# 多线程

## 多线程介绍

学习多线程之前，我们先要了解几个关于多线程有关的概念。

进程：进程指正在运行的程序。确切的来说，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。

线程：线程是进程中的一个执行单元，负责当前进程中程序的执行，一个进程中至少有一个线程。一个进程中是可以有多个线程的，这个应用程序也可以称之为多线程程序。

简而言之：一个程序运行后至少有一个进程，一个进程中可以包含多个线程

什么是多线程呢？即就是一个程序中有多个线程在同时执行。

通过下图来区别单线程程序与多线程程序的不同：

* 单线程程序：即，若有多个任务只能依次执行。当上一个任务执行结束后，下一个任务开始执行。如，去网吧上网，网吧只能让一个人上网，当这个人下机后，下一个人才能上网。
* 多线程程序：即，若有多个任务可以同时执行。如，去网吧上网，网吧能够让多个人同时上网。

## 主线程

回想我们以前学习中写过的代码，当我们在dos命令行中输入java空格类名回车后，启动JVM，并且加载对应的class文件。虚拟机并会从main方法开始执行我们的程序代码，一直把main方法的代码执行结束。如果在执行过程遇到循环时间比较长的代码，那么在循环之后的其他代码是不会被马上执行的。如下代码演示：

|  |
| --- |
| **class** ThreadDemo {  **public static void** main(String[] args) {  Demo d = **new** Demo(**"张三"**);  Demo d2 = **new** Demo(**"李四"**);  d.show();  d2.show();  System.***out***.println(**"Hello World!"**);  }  }    **class** Demo {  String **name**;   Demo(String name) {  **this**.**name** = name;  }   **void** show() {  **for** (**int** i = 1; i <= 10000; i++) {  System.***out***.println(**"name="** + **name** + **",i="** + i);  }  } } |

若在上述代码中show方法中的循环执行次数很多，这时在d.show();下面的代码是不会马上执行的，并且在dos窗口会看到不停的输出name=小强,i=值，这样的语句。为什么会这样呢？

原因是：jvm启动后，必然有一个执行路径(线程)从main方法开始的，一直执行到main方法结束，这个线程在java中称之为主线程。当程序的主线程执行时，如果遇到了循环而导致程序在指定位置停留时间过长，则无法马上执行下面的程序，需要等待循环结束后能够执行。那么，能否实现一个主线程负责执行其中一个循环，再由另一个线程负责其他代码的执行，最终实现多部分代码同时执行的效果？能够实现同时执行，通过Java中的多线程技术来解决该问题。

## 线程的创建

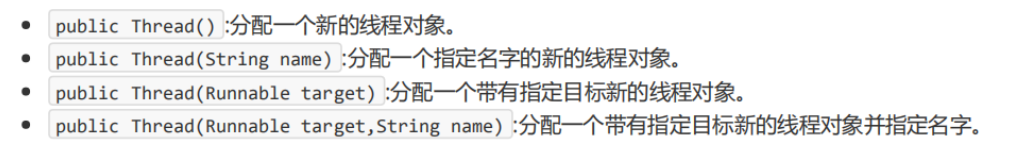
该如何创建线程呢？创建新执行线程有两种方法。

* 一种方法是将类声明为 Thread 的子类。该子类应重写 Thread 类的 run 方法。创建对象，开启线程。run方法相当于其他线程的main方法。
* 另一种方法是声明一个实现 Runnable 接口的类。该类然后实现 run 方法。然后创建Runnable的子类对象，传入到某个线程的构造方法中，开启线程。

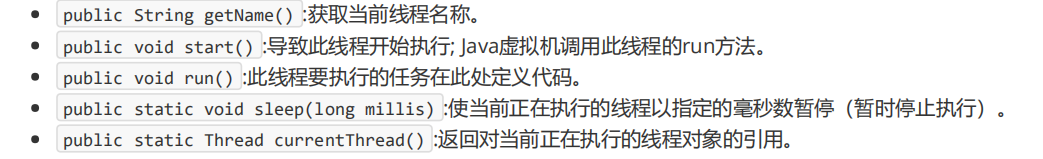
Thread是程序中的执行线程。Java 虚拟机允许应用程序并发地运行多个执行线程。

## 创建线程方式一继承Thread类

* 每个线程都是通过某个特定Thread对象的run()方法来完成操作的，经常把run()方法的主体称为线程体
* 通过该Thread对象的start()方法来启动这个线程，而非直接调用run()
* 构造方法



* 常用方法



创建线程的步骤：

1 定义一个类继承Thread。

2 重写run方法。

3 创建子类对象，就是创建线程对象。

4 调用start方法，开启线程并让线程执行，同时还会告诉jvm去调用run方法。

* 测试类

|  |
| --- |
| **package** com.syedu.java1;  **public class** Demo1 {  **public static void** main(String[] args) {  *//创建自定义线程对象* MyThread mt = **new** MyThread(**"新的线程！"**);  *//开启新线程* mt.start();  *//在主方法中执行for循环* **for** (**int** i = 0; i < 10; i++) {  System.***out***.println(**"main线程！"**+i);  }  }  } |

|  |
| --- |
| **package** com.syedu.java1;  **public class** MyThread **extends** Thread {  *//定义指定线程名称的构造方法* **public** MyThread(String name) {  *//调用父类的String参数的构造方法，指定线程的名称* **super**(name);  }  */\*\*  \* 重写run方法，完成该线程执行的逻辑  \*/* @Override  **public void** run() {  **for** (**int** i = 0; i < 10; i++) {  System.***out***.println(getName()+**"：正在执行！"**+i);  }  } } |

