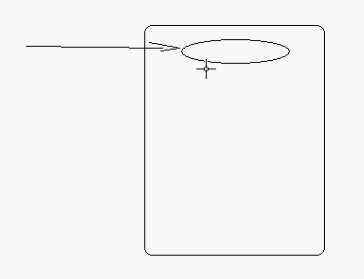
==== 线程池===

Java5中的一个非常显著的功能就是线程池

要做一个Tomcat是非常难的

因为性能升不上去

现在假设写了一个Tomcat程序



一个客户端连接到一个Tomcat服务器上之后

Tomcat服务器就会启动一个线程 这个线程来和服务器进行对话

为什么要启动线程呢？

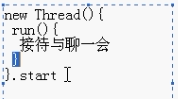
因为 如果不启动线程的话 如果这个链接和服务器端唠唠叨叨的对话

别的客户无法进行连接 --- 别的客户堵了

此时 应该启动多线程 ---- 每一个线程都去接待一个客户

线程服务完之后 就完了

写的程序如下：



run方法一结束 这个线程就结束了

现在 有60名同学

这样 在后面就有60名服务人员 对应着60名同学 服务

每一个工作人员都是服务一次 服务完成就没用了

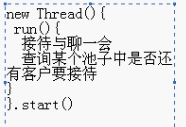
这样浪费资源

现在想法就是

有几个固定的接待人员

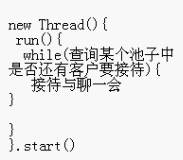
服务数量不可知的客户

代码如下：



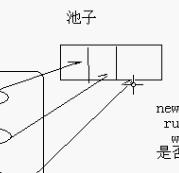
如果有--取出来--聊--聊完--就--发现没有--就等待

---- 这样 就比较复杂



所以 这时候 应该有一个池子

当主程序接到一个客户之后 就加到池子中



现在 多个线程到池子中去获取

现在很多个线程都去池子里面查 并处理

取到一个 就把对应的池子搞空

别的客户又加进来 池子就相当于一个队列

---- 当队列空的时候 这些线程都在等待

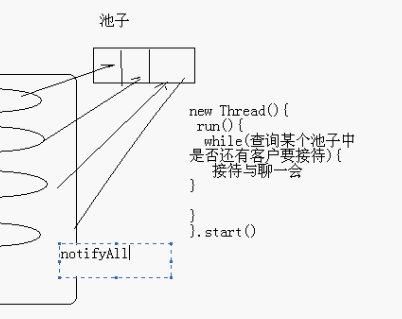
当我再接到一个客户的时候

加到池子中

但是 线程都在等着睡觉 --- 此时 有的话

此时 就要有线程来服务 --- 唤醒谁 不一定

要唤醒所有的线程 ---- 谁抢到了 就给谁 没抢到的 继续睡觉



现在 Java5帮我们提供了线程池



直接使用线程池 就可以解决这个问题

现在演示一下这个线程池



【**多个线程 看到有客户 就去执行 没拿到 就进行等待**】

**不用记住什么方法**

**就记住这个工具类【注意 这个类是s结尾的】**

这个Executors上面有一大堆静态方法类



可以看到 有很多new出来的线程池



方法的返回值是ExecutorService --- 这个类就代表一个线程池



ExecutorService---执行器服务 就是线程池

现在进行操作



现在看到 每一个任务是一种Runnable

【这里面就看到了 线程 和 任务 是分开的类

更符合面向对象的思想 ---- 职责最小化

实际上 里面的代码就是

New Thread(new Runnable(){

@Override

public void run(){

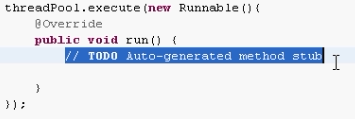
}

}).start();

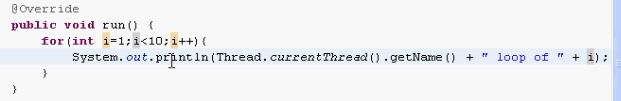
黄色背景就是传入的Runnable

任务

】



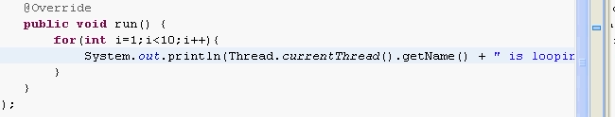
这里面 打印线程的名字 --- 并且每个任务循环10遍



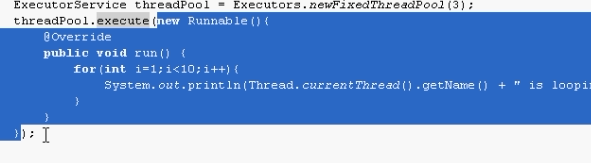
现在 还是理解 **池子 --- 数据队列 和线程池【数据队列 + 线程池】**

