时间片”。

在 Solaris 平台上的运行环境中，相同优先级的线程不能相互抢占对方的 cpu 时间。但是，在使用时间片的 windows 平台运行环境中，可以抢占相同甚 至更高优先级的线程的 cpu 时间。抢占并不是绝对的，可是大多数的 JVM 的实现结果在行为上表现出了严格的抢占。纵观 JVM 的实现，并没有绝对的抢占或是 时间片，而是依赖于编码者对 wait 和 sleep 这两个方法的使用。
抢占式调度模型就是许多线程属于可以运行状态（等待状态），但实际上只有 一个线程在运行。该线程一直运行到它终止进入可运行状态（等待状态）或是另一个具有更高优先级的线程变成可运行状态。在后一种情况下，底优先级的线程被高 优先级的线程抢占，高优先级的线程获得运行的机会。
线程可以因为各种各样的原因终止并进入可运行状态（因为堵塞）。例如，线程的代码可以在适当时候执行 Thread.sleep()方法，故意让线程中止；线程可能为了访问资源而不得不等待直到该资源可用为止。
所有可运行的线程根据优先级保持在不同的池中。一旦被堵塞的线程进入可运行状态，它将会被放回适当的可运行池中。非空最高优先级的池中的线程将获得 cpu 时间。
最后一个句子是不精确的，因为：
（1）在大多数的 JVM 实现中，虽然不能保证说优先级有任何意义，但优先级看起来象是用抢占方式工作。
（2）微软 windows 的评价影响线程的行为，以至尽管一个处于可运行状态的优先级为 5 的 java 线程正在等待 cpu 时间，但是一个优先级为 4 的 java 线程却可能正在运行。
实际上，许多 JVM 用队列来实现池，但没有保证行为。
from-Colin

Java Thread in JVM

本文从 JVM 的角度探讨 Java Thread 的语法和编译结果。如果需要获得第一手资料，请直接访问以下的资源——Java 语言规范，Java 虚拟机规范中有关线程的定义说明。

本文旨在介绍这些比较重要的线程相关的规范，基本上不另作发挥。（除了提到微软的“公共语言基础构造”。:-)

Java Language Specification

http://java.sun.com/docs/books/jls/second_edition/html/classes.doc.html#30531

JVM Specification

http://java.sun.com/docs/books/vmspec/2nd-edition/html/Compiling.doc.html#6530

http://java.sun.com/docs/books/vmspec/2nd-edition/html/Instructions2.doc9.html

http://java.sun.com/docs/books/vmspec/2nd-edition/html/Threads.doc.html

Microsoft CLI -- Common Language Infrastructure (sorry, off the topic :-)

http://msdn.microsoft.com/net/ecma/

1．synchronized method 的 java 语言规范

详见 http://java.sun.com/docs/books/jls/second_edition/html/classes.doc.html#30531。

用 synchronized 关键字修饰的方法，分为两种情况：(static)静态方法，和实例方法。

(static)静态方法的“锁”是这个拥有这个方法的对象的 Class 对象；实例方法的“锁”是 this，拥有这个方法的当前对象实例。

怎么理解这段话，看一看下面的例子就明白了。

下面两段代码的效果完全相同。代码 1 ==代码 2。

代码 1：

```
class Test {

int count;

synchronized void bump() { count++; }

static int classCount;

static synchronized void classBump() {

classCount++;

}

}
```

代码 2：

```
class BumpTest {
int count;
void bump() {
synchronized (this) {
count++;
}
}
static int classCount;
static void classBump() {
try {
synchronized (Class.forName("BumpTest")) {
classCount++;
}
} catch (ClassNotFoundException e) {
...
}
}
}
```

