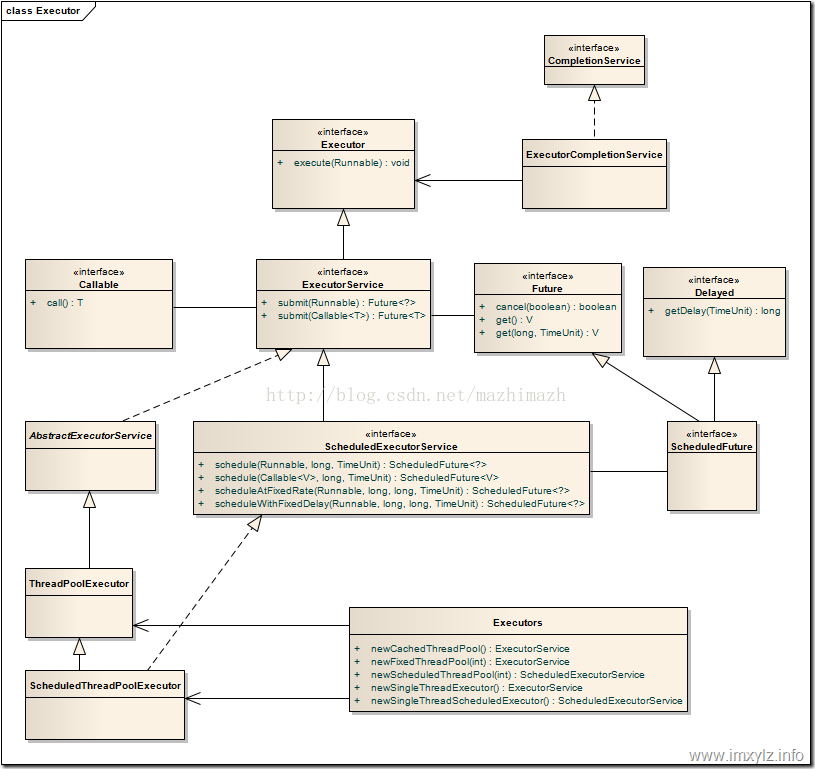
**线程和线程池**

* **线程池**

<http://www.tianshouzhi.com/api/tutorials/mutithread/317>

参考http://www.jianshu.com/p/153053aad89e

下面是线程池的框架图：

****

* **Executor**

Android中的线程池概念来自于java的Executor,Executor是一个接口，真正线程池实现类ThreadPoolExecutor。这个接口存在的目的是：提供一种将“任务提交”和“任务如何运行”分离开来的机制：

**public interface** Executor {**void** execute(Runnable command);//执行已提交的Runnable任务对象  
 }

虽然只有一个方法，却为灵活强大的异步任务执行框架提供了基础，它将任务的提交过程和执行过程完美的解耦开来，并用Runnable来表示任务。我们想要得到Executor对象需要借助Executors工具类。

* **Executors**

它主要用来创建线程池ThreadPoolExecutor，代理了线程池的创建，使你的创建ThreadPoolExecutor的入口参数变得简单，通过方法名便知道了你要创建线程池是什么样，常见的方法有（都是静态方法）：

1. **newFixedThreadPool**:

**public static** ExecutorService newFixedThreadPool(**int** nThreads) {  
 **return new** ThreadPoolExecutor(nThreads, nThreads,  
 0L, TimeUnit.***MILLISECONDS***,  
 **new** LinkedBlockingQueue<Runnable>());  
 }

它是线程数量固定的线程池，当任务数量大于设定的核心线程数量时，多余的任务将会被放到队列中去（任务队列没有大小限制），当有线程空闲时会继续执行队列中的任务，这些线程都是核心线程，没有超时机制，他们只要在线程池被关闭的时候才会被回收：

1. **newCachedThreadPool**:

**public static** ExecutorService newCachedThreadPool() {  
 **return new** ThreadPoolExecutor(0, Integer.***MAX\_VALUE***,  
 60L, TimeUnit.***SECONDS***,  
 **new** SynchronousQueue<Runnable>());  
 }

它是一个没有固定大小且能比较快速就能完成小任务的线程池，它将为每一个任务创建一个线程，不会让任务等待。它和普通线程的唯一区别是，它开启单个的线程能缓存60秒（也可以看成是60s超时的超时机制），当一个线程执行完任务便进入空闲状态时，空闲状态超过60s线程就会被收回。否则有新的任务来，会将任务交给这个空闲的线程来处理，当所有的线程都处于空闲状态时，线程池中的线程会都会因为超时而被停止，这时候线程池中实际上没有任何线程的，他几乎不占用任何系统资源。

1. **newSingleThreadPool：**

**public static** ExecutorService newSingleThreadExecutor() {  
 **return new** FinalizableDelegatedExecutorService  
 (**new** ThreadPoolExecutor(1, 1,  
 0L, TimeUnit.***MILLISECONDS***,  
 **new** LinkedBlockingQueue<Runnable>()));  
 }

这类线程内部只有一个核心线程，他确保所有任务都在同一个线程中按顺序执行，它的意义在于统一所有的外界任务到一个线程中，这样就不需要处理线程同步问题。它和调用Executors.newFixedThreadPool(1)得到的结果一样。

1. **newScheduledThreadPool：**

**public static** ScheduledExecutorService newScheduledThreadPool(**int** corePoolSize) {  
 **return new** ScheduledThreadPoolExecutor(corePoolSize);  
 }

**public** ScheduledThreadPoolExecutor(**int** corePoolSize) {  
 **super**(corePoolSize, Integer.***MAX\_VALUE***,  
 ***DEFAULT\_KEEPALIVE\_MILLIS***, ***MILLISECONDS***,  
 **new** DelayedWorkQueue());  
 }

