## 多线程

# 线程池：

Java通过Executors提供四种线程池，分别为：

Executors.newCachedThreadPool() 创建一个可缓存线程池，如果线程池长度超过处理需要，可灵活 回收空闲线程，若无可回收线程，则创建线程；

Executors.newFixedThreadPool() 创建一个定长线程池，可控制线程最大并发数，超出的线程会在队列中等待；

Executors.newScheduledThreadPool() 创建一个定长线程池，支持定时周期性执行任务；（该定时线程中，submit方法执行线程和其他线程一样，调用schedule方法及其他相关方法，可设置定时任务执行线程）

Executors.newSingleThreadExecutor() 创建一个单线程化的线程池，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序（FIFO，LIFO，优先级）执行

通过线程池来executor(Runnable r)来执行线程。

Executor接口是线程池的顶级接口，

ExecutorService 是Executor接口的子类，也是真正的线城池。包含线程池的相关方法，返回一个future对象。

Executors用来创建线程池的工具类。

ThreadPoolExecutor：是executorService的默认实现，executors创建线程池，实际上就是调用了ThreadPoolExecutor的构造方法。

# 一、多线程

线程的状态：创建，就绪，运行，休眠，终止。

## 1.1、使用多线程

进程与线程的区别：

进程是操作系统结构的基础，是一次程序的执行；是一个程序及其数据在处理机上顺序执行时所发生的活动，是系统进行资源分配和调度的一个独立单位。

线程是在进程中独立运行的子任务。 多线程是异步的

一个进程正在运行时至少会有一个线程在运行。

实现多线程编程的方式主要有两种：一种是继承Thread类，一种是实现Runable接口。

通过源码可以发现，Thread类也是实现了Runable接口，他们之间具有多态关系。

Thread类只能单继承，而Runable接口可以实现多实现。

使用多线程时，代码的运行结果与代码的执行顺序或调用顺序是无关的。

### 1.1.1、继承Thread类

首先，通过继承Thread类来实现多线程。

继承Thread类，同时要实现run()方法；调用start()方法执行线程。但如果多次调用start()方法会报异常。应注意的是，调用start()方法的顺序不代表线程启动的顺序。

线程的执行代码写在run（）方法中。

### 1.1.2、实现Runable接口

因为Thread类只能单继承，所以创建线程可以使用实现Runable接口的方式，可以多实现。同样，实现Runable接口，需要实现run()方法。执行多线程，将实现了Runable接口的实例化对象方法Thread构造函数中，然后调用start()方法。

Thread类的构造函数可以接收实现了Runable接口的对象，所以同样可以接收Thread类的对象。

注：创建线程还可以通过实现Callable接口实现call方法：

1. 创建实现Callable接口的类
2. 创建线程池
3. 将实现Callable接口的类传入线程池，并调用submit方法，返回Future类调用get方法可接收返回值。

### 1.1.3、实例变量和线程安全

自定义线程类中的实例变量针对其他线程可以由共享与不共享之分。

（1）不共享数据的情况

每个线程都有自己的数据，互不影响。

（2）共享数据

共享数据是多个线程可以访问同一个变量。由于多个线程共享相同的数据造成了线程不安全。

通过在run()方法前面加上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。

synchronized可以在任意对象或方法前加上锁，加锁的代码区被称为“互斥区”或“临界区”。Synchronized可形成同步代码块或同步方法。使用同步方法锁时，一般使用本类的class。

当一个线程想要执行同步方法里的代码时，线程首先尝试去获取锁，如果能够获取这把锁线程就可以执行synchronized中的代码；如果不能够获取这把锁，线程就会不断重复尝试去获取锁，直至获取到。并且是多个线程资源同时去争抢这把锁。

