# 线程基本概念

有时需要程序“同时”可以做很多事，即所谓的多线程（Multi-thread）程序。

进程(Process)是一个包含自身执行地址的程序，在一个多任务的从操作系统中，可以分配CPU时间给每一个进程，CPU在某个时间片中中执行某个进程，然后下一个时间片执行另一个进程，由于转换速速很快，使得每个程序好像是在同时处理。

线程（Thread）则是进程中的执行流程，一个进程中可以同时包括多个线程，也就是说一个程序中同时可能进行多个不同的子流程，每个子流程可以得到一小段程序的执行时间，每执行完一个线程就跳到下一个线程，由于转换的速度很快，这使得一个程序可以像是同时处理多个事务，例如网络程序可以一方面接受网络上的数据，另一方面同时计算数据并显示结果。一个多线程程序可以同时处理多个子流程。

# 创建线程的两种传统方式

想要让对象具有多线程功能，只要继承java.lang.Thread类或是实现java.lang.Runnable接口。建议以实现Runnable的方式让对象具有线程功能，保留日后修改的弹性。

**public** **class** TraditionalThread {

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

Thread thread = **new** Thread() {

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

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

**try** {

Thread.*sleep*(500);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

//不管任何时候都代表当前线程对象

System.*out*.println("1:"+Thread.*currentThread*().getName());

//this代表的就是这个thread对象

System.*out*.println("2:"+**this**.getName());

}

}

};

thread.start();

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

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

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

**try** {

Thread.*sleep*(500);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

// System.out.println("2:"+this.getName()); //会报错，因为this指的是Runnable，而Runnable是线程要调用的代码，而非线程本身

System.*out*.println("1:"+Thread.*currentThread*().getName());

}

}

});

thread2.start();

}

}

# Daemon线程

一个Daemon(守护进程、后台程序)线程是在后台执行服务线程，例如网络服务器侦听连接端口的服务、隐藏系统线程，如垃圾收集线程或其它JVM建立的线程。如果所有的非Daemon的线程都结束了，则Daemon线程自动就会终止。

如果你希望某个线程在产生它的线程结束后也跟着终止，那么你要将它设置为Daemon线程。

**public** **class** DaemonThread {

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

Thread thread = **new** Thread(**new** Runnable(){

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

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

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

}

}

});

//设置为Daemon线程

thread.setDaemon(**true**);

thread.start();

//下面代码让主线程(main)暂停一下，否则主线程执行完了，

//上面Daemon线程可能还没来得及执行run里面的代码,就跟着结束了

**try** {

Thread.*sleep*(10);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

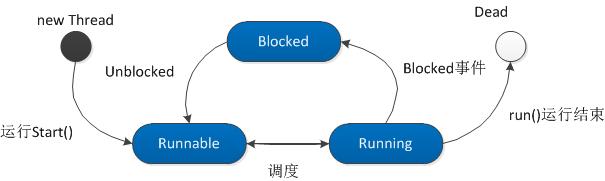
}

这个程序在主线程结束之后，Daemon线程也会跟着结束，你可以使用 setDaemon()方法来设置一个线程是否为Daemon线程，示例代码如果使用setDaemon()方法设置为true,则程序会不断地打印出T字符而不终止(只能按ctrl+c强迫终止程序)；使用isDaemon()可以判断该线程是否为Daemon线程。

Java默认所有从Daemon线程产生的线程也是Daemon线程，因为基本上由一个后台服务线程衍生出来的线程，也应该是为了在后台服务而产生的，所以在产生它的线程停止后，衍生线程也应该跟着停止。

# 线程的生命周期

线程的生命周期比较复杂，我们先从基本的周期状态开始介绍，分别为创建线程(new Thread) 、就绪(Runnbale) 、 堵塞 （Blocked ）、消亡(Dead) , 状态间的转移如下图所示：



当您范例化一个Thread并执行start()之后，线程进入Runnbale状态，此时线程尚未真正开始执行，必须等待调度(Scheduler)的调度，被排入执行的线程才会执行run()方法中的定义。

虽然线程看起来像是同步执行，但实际上在某一时间点上，仍然只有一个线程在执行，只是线程之间切换迅速，所以看起来是同时执行。

线程有优先权，从1(Thread.MIN\_PRIORITY)到10(Thread.MAX\_PRIORITY),默认是5(Thread.NORM\_PRIORITY).您可以使用Thread的setPriority()方法来设置线程的优先级，设置必须在1到10之间，否则会抛出IllegalArgumentException.

优先权越高，调度越优先排入执行，如果优先权相同，则采取轮转法调度(Round-robin方式).

绝大多数的操作系统都支持Timeslicing,简单地说就是操作系统会为每个线程分配一小段CPU时间(Quantum),时间一到就换下一个线程，即使现有的线程还没结束。对于不支持Timeslicing的操作系统，必须在线程完成后，才能轮到下一个线程。在这样的操作系统中，如果想让目前线程礼让一下其它线程，让它们有机会获得执行权，可以调用yield()方法，如：

public class SomeClass{

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

Thread thread = **new** Thread(**new** Runnable(){

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

//。。。

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

//。。。

yield();//暂时让出执行权

}

}

});

thread.start();

}

yield()方法让同样优先权的线程有被执行的机会，当线程执行yield()方法让出执行权时，它会再度处于Runnable状态，等待调度。对于支持Timeslicing的操作系统，不需要调用yield()方法，因为操作系统会自动分配时间给线程轮流执行。

有几种状况会让线程进入Blocked状态：

* 等待输入输出完成
* 调用sleep()方法
* 尝试获得对象锁定
* 调用wait()方法

当线程因为等待用户的文字输入，侦听网络连接或以上情况时，会阻止线程执行而进入Blocked 状态，线程调度将不给这个线程分配执行时间，直到以下几种情况回到Runnable状态：

* 输入输出完成
* 调用interrupt()
* 获得对象锁定
* 调用notify()或notifyAll()

这里举个简单的例子，当您使用Thread.sleep()让线程执行进入Blocked状态，您可以使用interrupt()让它离开Blocked状态。

使用interrupt()时，会抛出java.lang.InterruptedException异常对象，如下：

**public** **class** InterruptDemo {

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

Thread thread = **new** Thread(**new** Runnable() {

@Override

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

**try** {

Thread.*sleep*(99999);

} **catch** (InterruptedException e) {

System.*out*.println("I'm interrupted!");

}

}

});

thread.start();

thread.interrupt();

}

}

执行结果： I'm interrupted!

当然，如果你要暂停线程，但暂停的时间未知，使用sleep()并不是个好方法，您可以使用wait()让线程进入Blocked状态，然后让别的线程用notify()或notifyAll()来通知被Blocked的线程回到Runnable状态。

# 线程的加入（join）

如果有一个A线程正在运行，您希望插入一个B线程，并要求B线程先执行完毕，然后再继续A线程的流程，你可以使用join()方法来完成。这就好比手头上有一个工作正在进行，老板插入一个工作要您先做好，然后您再进行原先正在进行的工作。

当线程使用join()方法加入至另一个线程时，另一个线程会等待这个被加入的线程执行完毕，然后再继续它的动作，join()的意思表示将线程加入成为另一个线程的流程之一。

**public** **class** ThreadA {

/\*\*

\* 线程的加入（join)

\*/

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

System.*out*.println("Thread A 执行");

Thread threadB = **new** Thread(**new** Runnable() {

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

**try** {

System.*out*.println("Thread B 开始...");

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

Thread.*sleep*(1000);

System.*out*.println("Thread B 执行...");

}

System.*out*.println("Thread B 即将结束...");

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

});

threadB.start();

**try** {

threadB.join();

} **catch** (InterruptedException e) {

e.printStackTrace();

}

System.*out*.println("Thread A 执行");

}

}

执行结果如下：

Thread A 执行

Thread B 开始...

