# 多线程

多线程相关知识：

1. JavaSE : 多线程基础
2. JUC : 多线程高级

一些概念：

1. 程序： 完成某个任务、功能 用某种语言编写的一组指令的集合。
2. 软件： 软件 = 程序 + 程序运行所需文件 一个软件中可能有很多个程序
3. 进程： 程序的一次运行。

每个进程之间是独立的，操作系统在分配资源(内存)的时候，是以进程为单位的。因此进程之间进行通信（交换数据）时，成本比较高，相当于两个软件之间进行通信。因此引入线程

1. 线程： 进程中的某一条执行路径

同一个进程中的多个线程之间是可以共享部分内存的(方法区、堆)，每个线程的有些内存又是独立的。同一个进程的线程之间数据交换成本较低，线程的切换对于CPU和操作系统来说，成本比进程低的多。因此通常用多线程来代替多进程方式，实现多任务开发。

线程是CPU调度的最小单位

1. 并行：多个线程同时运行
2. 并发：多个进程同时运行

并发（Concurrent），在操作系统中，是指一个时间段中有几个程序都处于已启动运行到运行完毕之间，且这几个程序都是在同一个处理机上运行。

并发不是真正意义上的“同时进行”，只是CPU把一个时间段划分成几个时间片段（时间区间），然后在这几个时间区间之间来回切换，由于CPU处理的速度非常快，只要时间间隔处理得当，即可让用户感觉是多个应用程序同时在进行。如：打游戏和听音乐两件事情在同一个时间段内都是在同一台电脑上完成了从开始到结束的动作。那么，就可以说听音乐和打游戏是并发的。

并行：

并行（Parallel），当系统有一个以上CPU时，当一个CPU执行一个进程时，另一个CPU可以执行另一个进程，两个进程互不抢占CPU资源，可以同时进行，这种方式我们称之为并行(Parallel)。

其实决定并行的因素不是CPU的数量，而是CPU的核心数量，比如一个CPU多个核也可以并行。

1. CPU: 一个CPU同一时刻只能运行一个线程的任务

如何实现一个CPU多个线程并发？

CPU在多个线程之间快速的切换，人感觉不到

前面我们写的程序都是单线程的，它们都只有一条顺序执行流：程序从main方法开始执行，依次向下执行每行代码，如果程序执行某行代码遇到了阻塞，则程序将会停滞在该处。如果我们使用Eclipse等IDE工具的单步调试功能将可以非常清楚地看出这一点。

多线程听上去是非常专业的概念，但其实非常简单：单线程的程序只有一条执行路径，多线程的程序则包含多条执行路径，多条执行路径之间互不干扰，例如：

举例说明：

（1）单线程：

一旦前面的车，阻塞了，后面的车只能一直等待



（2）多线程：

其中一条车道阻塞了，不影响其他车道



Java语言提供了非常优秀的多线程支持，程序可以非常简单的方式来启动多线程，通过本章的学习，大家将会get到Java多线程编程的很多技能，包括创建、启动、控制线程，以及多线程的同步、通信操作，还可以通过线程池提高多线程性能。

## 14.1 线程概述

### 14.1.1 进程

几乎所有操作系统都支持进程的概念，所有运行中的任务通常对应一条进程（Process）。当一个程序进入内存运行，即变成一个进程。进程是处于运行过程中的程序，并且具有一定独立功能，进程是操作系统进行资源分配和调度的一个独立单位。

一般而言，进程包含如下三个特性：

* 独立性：进程是操作系统进行资源分配和调度的一个独立单位，每一个进程都拥有自己私有的地址空间。在没有经过进程本身允许的情况下，一个用户进程不可以直接访问其他进程的地址空间。哪怕在同一台计算机上运行，进程之间通信可能也需要通过网络、独立于进程的文件来交换数据。
* 动态性：程序只是一个静态的指令的集合，而进程是一个正在系统中运行的活动的指令集合，即进程是处于运行过程中的程序。在进程中加入了时间的概念，进程具有自己的生命周期和各种不同状态，这些概念在程序中都是不具备的。
* 并发性：多个进程可以在单个处理器上并发执行，多个进程之间不会相互影响。

现代的操作系统都支持多进程的并发，但在具体的实现细节上可能因为硬件和操作系统的不同而采用不同的策略，目前大多数采用效率更高的抢占式多任务策略。对于一个CPU而言，它在某个时间点上只能执行一个程序，也就是只能运行一个进程，CPU不断的在这些进程之间轮换执行。那么，我们为什么可以一边用开发工具写程序，一边听音乐，一边还上网查资料.....，我们没有感觉到中断和轮换呢？这是因为CPU的执行速度相对于我们的感知速度来说实在是太快了，所以我们感觉像是同时运行一样。但是当我们启动足够多的程序时，依然可以感觉到运行速度下降的。

### 14.1.2 线程

多线程则扩展了多进程的概念，使得通一个进程可以同时并发处理多个任务。线程（Thread）也被称为轻量级进程。就像进程在操作系统中地位一样，线程在进程中也是独立的、并发的执行流。当进程被初始化后，主线程就被创建了，对于Java程序来说，main线程就是主线程，但我们可以在该进程内创建多条顺序执行路径，这些独立的顺序执行路径就是线程。

进程中的每一个线程可以完成一定的任务，并且是独立的，线程可以拥有自己独立的堆栈、自己的程序计数器和自己的局部变量，但不再拥有系统资源，它与父进程的其他线程共享该进程所拥有的全部资源。由于线程间的通信是在同一个地址空间上进行的，所以不需要额外的通信机制，这就使得通信更简便而且信息传递的速度也更快，因此可以通过简单编程实现多线程相互协同来完成进程所要完成的任务。但是也存在安全问题，因为其中一个线程对共享的系统资源的操作都会给其他线程带来影响，由此可知，多线程中的同步是非常重要的问题。

线程的执行也是抢占式的，也就是说，当前运行的线程在任何时候都可能被挂起，以便另一个线程可以运行。我们说CPU在不同的进程之间轮换，进程又在不同的线程之间轮换，因此线程是CPU执行和调度的最小单元。

总之，一个程序运行后至少有一个进程，一个进程里可以包含多个线程，但至少要包含一个线程。当操作系统创建一个进程时，必须为该进程分配独立的内存空间，并分配大量的相关资源；但创建一个线程简单的多，而且多个线程共享同一个进程的虚拟空间，所以，使用多线程来实现并发比使用多进程实现并发的性能要高的多。

在实际应用中，多线程是非常有用的，一个Web服务器必须能同时响应多个客户端的请求；一个浏览器必须能同时下载多个图片；一个在线播放器必须能一边下载一边播放......

|  |
| --- |
| 注意：  并发性（concurrency）和并行性（parallel）是两个概念，并行是指在同一时刻，有多条指令在多个处理器上同时执行；并发是指在同一个时刻只能有一条指令执行，但多个进程的指令被快速轮换执行，使得在宏观上具有多个进程同时执行的效果。 |