它的核心线程是固定的，非核心线程数量没有限制，并且当非核心线程闲置时会被立即回收，它以延迟或定时的方式来执行任务。

* **ExecutorService**

它是一个任务周期管理接口，它定义了一些生命周期的方法：

**public interface** ExecutorService **extends** Executor {  
  
 //顺次地关闭ExecutorService,停止接收新的任务，等待所有已经提交的任务执行完毕之后，关闭

ExecutorService**void** shutdown();  
  
 //阻止等待任务启动并试图停止当前正在执行的任务，停止接收新的任务，返回处于等待的任务列表

List<Runnable> shutdownNow();  
  
 //判断线程池是否已经关闭**boolean** isShutdown();  
  
 //如果关闭后所有任务都已完成，则返回 true。注意，除非首先调用 shutdown 或 shutdownNow，否

则 isTerminated 永不为 true。**boolean** isTerminated();  
  
 //等待（阻塞）直到关闭或最长等待时间或发生中断,timeout - 最长等待时间 ,unit - timeout 参

数的时间单位如果此执行程序终止，则返回 true；如果终止前超时期满，则返回 false **boolean** awaitTermination(**long** timeout, TimeUnit unit)  
 **throws** InterruptedException;  
  
 //提交一个返回值的任务用于执行，返回一个表示任务的未决结果的 Future。该 Future 的 get 方

法在成功完成时将会返回该任务的结果。

<T> Future<T> submit(Callable<T> task);  
  
 //提交一个 Runnable 任务用于执行，并返回一个表示该任务的 Future。该 Future 的 get 方法在

成功 完成时将会返nullFuture<?> submit(Runnable task);  
  
 //执行给定的任务，当所有任务完成时，返回保持任务状态和结果的 Future 列表。返回列表的所有

元素的 Future.isDone() 为true。<T> List<Future<T>> invokeAll(Collection<? **extends** Callable<T>> tasks)  
 **throws** InterruptedException;  
//执行给定的任务，当所有任务完成时，返回保持任务状态和结果的 Future 列表。返回列表的所有

元素的 Future.isDone() 为 true。<T> List<Future<T>> invokeAll(Collection<? **extends** Callable<T>> tasks,  
 **long** timeout, TimeUnit unit)  
 **throws** InterruptedException;  
  
 //执行给定的任务，如果在给定的超时期满前某个任务已成功完成（也就是未抛出异常），则返回其

结果。一旦正常或异常返回后，则取消尚未完成的任务。<T> T invokeAny(Collection<? **extends** Callable<T>> tasks)  
 **throws** InterruptedException, ExecutionException;  
<T> T invokeAny(Collection<? **extends** Callable<T>> tasks,  
 **long** timeout, TimeUnit unit)  
 **throws** InterruptedException, ExecutionException, TimeoutException;  
}

* **ThreadPoolExecutor**

ThreadPoolExecutor是线程池的真正实现类，它的构造方法提供了一些列参数来配置线程池：

**public** ThreadPoolExecutor(**int** corePoolSize,  
 **int** maximumPoolSize,  
 **long** keepAliveTime,  
 TimeUnit unit,  
 BlockingQueue<Runnable> workQueue,  
 ThreadFactory threadFactory,  
 RejectedExecutionHandler handler) {  
 **if** (corePoolSize < 0 ||  
 maximumPoolSize <= 0 ||  
 maximumPoolSize < corePoolSize ||  
 keepAliveTime < 0)  
 **throw new** IllegalArgumentException();  
 **if** (workQueue == **null** || threadFactory == **null** || handler == **null**)  
 **throw new** NullPointerException();  
 **this**.**corePoolSize** = corePoolSize;  
 **this**.**maximumPoolSize** = maximumPoolSize;  
 **this**.**workQueue** = workQueue;  
 **this**.**keepAliveTime** = unit.toNanos(keepAliveTime);  
 **this**.**threadFactory** = threadFactory;  
 **this**.**handler** = handler;  
}

* **corePoolSize ：**

线程池的的核心线程数，默认情况下即使在闲置状态也会一直存活。如果将ThreadPoolExecutor的allowCoreThreadTimeOut设置为true，那么闲置的核心线程会有超时策略。这个时间由keepAliveTime所指定，当闲置的核心线程等待时间超过这个时间，核心线程就会被终止。

* **maximumPoolSize ：**

线程池所维护的最大线程数，当活动的线程数达到这个数值后，后续的新任务将会被阻塞。

注意这个值和corePoolSize值的关系：**maximumPoolSize = corePoolSize + 非核心线程数**。

maximumPoolSize > corePoolSize :

当所有核心线程执行任务时且任务队列满了之后再添加任务进来线程池将会开启一个非核心线程来

执行任务，如果继续有任务进来当maximumPoolSize = corePoolSize + 非核心线程数，线程池将拒

绝执行任务。

maximumPoolSize = corePoolSize :

这两个值相等说明线程池不会开启非核心线程，当所有核心线程执行任务时且任务队列满了之后再添

加任务进来线程池将会拒绝执行任务。

我们可以指定maximumPoolSize 和corePoolSize 的大小，但是corePoolSize 永远要小于等于

maximumPoolSize。：

* **workQueue** （BlockingQueue<Runnable> workQueue）**：**

任务队列，当核心线程全部处理任务后，当有新的任务来就会放到这个队列中。任务队列可以在定义线程池的时候选择有限的队列或者无限的队列，使用无限的队列后，任务队列永远不会满，则线程池就不会开启非核心线程来执行任务。

* **keepAliveTime ：**

非核心线程闲置时的超时时长，超过这个时长，闲置的非核心线程就会被回收。当allowCoreThreadTimeOut设置为true时，这个值也作用于核心线程。

* **threadFactory ：**

线程工厂，为线程池提供创建新线程的功能。ThreadFactory是一个接口，他只有一个方法，Thread newThread(Runnable r);