2．synchronized 关键字的编译结果

这一节，我们来看一看 synchronized 关键字编译之后的 java 虚拟机指令是什么。

如果需要第一手资料，请参见 java 虚拟机规范相关的部分

http://java.sun.com/docs/books/vmspec/2nd-edition/html/Compiling.doc.html#6530

这 段规范里面讲到，java 虚拟机规范提供两条指令，monitorenter 和 monitorexit，来支持线程。但是对于上一节讲到的，用 synchronized 修饰的方法来说，并不使用这两个方法，而只是简单地用 ACC_SYNCHRONIZED 标志修饰。虚拟机调用方法的时候会检查这 个标志，进行同步。

synchronized 语句的编译结果对应 monitorenter 和 monitorexit 两条指令。

比如，下面的代码：

```
void onlyMe(Foo f) {

synchronized(f) {

doSomething();

}

}
```

的编译结果是

```
Method void onlyMe(Foo)

0 aload_1 // Push f

1 astore_2 // Store it in local variable 2

2 aload_2 // Push local variable 2 (f)

3 monitorenter // Enter the monitor associated with f

4 aload_0 // Holding the monitor, pass this and...

5 invokevirtual #5 // ...call Example.doSomething()V

8 aload_2 // Push local variable 2 (f)
```

9 monitorexit // Exit the monitor associated with f

10 return // Return normally

11 aload_2 // In case of any throw, end up here

12 monitorexit // Be sure to exit monitor...

13 athrow // ...then rethrow the value to the invoker

3．monitorenter 和 monitorexit

详见 http://java.sun.com/docs/books/vmspec/2nd-edition/html/Instructions2.doc9.html

monitorenter 定义的一段节录：

Operation : Enter monitor for object

Operand Stack : ..., objectref ...

Description :

The objectref must be of type reference.

Each object has a monitor associated with it. The thread that executes monitorenter gains ownership of the monitor associated with objectref. If another thread already owns the monitor associated with objectref, the current thread waits until the object is unlocked, then tries again to gain ownership. If the current thread already owns the monitor associated with objectref, it increments a counter in the monitor indicating the number of times this thread has entered the monitor. If the monitor associated with objectref is not owned by any thread, the current thread becomes the owner of the monitor, setting the entry count of this monitor to 1.

这段话的意思是说，monitorenter 操作的目标一定要 是一个对象，类型是 reference。Reference 实际就是堆里的一个存放对象的地址。每个对象（reference）都有一个 monitor 对 应，如果有其它的线程获取了这个对象的 monitor，当前的线程就要一直等待，直到获得 monitor 的线程放弃 monitor，当前的线程才有机会获 得 monitor。

如果 monitor 没有被任何线程获取，那么当前线程获取这个 monitor，把 monitor 的 entry count 设置为 1。表示这个 monitor 被 1 个线程占用了。

当前线程获取了 monitor 之后，会增加这个 monitor 的时间计数，来记录当前线程占用了 monitor 多长时间。

我们看到，monitor 这个词在 java 虚拟机规范规定出现，但是在 java 语言和 API 文档里面并没有出现。monitor 是藏在线程同步后面的原理和概念。

4. Threads and Locks

详见 http://java.sun.com/docs/books/vmspec/2nd-edition/html/Threads.doc.html。

这段规范详细地介绍了 thread 和 lock 的原理。下面给出这段规范的 highlight。

8.4 Nonatomic Treatment of double and long Variables （double 和 long 类型的非原子操作。）

8.7 Rules for volatile Variables

8.10 Example: Possible Swap

8.11 Example: Out-of-Order Writes

如果对列出的这些 highlight 感兴趣，请访问相应的 java 虚拟机规范网址。

5. Why specification?

本文主要讨论 java 相关规范的内容。规范文档非常重要，尤其对于 java，C#这种生成中间代码的语言来说。

上面说的是 java 的相关规范。这里顺便提一下微软.Net 的相关规范。

微软的“公共语言基础构造”规范：

Microsoft CLI -- Common Language Infrastructure (sorry, off the topic :-)

http://msdn.microsoft.com/net/ecma/

这个网址上有 C#语言规范，CLI 规范的下载。

Java 中利用管道实现线程间的通讯

在 Java 语言中，提供了各种各样的输入输出流（stream）,使我们能够很方便的对数据进行操作，其中，管道（pipe）流是一种特殊的流，用于在不同线程 （threads）间直接传送数据。一个线程发送数据到输出管道，另一个线程从输入管道中读数据。通过使用管道，实现不同线程间的通讯。无需求助于类似临 时文件之类的东西。本文在简要介绍管道的基本概念后，将以一个具体的实例 pipeapp 加以详细说明。