## 14.2 线程的创建和启动

Java使用Thread类代表线程，所有的线程对象都必须是Thread类或其子类的对象。每个线程的作用是完成一定的任务，实际上就是执行一段代码，我们称之为线程执行体。Java使用run方法来封装这段代码，即run方法的方法体就是线程执行体。

### 14.2.1 继承Thread类创建线程类

通过继承Thread类来创建并启动多线程的步骤如下：

（1）定义继承Thread的子类，并重写该类的run()。查看Thread类的源代码，发现当我们没有传target时，那么Thread类的run()就是一个什么也没干的方法，因此我们必须重写run()来完成线程的任务。

|  |
| --- |
| public void run() {  if (target != null) {  target.run();  }  } |

（2）创建Thread子类的实例，一个实例对象就是一个线程。

（3）调用线程对象的start()来启动线程。

示例代码：

|  |
| --- |
| **package** com.atguigu.part01.thread;  **public** **class** TestMyThread {  **public** **static** **void** main(String[] args) {  MyThread my1 = **new** MyThread();  my1.start();    MyThread my2 = **new** MyThread();  my2.start();    **for** (**int** i = 10; i >= 1; i--) {  System.***out***.println(Thread.currentThread().getName() + "线程：" + i);  }  }  }  **class** MyThread **extends** Thread{  **public** **void** run(){  **for** (**int** i = 1; i <= 10; i++) {  System.***out***.println(**super**.getName() + "线程：" + i);  }  }  } |

说明：

* Thread.currentThread()方法总是返回当前在执行的线程对象；
* getName()方法是Thread的实例方法，该方法返回当前线程对象的名字，可以通过setName(String name)方法设置线程名称，否则依次为Thread-0，Thread-1......等

提示：

（1）JavaSE的程序至少有一个main主线程，它的方法体即是线程体。

（2）Thread-0和Thread-1线程虽然都是MyThread类的线程对象，但是各自调用自己的run()，互相之间是独立的，因此各打印1-10。

（3）启动线程用start()方法，而不是run()！调用start()方法来启动线程，系统会把该run()方法当成线程执行体来处理。但是如果直接调用run()，系统会把线程对象当成一个普通对象，而run()就是一个普通方法。

### 14.2.2 实现Runnable接口

实现Runnable接口来创建并启动多线程的步骤：

1. 定义Runnable接口的实现类，并重写该接口的run()方法
2. 创建Runnable实现类的对象
3. 创建Thread类的对象，并将Runnable实现类的对象作为target。其实该Thread类的对象才是真正的线程对象。当JVM调用线程对象的run()时，如果target不为空，就会调用target的run()。

|  |
| --- |
| public void run() {  if (target != null) {  target.run();  }  } |

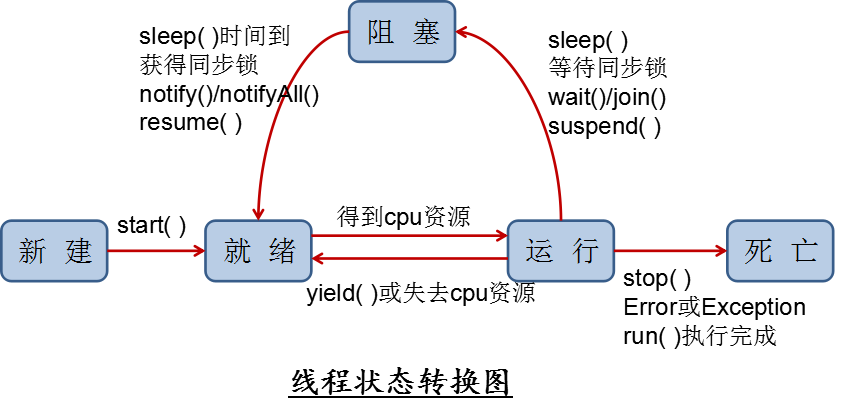
1. 调用线程对象的start()启动线程。

示例代码：

|  |
| --- |
| **package** com.atguigu.part01.thread;  **public** **class** TestMyRunnable {  **public** **static** **void** main(String[] args) {  MyRunnable my = **new** MyRunnable();  **new** Thread(my).start();  **new** Thread(my).start();  **for** (**int** i = 10; i >= 1; i--) {  System.***out***.println(Thread.*currentThread*().getName() + "线程：" + i);  }  }  }  **class** MyRunnable **implements** Runnable{  **public** **void** run(){  **for** (**int** i = 1; i <= 10; i++) {  System.***out***.println(Thread.*currentThread*().getName() + "线程：" + i);  }  }  } |

## 14.3 线程的生命周期

一个完整的生命周期中通常要经历如下的五种状态：新建（New）、就绪（Runnable）、运行（Running）、阻塞（Blocked）、死亡（Dead）。CPU需要在多条线程之间切换，于是线程状态会多次在运行、阻塞、就绪之间切换。



1. **新建**

**new之后**

当一个Thread类或其子类的对象被声明并创建时，新生的线程对象处于新建状。此时它和其他Java对象一样，仅仅由JVM为其分配了内存，并初始化了实例变量的值。此时的线程对象并没有任何线程的动态特征，程序也不会执行它的线程体run()。

1. **就绪**

**Start()之后**

当线程对象调用了start()方法之后，就不一样了，线程就从新建状态转为就绪状态。JVM会为其创建方法调用栈和程序计数器，当然，处于这个状态中的线程并没有开始运行，只是表示已具备了运行的条件，随时可以被调度。至于什么时候被调度，取决于JVM里线程调度器的调度。

1. **运行**

**获取cpu**

如果处于就绪状态的线程获得了CPU，开始执行run()方法的线程体代码，则该线程处于运行状态。如果计算机只有一个CPU，在任何时刻只有一个线程处于运行状态，如果计算机有多个处理器，将会有多个线程并行（Parallel）执行。

当然，美好的时光总是短暂的，而且CPU讲究雨露均沾。对于抢占式策略的系统而言，系统会给每个可执行的线程一个小时间段来处理任务，当该时间用完，系统会剥夺该线程所占用的资源，让其回到就绪状态等待下一次被调度。此时其他线程将获得执行机会，而在选择下一个线程时，系统会适当考虑线程的优先级。

1. **阻塞**

当在运行过程中的线程遇到如下情况时，线程会进入阻塞状态：

* Sleep(): 线程调用了sleep()方法，主动放弃所占用的CPU资源；
* 线程调用了一个阻塞式IO方法，在该方法返回之前，该线程被阻塞；
* 没有锁：线程试图获取一个**同步监视器**，但该同步监视器正被其他线程持有；
* Wait():线程执行过程中，**同步监视器**调用了wait()，让它等待某个通知（notify）；
* Join():线程执行过程中，遇到了其他线程对象的加塞（join）；
* 线程被调用suspend方法被挂起（已过时，因为容易发生死锁）；

