笔记4

第14章 多线程

多线程程序在较低的层次扩展了多任务的概念 一个程序可以运行多个任务 通常 每个任务被称为一个**线程** 它是线程控制的简称 关于进程与线程的本质的区别在于每个进程是否拥有自己的一套变量 而线程则是共享数据的 **共享变量**使得线程之间的通信比进程更有效 与进程相比 线程更**轻量级**
创建 撤销 一个线程比启动新线程的开销要小的多

为了使用线程的话 主要是使用一个类和一个接口 这个接口是Runnable接口 这个类是Thread类
这俩个都有run方法 我们需要实现或者覆盖就可以 run方法就是我们线程的具体实现的方法

如果使用的是一个比较耗时的任务 应该使用独立的线程

警告：不要调用Thread类或Runnable对象的run方法 直接调用run方法 只会执行同一个线程中的任务 而不会启动新线程 应该调用Thread.start方法 这个方法将创建个执行run方法的新线程

关于中断线程

早期有一个stop方法 现在已经放弃了
现在还有一种中断的方法 就是interrupt方法 intrrupt方法可以用来请求中止线程
当对一个线程调用interrupt方法时 线程的中断位置将被重置
正确的停止线程方式是设置共享变量，并调用interrupt()（注 意变量应该先设置）。
想弄清楚线程是否被重置 我们可以调用currentThread方法获得当前进程 然后调用isInterrupted 方法

```
public void run() {
try {
do some work
while(! Thread.currentThread().isInterrupted() && more work to do) {
do more work
        }
    } catch(InterruptedException e) {
// thread was interrupted during sleep or wait (in blocked)
} finally {
        cleanup if required
    }
}
```

如果当前线程被阻塞 就无法检测中断状态 这是产生InterruptedException异常的地方 当在一个被阻塞的线程（调用sleep或wait）上调用interrupt方法时 阻塞调用将会被Interrupted Exception异常中断 被中断的线程可以决定如何响应中断
需要说明的是：**中断一个线程不过是引用它的注意** 被中断的线程可以决定如何响应中断 某些线程是如此的重要以至于应该处理完异常后继续执行 而不理会中断
中断状态被置位时调用sleep方法等阻塞方法时 它不会休眠 相反 **它会清除这一状态 并抛出InterruptedException**
不是所有的阻塞方法收到中断后都可以取消阻塞状态，输入和输出流类会阻塞等待 I/O 完成，但是它们不抛出 InterruptedException，而且在被中断的情况下也不会退出阻塞状态.
尝试获取一个内部锁的操作（进入一个 synchronized 块）是不能被中断的，但是 ReentrantLock 支持可中断的获取模式即 tryLock(long time, TimeUnit unit)。

Thread.interrupt()方法不会中断一个正在运行的线程。这一方法实际上完成的是，在线程受到阻塞时抛出一个中断信号，这样线程就得以退出阻塞的状态。更确切的说，如果线程被Object.wait, Thread.join和Thread.sleep三种方法之一阻塞，那么， 它将接收到一个中断异常（InterruptedException），从而提早地终结被阻塞状态。
在这里拥有俩种方法进行结束 一种是使用catch到异常之后 本来是处于阻塞状态下 由于中断提前出来 在catch块中使用特定的方法步骤将其停止 另外一种就是不能使用阻塞方法（因为使用阻塞方法 isInterrupted方法会失效）使用isInterrupted方法来判断是否是中断 然后进行特定的方法步骤将其停止

```
public class Ceshi {
public static void main(String[] args) {
Runnable2 runnable2 = new Runnable2();
Thread thread = new Thread(runnable2);
thread.start();

thread.interrupt();
System.out.println("已经中断。。。。");
System.out.println("中断完毕。。。");
    }
```

```
}

class Runnable2 implements Runnable {

int i = 0;

@Override
public void run() {

while (i < 4) {
try {
i++;
System.out.println("1");
if (!Thread.currentThread().isInterrupted()) System.out.println("没有被中断  非常好");
if (Thread.currentThread().isInterrupted()) i = 4;
System.out.println("i:" + i);

            } catch (Exception e) {
System.out.println("Thread catch Exception");
            }
        }
    }
}
```

下面是网上摘抄部分 上面都有说过了：

当产生异常时，有两种处理方式选择。

1) 在`catch`子句中设置中断状态。

```
catch(InterruptedException e) {
    Thread.currentThread().interrupt();
}
```

2) 不采用`try`语句块捕获异常，交给调用者处理。

```
void mySubTask() throws InterruptedException {}
```

测试当前线程是否被中断有`interrupted()`和`isInterrupted()`两个方法。
`interrupted()`是一个静态方法，有副作用，会把中断状态位置为`false`;
`isInterrupted()`是一个实例方法，无副作用。

http://blog.csdn.net/canot/article/details/51087772这是关于中断的一部分详解

线程的状态
线程有6种状态
NEW（创建）
Runnable（可运行）
Waiting（等待）
Timed waiting（计时等待）
Terminated（终止）

新创建的线程 如：NEW Thread() 这个状态是new 程序还没有运行线程中的代码

然后是**可运行状态**

一旦调用了start方法 就到了可运行状态 处于Runable 一个可运行的线程可能正在运行肯也没有运行 这取决于操作系统给线程的提供运行的时间

一旦一个线程开始运行 它不必始终保持运行 事实上 运行中的线程被中断 目的是为了让其他的线程获得运行机会 线程调度的细节依赖于操作系统的实现 **抢占式调度系统**给每一个可运行线程一个时间片来执行任务 当时间片用完 操作系统剥夺该线程的运行权 并给另一个线程运行机会 当选择一个线程时 操作系统考虑线程的优化级

现在所有的桌面和服务器操作都采用的是抢占式调度 **但是在手机端 可能采用的是协作式调度** 在这样的设备中 一个线程只有在调用yield方法 或者阻塞或等待，线程才失去控制权

在具有多个处理器的机器上 每一个处理器运行一个线程 可以有多个线程并行运行 当然 如果线程的数目多与处理器的数目 调度器依然采用时间片机制

在任何给定时刻 一个可运行的线程可能正在运行也可能没有运行

被阻塞线程与等待线程
当线程处于阻塞或等待状态时 它不运行任何代码并且消耗最小 直到线程调度器激活它 细节取决它是怎样到达非活动状态的
1 当一个线程视图获取一个内部的对象锁 而该锁被其他线程持有 则该线程进入阻塞状态 当所有其他线程释放该锁 并且线程调度器允许本线程持有它的时候 该线程变成非阻塞状态

2 当线程等待另一个线程通知调度器一个条件时 它自己进行等待状态
在调用Object.wait或者Thread.join方法 或者等待java.util.concurrent库中Lock或Condition时 就会出现这种情况 实际上 阻塞状态与等待状态时是很大不同的

3 有几个方法有一个超时参数 调用它们导致线程进入**计时等待**
带有延时参数的方法有Thread.sleep，Object.wait,Thread.join,Lock,tryLock以及Condition.await
当一个线程被阻塞或等待时 另外一个线程处于运行状态 然后这个线程重新被激活时 然后对比与当前的线程的优先级是否更高 如果是的就剥夺当前运行的一个线程 将这个填上去

线程终止
因为俩个原因被终止
1 run方法正常退出而死亡
2 有一个异常强制终止了run方法而意外死亡

还有另外一种 就是stop强制死亡该方法抛出ThreadDeath错误对象 由此杀死线程 不推荐使用stop方法了

线程属性：

线程优先级 守护线程 线程组 处理未捕获异常的处理器

线程的优先级
在java设计语言中 每一个线程都有一个优先级 默认情况下 一个线程继承它的父类的优先级 可以用setPriortiy方法来提高或降低任何一个线程的优先级

每当线程调度器有机会选择新线程时 它首先选择具有较高优先级线程 但是 线程优先级是高度依赖于系统的 所以在不同的平台是不同的优先级

注意 不要将线程的构建为功能的正确性依赖于优先级

每将调度器调度一个新的线程时 首先会在具有高优先级的线程中进行选择 **尽管这样会使优先级低的线程完全饿死**

守护线程
在Java中有两类线程：User Thread(用户线程)、Daemon Thread(守护线程)
，任何一个守护线程都是整个JVM中所有非守护线程的保姆：
只要当前JVM实例中尚存在任何一个非守护线程没有结束，守护线程就全部工作；只有当最后一个非守护线程结束时，守护线程随着JVM一同结束工作
可以通过

```
thread.setDaemon(true);
thread.start();
```