### 线程对象调用 run方法和调用start方法区别？

线程对象调用run方法不开启线程。仅是对象调用方法。线程对象调用start开启线程，并让jvm调用run方法在开启的线程中执行。

### 继承Thread类原理

我们为什么要继承Thread类，并调用其的start方法才能开启线程呢？

继承Thread类：因为Thread类用来描述线程，具备线程应该有功能。那为什么不直接创建Thread类的对象呢？如下代码：

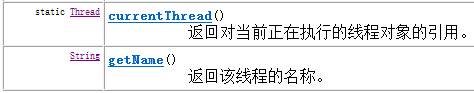
|  |
| --- |
| Thread **t1** = **new** Thread(); t1.start();*//该start调用的是Thread类中的run方法，而这个run方法没有做什么事情，更重要的是这个run方法中并没有定义我们需要让线程执行的代码。* |

创建线程的目的是什么？

是为了建立程序单独的执行路径，让多部分代码实现同时执行。也就是说线程创建并执行需要给定线程要执行的任务。对于之前所讲的主线程，它的任务定义在main函数中。自定义线程需要执行的任务都定义在run方法中。Thread类run方法中的任务并不是我们所需要的，只有重写这个run方法。既然Thread类已经定义了线程任务的编写位置（run方法），那么只要在编写位置（run方法）中定义任务代码即可。所以进行了重写run方法动作。

### 获取线程名称

开启的线程都会有自己的独立运行栈内存，那么这些运行的线程的名字是什么呢？该如何获取呢？既然是线程的名字，按照面向对象的特点，是哪个对象的属性和谁的功能，那么我们就去找那个对象就可以了。查阅Thread类的API文档发现有个方法是获取当前正在运行的线程对象。还有个方法是获取当前线程对象的名称。既然找到了，我们就可以试试。



* Thread.currentThread()获取当前线程对象
* Thread.currentThread().getName();获取当前线程对象的名称

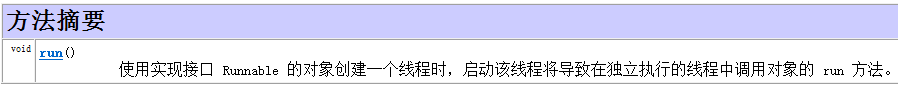
|  |
| --- |
| **package** com.syedu.java1;  **public class** Demo1 {   **public static void** main(String[] args) {  *//创建两个线程任务* MyThread d = **new** MyThread();  MyThread d2 = **new** MyThread();  d.run();*//没有开启新线程, 在主线程调用run方法* d2.start();*//开启一个新线程，新线程调用run方法* } }  **class** MyThread **extends** Thread { *//继承Thread* MyThread(String name){  **super**(name);  }  *//复写其中的run方法* **public void** run(){  **for** (**int** i=1;i<=20 ;i++ ){  System.***out***.println(Thread.*currentThread*().getName()+**",i="**+i);  }  } } |

通过结果观察，原来主线程的名称：main；自定义的线程：Thread-0，线程多个时，数字顺延。如Thread-1......进行多线程编程时，不要忘记了Java程序运行是从主线程开始，main方法就是主线程的线程执行内容。

## 创建线程方式二实现Runnable接口

创建线程的另一种方法是声明实现 Runnable 接口的类。该类然后实现 run 方法。然后创建Runnable的子类对象，传入到某个线程的构造方法中，开启线程。为何要实现Runnable接口，Runnable接口用来指定每个线程要执行的任务。包含了一个 run 的无参数抽象方法，需要由接口实现类重写该方法。

* 接口中的方法



* Thread类构造方法



创建线程的步骤。

* 1、定义类实现Runnable接口。
* 2、覆盖接口中的run方法。。
* 3、创建Thread类的对象
* 4、将Runnable接口的子类对象作为参数传递给Thread类的构造函数。
* 5、调用Thread类的start方法开启线程。
* 代码演示：

|  |
| --- |
| **package** com.syedu.java1;  **public class** Demo1 {   **public static void** main(String[] args) {  *//创建线程执行目标类对象* Runnable runn = **new** MyRunnable();  *//将Runnable接口的子类对象作为参数传递给Thread类的构造函数* Thread thread = **new** Thread(runn);  Thread thread2 = **new** Thread(runn);  *//开启线程* thread.start();  thread2.start();  **for** (**int** i = 0; i < 10; i++) {  System.***out***.println(**"main线程：正在执行！"**+i);  }  } }  **class** MyRunnable **implements** Runnable{   *//定义线程要执行的run方法逻辑* @Override  **public void** run() {   **for** (**int** i = 0; i < 10; i++) {  System.***out***.println(**"我的线程：正在执行！"**+i);  }  } } |

### 实现Runnable的原理

为什么需要定一个类去实现Runnable接口呢？继承Thread类和实现Runnable接口有啥区别呢？实现Runnable接口，避免了继承Thread类的单继承局限性。覆盖Runnable接口中的run方法，将线程任务代码定义到run方法中。创建Thread类的对象，只有创建Thread类的对象才可以创建线程。线程任务已被封装到Runnable接口的run方法中，而这个run方法所属于Runnable接口的子类对象，所以将这个子类对象作为参数传递给Thread的构造函数，这样，线程对象创建时就可以明确要运行的线程的任务。

