## 进程和线程的概念

**进程：**查看“Windows任务管理器”中的列表，我们可以讲将这些可以运行在内存中exe文件理解为进程，进程是受操作系统管理的基本运行单元。

**线程：**在进程中独立运行的子任务，一个进程至少包含一个线程。比如：QQ.exe运行时就有很多子任务，在线视频线程，传输数据线程和听音乐线程等子任务。

**多线程的优点：**最大限度地利用CPU的空闲时间来处理其他的任务，降低响应时间，提高用户体验。

每个进程都有独立的代码和数据空间，进程间的切换会有较大的开销。同一类线程共享代码和数据空间，每个线程有独立的运行栈和程序计数器（PC），线程切换开销小。

## Thread类和runnable接口

实现多线程编程的方式主要有两种，一种是继承Thread类，另一种是实现Runnable接口。查看API可以发现 public class Thread implements runnable，Thread实现了Runnable接口，使用这两种来创建线程并没有本质区别，因为Java是单继承，所以为了支持多继承我们可以现实Runnable接口的方式，一边实现一边继承。例如，如果欲创建的线程类已经有一个父类了，这时就不能再继承Thread类了，所以我们可以使用实现Runnable接口来应对这样的情况。

示例：

**public** **class** MyThread **extends** Thread {

@Override

**public** **void** run() {

**super**.run();

System.***out***.println("MyThread");

}

}

**public** **class** Test {

**public** **static** **void** main(String[] args){

MyThread myThread = **new** MyThread();

myThread.start();

System.***out***.println("运行结束");

}

}

**运行结果：**

运行结束

MyThread

从运行结果我们可以发现，使用多线程技术时，代码的运行结果与代码执行顺序或调用顺序无关。另外start()方法的顺序也不代表线程启动的顺序。

## 线程的五种状态

**在Java当中，线程通常都有五种状态，创建（new）、就绪（start）、运行（run）、阻塞和死亡。**  
　　第一是创建状态。在生成线程对象，并没有调用该对象的start方法，这时线程处于创建状态。

　　第二是就绪状态。当调用了线程对象的start方法之后，该线程就进入了就绪状态，但是此时线程调度程序还没有把该线程设置为当前线程，此时处于就绪状态。Thread类中start()方法是通知“线程规划器”此线程已经准备就绪，等待调用线程对象的run()方法。这个过程是让系统安排一个时间来调用run()方法，方法run()称为线程体，它包含了要执行的这个线程的内容， Run方法运行结束， 此线程终止。

第三是运行状态。线程调度程序将处于就绪状态的线程设置为当前线程，此时线程就进入了运行状态，也就是开始运行run函数当中的代码。

　　第四是阻塞状态。线程正在运行的时候，被暂停，通常是为了等待某个时间的发生(比如说某项资源就绪)之后再继续运行。sleep , suspend，wait等方法都可以导致线程阻塞。

　　第五是死亡状态。如果一个线程的run方法执行结束或者调用stop方法后，该线程就会死亡。对于已经死亡的线程，无法再使用start方法令其进入就绪。

## start和run方法

**start()方法**用来启动线程，真正实现了多线程运行。这时无需等待其他线程对象run方法体代码执行完毕，可以直接继续执行下面的代码；通过调用Thread类的start()方法来启动一个线程， 这时此线程是处于就绪状态， 并没有运行。 然后通过此Thread类调用方法run()来完成其运行操作的， 这里方法run()称为线程体，它包含了要执行的这个线程的内容， Run方法运行结束， 此线程终止。然后CPU再调度其它线程。

**run()方法**当作普通方法的方式调用。如果调用Run方法就不是异步执行而是同步，那么子线程对象并不提交给线程规划器来处理，而是有主线程main来调用run方法，程序还是要顺序执行，要等待run方法体执行完毕后，才可继续执行下面的代码

## 线程安全和非线程安全

### 多线程共享数据对象带来的非线程安全问题：

**public** **class** MyThread **extends** Thread {

**private** **int** count = 5;

@Override

**public** **void** run() {

**super**.run();

count--;

System.***out***.println("由 "+**this**.*currentThread*().getName()+" 计算，count="+count);

}

}

**public** **class** Test {

**public** **static** **void** main(String[] args) {

MyThread mythread=**new** MyThread();

//多个线程共享mythread中的count变量

Thread a=**new** Thread(mythread,"A");

Thread b=**new** Thread(mythread,"B");

Thread c=**new** Thread(mythread,"C");

Thread d=**new** Thread(mythread,"D");

Thread e=**new** Thread(mythread,"E");

a.start();

b.start();

c.start();

d.start();

e.start();

}

}

**可能出现运行结果：**

由 A 计算，count=4

由 E 计算，count=3

由 B 计算，count=3

由 C 计算，count=2

由 D 计算，count=1

从运行结果我们知道线程E和B同时对count进行处理，产生了“非线程安全”问题，而我们想要的依次递减，结果不重复。在某些JVM中，i--的操作要分成如下3个步骤：

1. 取得原有i的值
2. 计算i-1
3. 对i进行赋值

在这个3个步骤中，**如果有多个线程同时访问，那么一定会出现非线程安全问题。非线程安全主要是指多个线程对同一个对象中的同一个实例变量进行操作是会出现值被更改、值不同步的情况，进而影响程序的执行流程。**

### 使用synchronized关键字处理非线程安全问题

对上面的MyThread 类的run方法前添加**synchronized** 关键字来处理非线程安全问题

**public** **class** MyThread **extends** Thread {

**private** **int** count=5;

@Override

**synchronized public** **void** run() {

**super**.run();

count--;

System.***out***.println("由 "+**this**.*currentThread*().getName()+" 计算，count="+count);

}

}

运行结果：

由 A 计算，count=4

由 E 计算，count=3

由 B 计算，count=2

由 C 计算，count=1

由 D 计算，count=0

## Synchronized详解

通过在run方法前加入synchronized关键字，使多个线程在执行run方法时，以排队的方式进行处理。当一个线程调用run前，先判断run方法有没有被上锁，如果上锁，说明有其他线程正在调用run方法，必须等其他线程对run方法调用结束后才可以开始执行run方法。这样就实现了排队调用run方法的目的。Synchronized可以在任意对象及方法上加锁，而加锁的这段代码称为“互斥区”或“临界区”。当一个线程想要执行同步方法里面的代码时，线程首先尝试去拿掉这把锁，如果能拿掉，那么这个线程可以执行synchronized里的代码，如果不能拿掉，就一直尝试着拿掉，直到能拿掉为止，而且是有多个线程同时去争抢这把锁。

### ①synchronized取得的锁都是对象锁

另外，关键字synchronized取得的锁都是对象锁，如果两个线程分别执行不同实例对象的run方法，那么这两个线程就无需去排队等待，因为这样会创建两个锁，分别对两个实例对象的run方法进行锁定。只有在“共享资源”的读写访问才需要同步化，如果不是共享资源就没有同步的必要。

线程锁的是对象实例，而不是针对实例方法，且看如下示例：

**public** **class** MyObject {

**synchronized** **public** **void** methodA() {

**try** {

System.***out***.println("begin methodA threadName=" + Thread.*currentThread*().getName());

Thread.*sleep*(5000);

System.***out***.println("end endTime=" + System.*currentTimeMillis*());

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

**synchronized** **public** **void** methodB() {

**try** {

System.***out***.println("begin methodB threadName=" + Thread.*currentThread*().getName() + " begin time=" + System.*currentTimeMillis*());

Thread.*sleep*(5000);

System.***out***.println("end");

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

**public** **class** ThreadA **extends** Thread {

**private** MyObject object;

**public** ThreadA(MyObject object) {

**super**();

**this**.object = object;

}

@Override

**public** **void** run() {

**super**.run();

object.methodA();

}

}

**public** **class** ThreadB **extends** Thread {

**private** MyObject object;

**public** ThreadB(MyObject object) {

**super**();

**this**.object = object;

}

@Override

**public** **void** run() {

**super**.run();

object.methodB();

}

}

**public** **class** Test {

**public** **static** **void** main(String[] args) {

MyObject object = **new** MyObject();

ThreadA a = **new** ThreadA(object);

a.setName("A");

ThreadB b = **new** ThreadB(object);

b.setName("B");

a.start();

b.start();

}

}

**运行结果：**

begin methodA threadName=A

end endTime=1467034365912

begin methodB threadName=B begin time=1467034365912

End

**MyObject类的方法methodA和methodB都带有synchronized关键字，所以即使线程A和线程B调用了同对象实例不同的方法，也需要等待先调用的methodA结束才能才开始methodB。所以说线程锁的是针对对象实例，而不是针对实例方法。当然如果MyObject的methodB没有加synchronized关键字，则线程B可以以异步的方式调用object对象中methodB（非synchronized类型的方法），则运行结果为：**

begin methodA threadName=A

begin methodB threadName=B begin time=1467035017829

end

end endTime=1467035022829

### ②Synchronized锁重入

当一个线程得到一个对象锁后，如果synchronized方法（或块）的内部调用了本类的其他synchronized方法（或块），那么该线程可以一直拥有该对象的锁。从另一方面解释，如果一个线程已经获得了该对象的锁，并且还没有释放，当想要再次获取这个对象的锁的时候还是可以获取的，如果不可获取就会造成死锁，这就是“可重入锁”的概念。