这个方法进行将一个线程变为守护线程 注意：这个守护线程必须要在start之前使用

守护线程应该永远不去访问固有资源 比如很典型的一个守护线程：垃圾回收器

未捕获异常处理器

线程的run方法不能抛出任何被检测的异常 但是 不被检测的异常会导致线程的终止 在这种情况下 线程就死亡了
但是 不需要任何catch子句来处理的传播的异常 相反 **就在线程死亡之前 异常被传递到一个用于未捕获异常的处理器**
该处理器必须实现一个Thread.UncaughtExceptionHandler接口的类

也可以使用**Thread类的静态方法setDefaultUncaughtExceptionHandler 为所有的线程来安装一个默认的处理器** 替换处理器可以使用日志API发送未捕获的异常的报告到日志文件

如果不安装默认的处理器 默认的处理器为空 但是 如果**不为独立的线程安装处理器 此时的处理器就是该线程的ThreadGroup对象**
关于线程组 是一个管理线程的集合 但是这个线程组 现在已经有了更好的解决办法 不建议使用

ThreadGroup类实现Thread.UncaughtExceptionHandler接口 它的uncaughtException方法操作如下：

1 如果该线程组有父线程组 那么父线程组的uncaughtException方法被调用
2 否则 如果Thread.getDefaultExceptionHandler方法返回一个非空的处理器 则调用该处理器
3 否则 如果Throwable是ThreadDeath的一个实例 什么都不做（ThreadDeath是stop的一个实例 但是现在已经弃用）
4 否则 线程的名字以及Throwable的栈踪迹被输出到System.err上

这是你在程序中看到许多次的栈踪迹

```
public class Test {
public static void main(String[] args) {
System.out.println("创建一个新的线程");
ExceptionThread2 thread2 = new ExceptionThread2();
Thread t = new Thread(thread2);
Thread.setDefaultUncaughtExceptionHandler(new MycaughtUnCheckExceptionHandler());//这是给所有的设置默认的异常处理器
//t.setUncaughtExceptionHandler(new MycaughtUnCheckExceptionHandler());//这是设置单独的一个线程的异常处理器
t.start();
System.out.println("eh121=" + t.getUncaughtExceptionHandler());

}

}


class ExceptionThread2 implements Runnable {

@Override
public void run() {
        Thread t = Thread.currentThread();
System.out.println("run() by " + t);
System.out.println("eh=" + t.getUncaughtExceptionHandler());
        throw  new RuntimeException("抛出运行时异常 ");


}
}

class  MycaughtUnCheckExceptionHandler implements Thread.UncaughtExceptionHandler{
@Override
public void uncaughtException(Thread t, Throwable e) {
System.out.println("捕捉了到线程 \t" + t + "\n 捕捉到了异常" + e);
}
}
```

同步

竞争条件的一个列子

为了避免多线程引起对共享数据的讹传 必须要学习同步存取

是否原子操作是十分重要的

我们这里实现了一个关于银行的案例 代码太多就不全部贴上来了
结果出错了 至于原因
如果进行相关的虚拟机下的字节码查看的话 就可以发现重要的增殖命令是由几条指令组成的 执行他们的线程可能在任何一条指令上被中断

如果删除打印语句或者其他消耗资源的方法 讹传的风险就低一点 因为每个线程在再次睡眠之前的工作量少 调度器在计算中剥夺线程的运行权可能性很小
主要就是在于cpu的使用情况 如果负载好就还好 如果负载很大 出错依然会存在

真正的问题是**具体变量的执行方法的执行过程中可能被中断** 如果能保证线程在失去控制之前方法运行完成 那么状态就永远不会出现讹传

java语言提供了一个**synchroized关键字**达到这一目的 并且在 JAVA SE5引入了ReentranLock类

synchronize关键字提供了一个锁以及相关的条件

ReentrantLock保护代码块的结构如下
myLock.lock();
try{

}finally{
myLock.lock();

}
这确保了任何时刻只有一个线程进入临界区 一旦一个线程封锁了锁对象 其他任何子句都无法通过lock语句
把解锁操作括在finally子句之内是很重要的 不然无法释放这个线程 其他线程也永远被堵塞
如果使用了锁 就不能用带资源的try语句

下面未改之前的transfer方法

```
public void transfer(int from, int to, double amount)
{
if (accounts[from] < amount) return;
System.out.print(Thread.currentThread());
accounts[from] -= amount;
System.out.printf(" %10.2f from %d to %d", amount, from, to);
accounts[to] += amount;
System.out.printf(" Total Balance: %10.2f%n", getTotalBalance());
}
```

下面是改过之后的

```
public void transfer(int from, int to, double amount) {
bankLock.lock();
    try {
if (accouts[from] < amount) return;
System.out.println(Thread.currentThread());
accouts[from] -= amount;
System.out.printf("%10.2f from %d to %d ", amount, from, to);
accouts[to] += amount;
System.out.printf("Total Balance: %10.2f%n", getTotalBalance());
}finally {
bankLock.unlock();
}

}
```

改过就不会出现上面未改过的状况了

注意 每一个的Bank对象都有自己的ReentranLock对象 如果俩个线程视图访问一个Lock对象 那么锁以串行的方式提供服务 但是 如果俩个线程访问不同的的Bank对象 每一个线程得到不同的线程对象 俩个线程不会阻塞

锁是可重入的 因为线程可以重复获得已持有的锁 锁保持一个**持有计数** 来跟踪对lock方法的嵌套使用 线程在每一次调用lock都要调用unlock来释放锁 由于这一特性 被一个锁保护的代码可以调用另一个使用相同的锁的方法

比如 transfer调用了getTotalBalance放 这也会封锁bankLock对象 当BankLock对象的持有计数为2 当getTotalBalance方法退出的时候 持有计数变了1 当transfer退出的时候 持有计数变成了0 线程释放锁

如果要保护一个经常被更新或检测共享对象的代码块 要确保这些操作是否完整 是否完成 然后另一个线程才能使用相同的对象

注意
要**留意临界区的代码** 不要因为异常就跳出了临界区 **如果在临界区代码结束之前抛出了异常 finally子句将释放锁 但会使对象可能处于一种受损状态**

条件对象
条件变量需要与锁绑定
通常 线程进入临界区 却发现在某一条件上满足之后才能执行 要使用一个条件对象来管理那些已经获得一个锁但是不能做有用工作的线程

注意下面这段代码

```
if (bank.getBalance(from)>=amount) bank.transfer(from,to,amout);
```

有可能是在判断之前就执行了后面的代码 这就出问题了

所以说 当这个账号没有足够的余额时 应该等待另外一个线程注入资金 但是 这一个线程刚刚获得了BankLock的排他性访问 因此 其他线程没有访问的机会 这也是我们需要条件对象的时候
一个锁对象可以有一个或多个相关的调节对象 可以用newCondition方法获得一个条件对象 习惯上给每一个对象命名为可以反映它所代表的条件的名字

```
private Condition sufficientFunds;
```

如果transfer这个转移方法发现余额不足 直接调用sufficientFunds.await();
当前线程被阻塞了 并放弃了锁 使得另外一个线程可以进行增加账户余额的操作

等待获得锁的线程和调用await方法的线程存在**本质的不同** 一旦一个线程调用await方法 它进入该条件的等待集 当锁可用时 该线程不能马上解除阻塞 相反 它处于阻塞状态 直到另一个线程调用同一条件上的singAll方法为止
当另一个线程转账时 应该调用
sufficientFunds.singAll();方法

这一调用**重新激活因为这一条件而等待的所有线程** 它们中的某个将从await调用返回 获得该锁并从阻塞的地方**继续执行**
线程应该再次测试该条件 由于无法确保该条件被满足 ---sigalAll方法**仅仅是通知正在等待的线程** ：**此时可能已满足条件 值得再去检测该条件**

至关重要的最终需要某个其他线程调用singlAll方法 当一个线程调用await方法时 它没有办法重新激活自身 它寄希望于其他线程 如果没有其他线程来重新激活等待的线程 它就永远不再运行了 它将导致令人不快的**死锁现象**
应该何时调用signalAll呢 **经验上讲 在对象的状态有利于等待线程的方向改变时调用siganAll**