**阻塞有两种原因：1.没有CPU 2.没有锁**

当前正在执行的线程被阻塞后，其他线程就有机会执行了。针对如上情况，当发生如下情况时会解除阻塞，让该线程重新进入就绪状态，等待线程调度器再次调度它：

* 线程的sleep()时间到,或者sleep()被打断interrupt()
* 线程调用的阻塞式IO方法已经返回；
* 线程成功获得了同步监视器；
* 线程等到了通知(notify)；
* 加塞的线程结束了；
* 被挂起的线程又被调用了resume恢复方法（已过时，因为容易发生死锁）；

1. **死亡**

线程会以以下三种方式之一结束，结束后的线程就处于死亡状态：

* run()方法执行完成，线程正常结束
* 线程执行过程中遇到了异常没有处理
* 被其他线程调用该线程的stop()来结束该线程（已过时，因为容易发生死锁）

可以调用线程的isAlive()方法判断该线程是否死亡，当线程处于就绪、运行、阻塞三种状态时，该方法返回true，当线程处于新建、死亡两种状态时，该方法将返回false。

|  |
| --- |
| 注意：  程序只能对新建状态的线程调用start()，并且只能调用一次，如果对非新建状态的线程，如已启动的线程或已死亡的线程调用start()都会报错IllegalThreadStateException异常。 |

## 14.4 Thread类的方法

### 14.4.1 创建线程对象相关

* 构造器：
* Thread()：创建新的Thread对象
* Thread(String threadname)：创建线程并指定线程实例名
* Thread(Runnable target)：指定创建线程的目标对象，它实现了Runnable接口中的run方法
* Thread(Runnable target, String name)：创建新的Thread对象，并命名
* 线程体和启动线程：
* public void run()：子类必须重写run()以编写线程体
* public void start()：启动线程

### 14.4.2 获取和设置线程信息

1. public **static** Thread currentThread()：这是一个静态方法，总是返回当前在执行的线程对象
2. public final boolean isAlive()：测试线程是否处于活动状态。如果线程已经启动且尚未终止，则为活动状态。
3. public final String getName()：getName()方法是Thread的实例方法，该方法返回当前线程对象的名字，可以通过
4. public final void setName(String name)：设置该线程名称。除了主线程main之外，其他线程可以在创建时指定线程名称或通过setName(String name)方法设置线程名称，否则依次为Thread-0，Thread-1......等。
5. public final int getPriority() ：返回线程优先值
6. public final void setPriority(int newPriority) ：改变线程的优先级

每个线程都有一定的优先级，优先级高的线程被CPU调度的概率增高。因此优先级不能够解决线程调度的先后顺序，只能解决频率。

每个线程默认的优先级都与创建它的父线程具有相同的优先级。

Thread类提供了setPriority(int newPriority)和getPriority()方法类设置和获取线程的优先级，其中setPriority方法需要一个整数，并且范围在[1,10]之间。

Thread类中维护了三个优先级常量，通常推荐设置Thread类的三个优先级常量：

* MAX\_PRIORITY （10）：最高优先级
* MIN \_PRIORITY （1）：最低优先级
* NORM\_PRIORITY （5）：普通优先级，默认情况下main线程具有普通优先级。

### 14.4.3 控制线程

Thread类提供了一些方法，可以控制线程的执行。

#### 1、线程睡眠：sleep

如果当我们需要让当前正在执行的线程暂停一段时间，则可以通过调用Thread类的静态sleep()方法，它会导致当前线程进入阻塞状态：

* public static void sleep(long millis)：在指定的毫秒数内让当前正在执行的线程休眠（暂停执行），此操作受到系统计时器和调度程序精度和准确性的影响。
* public static void sleep(long millis,int nanos)：在指定的毫秒加纳秒内让当前正在执行的线程休眠（暂停执行），此操作受到系统计时器和调度程序精度和准确性的影响。

代码示例：倒计时

|  |
| --- |
| **package** com.atguigu.part02;  **public** **class** TestSleep {  **public** **static** **void** main(String[] args) {  **for** (**int** i = 10; i>=0; i--) {  System.***out***.println(i);  **try** {  Thread.*sleep*(1000);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  System.***out***.println("新年快乐！");  }  } |

#### 2、线程让步：yield

yield()也是一个静态方法，它可以让当前正在执行的线程暂停，但它不会阻塞该线程，它只是将该线程转入就绪状态。

yield只是让当前线程暂停一下，让系统的线程调度器重新调度一次，希望优先级与当前线程相同或更高的其他线程能够获得执行机会，但是这个不能保证，完全有可能的情况是，当某个线程调用了yield方法暂停之后，线程调度器又将其调度出来重新执行。

|  |
| --- |
| **package** com.atguigu.part02;  **public** **class** TestYield {  **public** **static** **void** main(String[] args) {  MyYieldThread m1 = **new** MyYieldThread("低");  m1.setPriority(Thread.***MIN\_PRIORITY***);  MyYieldThread m2 = **new** MyYieldThread("高");  m2.setPriority(Thread.***MAX\_PRIORITY***);    m1.start();  m2.start();  }  }  **class** MyYieldThread **extends** Thread{  **public** MyYieldThread(String name) {  **super**(name);  }  **public** **void** run(){  **for** (**int** i = 1; i <= 10; i++) {  System.***out***.println(getName() + ":" +i);  Thread.*yield*();  }  }  } |

#### 3、线程加塞：join

当在某个线程的线程体中调用了另一个线程的join()方法，当前线程将被阻塞，直到join进来的线程执行完它才能继续。

* void join() ：等待该线程终止。
* void join(long millis) ：等待该线程终止的时间最长为 millis 毫秒。如果millis时间到，将不再等待。
* void join(long millis, int nanos) ：等待该线程终止的时间最长为 millis 毫秒 + nanos 纳秒。

示例代码：

|  |
| --- |
| **package** com.atguigu.part02;  **import** java.util.Scanner;  **public** **class** TestJoin {  **public** **static** **void** main(String[] args) {  ChatThread t = **new** ChatThread();  t.start();    **for** (**int** i = 1; i < 10; i++) {  System.***out***.println("main:" + i);  **try** {  Thread.*sleep*(10);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  //当main打印到5之后，需要等join进来的线程停止后才会继续了。  **if**(i==5){  **try** {  t.join();  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  }  }  }  **class** ChatThread **extends** Thread{  **public** **void** run(){  Scanner input = **new** Scanner(System.***in***);  **while**(**true**){  System.***out***.println("是否结束？（Y、N）");  **char** confirm = input.next().charAt(0);  **if**(confirm == 'Y' || confirm == 'y'){  **break**;  }  }  input.close();  }  } |