非线程安全：主要指多个线程对同一对象的同一实例进行操作时会出现值被更改、不同步的情况，进而影响程序执行流程。

如何解决非线程安全？

解决非线程安全的方法时使用synchronized关键字。

## 1.2、currentThread()方法

currentThread()方法返回代码段正在被哪个线程调用的信息。

## 1.3、isAlive()方法

该方法用于判断线程是否处于活动状态。即处于已启动尚未终止。线程处于正在运行或准备开始运行的状态认为线程是“存活”的。

## 1.4、sleep()方法

该方法是在指定的毫秒数内让当前“正在执行的线程”休眠（暂停执行），即this.currentThread()返回的线程。

**谁调用的谁去睡觉。**sleep方法没有释放锁，不让出系统资源。

## 1.5、getId()方法

该方法是获得线程的唯一标识。

## 1.6、停止线程

停止一个线程可以使用Thread.stop()方法，但该方法是“不安全”，已被废弃；

大多数线程停止使用Thread.interrupt()方法。该方法不会终止一个正在运行的线程，需加入一个判断才能完成线程的终止。

在java中有3中方式可以终止正在运行的线程：

1.使用退出标志，使线程正常退出，即run()方法完成后线程终止；

2.使用stop()方法强行终止线程，不推荐此用法。以过期作废。

3、使用interrupt()方法终止线程。

### 1.6.1、停止不了的线程

使用interrupt()方法并不是真正停止线程，仅仅是给线程打了一个停止的标记。即标记终止

### 1.6.2、判断线程是否是停止状态

Java中提供了两种方法判断是否处于停止状态：

1.this.interrupted()：测试当前线程是否已经中断；

2.this.isInterrupted()：测试线程是否已经中断。

两者的区别：

1）this.interrupted（）方法测试当前线程是否中断，是指运行该方法的线程，该方法为静态方法，

官方文档对interrupted()方法的解释是：测试当前线程是否中断，线程的中断状态由该方法清除。即如果连续两次调用该方法，则第二次会返回false（在第一次调用已经清除了其中的中断状态，且第二次调用校验完中断状态前，当前线程再次中断的情况除外）。

可见，interrupted（）方法具有清除状态的功能。

2）this.isInterrupted() 方法测试线程是否中断，该方法为非静态方法；isInterrupted（）方法并未清除状态标志，

* this.interrupted()：测试当前线程是否已经中断，执行完成后具有将状态标志置清除为false的功能；
* this.isInterrupted()：测试线程是否已经中断，但不清楚状态标志。

### 1.6.3、能停止的线程—异常法

通过手动抛出异常throw new InterruptedException()，进而中断后续代码。

### 1.6.4、在沉睡中停止

如果在sleep状态下停止某一线程，会进入catch语句，并且清除停止状态值，使之变为false；

如果反之呢？

### 1.6.5、能停止的线程---暴力停止

使用stop()方法停止线程是非常暴力的。由于stop方法已经过期作废，所有应避免使用。使用stop释放锁将会给数据造成不一致的结果。

### 1.6.6、使用return停止线程

将interrupt()方法与return结合使用也能实现停止线程的效果。使用interrupt()标记停止状态，通过isInterrupted()方法测试线程是否中断。

但优先选择使用抛异常throw new InterruptedException()的方式停止线程。

## 1.7、暂停线程

暂停线程意味着还可以恢复运行；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。

1.7.1、suspend()与resume()方法的使用

通过suspend()方法可以暂停线程的执行，通过resume()方法可以继续执行线程。

1.7.2、suspend()与resume()方法的缺点

---独占

当两个方法使用不当时，极易造成公共的同步对象的独占，使其他线程无法访问公共同步对象。suspend()方法已经过期作废。

---不同步

使用两个方法也容易造成因为线程暂停而导致数据不同步的情况。

## 1.8、yield方法

yield()方法的作用是放弃当前的CPU资源，将它让给其他的任务去占用CPU执行时间；但放弃的时间不确定，可能刚放弃就立刻重新获取CPU时间片。

yield()方法会释放资源释放锁，短暂放弃CPU资源的占用；但释放锁的时间是不确定的。

## 1.9、线程的优先级

在操作系统中，线程具有优先级，优先级较高的线程得到的CPU资源较多，即CPU优先执行优先级较高的线程对象的任务。

设置线程的优先级有助于确定下一次选择哪个线程来优先执行。设置线程优先级使用setPriority()方法。

在java中线程的优先级分为1~10这10个等级，如果小于1或大于10 则会抛出异常。JDK中使用3个预置定义优先级的值：

public final static int MIN\_PRIORITY = 1;

public final static int NORM\_PRIORITY = 5;

public final static int MAX\_PRIORITY = 10;

### 1.9.1、线程优先级的继承特性

在java中线程的优先级具有继承性。比如，A线程启动B线程，B线程的优先级与A线程的优先级是一样的。

### 1.9.2、优先级具有规则

虽然使用setPriority()方法可以设置线程的优先级。但高优先级的线程总是大部分先执行完成，但不代表高优先级的线程全部执行完成。当线程优先级的等级差距比较大时，谁先执行和代码的调用顺序无关。

可以看出，线程的优先级具有一定的规则性，即CPU尽量将执行资源让给优先级比较高的线程。

### 1.9.3、优先级具有随机性

线程的优先级具有“随机性”，，即优先级较高的线程不一定每一次都先执行完成。

所以，不能把线程的优先级与运行结果的顺序作为衡量的标准，优先级较高的线程并不是每一次都先执行完成，线程的执行具有不确定性和随机性。

## 1.10、守护线程

在java线程中有两种线程：一是用户线程；二是守护线程。

守护线程是一种特殊的线程，它具有“陪伴”的含义。当线程中不存在非守护线程，守护线程会自动销毁。典型的线程就是垃圾回收线程。当进程中没有非守护线程了，那么垃圾回收线程也就没有存在的必要了。守护线程最典型的就是GC（垃圾回收器）。