* **unit ：**

用于指定keepAliveTime参数的时间单位，这是一个枚举。

* **handler ：**

当线程池无法执行新任务时，这可能是任务队列已满或者是无法成功执行任务，这个时候ThreadPoolExetor会调用handler的rejectedExecution方法来通知调用者。rejectedExecution是一个接口

**public interface** RejectedExecutionHandler {**void** rejectedExecution(Runnable r, ThreadPoolExecutor executor);  
 }

ThreadPoolExetor为RejectedExecutionHandler提供了几个可选类，它们实现

了这个接口：

1、CallerRunsPolicy：如果发现线程池还在运行，就直接运行这个线程。否则丢弃这个任务。

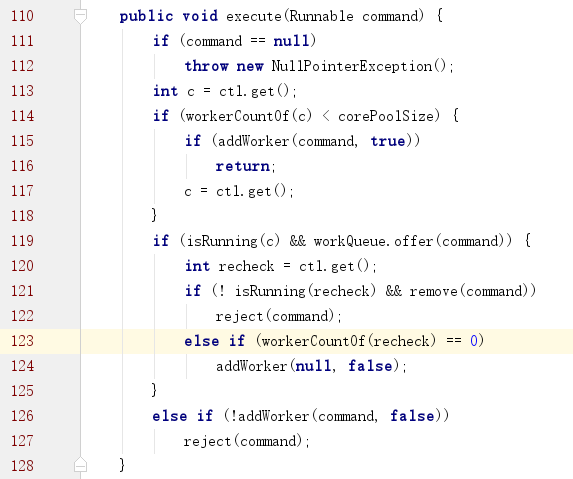
2、DiscardOldestPolicy：在线程池的等待队列中，将队列第一个任务取出一个抛弃，然后将 当前线程放进去。

3、DiscardPolicy：什么也不做。

4、AbortPolicy：默认值，抛出一个异常：RejectedExecutionException。

* **ThreadPoolExecutor源码解析：**

首先从实现Executor接口的execute（Runnable command）方法开始：



**ctl**是一个AtomicInteger对象，它能线程安全的操作Integer类。它存储的值代表两个部分：一个是当前已经产生的线程数量包括核心线程和非核心线程，一个线程池状态。也就是说线程池将当前线程数量和线程池状态打包成一个Integer值赋给ctl对象：

打包和解包方式：

*// Packing and unpacking ctl* **private static int** runStateOf(**int** c) { **return** c & ~***CAPACITY***; }  
 **private static int** workerCountOf(**int** c) { **return** c & ***CAPACITY***; }  
 **private static int** ctlOf(**int** rs, **int** wc) { **return** rs | wc; }

其中线程池状态包括：

***RUNNING = -1***：运行状态，接受新任务并且处理新任务。

***SHUTDOWN = 0***：不接受新任务，但是处理队列任务。

***STOP = 1***：不接受新任务，也不处理队列任务，并且中断正在运行的任务。

***TIDYING = 2***：所有的任务已经终止，workercount为零。转换到这个状态后，将运行

terminated()方法。

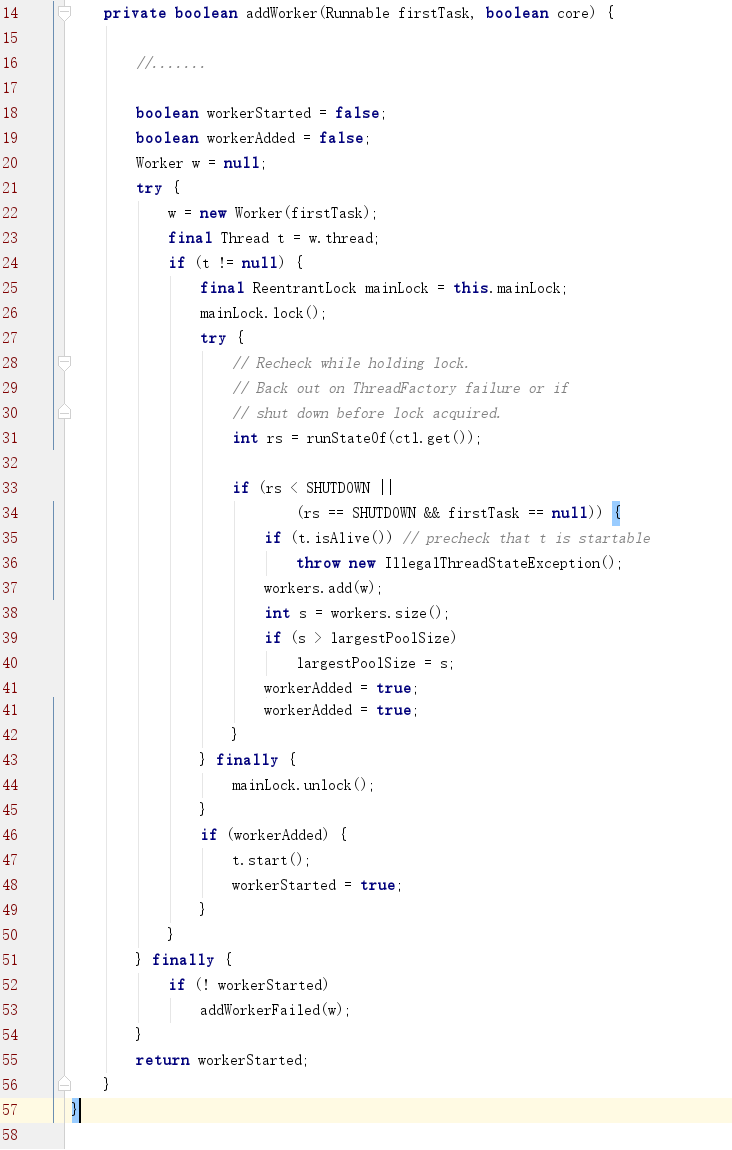
***TERMINATED = 3***：运行完terminated()方法，线程池将会切换到这个状态。