案例：编写龟兔赛跑多线程程序，设赛跑长度为30米

兔子的速度是10米每秒，兔子每跑完10米休眠的时间10秒

乌龟的速度是1米每秒，乌龟每跑完10米的休眠时间是1秒

要求：要等兔子和乌龟的线程结束，主线程（裁判）才能公布最后的结果。

|  |
| --- |
| **package** com.atguigu.part02;  **public** **class** Racer **extends** Thread {  **private** String name;//运动员名字  **private** **long** runTime;//每米需要时间，单位毫秒  **private** **long** restTime;//每10米的休息时间，单位毫秒  **private** **long** distance;//全程距离，单位米  **private** **long** time;//跑完全程的总时间  **public** Racer(String name, **long** distance, **long** runTime, **long** restTime) {  **super**();  **this**.name = name;  **this**.distance = distance;  **this**.runTime = runTime;  **this**.restTime = restTime;  }  @Override  **public** **void** run() {  **long** sum = 0;  **long** start = System.*currentTimeMillis*();  **while** (sum < distance) {  System.***out***.println(name + "正在跑...");  **try** {  Thread.*sleep*(runTime);// 每米距离，该运动员需要的时间  } **catch** (InterruptedException e) {  **return** ;  }  sum++;  **try** {  **if** (sum % 10 == 0 && sum < distance) {  // 每10米休息一下  System.***out***.println(name+"已经跑了"+sum+"米正在休息....");  Thread.*sleep*(restTime);  }  } **catch** (InterruptedException e) {  **return** ;  }  }  **long** end = System.*currentTimeMillis*();  time = end - start;  System.***out***.println(name+"跑了"+sum+"米，已到达终点，共用时"+time/1000.0+"秒");  }  **public** **long** getTime() {  **return** time;  }    } |
| **package** com.atguigu.part02;  **public** **class** TestJoin2 {  **public** **static** **void** main(String[] args) {  Racer rabbit = **new** Racer("兔子", 30, 100, 10000);  Racer turtoise = **new** Racer("乌龟", 30, 1000, 1000);    rabbit.start();  turtoise.start();    **try** {  rabbit.join();  } **catch** (InterruptedException e) {  e.printStackTrace();  }  **try** {  turtoise.join();  } **catch** (InterruptedException e) {  e.printStackTrace();  }  //因为要兔子和乌龟都跑完，才能公布结果  System.***out***.println("比赛结束");  System.***out***.println(rabbit.getTime()<turtoise.getTime()?"兔子赢":"乌龟赢");  }  } |

#### 4、守护线程

有一种线程，它是在后台运行的，它的任务是为其他线程提供服务的，这种线程被称为“守护线程”。JVM的垃圾回收线程就是典型的守护线程。

守护线程有个特点，就是如果所有非守护线程都死亡，那么守护线程自动死亡。

调用setDaemon(true)方法可将指定线程设置为守护线程。必须在线程启动之前设置，否则会报IllegalThreadStateException异常。

调用isDaemon()可以判断线程是否是守护线程。

|  |
| --- |
| **package** com.atguigu.part02;  **public** **class** TestDaemon {  **public** **static** **void** main(String[] args) {  MyDaemon m = **new** MyDaemon();  m.setDaemon(**true**);  m.start();    **for** (**int** i = 0; i < 20; i++) {  System.***out***.println("main:" + i);  }  }  }  **class** MyDaemon **extends** Thread{  **public** **void** run(){  **while**(**true**){  System.***out***.println("MyDaemon");  **try** {  Thread.*sleep*(1);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  }  } |

#### 5、停止线程

我们知道线程体执行完，或遇到未捕获的异常自然会停止，但是有时我们希望由另一个线程检测到某个情况时，停止一个线程，该怎么做呢？

Thread类提供了stop()来停止一个线程，但是该方法具有固有的不安全性，已经标记为@Deprecated不建议再使用，那么我们就需要通过其他方式来停止线程了，其中一种方式是使用标识。

案例：编写龟兔赛跑多线程程序，设赛跑长度为30米

兔子的速度是10米每秒，兔子每跑完10米休眠的时间10秒

乌龟的速度是1米每秒，乌龟每跑完10米的休眠时间是1秒

要求：只要兔子和乌龟中有人到达终点，就宣布比赛结束，没到达终点的也停下来。

|  |
| --- |
| **package** com.atguigu.part02;  **public** **class** Player **extends** Thread{  **private** String name;//运动员名字  **private** **long** runTime;//每米需要时间，单位毫秒  **private** **long** restTime;//每10米的休息时间，单位毫秒  **private** **long** distance;//全程距离，单位米  **private** **long** time;//跑完全程的总时间  **private boolean flag = true;//用于标记是否继续跑，即结束线程的标记**  **private volatile boolean ended = false;**//用于标记是否到达终点  **public** Player(String name, **long** distance, **long** runTime, **long** restTime) {  **super**();  **this**.name = name;  **this**.distance = distance;  **this**.runTime = runTime;  **this**.restTime = restTime;  }  @Override  **public** **void** run() {  **long** sum = 0;  **long** start = System.*currentTimeMillis*();  **while** (sum < distance **&& flag**) {  System.***out***.println(name + "正在跑...");  **try** {  Thread.*sleep*(runTime);// 每米距离，该运动员需要的时间  } **catch** (InterruptedException e) {  **return** ;  }  sum++;  **try** {  **if** (sum % 10 == 0 && sum < distance **&& flag)** {  // 每10米休息一下  System.***out***.println(name+"已经跑了"+sum+"米正在休息....");  Thread.*sleep*(restTime);  }  } **catch** (InterruptedException e) {  **return** ;  }  }  **long** end = System.*currentTimeMillis*();  time = end - start;  ended = sum == distance ? **true** : **false**;  System.***out***.println(name+"跑了"+sum+"米，共用时"+time/1000.0+"秒");  }  **public** **long** getTime() {  **return** time;  }  **public** **void** setFlag(**boolean** flag) {  **this**.flag = flag;  }  **public** **boolean** isEnded() {  **return** ended;  }    } |
| **package** com.atguigu.part02;  **public** **class** TestStop {  **public** **static** **void** main(String[] args) {  Thread.*currentThread*().setPriority(Thread.***MAX\_PRIORITY***);  Player rabbit = **new** Player("兔子", 30, 100, 10000);  Player turtoise = **new** Player("乌龟", 30, 1000, 1000);    rabbit.start();  turtoise.start();    **while**(**true**){  **if**(rabbit.isEnded() || turtoise.isEnded()){  rabbit.setFlag(**false**);  turtoise.setFlag(**false**);  //只要又人跑完，就结束比赛，并公布结果  **break**;  }  }    System.***out***.println("比赛结束");  **if**(rabbit.isEnded() && turtoise.isEnded()){  System.***out***.println(rabbit.getTime()<turtoise.getTime()?"兔子赢":"乌龟赢");  }**else** **if**(rabbit.isEnded()){  System.***out***.println("兔子赢");  }**else** **if**(turtoise.isEnded()){  System.***out***.println("乌龟赢");  }  }  } |

提示：

volatile的作用是确保不会因编译器的优化而省略某些指令，volatile的变量是说这变量可能会被意想不到地改变，每次都小心地重新读取这个变量的值，而不是使用保存在寄存器里的备份，这样，编译器就不会去假设这个变量的值了。