可重入锁也支持父子类继承的环境，子类可以通过“可重入锁”调用父类的同步方法，就像调用自己同步方法一样。

### ③出现异常时，锁会被自动释放

### ④同步不具有继承性

如果父类的methodA带有synchronized关键字修饰，当有子类继承该父类时，子类的methodA不具有同步特性。同步是不能被继承的。

### ⑤synchronized同步代码块和synchronized方法

**应用背景：**用关键字synchronized 声明方法是有弊端的，比如A线程条用同步方法执行一个长时间的任务，那么B线程必须等待很长时间，这种情况可以使用synchronized 同步代码块来解决。

当一个线程访问object的一个synchronized 同步代码块时，另一个线程仍然可以访问该object对象中的非synchronized(this)同步代码块。在块中代码同步执行的，而不再块中的是异步执行。

**代码块间的同步性：**在使用同步代码块时需要注意的是，当一个线程访问object的一个synchronized同步代码块时，其他线程对象同一个object中所有其他synchronized(this)同步代码块的访问都将阻塞。

先来看下使用synchronized 方法的弊端的示例：

**public** **class** Task {

**private** String data;

**public** **synchronized** **void** doLongTimeTask() {

**try** {

System.***out***.println("begin task");

Thread.*sleep*(3000);

data = "长时间处理任务后从远程返回的值1 threadName=" + Thread.*currentThread*().getName();

System.***out***.println(data);

System.***out***.println("end task");

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

**public** **class** Test {

**public** **static** **long** *beginTime1*;

**public** **static** **long** *endTime1*;

**public** **static** **long** *beginTime2*;

**public** **static** **long** *endTime2*;

**public** **static** **void** main(String[] args) {

Test test = **new** Test();

Task task = **new** Task();

MyThread1 thread1 = test.**new** MyThread1(task);

thread1.start();

MyThread2 thread2 = test.**new** MyThread2(task);

thread2.start();

**try** {

Thread.*sleep*(10000);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

**long** beginTime = *beginTime1*;

**if** (*beginTime2* < *beginTime1*) {

beginTime = *beginTime2*;

}

**long** endTime = *endTime1*;

**if** (*endTime2* > *endTime1*) {

endTime = *endTime2*;

}

System.***out***.println("耗时：" + ((endTime - beginTime) / 1000));

}

//定义线程1

**class** MyThread1 **extends** Thread {

**private** Task task;

**public** MyThread1(Task task) {

**super**();

**this**.task = task;

}

@Override

**public** **void** run() {

**super**.run();

*beginTime1* = System.*currentTimeMillis*();

task.doLongTimeTask();

*endTime1* = System.*currentTimeMillis*();

}

}

//定义线程2

**class** MyThread2 **extends** Thread {

**private** Task task;

**public** MyThread2(Task task) {

**super**();

**this**.task = task;

}

@Override

**public** **void** run() {

**super**.run();

*beginTime2* = System.*currentTimeMillis*();

task.doLongTimeTask();

*endTime2* = System.*currentTimeMillis*();

}

}

}

**运行结果：**

begin task

长时间处理任务后从远程返回的值1 threadName=Thread-0

end task

begin task

长时间处理任务后从远程返回的值1 threadName=Thread-1

end task

耗时：6

**使用synchronized 语句块进行处理，将Task.java类修改如下：**

**public** **class** Task {

**private** String data;

**public** **void** doLongTimeTask() {

**try** {

System.***out***.println("begin task");

Thread.*sleep*(3000);

**synchronized** (**this**) {

data = "长时间处理任务后从远程返回的值1 threadName=" + Thread.*currentThread*().getName();

}

System.***out***.println(data);

System.***out***.println("end task");

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

**运行结果：**

begin task

begin task

长时间处理任务后从远程返回的值1 threadName=Thread-1

end task

长时间处理任务后从远程返回的值1 threadName=Thread-0

end task

耗时：3

### ⑥synchronized(非this对象x)同步代码块

当同步代码块持有同一个“对象监视器”的情况下，同一时间只有一个线程可以执行synchronized(x)同步代码块中的代码。

锁x对象的好处：如果一个类中又很多synchronized方法，这是虽然能实现同步，但会受到阻塞，所以影响运行效率；但如果使用同步代码块锁x对象，则synchronized(x)代码块中的程序与同步方法是异步的，不与其他锁this同步方法争抢this锁，则可以大大提高效率。

### ⑦静态同步synchronized方法与synchronized(class)代码块

关键字synchronized加到static静态方法上是给Class类上锁，而synchronized关键字加到非static静态方法上是给对象上锁。同步synchronized(class)代码块的作用其实和synchronized static 方法的作用一样。

### ⑧同步代码块一般不适用String作为锁对象

在大多数的情况下，同步代码块都不使用String作为锁对象，而改用其他，比如new Object()实例化一个Object对象。

将synchronized(string)同步块与String联合使用时，要注意常量池所带来的一些例外，下面例子是使用String作为锁对象带来的问题：

**public** **class** Service {

**public** **static** **void** print(String stringParam) {

**try** {

**synchronized** (stringParam) {

**while** (**true**) {

System.***out***.println(Thread.*currentThread*().getName());

Thread.*sleep*(1000);

}

}

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

**public** **class** ThreadA **extends** Thread {

**private** Service service;

**public** ThreadA(Service service) {

**super**();

**this**.service = service;

}

@Override

**public** **void** run() {

service.print("AA");

}

}

**public** **class** ThreadB **extends** Thread {

**private** Service service;

**public** ThreadB(Service service) {

**super**();

**this**.service = service;

}

@Override

**public** **void** run() {

service.*print*("AA");

}

}

**public** **class** Test {

**public** **static** **void** main(String[] args) {

Service service = **new** Service();

ThreadA a = **new** ThreadA(service);

a.setName("A");

a.start();

ThreadB b = **new** ThreadB(service);

b.setName("B");

b.start();

}

}

**运行结果：**

A

A

A

A

A

A

出现这样的情况就是因为String的两个值都是AA，两个线程持有相同的锁，所以造成线程B不能执行，这就是String常量池锁带来的问题，因为在JVM中具有String常量池缓存的功能，如下：

String a=”a”;

String b=”b”;

System.out.println(a == b);

输出的值为true。

## Volatile关键字

### volatile可见性特性

关键字volatile 的主要作用是使变量在多个线程间可见，它是强制从公共堆栈中取得变量的值，而不是从线程私有数据栈中取得变量值，看如下例子：

**public** **class** RunThread **extends** Thread {

**volatile private** **boolean** isRunning = **true**;

**public** **boolean** isRunning() {

**return** isRunning;

}

**public** **void** setRunning(**boolean** isRunning) {

**this**.isRunning = isRunning;

}

@Override

**public** **void** run() {

System.***out***.println("进入run了");

**while** (isRunning == **true**) {

}

System.***out***.println("线程被停止了！");

}

}

**public** **class** Run {

**public** **static** **void** main(String[] args) {

**try** {

RunThread thread = **new** RunThread();

thread.start();

Thread.*sleep*(1000);

thread.setRunning(**false**);

System.***out***.println("已经赋值为false");

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

**在使用Eclipse开发环境运行后的结果如下：**

进入run了

已经赋值为false

线程被停止了！

**但是运行在JVM设置为server服务器的环境中（配置Eclipse中JVM的运行参数为-server），运行后出现了死循环，运行结果如下：**

进入run了

已经赋值为false

在启动RunTread.java线程时，变量private boolean isRunning=true;存在于公共堆栈和线程的私有堆栈中。在JVM被设置为-server模式时为了线程运行效率，线程一直在私有堆栈中取得isRunning的值是true。而代码thread.setRunning(false);虽然被执行，更新的缺少公共堆栈中的isRunning变量的值false，所以一直出现死循环。

这个问题其实就是私有堆栈中的值和公共堆栈中的值不同步造成的。解决这样的问题就要在isRunning变量使用volatile 关键字了，它的主要作用就当线程访问isRuning这个变量时，强制从公共堆栈中进行取值。

1. 关键字volatile是线程同步的轻量级实现，所以volatile性能比synchronized要好
2. Volatile 只能修饰变量
3. 多线程访问volatile不会发生阻塞，而synchronized会出现阻塞
4. Volatile能保证数据的可见性，但不能保证原子性，它是解决多个线程之间的可见性；synchronized是解决多个线程之间的访问资源的同步性。