Thread B 执行...

Thread B 执行...

Thread B 执行...

Thread B 执行...

Thread B 执行...

Thread B 即将结束...

Thread A 执行

如果程序中threadB没有使用join()将之加入主线程的流程中，则最后一行显示"Thread A 执行"的语句会先执行完毕(因为ThreadB使用了sleep()，这让主线程有机会获得时间来执行).有时加入的线程可能处理太久，你不想无止境地等待这个线程，则可以在join上指定时间，例如join(10000)，表示加入成为流程之一的线程至多处理10000毫秒，也就是10秒，如果加入的线程还没执行完毕就不理它了，目前的线程可以继续执行原本的工作流程。

# 线程的停止

如果你想要停止一个线程的执行，当您查看API时，您会发现Thread的stop()方法已经被表示为deprecated,不建议使用stop()来停止一个线程的执行。

如果想要停止一个线程，最好自行实现。一个线程要进入Dead状态，就是执行完run()方法，简单地说，如果你想要停止一个线程的执行，既要提供一个让线程可以完成run()的流程，而这也是自行实现线程停止的基本概念。

如果线程的run()方法中执行的是一个重复执行的循环，可以提供一个标志（flag）来控制循环是否执行，从而循环有机会终止，使线程可以离开run()方法以终止线程：

**public** **class** SomeThread **implements** Runnable {

**private** **boolean** isContinue = **true**;

**public** **void** teminated() {

isContinue = **false**;

}

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

**while** (isContinue) {

// ...some statements

}

}

}

如果程序因为等待输入输出设备而停滞（进入Blocked状态）,一般必须等待输入输出的动作完成才能离开Blocked状态，你无法使用interrupt()来使得线程来离开run()方法，要提供替代的方法，比如引发一个异常，而这个异常要如何引发，要看所使用的输入输出而定。例如您使用readLine()等待网络上的一个信息，此时线程进入Blocked直到读到一个信息，您要让它离开run()方法就是使用close()关闭它的字符串刘，这时会引发一个IOException异常而使得线程离开run()方法，例如：

**public** **class** Client **implements** Runnable {

**private** Socket skt;

**public** **void** terminate() {

skt.close();

}

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

// ...

**try** {

BufferedReader buf = **new** BufferedReader(**new** InputStreamReader(

skt.getInputStream()));

// 读取客户信息

// 执行readLine()会进入Blocked状态

// 直到读到客户端信息

**while** ((userMessage = buf.readLine()) != **null**) {

// ...

}

} **catch** (IOException e) {

System.*out*.println("线程被终止....");

}

}

}

上面这个程序是个简单的架构示范，实际的设计必须等您的程序功能与I/O类型而定。除了stop()之外，suspend()、resume()方法也被标志为deprecated,如果要达成与这些方法相同的功能，都必须自行设计。

# 线程组（ThreadGroup）

在java中每个线程都属于某个线程组（ThreadGroup）。例如在main()中产生一个线程，则这个线程属于main线程组管理的一员，您可以使用下面的指令来获得目前线程所属的线程组名称：

Thread.currentThread().getThreadGroup().getName()

每一个线程产生时，都会被归入某个线程组，视线程是在哪个线程组中产生而定。如果没有指定，则归入产生该子线程的线程组中。您也可以自行指定线程组，线程一旦归入到某个组，就无法更换组。

ThreadGroup 类可以统一管理整个线程组中的线程，可以用以下方式来产生线程组，而且一并指定其线程组：

ThreadGroup threadGroup1 = **new** ThreadGroup("group1");

ThreadGroup threadGroup2 = **new** ThreadGroup("group2");

Thread thread1 = **new** Thread(threadGroup1,"group1's member");

Thread thread2 = **new** Thread(threadGroup2,"group2's member");

ThreadGroup中的某些方法，可以对所有的线程产生作用，例如interrupt()方法可以interrupt线程组中所有的线程,setMaxPriority()方法可以设置线程组中线程所能拥有的最高优先权（本来就拥有更高优先权的线程不受影响）

如果你想要一次获得线程组中所有的线程来进行某种操作，可以使用enumerate()方法，例如：

Thread[] threads = new Thread[threadGroup1.activeCount()];

threadGroup1.enumerate(threads); //将属于threadGroup1的线程设置到threads数组中

activeCount()方法获得线程组中正在运行的线程数量，enumerate()方法要传入一个Thread数组，它将线程对象设置到每个数组字段中，然后就可以通过数组索引来操作这些线程。

ThreadGroup 中 有一个uncaughtException()方法。当线程组中某个线程发生Unchecked exception异常时，由执行环境调用此方法进行相关处理，如果有必要，您可以重新定义该方法，范例如下：

**public** **class** ThreadGroupDemo {

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

ThreadGroup threadGroup1

//这是匿名类写法

= **new** ThreadGroup("group1"){

//继承ThreadGroup并重新定义以下方法

//在线程成员抛出unchecked Exception

//会执行此方法

@Override

**public** **void** uncaughtException(Thread t, Throwable e) {

System.*out*.println(t.getName()+":"+e.getMessage());

}

};

Thread thread1 = **new** Thread(threadGroup1,**new** Runnable(){

@Override

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

**throw** **new** RuntimeException("测试异常");

}

});

thread1.start();

}

}

在uncaughtException()方法的参数中，第一个参数可以获得发生异常的线程实例，而第二个参数可以获得异常对象，范例中显示了线程的名称及异常信息，结果如下：

Thread-0:测试异常

在J2SE 5.0之前，如果要统一处理某些线程的Unchecked Exception，可以使用 ThreadGroup 来管理。在继承 ThreadGroup 之后重新定义其 uncaughtException()方法，如上范例所示。在J2SE5.0之后，就不用这么麻烦，可以让异常处理类使用Thread.UncaughtExceptionHandler接口，并实现其uncaughtException()方法，例如可以将上面范例改写如下：

**public** **class** ThreadExceptionHandler **implements** Thread.UncaughtExceptionHandler{

@Override

**public** **void** uncaughtException(Thread t, Throwable e) {

System.*out*.println(t.getName()+":"+e.getMessage());

}

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

//建立异常处理者

ThreadExceptionHandler handler = **new** ThreadExceptionHandler();

ThreadGroup threadGroup1 = **new** ThreadGroup("group1");

Thread thread1 = **new** Thread(threadGroup1,**new** Runnable(){

@Override

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

**throw** **new** RuntimeException("测试异常");

}

});

//设置异常处理者

thread1.setUncaughtExceptionHandler(handler);

thread1.start();

}

}

# 同步(synchronize)

多个线程操纵同一份数据的时候，数据的同步(一致性、完整性)问题就要特别注意，就会出现线程安全的问题。

线程的同步互斥synchronized用来保护多线程下被共享的数据。

线程如何同步和通讯？

同学回答说synchronized方法或代码块！面试官似乎不太满意！

只有多个synchronized代码块使用的是同一个监视器对象，这些synchronized代码块之间才具有线程互斥的效果，假如a代码块用obj1作为监视器对象，假如b代码块用obj2作为监视器对象，那么，两个并发的线程可以同时分别进入这两个代码块中。