### 实现Runnable的好处

第二种方式实现Runnable接口避免了单继承的局限性，所以较为常用。实现Runnable接口的方式，更加的符合面向对象，线程分为两部分，一部分线程对象，一部分线程任务。继承Thread类，线程对象和线程任务耦合在一起。一旦创建Thread类的子类对象，既是线程对象，有又有线程任务。实现runnable接口，将线程任务单独分离出来封装成对象，类型就是Runnable接口类型。Runnable接口对线程对象和线程任务进行解耦。

## 创建线程方式三实现Callable接口

* 与使用Runnable相比，Callable功能更强大些
* 相比run()方法，可以有返回值
* 方法可以抛出异常
* 支持泛型的返回值
* 需要借助FutureTask类，比如获取返回结果

#### Future接口

可以对具体Runnable、Callable任务的执行结果进行取消、查询是否完成、获取结果等。

FutrueTask是Futrue接口的唯一的实现类

FutureTask 同时实现了Runnable, Future接口。它既可以作为Runnable被线程执行，又可以作为Future得到Callable的返回值

|  |
| --- |
| **package** com.syedu.java1;  **import** java.util.concurrent.\*;   **class** CallableDemo {   **static class** SumTask **implements** Callable<Long> {   @Override  **public** Long call() **throws** Exception {   **long** sum = 0;  **for** (**int** i = 0; i < 9000; i++) {  sum += i;  }   **return** sum;  }  }   **public static void** main(String[] args) **throws** ExecutionException, InterruptedException {  System.***out***.println(**"Start:"** + System.*nanoTime*());  FutureTask<Long> futureTask = **new** FutureTask<Long>(**new** SumTask());  Executor executor= Executors.*newSingleThreadExecutor*();  executor.execute(futureTask);  System.***out***.println(futureTask.get());  System.***out***.println(**"End:"** + System.*nanoTime*());  } } |

|  |
| --- |
| **package** com.syedu.java1;  **import** java.util.concurrent.Callable; **import** java.util.concurrent.ExecutorService; **import** java.util.concurrent.Executors;  **public class** Demo1 {   **public static void** main(String[] args) {   *//创建线程池对象* ExecutorService pool = Executors.*newFixedThreadPool*(2) ;   *//提交任务* pool.submit(**new** MyCallable()) ;  pool.submit(**new** MyCallable()) ;   *//关闭线程池* pool.shutdown();  }  }    **class** MyCallable **implements** Callable {   @Override  **public** Object call() **throws** Exception {  **for**(**int** x = 0 ; x < 100 ; x ++) {  System.***out***.println(Thread.*currentThread*().getName()+**":"**+x);  }  **return null**;  } } |

## 匿名内部类方式实现多线程

使用线程的内匿名内部类方式，可以方便的实现每个线程执行不同的线程任务操作。

* 方式1：创建线程对象时，直接重写Thread类中的run方法

|  |
| --- |
| **package** com.syedu.java1;  **public class** Demo1 {   **public static void** main(String[] args) {  **new** Thread() {  **public void** run() {  **for** (**int** x = 0; x < 40; x++) {  System.***out***.println(Thread.*currentThread*().getName()  + **"...X...."** + x);  }  }  }.start();  } } |

* 方式2：使用匿名内部类的方式实现Runnable接口，重新Runnable接口中的run方法

|  |
| --- |
| **package** com.syedu.java1;  **public class** Demo1 {   **public static void** main(String[] args) {  Runnable r = **new** Runnable() {  **public void** run() {  **for** (**int** x = 0; x < 40; x++) {  System.***out***.println(Thread.*currentThread*().getName()  + **"...Y...."** + x);  }  }  };  **new** Thread(r).start();  } } |

## Thread和Runnable的区别

如果一个类继承Thread，则不适合资源共享。但是如果实现了Runable接口的话，则很容易的实现资源共享。

实现Runnable接口比继承Thread类所具有的优势：

1. 适合多个相同的程序代码的线程去共享同一个资源。

2. 可以避免java中的单继承的局限性。

3. 增加程序的健壮性，实现解耦操作，代码可以被多个线程共享，代码和线程独立。

4. 线程池只能放入实现Runable或Callable类线程，不能直接放入继承Thread的类。

# 线程安全

## 线程安全（抢购火车票案例）

如果有多个线程在同时运行，而这些线程可能会同时运行这段代码。程序每次运行结果和单线程运行的结果是一样的，而且其他的变量的值也和预期的是一样的，就是线程安全的。

我们通过一个案例，演示线程的安全问题：

我们模拟购买火车票过程。火车票座位共100个

我们来模拟火车票的售票窗口，实现多个窗口同时卖火车票(多个窗口一起卖这100张票)

