|  |
| --- |
| 1、进程和线程  一个程序至少有一个进程，一个进程至少有一个线程。  在java中，每次程序运行至少启动2个线程。一个是main线程，一个是垃圾收集线程。  多个进程可以并发执行，数据资源是独立，不共享，进程崩溃互不影响。  多个线程可以并发执行，数据资源可以共享（也就出现了同步的问题）。  进程和线程就不是可以用来比较的概念，其实不是一个层面的东西。  2.多线程并发——线程安全与同步  线程安全：经常用来描绘一段代码。指在并发的情况之下，该代码经过多线程使用，线程的调度顺序不影响任何结果。  同步：Java中的同步指的是通过人为的控制和调度，保证共享资源的多线程访问成为线程安全，来保证结果的准确。  3、线程的状态：  初始状态（NEW，新建状态）  就绪状态（READY）  运行状态（RUNNING）  阻塞状态(BLOCKED)  终止状态（TERMINATED）  4、线程的创建  Java使用Thread类代表线程，所有的线程对象都必须是Thread类或其子类的实例。Java可以用四种方式来创建线程，如下所示：  1）继承Thread类创建线程  2）实现Runnable接口创建线程  3）使用Callable和Future创建线程  和Runnable接口不一样，Callable接口提供了一个call（）方法作为线程执行体，call()方法比run()方法功能要强大。call()方法可以有返回值，call()方法可以声明抛出异常。  4）使用线程池例如用Executor框架  5、守护线程，它有一个特征——如果所有前台线程都死亡，后台线程自动死亡。  通过setDaemon(true)来设置一个线程。  6、线程同步  1）同步函数  用synchronize关键字修饰的方法。因为每个java对象都有一个内置锁，当用synchronize关键字修饰方法时内置锁会保护整个方法，而在调用该方法之前，要先获得内置锁，否则就会处于阻塞状态。  2）同步代码块  用synchronize关键字修饰的语句块，被该关键字修饰的语句块会自动被加上内置锁，从而实现同步。  3）静态同步方法  如果同步函数被静态修饰之后，使用的锁是类名.class，是该方法所在类的字节码文件对象。  7、死锁：多重锁才会出现死锁，锁中锁，导致相互取取不到另一个锁。  8、start()与run()  用start()来启动线程，实现了真正意义上的启动线程，此时会出现异步执行的效果，即在线程的创建和启动中所述的随机性。  如果使用run()来启动线程，就不是异步执行了，而是同步执行，不会达到使用线程的意义。  9、volatile是轻量级的synchronized，在正确使用的前提下，它可以达到与synchronized一样的线程安全的语义，而且不会带来线程切换的开销。  volatile的作用是什么？  volatile保证了共享变量的“可见性”。可见性的意思是当一个线程修改一个共享变量时，另外一个线程能读到这个修改的值。它在某些情况下比synchronized的开销更小。  10、线程间通信  A.多个线程通过synchronized关键字这种方式来实现线程间的通信。既然共享了同一个数据资源，也就是通信啰。  B.android线程间通信比较常用的Handler。  11、线程池  ExecutorService是Java提供的用于管理线程池的类。  Java通过Executors提供四种线程池，分别为：  newCachedThreadPool：用来创建一个可以无限扩大的线程池，适用于负载较轻的场景，执行短期异步任务。可缓存线程池，先查看池中有没有以前建立的线程，如果有，就直接使用。如果没有，就建一个新的线程加入池中，缓存型池子通常用于执行一些生存期很短的异步型任务。空闲线程超过60s就销毁。  ExecutorService cachedThreadPool = Executors.newCachedThreadPool();  cachedThreadPool.execute(new Runnable(){……});  newFixedThreadPool 创建一个固定大小的线程池，超出的线程会在队列中等待。因为采用无界的阻塞队列，所以实际线程数量永远不会变化，适用于负载较重的场景，对当前线程数量进行限制。（保证线程数可控，不会造成线程过多，导致系统负载更为严重）。  ExecutorService fixedThreadPool = Executors.newFixedThreadPool(3);  fixedThreadPool.execute(new Runnable() {……});  newScheduledThreadPool 创建一个定长线程池，支持定时及周期性任务执行，适用于执行延时或者周期性任务。  ScheduledExecutorService scheduledThreadPool = Executors.newScheduledThreadPool(5);  scheduledThreadPool.schedule(new Runnable() {  public void run() {  System.out.println("delay 3 seconds");  }  }, 3, TimeUnit.SECONDS);  newSingleThreadExecutor 创建一个单线程的线程池，适用于需要保证顺序执行各个任务。  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。  ExecutorService singleThreadExecutor = Executors.newSingleThreadExecutor();  singleThreadExecutor.execute(new Runnable() { ……}); |

# 多线程是什么？线程与进程？

## 1.1 进程与线程

进程：是一个正在执行中的程序，每一个进程执行都有一个执行顺序，该顺序是一个执行路径，或者叫一个控制单元；

线程：就是进程中的一个独立控制单元。一个进程中至少有一个线程。

多线程：指这个程序（一个进程）运行时产生了不止一个线程。

线程是指进程内的一个执行单元，也是进程内的可调度实体。

1. 地址空间：线程是进程内的一个执行单元；进程至少有一个线程；它们共享进程的地址空间（即内存单元）；而进程有自己独立的地址空间，共享数据只能通过进程间通信或储存介质。

所以，一个进程崩溃后，在保护模式下不会对其它进程产生影响。

线程有自己的堆栈和局部变量，但线程之间没有单独的地址空间，一个线程死掉就等于整个进程死掉。

所以多进程的程序要比多线程的程序健壮，但在进程切换时，耗费资源较大，效率要差一些。但对于一些要求同时进行并且又要共享某些变量的并发操作，只能用多线程，不能用进程。