对于同步方法的分析，所用的同步监视器对象是this。

接着对于静态同步方法的分析，所用的同步监视器对象是该类的Class对象。

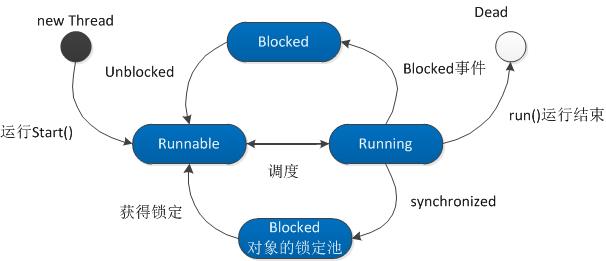
将多个线程要用到共同数据(包括同步锁)或共同算法的若干方法应该归在同一个类身上，这种设计正好体现了高内聚和程序的健壮性。

synchronized应该用在需要同步的表示业务逻辑对象的的方法上，而尽量不要将synchronized用在线程的方法(例如：run)内部，这样在逻辑改变时，就不需要修改线程内部的代码。

这里引进对象的锁定（lock）观念。每个对象在内部都会有一个锁定。对象的锁定在平时是没有作用的。被表示为synchronized的方法会成为同步区域，当线程执行某个对象的同步区域时，对象的锁定就有作用了。要执行同步区域的线程，都必须先获得对象的锁定。

因为对象的锁只有一个，当有一个线程已取走锁定而正在执行同步区域中的程序代码时，若其它线程也想执行synchronized的区域，但这些线程无法获得锁定，所以只好在对象的锁定池(Lock Pool)等待，直到锁定被前一个线程归还为止，此时在锁定池中的线程竞争被归还的对象锁定，只有获得锁定的线程才能进入Runnable状态，等待调度并执行同步区域。

加入锁定的线程状态图：



同步区域在有线程占据时就像一个禁区，不允许其它线程进入，由于同时只能有一个线程在同步区域，所以更新共享数据时，就有如单线程程序在更新数据一样，从而保证对象中的数据会与给定的数据同步。

也可以标识某个对象要求同步化，例如在多线程存取同一个ArrayList对象时，由于ArrayList并没有实现数据存取时的同步化，所以当多个线程存取同一个ArrayList时，有可能发生两个以上的线程将数据存入同一个位置，造成数据的相互覆盖。为了确保数据存入时的正确性，可以在存取ArrayList对象时要求同步化，例如：

synchronized(arrayList){

arrayList.add(new SomeClass());

}

同步化确保数据的同步，但当一个线程获得对象锁定而占据同步区块，将导致其它线程等待它释放锁定时出现延迟。在线程少时可能看不出来，但在线程多的环境中必然造成一个效能问题(例如大型网站多人连机时)。

代码示例：

**public** **class** TraditionalThreadSynchronized {

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

**new** TraditionalThreadSynchronized().init();

}

**private** **void** init(){

**final** Outputer outputer = **new** Outputer();//在静态方法中不能创建内部类的实例对象

**new** Thread(**new** Runnable(){

@Override

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

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

**try** {

Thread.*sleep*(10);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

outputer.output("zhangxiaoxiang"); //和下面的线程用到的是同一个Outputer对象outputer

}

}

}).start();

**new** Thread(**new** Runnable(){

@Override

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

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

**try** {

Thread.*sleep*(10);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

outputer.output("lihuoming");

//outputer.output2("lihuoming"); //也可以实现互斥，因为output，output2两个方法检查的是同一个锁对象

// new Outputer().print("lihuoming"); 不行，线程同步时必须是同一个对象(锁的是一个对象)

}

}

}).start();

}

//内部类可以访问外部类的成员变量

//下面代码要实现原子性，也就是线程的互斥，否则多个线程同时调用时就有可能打印出混乱的名字

//output和output2可以同步，因为它们synchronized用的都是this对象

//output和output3不可以同步，因为它们synchronized用的不是同一个对象，static方法是由类的字节码调用的

//如果将output方法中的synchronized对象this更改为synchronized(Outputer.class)，那么就可以同步了

**static** **class** Outputer{//在内部类前面加一个static就相当于外部类，在此地方加static,是为了能在其中定义静态方法output3。因为内部类本身不能定义static方法。

String xxx = "";

**public** **void** output(String name){

**int** len = name.length();

//上面两个线程用到的是Outputer的同一个实例对象outputer进行打印的，

//所以synchronized (xxx)中的 xxx 也是同一个对象，就会解决问题

//如果synchronized (name)就会出问题，因为name变量是传入的不同字符串对象

//synchronized (xxx) { //也可以改为synchronized (this)表示的就是 调用我的那个对象，上面两个线程调用我时用的是一个Outputer对象，所以也没问题。

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

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

System.*out*.print(name.charAt(i));

}

System.*out*.println();

}

}

/\*如果要保护的是方法中的所有代码可以直接在方法上加关键字synchronized，一段代码中最好出现一个synchronized，

如果方法定义中加了synchronized，而方法的内部代码也加了synchronized,那么就很可能出现死锁

方法上的synchronized用的就是this对象synchronized (this)

\*/

**public** **synchronized** **void** output2(String name){

**int** len = name.length();

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

System.*out*.print(name.charAt(i));

}

System.*out*.println();

}

**public** **static** **synchronized** **void** output3(String name){

**int** len = name.length();

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

System.*out*.print(name.charAt(i));

}

System.*out*.println();

}

}

}

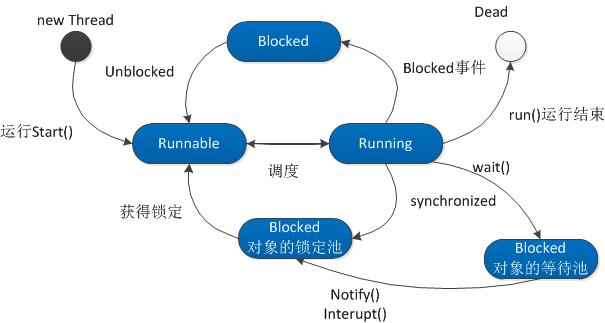
# wait()、notify()

wait()、notify()与notifyAll()是由Objec类所提供的方法，在定义类时会继承下来，wait()、notify()、notifyAll()都被宣告为final，所以您无法重新定义它们。通过wait()方法您可以要求线程进入对象的等待池(Wait Pool)，或是通知线程回到锁定池的Blocked状态。

必须在同步的方法或区块中才能调用wait()方法（也就是线程获得锁定时）。当对象的wait()方法被调用，目前的线程会被放入对象的等待池中，线程归还对象的锁定，其它的线程可竞争对象的锁定；被放在等待池中的线程也是处于Blocked状态，所以不参与线程的调度。

wait()可以指定等待的时间。如果指定时间的话，则时间到了之后，线程会再度回到锁定池的Blocked状态，等待竞争对象锁的机会，如果指定时间为0或不指定，则线程会持续等待，直到被中断(interrupt),或是被告知(notify)回到锁定池的Blocked状态。

下图是加入等待池的线程状态图：



当对象的notify()被调用，它会从目前对象的等待池中通知“一个”线程加入到锁定池的Blocked状态，被通知的线程是随机的，且会与其它线程共同竞争对象的锁定；如果调用notifyAll(),则“所有”在等待池中的线程都会被通知回到锁定池的Blocked状态，这些线程会与其它线程共同竞争对象的锁定。