这些值的顺序很重要，可以顺序比较的。

这个方法执行主要内容：

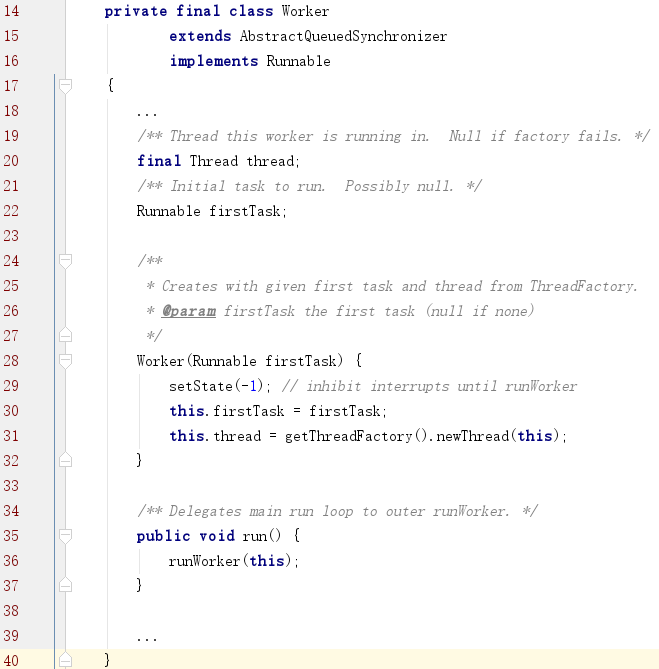
1. 我们先从114行开始看，如果工作线程数量<核心线程数量，通过addWork()向线程池添加一个任务，并使用核心线程来立即执行。
2. 如果工作线程数量>=核心线程数量，执行119行任务会被添加到任务队列workQueue中，等待执行。
3. 如果上面将任务添加到workQueue中失败，就会执行到126行，使用addWork()向线程池中添加一个任务使用非核心线程来立即执行，如果添加失败表明线程池拒绝执行，那么会通过handler通知调用者来处理这个任务。

在execute()方法中执行的比较重要的一个方法就是addWorker()方法，第一个参数假如不为空则执行传进来的任务，如果为空，则执行任务队列中的任务。它的第二个参数core比较重要，如果core为true说明会添加一个核心线程，如果为false说明会添加一个非核心线程，这个判断在下面源码图片中被省略：



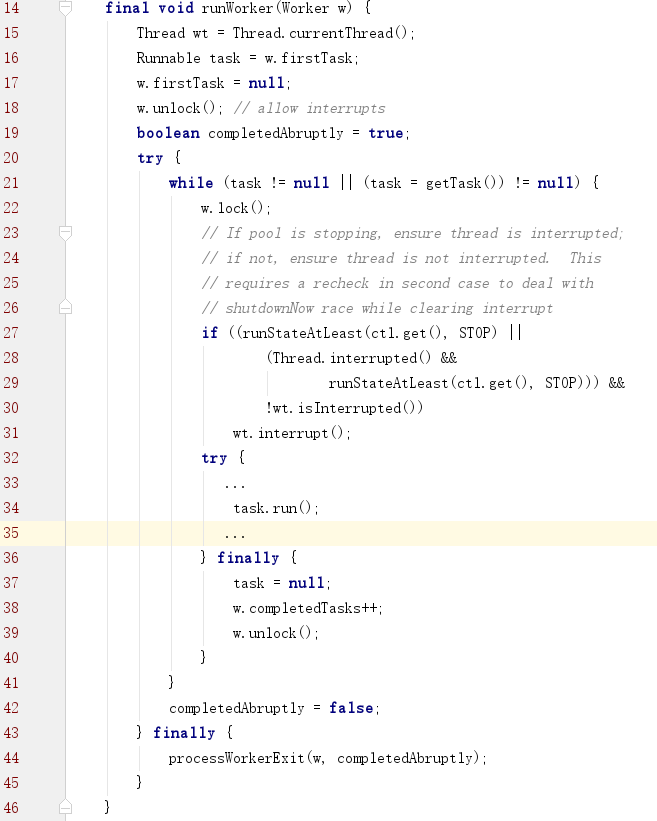
第16行省略了一部分代码，这部分代码主要是让**ctl**存储的线程数量加1。线程数增加后。继续22行创建一个Worker对象w并将任务交给它,Worker的构造方法中会创建一个线程。继续33行判断线程池状态，假如线程池正在运行会将w对象添加到workers集合中然后运行w中的线程，最终这个线程池开启了一个线程执行了传进来的任务。