1. 系统资源拥有：进程是系统进行资源分配和调度的一个独立单位。同一个进程内的线程共享进程的资源。

线程是进程的一个实体，是CPU调度和分派的基本单位，它是比进程更小的能独立运行的基本单位。线程自己基本上不拥有系统资源，只拥有一点在运行中必不可少的资源(如程序计数器，一组寄存器和栈)，但是它可与同属一个进程的其他的线程共享进程所拥有的全部资源。

一个线程可以创建和撤销另一个线程；同一个进程中的多个线程之间可以并发执行。

（3）二者均可并发执行。

简而言之，一个程序至少有一个进程，一个进程至少有一个线程。

线程的划分尺度小于进程，使得多线程程序的并发性高。

每个独立的线程有一个程序运行的入口、顺序执行序列和程序的出口。但是线程不能够独立执行，必须依存在应用程序中，由应用程序提供多线程执行控制。

（4）从逻辑角度来看，多线程的意义在于一个应用程序中，有多个执行部分可以同时执行。但操作系统并没有将多个线程看做多个独立的应用，来实现进程的调度和管理以及资源分配。这就是进程和线程的重要区别。

提醒一下大家：main方法其实也是一个线程。在java中所以的线程都是同时启动的，至于什么时候，哪个先执行，完全看谁先得到CPU的资源。

在java中，每次程序运行至少启动2个线程。一个是main线程，一个是垃圾收集线程。因为每当使用java命令执行一个类的时候，实际上都会启动一个JVM，每一个JVM实例在就是在操作系统中启动了一个进程。

## 1.2 为什么要用多线程

（1）为了更好的利用cpu的资源，如果只有一个线程，则第二个任务必须等到第一个任务结束后才能进行，如果使用多线程则在主线程执行任务的同时可以执行其他任务，而不需要等待；

（2）进程之间不能共享数据，线程可以；

（3）系统创建进程需要为该进程重新分配系统资源，创建线程代价比较小；

（4）Java语言内置了多线程功能支持，简化了java多线程编程。

用多线程只有一个目的，那就是更好的利用cpu的资源，因为所有的多线程代码都可以用单线程来实现。说这个话其实只有一半对，因为反应“多角色”的程序代码，最起码每个角色要给他一个线程吧，否则连实际场景都无法模拟，当然也没法说能用单线程来实现：比如最常见的“生产者，消费者模型”。

总结：利用CPU资源更加的灵活性；满足模拟现实生活的多角色多线程场景。

## 1.3 并行与并发

并行：多个cpu实例或者多台机器同时执行一段处理逻辑，是真正的同时。

并发：通过cpu调度算法，让用户看上去同时执行，实际上从cpu操作层面不是真正的同时。并发往往在场景中有公用的资源，那么针对这个公用的资源往往产生瓶颈，我们会用TPS或者QPS来反应这个系统的处理能力。

在单处理机系统中，每一时刻却仅能有一道程序执行。只有多处理器系统才能实现并行处理。常见的window、手机系统都是可以多个程序同时运行，也就是多进程系统。建议阅读《现代操作系统》或者《操作系统的设计与实现》。

真八核就是指一个中央处理器中，有8个小的处理器。现在安卓手机的CPU处理器基本都是八核心了。但是其实有很大一部分都是伪八核。比如现在我现在用的电脑是4核的。

## 1.4 多线程并发——线程安全与同步

线程安全：经常用来描绘一段代码。指在并发的情况之下，该代码经过多线程使用，线程的调度顺序不影响任何结果。这个时候使用多线程，我们只需要关注系统的内存，cpu是不是够用即可。反过来，线程不安全就意味着线程的调度顺序会影响最终结果。

而，在多线程编程中，由于线程的调度顺序可以认为是随机的。所以，对于共享资源数据的线程安全就显得尤为重要。也就，提出了同步处理。

同步：Java中的同步指的是通过人为的控制和调度，保证共享资源的多线程访问成为线程安全，来保证结果的准确。如上面的代码简单加入@synchronized关键字。在保证结果准确的同时，提高性能，才是优秀的程序。线程安全的优先级高于性能。

# 2、线程的生命周期：

（1）初始状态（NEW，新建状态） ：从新建一个线程对象到程序start() 这个线程之间的状态，都是新建状态；

（2）就绪状态（READY） ：线程对象调用start()方法后，就处于就绪状态，等到JVM里的线程调度器的调度；或者说：线程对象创建后，其他线程(比如main线程）调用了该对象的start()方法。该状态的线程位于可运行线程池中，等待被线程调度选中，获取CPU的使用权，此时处于就绪状态（ready）。

（3）运行状态（RUNNING） ：就绪状态下的线程在获取CPU资源后就可以执行run(),此时的线程便处于运行状态，运行状态的线程可变为就绪、阻塞及死亡三种状态。

（4）阻塞状态(BLOCKED)：阻塞状态是指线程因为某种原因放弃了cpu 使用权，也即让出了cpu timeslice，暂时停止运行。直到线程进入可运行(runnable)状态，才有机会再次获得cpu timeslice 转到运行(running)状态。阻塞的情况分三种：

（4-1）等待阻塞：运行(running)的线程执行o.wait()方法，JVM会把该线程放入等待队列(waitting queue)中。

（4-2）同步阻塞：运行(running)的线程在获取对象的同步锁时，若该同步锁被别的线程占用，则JVM会把该线程放入锁池(lock pool)中。

（4-2）其他阻塞：运行(running)的线程执行Thread.sleep(long ms)或t.join()方法，或者发出了I/O请求时，JVM会把该线程置为阻塞状态。当sleep()状态超时、join()等待线程终止或者超时、或者I/O处理完毕时，线程重新转入可运行(runnable)状态。