线程安全包含原子性和可见性两个方面，Java的同步机制都是围绕这两个方法来确保线程安全的。

### volatile非原子的特性

**public** **class** **MyThread extends** Thread {

**volatile** **public** **static** **int** *count*;

//如果在这边addCount方法前声明为synchronized，就可以使count达到同步效果，同时也就没必要使用volatile关键字了

**private** **static** **void** addCount() {

**for** (**int** i = 0; i < 100; i++) {

*count*++;

}

System.***out***.println("count=" + *count*);

}

@Override

**public** **void** run() {

*addCount*();

}

}

**public** **class** **Run** {

**public** **static** **void** main(String[] args) {

MyThread[] mythreadArray = **new** MyThread[100];

**for** (**int** i = 0; i < 100; i++) {

mythreadArray[i] = **new** MyThread();

}

**for** (**int** i = 0; i < 100; i++) {

mythreadArray[i].start();

}

}

}

**运行结果:**

count=8743

count=9043

count=9143

count=9243

count=9343

count=9443

count=9743

count=9743

count=9543

count=9843

count=9943

关键字volatile每次从共享内存中读取变量,而不是从私有内存中读取,这样就保证了同步数据的可见性。**但在这里需要注意的是：如果修改实例变量中的数据，比如i++，也就是i=i+1，这样不是一个原子操作，就无法达到同步效果，也就是非线程安全。如果仅仅只是简单复制操作（即原子操作），就是线程安全的。**

## Thread类的几个方法介绍

static [Thread](C:/Users/Administrator/Desktop/html/zh_CN/api/java/lang/../../java/lang/Thread.html" \o "java.lang 中的类) currentThread()：

返回代码段正在被哪个线程调用的引用。

Boolean isAlive()：

测试当前线程是否处于活动状态。活动状态是指线程已经启动且尚未终止。线程处于正在运行或准备开始运行的状态，就认为线程是存活的。

static void [sleep](C:/Users/Administrator/Desktop/html/zh_CN/api/java/lang/../../java/lang/Thread.html" \l "sleep(long))(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，这个“正在执行的线程”是指this.currentThead()返回的线程。

## 停止（中断）线程和暂停线程

## 线程中断

### 什么时候需要用到线程中断

一个线程在未正常结束之前, 被强制终止是很危险的事情. 因为它可能带来完全预料不到的严重后果. 所以你看到suspend, stop等方法都被Deprecated了.  
 那么不能直接把一个线程搞挂掉, 但有时候又有必要让一个线程死掉, 或者让它结束某种等待的状态 该怎么办呢? 优雅的方法就是, 给那个线程一个中断信号, 让它自己决定该怎么办. 比如说, 在某个子线程中为了等待一些特定条件的到来, 你调用了Thread.sleep(10000), 预期线程睡10秒之后自己醒来, 但是如果这个特定条件提前到来的话, 你怎么通知一个在睡觉的线程呢? 又比如说, 主线程通过调用子线程的join方法阻塞自己以等待子线程结束, 但是子线程运行过程中发现自己没办法在短时间内结束, 于是它需要想办法告诉主线程别等我了. 这些情况下, 就需要中断.

### 为什么不推荐使用stop方法中断线程

使用stop()方法停止线程是非常暴力的，且看如下例子：

**public** **class** Test **implements** Runnable{

**public** **void** run() {

**int** count = 0;

**while** ( count <20 ) {

System.***out***.println("Running ... count=" + count);

count++;

**try** {

Thread.*sleep*(300);

} **catch** ( InterruptedException x ) { }

}

System.***out***.println("stoped");

}

**public** **static** **void** main(String[] args) **throws** InterruptedException {

Test test = **new** Test();

Thread t = **new** Thread(test);

t.start();

**try** {

Thread.*sleep*(2000);

} **catch** ( InterruptedException x ) { }

t.~~stop~~();

}

}

可能的运行结果：

Running ... count=0

Running ... count=1

Running ... count=2

Running ... count=3

Running ... count=4

Running ... count=5

Running ... count=6

可以发现程序中的打印stoped并没有执行，所以说如果在程序中有其他操作，如果线程突然stop是会带来严重的影响的。当然如果是我上面的程序代码突然stop的影响其实是没有的，但是如果是其他打开文件最后需要释放或者什么的就会带来严重的影响了。stop()方法已经作废，我们不推荐使用，因为如果强行停止线程则有可能使一些清理性的工作得不到完成。另外一个情况就对锁定的对象进行了“解锁”，导致数据得不到同步的处理，出现数据不一致的问题。

### interrupt()，interrupted()和isInterrupted()方法介绍

* **interrupt()方法：**该方法仅仅是在当前线程中打一个停止的标记，并不是真的停止线程。
* **interrupted()方法：**测试当前线程是否已经是中断状态，执行后具有将状态标志置为false（表示线程停止）的功能。
* **isInterrupted()方法：**进测试线程Thread对象是否已经是中断状态。不清除状态标志。

注意：这里的当前线程和线程对象，当前线程是指方法被调用执行时，正在运行的线程；线程对象是指调用了thread.isInterrupted()方法的实例对象thread。以上三个方法仅仅是判断或是打上一个停止的标记，没没有去停止线程执行的功能，但我们可以根据这个标志结合异常和return来中断线程。

### interrupt结合异常来中断线程

**public** **class** MyThread **extends** Thread {

@Override

**public** **void** run() {

**super**.run();

**for** (**int** i = 0; i < 500000; i++) {

**if** (**this**.*interrupted*()) {

System.***out***.println("已经是停止状态了!我要退出了!");

**break**;

}

System.***out***.println("i=" + (i + 1));

}

System.***out***.println("我被输出，如果此代码是for又继续运行，线程并未停止！");

}

}

**public** **class** Run {

**public** **static** **void** main(String[] args) {

**try** {

MyThread thread = **new** MyThread();

thread.start();

Thread.*sleep*(2000);

thread.interrupt();

} **catch** (InterruptedException e) {

System.***out***.println("main catch");

e.printStackTrace();

}

System.***out***.println("end!");

}

}

**运行结果：**

...

i=281203

i=281204

i=281205

i=281206

i=281207

已经是停止状态了!我要退出了!

我被输出，如果此代码是for又继续运行，线程并未停止！

end!

### interrupt结合return来中断线程

**public** **class** MyThread **extends** Thread {

@Override

**public** **void** run() {

**while** (**true**) {

**if** (**this**.isInterrupted()) {

System.***out***.println("停止了！");

**return**;

}

System.***out***.println("timer=" + System.*currentTimeMillis*());

}

}

}

**public** **class** Run {

**public** **static** **void** main(String[] args) **throws** InterruptedException {

MyThread t=**new** MyThread();

t.start();

Thread.*sleep*(2000);

t.interrupt();

}

}

**运行结果：**

**...**

timer=1475399311778

timer=1475399311778

timer=1475399311778

timer=1475399311778

timer=1475399311778

停止了！

## 暂停线程：suspend和resume

暂停线程意味着线程还可以恢复运行，在java多线程中，可以使用suspend()方法暂停线程，使用resume()方法恢复线程的执行。suspend()和resume()极易造成公共的同步对象的独占（就是线程暂停后，锁没有释放），使得其他线程无法访问公共同步对象。

### 锁独占

**public** **class** MyThread **extends** Thread {

**private** **long** i = 0;

@Override

**public** **void** run() {

**while** (**true**) {

i++;

System.***out***.println(i);

}

}

}

**public** **class** Run {

**public** **static** **void** main(String[] args) {

**try** {

MyThread thread = **new** MyThread();

thread.start();

Thread.*sleep*(1000);

thread.~~suspend~~();

System.***out***.println("main end!");

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

运行结果：

...