需要窗口，采用线程对象来模拟；Runnable接口子类来模拟

|  |
| --- |
| **package** com.syedu.java1;  **public class** Ticket **implements** Runnable {  **private int ticket** = 100;  */\*  \* 执行卖票操作  \*/* @Override  **public void** run() { *//每个窗口卖票的操作 //窗口 永远开启* **while** (**true**) {  **if** (**ticket** > 0) {*//有票 可以卖 //出票操作 //使用sleep模拟一下出票时间* **try** {  Thread.*sleep*(100);  } **catch** (InterruptedException e) { *//* ***TODO Auto‐generated catch block*** e.printStackTrace();  } *//获取当前线程对象的名字* String name = Thread.*currentThread*().getName();  System.***out***.println(name + **"正在卖:"** + **ticket**--);  }  }  } } |

|  |
| --- |
| **package** com.syedu.java1;  **import** java.util.concurrent.ExecutionException;  **public class** Demo1 {   **public static void** main(String[] args) **throws** InterruptedException, ExecutionException { *//创建线程任务对象* Ticket ticket = **new** Ticket(); *//创建三个窗口对象* Thread t1 = **new** Thread(ticket, **"窗口1"**);  Thread t2 = **new** Thread(ticket, **"窗口2"**);  Thread t3 = **new** Thread(ticket, **"窗口3"**); *//同时卖票* t1.start();  t2.start();  t3.start();  } } |

## 线程同步（抢购火车票案例）

当我们使用多个线程访问同一资源的时候，且多个线程中对资源有写的操作，就容易出现线程安全问题。要解决上述多线程并发访问一个资源的安全性问题:也就是解决重复票与不存在票问题，Java中提供了同步机制(synchronized)来解决。

窗口1线程进入操作的时候，窗口2和窗口3线程只能在外等着，窗口1操作结束，窗口1和窗口2和窗口3才有机会进入代码去执行。也就是说在某个线程修改共享资源的时候，其他线程不能修改该资源，等待修改完毕同步之后，才能去抢夺CPU资源，完成对应的操作，保证了数据的同步性，解决了线程不安全的现象。

为了保证每个线程都能正常执行原子操作,Java引入了线程同步机制。那么怎么去使用呢？有三种方式完成同步操作

* 1. 同步代码块。
* 2. 同步方法。
* 3. 锁机制。

## 同步代码块（抢购火车票案例）

同步代码块： synchronized 关键字可以用于方法中的某个区块中，表示只对这个区块的资源实行互斥访问。

格式:

|  |
| --- |
| synchronized(同步锁){  需要同步操作的代码  } |

**同步锁:**

对象的同步锁只是一个概念,可以想象为在对象上标记了一个锁.

1. 锁对象 可以是任意类型。

2. 多个线程对象 要使用同一把锁

|  |
| --- |
| **package** com.syedu.java1;  **public class** Ticket **implements** Runnable {  **private int ticket** = 100;  Object **lock** = **new** Object();   */\*  \* 执行卖票操作  \*/* @Override  **public void** run() { *//每个窗口卖票的操作 //窗口 永远开启* **while** (**true**) {  **synchronized** (**lock**) {  **if** (**ticket** > 0) {*//有票 可以卖 //出票操作 //使用sleep模拟一下出票时间* **try** {  Thread.*sleep*(50);  } **catch** (InterruptedException e) { *//* ***TODO Auto‐generated catch block*** e.printStackTrace();  } *//获取当前线程对象的名字* String name = Thread.*currentThread*().getName();  System.***out***.println(name + **"正在卖:"** + **ticket**--);  }  }  }  } } |

## 同步方法（抢购火车票案例）

同步方法:使用synchronized修饰的方法,就叫做同步方法,保证A线程执行该方法的时候,其他线程只能在方法外等着。

|  |
| --- |
| public synchronized void method(){  可能会产生线程安全问题的代码  } |

同步锁是谁?

对于非static方法,同步锁就是this。

对于static方法,我们使用当前方法所在类的字节码对象(类名.class)。

|  |
| --- |
| **package** com.syedu.java1;  **public class** Ticket **implements** Runnable{  **private int ticket** = 100;  */\*  \* 执行卖票操作  \*/* @Override  **public void** run() { *//每个窗口卖票的操作 //窗口 永远开启* **while**(**true**){  sellTicket();  }  }  */\*  \* 锁对象 是 谁调用这个方法 就是谁  \* 隐含 锁对象 就是 this  \*  \*/* **public synchronized void** sellTicket(){  **if**(**ticket**>0){*//有票 可以卖 //出票操作 //使用sleep模拟一下出票时间* **try** {  Thread.*sleep*(100);  } **catch** (InterruptedException e) { *//* ***TODO Auto‐generated catch block*** e.printStackTrace();  } *//获取当前线程对象的名字* String name = Thread.*currentThread*().getName();  System.***out***.println(name+**"正在卖:"**+**ticket**--);  }  } } |

## Lock锁（抢购火车票案例）

java.util.concurrent.locks.Lock 机制提供了比synchronized代码块和synchronized方法更广泛的锁定操作,

同步代码块/同步方法具有的功能Lock都有,除此之外更强大,更体现面向对象。

Lock锁也称同步锁，加锁与释放锁方法化了，如下：

public void lock() :加同步锁。

public void unlock() :释放同步锁。

|  |
| --- |
| **package** com.syedu.java1;  **import** java.util.concurrent.locks.Lock; **import** java.util.concurrent.locks.ReentrantLock;  **public class** Ticket **implements** Runnable{  **private int ticket** = 100;  Lock **lock** = **new** ReentrantLock();  */\*  \* 执行卖票操作  \*/* @Override  **public void** run() { *//每个窗口卖票的操作 //窗口 永远开启* **while**(**true**){  **lock**.lock();  **if**(**ticket**>0){*//有票 可以卖 //出票操作 //使用sleep模拟一下出票时间* **try** {  Thread.*sleep*(50);  } **catch** (InterruptedException e) {  e.printStackTrace();  } *//获取当前线程对象的名字* String name = Thread.*currentThread*().getName();  System.***out***.println(name+**"正在卖:"**+**ticket**--);  }  **lock**.unlock();  }  } } |