注意调用signalAll**不会立即激活一个线程 它仅仅接触等待线程的阻塞** 以便这些线程可以在当前线程退出同步方法之后 通过竞争条件实现对象的访问

synchronized关键字

锁用来保护代码片段 任何时刻只能有一个线程执行被保护的代码
锁可以管理试图进入被保护代码的片段
锁可以有一个或多个相关的条件对象
每个条件对象管理那些已经进入被保护代码但还不能运行的线程

Lock和Condition接口为程序设计人员提供**了高度的锁定机制** 在大多数情况下 是不需要这么的控制的 并且可以使用一种嵌入到java语言内部的机制
从JavaSe1.0版开始 **java的每一个对象都有一个内部锁** 如果一个方法用**synchronized关键字声明** 那么**对象的锁将保护整个方法 也就是说 要调用该方法 线程必须获得内部的对象锁**

下面贴代码示例：

```
/*1*/
public void transfer(int from, int to, double amount) {
bankLock.lock();

    try {
/*这里的while循环不能换成if条件语句，因为被别的线程用signalAll方法唤醒的线程
        ，仅仅是条件可能满足了，而不是条件一定满足了，
        如果不用while循环继续检测的话，就会造成条件不满足的线程继续向下执行，从而产生错误。*/
while (accouts[from]<amount)
sufficientFunds.await();
System.out.println(Thread.currentThread());
accouts[from] -= amount;
System.out.printf("%10.2f from %d to %d ", amount, from, to);
accouts[to] += amount;
System.out.printf("Total Balance: %10.2f%n", getTotalBalance());
sufficientFunds.signalAll();

} catch (InterruptedException e) {
        e.printStackTrace();
} finally {
bankLock.unlock();
}

 }
/*2*/
public  synchronized void transfer2(int from, int to, double amount) throws InterruptedExcept
```

```
ion {
while (accouts[from]<amount)    wait();
System.out.println(Thread.currentThread());
accouts[from] -= amount;
System.out.printf("%10.2f from %d to %d ", amount, from, to);
accouts[to] += amount;
System.out.printf("Total Balance: %10.2f%n", getTotalBalance());
notifyAll();

}
```

1 与2 的方法代码其效果是一样 只不过synchronized关键字是不需要显示的声明一个锁

内部对象锁只有一个相关条件 wait方法添加一个线程到等待集中 notifyAll/notify方法解除等待线程的阻塞操作 换句话说 **调用wait或notifyAll是等价于**
await与signalAll方法的 (wait方法与notifyAll/notify方法是Object的final方法 **)**

使用synchronized关键字比编写代码要简洁的多 当然 需要理解这一代码 就**必须明白每一个java对象都有一个内部锁** 并且该锁有一个内部条件 由锁来管理那些试图进入synchronized方法的进程 由条件来管理那些调用wait的线程

将**静态方法声明为synchronized**也是合法的 如果调用这种方法 **该方法获得相关的类对象的内部锁**
列如：如果Bank有一个静态同步的方法 如果调用这方法 这个Bank.class就被锁住 因此 没有其他线程可以调用同一个类的这个或任何其他的同步静态方法

内部锁与条件在一起有一些限制

不能中断一个正在试图获得锁的线程
试图获得锁不能设定超时
每个锁仅有单一的条件 可能是不够的

有几个建议：
在代码中 最好不要使用synchronized也不要使用Lock/Condition 在许多情况下可以使用java.util.concurrent包中的一种机制 它会处理所有的加锁

如果synchronized关键字适合程序的话 尽量选择synchronized 因为简洁 减少出错

只有在对Lock/Condition有特殊要求时 才去使用Lock/Condition

同步阻塞

每一个java对象都有一个锁 线程可以通过调用同步方法获得锁 还有另一种机制可以获得锁 (其他不是当前的锁 而是其他的) 通过进入一个同步阻塞

```
synchronized (obj){

}
```

于是获得了obj的锁
有时会发现 特别的锁

```
private  final  double[] accouts;

        private Object lock = new Object();

    public Bank(double[] accouts) {
         this.accouts = accouts;
    }

public void transfer(int from, int to, double amount) {

synchronized (lock) {
accouts[from] -= amount;
accouts[to] += amount;

}
    }
```

```
}
```

在此 lock对象被创建仅仅是用来使用每个Java对象持有的锁

有时程序员使用一个对象的锁来实现额外的原子操作 实际上成为客户端锁定 比如Vector类 一个列表 它的方法是同步的 下面是代码实现

```
public   void transfer3(Vector<Double> accounts2,int from, int to, double amount)  {
  accounts2.set(from,accounts2.get(from)-amount);
accounts2.set(to, accounts2.get(to) + amount);
System.out.println("....");

}
```

vector类的get方法和set方法是同步的 但是 在第一层调用完成之后 一个线程完全有可能在transfer方法中剥夺运行权 于是 另一可能在相同位置的存储位置存入不同的值 （虽然Vector类的get和set方法时同步的，但是，并没有什么用。在get之后set之前，线程完全有可能被剥夺运行权。于是，另一个线程可能在相同的存储位置存入不同的值。但是可以如下修改：）
但是 我们可以截获这个锁

```
public   void transfer3(Vector<Double> accounts2,int from, int to, double amount)  {
synchronized (accounts2) {
     accounts2.set(from, accounts2.get(from) - amount);
accounts2.set(to, accounts2.get(to) + amount);
System.out.println("....");
}
}
```

这样可以工作 但是 Vector类必须要对自己的所有可修改方法都能使用内部锁 否则就可能出错 但是Vector文档并没有这么说
所以客户端锁定是非常脆弱的 通常不建议使用

监视器
锁和条件不是一种面向对象的概念 研究人员一直在视图寻找一种办法 可以在不需要程序考虑加锁的情况下 就能保证多线程的安全性
监视器就是其中最成功的解决办法之一
监视器的概念：
http://ifeve.com/monitors-java-synchronization-mechanism/
其中监视器有如下特性

监视器只包含私有域的类
每个监视器类对象有一个相关的锁
使用该锁对所有的方法进行加锁
该锁可以有任意多个相关条件

但是java对象有不同于监视器的几个地方
域不要求是private的
方法不要求必须是synchronized
内部锁对客户是可用的

使得监视器在java中线程的安全性有所下降

关于synchronized关键字的说明
http://www.cnblogs.com/gnagwang/archive/2011/02/27/1966606.html

Volatile域

为了读写一个或俩个实例域就使用同步 实在是有点浪费性能 如果我们不用的话 在什么情况容易出错呢 使用现代的处理器与编译器 出错的可能性较大

多处理器的计算机能够暂时在寄存器或本地内存缓冲区保存内存的值 结果是 运行在不同处理器上的线程可能在同一个内存位置取到不同的值

编译器可以改变指令执行顺序以使吞吐量最大化
但是编译器假定内存的值仅仅在代码有显示的修改指令时才改变 然而 内存的值可以被另一个线程改变

原子性与可见性 有序性 都有问题
http://blog.csdn.net/itachi85/article/details/50274169 这里有相关的博客讲解

如果使用锁来保护可以保护多个线程访问的代码 可以不考虑这个问题

一旦一个共享变量被volitile修饰之后 那么就具备了俩层语义

- 保证了不同线程对这个变量进行操作时的可见性，即一个线程修改了某个变量的值，这新值对其他线程来说是立即可见的。
- 禁止进行指令重排序。

```
//线程1
boolean stop = false;
while(!stop){
    doSomething();
}

//线程2
stop = true;
```

很多人在中断线程时可能都会采用这种标记办法。但是事实上，这段代码会完全运行正确么？即一定会将线程中断么？不一定，也许在大多数时候，这个代码能够把线程中断，但是也有可能会导致无法中断线程（虽然这个可能性很小，但是只要一旦发生这种情况就会造成死循环了）。

为何有可能导致无法中断线程？每个线程在运行过程中都有自己的工作内存，那么线程1在运行的时候，会将stop变量的值拷贝一份放在自己的工作内存当中。那么当线程2更改了stop变量的值之后，但是还没来得及写入主存当中，线程2转去做其他事情了，那么线程1由于不知道线程2对stop变量的更改，因此还会一直循环下去。
但是用**volatile修饰之后就变得不一样了**：

- 使用volatile关键字会强制将修改的值立即写入主存；
- 使用volatile关键字的话，当线程2进行修改时，会导致线程1的工作内存中缓存变量stop的缓存行无效；
- 由于线程1的工作内存中缓存变量stop的缓存行无效，所以线程1再次读取变量stop的值时会去主存读取。