简单地说，当线程调用到对象的wait()方法时，表示它要先让出对象的锁定并等待通知，或是等待一段指定的时间，直到被通知或时间到时，再与其它线程竞争对象的锁定，如果获得锁定，就从等待点开始执行。这就好比您要让某人做事，做到一半时某人叫你等候通知(或等候1分钟之类)，当您被通知(或时间到时)，默认会继续为您服务。

/\*传统线程同步通信技术

\*子线程循环10次，接着主线程循环100，接着又回到子线程循环10次，

\*接着再回到主线程又循环100，如此循环50次，请写出程序

\*\*/

**public** **class** TraditionalThreadCommunication {

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

**final** Business business = **new** Business();

**new** Thread(**new** Runnable() {

@Override

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

**for** (**int** i = 1; i <= 50; i++) {

business.sub(i);

}

}

}).start();

// 本身main方法就占用一个线程，所以就在main方法中接着写

**for** (**int** i = 1; i <= 50; i++) {

business.main(i);

}

}

}

//业务是 主线程循环100，子线程循环10次，然后由别人调用这两个线程

**class** Business{

**private** **boolean** beShouldSub = **true**;

**public** **synchronized** **void** sub(**int** i){

**while**(!beShouldSub){//也可用if进行判断，但此处用while程序更健壮，防止伪唤醒；打比方：防止自己被梦惊醒，而非被别人叫醒

**try** {

**this**.wait();

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

**for** (**int** j = 1; j <= 10; j++) {

System.*out*.println("sub thread sequence of : " + j

+ ",loop of:" + i);

}

beShouldSub = **false**;

**this**.notify();

}

**public** **synchronized** **void** main(**int** i){

**while**(beShouldSub){//也可用if进行判断，但此处用while程序更健壮

**try** {

**this**.wait();

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

**for** (**int** j = 1; j <= 100; j++) {

System.*out*.println("main thread sequence of : " + j

+ ",loop of:" + i);

}

beShouldSub = **true**;

**this**.notify();

}

}

# Lock&Condition实现线程同步通信

* Lock比传统线程模型中的synchronized方式更加面向对象，与生活中的锁类似，锁本身也应该时一个对象。两个线程执行的代码片段要实现同步互斥的效果，它们必须用同一个Lock对象。锁是上在代表要操作的资源的类的内部方法中，而不是线程代码中。
* 读写锁：分为读锁和写锁，多个读锁不互斥，读锁和写锁互斥，写锁和写锁互斥，这时由jvm自己控制的，你只要上好相应的锁即可。如果你的代码只读数据，可以让很多人同时读，但不能同时写，那就上读锁；如果你的代码修改数据，只能有一个人在写，且不能同时读取，那就上写锁。总之，读的时候上读锁，写的时候上写锁。
* Condition的功能类似在传统线程技术中的Object.wait()和Object.notify()功能。在等待Condition时，允许发生“虚假唤醒”，这通常作为对基础平台语义的让步。对于大多数应用程序，这带来的实际影响很小，因为Condition应该总是在一个循环中被等待，并测试正被等待的状态声明。某个时间可以随时移除可能的虚假唤醒，但建议应用程序员总是假定这些“虚假唤醒”可能发生，因此总在一个循环中等待。
* 一个锁内部可以有多个Condition，即有多路等待和通知，可以参看Jdk1.5提供的Lock与Condition实现的可阻塞队列的应用案例，从中除了要体味算法，还要体味面向对象的封装。在传统的线程机制中，一个监视对象上只能有一路等待和通知，要想实现多路等待和通知，必须嵌套使用多个同步监视对象。

代码示例：

/\*\*

\* Lock 比 Synchronized 更面向对象，实现线程互斥同步

\* \*/

**public** **class** LockTest {

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

**new** LockTest().init();

}

**private** **void** init() {

**final** Outputer outputer = **new** Outputer();// 在静态方法中不能创建内部类的实例对象

**new** Thread(**new** Runnable() {

@Override

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

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

**try** {

Thread.*sleep*(10);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

outputer.output("zhangxiaoxiang"); // 和下面的线程用到的是同一个Outputer对象outputer

}

}

}).start();

**new** Thread(**new** Runnable() {

@Override

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

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

**try** {

Thread.*sleep*(10);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

outputer.output("lihuoming");

}

}

}).start();

}

**static** **class** Outputer {// 在内部类前面加一个static就相当于外部类，在此地方加static,是为了能在其中定义静态方法。因为内部类本身不能定义static方法。

Lock lock = **new** ReentrantLock();

**public** **void** output(String name) {

**int** len = name.length();

lock.lock();

**try** {

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

System.*out*.print(name.charAt(i));

}

System.*out*.println();

} **finally** {

lock.unlock();

}

}

}

}

/\*\*

\* 读写锁ReentrantReadWriteLock

\* \*/

**public** **class** ReadWriteLockTest {

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

**final** Queue3 q3 = **new** Queue3();

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

{

**new** Thread(){

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

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

q3.get();

}

}

}.start();

**new** Thread(){

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

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

q3.put(**new** Random().nextInt(10000));

}

}

}.start();

}

}

}

**class** Queue3{

**private** Object data = **null**;//共享数据，只能有一个线程能写该数据，但可以有多个线程同时读该数据。

ReadWriteLock rwl = **new** ReentrantReadWriteLock();

**public** **void** get(){

rwl.readLock().lock(); //读锁

**try** {

System.*out*.println(Thread.*currentThread*().getName() + " be ready to read data!");

Thread.*sleep*((**long**)(Math.*random*()\*1000));

System.*out*.println(Thread.*currentThread*().getName() + "have read data :" + data);

} **catch** (InterruptedException e) {

e.printStackTrace();

}**finally**{

rwl.readLock().unlock();

}

}

**public** **void** put(Object data){

rwl.writeLock().lock(); //写锁

**try** {

System.*out*.println(Thread.*currentThread*().getName() + " be ready to write data!");

Thread.*sleep*((**long**)(Math.*random*()\*1000));

**this**.data = data;

System.*out*.println(Thread.*currentThread*().getName() + " have write data: " + data);

} **catch** (InterruptedException e) {

e.printStackTrace();

}**finally**{

rwl.writeLock().unlock();

}

}

}

/\*传统线程同步通信技术改为Condition将方式

\*子线程循环10次，接着主线程循环100，接着又回到子线程循环10次，

\*接着再回到主线程又循环100，如此循环50次，请写出程序

\*\*/

**public** **class** ConditionCommunication {

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

**final** Business business = **new** Business();

**new** Thread(**new** Runnable() {

@Override

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

**for** (**int** i = 1; i <= 50; i++) {

business.sub(i);

}

}

}).start();

// 本身main方法就占用一个线程，所以就在main方法中接着写

**for** (**int** i = 1; i <= 50; i++) {

business.main(i);

}

}

//业务是 主线程循环100，子线程循环10次，然后由别人调用这两个线程

/\*\*

\* //内部类，静态方法不能创建内部类的实例，前面加static后作用相当于外部类，

\* 静态方法中可以创建实例；TraditionalThreadCommunication中也有Business类，

\* 所以在该中将Business定义为内部类。

\* \*/

**static** **class** Business{

Lock lock = **new** ReentrantLock();

Condition condition = lock.newCondition();