172405

172406

172407

172408

当程序运行到println()方法内部停止时，同步锁未被释放。方法println()源码如下：

Pubic void pringln(long x){

**synchronized** (this){

Print(x);

newLine();

}

}

### 不同步

使用suspend()和resume()方法容易出现因为线程的暂停而导致数据不同步的情况。

**public** **class** MyObject {

**private** String username = "1";

**private** String password = "11";

**public** **void** setValue(String u, String p) {

**this**.username = u;

**if** (Thread.*currentThread*().getName().equals("a")) {

System.***out***.println("停止a线程！");

Thread.*currentThread*().~~suspend~~();

}

**this**.password = p;

}

**public** **void** printUsernamePassword() {

System.***out***.println(username + " " + password);

}

}

**public** **class** Run {

**public** **static** **void** main(String[] args) **throws** InterruptedException {

**final** MyObject myobject = **new** MyObject();

Thread thread1 = **new** Thread() {

**public** **void** run() {

myobject.setValue("a", "aa");

};

};

thread1.setName("a");

thread1.start();

Thread.*sleep*(500);

Thread thread2 = **new** Thread() {

**public** **void** run() {

myobject.printUsernamePassword();

};

};

thread2.start();

}

}

**运行结果：**

停止a线程！

a 11

## yield()方法：放弃当前CPU资源

yield方法的作用是放弃当前的CPU资源，将它让给其他线程，但是放弃的时间不能确定，并且有可能一放弃有马上获取CPU时间片，且看如下实例：

**public** **class** MyThread **extends** Thread {

@Override

**public** **void** run() {

**long** beginTime = System.*currentTimeMillis*();

**int** count = 0;

**for** (**int** i = 0; i < 50000000; i++) {

Thread.*yield*();

count = count + (i + 1);

}

**long** endTime = System.*currentTimeMillis*();

System.***out***.println("用时：" + (endTime - beginTime) + "毫秒！");

}

}

**public** **class** Test {

**public** **static** **void** main(String[] args) {

MyThread thread = **new** MyThread();

thread.start();

}

}

**运行结果：**

用时：10600毫秒！

如果将Thread.yield()代码注释掉，用时将会大大减少。

## 线程的优先级

现在操作系统中，线程可以划分优先级，优先级较高的线程得到的CPU资源较多，也就是CPU优先执行优先级较高的线程任务。设置线程有助于帮“线程规划器”确定下一次优先选择哪个线程来执行。

使用setPriority方法设置优先级。在Java中，线程的优先级分为1~10，如果小于1或大于10，则抛出异常throw new IllegalArgumentException()。

线程的优先级具有3个特性：**继承性**、**规则性**和**随机性**。

### 继承性：

如果A线程启动B线程，则B线程的优先级与A是一样的。

**public** **class** MyThread1 **extends** Thread {

@Override

**public** **void** run() {

System.***out***.println("MyThread1 run priority=" + **this**.getPriority());

MyThread2 thread2 = **new** MyThread2();

thread2.start();

}

}

**public** **class** MyThread2 **extends** Thread {

@Override

**public** **void** run() {

System.***out***.println("MyThread2 run priority=" + **this**.getPriority());

}

}

**public** **class** Test {

**public** **static** **void** main(String[] args) {

System.***out***.println("main thread begin priority=" + Thread.*currentThread*().getPriority());

Thread.*currentThread*().setPriority(6);

System.***out***.println("main thread end priority=" + Thread.*currentThread*().getPriority());

MyThread1 thread1 = **new** MyThread1();

thread1.start();

}

}

**运行结果：**

main thread begin priority=5

main thread end priority=6

MyThread1 run priority=6

MyThread2 run priority=6

### 规则性：

优先级高的线程总是大部分先执行完，但不代表总是全部先执行完，另外线程的优先级与代码执行顺序无关，当两个线程优先级等级差距大时，谁先执行完和代码的调用顺序无关，线程规划器总是将CPU尽量让给优先级高的线程，这就是线程优先级的规则性。

**public** **class** MyThread1 **extends** Thread {

@Override

**public** **void** run() {

**long** beginTime = System.*currentTimeMillis*();

**long** addResult = 0;

**for** (**int** j = 0; j < 10; j++) {

**for** (**int** i = 0; i < 50000; i++) {

Random random = **new** Random();

random.nextInt();

addResult = addResult + i;

}

}

**long** endTime = System.*currentTimeMillis*();

System.***out***.println("★★★★★thread 1 use time=" + (endTime - beginTime));

}

}

**public** **class** MyThread2 **extends** Thread {

@Override

**public** **void** run() {

**long** beginTime = System.*currentTimeMillis*();

**long** addResult = 0;

**for** (**int** j = 0; j < 10; j++) {

**for** (**int** i = 0; i < 50000; i++) {

Random random = **new** Random();

random.nextInt();

addResult = addResult + i;

}

}

**long** endTime = System.*currentTimeMillis*();

System.***out***.println("☆☆☆☆☆thread 2 use time=" + (endTime - beginTime));

}

}

**public** **class** Test {

**public** **static** **void** main(String[] args) {

**for** (**int** i = 0; i < 5; i++) {

MyThread1 thread1 = **new** MyThread1();

thread1.setPriority(1);

thread1.start();

MyThread2 thread2 = **new** MyThread2();

thread2.setPriority(10);

thread2.start();

}

}

}

**可能的运行结果：**

☆☆☆☆☆thread 2 use time=190

☆☆☆☆☆thread 2 use time=250

☆☆☆☆☆thread 2 use time=260

★★★★★thread 1 use time=270

☆☆☆☆☆thread 2 use time=280

★★★★★thread 1 use time=300

☆☆☆☆☆thread 2 use time=300

★★★★★thread 1 use time=310

★★★★★thread 1 use time=310

★★★★★thread 1 use time=300

### 随机性：

优先级高线程不一定每次都先执行完，这就是线程的随机性。

规则性中的示例代码也可能出现如下的运行结果，**优先级较低线程1先执行完**。

☆☆☆☆☆thread 2 use time=190

☆☆☆☆☆thread 2 use time=240

☆☆☆☆☆thread 2 use time=270

☆☆☆☆☆thread 2 use time=310

★★★★★thread 1 use time=300

★★★★★thread 1 use time=290

★★★★★thread 1 use time=330

★★★★★thread 1 use time=340

★★★★★thread 1 use time=320

☆☆☆☆☆thread 2 use time=330

### 拓展：设置优先级比较谁比较快

**public** **class** ThreadA **extends** Thread {

**private** **int** count = 0;

**public** **int** getCount() {

**return** count;

}

@Override

**public** **void** run() {

**while** (**true**) {

count++;

}

}

}

**public** **class** ThreadB **extends** Thread {

**private** **int** count = 0;

**public** **int** getCount() {

**return** count;

}

@Override

**public** **void** run() {

**while** (**true**) {

count++;

}

}

}

**public** **class** Test {

**public** **static** **void** main(String[] args) {

**try** {

ThreadA a = **new** ThreadA();

a.setPriority(1);

a.start();

ThreadB b = **new** ThreadB();

b.setPriority(10);

b.start();

Thread.*sleep*(1000);

a.~~stop~~();

b.~~stop~~();

System.***out***.println("a=" + a.getCount());

System.***out***.println("b=" + b.getCount());

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

**可能运行结果：**

a=385238748

b=387317221

通常优先级高的数值越大，但是由于线程的随机性，不一定优先级高的总是越大。

## 什么是守护线程

在Java线程中有两种线程：非守护线程（用户线程）和守护线程。

守护线程：任何一个守护线程都是整个JVM中所有非守护线程的“保姆”，当JVM中存在任何一个非守护线程没有结束，守护线程就在工作，并且只有最后一个非守护线程结束了，守护线程才自动销毁。最典型的守护线程的应用就是GC（垃圾回收器）。

## join方法的使用

join()方法的作用是等待线程对象销毁，例如：

RunThread thread = **new** RunThread();

thread.start();

// Thread.sleep(?);

System.***out***.println("我想等thread对象执行完毕后我再执行，那么sleep方法中的？要填多少呢");

使用join 方法解决上面的需求：

RunThread thread = **new** RunThread();

thread.start();

thread.join();

System.***out***.println("thread对象执行完毕了");

方法join的作用是使所属的线程对象x正常执行run方法中的任务，而使用当前线程z进行无限期的阻塞，等待线程x销毁后再继续执行线程z后面的代码。

方法join具有使线程排队运行的作用，有些类似同步的运行效果。

**Join与synchronized的区别：**Join在内部使用wait方法进行等待，而synchronized关键字使用的是对象监视器原理作为同步。

如果当前线程对象被中断，则当前线程出现异常。

Join方法与interrupt方法遇到也会出现异常。

**Join(long)的使用：**

MyThread threadTest = **new** MyThread();

threadTest.start();

threadTest.join(2000);// 只等2秒

// Thread.sleep(2000);

System.***out***.println(" end timer=" + System.*currentTimeMillis*());

Join和sleep在上面示例中的运行效果并没有上面区别，主要区别还是来自于这2个方法对同步的处理上，方法join(long)的功能在内部是使用wait(long)方法来实现的，所以join(long)方法具有释放锁的特点，当执行wait(long)方法后，当前线程的锁被释放，那么其他线程就可以调用此线程中的同步方法了，而sleep(long)方法却不释放锁。

**Join方法的使用陷阱**

**public** **class** ThreadA **extends** Thread {

**private** ThreadB b;

**public** ThreadA(ThreadB b) {

**super**();

**this**.b = b;

}

@Override

**public** **void** run() {

**try** {

**synchronized** (b) {

System.***out***.println("begin A ThreadName=" + Thread.*currentThread*().getName() + " " + System.*currentTimeMillis*());

Thread.*sleep*(5000);

System.***out***.println(" end A ThreadName=" + Thread.*currentThread*().getName() + " " + System.*currentTimeMillis*());