这里需要警告的是 **Volitile变量是不能提供原子性的操作的**

synchronized关键字是防止多个线程同时执行一段代码，那么就会很影响程序执行效率，而volatile关键字在某些情况下性能要优于synchronized，但是要注意volatile关键字是无法替代synchronized关键字的，因为volatile关键字无法保证操作的原子性

。通常来说，使用volatile必须具备以下2个条件：

对变量的写操作不依赖于当前值

- 该变量没有包含在具有其他变量的不变式中

第一个条件就是不能是自增自减等操作，上文已经提到volatile不保证原子性。
第二个条件我们来举个例子它包含了一个不变式 ：下界总是小于或等于上界

```
public class NumberRange {
    private volatile int lower, upper;
    public int getLower() { return lower; }
    public int getUpper() { return upper; }
    public void setLower(int value) {
        if (value > upper)
            throw new IllegalArgumentException(...);
        lower = value;
    }
    public void setUpper(int value) {
        if (value < lower)
            throw new IllegalArgumentException(...);
        upper = value;
    }
}
```

https://www.nowcoder.com/discuss/4324?type=0&order=0&pos=199&page=1

这种方式限制了范围的状态变量，因此将 lower 和 upper 字段定义为 volatile 类型不能够充分实现类的线程安全，从而仍然需要使用同步。否则，如果凑巧两个线程在同一时间使用不一致的值执行 setLower 和 setUpper 的话，则会使范围处于不一致的状态。例如，如果初始状态是 (0, 5)，同一时间内，线程 A 调用 setLower(4) 并且线程 B 调用 setUpper(3)，显然这两个操作交叉存入的值是不符合条件的，那么两个线程都会通过用于保护不变式的检查，使得最后的范围值是 (4, 3)，这显然是不对的。

Volatile 变量是一种非常简单但同时又非常脆弱的同步机制，它在某些情况下将提供优于锁的性能和伸缩性

这下面俩个链接是关于JVM内存模型的具体讲解

http://www.infoq.com/cn/articles/java-memory-model-1
http://wiki.jikexueyuan.com/project/java-memory-model/reorder.html

关于volatile的具体使用模式讲解
http://www.ibm.com/developerworks/cn/java/j-jtp06197.html

final变量

在上面已经知道 除非使用锁或者volatile修饰符 否则无法多个线程安全的读取一个域

与前面介绍的锁和volatile相比较，**对final域的读和写更像是普通的变量访问**（定义为final的字段只初始化一次，在正常情况下将不能再被改变）。对于final域，**编译器和处理器要遵守两个重排序规则：**

1. 在**构造函数内**对一个final域的写入，与随后把这个被构造对象的引用赋值给一个引用变量，这两个操作之间不能重排序。
2. 初次读一个包含final域的对象的引用，与随后初次读这个final域，这两个操作之间不能重排序。

final Map<String,Double>accounts=new HashMap();
其他线程会在构造函数完成构造之后才看到这个accounts

如果不是使用final的话 就不能保证其他线程看到的是accounts更新后的值 它们可能都只是看到null 而不是新构建的HashMap<>
当然 **对这个映射表的操作并不是线程安全的 如果多个线程在读写这个映射表 仍然需要同步**

原子性

假设对共享变量除了赋值之外并不完成其他操作 那么可以将这些共享变量声明为volatile
在java.util.concurrent.atomic包中有很多类使用了很高效的机器级指令（而不是使用锁）来保证其他操作的原子性

死锁
在多线程中 锁和条件不能解决多线程中的所有问题
账户 1 $200
账户 2 $300
线程1 从账户一转移300元到账户2
线程2 从账号2转移400元到账户一
这样就出错了 这就是死锁的一种形式 因为账户一和账户二的余额都不能完成工作 俩个线程无法执行下去 这样就有可能因为每一个线程要等待更多的钱款存入而导致所有的线程都阻塞 这样的状态被称为死锁

我们回到开始的列子 银行转账
1如果我们将最大金额改变不为1000元 就会进行出错 也是上面写的原因
2 如果我们将toaccount与formaccount的参数变换位置的话 也会导致出错 这是因为线程一直在试图想转账超过这个账号目前拥有的金额
3 还有一种 就是将signalAll方法改为signal方法 也会导致出错 根本原因是 它仅仅是一个线程解锁 而且 它很有可能选择一个不能继续运行的线程

但是 **java编程语言是没有任何东西可以避免或打破这种死锁现象 必须仔细设计程序 以确保不会出现死锁**

线程局部变量
有时可能要避免共享变量 使用ThreadLocal辅助类为各个线程提供各自的实例

比如：
有下面这么一个静态变量

```
public static final SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");
```

如果俩个线程都执行如下操作

```
String dateStamp = dateFormat.format(new Date());
```

因为dateFormat使用的内部数据结构可能会被并发的访问所破坏 当然可以使用同步 但开销很大 或者也可以在需要时构造时构造一个局部SimpleDateFormat 但是这也很浪费

如果要为每个线程构造一个实例 可以像下面一样

```
public static final ThreadLocal<SimpleDateFormat> dateFormat = new ThreadLocal<SimpleDateFormat>(){
protected SimpleDateFormatinitialValue() {
return new SimpleDateFormat("yyyy-MM-dd");
}
};
```

要访问具体的格式化方法 可以像下面这样调用就可以达到 调用get方法会返回当前线程的那个实例

```
String dateStamp = dateFormat.get().format(new Date());
```

在多线程中生成随机数也会出现这样的问题

Random类是线程安全的 但是如果多个线程需要等待一个共享的随机数生成器 这会很低效
可以使用ThreadLocal辅助类为各个线程提供一个单独的生成器 不过Java SE7还另外提供了一个类 只需要做以下调用

```
int random= ThreadLocalRandom.current().nextInt(upperBound);
```

ThreadLocalRandom.current()调用会返回特定于当前线程的Random实例

http://blog.csdn.net/ruyun126/article/details/5955260ThreadLocal原理

锁测试与超时
下面的链接是关于锁测试与超时的博客
http://blog.csdn.net/xxxknight/article/details/47057417
线程在调用lock方法来获得另一个线程所持有的锁的时候 很可能发生阻塞 应该更加谨慎的申请锁 tryLock方法试图申请一个锁 在成功获得锁之后返回true 否则 立即返回false 而且线程可以立即离开去做其他事情
可以调用tryLock时，使用超时参数

```
if(myLock.tryLock()){
try{}finally {
myLock.unlock();
}
 }else{
//dosomething
}
```

可以调用tryLock时 使用超时函数 像这样

```
if(myLock.tryLock(100, TimeUnit.MICROSECONDS)){
```

lock方法不能被中断。如果一个线程在等待获得一个锁时被中断，中断线程在获得锁之前一直处于阻塞状态。如果出现死锁，那么，lock方法就无法终止。
如果调用带有超时参数的trylock 那么如果线程在等待期间被打断 将抛出InterruptedException异常 这是一个非常有用的特性 可以打断异常

也可以调用lockInterruptibly方法 它就相当于一个超时设为无限的trylock方法
在等待一个条件时 也可以设置超时的 这在前面有说过

```
myCondition.await(100, TimeUnit.DAYS);
```

如果一个线程被另一个线程通过调用signalAll或signal激活 或时限已到 或线程被中断 那么await方法将返回

如果等待的线程被中断 await方法将抛出一个InterruptedException异常 如果你希望出现这种情况时线程继续等待 可以使用awaitUninterruptibly方法代替await

读/写锁
在java.util.concurrent.locks包类中定义了俩个类 我们已经讨论过的ReentrantLock类和ReentrantReadWriteLock类 如果很多线程从一个数据结构读取数据而很少线程修改其中数据的话 后者是十分有用的 在这种情况下 **允许对读者线程共享是合适的 当然 写者线程依然是必须是互斥访问的** （多个读锁之间是不需要互斥的(读操作不会改变数据，如果上了锁，反而会影响效率）)这里粘贴一个使用的博客 http://blog.csdn.net/lzm1340458776/article/details/27964243

```
//构造一个ReentrantReadWriteLock对象
private ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();
//抽取读锁和写锁
private Lock readLock = rwl.readLock();
```

```
private Lock writeLock = rwl.writeLock();
//对所有的获取方法加读锁
public  double getTotalBalance() {
readLock.lock();
    try{
//......
}finally {
readLock.unlock();
}
return 0;
}
//对所有的修改方法加写锁
public  void transfer(/*......*/) {
writeLock.lock();
    try{/*...*/}
finally {
writeLock.unlock();
}
}
```