扔进来一个任务【也就是前面的客户】

前面线程池中的3个线程 就跑来执行

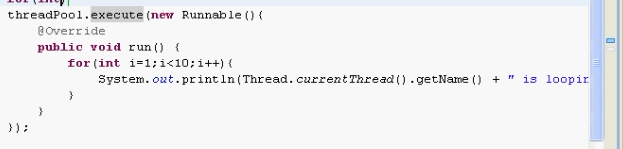


ExecutorService-🡪threadPool是一个线程池



**看成：向线程池threadPool扔入了一个任务是new Run(){}**

蓝色背景的是被扔入池中的一个任务

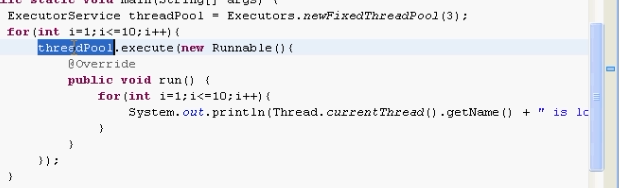


这句代码 就翻译成：

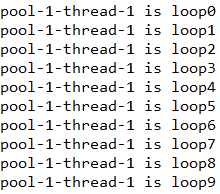
**向线程池threadPool扔入了一个任务是new Run(){}**

---- 这只是一个任务

现在改成



【运行结果是：

 ---- 可以看到 虽然我的线程池的大小是3 但是我仅仅使用了其中的一个线程 --- pool-1-thread-1这个线程 去执行唯一的一个任务 ---- 这个任务就是打印10次 其余的两个线程是空闲的】

现在 还是一个池子 每次循环操作的都是这个池子

每一次外围循环都是往这个池子中扔入一个任务

**【也就是想让这个线程池的多个线程去运行多个不同的任务】---- 所以外围是一个循环**

**但是请注意到 每次外围循环都是使用的同一个线程池的引用threadPool 没有因为外围循环的改变导致线程池的引用的变化**

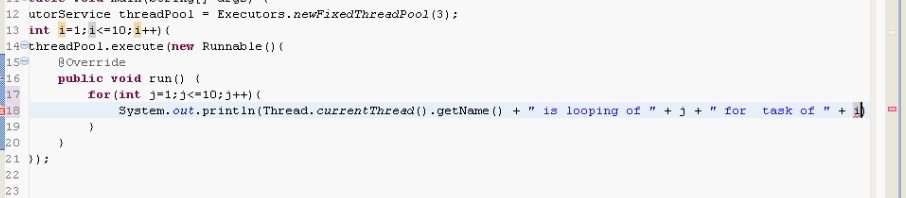
这样 一共扔入10个任务

【现在就是一共10个任务 --- 每一个任务的业务逻辑是循环打印输出10遍】

现在 内部任务打印的过程中

要访问这个外部循环变量i

内部修改为J



必须是final的变量

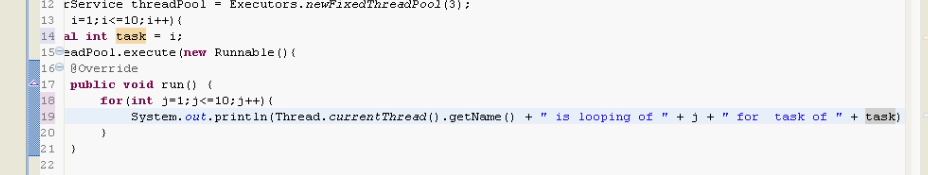


又错了 --- final的变量不能修改

--- 所以 我们保证i可以修改

然后 把这个修改之后的i复制给一个临时的final变量





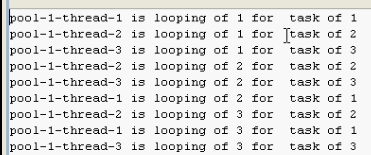
---- 这就搞定了

这样运行一遍

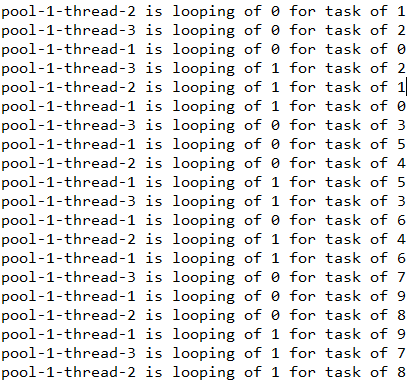
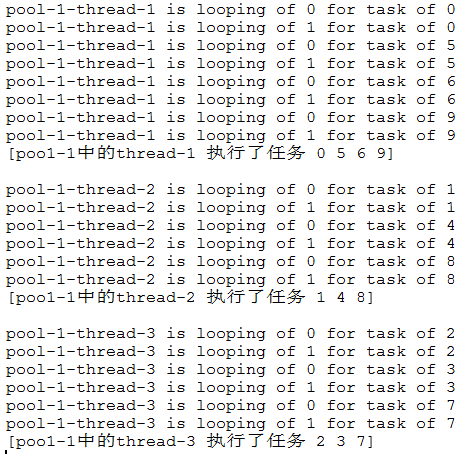
这个太快了