1. 管道的创建与使用

Java 提供了两个特殊的专门的类专门用于处理管道，它们就是 pipedinputstream 类和 pipeoutputstream 类。

Pipedinputstream 代表了数据在管道中的输出端，也就是线程向管道读数据的一端；pipeoutputstream 代表了数据在管道中的输入端，也就是线程向管道写数据的一端，这两个类一起使用可以提供数据的管道流。

为了创建一个管道流，我们必须首先创建一个 pipeoutstream 对象，然后，创建 pipeinputstream 对象，实例如下：

```
pipeout= new pipedyoutstream();
pipein= new pipedputsteam(pipepout);
```

一旦创建了一个管道后，就可以象操作文件一样对管道进行数据的读写。

2. 演示程序： pipeapp

应用程序由三个程序组成：主线程（pipeapp.Java）及由主线程启动的两个二级线程（ythread.Java 和 zthread.Java），它们使用管道来处理数据。程序从一个内容为一行一行"x"字母的"input.txt"文件中读取数据，使用管道传输数据，第一次是利用线程 ythread 将数据"x"转换为"y"，最后利用线程 zthread 将"y"转换为"z",之后，程序在屏幕上显示修改后的数据。

主线程 （pipeapp.Java）

在 main()方法中，程序首先创建一个应用对象：pipeapp pipeapp=new pipeapp();

由于程序中流操作都需要使用 IOException 异常处理，所以设置了一个 try 块。在 try 中，为了从源文件中读取数据，程序为"input.txt"文件创建了一个输入流 Xfileln,:

fileinputstream xfileln= new fileinputstream("input.txt");

新的输入流传递给 changetoy()方法，让线程 ythread 能读取该文件：

inputstream ylnpipe =pipeapp.changetoy(xfileln);

changetoy()方法创建将输入数据"x"改变到"y"的线程 ythread,并返回该线程的输入管道：

inputstream zlnpipe = pipeapp.changetoz(ylnpipe);

changetoz()方法启动将数据从"y"改变到"z"的线程 zehread,主程序将使用从 changetoz()返回的输入管道。得到以修改的数据。

然后，程序将管道输入流定位到 datainputstream 对象，使程序能够使用 readline()方法读取数据：

datainputstream inputstream = new datainputstream(zlnpiepe);

创建了输入流以后，程序就可以以行一行的读取数据病显示在屏幕上。

```
String str= inputstream.readline();
While(str!=null)
{
    system.out.println(str);
    str=inputstream.readline();
}
```

显示完成之后，程序关闭输入流：

```
inputstream.close();
```

changetoy()方法

changetoy()方法首先通过传递一个参数 inputstream 给 datainputstream 对象来定位资源的输入流，使程序能使用 readline()方法从流中读取数据：

datainputstream xfileln =new datainutstream(inputstream)；

然后，changetoy()创建输出管道和输入管道：

```
pipeoutstream pipeout = new pipeoutputstream();
pipeinputstream pipeln = new pipedinputsteam(pipeout);
```

为了能够使用 println()方法输出修改的后的文本行到管道，程序将输出管道定位到 printstream 对象：

printstream printstream = new printstream(pipeout);

现在，程序可以创建将数据从 x 改变到 y 的线程，该线程是 ythread 类的一个对象，他传递两个参数：输入文件（xfileln）和输出管道（调用 printstream）

ythread ythread =new thread(xfileln,printstream);

之后，程序启动线程：

changetoz（）方法

changetoz()方法与 changetoy()方法很相似，他从 changetoy()返回的输入流开始：

datainputstream yfileln= new datainputstream(inputstream);