（5）终止状态（TERMINATED） ：run（）方法完成后，即线程执行完毕或发生其他终止条件时就会切换到终止状态。



## ****2.1 初始状态****

实现Runnable接口和继承Thread可以得到一个线程类，new一个实例出来，线程就进入了初始状态

## ****2.2 就绪状态****

（1）可运行状态只是说你资格运行，调度程序没有挑选到你，你就永远是可运行状态。

（2）调用线程的start()方法，此线程进入可运行状态。

（3）当前线程sleep()方法结束，其他线程join()结束，等待用户输入完毕，某个线程拿到对象锁，这些线程也将进入可运行状态。

（4）当前线程时间片用完了，调用当前线程的yield()方法，当前线程进入可运行状态。

（5）锁池里的线程拿到对象锁后，进入可运行状态。

## ****2.3 运行状态****

线程调度程序从可运行池中选择一个线程作为当前线程时线程所处的状态。

## ****2.4 死亡状态****

当线程的run()方法完成时，或者主线程的main()方法完成时，我们就认为它死去。这个线程对象也许是活的，但是，它已经不是一个单独执行的线程。线程一旦死亡，就不能复生。

在一个死去的线程上调用start()方法，会抛出java.lang.IllegalThreadStateException异常。

## ****2.5阻塞状态****

（1）当前线程T调用Thread.sleep()方法，当前线程进入阻塞状态。

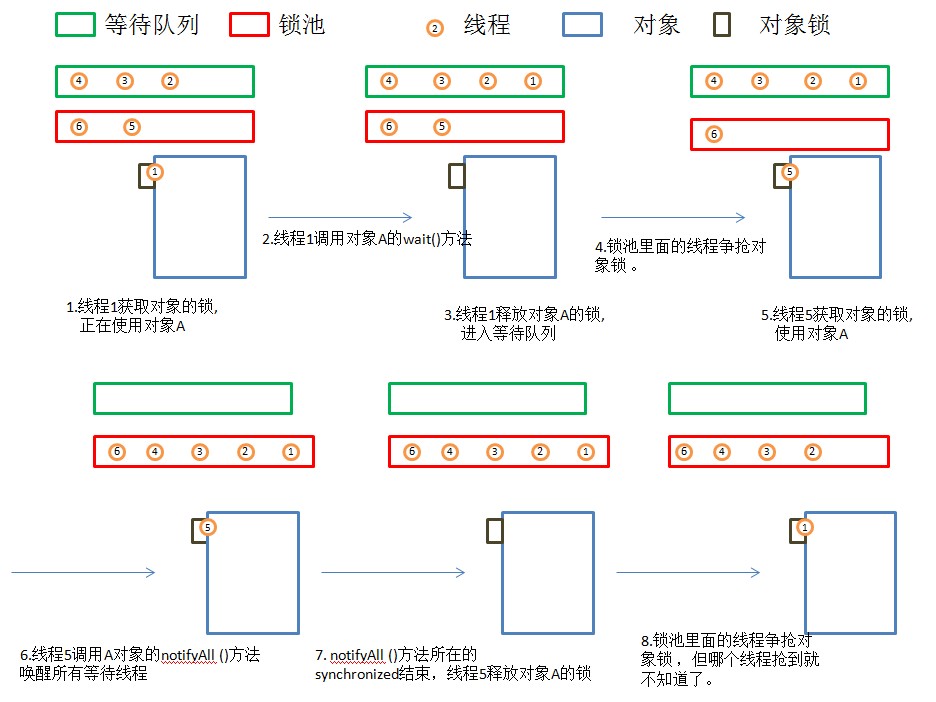
（2）运行在当前线程里的其它线程t2调用join()方法，当前线程进入阻塞状态。

（3）等待用户输入的时候，当前线程进入阻塞状态。

## ****2.6 等待队列****

**（1）本是Object里的方法，但影响了线程。**调用obj的wait(), notify()方法前，必须获得obj锁，也就是必须写在synchronized(obj) 代码段内。

（2）与等待队列相关的步骤和图



## ****2.7 锁池状态****

（1）当前线程想调用对象A的同步方法时，发现对象A的锁被别的线程占有，此时当前线程进入锁池状态。简言之，锁池里面放的都是想争夺对象锁的线程。

（2）当一个线程1被另外一个线程2唤醒时，1线程进入锁池状态，去争夺对象锁。

（3）锁池是在同步的环境下才有的概念，一个对象对应一个锁池。

# 3、线程的创建

Java使用Thread类代表线程，所有的线程对象都必须是Thread类或其子类的实例。Java可以用四种方式来创建线程，如下所示：

1）继承Thread类创建线程

2）实现Runnable接口创建线程

3）使用Callable和Future创建线程

4）使用线程池例如用Executor框架

## 3.1 继承Thread类创建线程

通过继承Thread类来创建并启动多线程的一般步骤如下：

1）定义Thread类的子类，并重写该类的run()方法，该方法的方法体就是线程需要完成的任务，run()方法也称为线程执行体。

2）创建Thread子类的实例，也就是创建了线程对象。

3）启动线程，即调用线程的start()方法。

代码实例

public class MyThread extends Thread{//继承Thread类

　　public void run(){

　　//重写run方法

　　}

}

public class Main {

　　public static void main(String[] args){

　　　　new MyThread().start();//创建并启动线程

　　}

}

## 3.2 实现Runnable接口创建线程

通过实现Runnable接口创建并启动线程一般步骤如下：

1）定义Runnable接口的实现类，一样要重写run()方法，这个run（）方法和Thread中的run()方法一样是线程的执行体。

2）创建Runnable实现类的实例，并用这个实例作为Thread的target来创建Thread对象，这个Thread对象才是真正的线程对象。

1. 第三部依然是通过调用线程对象的start()方法来启动线程。

代码实例：

public class MyThread2 implements Runnable {//实现Runnable接口

　　public void run(){

　　//重写run方法

　　}

}

public class Main {

　　public static void main(String[] args){

　　　　//创建并启动线程

　　　　MyThread2 myThread=new MyThread2();

　　　　Thread thread=new Thread(myThread);

　　　　thread().start();

　　　　//或者    new Thread(new MyThread2()).start();