慢一点





【我们把每一次任务的循环次数降低为2次执行 VS 整理后的结果

 】

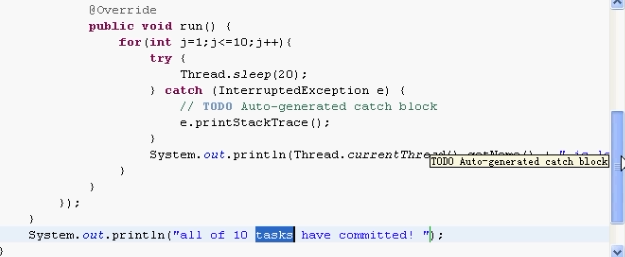
这样 大家始终看到了 1 2 3[前面的]线程在执行任务

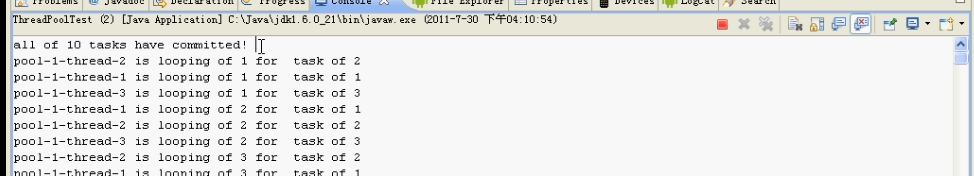
因为只有三个线程 所以 只能同时为三个任务服务

剩下的7个任务还在排队等待

加一句代码 表示10个任务都提交了

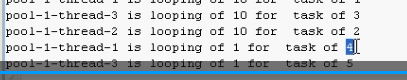
就在主线程中





10个任务都被提交了 但是 只有3个任务被处理

**这3个任务完成之后 三个线程就继续为后面的任务服务**



【做服务器 就用这个线程池】

但是 执行完之后 看到

 --- 这个地方一直是红的

**这是因为 池子中只有三个线程 是不会死掉的**

**线程不死 程序就不结束【线程池的程序执行完 还是红的按钮的原因】**

如果 想让所有的任务被执行完 程序就结束

做法是：【也就是池子里面没有任何任务了】

就把所有的线程都干掉





shutdown, shutdownNow的真实含义

http://blog.csdn.net/jkunzhang/article/details/6109217

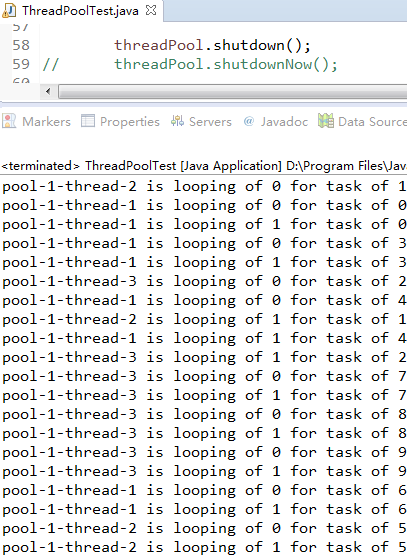
【 --- 一个人的说法

主线程和线程池之间没有直接关系，线程池使用自己的线程。生命周期也相互独立。  
  
shutdown()可以理解为：主线程要求线程池关闭，但不会为此等待线程池执行完毕。你这里面实际发挥了等待作用的并不是线程池所提供的能力（当然线程池也确实提供了这类能力），而是：position.acquireUninterruptibly(2) 这句话。  
  
  
shutdown() 作为函数，当然是立即执行，也即是不再接受新任务了；但是它即不会强行终止正在执行的任务，也不会取消已经提交的任务。也就是说之前提交的5个任务，仍然会执行完毕，且跟主线程生命周期无关，也就是即便你直接在后面写上： if (1==1) return; 来立即结束主函数，你也会发现线程池的5个任务会顺利执行完毕。  
  
  
另一个长得很像的函数是：  
shutdownNow()，这个函数比shutdown()更狠，两点：  
1、对于尚未执行的任务，全部取消掉；  
2、对于正在执行的任务，发出interrupt()。  
不过你的程序因为在发生异常时没有正确释放信号量，所以如果改为shutdownNow()会出问题：主线程死等。

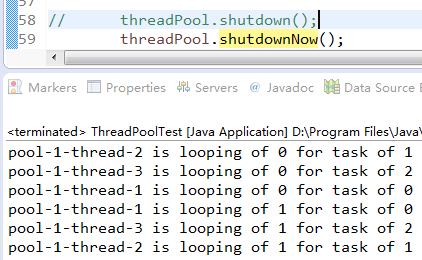
我的测试

现在 我使用threadPool.shutdown();

执行结果



修改成 shutdownNow()



这样 很明显 所有的任务没有执行完 一旦主线程结束了 对应的ThreadPool中的内容也跟着结束了】

【个人理解：

注意都写在外层for循环的两条语句



这两句是主线程发出来的 都马上被执行(如果主线程) ---- 但是 结果看到了

第一句最开始就打出来了(主线程一下子就执行完)