}

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

**public** **class** ThreadB **extends** Thread {

@Override

**synchronized** **public** **void** run() {

**try** {

System.***out***.println("begin B ThreadName=" + Thread.*currentThread*().getName() + " " + System.*currentTimeMillis*());

Thread.*sleep*(5000);

System.***out***.println(" end B ThreadName=" + Thread.*currentThread*().getName() + " " + System.*currentTimeMillis*());

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

**public** **class** Run {

**public** **static** **void** main(String[] args) {

**try** {

ThreadB b = **new** ThreadB();

ThreadA a = **new** ThreadA(b);

a.start();

b.start();

b.join(2000);

System.***out***.println("main end " + System.*currentTimeMillis*());

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

下面对可能出现的运行结果进行解释：

**运行结果1：**

begin A ThreadName=Thread-1 1467426236576

end A ThreadName=Thread-1 1467426241576

begin B ThreadName=Thread-0 1467426241576

end B ThreadName=Thread-0 1467426246576

main end 1467426246576

1. b.join(2000)方法先抢到B锁，然后将B锁释放；
2. ThreadA抢到锁，打印begin A ThreadName=Thread-1 1467426236576，并sleep(5000);
3. ThreadA打印end A ThreadName=Thread-1 146742624157，并释放锁；
4. 这时join(2000)和ThreadB争抢锁，而join(2000)再次抢到锁，发现时间已过，释放锁后打印main end；
5. ThreadB抢到锁打印begin B ThreadName=Thread-0 1467426241576；
6. 5秒后再打印end B ThreadName=Thread-0 1467426246576。

**运行结果2：**

begin A ThreadName=Thread-1 1467426035396

end A ThreadName=Thread-1 1467426040396

main end 1467426040396

begin B ThreadName=Thread-0 1467426040397

end B ThreadName=Thread-0 1467426045398

1）b.join(2000)方法先抢到B锁，然后将B锁释放；

2）ThreadA抢到锁，打印begin A ThreadName=Thread-1 1467426035396，并sleep(5000);

3）ThreadA打印end A ThreadName=Thread-1 1467426040396，并释放锁；

4）这时join(2000)和ThreadB争抢锁，而join(2000)抢到锁后执行，sleep(5000)后释放锁；

5）main end在最后输出。

**运行结果3：**

begin A ThreadName=Thread-1 1429495462890

end A ThreadName=Thread-1 1429495467890

begin B ThreadName=Thread-0 1429495467890

main end 1429495467890

end B ThreadName=Thread-0 1429495472890

1. b.join(2000)方法先抢到B锁，然后将B锁释放；
2. ThreadA抢到锁，打印begin A ThreadName=Thread-1 1429495462890 并且sleep(5000)；
3. ThreadA打印end A ThreadName=Thread-1 1429495467890 并释放锁；
4. 这时join(2000)和ThreadB争抢锁，而join(2000)再次抢到锁，发现时间已过，释放锁；
5. B抢到锁打印begin B ThreadName=Thread-0 1429495467890，这时main end也异步输出；
6. 打印end B ThreadName=Thread-0 1429495472890。

## ThreadLocal的使用

变量值的共享可以public static 变量的形式，所有的线程都使用同一个public static变量。如果想实现每个线程都有自己的共享变量该如何解决呢？JDK中提供ThreadLocal来解决该问题。

ThreadLocal解决的是变量在不同线程间的隔离性，也就是不同线程拥有自己的值，不同线程中的值是可以放入Threadload类中进行保存的。

使用示例：

**public** **class** ThreadLocalExt **extends** ThreadLocal {

@Override

**protected** Object initialValue() {

**return** **new** Date().getTime();

}

}

**public** **class** Tools {

**public** **static** ThreadLocalExt *tl* = **new** ThreadLocalExt();

}

**public** **class** ThreadA **extends** Thread {

@Override

**public** **void** run() {

**try** {

**for** (**int** i = 0; i < 3; i++) {

System.out.println("在ThreadA线程中取值=" + Tools.tl.get());

Thread.sleep(100);

}

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

**public** **class** Run {

**public** **static** **void** main(String[] args) {

**try** {

**for** (**int** i = 0; i < 3; i++) {

System.***out***.println("在Main线程中取值=" + Tools.*tl*.get());

Thread.*sleep*(100);

}

Thread.*sleep*(5000);

ThreadA a = **new** ThreadA();

a.start();

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

**运行结果：**

在Main线程中取值=1467437749159

在Main线程中取值=1467437749159

在Main线程中取值=1467437749159

在ThreadA线程中取值=1467437754461

在ThreadA线程中取值=1467437754461

在ThreadA线程中取值=1467437754461

## InheritableThreadLocal的使用

使用InheritableThreadLocal类可以让子线程从父线程中取得值。

**public** **class** InheritableThreadLocalExt **extends** InheritableThreadLocal {

@Override

**protected** Object initialValue() {

**return** **new** Date().getTime();

}

}

**public** **class** Tools {

**public** **static** InheritableThreadLocalExt *tl* = **new** InheritableThreadLocalExt();

}

**public** **class** ThreadA **extends** Thread {

@Override

**public** **void** run() {

**try** {

**for** (**int** i = 0; i < 3; i++) {

System.***out***.println("在ThreadA线程中取值=" + Tools.*tl*.get());

Thread.*sleep*(100);

}

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

**public** **class** Run {

**public** **static** **void** main(String[] args) {

**try** {

**for** (**int** i = 0; i < 3; i++) {

System.***out***.println(" 在Main线程中取值=" + Tools.*tl*.get());

Thread.*sleep*(100);

}

Thread.*sleep*(5000);

ThreadA a = **new** ThreadA();

a.start();

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

**运行结果：**

在Main线程中取值=1467438453588

在Main线程中取值=1467438453588

在Main线程中取值=1467438453588

在ThreadA线程中取值=1467438453588

在ThreadA线程中取值=1467438453588

在ThreadA线程中取值=1467438453588

**重写childValue方法，可以修改子类继承的值，如：**

**public** **class** InheritableThreadLocalExt **extends** InheritableThreadLocal {

@Override

**protected** Object initialValue() {

**return** **new** Date().getTime();

}

@Override

**protected** Object childValue(Object parentValue) {

**return** parentValue + " 我在子线程加的~!";

}

}

**运行结果：**

在Main线程中取值=1467438828569

在Main线程中取值=1467438828569

在Main线程中取值=1467438828569

在ThreadA线程中取值=1467438828569 我在子线程加的~!

在ThreadA线程中取值=1467438828569 我在子线程加的~!

在ThreadA线程中取值=1467438828569 我在子线程加的~!

## 等待/通知机制：wait()和notify()方法

如果不使用等待/通知机制实现线程通信，需要用while语句轮询机制来检测某一个条件，这样会浪费CPU资源，如果轮询的时间间隔很小，更浪费CPU资源；如果轮询的时间间隔很大，有可能会取不到想要的数据。“wait/notify”机制可以实现减少CPU的资源浪费，而且可以实现多线程间的通信。

wait()方法的作用是使当前执行代码的线程进行等待，wait方法是Object的方法，该方法用来将当前线程植入“预执行队列”中，并且在wait所在的代码行处停止执行，直到接到通知或被中断为止。在调用wait之前，线程必须获得该对象级别锁，即只能在同步方法或同步块中条用wait方法。执行wait方法后，当前线程释放锁。当线程呈wait状态时，如果调用线程对象的interrupt()方法会出现InterruptedException异常。

notify方法用来通知那些可能等待该对象的对象锁的其他线程，如果有多个线程等待，则由线程规划器挑选一个呈现wait状态的线程，对其发出通知notify，并使它获取等待该对象的对象锁。在执行notify方法后，当前线程不会马上释放该对象锁，呈wait状态的线程也并不能马上获取该对象锁，要等到notify方法的线程将程序执行处synchronized代码块后，当前线程才会释放锁。当第一个获得该对象锁的wait线程运行完毕后，一会释放掉该对象锁，此时如果该对象没有再次使用notify语句，则即便该对象已经空闲，其他wait状态等待的线程由于没有得到该对象的通知，还会继续阻塞在wait状态，直到这个对象发出一个notify或notifyAll。

当调用wait和notify方法时，如果当前线程没有获取到该对象的对象级别锁，都会抛出IllegalMonitorStateException异常。

**wait(long)方法：**该方法的功能是等待某一时间内是否有线程对锁进行唤醒，如果超过这个时间则自动唤醒。

**notifyAll方法：**notify方法一次只唤醒一个线程，可以使用notifyAll方法唤醒全部线程。

## 生产者/消费者模式（未完成）

### 一生产与一消费：操作值

### 多生产与多消费：操作值-假死

### 多生产与多消费：假死解决方案

### 一生产与一消费：操作栈

### 一生产与多消费：操作栈

### 多生产与一消费：操作栈

### 多生产与多消费：操作栈

## 通过管道进行线程间通信

### 字节流

在Java中，管道流（pipeStream）是一种特殊的流，用于在不同线程间直接传送数据。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取数据。通过使用管道，实现不同线程间的通信，而无须借助于类似临时文件的东西。Java的JDK提供了4个类来使线程间可以进行通信：