## 线程的死锁问题

**死锁：**不同的线程分别占用对方需要的同步资源不放弃，都在等待对方放弃自己需要的同步资源，就形成了线程的死锁出现死锁后，不会出现异常，不会出现提示，只是所有的线程都处于阻塞状态，无法继续

解决方法

专门的算法、原则

尽量减少同步资源的定义

尽量避免嵌套同步

# 线程调度

通常我们的计算机只有一个 CPU，CP U 在某一个时刻只能执行一条指令，线程只有得到 CPU时间片，也就是使用权，才可以执行指令。在单 CPU 的机器上线程不是并行运行的，只有在多个 CPU 上线程才可以并行运行。Java 虚拟机要负责线程的调度，取得 CPU 的使用权 ，目前有两种调度模型：分时调度模型和抢占式调度模型，Java使用抢占式调度模型。

分时调度模型：所有线程轮流使用 CPU 的使用权，平均分配每个线程占用 CPU 的时间片

抢占式调度模型：优先让优先级高的线程使用 CPU，如果线程的优先级相同，那么会随机

选择一个，优先级高的线程获取的 CPU 时间片相对多一些。

## 程序运行

* 分时调度

所有线程轮流使用 CPU 的使用权，平均分配每个线程占用 CPU 的时间。

* 抢占式调度

优先让优先级高的线程使用 CPU，如果线程的优先级相同，那么会随机选择一个(线程随机性)，Java使用的为抢占式调度。

### 抢占式调度详解

大部分操作系统都支持多进程并发运行，现在的操作系统几乎都支持同时运行多个程序。比如：现在我们上课一边使用编辑器，一边使用录屏软件，同时还开着画图板，dos窗口等软件。此时，这些程序是在同时运行，”感觉这些软件好像在同一时刻运行着“。

实际上，CPU(中央处理器)使用抢占式调度模式在多个线程间进行着高速的切换。对于CPU的一个核而言，某个时刻，只能执行一个线程，而 CPU的在多个线程间切换速度相对我们的感觉要快，看上去就是在同一时刻运行。

其实，多线程程序并不能提高程序的运行速度，但能够提高程序运行效率，让CPU的使用率更高。

## 线程优先级

线 程 优 先 级 主 要 分 三 种 ： MAX\_PRIORITY( 最 高 级 );MIN\_PRIORITY （ 最 低 级 ）NORM\_PRIORITY(标准)默认

优先级高的线程会得到的CPU 时间多一些，优先执行完成

|  |
| --- |
| **package** com.syedu.java1;   **import** java.util.concurrent.ExecutionException;  **public class** Demo1 {   **public static void** main(String[] args) {  Runnable r1 = **new** Processor();  Thread t1 = **new** Thread(r1, **"t1"**); *//设置线程的优先级，线程启动后不能再次设置优先级 //必须在启动前设置优先级 //设置最高优先级* t1.setPriority(Thread.***MAX\_PRIORITY***); *//启动线程* t1.start(); *//取得线程名称 //System.out.println(t1.getName());* Thread t2 = **new** Thread(r1, **"t2"**); *//设置最低优先级* t2.setPriority(Thread.***MIN\_PRIORITY***);  t2.start();  System.***out***.println(Thread.*currentThread*().getName());  } }   **class** Processor **implements** Runnable {  **public void** run() {  **for** (**int** i = 0; i < 100; i++) {  System.***out***.println(Thread.*currentThread*().getName() + **","** + i);  }  } } |

## 线程的生命周期

当线程被创建并启动以后，它既不是一启动就进入了执行状态，也不是一直处于执行状态。在线程的生命周期中，有几种状态呢？要想实现多线程，必须在主线程中创建新的线程对象。Java语言使用Thread类及其子类的对象来表示线程，在它的一个完整的生命周期中通常要经历如下的五种状态：

新建：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态

就绪：处于新建状态的线程被start()后，将进入线程队列等待CPU时间片，此时它已具备了运行的条件，只是没分配到CPU资源

运行：当就绪的线程被调度并获得CPU资源时,便进入运行状态，run()方法定义了线程的操作和功能

阻塞：在某种特殊情况下，被人为挂起或执行输入输出操作时，让出CPU并临时中止自己的执行，进入阻塞状态

死亡：线程完成了它的全部工作或线程被提前强制性地中止或出现异常导致结束

## Thread.sleep

sleep 设置休眠的时间,单位毫秒，当一个线程遇到sleep 的时候，就会睡眠，进入到阻塞状态,放弃CPU，腾出cpu 时间片，给其他线程用，所以在开发中通常我们会这样做，使其他的线程能够取得CPU 时间片，当睡眠时间到达了，线程会进入可运行状态，得到CPU 时间片继续执行，如果线程在睡眠状态被中断了，将会抛出IterruptedException

|  |
| --- |
| **package** com.syedu.java1;   **public class** Demo1 {   **public static void** main(String[] args) {  Runnable r1 = **new** Processor();  Thread t1 = **new** Thread(r1, **"t1"**);  t1.start();  Thread t2 = **new** Thread(r1, **"t2"**);  t2.start();  } }  **class** Processor **implements** Runnable {  **public void** run() {  **for** (**int** i=0; i<100; i++) {  System.***out***.println(Thread.*currentThread*().getName() + **","** + i);  **if** (i % 10 == 0) {  **try** { *//睡眠100 毫秒，主要是放弃CPU 的使用，将CPU 时间片交给其他线程使用* Thread.*sleep*(100);  }**catch**(InterruptedException e) {  e.printStackTrace();  }  }  }  } } |