addwork()方法中重点是Worker这个类,下面看主要的代码：

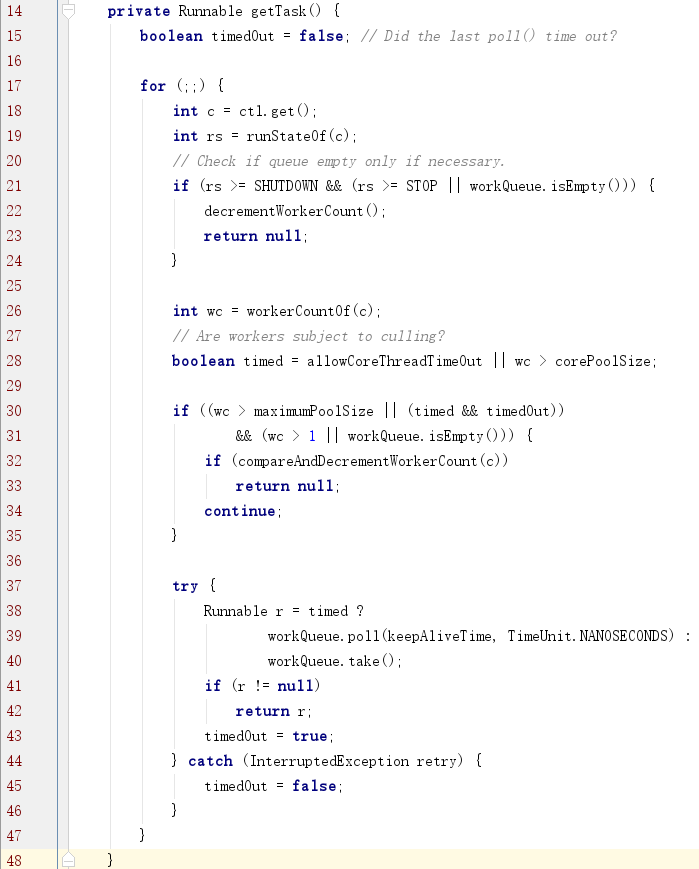


我们可以看到Worker这个类有一个Thread和一个Runnable，在构造方法中会初始化这两个对象。Worker类实现了Runnable接口，并且创建thread对象的时候将woeker当前对象传进去，所以Worker类中的run方法才是真正的线程体。这说明Worker类才是真正的工作线程类，创建的worker对象可以执行很多外界传递进来任务。

在addWork()中启动线程会调用Worker类中的线程体run方法。run方法主要调用了runWorker(),并将worker对象传递进去，这个方法实现在ThreadPoolExecutor类中：



16行中把Worker中任务拿出来，然后21行判断task是否为空，假如不为空则在这个工作线程中执行这个任务task.run，假如为空则通过getTask()获取任务队列中的任务来执行:



这就一下子很明白了原来getTask()方法中是一个死循环，它的任务就是从人物队列中拿出一个任务让工作线程去执行，工作线程的生死是由这个方法决定的，首先看21行，当程序关闭了线程池，如果线程池的状态为SHUTDOWN的时候任务队列为空或者状态为STOP，那么会立即返回一个空任务结束掉工作线程。继续30行，如果线程池内线程数量大于设置的最大线程数量，或者当前工作线程超时那么也会返回一个空任务结束掉工作线程。继续38行，**如果当前工作线程没有设置超时时间，那么由任务队列workQueue会通过take()来取任务，如果workQueue不为空，那么会立即返回队列中第一个任务，如果为空只要线程池的状态小于SHUTDOWN则这个工作线程会一直阻塞在take()方法里，当有新的任务添加到任务队列当中，则立即返回这个新任务，这就是没有设置超时时间的核心线程不会被回收的机制。如果当前工作线程设置了超时时间，任务队列workQueue会通过poll()来取任务，并且将超时时间传递进去。假如任务队列中有任务，则立即返回队列中第一个任务。如果队列为空，那么由poll这个方法来计算超时时间，如果在超时时间内有新的任务添加到任务队列中，那么会立即返回这个这个新的任务，一旦时间超时则立即返回一个null对象。接下来在第43行timeOut会被设置为true，继续循环当执行到第30行的判断时会使getTask()方法返回一个空对象，进而结束掉当前的工作线程，这就是设置了超时时间的工作线程的超时机制（包括核心和非核心的线程）。**

通过对上面的源码分析，我们基本上算是搞清了线程池的运行机制，下面来大体梳理一下：

1、首先我们创建一个线程池，假设设置的maximumPoolSize比corePoolSize要大，当我们添加第一个任务后，通过execute(Runnable command)方法可以看出会调用addWork(command,true);第二个参数为true表明要添加一个核心线程。

1. 在addWork()方法中将线程池中ctl线程数加1，接着创建一个工作线程Worker对象，并添加到workers集合中，这个Worker对象里面的线程就是我们的核心线程。最后启动这个核心线程并执行Worker类中的线程体run方法，run方法调用ThreadPoolExecutor类中的runWorker(Worker w);
2. runWorker(Worker w)方法中将w工作线程中的任务取出来赋给task并删除w工作线程中的任务。然后用一个while循环来监听task对象是否为空，假如不为空则执行这个任务，执行完了之后将task置空。假如为空则调用getTask()方法返回一个任务赋给task继续执行，这样这个工作线程Worker对象就会一直执行任务；当getTask()返回为空则跳出while循环，工作线程将不在执行新的任务，最后调用processWorkerExit方法退出，退出后工作线程结束。

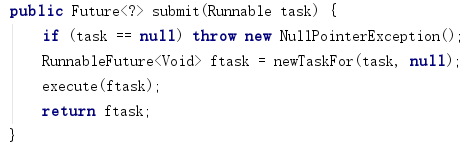
调用processWorkerExit(Worker w, boolean completedAbruptly)方法，这个方法的第二个参数，

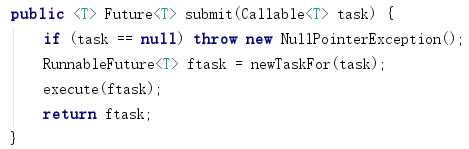
若为true：说明while循环是异常跳出,执行这个方法时首先线程池内ctl中的线程数量会减1，接着会在workers中将这个工作线程移除。假如当前线程池的运行状态小于STOP，说明线程池至少还能处理队列中的任务，则会继续启动一个非核心线程来继续执行队列中的任务。

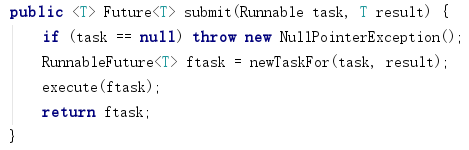
若为false:说明while循环是执行完任务正常跳出，任务队列中已经没有任务，执行这个方法时线程池中的线程数量不会减1，接着会在workers中将这个工作线程移除。假如当前线程池的运行状态小于STOP会继续判断，如果当前线程池数量小于设置的核心线程数量那么会继续启动一个非核心线程来继续执行队列中的任务，如果不小于则不启动线程。