1. PipedInputStream和PipedOutStream
2. PipedReader和PipedWriter

**public** **class** WriteData {

**public** **void** writeMethod(PipedOutputStream out) {

**try** {

System.***out***.println("write :");

**for** (**int** i = 0; i < 300; i++) {

String outData = "" + (i + 1);

out.write(outData.getBytes());

System.***out***.print(outData);

}

System.***out***.println();

out.close();

} **catch** (IOException e) {

e.printStackTrace();

}

}

}

**public** **class** ReadData {

**public** **void** readMethod(PipedInputStream input) {

**try** {

System.***out***.println("read :");

**byte**[] byteArray = **new** **byte**[20];

**int** readLength = input.read(byteArray);

**while** (readLength != -1) {

String newData = **new** String(byteArray, 0, readLength);

System.***out***.print(newData);

readLength = input.read(byteArray);

}

System.***out***.println();

input.close();

} **catch** (IOException e) {

e.printStackTrace();

}

}

}

**public** **class** ThreadWrite **extends** Thread {

**private** WriteData write;

**private** PipedOutputStream out;

**public** ThreadWrite(WriteData write, PipedOutputStream out) {

**super**();

**this**.write = write;

**this**.out = out;

}

@Override

**public** **void** run() {

write.writeMethod(out);

}

}

**public** **class** ThreadRead **extends** Thread {

**private** ReadData read;

**private** PipedInputStream input;

**public** ThreadRead(ReadData read, PipedInputStream input) {

**super**();

**this**.read = read;

**this**.input = input;

}

@Override

**public** **void** run() {

read.readMethod(input);

}

}

**public** **class** Run {

**public** **static** **void** main(String[] args) {

**try** {

WriteData writeData = **new** WriteData();

ReadData readData = **new** ReadData();

PipedInputStream inputStream = **new** PipedInputStream();

PipedOutputStream outputStream = **new** PipedOutputStream();

// inputStream.connect(outputStream);

outputStream.connect(inputStream);

ThreadRead threadRead = **new** ThreadRead(readData, inputStream);

threadRead.start();

Thread.*sleep*(2000);

ThreadWrite threadWrite = **new** ThreadWrite(writeData, outputStream);

threadWrite.start();

} **catch** (IOException e) {

e.printStackTrace();

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

**运行结果：**

read :

write :

123456789101112131415161718192021222324252627282930313233343536373839404142434445464748495051525354555657585960616263646566676869707172737475767778798081828384858687888990919293949596979899100101102103104105106107108109110111112113114115116117118119120121122123124125126127128129130131132133134135136137138139140141142143144145146147148149150151152153154155156157158159160161162163164165166167168169170171172173174175176177178179180181182183184185186187188189190191192193194195196197198199200201202203204205206207208209210211212213214215216217218219220221222223224225226227228229230231232233234235236237238239240241242243244245246247248249250251252253254255256257258259260261262263264265266267268269270271272273274275276277278279280281282283284285286287288289290291292293294295296297298299300

123456789101112131415161718192021222324252627282930313233343536373839404142434445464748495051525354555657585960616263646566676869707172737475767778798081828384858687888990919293949596979899100101102103104105106107108109110111112113114115116117118119120121122123124125126127128129130131132133134135136137138139140141142143144145146147148149150151152153154155156157158159160161162163164165166167168169170171172173174175176177178179180181182183184185186187188189190191192193194195196197198199200201202203204205206207208209210211212213214215216217218219220221222223224225226227228229230231232233234235236237238239240241242243244245246247248249250251252253254255256257258259260261262263264265266267268269270271272273274275276277278279280281282283284285286287288289290291292293294295296297298299300

从程序打印结果看，两个线程通过管道流成功进行数据的传输。首先读取线程new ThreadRead(inputStream)启动，由于当时没有数据被写入，所以线程阻塞在int readLength=in.read(byteArray);代码中，直到有数据被写入，才继续向下运行。

### 字符流

**public** **class** WriteData {

**public** **void** writeMethod(PipedWriter out) {

**try** {

System.***out***.println("write :");

**for** (**int** i = 0; i < 300; i++) {

String outData = "" + (i + 1);

out.write(outData);

System.***out***.print(outData);

}

System.***out***.println();

out.close();

} **catch** (IOException e) {

e.printStackTrace();

}

}

}

**public** **class** ReadData {

**public** **void** readMethod(PipedReader input) {

**try** {

System.***out***.println("read :");

**char**[] byteArray = **new** **char**[20];

**int** readLength = input.read(byteArray);

**while** (readLength != -1) {

String newData = **new** String(byteArray, 0, readLength);

System.***out***.print(newData);

readLength = input.read(byteArray);

}

System.***out***.println();

input.close();

} **catch** (IOException e) {

e.printStackTrace();

}

}

}

**public** **class** ThreadWrite **extends** Thread {

**private** WriteData write;

**private** PipedWriter out;

**public** ThreadWrite(WriteData write, PipedWriter out) {

**super**();

**this**.write = write;

**this**.out = out;

}

@Override

**public** **void** run() {

write.writeMethod(out);

}

}

**public** **class** ThreadRead **extends** Thread {

**private** ReadData read;

**private** PipedReader input;

**public** ThreadRead(ReadData read, PipedReader input) {

**super**();

**this**.read = read;

**this**.input = input;

}

@Override

**public** **void** run() {

read.readMethod(input);

}

}

**public** **class** Run {

**public** **static** **void** main(String[] args) {

**try** {

WriteData writeData = **new** WriteData();

ReadData readData = **new** ReadData();

PipedReader inputStream = **new** PipedReader();

PipedWriter outputStream = **new** PipedWriter();

// inputStream.connect(outputStream);

outputStream.connect(inputStream);

ThreadRead threadRead = **new** ThreadRead(readData, inputStream);

threadRead.start();

Thread.*sleep*(2000);

ThreadWrite threadWrite = **new** ThreadWrite(writeData, outputStream);

threadWrite.start();

} **catch** (IOException e) {

e.printStackTrace();

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

**运行结果：**

read :

write :

123456789101112131415161718192021222324252627282930313233343536373839404142434445464748495051525354555657585960616263646566676869707172737475767778798081828384858687888990919293949596979899100101102103104105106107108109110111112113114115116117118119120121122123124125126127128129130131132133134135136137138139140141142143144145146147148149150151152153154155156157158159160161162163164165166167168169170171172173174175176177178179180181182183184185186187188189190191192193194195196197198199200201202203204205206207208209210211212213214215216217218219220221222223224225226227228229230231232233234235236237238239240241242243244245246247248249250251252253254255256257258259260261262263264265266267268269270271272273274275276277278279280281282283284285286287288289290291292293294295296297298299300

123456789101112131415161718192021222324252627282930313233343536373839404142434445464748495051525354555657585960616263646566676869707172737475767778798081828384858687888990919293949596979899100101102103104105106107108109110111112113114115116117118119120121122123124125126127128129130131132133134135136137138139140141142143144145146147148149150151152153154155156157158159160161162163164165166167168169170171172173174175176177178179180181182183184185186187188189190191192193194195196197198199200201202203204205206207208209210211212213214215216217218219220221222223224225226227228229230231232233234235236237238239240241242243244245246247248249250251252253254255256257258259260261262263264265266267268269270271272273274275276277278279280281282283284285286287288289290291292293294295296297298299300

## 定时器：Timer

**Time类介绍：**

①一种工具，线程用其安排以后在后台线程中执行的任务。可安排任务执行一次，或者定期重复执行。

②与每个 Timer 对象相对应的是单个后台线程，用于顺序地执行所有计时器任务。此类不 提供实时保证：它使用 Object.wait(long) 方法来安排任务。

③计时器任务应该迅速完成。如果完成某个计时器任务的时间太长，那么它会“独占”计时器的任务执行线程。因此，这就可能延迟后续任务的执行，而这些任务就可能“堆在一起”。

④对 Timer 对象最后的引用完成后，并且 所有未处理的任务都已执行完成后，计时器的任务执行线程会正常终止（并且成为垃圾回收的对象）。但是这可能要很长时间后才发生。默认情况下，任务执行线程并不作为守护线程 来运行，所以它能够阻止应用程序终止。如果调用者想要快速终止计时器的任务执行线程，那么调用者应该调用计时器的 cancel 方法。如果意外终止了计时器的任务执行线程，例如调用了它的 stop 方法，那么所有以后对该计时器安排任务的尝试都将导致 IllegalStateException，就好像调用了计时器的 cancel 方法一样。