## 14.5 线程安全问题与线程同步

多线程编程是有趣且复杂的事情，它常常容易突然出现“错误情况”，这是由于系统的线程调度具有一定的随机性。即使程序在运行过程中偶尔会出现问题，那也是由于我们的代码有问题导致的。当多个线程访问同一个数据时，非常容易出现线程安全问题。

### 14.5.1 线程安全问题

关于线程安全问题，有一个经典的问题：卖票问题。卖票的基本流程很简单：看是否还有票，如果有就可以卖。

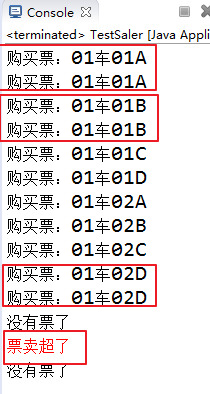
现在开启多个窗口同时卖票：

|  |
| --- |
| **package** com.atguigu.part03;  **import** java.util.ArrayList;  **public** **class** TicketService {  **private** ArrayList<String> list;  **public** TicketService(){  list = **new** ArrayList<String>();  list.add("01车01A");  list.add("01车01B");  list.add("01车01C");  list.add("01车01D");  list.add("01车01E");  list.add("01车02A");  list.add("01车02B");  list.add("01车02C");  list.add("01车02D");  list.add("01车02E");  }  **public** **boolean** hasTicket(){  **return** list.size()>0;  }    **public** String buy(){  **try** {  **return** list.remove(0);  } **catch** (IndexOutOfBoundsException e) {  **throw** **new** RuntimeException("票卖超了");  }  }  } |

现在有两个窗口同时卖票。

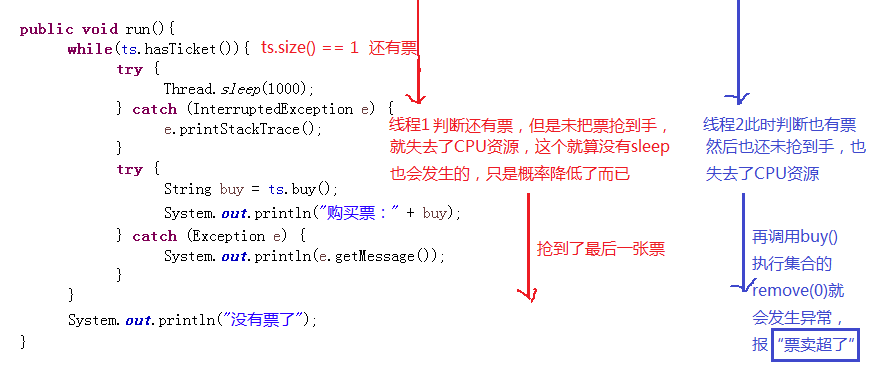
|  |
| --- |
| **package** com.atguigu.part03;  **public** **class** Saler **extends** Thread{  **private** TicketService ts;  **public** Saler(TicketService ts) {  **super**();  **this**.ts = ts;  }    **public** **void** run(){  **while**(ts.hasTicket()){  **try** {  Thread.*sleep*(100);  //这里加入休眠时间，是强制让线程切换发生，增大问题出现的概率，好让大家看效果  } **catch** (InterruptedException e) {  e.printStackTrace();  }  **try** {  String buy = ts.buy();  System.***out***.println("购买票：" + buy);  } **catch** (Exception e) {  System.***err***.println(e.getMessage());  }  }  System.***out***.println("没有票了");  }  } |
| **package** com.atguigu.part03;  **public** **class** TestSaler {  **public** **static** **void** main(String[] args) {  TicketService ts = **new** TicketService();  Saler s1 = **new** Saler(ts);  Saler s2 = **new** Saler(ts);    s1.start();  s2.start();  }  } |

运行结果：

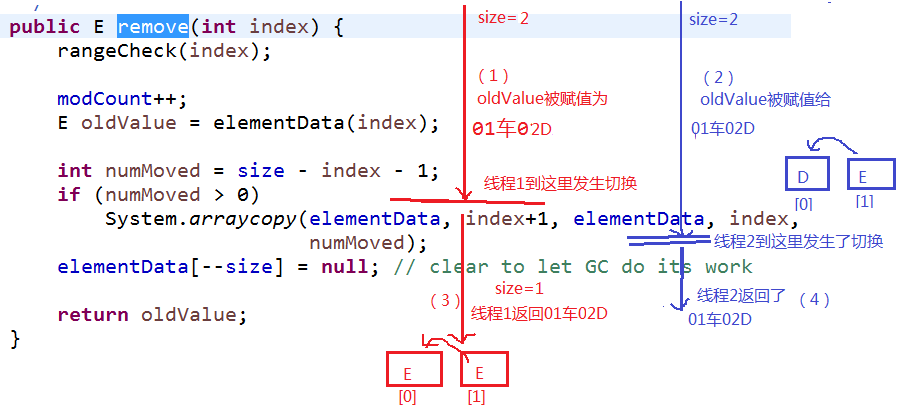


这里就发生了线程安全问题，当多个线程多条语句（size()和remove()）访问共享数据（这里TicketService的list存票的集合）时，就会发生线程安全问题。

解释1：“票卖超了”如何发生的，不是先判断是否有票，才买的吗？



解释2：重票01车02D又是怎么回事？



同理重票01车01A和01车01B的重票也是类似的情况，只不过发生线程切换的代码位置在System.arraycopy()方法中，线程1把[1]赋值给[0]后发生切换。这里就不再累述。

### 14.5.2 同步代码块

如何解决线程安全问题呢？

解决思路：对多条操作共享数据的语句，只能让一个线程都执行完，在执行过程中，其他线程不可以参与执行。



为了解决这个问题，Java的多线程支持引入同步监视器来解决这个问题。使用同步监视器的方式有两种：同步代码块和同步方法。

同步代码块的语法格式如下：

|  |
| --- |
| synchronized(同步监视器对象){  //.....  } |

上面代码的含义，线程开始执行同步代码块之前，必须先获得对同步监视器的锁定，换句话说没有获得对同步监视器的锁定，就不能进入同步代码块的执行，线程就会进入阻塞状态，直到对方释放了对同步监视器对象的锁定。

**任何时刻只能有一个线程可以获得对同步监视器的锁定，当同步代码块执行结束后，该线程自然会释放对同步监视器对象的锁定。**

Java程序运行使用任何对象来作为同步监视器对象，只要保证共享资源的这几个线程，锁的是**同一个**同步监视器对象即可。

#### 1、选择共享资源对象作为同步监视器对象

|  |
| --- |
| **package** com.atguigu.part03;  **public** **class** Saler **extends** Thread{  **private** TicketService ts;  **public** Saler(TicketService ts) {  **super**();  **this**.ts = ts;  }  **public** **void** run(){  **while**(**true**){  **synchronized** (ts) {  **if**(ts.hasTicket()){  **try** {  Thread.*sleep*(100);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  **try** {  String buy = ts.buy();  System.***out***.println("购买票：" + buy);  } **catch** (Exception e) {  System.***out***.println(e.getMessage());  }  }**else**{  System.***out***.println("没有票了");  **break**;  }  }  }  }  } |