程序创建一个新的管道：

pipedoutstream pipeout2 = new pipedoutputstream();
pipedinputstream pipeln2 = new pipedinputsream(pipeout2);

该线程通过这个新的管道发出修改后的数据（输入流 pipeln2）给主程序。

源程序如下：

```
//
//pipeapp.Java-pipeapp 的主应用程序
//
import Java.io.*
class pipeapp
{
public static void main(string[] args)
{
pipeapp pipeapp=new pipeapp();
try
{
fileinputstream xfile =new fileinputstream("input.txt");
inputstream ylnpipe = pipeapp.changetoy(xfileln);
inputstream zlnpipe=pipeapp.changetoz(ylnpipe);
```

```
system.out.println();
system.out.println("here are the results");
system.out.pringln();
datainputstream inputstream = nes datainputstream(zlnpipe);
string str = inputstream.readline();
while (str!=null)
{
system.out.println(str);
str=inputstream.readline();
}
inputstream.close();
}
catch(exception e)
{
system.out.println(e.tostring());
}
}
public inputstream changetoy(inputstream inputstream)
{
try
{
datainputstream pipeout = new datainputsteam(inputstream);
pipedoutstream pipeout = new pipedoutputstream();
pipedlnsteam pipeln = new pipedlnputstream(pipeout);
printstream printstream = new printstream(pipeout);
ythread ythread = new ythread(xfileln,printstream);
ythread.start();
return pipeln;
}
catch(exeption e)
{
system.out.println(x.tostring());
}
return null;
}
public inputstream changetoz(inputstream inputsteam)
{
try
{
datainputstream yfileln = new datainputstream(inputstream);
pipeoutputstream pipeln2 = new pipedinputstream(pipeout2);
printrstream printstream2 = new printsteam(pipeout2);
zthread zthread = new zthread(yfileln,printstream2);
zthread.start();
```

```
return pipeln2;
}
catch(exception e)
{
system.out.println(e.tostring());
}
return null;
}
}
```

3．Ythread 类和 Zthread 类

由于 ythread 类与 zthread 类基本一样，在此仅以 ythread 为例加以说明。

Ythread 的构造器接收两个参数：输入的文件和第一个管道的输出端，构造器存储这两个参数作为类的数据成员：

```
Ythread(datainputstream xfileln,pringstream printstream)
{
    this.xfileln = xfileln;
    this.printstream = printstream;
}
```

线程通过 run()方法来处理数据。首先读取一行数据，确保 xstring 不为空的情况下循环执行：

```
string xstring = xfileln.readline();
```

每读一行数据，完成一次转换

```
string ystring = xstring.replace('x','y');
```

然后将修改后的数据输出到管道的输出端：

```
prinstream.prinrln(ystring);
```

为了确保所有缓冲区的数据完全进入管道的输出端：

pringstram.flush();

循环完成后，线程关闭管道输出流：

pringstram.close();

ythread 类的源程序如下：

```
//
//ythread.Java
//
import Java.io.*;
class ythread exteads thread
{
datainputstream xfileln;
pringstream printstream;
ythread(datainputstream xfileln,pringstream.printstream)
{
this.xfileln = xfileln;
this.printstream = printstream;
}
public void run()
{
try
{
string xstring = xfileln.readline();
while(xstring!=null)
{
string ystring= xstring.replace('x','y');
printstream.pringln(ystring);
printstream.flush();
xstring= xfileln.readline();
}
printstream.close();
}
catch{ioexception e}
{
system.out.println(e.tostring());
}
}
}
```

实战 Java 多线程编程精要之高级支持

但是，不可能预知哪个线程会获得这个通知，因为这取决于 Java 虚拟机 (JVM) 调度算法。将 CPU 让给另一个线程　　当线程放弃某个稀有的资源（如数据库连接或网络端口）时，它可能调用 yield() 函数临时降低自己的优先级，以便某个其他线程能够运行。