　　}

}

## 3.3 使用Callable和Future创建线程

和Runnable接口不一样，Callable接口提供了一个call（）方法作为线程执行体，call()方法比run()方法功能要强大。

——>call()方法可以有返回值

——>call()方法可以声明抛出异常

Java5提供了Future接口来代表Callable接口里call()方法的返回值，并且为Future接口提供了一个实现类FutureTask，这个实现类既实现了Future接口，还实现了Runnable接口，因此可以作为Thread类的target。在Future接口里定义了几个公共方法来控制它关联的Callable任务。

——>boolean cancel(boolean mayInterruptIfRunning)：视图取消该Future里面关联的Callable任务。

——>V get()：返回Callable里call（）方法的返回值，调用这个方法会导致程序阻塞，必须等到子线程结束后才会得到返回值。

——>V get(long timeout,TimeUnit unit)：返回Callable里call（）方法的返回值，最多阻塞timeout时间，经过指定时间没有返回抛出TimeoutException。

——>boolean isDone()：若Callable任务完成，返回True。

——>boolean isCancelled()：如果在Callable任务正常完成前被取消，返回True。

介绍了相关的概念之后，创建并启动有返回值的线程的步骤如下：

1）创建Callable接口的实现类，并实现call()方法，然后创建该实现类的实例（从java8开始可以直接使用Lambda表达式创建Callable对象）。

2）使用FutureTask类来包装Callable对象，该FutureTask对象封装了Callable对象的call()方法的返回值。

3）使用FutureTask对象作为Thread对象的target创建并启动线程（因为FutureTask实现了Runnable接口）。

4）调用FutureTask对象的get()方法来获得子线程执行结束后的返回值。

代码实例：

public class MyThread implements Callable<String> {

private int count = 20;

@Override

public String call() throws Exception {

for (int i = count; i > 0; i--) {

// Thread.yield();

System.out.println(Thread.currentThread().getName()+"当前票数：" + i);

}

return "sale out";

}

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

Callable<String> callable = new MyThread();

FutureTask <String>futureTask=new FutureTask<>(callable);

Thread mThread=new Thread(futureTask);

Thread mThread2=new Thread(futureTask);

Thread mThread3=new Thread(futureTask);

// mThread.setName("hhh");

mThread.start();

mThread2.start();

mThread3.start();

System.out.println(futureTask.get());

}

}

## 3.4 各种方法比较

（1）继承Thread：线程代码存放在Thread子类run方法中。

优势：编写简单，可直接用this.getname（）获取当前线程，不必使用Thread.currentThread()方法。

劣势：已经继承了Thread类，无法再继承其他类。

（2）实现Runnable：线程代码存放在接口的子类的run方法中。

优势：避免了单继承的局限性、多个线程可以共享一个target对象，非常适合多线程处理同一份资源的情形。

劣势：比较复杂、访问线程必须使用Thread.currentThread（）方法、无返回值。

（3）实现Callable：

优势：有返回值、避免了单继承的局限性、多个线程可以共享一个target对象，非常适合多线程处理同一份资源的情形。

劣势：比较复杂、访问线程必须使用Thread.currentThread（）方法。

　　建议使用实现接口的方式创建多线程。

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。

# 线程状态管理

## 4.1 线程睡眠---sleep()

线程睡眠的原因：线程执行的太快，或需要强制执行到下一个线程。

线程睡眠的方法（两个）：sleep（long millis）在指定的毫秒数内让正在执行的线程休眠。

　　　　　　　　　　　　　sleep（long millis，int nanos）在指定的毫秒数加指定的纳秒数内让正在执行的线程休眠。

代码演示：

public class SynTest {

public static void main(String[] args) {

new Thread(new CountDown(),"倒计时").start();

}

}

class CountDown implements Runnable{

int time = 10;

public void run() {

while (true) {

if(time>=0){

System.out.println(Thread.currentThread().getName() + ":" + time--);

try {

Thread.sleep(1000); //睡眠时间为1秒

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}

}

复制代码

每隔一秒则会打印一次，打印结果为：

复制代码

倒计时:10

倒计时:9

倒计时:8

倒计时:7

倒计时:6

倒计时:5

倒计时:4

倒计时:3

倒计时:2

倒计时:1

倒计时:0

　　扩展：Java线程调度是Java多线程的核心，只有良好的调度，才能充分发挥系统的性能，提高程序的执行效率。但是不管程序员怎么编写调度，只能最大限度的影响线程执行的次序，而不能做到精准控制。因为使用sleep方法之后，线程是进入阻塞状态的，只有当睡眠的时间结束，才会重新进入到就绪状态，而就绪状态进入到运行状态，是由系统控制的，我们不可能精准的去干涉它，所以如果调用Thread.sleep(1000)使得线程睡眠1秒，可能结果会大于1秒。

## 4.2 线程让步---yield()暂停

该方法和sleep方法类似，也是Thread类提供的一个静态方法，可以让正在执行的线程暂停，但是不会进入阻塞状态，而是直接进入就绪状态。相当于只是将当前线程暂停一下，然后重新进入就绪的线程池中，让线程调度器重新调度一次。也会出现某个线程调用yield方法后暂停，但之后调度器又将其调度出来重新进入到运行状态。

public class SynTest {

public static void main(String[] args) {

yieldDemo ms = new yieldDemo();

Thread t1 = new Thread(ms,"张三吃完还剩");

Thread t2 = new Thread(ms,"李四吃完还剩");

Thread t3 = new Thread(ms,"王五吃完还剩");

t1.start();

t2.start();

t3.start();

}

}

class yieldDemo implements Runnable{

int count = 20;

public void run() {

while (true) {

if(count>0){

System.out.println(Thread.currentThread().getName() + count-- + "个瓜");

if(count % 2 == 0){

Thread.yield(); //线程让步

}

}

}

}

}

sleep和yield的区别：

　　①、sleep方法声明抛出InterruptedException，调用该方法需要捕获该异常。yield没有声明异常，也无需捕获。

　　②、sleep方法暂停当前线程后，会进入阻塞状态，只有当睡眠时间到了，才会转入就绪状态。而yield方法调用后 ，是直接进入就绪状态。

## 4.3 线程合并---join() 临时加入

当B线程执行到了A线程的.join（）方法时，B线程就会等待，等A线程都执行完毕，B线程才会执行。

　　join可以用来临时加入线程执行。

以下为代码演示：

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

yieldDemo ms = new yieldDemo();