⑤此类是线程安全的：多个线程可以共享单个 Timer 对象而无需进行外部同步。

**Time方法介绍：**

**void** cancel() 终止此计时器，丢弃所有当前已安排的任务。

**int** purge() 从此计时器的任务队列中移除所有已取消的任务。

**void** schedule(TimerTask task, Date time) 安排在指定的时间执行指定的任务。

**void** schedule(TimerTask task, Date firstTime, **long** period) 安排指定的任务在指定的时间开始进行重复的固定延迟执行。

**void** schedule(TimerTask task, **long** delay) 安排在指定延迟后执行指定的任务。

**void** schedule(TimerTask task, **long** delay, **long** period) 安排指定的任务从指定的延迟后开始进行重复的固定延迟执行。

**void** scheduleAtFixedRate(TimerTask task, Date firstTime, **long** period) 安排指定的任务在指定的时间开始进行重复的固定速率执行。

**void** scheduleAtFixedRate(TimerTask task, **long** delay, **long** period) 安排指定的任务在指定的延迟后开始进行重复的固定速率执行。

### void schedule(TimerTask task, Date time) 安排在指定的时间执行指定的任务

**public** **class** Run2Later {

**private** **static** Timer *timer* = **new** Timer();

**static** **public** **class** MyTask1 **extends** TimerTask {

@Override

**public** **void** run() {

**try** {

System.***out***.println("1 begin 运行了！时间为：" + **new** Date());

Thread.*sleep*(20000);

System.***out***.println("1 end 运行了！时间为：" + **new** Date());

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

**static** **public** **class** MyTask2 **extends** TimerTask {

@Override

**public** **void** run() {

System.***out***.println("2 begin 运行了！时间为：" + **new** Date());

System.***out***.println("运行了！时间为：" + **new** Date());

System.***out***.println("2 end 运行了！时间为：" + **new** Date());

}

}

**public** **static** **void** main(String[] args) {

**try** {

MyTask1 task1 = **new** MyTask1();

MyTask2 task2 = **new** MyTask2();

SimpleDateFormat sdf1 = **new** SimpleDateFormat("yyyy-MM-dd HH:mm:ss");

SimpleDateFormat sdf2 = **new** SimpleDateFormat("yyyy-MM-dd HH:mm:ss");

String dateString1 = "2014-10-12 11:33:00";

String dateString2 = "2014-10-12 11:33:10";

Date dateRef1 = sdf1.parse(dateString1);

Date dateRef2 = sdf2.parse(dateString2);

System.***out***.println("字符串时间：" + dateRef1.~~toLocaleString~~() + "当前时间：" + **new** Date().~~toLocaleString~~());

System.***out***.println("字符串时间：" + dateRef2.~~toLocaleString~~() + "当前时间：" + **new** Date().~~toLocaleString~~());

*timer*.schedule(task1, dateRef1);

*timer*.schedule(task2, dateRef2);

} **catch** (ParseException e) {

e.printStackTrace();

}

}

}

**运行结果：**

字符串时间：2014-10-12 11:33:00当前时间：2016-7-10 13:25:59

字符串时间：2014-10-12 11:33:10当前时间：2016-7-10 13:25:59

1 begin 运行了！时间为：Sun Jul 10 13:25:59 CST 2016

1 end 运行了！时间为：Sun Jul 10 13:26:19 CST 2016

2 begin 运行了！时间为：Sun Jul 10 13:26:19 CST 2016

运行了！时间为：Sun Jul 10 13:26:19 CST 2016

2 end 运行了！时间为：Sun Jul 10 13:26:19 CST 2016

**说明：**

该程序执行完成后，进程并未结束，这是有创建的Time默认不是守护线程，所以它一直在运行不会终止，我们可以使用Timer *timer* = **new** Timer(**true**);该构造方法设置Timer是一个守护线程，这样程序执行完成后，进程也会被销毁。

当指定执行任务的时间晚于当前时间，则在指定的时间执行；

如果指定时间早于当前时间，则立即执行task任务；

Time中允许执行多个TimeTask任务，并且按顺序执行，如果前一个任务的执行时间很长，当前一个任务执行完成后的时间晚于该当前任务的计划时间时，则该当前任务立即执行，后面多任务以此类推。

### void schedule(TimerTask task, Date firstTime, long period) 安排指定的任务在指定的时间开始进行重复的固定延迟执行

**public** **class** Run2\_1 {

**static** **public** **class** MyTaskA **extends** TimerTask {

@Override

**public** **void** run() {

**try** {

System.***out***.println("A开始运行，时间为：" + **new** Date());

Thread.*sleep*(5000);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

**public** **static** **void** main(String[] args) {

**try** {

MyTaskA taskA = **new** MyTaskA();

SimpleDateFormat sdf = **new** SimpleDateFormat("yyyy-MM-dd HH:mm:ss");

String dateString = "2015-3-19 14:14:00";

Timer timer = **new** Timer();

Date dateRef = sdf.parse(dateString);

System.***out***.println("字符串时间：" + dateRef.~~toLocaleString~~() + "当前时间：" + **new** Date().~~toLocaleString~~());

timer.schedule(taskA, dateRef, 4000);

} **catch** (ParseException e) {

e.printStackTrace();

}

}

}

**运行时间：**

字符串时间：2015-3-19 14:14:00当前时间：2016-7-10 13:56:25

A开始运行，时间为：Sun Jul 10 13:56:25 CST 2016

A开始运行，时间为：Sun Jul 10 13:56:30 CST 2016

A开始运行，时间为：Sun Jul 10 13:56:35 CST 2016

A开始运行，时间为：Sun Jul 10 13:56:40 CST 2016

...

**说明：**程序正常的输出结果是每隔4秒执行一次任务，但是由于A任务的执行时间是5秒，所以执行的时间被延迟了。

### void schedule(TimerTask task, long delay, long period) 安排指定的任务从指定的延迟后开始进行重复的固定延迟执行

该方法的作用是以执行schedule该方法当前的时间为参考时间，在此时间基础上延迟指定的毫秒数，在以某一间隔时间无限次的执行某一任务。

### Schedule和scheduleAtFiexedRate方法的区别

这两个方法都会按顺序执行，所以不考虑非线程安全问题的情况。他们的主要区别在于不延迟的情况：

使用schedule方法，如果执行任务的时间没有被延时，那么下一次任务的执行时间参考的上一次任务的“开始”时间来计算，而scheduleAtFiexedRate是参考上一次任务的“结束”时间来计算。

### TimerTask和Timer类的cancel方法的区别

TimerTask类的cancel方法的作用是将自身从任务队列中清除，而Timer类中的cancel方法是将任务队列中的全部任务清空。且看如下两个示例：

**public** **class** Run2 {

**static** **public** **class** MyTaskA **extends** TimerTask {

@Override

**public** **void** run() {

System.***out***.println("A运行时间：" + **new** Date());

**this**.cancel();

}

}

**static** **public** **class** MyTaskB **extends** TimerTask {

@Override

**public** **void** run() {

System.***out***.println("B运行时间：" + **new** Date());

}

}

**public** **static** **void** main(String[] args) {

**try** {

MyTaskA taskA = **new** MyTaskA();

MyTaskB taskB = **new** MyTaskB();

SimpleDateFormat sdf = **new** SimpleDateFormat("yyyy-MM-dd HH:mm:ss");

String dateString = "2014-10-12 09:12:00";

Timer timer = **new** Timer();

Date dateRef = sdf.parse(dateString);

System.***out***.println("字符串时间：" + dateRef.~~toLocaleString~~() + "当前时间：" + **new** Date().~~toLocaleString~~());

timer.schedule(taskA, dateRef, 4000);

timer.schedule(taskB, dateRef, 4000);

} **catch** (ParseException e) {

e.printStackTrace();

}

}

}

**运行结果：**

字符串时间：2014-10-12 9:12:00当前时间：2016-7-10 14:12:22

A运行时间：Sun Jul 10 14:12:22 CST 2016

B运行时间：Sun Jul 10 14:12:22 CST 2016

B运行时间：Sun Jul 10 14:12:26 CST 2016

B运行时间：Sun Jul 10 14:12:30 CST 2016

...