**private** **boolean** beShouldSub = **true**;

**public** /\*\*synchronized\*/ **void** sub(**int** i){

lock.lock();

**try**{

**while**(!beShouldSub){//也可用if进行判断，但此处用while程序更健壮，防止伪唤醒；打比方：防止自己被梦惊醒，而非被别人叫醒

**try** {

condition.await();

} **catch** (Exception e) {

e.printStackTrace();

}

}

**for** (**int** j = 1; j <= 10; j++) {

System.*out*.println("sub thread sequence of : " + j

+ ",loop of:" + i);

}

beShouldSub = **false**;

//this.notify();

condition.signal();

}**finally**{

lock.unlock();

}

}

**public** /\*\*synchronized\*/ **void** main(**int** i){

lock.lock();

**try**{

**while**(beShouldSub){//也可用if进行判断，但此处用while程序更健壮

**try** {

//this.wait();

condition.await();

} **catch** (Exception e) {

e.printStackTrace();

}

}

**for** (**int** j = 1; j <= 100; j++) {

System.*out*.println("main thread sequence of : " + j

+ ",loop of:" + i);

}

beShouldSub = **true**;

//this.notify();

condition.signal();

}**finally**{

lock.unlock();

}

}

}

}

/\*

ThreeConditionCommunication

\*\*/

**public** **class** ThreeConditionCommunication {

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

**final** Business business = **new** Business();

**new** Thread(

**new** Runnable() {

@Override

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

**for**(**int** i=1;i<=50;i++){

business.sub2(i);

}

}

}

).start();

**new** Thread(

**new** Runnable() {

@Override

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

**for**(**int** i=1;i<=50;i++){

business.sub3(i);

}

}

}

).start();

**for**(**int** i=1;i<=50;i++){

business.main(i);

}

}

**static** **class** Business {

Lock lock = **new** ReentrantLock();

Condition condition1 = lock.newCondition();

Condition condition2 = lock.newCondition();

Condition condition3 = lock.newCondition();

**private** **int** shouldSub = 1;

**public** **void** sub2(**int** i){

lock.lock();

**try**{

**while**(shouldSub != 2){

**try** {

condition2.await();

} **catch** (Exception e) {

e.printStackTrace();

}

}

**for**(**int** j=1;j<=10;j++){

System.*out*.println("sub2 thread sequence of " + j + ",loop of " + i);

}

shouldSub = 3;

condition3.signal();

}**finally**{

lock.unlock();

}

}

**public** **void** sub3(**int** i){

lock.lock();

**try**{

**while**(shouldSub != 3){

**try** {

condition3.await();

} **catch** (Exception e) {

e.printStackTrace();

}

}

**for**(**int** j=1;j<=20;j++){

System.*out*.println("sub3 thread sequence of " + j + ",loop of " + i);

}

shouldSub = 1;

condition1.signal();

}**finally**{

lock.unlock();

}

}

**public** **void** main(**int** i){

lock.lock();

**try**{

**while**(shouldSub != 1){

**try** {

condition1.await();

} **catch** (Exception e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

**for**(**int** j=1;j<=100;j++){

System.*out*.println("main thread sequence of " + j + ",loop of " + i);

}

shouldSub = 2;

condition2.signal();

}**finally**{

lock.unlock();

}

}

}

}

/\*Condition 实例实质上被绑定到一个锁上。要为特定 Lock 实例获得 Condition 实例，请使用其 newCondition() 方法。

作为一个示例，假定有一个绑定的缓冲区，它支持 put 和 take 方法。

如果试图在空的缓冲区上执行 take 操作，则在某一个项变得可用之前，线程将一直阻塞；如果试图在满的缓冲区上执行 put 操作，则在有空间变得可用之前，线程将一直阻塞。

我们喜欢在单独的等待 set 中保存 put 线程和 take 线程，

这样就可以在缓冲区中的项或空间变得可用时利用最佳规划，一次只通知一个线程。可以使用两个 Condition 实例来做到这一点。\*/

/\*ArrayBlockingQueue 类提供了这项功能，因此没有理由去实现这个示例类。\*/

**public** **class** BoundedBuffer {

**final** Lock lock = **new** ReentrantLock();

**final** Condition notFull = lock.newCondition();

**final** Condition notEmpty = lock.newCondition();

**final** Object[] items = **new** Object[100];

**int** putptr, takeptr, count;

**public** **void** put(Object x) **throws** InterruptedException {

lock.lock();

**try** {

**while** (count == items.length)

notFull.await(); /\*\* 当多个线程同时运行到该行时；其中一个先往下执行，其它阻塞；如果只定义一个Condition, 那么执行到notEmpty.signal()时，会重新唤醒在此阻塞的其它线程在此继续向下执行，而非 唤醒take()方法中的代码\*/

items[putptr] = x;

**if** (++putptr == items.length)

putptr = 0;

++count;

notEmpty.signal();

} **finally** {

lock.unlock();

}

}

**public** Object take() **throws** InterruptedException {

lock.lock();

**try** {

**while** (count == 0)

notEmpty.await();

Object x = items[takeptr];

**if** (++takeptr == items.length)

takeptr = 0;

--count;

notFull.signal();

**return** x;

} **finally** {

lock.unlock();

}

}

}

## //一个面试题：写一个缓存类。

**public** **class** CacheDemo {

**private** Map<String, Object> cache = **new** HashMap<String, Object>();

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

}

**private** ReadWriteLock rwl = **new** ReentrantReadWriteLock();

**public** Object getData(String key){

rwl.readLock().lock();

Object value = **null**;

**try**{

value = cache.get(key);

**if**(value == **null**){

rwl.readLock().unlock();

rwl.writeLock().lock(); /\*\*当多个线程都执行到此处，只有第一个线程可以 上“写锁”成功，其它线程都堵塞了，

\* 当第一个线程执行完剩下的代码后，其它线程可以接着在此执行，这样就造成下面 value = "aaaa"; 多次从数据库查询执行，

\* 故下面代码要再次加上 if(value==null) 代码进行判断

\* \*/

**try**{

**if**(value==**null**){

value = "aaaa";//实际失去queryDB();

}

}**finally**{

rwl.writeLock().unlock();

}

rwl.readLock().lock();

}

}**finally**{

rwl.readLock().unlock();

}

**return** value;

}

}

# 容器类的线程安全(Thread-safe)

容器类默认没有考虑线程安全，你必须自行实现同步以确保共享数据在多线程下不会出错。

例如，若您使用List对象时，可以这样实现：

// arrayList参考值一个ArrayList的一个范例

synchronized(arrayList){

arrayList.add(new SomeClass());

}

事实上，您也可以使用java.util.Collections的synchronizedXXX()等方法来传回一个同步化的容器对象，例如传回一个同步化的List:

List list = Collections.synchronizedList(new ArrayList());

以这种方式返回List对象，在存取数据时会进行同步化的工作。不过在您使用Iterator遍访对象时，仍必须实现同步化，因为这样的List使用iterator()方法返回的Iterator对象，并没有保证线程安全(Thread-safe),一个实现遍历的例子如下：

List list = Collections.synchronizedList(new ArrayList());

…

synchronized(list){

Iterator i = list.iterator();

while(i.hasNext()){

foo(i.next());

}

}

在J2SE5.0之后，新增了一个package，即java.util.concurrent.其中包括了一些确保线程安全的Collection类，例如 ConcurrentHashMap 、 CopyOnWriteArrayList 、CopyOnWriteArraySet 等 。这些新增的 Collection 类的基本操作与先前介绍的Map、List 、Set 等对象是相同的，所不同的是增加了同步化的功能，而且依对象存取时的需求不同而有不同的同步化实现，以同时确保效率与安全性。