Thread t1 = new Thread(ms,"张三吃完还剩");

Thread t2 = new Thread(ms,"李四吃完还剩");

Thread t3 = new Thread(ms,"王五吃完还剩");

t1.start();

t1.join();

t2.start();

t3.start();

System.out.println( "主线程");

}

## 4.4 停止线程

　　原stop方法因有缺陷已经停用了，那么现在改如何停止线程？现在分享一种，就是让run方法结束。

开启多线程运行，运行的代码通常是循环结构，只要控制住循环，就可以让run方法结束，也就是线程结束。

　 具体代码如下：

1 public class StopThread {

2

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

4 int num = 0;

5 StopTh st = new StopTh();

6 Thread t1 = new Thread(st);

7 Thread t2 = new Thread(st);

8 t1.start();

9 t2.start();

10 //设置主线程执行50次，执行结束之后停止线程

11 while (true) {

12 if(num++ == 50){

13 st.flagChange();

14 break;

15 }

16 System.out.println(Thread.currentThread().getName() + "..." + num);

17 }

18 }

19 }

20

21 class StopTh implements Runnable{

22

23 private boolean flag = true;

24 public void run() {

25 while(flag){

26 System.out.println(Thread.currentThread().getName() + "stop run" );

27 }

28 }

29 public void flagChange(){

30 flag = false;

31 }

32 }

特殊情况：当线程处于了冻结状态，就不会读取到标记，也就不会结束。当没有指定方法让冻结的线程回复到运行状态时，我们需要对冻结状态进行清除，也就是强制让线程恢复到运行状态中来，这样可就可以操作标记让线程结束。

Thread类提供该方法： interrupt（）；（如果线程在调用Object类的wait（）、wait（long）、wait（long，int）方法，或者该类的join（）、join（long）、join（long、int）、sleep（long）或sleep（long、int）方法过程中受阻，则其中断状态将被清除，还将收到一个InterruptedException。）

## 4.5 设置优先级

每个线程执行时都有一个优先级的属性，优先级高的线程可以获得较多的执行机会，而优先级低的线程则获得较少的执行机会。与线程休眠类似，线程的优先级仍然无法保障线程的执行次序。只不过，优先级高的线程获取CPU资源的概率较大，优先级低的也并非没机会执行。

Thread类中提供了优先级的三个常量，代码如下：

MAX\_PRIORITY =10

MIN\_PRIORITY =1

NORM\_PRIORITY =5

------------------------------------------------------

ThreadDemo td = new ThreadDemo();

Thread t1 = new Thread(td,"张三");

t1.priority(9);　　　　　　　　　　　　//设置优先级

t1.start();　　　　　　　　　　　　　　//设置完毕

## 4.6 守护线程

java 中有两种线程：用户线程和守护线程。可以通过isDaemon()方法来区别它们：如果返回false，则说明该线程是“用户线程”；否则就是“守护线程”。

用户线程一般用户执行用户级任务，而守护线程也就是“后台线程”，一般用来执行后台任务。需要注意的是：Java虚拟机在“用户线程”都结束后会后退出。

守护线程又称“后台线程”、“精灵线程”，它有一个特征——如果所有前台线程都死亡，后台线程自动死亡。

通过setDaemon(true)来设置一个线程。

# 5、线程同步的使用

为什么要进行线程同步？

　　java允许多线程并发控制，当多个线程同时操作一个可共享资源变量时（如对其进行增删改查操作），会导致数据不准确，而且相互之间产生冲突。所以加入同步锁以避免该线程在没有完成操作前被其他线程调用，从而保证该变量的唯一性和准确性。

同步的理解

我在方法声明的时候前面加一个synchronized关键字，public synchronized void method(){ },它代表的意思是在执行这个方法的时候当前对象被锁定起来。

Java中的每个对象都有一个lock，当访问某个对象的synchronized方法时，该对象就会被上锁（注意，是对象，不是方法，假如你在这个类中定义了多个方法，如果你的线程访问到了其中的任意一个synchronized方法，那么其它的就暂时不能被访问了，必须等到该对象被解锁以后，即方法执行结束才行）。解锁的意思是值线程执行该方法完毕，或者说过程中抛出了异常。

再换一种说法，就是，一个类中有synchronized方法，如果该类的对象的该方法被访问时，那么整个该对象都被锁定了，但是这个意思是其它非synchronized方法和成员变量还是可以被访问，注意区分这一点。因为synchronized方法会锁定对象，所以一旦有一个synchronized方法被某个线程启动了，那么对象已经被独占了，其它的synchronized方法就不能再同时独占对象了，但是普通方法和成员变量并不独占对象，所以仍然可以被调用。

需要注意的是，如果同步方法里面有sleep方法，它仍然是同步方法的一部分，在它被执行的过程中，锁仍然不会被解开。

其实同步的意思就是上锁，同步方法，进而达到对象上锁的目的。假如有一个数据库，有读和修改2个方法，你可以允许多个线程同时读，但是你不能让多个线程同时改，所以说改的方法要同步，读的方法不需要。(其实这里我更加觉得应该同步的不是方法，而是数据本身，只要有对象访问对象，对象就应该被锁定，避免读的时候有对象要修改，修改的时候有对象要读，甚至是多个对象同时都想改)。还有，如果2个方法都修改了同一个值的话，那么2个方法都应该加同步。

线程同步我觉得是这样的，你说概念吧，也还不难理解，我觉得真正难的是实际中的应用，你必须考虑很多相关的问题，哪一个方法要同步，都需要好好琢磨。

## 5.1 同步方法1--同步函数

同步函数：就是用synchronize关键字修饰的方法。因为每个java对象都有一个内置锁，当用synchronize关键字修饰方法时内置锁会保护整个方法，而在调用该方法之前，要先获得内置锁，否则就会处于阻塞状态。