#### 2、选择this对象作为同步监视器对象

如果线程是继承Thread类实现的，那么把同步监视器对象换成this，那么就没有起到作用，仍然会发生线程安全问题。因为两个线程的this对象是不同的。

但是如果线程是实现Runnable接口实现的，那么如果两个线程共用同一个Runnable接口实现类对象作为target的话，就可以把同步监视器对象换成this。

|  |
| --- |
| **package** com.atguigu.part03;  **public** **class** Window **implements** Runnable{  **private** TicketService ts;  **public** Window(TicketService ts) {  **super**();  **this**.ts = ts;  }  **public** **void** run(){  **while**(**true**){  **synchronized** (**this**) {  **if**(ts.hasTicket()){  **try** {  Thread.*sleep*(100);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  **try** {  String buy = ts.buy();  System.***out***.println("购买票：" + buy);  } **catch** (Exception e) {  System.***out***.println(e.getMessage());  }  }**else**{  System.***out***.println("没有票了");  **break**;  }  }  }  }  } |
| **package** com.atguigu.part03;  **public** **class** TestWindow {  **public** **static** **void** main(String[] args) {  TicketService ts = **new** TicketService();  Window w = **new** Window(ts);  Thread t1 = **new** Thread(w);  Thread t2 = **new** Thread(w);    t1.start();  t2.start();  }  } |

#### 3、继承Thread类和实现Runnable接口两种方式的区别

通过继承Thread类或实现Runnable接口都可以实现多线程，但是两种方式存在一定的差别：

**采用继承Thread的方式：**

**优势：**

（1）直接线程对象启动，比较简单。

（2）如果在线程体中要访问当前线程，直接this即可。

**劣势：**

（1）Java有单继承线程，继承了Thread类就不能再继承其他类了。

（2）如果多个线程要共享数据，就比较麻烦，使用static的变量共享，范围又太大。

**采用Runnable的方式：**

**优势：**

（1）避免单继承；

（2）可以多个线程共享同一个target对象，所以非常适合多个相同线程来处理同一份资源的情况，从而可以将CPU、代码和数据分开，形成清晰的模型，较好的体现了面向对象的思想。

**劣势：**

（1）启动线程需要再new Thread的对象；

（2）在线程体中要访问当前对象，需要用Thread.currentThread()先获取当前线程对象。

### 14.5.3 同步方法

与同步代码块对应的，Java的多线程安全支持还提供了同步方法，同步方法就是使用synchronized关键字来修饰某个方法，则该方法称为同步方法。对于同步方法而言，无须显式指定同步监视器，静态方法的同步监视器对象是当前类的Class对象，非静态方法的同步监视器对象是调用当前方法的this对象。

|  |
| --- |
| **package** com.atguigu.part03;  **import** java.util.ArrayList;  **public** **class** TicketService {  **private** ArrayList<String> list;  **public** TicketService(){  list = **new** ArrayList<String>();  list.add("01车01A");  list.add("01车01B");  list.add("01车01C");  list.add("01车01D");  list.add("01车01E");  list.add("01车02A");  list.add("01车02B");  list.add("01车02C");  list.add("01车02D");  list.add("01车02E");  }  **public** **synchronized** **boolean** hasTicket(){  **return** list.size()>0;  }    **public** **synchronized** String buy(){  **try** {  **return** list.remove(0);  } **catch** (IndexOutOfBoundsException e) {  **throw** **new** RuntimeException("票卖超了");  }  }  } |
| **package** com.atguigu.part03;  **public** **class** Saler **extends** Thread{  **private** TicketService ts;  **public** Saler(TicketService ts) {  **super**();  **this**.ts = ts;  }  **public** **void** run(){  **while**(ts.hasTicket()){  **try** {  Thread.*sleep*(100);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  **try** {  String buy = ts.buy();  System.***out***.println("购买票：" + buy);  } **catch** (Exception e) {  System.***err***.println(e.getMessage());  }  }  System.***out***.println("没有票了");  }  } |
| **package** com.atguigu.part03;  **public** **class** TestSaler {  **public** **static** **void** main(String[] args) {  TicketService ts = **new** TicketService();  Saler s1 = **new** Saler(ts);  Saler s2 = **new** Saler(ts);    s1.start();  s2.start();  }  } |

不要对线程安全类的所有方法都加同步，只对那些会影响竞争资源（即共享资源）的方法进行同步即可。而且也要注意同步方法的默认同步的监视器对象对于多个线程来说是否是同一个。

### 14.5.4 线程安全的集合

这里如果把ArrayList换成Vector，发现也可以解决问题。

|  |
| --- |
| **package** com.atguigu.part03;  **import** java.util.Vector;  **public** **class** TicketService {  **private** Vector<String> list;  **public** TicketService(){  list = **new** Vector<String>();  list.add("01车01A");  list.add("01车01B");  list.add("01车01C");  list.add("01车01D");  list.add("01车01E");  list.add("01车02A");  list.add("01车02B");  list.add("01车02C");  list.add("01车02D");  list.add("01车02E");  }  **public** **boolean** hasTicket(){  **return** list.size()>0;  }    **public** String buy(){  **try** {  **return** list.remove(0);  } **catch** (IndexOutOfBoundsException e) {  **throw** **new** RuntimeException("票卖超了");  }  }  } |
| **package** com.atguigu.part03;  **public** **class** Saler **extends** Thread{  **private** TicketService ts;  **public** Saler(TicketService ts) {  **super**();  **this**.ts = ts;  }  **public** **void** run(){  **while**(ts.hasTicket()){  **try** {  Thread.*sleep*(100);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  **try** {  String buy = ts.buy();  System.***out***.println("购买票：" + buy);  } **catch** (Exception e) {  System.***err***.println(e.getMessage());  }  }  System.***out***.println("没有票了");  }  } |
| **package** com.atguigu.part03;  **public** **class** TestSaler {  **public** **static** **void** main(String[] args) {  TicketService ts = **new** TicketService();  Saler s1 = **new** Saler(ts);  Saler s2 = **new** Saler(ts);    s1.start();  s2.start();  }  } |

因为Vector是线程安全的集合，打开Vector类的源代码查看，它的很多方法都是同步发方法，即有synchronized修饰。

### 14.5.5 释放同步监视器的锁定

任何线程进入同步代码块、同步方法之前，必须先获得对同步监视器的锁定，那么何时会释放对同步监视器的锁定呢？

#### 1、释放锁的操作

* 当前线程的同步方法、同步代码块执行结束。
* 当前线程在同步代码块、同步方法中遇到break、return终止了该代码块、该方法的继续执行。
* 当前线程在同步代码块、同步方法中出现了未处理的Error或Exception，导致当前线程异常结束。
* 当前线程在同步代码块、同步方法中执行了锁对象的wait()方法，当前线程被挂起，并释放锁。