但是 第二句 真的是在所有线程都结束了 才执行

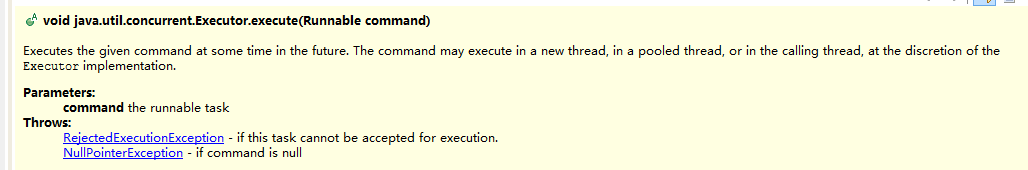
---- 翻一下JDK文档：



execute方法 不是直接定义在ExecutorService这个接口中

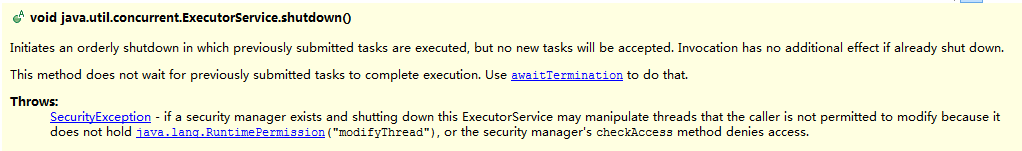
而是定义在Executor的父接口中

{1}. execute中的Runnable被执行的时机：



会在**将来的某个时刻**执行给定的命令/任务 ----所以 你这里面 execute顶多算是告诉线程池 这个任务我要执行 ---- **不是马上执行哦！**

{2}. shutdown()是ExecutorService中的方法



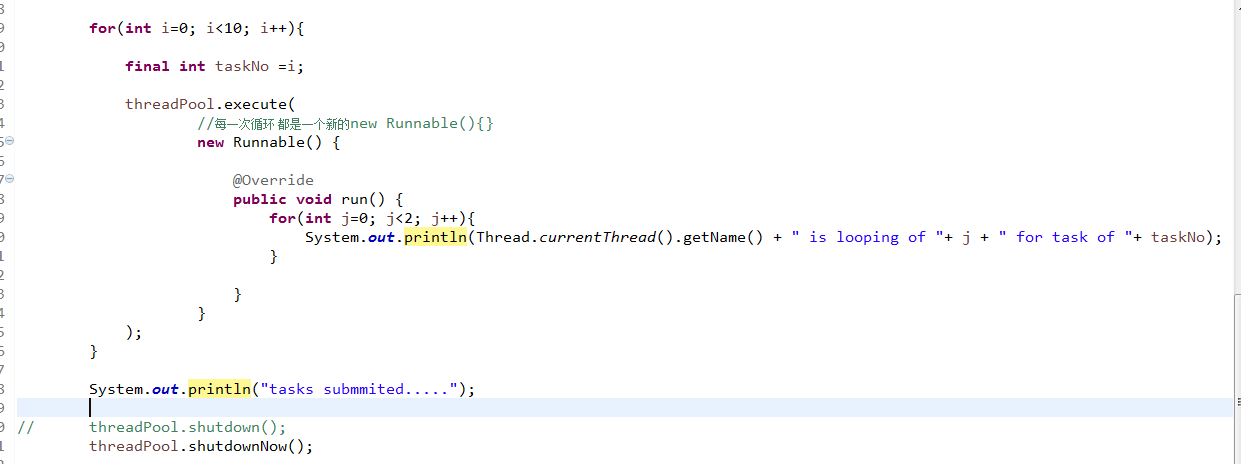
Shutdown()方法没有返回值

这个方法的源文档解释看清楚了 是“**初始化**”一个有顺序的关闭动作[可不是马上就关闭哦] ---- 在这个有顺序的关闭动作里面**之前提交的**任务**被执行**但是**不会接受新的任务**。

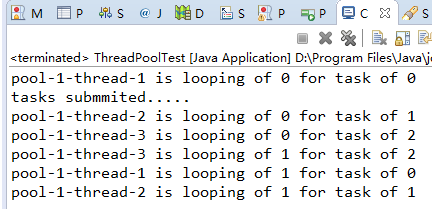
注意 先前提交的任务会被执行 ---- 但是 下面马上就说：这个方法不能够等待先前提交的任务执行完成！！！---- **提交的任务会被执行 但是 不一定能执行完哦！！！！**---- 直接使用 我觉得 很冒险 ---- 看上面的帖子 再做决定

----- 1-27新的理解：

execute ----- 不是立即执行 ---- 会在未来的某个时刻执行指定的任务 ----- 这个任务也许是在新的线程或者池化的线程或者调用线程中进行 ----- 这个是由Executor的实现来决定的 ----- Executor discretion--- 决定 判断 ----- 所以 这个循环执行完成 可能某些任务已经被运行了 还有一些没有运行 但是 可以肯定的是：所有的task都被提交到ExecutorService中了