线程组

线程是被个别创建的，但可以将它们归类到线程组中，以便于调试和监视。只能在创建线程的同时将它与一个线程组相关联。在使用大量线程的程序中，使用线程组组织线程可能很有帮助。可以将它们看作是计算机上的目录和文件结构。

线程间发信

当线程在继续执行前需要等待一个条件时，仅有 synchronized 关键字是不够的。虽然 synchronized 关键字阻止并发更新一个对象，但它没有实现线程间发信。Object 类为此提供了三个函数：wait()、notify() 和 notifyAll()。以全球气候预测程序为例。这些程序通过将地球分为许多单元，在每个循环中，每个单元的计算都是隔离进行的，直到这些值趋于稳定， 然后相邻单元之间就会交换一些数据。所以，从本质上讲，在每个循环中各个线程都必须等待所有线程完成各自的任务以后才能进入下一个循环。这个模型称为 屏蔽同步，下例说明了这个模型：

屏蔽同步

数 notify() 只通知一个正在等待的线程，当对每次只能由一个线程使用的资源进行访问限制时，这个函数很有用。但是，不可能预知哪个线程会获得这个通知，因为这取决于 Java 虚拟机 (JVM) 调度算法。

将 CPU 让给另一个线程

当线程放弃某个稀有的资源（如数据库连接或网络端口）时，它可能调用 yield() 函数临时降低自己的优先级，以便某个其他线程能够运行。

守护线程

有两类线程：用户线程和守护线程。用户线程是那些完成有用工作的线程。守护线程是那些仅提供辅助功能的线程。Thread 类提供了 setDaemon() 函数。

Java 程序将运行到所有用户线程终止，然后它将破坏所有的守护线程。在 Java 虚拟机（JVM）中，即使在 main 结束以后，如果另一个用户线程仍在运行，则程序仍然可以继续运行。

JAVA 的多线程浅析

一 JAVA 语言的来源、及特点

在这个高速信息的时代，商家们纷纷把信息、产品做到 Internet 国际互连网页上。再这些不寻常网页的背后，要属功能齐全、安全可靠的编程语言，Java 是当之无愧的。Java 是由 Sun Microsystem 开发的一种功能强大的新型程序设计语言。是与平台无关的编程语言。它是一种简单的、面象对象的、分布式的、解释的、键壮的、安全的、结构的中立的、可移植的、性能很优异的、多线程的、动态的、语言。Java 自问世以后，以其编程简单、代码高效、可移植性强，很快受到了广大计算机编程人士的青睐。Java 语言是 Internet 上具有革命性的编程语言，它具有 强大的动画、多媒体和交互功能，他使 World Web 进入了一个全新的时代。Java 语言与 C++极为类似，可用它来创建安全的、可移植的、多线程的交互式程序。另外用 Java 开发出来的程序与平台无 关，可在多种平台上运行。后台开发，是一种高效、实用的编程方法。人们在屏幕前只能看到例如图案、计算的结果等。实际上操作系统往往在后台来调度一些事 件、管理程序的流向等。例如操作系统中的堆栈，线程间的资源分配与管理，内存的创建、访问、管理等。可谓举不盛举。下面就多线程来谈一谈。

二 JAVA 的多线程理论

2.1 引入

Java 提供的多线程功能使得在一个程序里可同时执行多个小任务。线程有时也称小进程是一个大进程里分出来的小的独立的进程。因为 Java 实现的多线程技术，所以比 C 和 C++更键壮。多线程带来的更大的好处是更好的交互性能和实时控制性能。当然实时控制性能还取决于系统本身(UNIX,Windows,Macintosh 等)，在开发难易程度和性能上都比单线程要好。传统编程环境通常是单线程的，由于 JAVA 是多线程的。尽管多线程是强大而灵巧的编程工具，但要用好却不容易，且有许多陷阱，即使编程老手也难免误用。为了更好的了解线程，用办公室工作人员作比喻。办公室工作人员就象 CPU，根据上级指示做工作，就象执行一个线程。在单线程环境中，每个程序编写和执行的方式是任何时候程序只考虑一个处理顺序。用我们的比喻，就象办公室工作人员从头到尾不受打扰和分心，只安排做一个工作。当然，实际生活中工作人员很难一次只有一个任务，更常见的是工作人员要同时做几件事。老板将工作交给工作人员，希望工作人员做一这个工作，再做点那个工作，等等。如果一个任务无法做下去了，比如工作人员等待另一部门的信息，则工作人员将这个工作放在一边，转入另一个工作。一般来说，老板希望工作人员手头的各个任务每一天