1. 上面说正常情况下，只有getTask()返回空才会退出工作线程，这个方法里面也是一个死循环，它会一直监听任务队列，如果当前工作线程设置有超时机制，在超时的时间内如果任务队列中没有任务那么当前线程会被关闭，如果一直有任务，那么当前工作线程会一直执行。如果没有设置超时时间，则工作线程会一直堵塞在这个方法，除非线程池关闭。

* **submit()方法**

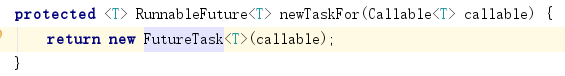


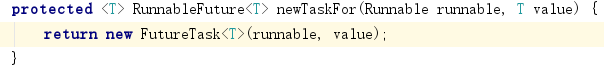




submit()方法最终还是调用execute()方法，只是使用summit()方法会返回一个带有泛型的Future对象。首先明确一下这种方式的不同，使用submit方式提交线程，当线程执行完毕后可以通过Future对象的get方法获取线程的返回值，这个返回值就是我们在第三个方法中传进去的result，不管传进什么类型的对象，当线程执行完毕后它又会把这个对象返回回来，我们就可以知道线程执行完毕了。第一个方法中传入的空，get()方法会返回null。第二个方法传入的是一个Callable对象，这是一个接口里面只有一个带返回参数的call方法，我们写实现call方法后，返回什么值，使用get方法就会获取到什么值。

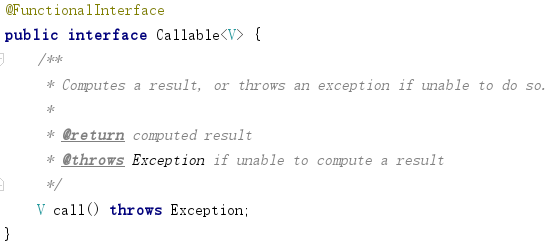
在submit()方法中我们调用execute()方法的时候传入的是一个RunnableFuture对象。这个对象接收的是newTaskFor()方法的返回值：





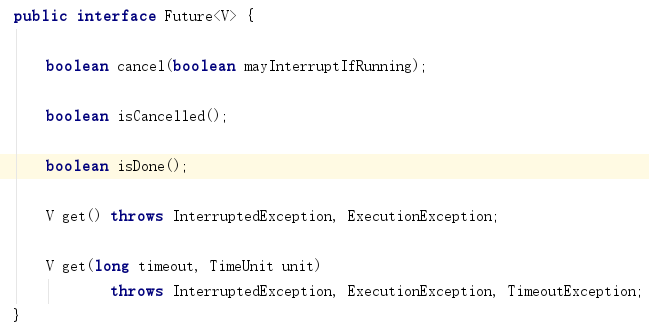
它里面又new了一个FutureTask对象，这个对象既能接受Runnable对象又能接受Callable对象。好了，到此为止又见到了好多新类和接口：Callable、FutureTask、Future、RunnableFuture，下一节一一介绍。

* **Callable**



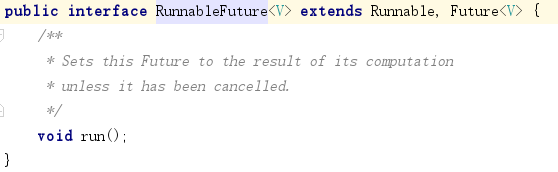
Callable是一个带返回值并且能抛出异常的泛型接口。call()方法返回的类型就是泛型里传进来的类型，如果不能返回计算值那么就会抛出一个异常。

* **Future**



Future是一个泛型接口，它里面的方法都是控制或者监听任务执行的方法。

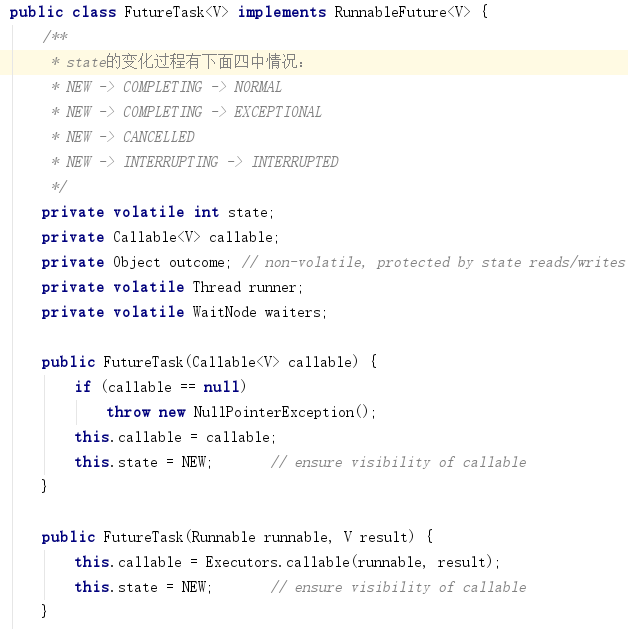
* **RunnableFuture**



RunnableFuture接口继承了Runnable接口和future接口，并且加载了run方法。

* **FutureTask**

FutureTask是对线程执行的任务的进一步封装，它可以监控任务的执行状态，取消任务执行，获取任务执行结果。



上面源码只包括变量和构造方法，FutureTask类实现了RunableFuture接口，也就是说它实现了Future和Runnable的全部方法，首先解读一下它里面包含的变量：