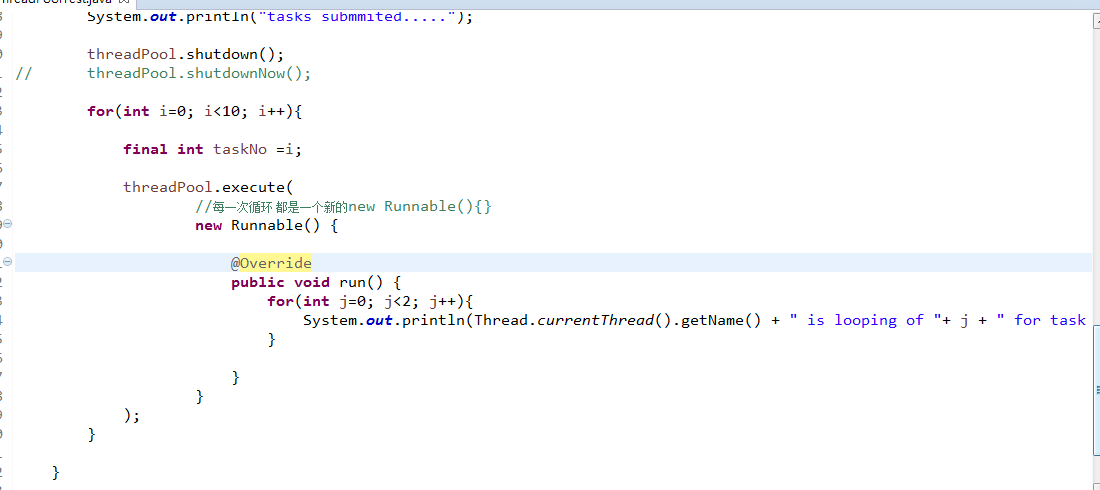
我们看一下执行的结果



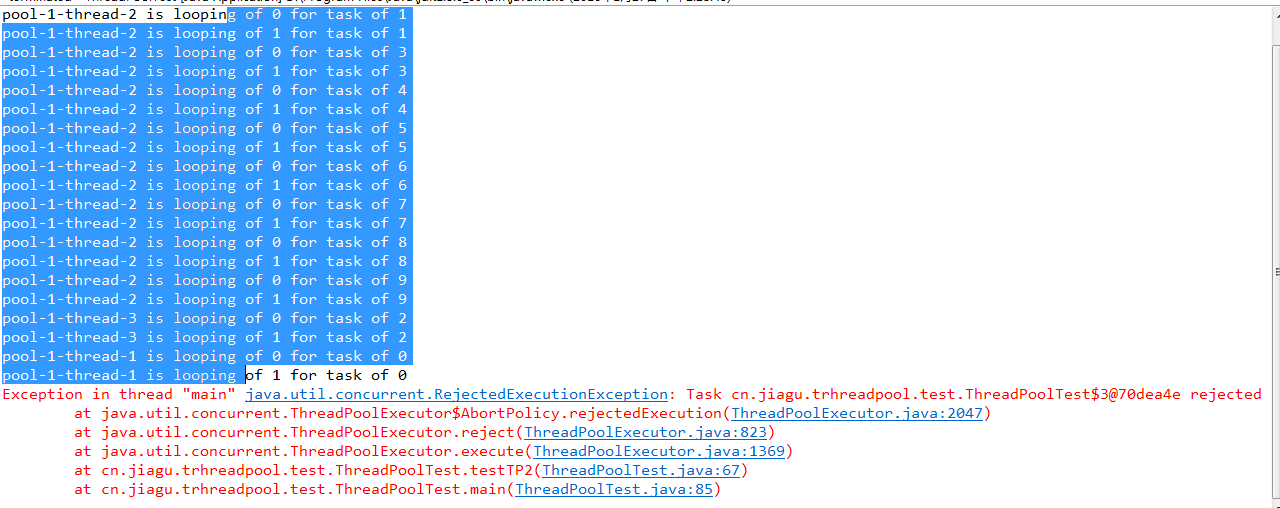
可以看到 某些task都已经被执行的过程中 这个主线程的打印才结束

=🡺 execute执行完成之后 任务都会被提交

这个时候 shutdown仅仅是告诉你的线程池中的线程执行完任务的时候 要释放掉在这个线程池中的资源 所以 execute执行完成 至少把所有的任务提交之后 这个shutdown就会在执行完成之后 释放掉这个线程池的资源 这时候 按照文档上面说的 shutdown会允许已经提交的任务 和 正咋运行的任务执行完成 但是不会接受新提交的任务 所以 我们这个时候 是可以保证execute中的所有的任务都执行完成的



现在我在shutdown()之后 又一次使用了前面的threadpool企图来提交新任务 发现

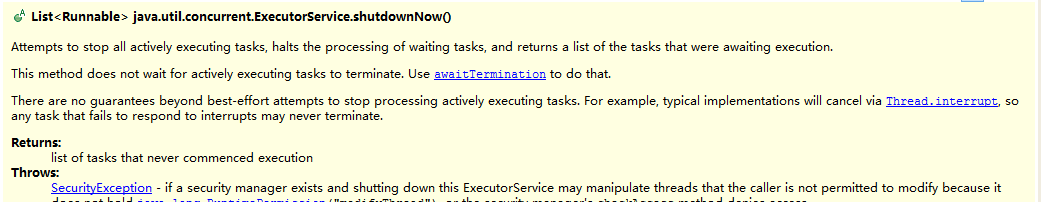


前面shutdown之前通过execute提交的10个任务是都完成了 但是 后面的10个任务被reject 并抛出了异常】

---- 张老师马上举了新的例子

将threadPool.shutdown()修改为threadPool.shutdownNow();

【看一下shutdownNow的文档：



看到 这个方法有返回值 是List<Runnable>

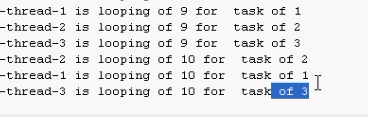
注意 这个方法有三个动作：

**试图停止**所有**活动的*正在执行***的任务，**停止**正在***等待任务***的处理并且**返回**（注意时态 前面是一般现在是 最后一个是过去进行时）***刚才正在等待执行***的任务的列表。

这个方法不会等待正在被执行的任务执行完成 ---- 使用awaitTermination()方法来完成

】

现在看一下张老师这个例子执行的效果



才做了三个任务 其他提交的任务都废掉了

【我们不用把任务亲自交给线程去做 而是 把任务丢到线程池中 由线程池决定执行相关任务的时机】

【**下面是缓存线程池 从newCachedThreadPool()->newFixedThreadPool 这里面使用了Effective Java中的使用静态工厂方法来构造对象**】