都有一些进展。这样就引入了多线程的概念。多线程编程环境与这个典型的办公室非常相似，同时给 CPU 分配了几个任务或线程。和办公室人员一样，计算机 CPU 实际上不可能同一时间做几件事，而是把时间分配到不同的线程，使每个线程都有点进展。如果一个线程无法进行，比如线程要求的键盘输入尚未取得，则转入另一线程的工作。通常，CPU 在线程间的切换非常迅速，使人们感觉到好象所有线程是同时进行的。

任何处理环境，无论是单线程还是多线程，都有三个关键方面。第一个是 CPU，它实际上进行计算机活动；第二个是执行的程序的代码；第三个是程序操作的数据.
。

在多线程编程中，每个线程都用编码提供线程的行为，用数据供给编码操作。多个线程可以同时处理同一编码和数据，不同的线程也可能各有不同的编码和数据。事实上编码和数据部分是相当独立的，需要时即可向线程提供。因此经常是几个线程使用同一编码和不同的数据。这个思想也可以用办公室工作人员来比喻。会计可能要做一个部门的帐或几个或几个部门的帐。任何情况的做帐的任务是相同的程序代码，但每个部门的数据是不同的。会计可能要做整个公司的帐，这时有几个任务，但有些数据是共享的，因为公司帐需要来自各个部门的数据。

多线程编程环境用方便的模型隐藏 CPU 在任务切换间的事实。模型允许假装成有多个可用的 CPU。为了建立另一个任务，编程人员要求另一个虚拟 CPU，指示它开始用某个数据组执行某个程序段。下面我们来建立线程。

建立线程

在 JAVA 中建立线程并不困难，所需要的三件事：执行的代码、代码所操作的数据和执行代码的虚拟 CPU。虚拟 CPU 包装在 Thread 类的实例中。建立 Thread 对象时，必须提供执行的代码和代码所处理的数据。JAVA 的面向对象模型要求程序代码只能写成类的成员方法。数据只能作为方法中的自动（或本地）变量或类的成员存在。这些规则要求为线程提供的代码和数据应以类的实例的形式出现。

```
Public class SimpleRunnable implemants Runable{
Private String message;
Public static void main(String args[]){
SimpleRunnable r1=new SimpleRunnable(“Hello”);
Thread t1=new Thread(r1);
t1.start();
}
public SimpleRunnable(String message){
this.message=message;
}
public void run(){
for(;;){
System.out.println(message);
}
}
}
```

线程开始执行时，它在 public void run()方法中执行。这种方法是定义的线程执行的起点，就象应用程序从 main()开始、小程序从 init()开始一样。线程操作的本地数据是传入线程的对象的成员。

首先，main()方法构造 SimpleRunnable 类的实例。注意，实例有自己的数据，这里是一个 String,初始化为”Hello”.由于实例 r1 传入 Thread 类构造器，这是线程运行时处理的数据。执行的代码是实例方法 run()。