#### 2、不会释放锁的操作

* 线程执行同步代码块或同步方法时，程序调用Thread.sleep()、Thread.yield()方法暂停当前线程的执行。
* 线程执行同步代码块时，其他线程调用了该线程的suspend()方法将该该线程挂起，该线程不会释放锁（同步监视器）。
  + 应尽量避免使用suspend()和resume()这样的过时来控制线程

### 14.5.6 死锁

不同的线程分别锁住对方需要的同步监视器对象不释放，都在等待对方先放弃时就形成了线程的死锁。一旦出现死锁，整个程序既不会发生异常，也不会给出任何提示，只是所有线程处于阻塞状态，无法继续。

|  |
| --- |
| **package** com.atguigu.part03;  **public** **class** TestDeadLock {  **public** **static** **void** main(String[] args) {  Object g = **new** Object();  Object m = **new** Object();  Owner s = **new** Owner(g,m);  Customer c = **new** Customer(g,m);  **new** Thread(s).start();  **new** Thread(c).start();  }  }  **class** Owner **implements** Runnable{  **private** Object goods;  **private** Object money;  **public** Owner(Object goods, Object money) {  **super**();  **this**.goods = goods;  **this**.money = money;  }  @Override  **public** **void** run() {  **synchronized** (goods) {  System.***out***.println("先给钱");  **synchronized** (money) {  System.***out***.println("发货");  }  }  }  }  **class** Customer **implements** Runnable{  **private** Object goods;  **private** Object money;  **public** Customer(Object goods, Object money) {  **super**();  **this**.goods = goods;  **this**.money = money;  }  @Override  **public** **void** run() {  **synchronized** (money) {  System.***out***.println("先发货");  **synchronized** (goods) {  System.***out***.println("再给钱");  }  }  }  } |

## 14.6 线程通信

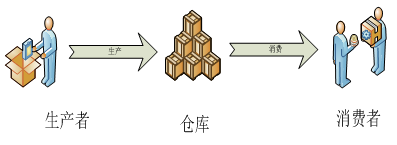
生产者与消费者问题（英语：Producer-consumer problem），也称有限缓冲问题（英语：Bounded-buffer problem），是一个多线程同步问题的经典案例。

该问题描述了两个（多个）共享固定大小缓冲区的线程——即所谓的“生产者”和“消费者”——在实际运行时会发生的问题。生产者的主要作用是生成一定量的数据放到缓冲区中，然后重复此过程。与此同时，消费者也在缓冲区消耗这些数据。该问题的关键就是要保证生产者不会在缓冲区满时加入数据，消费者也不会在缓冲区中空时消耗数据。

生产者与消费者问题中其实隐含了两个问题：

* 线程安全问题：因为生产者与消费者共享数据缓冲区，不过这个问题可以使用同步解决。
* 线程的协调工作问题：

要解决该问题，就必须让生产者线程在缓冲区满时等待(wait)，暂停进入阻塞状态，等到下次消费者消耗了缓冲区中的数据的时候，通知(notify)正在等待的线程恢复到就绪状态，重新开始往缓冲区添加数据。同样，也可以让消费者线程在缓冲区空时进入等待(wait)，暂停进入阻塞状态，等到生产者往缓冲区添加数据之后，再通知(notify)正在等待的线程恢复到就绪状态。通过这样的通信机制来解决此类问题。



Object类中提供了wait()、notify()、notifyAll()方法，这三个方法并不属于Thread类，那是因为这三个方法必须有同步监视器对象来调用，而同步监视器对象可以是任意类型的对象，因此它们只能声明在Object类中。

### 14.6.1 案例一：快餐店的厨师、服务员、取餐台

#### 1、一个生产者、一个消费者

案例：有加餐馆的取餐口比较小，只能放10份快餐，厨师做完快餐放在取餐口的工作台上，服务器从这个工作台取出快餐给顾客。现在有一个厨师和一个服务员。

|  |
| --- |
| **package** com.atguigu.part04;  **public** **class** Workbench {  **private** **static** **final** **int** ***MAX\_VALUE*** = 10;  **private** **int** num;  **public** **synchronized** **void** put() {  **if**(num >= ***MAX\_VALUE***){  **try** {  **this**.wait();  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  **try** {  Thread.*sleep*(100);  //加入睡眠时间是放大问题现象，去掉同步和wait等，可观察问题  } **catch** (InterruptedException e) {  e.printStackTrace();  }  num++;  System.***out***.println("厨师制作了一份快餐，现在工作台上有：" + num + "份快餐");  **this**.notify();  }  **public** **synchronized** **void** take() {  **if**(num <=0){  **try** {  **this**.wait();  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  **try** {  Thread.*sleep*(100);  //加入睡眠时间是放大问题现象，去掉同步和wait等，可观察问题  } **catch** (InterruptedException e) {  e.printStackTrace();  }  num--;  System.***out***.println("服务员取走了一份快餐，现在工作台上有：" + num + "份快餐");  **this**.notify();  }  } |
| **package** com.atguigu.part04;  **public** **class** Cook **extends** Thread{  **private** Workbench workbench;    **public** Cook(Workbench workbench) {  **super**();  **this**.workbench = workbench;  }  **public** **void** run(){  **for** (**int** i = 1; i <= 10; i++) {  workbench.put();  }  }  } |
| **package** com.atguigu.part04;  **public** **class** Waiter **extends** Thread{  **private** Workbench workbench;    **public** Waiter(Workbench workbench) {  **super**();  **this**.workbench = workbench;  }  **public** **void** run(){  **for** (**int** i = 1; i <= 10; i++) {  workbench.take();  }  }  } |
| **package** com.atguigu.part04;  **public** **class** TestOneAndOne {  **public** **static** **void** main(String[] args) {  Workbench bench = **new** Workbench();  Cook c = **new** Cook(bench);  Waiter w = **new** Waiter(bench);    c.start();  w.start();  }  } |