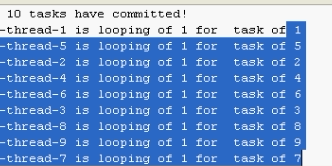
【Executors通过静态方法生成类型是ExecutorService类型的ThreadPool】

------ 刚才是生成固定线程池

下面是生成缓存线程池



当任务多了 线程不够的时候 会自动增加线程



看到左边 是10个线程并发

---- 池子中的线程数 是动态变化的

----- 池子中只有一个线程

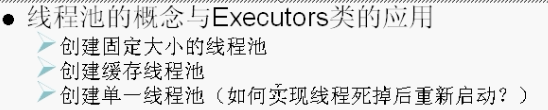


**虽然是单线程 好处是：如果池子中的线程死掉了**

**他也会找一个线程的替补保证一个线程可用 ----- newSingleThreadExecutor()的妙用**

**---- 所以 如何实现线程死掉之后 重新启动？ 就是用这个线程池**

【我的想法是 是不是这个线程被替代了？还是 他自己死了 还复活了？】



还有一种的叫**定时器的线程池**

前面谈到了定时器

-----

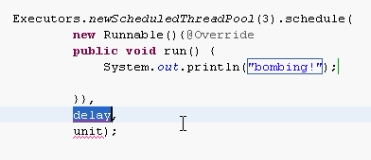


然后 schedule任务

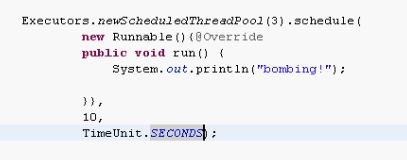
可以多次调用schedule 就表示给了多个定时任务

现在只有3个线程处理

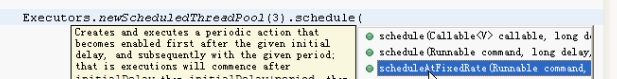




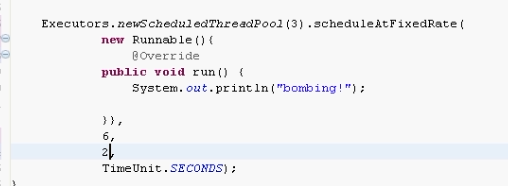
这里面提交了一个任务 --- 这样3个线程 谁抢到了 就只执行



10s钟之后 运行

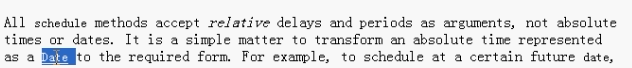


固定频率炸





这个定时器在文档中有一个不好的地方：

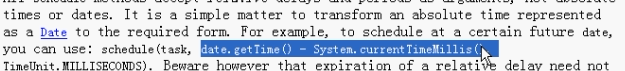


可以简单的指定一个相对的时间

但是 比如 一种 cron表达是那种 定在每晚3点运行

不能做到

文档给了一种解决办法：



【newFixThreadPool()

newCachedThreadPool()