通过setDaemon（true）来将线程设为守护线程。

# 二、对象及变量的并发访问

## 2.1、synchronized同步方法

“线程安全”与“非线程安全”

“非线程安全”会在多个线程对同一个对象中的实例变量进行并发访问时发生，产生的后果就是“脏读”，就是取到的数据其实是被更改过的。

“线程安全”已获得的实例变量的值是经过同步处理的，不会出现脏读现象。

### 2.1.1、方法内的变量为线程安全

“非线程安全”的问题存在于“实例变量”中。如果是方法内部的私有变量，则不存在“非线程安全”，即线程安全。

方法中的变量不存在非线程安全的问题，这是由于方法内部的变量时私有的特性造成的。

### 2.1.2、实例变量非线程安全

如果多个线程共同访问一个对象中的实例变量，则有可能出现“非线程安全”问题。

如果线程访问的对象中有多个实例变量，则运行结果可能出现交叉的情况。如果对象只有一个实例变量则有可能出现覆盖的情况。

在多个线程访问对象中的同步方法时一定是线程安全的。

### 2.1.3、多个对象多个锁

关键字synchronized取得的锁都是“对象锁”，而不是把一段代码或方法（函数）当做锁；所以无论哪个线程先执行带有synchronized关键字的方法，哪个线程就持有该方法所属对象的锁lock，那么其他线程就只能呈等待状态，前提是多个线程访问的是同一个对象。

如果多个线程访问多个对象，则JVM会创建多个锁。

同步的单词是synchronized，异步的单词是asynchronized.

# 线程池原理

1，线程池中的线程初始化

默认情况下，创建线程时线程池中是没有线程的，需要提交线程之后才会创建出现成。但可以根据实际需要在创建线程池之后，可以根据需要创建线程，可以通过两个方法：prestartCoreThread方法与prestartAllThreads方法创建线程。

prestartCoreThread：初始化一个核心线程

prestartAllThreads：初始化所有核心线程

2，任务缓存队列及排队策略

在前面我们多次提到了任务缓存队列，即workQueue，它用来存放等待执行的任务。

几种创建线程池的方式会默认任务队列

workQueue的类型为BlockingQueue<Runnable>，通常可以取下面三种类型：

1）ArrayBlockingQueue：基于数组的先进先出队列，此队列创建时必须指定大小；

2）LinkedBlockingQueue：基于链表的先进先出队列，如果创建时没有指定此队列大小，则默认为Integer.MAX\_VALUE；

3）synchronousQueue：这个队列比较特殊，它不会保存提交的任务，而是将直接新建一个线程来执行新来的任务。

3，拒绝策略（四种）

当线程池的任务缓存队列已满且线程池中的数目达到最大的maximumPoolSize，如果还有任务就会采用拒绝策略，通常有四种策略：

ThreadPoolExecutor.AbortPolicy:丢弃任务并抛出RejectedExecutionException异常。

ThreadPoolExecutor.DiscardPolicy：也是丢弃任务，但是不抛出异常。

ThreadPoolExecutor.DiscardOldestPolicy：丢弃队列最前面的任务，然后重新尝试执行任务（重复此过程）

ThreadPoolExecutor.CallerRunsPolicy：由调用线程处理该任务

4，增长策略

当一个任务通过execute(Runnable)方法欲添加到线程池时：

l  如果此时线程池中的数量小于corePoolSize，即使线程池中的线程都处于空闲状态，也要创建新的线程来处理被添加的任务。

2  如果此时线程池中的数量等于 corePoolSize，但是缓冲队列 workQueue未满，那么任务被放入缓冲队列。

3  如果此时线程池中的数量大于corePoolSize，缓冲队列workQueue满，并且线程池中的数量小于maximumPoolSize，建新的线程来处理被添加的任务。

4  如果此时线程池中的数量大于corePoolSize，缓冲队列workQueue满，并且线程池中的数量等于maximumPoolSize，那么通过 handler所指定的策略来处理此任务。也就是：处理任务的优先级为：核心线程corePoolSize、任务队列workQueue、最大线程maximumPoolSize，如果三者都满了，使用handler处理被拒绝的任务。

当线程池中的线程数量大于 corePoolSize时，如果某线程空闲时间超过keepAliveTime，线程将被终止。这样，线程池可以动态的调整池中的线程数。

5，如何合理配置线程池的大小？