关于为什么弃用Stop和suspend方法

在初始版的java中 采用了俩个方法来进行对线程的控制 一个是stop方法来进行终止线程 一个是suspend方法来组织一个线程直至另一个线程调用resume **stop方法和suspend方法都有一个共同点 都试图控制一个给定线程的行为**

其实stop方法天生就不安全，因为它在终止一个线程时会强制中断线程的执行，不管run方法是否执行完了，并且还会释放这个线程所持有的所有的锁对象。这一现象会被其它因为请求锁而阻塞的线程看到，使他们继续向下执行。这就会造成数据的不一致，我们还是拿银行转账作为例子，我们还是从A账户向B账户转账500元，我们之前讨论过，这一过程分为三步，第一步是从A账户中减去500元，假如到这时线程就被stop了，那么这个线程就会释放它所取得锁，然后其他的线程继续执行，这样A账户就莫名其妙的少了500元而B账户也没有收到钱。注意 这个过程是无法知晓的 这就是stop方法的不安全性。

suspend方法为什么被弃用

suspend被弃用的原因是因为它会造成死锁。suspend方法和stop方法不一样，它不会破换对象和强制释放锁，**相反它会一直保持对锁的占有，一直到其他的线程调用resume方法，它才能继续向下执行**。

假如有A，B两个线程，A线程在获得某个锁之后被suspend阻塞，这时A不能继续执行，线程B在或者相同的锁之后才能调用resume方法将A唤醒，但是此时的锁被A占有，B不能继续执行，也就不能及时的唤醒A，此时A，B两个线程都不能继续向下执行而形成了死锁。这就是suspend被弃用的原因。

这种状况在图形界面中经常发生。
这里需要注明 这本书里用的是图形化的例子来说明suspend为什么被弃用

如果想安全的挂起线程，引入一个变量suspendRequested并在run方法安全的地方测试它，安全的地方是指该线程没有封锁其他线程需要的对象的地方。当线程发现如果变量已被设置，将会保持等待状态直到它再次被获得为止。

```
1.  public volatile boolean suspendRequested = false;
2.      private Lock suspendLock = new ReentrantLock();
3.      private Condition suspendCondition = suspendLock.newCondition();
4.
5.      public void run() {
6.
7.          while (true) {
8.              if (suspendRequested) {
9.                  suspendLock.lock();
10.                 try {
11.                     while (suspendRequested)
12.                         suspendCondition.await();
13.                 } catch (Exception e) {
14.
15.                 } finally {
16.                     suspendLock.unlock();
17.                 }
18.             }
19.         }
20.     }
21.
22.     public void requestSuspend() {
23.         suspendRequested = true;
24.     }
25.
26.     public void requestResume() {
```

```
27.         suspendRequested = false;
28.         suspendLock.lock();
29.         try {
30.             suspendCondition.signalAll();
31.         } catch (Exception e) {
32.             e.printStackTrace();
33.         } finally {
34.             suspendLock.unlock();
35.         }
36.     }
```

http://blog.csdn.net/xingjiarong/article/details/47984659 这里是博客相关的讲解

阻塞队列

对于实际的编程开发来说 应该尽可能的远离底层结构 使用由并发处理的专业人士实现的较高层次的结构要方便的安全的多

对于许多安全问题 都可以使用一个或多个**队列**以优雅的方式将其形式化 生产者线程向队列插入元素 消费者线程则取出他们 **使用队列 可以安全的从一个线程向另外一个线程传递数据**

当试图向队列添加元素而队列已满 或是想从队列移出元素而队列为空的时候 阻塞队列导致线程阻塞 在协调多个线程之间的合作 阻塞队列是一个有用的工具 工作者线程可以周期性的将中间结果存储在阻塞队列中 其他的工作者线程移出中间结果并进一步加以修改 队列会自动地平衡负载

1.非阻塞队列中的几个主要方法：

add(E e):将元素e插入到队列末尾，如果插入成功，则返回true；如果插入失败（即队列已满），则会抛出异常；

remove()：移除队首元素，若移除成功，则返回true；如果移除失败（队列为空），则会抛出异常；

offer(E e)：将元素e插入到队列末尾，如果插入成功，则返回true；如果插入失败（即队列已满），则返回false；

poll()：移除并获取队首元素，若成功，则返回队首元素；否则返回null；

peek()：获取队首元素，若成功，则返回队首元素；否则返回null

对于非阻塞队列，一般情况下建议使用offer、poll和peek三个方法，不建议使用add和remove方法。因为使用offer、poll和peek三个方法可以通过返回值判断操作成功与否，而使用add和remove方法却不能达到这样的效果。注意，非阻塞队列中的方法都没有进行同步措施。

2.阻塞队列中的几个主要方法：

阻塞队列包括了非阻塞队列中的大部分方法，上面列举的5个方法在阻塞队列中都存在，但是要注意这5个方法在阻塞队列中都进行了同步措施。除此之外，阻塞队列提供了另外4个非常有用的方法：

put(E e)

take()

offer(E e,long timeout, TimeUnit unit)

poll(long timeout, TimeUnit unit)

put方法用来向队尾存入元素，如果队列满，则等待；

take方法用来从队首取元素，如果队列为空，则等待；

offer方法用来向队尾存入元素，如果队列满，则等待一定的时间，当时间期限达到时，如果还没有插入成功，则返回false；否则返回true；

poll方法用来从队首取元素，如果队列空，则等待一定的时间，当时间期限达到时，如果取到，则返回null；否则返回取得的元素

方法\处理方式	抛出异常	返回特殊值	一直阻塞	超时退出
插入方法	add(e)	offer(e)	put(e)	offer(e,time,unit)
移除方法	remove()	poll()	take()	poll(time,unit)
检查方法	element()	peek()	不可用	不可用

- 抛出异常：是指当阻塞队列满时候，再往队列里插入元素，会抛出IllegalStateException("Queue full")异常。当队列为空时，从队列里获取元素时会抛出NoSuchElementException异常 。
- 返回特殊值：插入方法会返回是否成功，成功则返回true。移除方法，则是从队列里拿出一个元素，如果没有则返回null
- 一直阻塞：当阻塞队列满时，如果生产者线程往队列里put元素，队列会一直阻塞生产者线程，直到拿到数据，或者响应中断退出。当队列空时，消费者线程试图从队列里take元素，队列也会阻塞消费者线程，直到队列可用。
- 超时退出：当阻塞队列满时，队列会阻塞生产者线程一段时间，如果超过一定的时间，生产者线程就会退出

还有带有超时的offer方法和poll方法的变体 下面是尝试在100毫秒内在队列的尾部插入一个元素 如果成功返回true 否则 达到超时时 就返回null
boolean success=q.offer(x,100,TimeUnit.MTLLSECONDS);
下面的调用尝试在100毫秒的时间内移出队列的头元素 如果成功返回头元素 否则 达到超时时 返回null
Object head =q.poll(100mTimeUnit.MILLISECONDS);
如果队列慢 则put方法阻塞 如果队列空 则take方法阻塞 在不带超时参数时 offer和poll方法等效
JDK7提供以下7种阻塞队列

- ArrayBlockingQueue ：一个由数组结构组成的有界阻塞队列。
- LinkedBlockingQueue ：一个由链表结构组成的有界阻塞队列。
- PriorityBlockingQueue ：一个支持优先级排序的无界阻塞队列。
- DelayQueue：一个使用优先级队列实现的无界阻塞队列。
- SynchronousQueue：一个不存储元素的阻塞队列。
- LinkedTransferQueue：一个由链表结构组成的无界阻塞队列。
- LinkedBlockingDeque：一个由链表结构组成的双向阻塞队列