newScheduledThreadPool()】

**===================有 执行结果 + 线程池(Callable + Future) =================**

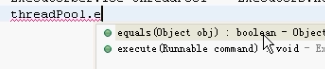
有一个线程 线程运行完了之后 会有一个返回结果

现在获取这个返回结果



先初始化一个线程池



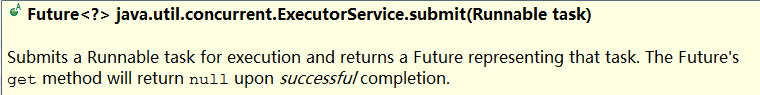


execute没有返回值 实际上 **execute也是任务提交 并发出了执行命令的请求]**

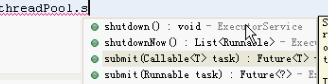
现在使用submit方法进行任务的提交

**[submit 和 execute都会提交任务 但是execute没有返回值 void**

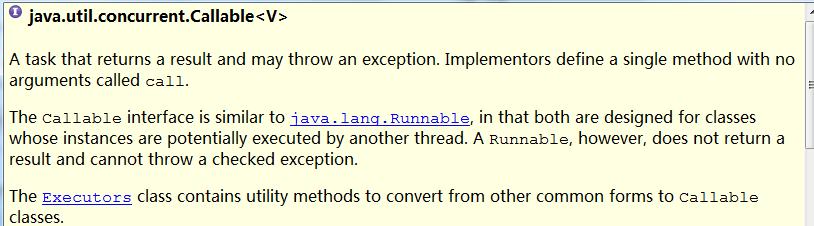
**Submit**



**所以 可以看到 这个submit方法就会提交一个用来执行的Runnable Task 并且返回一个代表那个task的Future对象 ---- 当这个task被成功执行的时候，这个Future的get方法就会返回null ]**



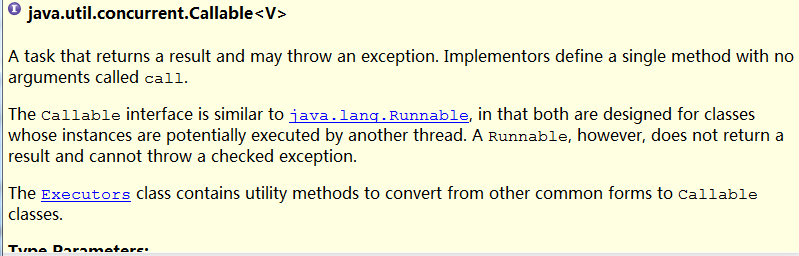
提交Runnable也可以

提交Callable也可以

我们使用后者

[

看一下Callable的文档



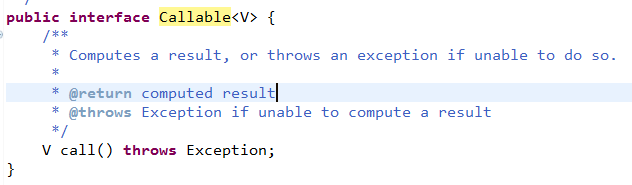
Callable: 一个能返回结果或者抛出异常的task ----- 这个接口的实现者需要实现一个无参数的call方法

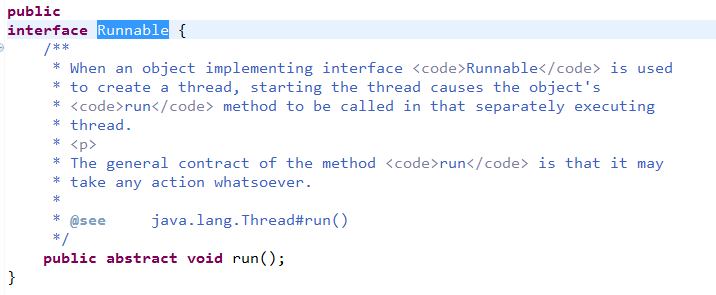
Callable VS Runnable:

Callable接口和java.lang.Runnable接口非常类似 ---- 这两个接口都是为那些潜在可能被另一个线程执行的子类实例而设计的

但是 一个Runnable没有返回值 并且不能抛出异常！！！

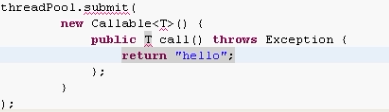
 <V>类型变量V就是call方法的返回值类型





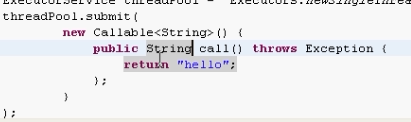
可以看到 在对应的源代码中Runnable#run方法不能抛出异常 但是 Callable#call方法是可以抛出异常 并且返回值就是<V>类型]

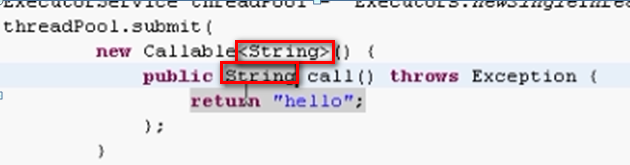
Callable是泛型类



这里面返回的是”hello” ---- String类型的 所以

Callable这个T泛型的类型就是String



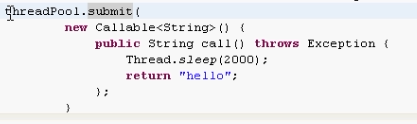
 ----外部的类型变量V就是内部call方法的返回值类型（**个人理解 设计思想 注意 这种一个接口中 有一个方法 有一个返回值 或者 有一个参数 或者 可以可数过来的参数类型 我们就可以把这个唯一的某一种类型向上定义成类的类型变量**）（

比如 public interface Callable<X, Y>{

public Y call(X x) throws Exception;

}

这种情况 就是可以返回多个 ==🡺 所以 这种类型参数的定义 实际上就是为了确定类中的方法的某一种提前设置的方式 --- -这样在类中设置了 实际上就可以变化成多种类型的方法）

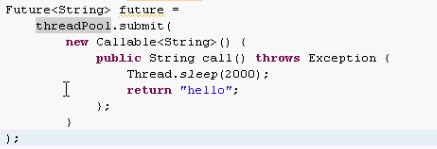


这个Submit一提交 就会有一个结果

这个结果的类型叫Future



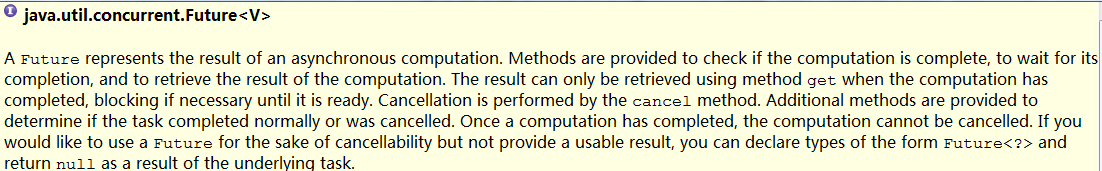
**这个泛型V的类型跟Callable的类型是一致的**



【**个人觉得 返回结果叫Future<V>的原因就是 execute或者submit这两个方法都是做最基本的任务提交 并且 发出执行task的命令 但是 并不能确定立刻执行 文档上面都说是未来的某一个时刻开始执行 ------ 所以 这里面更准确的应该是FutureResult<V> ---- 这里面设计者就简写成Future<V>**】