#### 2、多个生产者与多个消费者

案例：有加餐馆的取餐口比较小，只能放10份快餐，厨师做完快餐放在取餐口的工作台上，服务器从这个工作台取出快餐给顾客。现在有多个厨师和多个服务员。

|  |
| --- |
| **package** com.atguigu.part04;  **public** **class** Workbench {  **private** **static** **final** **int** ***MAX\_VALUE*** = 10;  **private** **int** num;  **public** **synchronized** **void** put() {  **while**(num >= ***MAX\_VALUE***){  **try** {  **this**.wait();  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  **try** {  Thread.*sleep*(100);  //加入睡眠时间是放大问题现象，去掉同步和wait等，可观察问题  } **catch** (InterruptedException e) {  e.printStackTrace();  }  num++;  System.***out***.println("厨师制作了一份快餐，现在工作台上有：" + num + "份快餐");  **this**.notifyAll();  }  **public** **synchronized** **void** take() {  **while**(num <=0){  **try** {  **this**.wait();  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  **try** {  Thread.*sleep*(100);  //加入睡眠时间是放大问题现象，去掉同步和wait等，可观察问题  } **catch** (InterruptedException e) {  e.printStackTrace();  }  num--;  System.***out***.println("服务员取走了一份快餐，现在工作台上有：" + num + "份快餐");  **this**.notifyAll();  } |
| 厨师和服务员线程的代码不变 |
| **package** com.atguigu.part04;  **public** **class** TestManyAndMany {  **public** **static** **void** main(String[] args) {  Workbench bench = **new** Workbench();  Cook c1 = **new** Cook(bench);  Cook c2 = **new** Cook(bench);  Waiter w1 = **new** Waiter(bench);  Waiter w2 = **new** Waiter(bench);    c1.start();  c2.start();  w1.start();  w2.start();  }  } |

### 14.6.2 案例二：交替打印数字

案例：两个线程交替打印1-100整数，一个打印奇数，一个打印偶数，要求输出结果有序。如下是示例代码：

|  |
| --- |
| **package** com.atguigu.number;  **public** **class** TestPrintNumber {  **public** **static** **void** main(String[] args) {  PrintNumber p1 = **new** PrintNumber();  p1.start();    PrintNumber p2 = **new** PrintNumber();  p2.start();  }  }  **class** PrintNumber **extends** Thread{  **private** **static** **int** *num*;  **public** **void** run(){  **while**(**true**){  **synchronized** (PrintNumber.**class**) {  **try** {  Thread.*sleep*(1000);  PrintNumber.**class**.notify();  System.***out***.println(getName() + ":" + ++*num*);  PrintNumber.**class**.wait();  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  }  }  } |

## 14.7 单例设计模式

单例设计模式，是软件开发中最常用的设计模式之一，它是指某个类在整个系统中只能有一个实例对象可被获取和使用的代码模式。例如：代表JVM运行环境的Runtime类。

通常有饿汉式和懒汉式两种。

### 14.7.1 饿汉式单例设计模式

所谓饿汉式，就是在类初始化时，直接创建对象。

优势：因为Java的类加载和初始化的机制可以保证线程安全，所以这类形式的单例设计模式不存在线程安全问题。

劣势：不管你暂时是否需要该实例对象，都会创建，使得类初始化时间加长。

#### 1、直接实例化饿汉式（简洁直观）

|  |
| --- |
| **package** com.atguigu.part05;  **public** **class** Singleton1 {  **public** **static** **final** Singleton1 ***INSTANCE*** = **new** Singleton1();  **private** Singleton1(){    }  } |

#### 2、枚举式（最简洁）

|  |
| --- |
| **package** com.atguigu.part05;  **public** **enum** Singleton2 {  ***INSTANCE***  } |

#### 3、静态代码块饿汉式（适合复杂实例化）

|  |
| --- |
| #key=value  info=atguigu |
| **package** com.atguigu.part05;  **import** java.io.IOException;  **import** java.util.Properties;  **public** **class** Singleton3 {  **public** **static** **final** Singleton3 ***INSTANCE***;  **private** String info;    **static**{  **try** {  Properties pro = **new** Properties();    pro.load(Singleton3.**class**.getClassLoader().getResourceAsStream("single.properties"));    ***INSTANCE*** = **new** Singleton3(pro.getProperty("info"));  } **catch** (IOException e) {  **throw** **new** RuntimeException(e);  }  }    **private** Singleton3(String info){  **this**.info = info;  }  **public** String getInfo() {  **return** info;  }  **public** **void** setInfo(String info) {  **this**.info = info;  }  } |

### 14.7.2 懒汉式单例设计模式

所谓懒汉式：即延迟创建对象，直到用户来获取这个对象时，再创建。

优势：不用不创建，用时再创建

劣势：有线程安全问题

#### 1、有线程安全问题的形式

线程不安全（适用于单线程）

|  |
| --- |
| **package** com.atguigu.part05;  **public** **class** Singleton4 {  **private** **static** Singleton4 *instance*;  **private** Singleton4(){    }  **public** **static** Singleton4 getInstance(){  **if**(*instance* == **null**){    **try** {  Thread.*sleep*(100);  } **catch** (InterruptedException e) {  e.printStackTrace();  }    *instance* = **new** Singleton4();  }  **return** *instance*;  }  } |
| **package** com.atguigu.part05;  **public** **class** TestSingleton4 {  **private** **static** Singleton4 *instance1*;  **private** **static** Singleton4 *instance2*;  **public** **static** **void** main(String[] args) {    Thread t1 = **new** Thread(){  **public** **void** run(){  *instance1* = Singleton4.*getInstance*();  }  };  t1.start();    Thread t2 = **new** Thread(){  **public** **void** run(){  *instance2* = Singleton4.*getInstance*();  }  };  t2.start();    **try** {  t1.join();  t2.join();  } **catch** (InterruptedException e) {  e.printStackTrace();  }  System.***out***.println(*instance1*);  System.***out***.println(*instance2*);  System.***out***.println(*instance1* == *instance2*);  }  } |

#### 2、无线程安全问题的形式（适用于多线程）

|  |
| --- |
| **package** com.atguigu.part05;  **public** **class** Singleton5 {  **private** **static** Singleton5 *instance*;  **private** Singleton5(){    }  **public** **static** Singleton5 getInstance(){  **if**(*instance* == **null**){  **synchronized** (Singleton5.**class**) {  **if**(*instance* == **null**){  **try** {  Thread.*sleep*(1000);  } **catch** (InterruptedException e) {  e.printStackTrace();  }    *instance* = **new** Singleton5();  }  }  }  **return** *instance*;  }  } |

#### 3、静态内部类形式（适用于多线程）

|  |
| --- |
| **package** com.atguigu.part05;  **public** **class** Singleton6 {  **private** Singleton6(){    }  **private** **static** **class** Inner{  **private** **static** **final** Singleton6 ***INSTANCE*** = **new** Singleton6();  }    **public** **static** Singleton6 getInstance(){  **return** Inner.***INSTANCE***;  }  } |

## 14.8 本章小结

本章主要介绍了Java的多线程编程支持。本章简要介绍了线程的基本概念，并讲解了线程和进程之间的区别和联系。本章详细讲解了如何创建、启动多线程，并对比了两种创建多线程之间的优势和劣势，也详细介绍了线程的生命周期。本章还通过示例程序示范了控制线程的几个方法，还详细讲解了线程同步的意义和必要性，并介绍了两种不同的线程同步的方式。另外也介绍了线程通信的方式。还对单例设计模式做了详细的讲解，使用了多种方式进行实现，这个可是在企业面试题中的一个高频面试题。

从JDK1.5之后还新增了java.util.concurrent包的新的API来支持新的多线程编程方式，我们将在多线程高级部分讲解，敬请期待。