## Thread.yield

它与sleep()类似，只是不能由用户指定暂停多长时间，并且yield()方法只能让同优先级的线程有执行的机会

|  |
| --- |
| **package** com.syedu.java1;   **public class** Demo1 {   **public static void** main(String[] args) {  Runnable r1 = **new** Processor();  Thread t1 = **new** Thread(r1, **"t1"**);  t1.start();  Thread t2 = **new** Thread(r1, **"t2"**);  t2.start();  } } **class** Processor **implements** Runnable {  **public void** run() {  **for** (**int** i=0; i<100; i++) {  System.***out***.println(Thread.*currentThread*().getName() + **","** + i);  **if** (i % 10 == 0) {  System.***out***.println(**"--------------"**); *//采用yieid 可以将CPU 的使用权让给同一个优先级的线程* Thread.*yield*();  }  }  } } |

## Thread.join

当前线程可以调用另一个线程的join 方法，调用后当前线程会被阻塞不再执行，直到被调

用的线程执行完毕，当前线程才会执行

|  |
| --- |
| **package** com.syedu.java1;   **public class** Demo1 {   **public static void** main(String[] args) {  Runnable r1 = **new** Processor();  Thread t1 = **new** Thread(r1, **"t1"**);  t1.start();  **try** {  t1.join();  }**catch**(InterruptedException e) {  e.printStackTrace();  }  System.***out***.println(**"------main end-------"**);  } } **class** Processor **implements** Runnable {  **public void** run() {  **for** (**int** i=0; i<10; i++) {  System.***out***.println(Thread.*currentThread*().getName() + **","** + i);  }  } } |

## Thread.interrupt（中断）

如果我们的线程正在睡眠，可以采用interrupt 进行中断

|  |
| --- |
| **package** com.syedu.java1;  **public class** Demo1 {   **public static void** main(String[] args) {  Runnable r1 = **new** Processor();  Thread t1 = **new** Thread(r1, **"t1"**);  t1.start();  **try** {  *//设置为500 毫秒，没有出现中断异常，因为  //500 毫秒之后再次调用t1.interrupt()时，  //此时的睡眠线程已经执行完成  //如果sleep 的时间设置的小一些，会出现中断异常，  //因为存在睡眠线程* Thread.*sleep*(500);  } **catch** (Exception e) {  e.printStackTrace();  }  *//中断睡眠中的线程* t1.interrupt();  } }  **class** Processor **implements** Runnable {  **public void** run() {   **for** (**int** i = 1; i < 100; i++) {  System.***out***.println(Thread.*currentThread*().getName() + **","** + i);  **if** (i % 50 == 0) {  **try** {  Thread.*sleep*(200);  } **catch** (Exception e) {  System.***out***.println(**"-------中断-------"**);  **break**;  }  }  }  } } |

## 线程间的通信

### 什么是线程间通信

多个线程并发执行时, 在默认情况下CPU是随机切换线程的，当我们需要多个线程来共同完成一件任务，并且我们希望他们有规律的执行, 那么多线程之间需要一些协调通信，以此来帮我们达到多线程共同操作一份数据。

### 线程间通信的方式

多个线程在处理同一个资源，并且任务不同时，需要线程通信来帮助解决线程之间对同一个变量的使用或操作。 就是多个线程在操作同一份数据时， 避免对同一共享变量的争夺。也就是我们需要通过一定的手段使各个线程能有效的利用资源。而这种手段即—— **等待唤醒机制。**

使用两个线程打印1-100。线程1, 线程2 交替打印

|  |
| --- |
| **package** com.syedu.java1;  **class** ThreadCommunication **implements** Runnable {  **int i** = 1;  **public void** run() {  **while** (**true**) {  **synchronized** (**this**) {  notify();  **if** (**i** <= 100) {  System.***out***.println(Thread.*currentThread*().getName() +  **":"** + **i**++);  } **else  break**;  **try** {  wait();  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  }  } } |

### 线程通信常用方法



wait() 与 notify() 和 notifyAll() 

* wait()：令当前线程挂起并放弃CPU、同步资源并等待，使别的线程可访问并修改共享资源，而当 前线程排队等候其他线程调用notify()或notifyAll()方法唤醒，唤醒后等待重新获得对监视器的所有 权后才能继续执行。 
* notify()：唤醒正在排队等待同步资源的线程中优先级最高者结束等待



* notifyAll ()：唤醒正在排队等待资源的所有线程结束等待.
* 这三个方法只有在synchronized方法或synchronized代码块中才能使用，否则会报 java.lang.IllegalMonitorStateException异常。 因为这三个方法必须有锁对象调用，而任意对象都可以作为synchronized的同步锁， 因此这三个方法只能在Object类中声明。

### wait() 方法

* 在当前线程中调用方法：对象名.wait()
* 使当前线程进入等待（某对象）状态，直到另一线程对该对象发出notify (或notifyAll) 为止。
* 调用方法的必要条件：当前线程必须具有对该对象的监控权（加锁）
* 调用此方法后，当前线程将释放对象监控权，然后进入等待
* 在当前线程被notify后，要重新获得监控权，然后从断点处继续代码的执行。