在concurrent包中提供了阻塞队列的几个变种 默认情况下 **LinkedBlockingQueue**的容量是没有上边界的 但是 也可以选择指定最大容量 **LinkedBlockingDeque**是一个双端的版本 **ArrayBlockingQueue** 在构造时需要指定容量 并且一个可选的采纳数来指定是否需要公平性
若设置公平参数 则那么等待了最长时间的线程会优先得到处理 通常 公平性会降低性能 只有确实非常需要时才使用它
PriorityBlockingQueue是一个带优先级的队列 而不是先进先出的队列 元素按照它们的优先级顺序被移出
该队列是没有容量上限 但是 如果队列是空的 取元素的超时会阻塞
最后**DelayQueue**包含实现了Delayed接口的对象

```
interface  Delayed extends Comparator<Delayed>
{
long getDelay(TimeUnit unit);
}
```

getDealy方法返回对象的残留延迟 负值表示延迟已经结束 元素只有在延迟用完的情况下才会从DelayQueue移出 还必须实现compareTo方法 DelayQueue使用该方法对元素进行排序
元素只有在延迟用完的情况下才能从DelayQueue移出
另外在java7增加了一个**TranferQueue**接口 允许生成线程等待 直到消费者准备就绪可以接受一个元素
如果调用 q.transfer(item);这个调用会阻塞 直到另一个线程将元素(item)删除 **LinkedTransferQueue**类实现了这个接口
下面是一个例子

```
package com.javaMultitThread;

import java.io.*;
import java.util.*;
import java.util.concurrent.*;
/**
 * Created by han on 2016/8/1.
 */
public class blockingQueue {

public static void main(String[] args) {

        Scanner in = new Scanner(System.in);
System.out.print("Enter base directory(e.g. C:\\Program Files\\Java\\jdk1.8.0_73\\);");
String directory = in.nextLine();
System.out.println("Enter keyword (e.g volatile);");
String keyword = in.nextLine();
        final  int FILE_QUEUE_SIZE=10;
```

```
        final  int SARCH_THREADS=100;

BlockingQueue<File> queue = new ArrayBlockingQueue<File>(FILE_QUEUE_SIZE);
FileEnumerationTask enumerator = new FileEnumerationTask(queue, new File(directory));
        new Thread(enumerator).start();
        for (int i = 0; i <SARCH_THREADS ; i++) {
new Thread(new SearchTask(queue,keyword)).start();
}
    }
}

class FileEnumerationTask implements Runnable
{
    public static File DUMMY = new File("");
    private BlockingQueue<File> queue;
    private File startingDirectory;

/**
     * Constructs a FileEnumerationTask.
     * @param queue the blocking queue to which the enumerated files are added
     * @param startingDirectory the directory in which to start the enumeration
     */
public FileEnumerationTask(BlockingQueue<File> queue, File startingDirectory)
    {
        this.queue = queue;
        this.startingDirectory = startingDirectory;
}

public void run()
    {
try
{
            enumerate(startingDirectory);
            queue.put(DUMMY);
}
catch (InterruptedException e)
        {

        }
    }

/**
     * Recursively enumerates all files in a given directory and its subdirectories.
     * @param directory the directory in which to start
     */
public void enumerate(File directory) throws InterruptedException
    {
        File[] files = directory.listFiles();
        for (File file : files)
        {
if (file.isDirectory()) enumerate(file);
            else queue.put(file);
}
    }
}

class SearchTask implements Runnable{

private BlockingQueue<File>queue;
    private String keyword;
    public  SearchTask(BlockingQueue<File> queue,String keyword) {
        this.queue=queue;
        this.keyword = keyword;
}


@Override
public void run() {
try {
boolean done = false;
        while (!done) {
```

```
            File file = queue.take();
            if(file ==FileEnumerationTask.DUMMY) {
            queue.put(file);
            done = true;
}
        else search(file);
}
    }
catch (IOException e){
        e.printStackTrace();
}
catch (InterruptedException e)
    {

    }
    }

public void search(File file) throws IOException
    {
try (Scanner in = new Scanner(file))
        {
              int lineNumber = 0;
            while (in.hasNextLine())
            {
                lineNumber++;
                String line = in.nextLine();
                if (line.contains(keyword))
                    System.out.printf("%s:%d:%s%n", file.getPath(), lineNumber, line);
}
        }
    }
}
```

这个具体的作用在书中有所介绍

在这个列子我们将队列数据结构作为一种同步机制

线程安全的集合

高效的散列表 集合和队列
在java.util.concurrent包中提供了散列表 有序集和队列的高效实现 ConcurrentHashMap ,ConcurrentSkipMap ConcurrentSkipListSet和ConcurrentLinkedQuerue
这些集合使用复杂的算法 通过允许并发的访问数据结构的不同部分来使竞争条件极小化
这里与大部分集合不同的是 size方法不必在常量时间内操作 **要确定这样的集合当前的大小需要遍历**
集合返回**弱一致性**的迭代器 **这意味着迭代器不一定能反映出它们被构造之后的所有的修改** 但是 它们不会将同一个值返回俩次 也不会抛出ConcurrentModiicationException
如果**集合在迭代器之后发生改变** java.util包中的迭代器则会抛出一个ConcurrentModiicationException异常
并发的线程列表 可高效地支持大量的读者和一定数量的写者
ConcurrentHashMap和ConcurrentSkipListMap类有相应的方法用于**原子性的关联插入以及关联删除** putifAbsent方法自动地添加新的关联 前提是原来没有这一关联 对于多线程访问的缓存来说这是非常有用的

```
 ConcurrentHashMap cache = new ConcurrentHashMap();
//确保只有一个线程向缓存添加项
cache.putIfAbsent(key, value);
//相反的操作是删除
cache.remove(key, value);
//将原子性地删除键值对 如果它们在映像表出现的话 最后
cache.replace(key, oldvalue, newvalue);
//原子性地用新值替换旧值
```

写数组的拷贝

CopyOnWriteArrayList和CopyWriteArraySet是线程安全的集合 其中所有的修改线程对底层数组进行复制
如果在**集合上进行迭代的线程数超过修改线程数** 这样的安排是很有用的

较早的线程安全集合

较早的线程安全集合比如**vector和hashtable** 目前这俩个已经被弃用了 代替的是Arraylist和HashMap 当然Arraylist和HashMap本身也不是线程安全的
但是集合框架提供了不同的机制 任何的集合都可以通过使用**同步包装器**变为线程安全的

```
List synchArrayList = Collections.synchronizedList(new ArrayList<>());
Map synchHashMap = Collections.synchronizedMap(new HashMap<>());
```

结果集合的方法使用锁加以保护 提供了线程的安全访问
应该**确保没有任何线程通过原始的非同步方法访问数据结构** 最便利的方法是确保不保存任何指向原始对象的引用 简单的构造一个集合并立即传递给包装器 像我们的列中中所做的那样

如果在另一个集合可能进行修改时要对集合进行迭代 仍然需要使用“客户端”编程

```
synchronized (synchHashMap) {
    Iterator iter = synchHashMap.keySet().iterator();
    while (iter.hasNext()) ..........
}
```

如果使用**foreach的话就必须使用同样的代码** 因为集合使用了迭代器 注意 如果在迭代过程中 别的线程修改集合 迭代器会失效 抛出ConcurrentModiicationException异常 同步仍然是需要的 因此并发的修改可以被可靠的检测出来

最好使用的是在java.util.concurrent中定义的集合 不使用同步包装器中的 特别是
假如他们访问的是不同的桶 由于ConcurrentHashMap已经精心实现了 多线程可以访问它而且不会彼此阻塞
但是**有一个例外就是经常被修改的数组列表** 在那种情况下 使用同步器包装的ArrayList比CopyWriteArrayList要好

Callable和Future
http://blog.csdn.net/ghsau/article/details/7451464具体讲解
Runable封装了一个异步运行的任务 可以把它想象为一个没有参数和返回值的异步方法
Callable和Future，它俩很有意思的，一个产生结果，一个拿到结果。
Callable接口类似于Runnable，从名字就可以看出来了，但是Runnable不会返回结果，并且无法抛出返回结果的异常，而Callable功能更强大一些，被线程执行后，可以返回值，这个返回值可以被Future拿到，也就是说，Future可以拿到异步执行任务的返回值，

Callable是一个没有参数和返回值的类型 只有一个方法 call
其中的类型参数是返回值的类型

```
@FunctionalInterface
public interface Callable<V> {
/**
     * Computes a result, or throws an exception if unable to do so.
     *
     * @return computed result
     * @throws Exception if unable to compute a result
     */
V call() throws Exception;
}
```