　代码演示：

public synchronized void run() {

　　同步方法2：

## 5.2 同步代码块

就是拥有synchronize关键字修饰的语句块，被该关键字修饰的语句块会自动被加上内置锁，从而实现同步。

代码演示：

public void run() {

while(true){

synchronized (this) { //同步代码块

if(tick>0){

try {

Thread.sleep(10); //执行中让线程睡眠10毫秒，

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println(Thread.currentThread().getName() + " " + tick--);

}

}

}

}

## 5.3 静态同步方法

　　追加问题：如果同步函数被静态修饰之后，使用的锁是什么？静态方法中不能定义this！

　　静态内存是：内存中没有本类对象，但是一定有该类对应的字节码文件对象。 类名.class 该对象类型是Class。

　　所以静态的同步方法使用的锁是该方法所在类的字节码文件对象。 类名.class。代码如下：

public static mySyn(String name){

synchronized (Xxx.class) {

Xxx.name = name;

}

}

## 5.4 总结

　　同步的前提：

　　1、必须要有两个或者两个以上的线程。

　　2、必须是多个线程使用同一个锁。

　　3、必须保证同步中只能有一个线程在运行。

　　4、只能同步方法，不能同步变量和类。

　　5、不必同步类中所有方法，类可以拥有同步和非同步的方法。

　　6、如果一个线程在对象上获得一个锁，就没有任何其他线程可以进入（该对象的）类中的任何一个同步方法。

　　7、线程睡眠时，它所持的任何锁都不会释放。

　　好处：解决了多线程的安全问题。

　　弊端：多个线程需要判断，消耗资源，降低效率。

　　如何找问题？

　　1、明确哪些代码是多线程运行代码。

　　2、明确共享数据。

3、明确多线程运行代码中哪些语句是操作共享数据的。

## 5.5 死锁

进程A中包含资源A,进程B中包含资源B，A的下一步需要资源B，B的下一步需要资源A，所以它们就互相等待对方占有的资源释放，所以也就产生了一个循环等待死锁。

1 public class DeadLock {

2

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

4 Thread t1 = new Thread(new DeadLockTest(true));

5 Thread t2 = new Thread(new DeadLockTest(false));

6 t1.start();

7 t2.start();

8 }

9 }

10

11 class DeadLockTest implements Runnable{

12

13 private boolean flag;

14 static Object obj1 = new Object();

15 static Object obj2 = new Object();

16 public DeadLockTest(boolean flag) {

17 this.flag = flag;

18 }

19 public void run(){

20 if(flag){

21 synchronized(obj1){

22 System.out.println("if lock1");

23 synchronized (obj2) {

24 System.out.println("if lock2");

25 }

26 }

27 }else{

28 synchronized (obj2) {

29 System.out.println("else lock2");

30 synchronized (obj1) {

31 System.out.println("else lock1");

32 }

33 }

34 }

35 }

36 }

死锁形成的必要条件总结（都满足之后就会产生）：

①、互斥条件：资源不能被共享，只能被同一个进程使用；

②、请求与保持条件：已经得到资源的进程可以申请新的资源；

③、非剥夺条件：已经分配的资源不能从相应的进程中强制剥夺；

④、循环等待条件：系统中若干进程形成环路，该环路中每个进程都在等待相邻进程占用的资源。

# 6、start() 与 run()方法比较

从运行结果肯明显可以看出，使用start()方法具有异步执行的效果，而使用run()方法是同步执行的效果，运行结果中规中矩。使用start()方法，是真的启动了相应的线程0-9，而使用run()方法并没有真的启动线程，而是由一个叫main的主线程去调用的run()方法。所以，正确启动线程的方式是使用start()方法。

现在来看一下Runnable接口，根据java api，Runnable接口只有一个方法，就是run()。

“当通过实现Runnable接口来创建线程时，启动线程会使得run()方法在那个独立执行的线程中被调用。”

根据java api，start()方法是Thread的类方法。

“start()方法会使得该线程开始执行；java虚拟机会去调用该线程的run()方法。”

因此，t.start()会导致run()方法被调用，run()方法中的内容称为线程体，它就是这个线程需要执行的工作。

用start()来启动线程，实现了真正意义上的启动线程，此时会出现异步执行的效果，即在线程的创建和启动中所述的随机性。而如果使用run()来启动线程，就不是异步执行了，而是同步执行，不会达到使用线程的意义。

# 7、常见线程名词

主线程：JVM调用程序main()所产生的线程。

当前线程：这个是容易混淆的概念。一般指通过Thread.currentThread()来获取的进程。

后台线程：指为其他线程提供服务的线程，也称为守护线程。JVM的垃圾回收线程就是一个后台线程。用户线程和守护线程的区别在于，是否等待主线程依赖于主线程结束而结束

前台线程：是指接受后台线程服务的线程，其实前台后台线程是联系在一起，就像傀儡和幕后操纵者一样的关系。傀儡是前台线程、幕后操纵者是后台线程。由前台线程创建的线程默认也是前台线程。可以通过isDaemon()和setDaemon()方法来判断和设置一个线程是否为后台线程。

众所周知，无限制下多线程操作共享变量是危险的，为了保证线程安全语义，一般的建议是在操作共享变量时加锁，比方说在用synchronized关键字修饰的方法内读写共享变量。