ConcurrentHashMap针对HashTable中不同的区段(Segment)进行同步化，而不是对整个对象进行同步化。ConcurrentHashMap默认有16段区域，当有线程存取第一个区段时，第一个区段进入同步化，然而另一个线程仍可以存取第一个区段以外的其它区段，而不用等待第一个线程存取完成，所以与同步化整个对象来说，新增的ConcurrentHashMap 、 CopyOnWriteArrayList 、CopyOnWriteArraySet 等类，在效率与安全上获得了较好的平衡。

# ThreadLocal类实现线程范围内的共享变量

要编写一个多线程安全（Thread-safe）的程序是困难的，为了让线程共享资源，必须小心地对共享资源进行同步，同步带来一定的效能延迟，而另一方面，在处理同步的时候，又要注意对象的锁定与释放，避免产生死结，种种因素都使得多线程程序变得困难。

尝试从另一个角度来思考多线程共享资源的问题，既然共享资源这么困难，那么干脆不要共享，何不为每个线程创造一个资源的副本。将每一个线程存取数据的行为加以隔离，实现的方法就是给予每一个线程一个特定空间来保管该线程所独享的资源，在Java中您可以使用java.lang.ThreadLocal来实现这个功能。

ThreadLocal的作用和目的：用于实现线程内的数据共享，即对于相同的程序代码，多个模块在同一个线程中运行时要共享一份数据，而在另外线程中运行时又共享另外一份数据。

每个线程调用全局ThreadLocal对象的set方法，就相当于往其内部的map中增加一条记录，key分别是各自的线程，value是各自的set方法传进去的值。在线程结束时可以调用ThreadLocal.clear()方法，这样会更改释放内存，不调用也可以，因为在线程结束后也可以自动释放相关的ThreadLocal变量。

通过ThreadLocal，您不用撰写复杂的线程共享互斥逻辑，其意义在与：有时不共享是好的，如果共享会产生危险，那就不要共享。当然，这种方式所牺牲掉的就是空间，您必须为每一个线程保留它们独立的空间，这是一种以空间换取时间与安全性的方法。

## // ThreadLocalTest 范例（structs2模式）

**public** **class** ThreadLocalTest {

**private** **static** ThreadLocal<Integer> *x* = **new** ThreadLocal<Integer>();

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

**for**(**int** i=0;i<2;i++){

**new** Thread(**new** Runnable(){

@Override

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

**int** data = **new** Random().nextInt();

System.*out*.println(Thread.*currentThread*().getName()+" has put data : "+ data);

*x*.set(data);

MyThreadScopeData.*getThreadInstance*().setName("name"+data);

MyThreadScopeData.*getThreadInstance*().setAge(data);

**new** A().get();

**new** B().get();

}

}).start();

}

}

**static** **class** A{

**public** **void** get(){

**int** data = *x*.get();

System.*out*.println("A from "+ Thread.*currentThread*().getName()+" get data : "+ data);

MyThreadScopeData myData =MyThreadScopeData.*getThreadInstance*();

System.*out*.println("A from "+ Thread.*currentThread*().getName()+" get myData : "+ myData.getName()+","+myData.getAge());

}

}

**static** **class** B{

**public** **void** get(){

**int** data = *x*.get();

System.*out*.println("B from "+ Thread.*currentThread*().getName()+" get data : "+ data);

MyThreadScopeData myData =MyThreadScopeData.*getThreadInstance*();

System.*out*.println("B from "+ Thread.*currentThread*().getName()+" get myData : "+ myData.getName()+","+myData.getAge());

}

}

}

/\*\*

\* 设计自己线程范围内的共享对象，structs2模式

\* \*/

**class** MyThreadScopeData{

**private** MyThreadScopeData(){}

**public** **static** MyThreadScopeData getThreadInstance(){

MyThreadScopeData instance = *map*.get();

**if**(instance == **null**){

instance = **new** MyThreadScopeData();

*map*.set(instance);

}

**return** instance;

}

**private** **static** ThreadLocal<MyThreadScopeData> *map* = **new** ThreadLocal<MyThreadScopeData>();

**private** String name;

**private** **int** age;

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** **int** getAge() {

**return** age;

}

**public** **void** setAge(**int** age) {

**this**.age = age;

}

}

// ThreadLocalLoggerTest范例

**public** **class** ThreadLocalLoggerTest {

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

**new** TestThread("thread1").start();

**new** TestThread("thread2").start();

**new** TestThread("thread3").start();

}

}

**class** TestThread **extends** Thread{

**public** TestThread(String name){

**super**(name);

}

@Override

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

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

SimpleThreadLogger.*log*(getName()+": message "+i);

**try**{

Thread.*sleep*(1000);

}**catch**(Exception e){

SimpleThreadLogger.*log*(e.toString());

}

}

}

}

**class** SimpleThreadLogger {

**private** **static** **final** ThreadLocal<Logger> *threadLocal* = **new** ThreadLocal<Logger>();

// 输出信息

**public** **static** **void** log(String msg) {

*getThreadLogger*().log(Level.*INFO*, msg);

}

// 根据线程获得专属Logger

**private** **static** Logger getThreadLogger(){

Logger logger = *threadLocal*.get();

**if**(logger==**null**){

**try** {

logger = Logger.*getLogger*(Thread.*currentThread*().getName());

/\*\*

\* Logger默认是在主控台输出，我们加入一个文件输出的Handler,

\* 它会输出xml的文件记录

\* \*/

logger.addHandler(**new** FileHandler(Thread.*currentThread*()

.getName() + ".log"));

} **catch** (Exception e) {}

*threadLocal*.set(logger);

}

**return** logger;

}

}

# java5中的线程并发库java.util.concurrent

## 线程池Executors

有时候需要建立一堆线程来执行一些小任务，然而频繁地建立线程有时会开销较大。因为线程的建立必须与操作系统互动，如果能建立一个线程池(Thread Pool)来管理这些小线程并加以重复使用，对于系统效能是个改善的方式。

/\*\*

\* 线程池

\* \*/

**public** **class** ThreadPoolTest {

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

//ExecutorService threadPool = Executors.newFixedThreadPool(3);//包括固定数量的线程

//ExecutorService threadPool = Executors.newCachedThreadPool();//建立可以快速获取的线程，每个线程默认可idle的时间为60秒，池子里线程数是动态变化的

ExecutorService threadPool = Executors.*newSingleThreadExecutor*();//单一线程池，如果该线程死了，会自动再创建一个线程，循环地执行指定给它的每个任务

**for** (**int** i = 1; i <= 10; i++) {

**final** **int** task = i;

threadPool.execute(**new** Runnable() {

@Override

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

**try** {

Thread.*sleep*(20);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

**for** (**int** j = 1; j <= 10; j++) {

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

+ " is looping of " + j + " for task of "+ task);

}

}

});

}

System.*out*.println("all of 10 tasks have commited! ");

//threadPool.shutdown(); // 池子里没有任何任务了，就结束

//threadPool.shutdownNow(); //无论池子有无任务，直接结束

//创建任务调度线程池含3个线程，10s后 bombing

/\* Executors.newScheduledThreadPool(3).schedule(

new Runnable(){

@Override

public void run() {

System.out.println("bombing!");