Future是**保存异步计算的结果** 可以启动一个计算 将Future对象交给某个线程 然后忘掉它 Future对象的所有者在结果计算好之后就可以获得它

```
public interface Future<V> {

boolean cancel(boolean mayInterruptIfRunning);

boolean isCancelled();

boolean isDone();
    V get() throws InterruptedException, ExecutionException;

V get(long timeout, TimeUnit unit)
throws InterruptedException, ExecutionException, TimeoutException;
}
```

get方法的调用被阻塞 直到计算完成 如果在计算完成之前 第二个get方法的调用超时 抛出一个TimeoutException异常 如果运行该计算的线程被中断 俩个方法都将抛出InterruptedException 如果计算已经完成 那么get方法立即返回

如果计算还在运行 isDone方法返回false 如果完成了 返回true
可以用cancel方法取消该运算 如果计算还没开始 它被取消且不再开始 如果计算处于运行之中 那么如果mayinteruptifRunning参数为true 它就被中断

FutureTask包装器是一种很有用的包装器 **可以将Callable转换为Future和Runable** 它同时实现了俩者的接口

FutureTask实现了两个接口，Runnable和Future，所以它既可以作为Runnable被线程执行，又可以作为Future得到Callable的返回值，那

么这个组合的使用有什么好处呢？假设有一个很耗时的返回值需要计算，并且这个返回值不是立刻需要的话，那么就可以使用这个组合，用另一个线程去计算返回值，而当前线程在使用这个返回值之前可以做其它的操作，等到需要这个返回值时，再通过Future得到，岂不美哉！这里有一个Future模式的介绍：http://openhome.cc/Gossip/DesignPattern/FuturePattern.htm。

下面就是一个关于这Callable和Future 的例子 代码当初刚粘贴过来的时候 是完好的 但是后来自然的就变成了这样 只能作为参考 看吧

FutureTest.java
2016/11/05 10:35, 2.67KB

执行器
构造一个新的线程是有代价的 因为涉及与操作系统的交互 如果程序中创建了大量的生命周期很短的线程 应该使用**线程池** 一个线程池中包含许多准备运行的空闲线程 将Runable对象交给线程池 就会有一个线程调用run方法 当run方法退出时 线程不会死亡 而是在池中准备下一个请求提供服务
另一个使用**线程池的理由时减少并发的数目** 创建大量的线程会大大降低性能甚至使虚拟机崩溃 如果有一个会创建许多线程的算法 应该使用一个线程数 固定的线程池以限制并发线程的总数

执行器类有许多静态工厂方法用来构造线程池
http://ifeve.com/java7-concurrency-cookbook-3/这是执行器的详细讲解

newCachedThreadPool	必要时创建新线程 空闲线程会被保持60秒
newFixedThreadPool	该池包含固定数量的线程 空闲线程会一直被保留
newSingleThreadExcutor	只有一个线程的池 该程序顺序执行每一个提交的任务
newScheduledThreadPool	用于预定执行而构建的固定线程池 替代java.util.Timer
newSingleThreadScheduledExecutor	用于执行而构建的单线程池

线程池
newChchedThreadPool方法构建了一个线程池 对于每个任务 如果有空闲线程可用 立即让它执行任务 如果没有可用的空闲线程 则创建一个新线程
newFixedThreadPool方法构建了具有固定大小的线程池 如果提交的任务数多于空闲的线程数 那么把得不到服务的任务放置到队列中 当其他任务完成以后再运行它们 **newSingleThreadExcutor**是一个退化了的大小为1的线程池 由一个线程执行提交的任务 一个接着一个 这3个方法返回实现了**ExecitorService**接口的**ThreadPoolExecutor**类的对象

可用下面的方法之一 将一个Runnable对象或Callable对象提交给ExecutorService
Future<?> submit(Runnable task)
Future<T> submit(Runnable task,T result)
Future<T> submit(Callable<T> task)

该池会在方便的时候尽早执行提交的任务 调用submit 会得到一个Future对象 可用来查询该任务的状态

第一个submit方法返回一个奇怪样子的Future<?> 可用使用这样一个对象来调用isDone cancel或isCancelld 但是 get方法在完成的时候只是简单的返回null
第二个submit方法也提交一个Runnable 并且Future的get方法在完成的时候返回指定的result对象
第三个submit方法提交一个Callable 并且返回Future对象将在计算结果准备好的时候得到它

当用完一个线程池的时候 调用shutdown 该方法启动该池的关闭序列 被关闭的执行器不再接受新的任务 当所有任务都完成后 线程池的线程死亡 另一种方法是调用shudownNow 该池取消尚未开始的所有任务并试图中断正在运行的线程

下面总结了在连接池应该做的事
1 调用Exectors类中静态的方法newCachedThreadPool或newFixedThreadPool
2 调用submit提交Runnable或Callable 对象
3 如果想要取消一个任务 或如果提交Callable对象 那就要保存好返回的Future对象
4 当不再提交任何任务时 调用shutdown
下面是一个代码示例

FutureTest.java
2016/11/05 10:35, 2.50KB

预定执行

ScheduledExecutorService接口具有预定执行或重复执行任务而设计的方法 它是一种允许使用线程池机制的java.util.Timer的泛化 Executors类的newScheduledThreadPool和newSingleThreadScheduledExecutor方法将返回实现了ScheduledExecutorService接口的对象

可以预定Runable或Callable在初始的延迟之后只运行一次
也可以指定一个Runable对象周期性的运行
预定执行的Demo 代码粘贴的很丑 直接拿来运行吧

```
importjava.util.concurrent.Executors;
importjava.util.concurrent.ScheduledExecutorService;
importjava.util.concurrent.ScheduledFuture;

importstaticjava.util.concurrent.TimeUnit.*;
publicclassBeeperControl {



privatefinalScheduledExecutorService scheduler =
Executors.newScheduledThreadPool(1);

publicvoidbeepForAnHour() {
//执行内容
finalRunnable beeper = newRunnable() {
publicvoidrun() { System.out.println("beep"); }
        };
//具体执行设置 为10秒延迟和连续成功间隔时间10秒
finalScheduledFuture<?> beeperHandle =
scheduler.scheduleAtFixedRate(beeper, 10, 10, SECONDS);
//1小时关闭这个线程
scheduler.schedule(newRunnable() {
publicvoidrun() {
beeperHandle.cancel(true);
            }
        }, 60 * 60, SECONDS);
    }

publicstaticvoidmain(String[] args) {
try{
//反射
BeeperControl beeperControl = BeeperControl.class.newInstance();
beeperControl.beepForAnHour();
         } catch(InstantiationException e) {
e.printStackTrace();
         } catch(IllegalAccessException e) {
e.printStackTrace();
         }
     }
}
```

控制任务组

我们已经知道如何将一个执行器服务作为线程池使用，以提高执行任务的效率。有时候，我们要使用执行器来做更有实际意义的事，控制一组相关的任务。例如，**可以在执行器中使用shutdownNow方法取消所有的任务。**

invokeAll方法提交所有对象到一个Callable对象的集合中，并返回某个已经完成了任务的结果 无法知道返回的究竟是哪个任务的结果 也许是最先完成的那个任务的结果　如果你愿意接受任何一种解决方案的话 就可以使用它

invokeAll方法提交所有对象到一个Callable对象的集合中，并返回一个Future对象的列表，**代表所有任务的解决方案** 当计算结果可获得时 可以像下面这样对结果进行处理

```
List<Callable<T>> tasks = .......;
List<Future<?>> results = pool.invokeAll(tasks);
for (Future<T> result:results) {
    processFuture(result.get());
}
```

这个方法的缺点是如果第一个任务花去了很多时间，那么就可能不得不进行等待。将结果按可获得的顺序保存起来更有意义。可以用ExecutorComeletionService来进行排序。

具体可以查询API。用常规的方法获得一个执行器。然后，构建一个ExecutorComeletionService，提交任务给完成服务。该服务管理Future对象的阻塞队列，其中包含已经提交的任务的执行结果。

```
ExecutorService executor = Executors.newCachedThreadPool();
ExecutorCompletionService service = new ExecutorCompletionService<>(executor);
for (Callable<T> task : tasks) {
```

```
  service.submit(task);
}
for (int i = 0; i < task.size(); i++) {
  processFurther(service.take().get());
}
```