### notify()/notifyAll()

* 在当前线程中调用方法：对象名.notify()
* 功能：唤醒等待该对象监控权的一个/所有线程。
* 调用方法的必要条件：当前线程必须具有对该对象的监控权（加锁）

## 关闭线程

通常定义一个标记，来判断标记的状态停止线程的执行

|  |
| --- |
| **package** com.syedu.java1;  **public class** Demo1 {   **public static void** main(String[] args) { *//Runnable r1 = new Processor();* Processor r1 = **new** Processor();  Thread t1 = **new** Thread(r1, **"t1"**);  t1.start();  **try** {  Thread.*sleep*(20);  } **catch** (Exception e) {  } *//停止线程* r1.setFlag(**true**);  } }  **class** Processor **implements** Runnable {  *//线程停止标记，true 为停止* **private boolean flag**;   **public void** run() {  **for** (**int** i = 1; i < 100; i++) {  System.***out***.println(Thread.*currentThread*().getName() + **","** + i); *//为true 停止线程执行* **if** (**flag**) {  **break**;  }  }  }   **public void** setFlag(**boolean** flag) {  **this**.**flag** = flag;  } } |

## 守护线程

从线程分类上可以分为：用户线程（以上讲的都是用户线程），另一个是守护线程。守护线

程是这样的，所有的用户线程结束生命周期，守护线程才会结束生命周期，只要有一个用户

线程存在，那么守护线程就不会结束，例如java 中垃圾回收器就是一个守护线程，

只有应用程序中所有的线程结束，它才会结束。

|  |
| --- |
| **package** com.syedu.java1;  **public class** Demo1 {   **public static void** main(String[] args) {  Runnable r1 = **new** Processor();  Thread t1 = **new** Thread(r1, **"t1"**);  t1.start();  **for** (**int** i=0; i<10; i++) {  System.***out***.println(Thread.*currentThread*().getName() + **", "** + i);  }  System.***out***.println(**"主线程结束！！！"**);  } } **class** Processor **implements** Runnable {  **public void** run() {  **for** (**int** i=0; i<10; i++) {  System.***out***.println(Thread.*currentThread*().getName() + **", "** + i);  }  } } |

## 修改守护线程

设置为守护线程后，当主线程结束后，守护线程并没有把所有的数据输出完就结束了，也即

是说守护线程是为用户线程服务的，当用户线程全部结束，守护线程会自动结束

|  |
| --- |
| **package** com.syedu.java1;  **public class** Demo1 {   **public static void** main(String[] args) {  Runnable r1 = **new** Processor();  Thread t1 = **new** Thread(r1, **"t1"**); *//将当前线程修改为守护线程 //在线程没有启动时可以修改以下参数* t1.setDaemon(**true**);  t1.start();  **for** (**int** i=0; i<10; i++) {  System.***out***.println(Thread.*currentThread*().getName() + **", "** + i);  }  System.***out***.println(**"主线程结束！！！"**);  } } **class** Processor **implements** Runnable {  **public void** run() {  **for** (**int** i=0; i<10; i++) {  System.***out***.println(Thread.*currentThread*().getName() + **", "** + i);  }  } } |

# 线程池

## 线程池概念

程序启动一个新线程成本是比较高的，因为它涉及到要与操作系统进行交互。而使用线程池可以很好的提高性能，尤其是当程序中要创建大量生存期很短的线程时，更应该考虑使用线程池。线程池里的每一个线程代码结束后，并不会死亡，而是再次回到线程池中成为空闲状态，等待下一个对象来使用。（从JDK5开始，Java内置支持线程池）

我们详细的解释一下为什么要使用线程池？

在java中，如果每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，创建和销毁线程花费的时间和消耗的系统资源都相当大，甚至可能要比在处理实际的用户请求的时间和资源要多的多。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，可能会使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，需要采取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。

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。通过对多个任务重复使用线程，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，使用应用程序响应更快。另外，通过适当的调整线程中的线程数目可以防止出现资源不足的情况。

## 合理利用线程池能够带来三个好处：

* 降低资源消耗。减少了创建和销毁线程的次数，每个工作线程都可以被重复利用，可执行多个任务。
* 提高响应速度。当任务到达时，任务可以不需要的等到线程创建就能立即执行。
* 提高线程的可管理性。可以根据系统的承受能力，调整线程池中工作线线程的数目，防止因为消耗过多的内存，而把服务器累趴下(每个线程需要大约1MB内存，线程开的越多，消耗的内存也就越大，最后死机)。

## 使用线程池方式--Runnable接口

通常，线程池都是通过线程池工厂创建，再调用线程池中的方法获取线程，再通过线程去执行任务方法。

* Executors：线程池创建工厂类
  + public static ExecutorService newFixedThreadPool(int nThreads)：返回线程池对象
* ExecutorService：线程池类
  + Future<?> **submit**(Runnable task)：获取线程池中的某一个线程对象，并执行
* Future接口：用来记录线程任务执行完毕后产生的结果。线程池创建与使用
* 使用线程池中线程对象的步骤：
  + 创建线程池对象
  + 创建Runnable接口子类对象
  + 提交Runnable接口子类对象
  + 关闭线程池