但是synchronized开销较大，有没有更轻量更优雅的解决方案呢？

volatile是轻量级的synchronized，在正确使用的前提下，它可以达到与synchronized一样的线程安全的语义，而且不会带来线程切换的开销。

volatile的作用是什么？

***volatile保证了共享变量的“可见性”。可见性的意思是当一个线程修改一个共享变量时，另外一个线程能读到这个修改的值。它在某些情况下比synchronized的开销更小。***

# 线程间通信

# 线程池的使用

## 9.1 线程池的原理

多线程技术主要解决处理器单元内多个线程执行的问题，它可以显著减少处理器单元的闲置时间，增加处理器单元的吞吐能力。

假设一个服务器完成一项任务所需时间为：T1 创建线程时间，T2 在线程中执行任务的时间，T3 销毁线程时间。

如果：T1 + T3 远大于 T2，则可以采用线程池，以提高服务器性能。

一个线程池包括以下四个基本组成部分：

1、线程池管理器（ThreadPool）：用于创建并管理线程池，包括 创建线程池，销毁线程池，添加新任务；

2、工作线程（PoolWorker）：线程池中线程，在没有任务时处于等待状态，可以循环的执行任务；

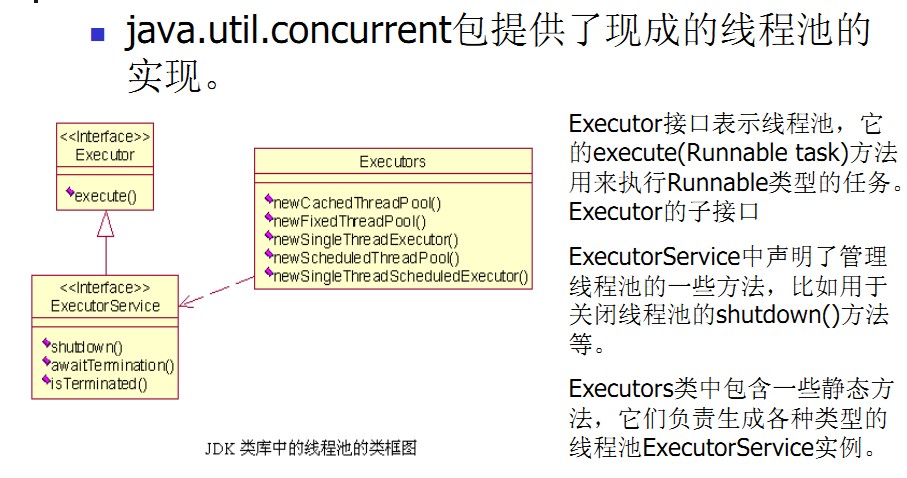
3、任务接口（Task）：每个任务必须实现的接口，以供工作线程调度任务的执行，它主要规定了任务的入口，任务执行完后的收尾工作，任务的执行状态等；

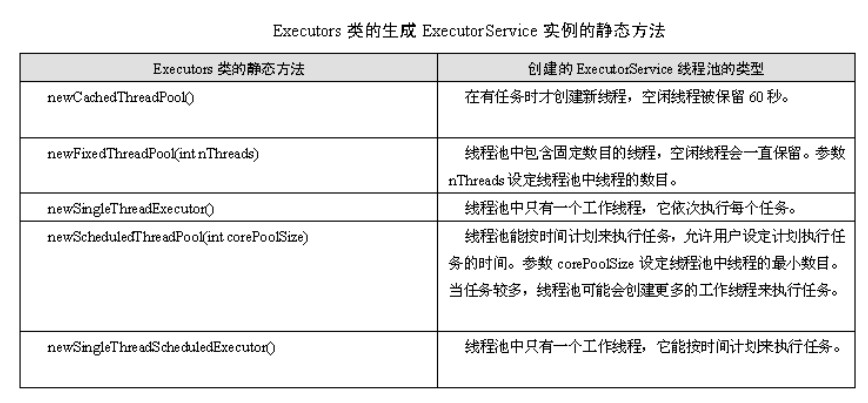
4、任务队列（taskQueue）：用于存放没有处理的任务。提供一种缓冲机制。

线程池技术正是关注如何缩短或调整T1,T3时间的技术，从而提高服务器程序性能的。它把T1，T3分别安排在服务器程序的启动和结束的时间段或者一些空闲的时间段，这样在服务器程序处理客户请求时，不会有T1，T3的开销了。

线程池不仅调整T1,T3产生的时间段，而且它还显著减少了创建线程的数目。

## 9.2常用的线程池类





Java通过Executors提供四种线程池，分别为：

newCachedThreadPool：用来创建一个可以无限扩大的线程池，适用于负载较轻的场景，执行短期异步任务。可缓存线程池，先查看池中有没有以前建立的线程，如果有，就直接使用。如果没有，就建一个新的线程加入池中，缓存型池子通常用于执行一些生存期很短的异步型任务。空闲线程超过60s就销毁。

ExecutorService cachedThreadPool = Executors.newCachedThreadPool();

cachedThreadPool.execute(new Runnable(){……});

newFixedThreadPool 创建一个固定大小的线程池，超出的线程会在队列中等待。因为采用无界的阻塞队列，所以实际线程数量永远不会变化，适用于负载较重的场景，对当前线程数量进行限制。（保证线程数可控，不会造成线程过多，导致系统负载更为严重）。

ExecutorService fixedThreadPool = Executors.newFixedThreadPool(3);

fixedThreadPool.execute(new Runnable() {……});

newScheduledThreadPool 创建一个定长线程池，支持定时及周期性任务执行，适用于执行延时或者周期性任务。

ScheduledExecutorService scheduledThreadPool = Executors.newScheduledThreadPool(5);

scheduledThreadPool.schedule(new Runnable() {

public void run() {

System.out.println("delay 3 seconds");

}

}, 3, TimeUnit.SECONDS);

newSingleThreadExecutor 创建一个单线程的线程池，适用于需要保证顺序执行各个任务。

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。

ExecutorService singleThreadExecutor = Executors.newSingleThreadExecutor();

singleThreadExecutor.execute(new Runnable() { ……});