2.2 线程的管理

单线程的程序都有一个 main 执行体，它运行一些代码，当程序结束执行后，它正好退出，程序同时结束运行。在 JAVA 中我们要得到相同的应答，必须稍微进行改动。只有当所有的线程退出后，程序才能结束。只要有一个线程一直在运行，程序就无法退出。线程包括四个状态：new(开始),running(运行),wait(等候)和 done(结束)。第一次创建线程时，都位于 new 状态，在这个状态下，不能运行线程，只能等待。然后，线程或者由方法 start 开始或者送往 done 状态，位于 done 中的线程已经结束执行，这是线程的最后一个状态。一旦线程位于这个状态，就不能再次出现，而且当 JAVA 虚拟机中的所有线程都位于 done 状态时，程序就强行中止。当前正在执行的所有线程都位于 running 状态，在程序之间用某种方法把处理器的执行时间分成时间片，位于 running 状态的每个线程都是能运行的，但在一个给定的时间内，每个系统处理器只能运行一个线程。与位于 running 状态的线程不同，由于某种原因，可以把已经位于 waiting 状态的线程从一组可执行线程中删除。如果线程的执行被中断，就回到 waiting 状态。用多种方法能中断一个线程。线程能被挂起，在系统资源上等候，或者被告知进入休眠状态。该状态的线程可以返回到 running 状态，也能由方法 stop 送入 done 状态，

方法
描述
有效状态
目的状态
Start()
开始执行一个线程
New
Running
Stop()
结束执行一个线程
New 或 running
Done
Sleep(long)
暂停一段时间，这个时间为给定的毫秒
Running
Wait
Sleep(long,int)
暂停片刻，可以精确到纳秒
Running
Wait
Suspend()
挂起执行
Running

Wait
Resume()
恢复执行
Wait
Running
Yield()
明确放弃执行
Running
Running

2.3 线程的调度

线程运行的顺序以及从处理器中获得的时间数量主要取决于开发者，处理器给每个线程分配一个时间片，而且线程的运行不能影响整个系统。处理器线程的系统或者是抢占式的，或者是非抢占式的。抢占式系统在任何给定的时间内将运行最高优先级的线程，系统中的所有线程都有自己的优先级。Thread.NORM_PRIORITY 是线程的缺省值，Thread 类提供了 setPriority 和 getPriority 方法来设置和读取优先权，使用 setPriority 方法能改变 Java 虚拟机中的线程的重要性，它调用一个整数，类变量 Thread.MIN_PRIORITY 和 Thread.MAX_PRIORITY 决定这个整数的有效范围。Java 虚拟机是抢占式的，它能保证运行优先级最高的线程。在 JAVA 虚拟机中我们把一个线程的优先级改为最高，那么他将取代当前正在运行的线程，除非这个线程结束运行或者被一条休眠命令放入 waiting 状态，否者将一直占用所有的处理器的时间。如果遇到两个优先级相同的线程，操作系统可能影响线程的执行顺序。而且这个区别取决于时间片（time slicing）的概念。

管理几个线程并不是真正的难题，对于上百个线程它是怎样管理的呢？当然可以通过循环，来执行每一个线程，但是这显然是冗长、乏味。JAVA 创建了线程组。线程组是线程的一个谱系组，每个组包含的线程数不受限制，能对每个线程命名并能在整个线程组中执行(Suspend)和停止(Stop)这样的操作。

2.4 信号标志：保护其它共享资源

这种类型的保护被称为互斥锁。某个时间只能有一个线程读取或修改这个数据值。在对文件尤其是信息数据库进行处理时，读取的数据总是多于写数据，根据这个情况，可以简化程序。下面举一例，假设有一个雇员信息的数据库，其中包括雇员的地址和电话号码等信息，有时要进行修改，但要更多的还是读数据，因此要尽可能防止数据被破坏或任意删改。我们引入前面互斥锁的概念，允许一个读取锁（red lock）和写入锁（write lock）,可根据需要确定有权读取数据的人员，而且当某人要写数据时，必须有互斥锁，这就是信号标志的概念。信号标志有两种状态，首先是 empty()状态，表示没有任何线程正在读或写，可以接受读和写的请求，并且立即提供服务；第二种状态是 reading()状态，表示有线程正在从数据库中读信息，并记录进行读操作的线程数，当它为 0 时，返回 empty 状态，一个写请求将导致这个线程进入等待状态。