代码演示：

|  |
| --- |
| **package** com.syedu.java1;  **import** java.util.concurrent.ExecutorService; **import** java.util.concurrent.Executors;  **public class** Demo1 {   **public static void** main(String[] args) {  *//创建线程池对象* ExecutorService service = Executors.*newFixedThreadPool*(2);*//包含2个线程对象  //创建Runnable实例对象* MyRunnable r = **new** MyRunnable();   *//自己创建线程对象的方式  //Thread t = new Thread(r);  //t.start(); ---> 调用MyRunnable中的run()   //从线程池中获取线程对象,然后调用MyRunnable中的run()* service.submit(r);  *//再获取个线程对象，调用MyRunnable中的run()* service.submit(r);  service.submit(r); *//注意：submit方法调用结束后，程序并不终止，是因为线程池控制了线程的关闭。将使用完的线程又归还到了线程池中  //关闭线程池  //service.shutdown();* } } |

|  |
| --- |
| **package** com.syedu.java1;  **public class** MyRunnable **implements** Runnable {  @Override  **public void** run() {  System.***out***.println(**"我要一个服务员"**);   **try** {  Thread.*sleep*(2000);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  System.***out***.println(**"服务员来了： "** +Thread.*currentThread*().getName());  System.***out***.println(**"吃完饭服务员走了"**);  } } |

## 使用线程池方式—Callable接口

* Callable接口：与Runnable接口功能相似，用来指定线程的任务。其中的call()方法，用来返回线程任务执行完毕后的结果，call方法可抛出异常。
* ExecutorService：线程池类
  + <T> Future<T> **submit**(Callable<T> task)：获取线程池中的某一个线程对象，并执行线程中的call()方法
* Future接口：用来记录线程任务执行完毕后产生的结果。线程池创建与使用
* 使用线程池中线程对象的步骤：
  + 创建线程池对象
  + 创建Callable接口子类对象
  + 提交Callable接口子类对象
  + 关闭线程池

代码演示：

|  |
| --- |
| **package** com.syedu.java1;  **import** java.util.concurrent.ExecutorService; **import** java.util.concurrent.Executors;  **public class** Demo1 {   **public static void** main(String[] args) {  *//创建线程池对象* ExecutorService service = Executors.*newFixedThreadPool*(2);*//包含2个线程对象  //创建Callable对象* MyCallable c = **new** MyCallable();   *//从线程池中获取线程对象,然后调用MyRunnable中的run()* service.submit(c);   *//再获取个教练* service.submit(c);  service.submit(c); *//注意：submit方法调用结束后，程序并不终止，是因为线程池控制了线程的关闭。将使用完的线程又归还到了线程池中  //关闭线程池  //service.shutdown();* } } |

|  |
| --- |
| **package** com.syedu.java1;  **import** java.util.concurrent.Callable;  **public class** MyCallable **implements** Callable {  @Override  **public** Object call() **throws** Exception {  System.***out***.println(**"我要一个服务员"**);  Thread.*sleep*(2000);  System.***out***.println(**"服务员来了： "** +Thread.*currentThread*().getName());  System.***out***.println(**"吃完饭服务员走了"**);  **return null**;  } } |

## 线程池练习：返回两个数相加的结果

要求：通过线程池中的线程对象，使用Callable接口完成两个数求和操作

* Future接口：用来记录线程任务执行完毕后产生的结果。线程池创建与使用
  + V get() 获取Future对象中封装的数据结果

代码演示：

|  |
| --- |
| **package** com.syedu.java1;  **import** java.util.concurrent.ExecutionException; **import** java.util.concurrent.ExecutorService; **import** java.util.concurrent.Executors; **import** java.util.concurrent.Future;  **public class** Demo1 {   **public static void** main(String[] args) **throws** InterruptedException, ExecutionException {  *//创建线程池对象* ExecutorService threadPool = Executors.*newFixedThreadPool*(2);   *//创建一个Callable接口子类对象  //MyCallable c = new MyCallable();* MyCallable c = **new** MyCallable(100, 200);  MyCallable c2 = **new** MyCallable(10, 20);   *//获取线程池中的线程，调用Callable接口子类对象中的call()方法, 完成求和操作  //<Integer> Future<Integer> submit(Callable<Integer> task)  // Future 结果对象* Future<Integer> result = threadPool.submit(c);  *//此 Future 的 get 方法所返回的结果类型* Integer sum = result.get();  System.***out***.println(**"sum="** + sum);   *//再演示* result = threadPool.submit(c2);  sum = result.get();  System.***out***.println(**"sum="** + sum);  *//关闭线程池(可以不关闭)* } } |

|  |
| --- |
| **package** com.syedu.java1;  **import** java.util.concurrent.Callable;  **public class** MyCallable **implements** Callable {  *//成员变量* **int x** = 5;  **int y** = 3;  *//构造方法* **public** MyCallable(){  }  **public** MyCallable(**int** x, **int** y){  **this**.**x** = x;  **this**.**y** = y;  }   @Override  **public** Integer call() **throws** Exception {  **return x**+**y**;  } } |

# 总结

## 知识点总结

* 创建线程的方式
  + 方式1，继承Thread线程类
    - 步骤

1. 自定义类继承Thread类
2. 在自定义类中重写Thread类的run方法
3. 创建自定义类对象(线程对象)
4. 调用start方法，启动线程，通过JVM，调用线程中的run方法
   * 方式2，实现Runnable接口
     + 步骤
5. 创建线程任务类 实现Runnable接口
6. 在线程任务类中 重写接口中的run方法
7. 创建线程任务类对象
8. 创建线程对象，把线程任务类对象作为Thread类构造方法的参数使用
9. 调用start方法，启动线程，通过JVM，调用线程任务类中的run方法