Fork-Join框架

fork/join框架是ExecutorService接口的一个实现，可以帮助开发人员充分利用多核处理器的优势，编写出并行执行的程序，提高应用程序的性能；设计的目的是为了处理那些可以被递归拆分的任务。

fork/join框架与其它ExecutorService的实现类相似，会给线程池中的线程分发任务，不同之处在于它使用了工作窃取算法，所谓工作窃取，指的是对那些处理完自身任务的线程，会从其它线程窃取任务执行。

fork/join框架的核心是ForkJoinPool类，该类继承了AbstractExecutorService类。ForkJoinPool实现了工作窃取算法并且能够执行 ForkJoinTask任务。

http://www.infoq.com/cn/articles/fork-join-introductionFork Join框架详解

有的应用程序使用了大量的线程，但其中大多数都是空闲的。举例来说，一个Web服务器可能会为每个连接分别使用一个线程。另外一些应用可能对每个处理器内核分别使用一个线程，来完成计算密集的任务，如图像或视频处理。Java SE 7 中新引入了fork-join框架，专门用来支持后一类的应用。

我们来讨论一个简单的例子。假设我们想统计一个数组中有多少个元素满足某个特定的属性。可以将这个数组一分为二，分别对着两部分进行统计，再将结果相加。

```
package com.javaMultitThread.forkjoin;

import java.util.concurrent.*;

/**
 * This program demonstrates the fork-join framework.
 * @version 1.00 2012-05-20
 * @author Cay Horstmann
 */
public class ForkJoinTest
{
public static void main(String[] args)
   {
final int SIZE = 10000000;
      double[] numbers = new double[SIZE];
      for (int i = 0; i < SIZE; i++) numbers[i] = Math.random();
Counter counter = new Counter(numbers, 0, numbers.length,
         new Filter()
         {
public boolean accept(double x) { return x > 0.5; }
         });
ForkJoinPool pool = new ForkJoinPool();
pool.invoke(counter);
System.out.println(counter.join());
}
}

interface Filter
{
boolean accept(double t);
}

class Counter extends RecursiveTask<Integer>
{
public static final int THRESHOLD = 1000;
   private double[] values;
   private int from;
   private int to;
   private Filter filter;
```

```
   public Counter(double[] values, int from, int to, Filter filter)
   {
this.values = values;
      this.from = from;
      this.to = to;
      this.filter = filter;
}

protected Integer compute()
   {
if (to - from < THRESHOLD)
      {
int count = 0;
         for (int i = from; i < to; i++)
         {
if (filter.accept(values[i])) count++;
}
return count;
}
else
{
int mid = (from + to) / 2;
Counter first = new Counter(values, from, mid, filter);
Counter second = new Counter(values, mid, to, filter);
invokeAll(first, second);
         return first.join() + second.join();
}
   }
}
```

在这里invokeAll方法接受到很多任务并阻塞 直到所有这些任务都已经完成　join方法将生成结果 我们对每个子任务应用了join 并返回总和

在后台 fork-join框架使用了一种有效的智能方法来平衡可用线程的工作负载 这种方法被称为**工作密取** 每个线程有一个双端队列来完成任务 一个工作线程将子任务压入其双端队列的队头 一个工作线程空闲时 它会从另一个双端队列的队尾密取

同步器

java.util.concurrent 包包含了几个能帮助人们管理相互合作的线程集的类见表 这些机制具有为线程之间的共用集合点模式 提供的预置功能
如果一个相互合作的线程集能满足这些行为模式之一 那么应该直接重用合适的库类而不要试图提供手工的锁与条件的集合

类	它能做什么	何时使用
CyclicBarrier	允许线程集等待直至其中预定数目的线程到达一个公共障栅 然后可以选择执行一个处理障栅的动作	当大量的线程需要在它们的结果可用之前完成时
CountDownLatch	允许线程集等待直到计算器减为0	当一个或多个线程需要等待直到指定数目的事件发生
Exchanger	允许俩个线程在要交换的对象准备好时交换对象	当俩个线程工作在同一数据结构的俩个实例上的时候 ，一个向实例添加数据 ，而另一个从实例清除数据。
Semaphre	允许线程集等待直到被允许继续运行为止	限制访问资源的线程总数。如果许可数是1 常常阻塞线程直到另一个线程给许可为止
SynchronousQueue	允许一个线程把对象交给另外一个线程	在没有显示同步的情况下 ，当俩个线程准备好将一个对象从一个线程传递到另一时

信号量
关于信号量 概念上讲 一个信号量是管理许多的许可证 为了通过信号量 线程通过调用acquire请求许可 其实没有实际的许可对象 信号量

仅维护一个计数 许可的数目是固定的 由此限制了通过的线程数量
线程是作为同步原语的 在许多操作系统的书中基本都有出现 由信号量实现了有界队列 通常 信号量不比直接映射到通用应用场景

倒计时门栓
一个倒计时门栓上一个线程集等待直到计数变为0 倒计时门栓是一次性 一旦计数为0 就不能再重用了
一个有用的特例是计数值为1 的门栓 实现一个只能通过一次的门

障栅

CyclicBarrier类实现了一个集结点(rendezvous)称为**屏障/障栅(barrier)**。考虑大量线程运行在一次计算的不同部分的情况，当所有部分都准备好时，需要把结果组合在一起。当一个线程完成了它那部分任务之后，我们让它运行到屏障/障栅处。一旦所有的线程都到达了这个屏障/障栅，屏障/障栅就撤销，线程就可以继续运行。
(1)构造一个屏障/障栅，并给出参与的线程数
CyclicBarrier barrier = new CyclicBarrier(nThreads);
(2)每一个线程做一些工作，完成后在屏障/障栅上调用await
public void run(){
doWork();
barrier.await();
...
}
(3)如果**任何一个在屏障/障栅上等待的线程离开了屏障/障栅，那么屏障/障栅就破坏了**（线程离开的原因：线程调用的await超时或因为它被中断了），在这种情况下所有其他线程的await方法抛出BrokenBarrierException异常。那些已经在等待的线程立即终止await的调用。
(4)可以提供一个**可选的屏障/障栅动作(barrier action)**，当所有线程到达屏障/障栅的时候就会执行这一动作，该动作可以**收集那些单个线程的运行结果。**
Runnable barrierAction = ...;
CyclicBarrier barrier = new CyclicBarrier(nThreads, barrierAction);

注意：屏障/障栅被称为是**循环的(cyclic)**，因为可以在所有等待线程被释放后重用，而CountDownLatch只能被使用一次。
下面是网上找的Demo

```
import java.util.Random;
import java.util.concurrent.CyclicBarrier;
import static java.lang.System.*;

public class CyclicBarrierTest
{
public static void main(String[] args)
    {
final int[] arr = new int[3];
//可以提供一个可选的 障栅 动作（Barrier Action），  当所有线程到达障栅的时候就会执行这个工作
//CyclicBarrier cyclicBarrier = new CyclicBarrier(nthreds, barrienAction)
CyclicBarrier cyclicBarrier = new CyclicBarrier(2, new Runnable()
        {
@Override
public void run()
            {
arr[2] = arr[0] + arr[1];
for (int i = 0; i < arr.length; i++)
out.println("arr[" + i +"] = " + arr[i]);
            }
        });

new Thread(new MyCyclicBarrier(1, arr, cyclicBarrier)).start();
new Thread(new MyCyclicBarrier(0, arr, cyclicBarrier)).start();
    }
}

class MyCyclicBarrier implements Runnable
{
int id;
int[] arr;
CyclicBarrier cyclicBarrier;

public MyCyclicBarrier(int id, int[] arr, CyclicBarrier barrier)
    {
this.id = id;
this.arr = arr;
this.cyclicBarrier = barrier;
    }

@Override
public void run()
    {
arr[id] = new Random().nextInt(100);
try
```

```
{
out.println(id + " starts to have a rest!");
cyclicBarrier.await();
out.println(id + " wake up!");
        } catch (Exception e)
        {
e.printStackTrace();
        }
    }

}
```

交换器
当俩个线程在同一个数据缓冲区的俩个实例上工作的时候 就可以使用交换器(Exchanger)了
典型的情况是，一个线程向一个缓冲区填入数据，另一个线程消耗这些数据，当它们都完成后，相互交换缓冲区。
同步队列
同步队列是一种将生产者与消费者线程配对的机制 当一个线程调用SynchronousQueue的put方法时 它会阻塞直到另一个线程调用take方法为止 反之亦然 与Exchanger的情况不同 数据仅仅沿着一个方向传递 从生产者到消费者
注意 即使SynchronousQueue类实现了BockingQueue接口 概念上讲 它依然不是一个队列 它没有包含任何元素