只能从 empty 状态进入 writing 状态，一旦进入 writing 状态后，其它线程都不能写

操作，任何写或读请求都必须等到这个线程完成写操作为止，而且 waiting 状态中的进程也必须一直等到写操作结束。完成操作后，返回到 empty 状态，发送一个通知信号，等待的线程将得到服务。
下面实现了这个信号标志

```
class Semaphore{
final static int EMPTY=0;
final static int READING=1;
final static int WRITING=2;
protected int state=EMPTY;
protected int readCnt=0;
public synchronized void readLock(){
if(state==EMPTY){
state=READING;
}
else if(state==READING){
}
else if(state==WRITING){
while(state==WRITING){
try {wait();}
catch(InterruptedException e){;}
}
state=READING;
}
readCnt++;
return;
}
public synchronized void writeLock(){
if(state==EMPTY){
state=WRITING;
}
else{
while(state!=EMPTY){
try {wait();}
catch(InterruptedException e) {;}
}
}
}
public synchronized void readUnlock(){
readCnt--;
if(readCnt==0){
state=EMPTY;
notify();
}
}
```

```
public synchronized void writeUnlock(){
state=EMPTY;
notify();
}
}
```
现在是测试信号标志的程序：
```
class Process extends Thread{
String op;
Semaphore sem;
Process(String name,String op,Semaphore sem){
super(name);
this.op=op;
this.sem=sem;
start();
}
public void run(){
if(op
catch(InterruptedException e){;}
System.out.println("Unlocking readLock:"+getName());
sem.readUnlock();
}
else if(op
catch(InterruptedException e){;}
System.out.println("Unlocking writeLock:"+getName());
sem.writeUnlock();
}
}
}
public class testSem{
public static void main(String argv[]){
Semaphore lock = new Semaphore();
new Process("1","read",lock);
new Process("2","read",lock);
new Process("3","write",lock);
new Process("4","read",lock);
}
}
```
testSem 类从 process 类的四个实例开始，它是个线程，用来读或写一个共享文件。Semaphore 类保证访问不会破坏文件，执行程序，输出结果如下：
```
Trying to get readLock:1
Read op:1
Trying to get readLock:2
Read op:2
Trying to get writeLock:3
```

Trying to get readLock:4
Read op:4
Unlocking readLock:1
Unlocking readLock:2
Unlocking readLock:4
Write op:3
Unlocking writeLock:3
从这可看到，

2.5 死锁以及怎样避免死锁：

为了防止数据项目的并发访问，应将数据项目标为专用，只有通过类本身的实例方法的同步区访问。为了进入关键区，线程必须取得对象的锁。假设线程要独占访问两个不同对象的数据，则必须从每个对象各取一个不同的锁。现在假设另一个线程也要独占访问这两个对象，则该进程必须得到这两把锁之后才能进入。由于需要两把锁，编程如果不小心就可能出现死锁。假设第一个线程取得对象 A 的锁，准备取对象 B 的锁，而第二个线程取得了对象 B 的锁，准备取对象 A 的锁，两个线程都不能进入，因为两者都不能离开进入的同步块，既两者都不能放弃目前持有的锁。避免死锁要认真设计。线程因为某个先决条件而受阻时，如需要锁标记时，不能让线程的停止本身禁止条件的变化。如果要取得多个资源，如两个不同对象的锁，必须定义取得资源的顺序。如果对象 A 和 B 的锁总是按字母顺序取得，则不会出现前面说道的饿死条件。

三 Java 多线程的优缺点

由于 JAVA 的多线程功能齐全，各种情况面面具到，它带来的好处也是显然易见的。多线程带来的更大的好处是更好的交互性能和实时控制性能。当然实时控制性能还取决于系统本身(UNIX,Windows,Macintosh 等)，在开发难易程度和性能上都比单线程要好。当然一个好的程序设计语言肯定也难免有不足之处。由于多线程还没有充分利用基本 OS 的这一功能。这点我在前面已经提到，对于不同的系统，上面的程序可能会出现截然不同的结果，这使编程者偶会感到迷惑不解。希望在不久的将来 JAVA 的多线程能充分利用到操作系统，减少对编程者的困惑。我期待着 JAVA 会更好