**state：**任务的运行状态。

**callable：**用于执行具体任务的Callable对象。

**outcome：**任务的返回值。只能调用get()方法获取。

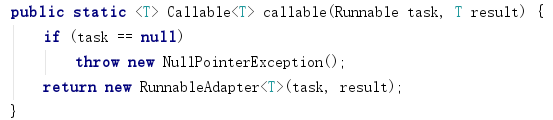
**runner：**当前执行任务的线程。

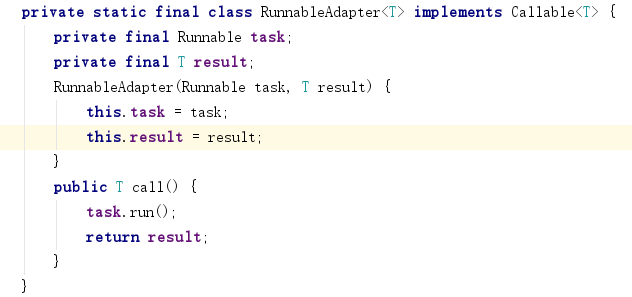
**waiters：**存储调用get()方法的线程的节点链表。

任务的状态：

//当FutureTask实例刚刚创建到callbale的call方法执行完成前，处于此状态  
 private static final int NEW = 0;  
 //callable的call方法执行完成或出现异常时，首先进行此状态  
 private static final int COMPLETING = 1;  
 //callable的call方法正常结束时，进入此状态，将outcom设置为正常结果  
 private static final int NORMAL = 2;  
 //callable的call方法异常结束时，进入此状态，将outcome设置为抛出的异常  
 private static final int EXCEPTIONAL = 3;  
 // FutureTask任务尚未执行，即还在任务队列的时候，调用了cancel方法，进入此状态  
 private static final int CANCELLED = 4;  
 // FutureTask的run方法已经在执行，收到中断信号，进入此状态  
 private static final int INTERRUPTING = 5;  
 // 任务成功中断后，进入此状态  
 private static final int INTERRUPTED = 6;

两个构造方法都是初始化callable和state变量，第二个构造方法中传入的Runnable对象和返回值result，通过Executors.callable()方法转换成一个callable对象如下：



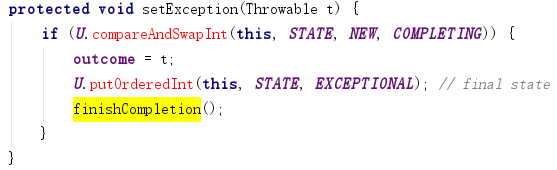


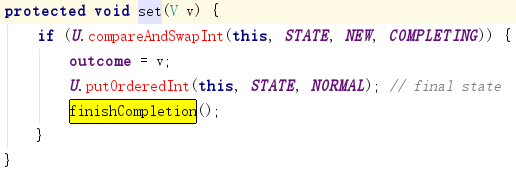
Executors.callable()返回的是一个RunnableAdapter适配类，这个类实现了Callable接口call()方法，call()方法中执行的是Runnable的run方法并且返回传进来的result返回值。这里主要是用了适配器模式，兼容了Runnable类型的任务，当适配类的call方法被执行，那么Runnable的run方法也就被执行了。

通过线程池submit()方法提交的任务最终被封装到了FutureTask类里并交给线程池的execute()方法去执行，FutureTsak类实现了Runnable接口的run方法，所以线程池里的工作线程从FutureTask对象的run方法开始执行：

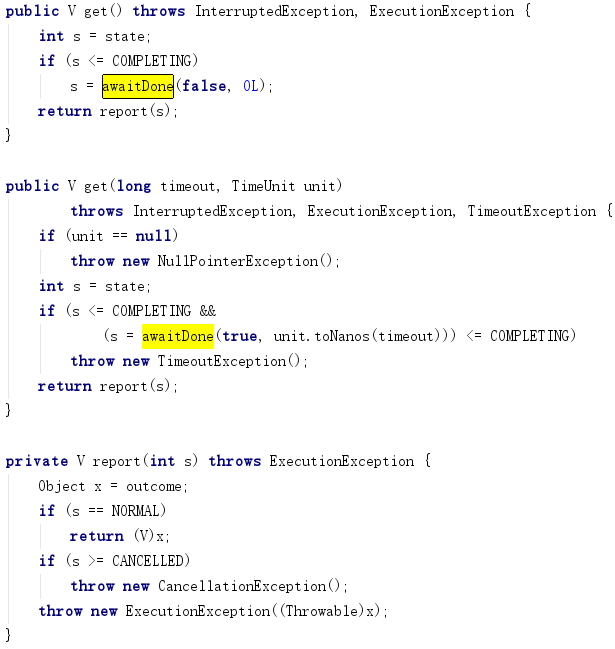


首先判断如果state不是NEW状态或者获取当前线程设置给runner为空，则直接返回。判断通过，继续233行继续检查当前state状态为NEW且callable不为空，则执行callable的call()方法，这样我们通过线程池的summit()方法提交进来的任务也就被执行了，执行完毕将返回值设置给result同时run设置为true。继续看242行如果出现异常则执行setException()方法，如果ran为true则执行set方法:





这两个方法使用Unsafe修改state的状态为COMPLETING，如果已经是COMPLETING则if判断会返回false，否则返回true。如果抛出异常setException()方法将异常设置给outcome，并且将state状态设置为EXCEPTIONAL。如果没有异常set()方法将返回值设置给outcome并且将state状态设置为NORMAL。最后调用finishCompletion()方法。outcome的值只能通过get()方法获取：



可以看到两个get()方法都没有直接返回outcome对象，而是最后通过report()方法返回的，如果outcome存储的是run()方法中抛出来的异常，就将异常继续往上抛出，所以调用get()方法是必须try{}catch{}一下异常。再return返回值之前还调用了awaitDown():