}

},

10,

TimeUnit.SECONDS);\*/

//创建任务调度线程池含3个线程，6s后 bombing,然后每隔2s bombing一下

Executors.*newScheduledThreadPool*(3).scheduleAtFixedRate(

**new** Runnable(){

@Override

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

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

}

},

6, 2,

TimeUnit.*SECONDS*);

//可以通过下面方式实现 每天的几点 bombing ,将date.getTime()设置成 要bombing的时间值

//schedule(task,date.getTime()-System.currentTimeMillis(),TimeUnit.MILLISECONDS);

}

}

## Semaphere实现信号灯

* Semaphore可以维护当前访问自身的线程个数，并提供了同步机制。使用Semaphore可以控制同时访问资源的线程个数，例如实现一个文件允许的并发访问数。
* Semaphore实现的功能就类似厕所有5个坑，假如有10个人要上厕所，那么同时能有多少个人去上厕所呢？同时只能有5个人能够占用，当5个人中的任何一个人让开后，其中在等待的5个人中又有一个可以占用了。
* 另外等待的5个人中可以是随机获得优先机会，也可以是按照先来后到的顺序获得机会，这取决于构造Semaphore对象时传入的参数选项。
* 单个信号量的Semaphore对象可以实现互斥锁的功能，并且可以是由一个线程获得了“锁”，再由另一个线程释放“锁”，这可应用于死锁恢复的一些场合。

**public** **class** SemaphoreTest {

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

ExecutorService service = Executors.*newCachedThreadPool*();

**final** Semaphore sp = **new** Semaphore(3);

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

Runnable runnable = **new** Runnable(){

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

**try** {

sp.acquire();

} **catch** (InterruptedException e1) {

e1.printStackTrace();

}

System.*out*.println("线程" + Thread.*currentThread*().getName() +

"进入，当前已有" + (3-sp.availablePermits()) + "个并发");

**try** {

Thread.*sleep*((**long**)(Math.*random*()\*10000));

} **catch** (InterruptedException e) {

e.printStackTrace();

}

System.*out*.println("线程" + Thread.*currentThread*().getName() +

"即将离开");

sp.release();

//下面代码有时候执行不准确，因为其没有和上面的代码合成原子单元

System.*out*.println("线程" + Thread.*currentThread*().getName() +

"已离开，当前已有" + (3-sp.availablePermits()) + "个并发");

}

};

service.execute(runnable);

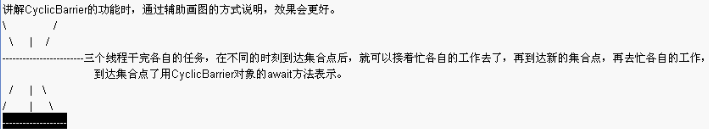
}

}

}

## CyclicBarrier

表示大家彼此等待(阻塞)，大家集合好后才开始出发，分散活动后又在指定地点集合碰面，这就好比整个公司的人员利用周末时间集体郊游一样，先各自从家出发到公司集合后，再同时出发到公园游玩，在指定地点集合后再同时开始就餐。



**public** **class** CyclicBarrierTest {

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

ExecutorService service = Executors.*newCachedThreadPool*();

**final** CyclicBarrier cb = **new** CyclicBarrier(3);

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

Runnable runnable = **new** Runnable(){

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

**try** {

Thread.*sleep*((**long**)(Math.*random*()\*10000));

System.*out*.println("线程" + Thread.*currentThread*().getName() +

"即将到达集合地点1，当前已有" + (cb.getNumberWaiting()+1) + "个已经到达，" + (cb.getNumberWaiting()==2?"都到齐了，继续走啊":"正在等候"));

cb.await(); *//想在什么地方集合，就在什么地方await()*

Thread.*sleep*((**long**)(Math.*random*()\*10000));

System.*out*.println("线程" + Thread.*currentThread*().getName() +

"即将到达集合地点2，当前已有" + (cb.getNumberWaiting()+1) + "个已经到达，" + (cb.getNumberWaiting()==2?"都到齐了，继续走啊":"正在等候"));

cb.await();

Thread.*sleep*((**long**)(Math.*random*()\*10000));

System.*out*.println("线程" + Thread.*currentThread*().getName() +

"即将到达集合地点3，当前已有" + (cb.getNumberWaiting() + 1) + "个已经到达，" + (cb.getNumberWaiting()==2?"都到齐了，继续走啊":"正在等候"));

cb.await();

} **catch** (Exception e) {

e.printStackTrace();

}

}

};

service.execute(runnable);

}

service.shutdown();

}

}

## CountDownLatch

* 犹如倒计时计数器，调用CountDownLatch对象的countDown方法就将计数器减1，当计数到达0时，则所有等待者或者单个等待者开始执行。
* 可以实现一个人（也可以是多个人）等待其他所有人都来通知他，可以实现一个人通知多个人的效果，类似裁判一声口令，运动员同时开始奔跑，或者所有运动员都跑到终点后裁判才可以公布结果，用这个功能做百米赛跑的游戏程序不错哦！还可以实现一个计划需要多个领导签字后才能继续向下实施的情况。

**public** **class** CountdownLatchTest {

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

ExecutorService service = Executors.*newCachedThreadPool*();

**final** CountDownLatch cdOrder = **new** CountDownLatch(1);

**final** CountDownLatch cdAnswer = **new** CountDownLatch(3);

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

Runnable runnable = **new** Runnable(){

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

**try** {

System.*out*.println("线程" + Thread.*currentThread*().getName() +

"正准备接受命令");

cdOrder.await();

System.*out*.println("线程" + Thread.*currentThread*().getName() +

"已接受命令");

Thread.*sleep*((**long**)(Math.*random*()\*10000));

System.*out*.println("线程" + Thread.*currentThread*().getName() +

"回应命令处理结果");

cdAnswer.countDown();

} **catch** (Exception e) {

e.printStackTrace();

}

}

};

service.execute(runnable);

}

**try** {

Thread.*sleep*((**long**)(Math.*random*()\*10000));

System.*out*.println("线程" + Thread.*currentThread*().getName() +

"即将发布命令");

cdOrder.countDown();

System.*out*.println("线程" + Thread.*currentThread*().getName() +

"已发送命令，正在等待结果");

cdAnswer.await();

System.*out*.println("线程" + Thread.*currentThread*().getName() +

"已收到所有响应结果");

} **catch** (Exception e) {

e.printStackTrace();

}

service.shutdown();

}

}

## Exchanger

用于实现两个人之间的数据交换，每个人在完成一定的事务后想与对方交换数据，第一个先拿出数据的人将一直等待第二个人拿着数据到来时，才能彼此交换数据。

**public** **class** ExchangerTest {

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

ExecutorService service = Executors.*newCachedThreadPool*();

**final** Exchanger exchanger = **new** Exchanger();

service.execute(**new** Runnable(){

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

**try** {

String data1 = "zxx";

System.*out*.println("线程" + Thread.*currentThread*().getName() +

"正在把数据" + data1 +"换出去");

Thread.*sleep*((**long**)(Math.*random*()\*10000));

String data2 = (String)exchanger.exchange(data1);

System.*out*.println("线程" + Thread.*currentThread*().getName() +

"换回的数据为" + data2);

}**catch**(Exception e){

}

}

});

service.execute(**new** Runnable(){

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

**try** {

String data1 = "lhm";

System.*out*.println("线程" + Thread.*currentThread*().getName() +

"正在把数据" + data1 +"换出去");