[

看一下这个Future<V>的文档



Future是能够表示一个异步计算的结果 ----- 这个接口提供了方法：

{1}. 检查异步计算是否完成

{2}. 等待计算完成

{3}. 获取计算的结果

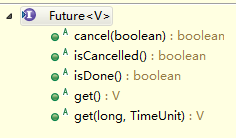
get ----- 当计算的结果已经完成的时候，只能使用get方法来获取计算的结果

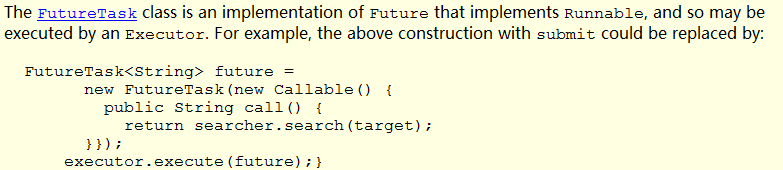
可以使用cancel方法来取消任务的执行

其他的方法都是用来判断这个任务是正常的执行完毕还是被取消了

一旦计算结束了，那么这个计算就不能够被取消了

--- 如果你单单是为了取消但是不想提供一个可用的结果，可以使用Future<?>并且在call中的返回null

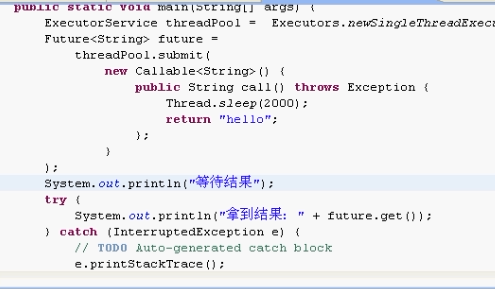


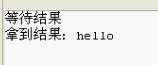


其中一个FutureTask类就是一个实现了Future和Runnable的类（Future代表异步计算执行的结果 Runnable代表任务 这样 这个即代表任务 有代表结果 所以 返回值就是结果）]

提交了一个命令

打印返回结果





这个Future这个结果有什么用呢？【这个我们的机组酒店用到的Spring的线程 一样

在机组酒店中 我们就使用一个boolean标志位来表示所有的线程的任务是否都完成了

但是 那个时候使用的是Runnable做的 是没有返回结果的 这样就比较麻烦 处理的时候】

但是 老师不清楚有什么用

---- 以上就是Future和Callable

Callable可以返回结果 --- 结果由Future拿到

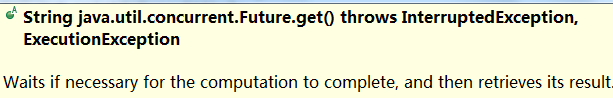
那么在运行的过程中 future.get()方法就是一直等

----- 有没有一种情况 不想等 就是 去看 如果 没有结果 我去做别的





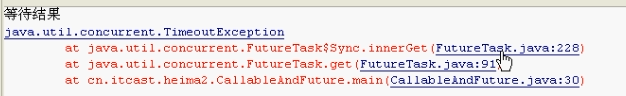
【



注意这个get方法是一定能获取正确结果的 这是因为 get写的是等待异步计算完成 然后获取他的结果】

也就是1s之后 看一下结果

没有的话 就超时 抛出异常



----- 因为等待结果 但是结果没有完成 我们的timeout参数 1s就到了 这个时候 就报错了 因为计算超时

【注意 Runnable和Callable的实例都是任务类

都表示任务 --- 一个有返回结果的任务 一个没有返回结果的任务

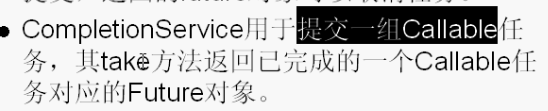
一个可以抛异常 一个不可以抛异常】

---- 以上是Callable和Future

==============================

现在的应用是 一下子提交一组Callable任务

然后用take去获取





演示



看有什么子类



参数是线程池



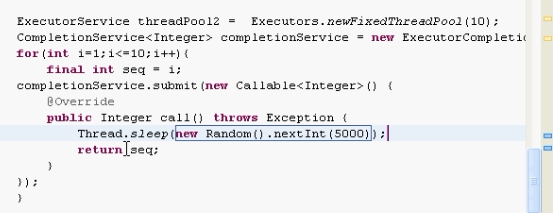


现在 使用CompletionService为其提交10个任务



现在使用CompletionService提交10个任务

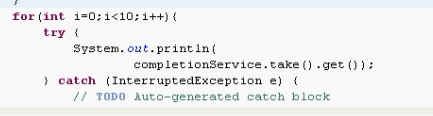
返回内容是自己的编号

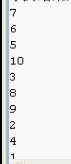


这样 提交了一堆任务

最后 等待收货 调用completion的take方法

返回的结果是Future



 ---- 提交完10个任务 就拿到10个结果