在java doc中，并不提倡我们直接使用ThreadPoolExecutor，而是使用Executors类中提供的几个静态方法来创建线程池：

Executors.newCachedThreadPool(); //创建一个缓冲池，缓冲池容量大小为Integer.MAX\_VALUE

Executors.newSingleThreadExecutor(); //创建容量为1的缓冲池

Executors.newFixedThreadPool(int); //创建固定容量大小的缓冲池

下面是这三个静态方法的具体实现：

public static ExecutorService newFixedThreadPool(int nThreads) {

return new ThreadPoolExecutor(nThreads, nThreads,

0L, TimeUnit.MILLISECONDS,

new LinkedBlockingQueue<Runnable>());

}

public static ExecutorService newSingleThreadExecutor() {

return new FinalizableDelegatedExecutorService

(new ThreadPoolExecutor(1, 1,

0L, TimeUnit.MILLISECONDS,

new LinkedBlockingQueue<Runnable>()));

}

public static ExecutorService newCachedThreadPool() {

return new ThreadPoolExecutor(0, Integer.MAX\_VALUE,

60L, TimeUnit.SECONDS,

new SynchronousQueue<Runnable>());

}

　　从它们的具体实现来看，它们实际上也是调用了ThreadPoolExecutor，只不过参数都已配置好了。

　　newFixedThreadPool创建的线程池corePoolSize和maximumPoolSize值是相等的，它使用的LinkedBlockingQueue；

　　newSingleThreadExecutor将corePoolSize和maximumPoolSize都设置为1，也使用的LinkedBlockingQueue；

　 newCachedThreadPool将corePoolSize设置为0，将maximumPoolSize设置为Integer.MAX\_VALUE，使用的SynchronousQueue，也就是说来了任务就创建线程运行，当线程空闲超过60秒，就销毁线程。

　　实际中，如果Executors提供的三个静态方法能满足要求，就尽量使用它提供的三个方法，因为自己去手动配置ThreadPoolExecutor的参数有点麻烦，要根据实际任务的类型和数量来进行配置。

另外，如果ThreadPoolExecutor达不到要求，可以自己继承ThreadPoolExecutor类进行重写。

## 9.3 ThreadPoolExecutor类解析

java.uitl.concurrent.ThreadPoolExecutor类是线程池中最核心的一个类，因此如果要透彻地了解Java中的线程池，必须先了解这个类。下面我们来看一下ThreadPoolExecutor类的具体实现源码。

在ThreadPoolExecutor类中提供了四个构造方法：

public class ThreadPoolExecutor extends AbstractExecutorService {

.....

public ThreadPoolExecutor(int corePoolSize,int maximumPoolSize,long keepAliveTime,TimeUnit unit,

BlockingQueue<Runnable> workQueue);

public ThreadPoolExecutor(int corePoolSize,int maximumPoolSize,long keepAliveTime,TimeUnit unit,

BlockingQueue<Runnable> workQueue,ThreadFactory threadFactory);

public ThreadPoolExecutor(int corePoolSize,int maximumPoolSize,long keepAliveTime,TimeUnit unit,

BlockingQueue<Runnable> workQueue,RejectedExecutionHandler handler);

public ThreadPoolExecutor(int corePoolSize,int maximumPoolSize,long keepAliveTime,TimeUnit unit,

BlockingQueue<Runnable> workQueue,ThreadFactory threadFactory,RejectedExecutionHandler handler);

...

}

　　从上面的代码可以得知，ThreadPoolExecutor继承了AbstractExecutorService类，并提供了四个构造器，事实上，通过观察每个构造器的源码具体实现，发现前面三个构造器都是调用的第四个构造器进行的初始化工作。

　　下面解释下一下构造器中各个参数的含义：

corePoolSize：核心池的大小，这个参数跟后面讲述的线程池的实现原理有非常大的关系。在创建了线程池后，默认情况下，线程池中并没有任何线程，而是等待有任务到来才创建线程去执行任务，除非调用了prestartAllCoreThreads()或者prestartCoreThread()方法，从这2个方法的名字就可以看出，是预创建线程的意思，即在没有任务到来之前就创建corePoolSize个线程或者一个线程。默认情况下，在创建了线程池后，线程池中的线程数为0，当有任务来之后，就会创建一个线程去执行任务，当线程池中的线程数目达到corePoolSize后，就会把到达的任务放到缓存队列当中；

maximumPoolSize：线程池最大线程数，这个参数也是一个非常重要的参数，它表示在线程池中最多能创建多少个线程；

keepAliveTime：表示线程没有任务执行时最多保持多久时间会终止。默认情况下，只有当线程池中的线程数大于corePoolSize时，keepAliveTime才会起作用，直到线程池中的线程数不大于corePoolSize，即当线程池中的线程数大于corePoolSize时，如果一个线程空闲的时间达到keepAliveTime，则会终止，直到线程池中的线程数不超过corePoolSize。但是如果调用了allowCoreThreadTimeOut(boolean)方法，在线程池中的线程数不大于corePoolSize时，keepAliveTime参数也会起作用，直到线程池中的线程数为0；

unit：参数keepAliveTime的时间单位，有7种取值，在TimeUnit类中有7种静态属性：

TimeUnit.DAYS; //天

TimeUnit.HOURS; //小时

TimeUnit.MINUTES; //分钟

TimeUnit.SECONDS; //秒

TimeUnit.MILLISECONDS; //毫秒

TimeUnit.MICROSECONDS; //微妙

TimeUnit.NANOSECONDS; //纳秒

workQueue：一个阻塞队列，用来存储等待执行的任务，这个参数的选择也很重要，会对线程池的运行过程产生重大影响，一般来说，这里的阻塞队列有以下几种选择：

ArrayBlockingQueue;

LinkedBlockingQueue;

SynchronousQueue;

　　ArrayBlockingQueue和PriorityBlockingQueue使用较少，一般使用LinkedBlockingQueue和Synchronous。线程池的排队策略与BlockingQueue有关。

threadFactory：线程工厂，主要用来创建线程；

handler：表示当拒绝处理任务时的策略，有以下四种取值：

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。

（

线程池的核心池大小、最大线程数、空闲线程保活时间；

阻塞队列，用于存储等待执行的任务；

线程工厂，用来创建线程；

拒绝处理任务的策略。

）