Thread.*sleep*((**long**)(Math.*random*()\*10000));

String data2 = (String)exchanger.exchange(data1);

System.*out*.println("线程" + Thread.*currentThread*().getName() +

"换回的数据为" + data2);

}**catch**(Exception e){

}

}

});

}

}

## 阻塞队列BlockingQueue

* 队列包含固定长度的队列和不固定长度的队列。
* 什么是可阻塞队列，阻塞队列的作用与实际应用，阻塞队列的实现原理。
* ArrayBlockingQueue
* 看BlockingQueue类的帮助文档，其中有各个方法的区别对比的表格。
* 只有put方法和take方法才具有阻塞功能。
* 用3个空间的队列来演示阻塞队列的功能和效果。
* 用2个具有1个空间的队列来实现同步通知的功能。

阻塞队列与Semaphore有些相似，但也不同，阻塞队列是一方存取数据，另一方释放数据，Semaphore通常则是由同一方设置和释放信号量。

队列(Queue)是一个先进先出(First In First Out)的数据结构。在J2SE5.0中增加了java.util.concurrent.BlockingQueue.在多线程情况下，如果BlockingQueue的内容为空，而有个线程试图从Queue中取出元素，则该线程会被Block,直到Queue有元素时才能解除Block;反过来，如果BlockingQueue满了，而有个线程试图再把数据填入Queue中，则该线程会被Block,直到Queue中有元素被取走后解除Block。

java.util.concurrent中提供了几种不同的BlockQueue 。ArrayBlockingQueue要指定容量大小来构建。LinkedBlockingQueue默认没有容量上限，但也可以指定容量上限。

PriorityBlockingQueue严格来说不是Queue，因为它是根据优先权（Priority）来移除元素。

/\*\*

\* 阻塞队列

\* \*/

**public** **class** BlockingQueueTest {

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

**final** BlockingQueue queue = **new** ArrayBlockingQueue(3);

**for**(**int** i=0;i<2;i++){

**new** Thread(){

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

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

**try** {

Thread.*sleep*((**long**)(Math.*random*()\*1000));

System.*out*.println(Thread.*currentThread*().getName() + "准备放数据!"); queue.put(1);

System.*out*.println(Thread.*currentThread*().getName() + "已经放了数据，" + "队列目前有" + queue.size() + "个数据");

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

}.start();

}

**new** Thread(){

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

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

**try** {

//将此处的睡眠时间分别改为100和1000，观察运行结果

Thread.*sleep*(1000);

System.*out*.println(Thread.*currentThread*().getName() + "准备取数据!");

queue.take();

System.*out*.println(Thread.*currentThread*().getName() + "已经取走数据，" + "队列目前有" + queue.size() + "个数据");

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

}.start();

}

}

/\*\*

\* 阻塞队列通信

\* \*/

**public** **class** BlockingQueueCommunication {

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

**final** Business business = **new** Business();

**new** Thread(

**new** Runnable() {

@Override

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

**for**(**int** i=1;i<=50;i++){

business.sub(i);

}

}

}

).start();

**for**(**int** i=1;i<=50;i++){

business.main(i);

}

}

**static** **class** Business { //内部类的实例对象不能在静态方法中创建；加static后，功能相当于外部类，然后才可以在静态方法中创建实例

BlockingQueue<Integer> queue1 = **new** ArrayBlockingQueue<Integer>(1);

BlockingQueue<Integer> queue2 = **new** ArrayBlockingQueue<Integer>(1);

/\*\*

\* 匿名构造方法，在对类实例化之前，会先运行该方法

\* \*/

{

// Collections.synchronizedMap(null); // 参数是hashmap,返回线程安全的 新Map

**try** {

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

queue2.put(1);

} **catch** (InterruptedException e) {}

}

**public** **void** sub(**int** i){

**try** {

queue1.put(1);

} **catch** (InterruptedException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

**for**(**int** j=1;j<=10;j++){

System.*out*.println("sub thread sequece of " + j + ",loop of " + i);

}

**try** {

queue2.take();

} **catch** (InterruptedException e) {}

}

**public** **void** main(**int** i){

**try** {

queue2.put(1);

} **catch** (InterruptedException e1) {}

**for**(**int** j=1;j<=100;j++){

System.*out*.println("main thread sequece of " + j + ",loop of " + i);

}

**try** {

queue1.take();

} **catch** (InterruptedException e) {}

}

}

}

## Callable&Future

java.util.concurrent.Callable与java.util.concurrent.Future类可以协助您完成Future模式。Future模式在请求发生时，会先产生一个Future对象给发出请求的客户。它的作用类似于代理（Proxy）对象，而同时所代理的真正目标对象的生成是由一个新的线程持续进行。真正的目标对象生成之后，将之设置到Future之中，而当客户端真正需要目标对象时，目标对象也已经准备好，可以让客户提取使用。

* Future取得的结果类型和Callable返回的结果类型必须一致，这是通过泛型来实现的。
* Callable要采取ExecutorService的submit方法提交，返回Future对象可以取消任务。
* CompletionService用于提交一组Callable任务，其take方法返回已完成的一个Callable任务对应的Future对象。
* 好比我同时种了几块地的麦子，然后就等待收割。收割时，则是哪块地先成熟了，则先去收割哪块麦子。

**public** **class** CallableFuture {

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

ExecutorService threadPool = Executors.*newSingleThreadExecutor*();

Future<String> future =

threadPool.submit(**new** Callable<String>(){

@Override

**public** String call() **throws** Exception {

// **TODO** Auto-generated method stub

**return** "hello";

}

});

System.*out*.println("等待结果");

**try** {

//System.out.println("拿到结果："+future.get());

System.*out*.println("拿到结果："+future.get(1,TimeUnit.*SECONDS*)); //1s内拿到结果否则会报错

} **catch** (Exception e) {

e.printStackTrace();

}

ExecutorService threadPool2 = Executors.*newFixedThreadPool*(10);

CompletionService<Integer> completionService = **new** ExecutorCompletionService<Integer>(threadPool2);

**for**(**int** i=1;i<=10;i++){

**final** **int** seq = i;

completionService.submit(**new** Callable<Integer>() {

@Override

**public** Integer call() **throws** Exception {

Thread.*sleep*(**new** Random().nextInt(5000));

**return** seq;

}

});

}

**for**(**int** i=1;i<=10;i++){

**try** {

System.*out*.println(

completionService.take().get());

} **catch** (Exception e) {

e.printStackTrace();

}

}

}

}

# 多线程会提高程序的运行效率吗？

不会，一般来说还有可能更慢，因为CPU在各个线程间切换运行会降低效率，程序性能会更低。

拷贝文件夹时，一次把这个文件夹拷贝过来比进入这个文件夹同时拷贝子文件夹要快。

那为什么会有多线程从服务器上下载速度快的说法呢？

其实你的计算机并没有快，而是去抢了服务器的带宽(服务器给你提供更多的服务)；

例如：服务器可同时为100人提供服务，为每个人提供20k；

你一个人去下载的话，一个线程代表一个人，服务器给你提供20k,你10个线程去下载，服务器就认为10个人来下载，就给你提供200k。

数组是连续的一片内存，链表是不连续的内存。

HashSet.HashMap.ArrayList.是非线程安全的。

同步集合类:

被多个线程访问用线程安全：ConcurrentHashMap

HashSet内部用的是HashMap实现的,只是没有用到HashMap的value部分。