**public** **class** Run3 {

**private** **static** Timer *timer* = **new** Timer();

**static** **public** **class** MyTaskA **extends** TimerTask {

@Override

**public** **void** run() {

System.***out***.println("A运行时间：" + **new** Date());

*timer*.cancel();

}

}

**static** **public** **class** MyTaskB **extends** TimerTask {

@Override

**public** **void** run() {

System.***out***.println("B运行时间：" + **new** Date());

}

}

**public** **static** **void** main(String[] args) {

**try** {

MyTaskA taskA = **new** MyTaskA();

MyTaskB taskB = **new** MyTaskB();

SimpleDateFormat sdf = **new** SimpleDateFormat("yyyy-MM-dd HH:mm:ss");

String dateString = "2014-10-12 09:12:00";

Date dateRef = sdf.parse(dateString);

System.***out***.println("字符串时间：" + dateRef.~~toLocaleString~~() + "当前时间：" + **new** Date().~~toLocaleString~~());

*timer*.schedule(taskA, dateRef, 4000);

*timer*.schedule(taskB, dateRef, 4000);

} **catch** (ParseException e) {

e.printStackTrace();

}

}

}

**运行结果：**

字符串时间：2014-10-12 9:12:00当前时间：2016-7-10 14:15:34

A运行时间：Sun Jul 10 14:15:34 CST 2016

## 单例模式与多线程

### 立即加载（饿汉模式）与延迟加载（懒汉模式）的概念

立即加载就是使用类的时候已经将对象创建完毕，常见的实现方式是直接new实例化。立即加载也称为“饿汉模式”；

延迟加载就是在调用get方法是实例才被创建，常见的实现办法就是在get方法中进行new实例化。延迟加载也称为“懒汉模型”。

**立即加载型单例设计模式（**以下代码的缺点是不能有其它实例变量，因为getInstance()方法没有同步，所以有可能出现非线程安全问题**）**

**public** **class** MyObject {

**private** **static** MyObject *myObject* = **new** MyObject();

**private** MyObject() { }

**public** **static** MyObject getInstance() {

// 此版本代码的缺点是不能有其它实例变量，因为getInstance()方法没有同步，所以有可能出现非线程安全问题

**return** *myObject*;

}

}

**public** **class** MyThread **extends** Thread {

@Override

**public** **void** run() {

System.***out***.println(MyObject.*getInstance*().hashCode());

}

}

**public** **class** Run {

**public** **static** **void** main(String[] args) {

MyThread t1 = **new** MyThread();

MyThread t2 = **new** MyThread();

MyThread t3 = **new** MyThread();

t1.start();

t2.start();

t3.start();

}

}

**运行结果：**

19780024

19780024

19780024

**延迟加载型单例设计模式（**以下代码在多线程环境中根本不能实现保持单例的状态**）**

**将上例中的立即加载模式代码中MyObject类改成如下，实现延迟加载型单例设计模式**

**public** **class** MyObject {

**private** **static** MyObject *myObject* = **new** MyObject();

**private** MyObject() { }

**public** **static** MyObject getInstance() {

//在多线程环境中，该代码无法实现保持单例

**if** (*myObject* == **null**) {

*myObject* = **new** MyObject();

}

**return** *myObject*;

}

}

### 立即/延迟（饿汉/懒汉）加载的单例设计模式在多线编程中的缺点

在上述两个例子中，由于getInstance()方法都不是线程安全方法，所以在立即加载的单例设计实现无法定义多个变量实例看，而延迟加载的单例模式设计根本无法实现单例。

### 多线程环境中的延迟加载单例设计模式的实现方式

可以使用声明synchronized关键字或同步代码块对方法进行持锁，这样就可以解决以上问题，但是这种方法的运行效率非常低下，是同步运行的，下一个线程想要取得对象，则必须等上一个线程释放锁之后，才可以继续执行。以下几种方式可以解决“懒汉模式”遇到多线程的情况。

**使用DCL双检查锁机制实现**

**public** **class** MyObject {

**private** **volatile** **static** MyObject *myObject*;

**private** MyObject() { }

// 使用双检测机制来解决问题，即保证了不需要同步代码的异步，又保证了单例的效果

**public** **static** MyObject getInstance() {

**try** {

**if** (*myObject* != **null**) {

} **else** {

// 模拟在创建对象之前做一些准备性的工作

Thread.*sleep*(3000);

**synchronized** (MyObject.**class**) {

**if** (*myObject* == **null**) {

*myObject* = **new** MyObject();

}

}

}

} **catch** (InterruptedException e) {

e.printStackTrace();

}

**return** *myObject*;

}

}

**public** **class** MyThread **extends** Thread {

@Override

**public** **void** run() {

System.***out***.println(MyObject.*getInstance*().hashCode());

}

}

**public** **class** Run {

**public** **static** **void** main(String[] args) {

MyThread t1 = **new** MyThread();

MyThread t2 = **new** MyThread();

MyThread t3 = **new** MyThread();

t1.start();

t2.start();

t3.start();

}

}

**运行结果：**

14806363

14806363

14806363

**使用内置静态类实现**

**public** **class** MyObject **implements** Serializable {

**private** **static** **final** **long** ***serialVersionUID*** = 888L;

**private** MyObject() { }

**public** **static** MyObject getInstance() {

**return** MyObjectHandler.***myObject***;

}

//如果注释掉该接口方法，则无法实现序列化和反序列化在单例模式中的实现

**protected** Object readResolve() **throws** ObjectStreamException {

System.***out***.println("调用了readResolve方法！");

**return** MyObjectHandler.***myObject***;

}

// 内部类

**private** **static** **class** MyObjectHandler {

**private** **static** **final** MyObject ***myObject*** = **new** MyObject();

}

}

**public** **class** SaveAndRead {

**public** **static** **void** main(String[] args) {

**try** {

MyObject myObject = MyObject.*getInstance*();

FileOutputStream fosRef = **new** FileOutputStream(**new** File("myObjectFile.txt"));

ObjectOutputStream oosRef = **new** ObjectOutputStream(fosRef);

oosRef.writeObject(myObject);

oosRef.close();

fosRef.close();

System.***out***.println(myObject.hashCode());

} **catch** (FileNotFoundException e) {

e.printStackTrace();

} **catch** (IOException e) {

e.printStackTrace();

}

**try** {

FileInputStream fisRef = **new** FileInputStream(**new** File("myObjectFile.txt"));

ObjectInputStream iosRef = **new** ObjectInputStream(fisRef);

MyObject myObject = (MyObject) iosRef.readObject();

iosRef.close();

fisRef.close();

System.***out***.println(myObject.hashCode());

} **catch** (FileNotFoundException e) {

e.printStackTrace();

} **catch** (IOException e) {

e.printStackTrace();

} **catch** (ClassNotFoundException e) {

e.printStackTrace();

}

}

}

**运行结果：**

29310343

调用了readResolve方法！

29310343

**使用static代码块实现**

静态代码块中的代码在使用的时候寄已经执行了，所以可以该特性来实现单例设计模式。

**public** **class** MyObject {

**private** **static** MyObject *instance* = **null**;

**private** MyObject() { }

**static** {

*instance* = **new** MyObject();

}

**public** **static** MyObject getInstance() {

**return** *instance*;

}

}

**public** **class** MyThread **extends** Thread {

@Override

**public** **void** run() {

**for** (**int** i = 0; i < 3; i++) {

System.***out***.println(MyObject.*getInstance*().hashCode());

}

}

}

**public** **class** Run {

**public** **static** **void** main(String[] args) {

MyThread t1 = **new** MyThread();

MyThread t2 = **new** MyThread();

MyThread t3 = **new** MyThread();

t1.start();

t2.start();

t3.start();

}

}

**运行结果：**

17388941

17388941

17388941

17388941

17388941

17388941

17388941

17388941

17388941

**使用enum枚举数据类型实现**

枚举enum和静态代码块的特性相似，在使用枚举类时，构造方法会被自动调用，也可以使用该特性实现单例设计单例模式。

**public** **class** MyObject {

**public** **enum** MyEnumSingleton {

***connectionFactory***;

**private** Connection connection;

**private** MyEnumSingleton() {

**try** {

System.***out***.println("创建MyObject对象");

String url = "jdbc:sqlserver://localhost:1079;databaseName=y2";

String username = "sa";

String password = "";

String driverName = "com.microsoft.sqlserver.jdbc.SQLServerDriver";

Class.*forName*(driverName);

connection = DriverManager.*getConnection*(url, username, password);

} **catch** (ClassNotFoundException e) {

e.printStackTrace();

} **catch** (SQLException e) {

e.printStackTrace();

}

}

**public** Connection getConnection() {

**return** connection;

}

}

**public** **static** Connection getConnection() {

**return** MyEnumSingleton.***connectionFactory***.getConnection();

}

}

**public** **class** MyThread **extends** Thread {

@Override

**public** **void** run() {

**for** (**int** i = 0; i < 3; i++) {

System.***out***.println(MyObject.*getConnection*().hashCode());

}

}

}

**public** **class** Run {

**public** **static** **void** main(String[] args) {

MyThread t1 = **new** MyThread();

MyThread t2 = **new** MyThread();

MyThread t3 = **new** MyThread();

t1.start();

t2.start();

t3.start();

}

}

**运行结果：**

创建MyObject对象

10440721

10440721

10440721

10440721

10440721

10440721

10440721

10440721

10440721

### Lock的使用（未完成）