21行判断为true说明已经执行完了set()方法或者setException()方法，直接返回状态交由report()处理；

26行判断为true说明outcome赋值还没有完成，则让出cup资源让其他线程执行或者自己这个线程继续执行；

30行判断检测线程是否中断，如果中断则移除链表中的相应节点并抛出异常；

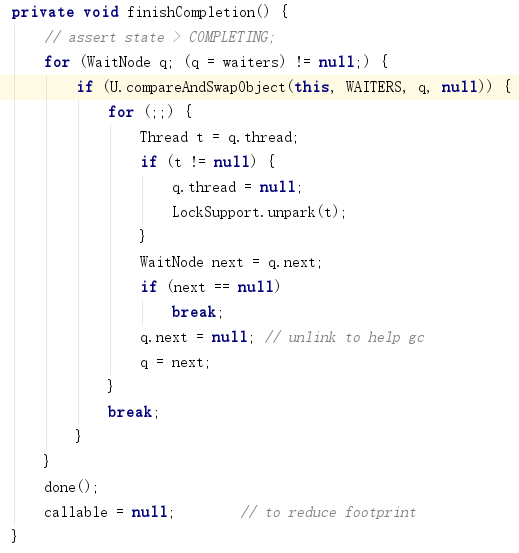
34行判断为true说明任务还没有执行到set()方法或者setException()方法，这里面会继续判断如果设置了超时并且超时时间已经到了，那么会直接返回任务状态给report()方法,否则会创建一个持有当前线程的等待节点；

39行判断检测如果存储当前调用线程的节点还没有加入waiters队列中，那么就加入；

42行判断如果设置了超时机制，并且已经超过了设置的等待时间，那么立即返回当前任务的state交由report()处理。如果没有超时，则判断当前任务状态如果小于COMPLETING则直接将当前线程阻塞指定时间，如果不小于则继续执行通过21行或者26行判断直到返回当前状态交由report()处理，不会再执行到42行判断了。也就是说我们**调用get(long timeout, TimeUnit unit)方法时如果在设置的时间内任务还没有计算完，那么会堵塞调用线程指定的时间，一旦到达指定时间，如果任务已经执行完那么正常返回结果，如果任务没有执行完则抛出异常；如果在设置的时间内任务已经计算完，则立即返回结果，不用等到超时时间再返回。**

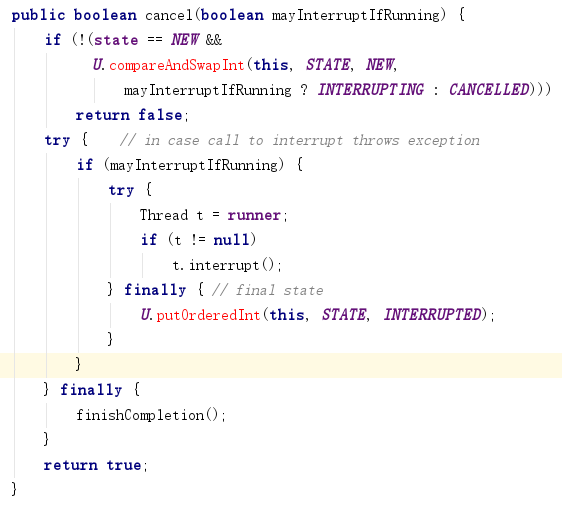
61行如果没有设置等待时间则立即堵塞调用线程。

通过对awaitDone()方法的解析，可以知道如果调用get()或者get(long timeout, TimeUnit unit)方法获取执行结果可能会调用LockSupport.parkNanos(this, parkNanos);或LockSupport.park(this)堵塞调用线程，这种线程堵塞最终会在任务执行完之后调用set()或者setException()方法中执行finishCompletion()方法来解除调用线程的堵塞状态：



无论任务是正常结束还是异常结束，或者任务被取消都会调用这个方法，将waiters链表中存储的所有的调用线程唤醒，并且从链表中移除。唤醒之前outcome已经有了返回值，任务的状态也已经大于COMPLETING状态了，这样get()方法就能正常的返回值了。

**取消任务**：



cancel()只能取消状态为NEW的任务，状态为NEW时分两种情况，任务是否已经执行，即FutureTask的run()方法是否已经执行：

没有执行run()：运行完cancel()方法后，这个任务永远不会被执行了。

已经执行run()：是否中断任务要看参数mayInterruptIfRunning的值，传入true则中断，传入false则让任务执行完毕。

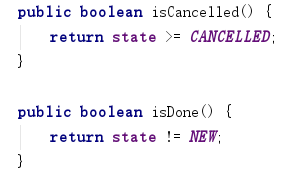
参数mayInterruptIfRunning：是否中断正在运行的任务。

传入true：任务的状态从NEW->INTERRUPTING，允许中断执行任务的线程，即中断任务。

传入false：任务的状态从NEW->CANCELLED，不会强制中断正在运行的任务，会让任务执行完。

cancel()返回true说明取消成功，然后执行finishCompletion()方法如果有被堵塞的调用线程会唤醒被堵塞住的调用线程,否则取消失败直接返回false。

**是否取消isCancelled()、是否执行isDone()**

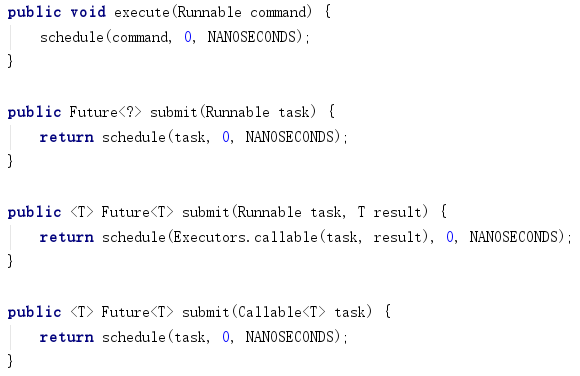


* **ScheduledThreadPoolExecutor**

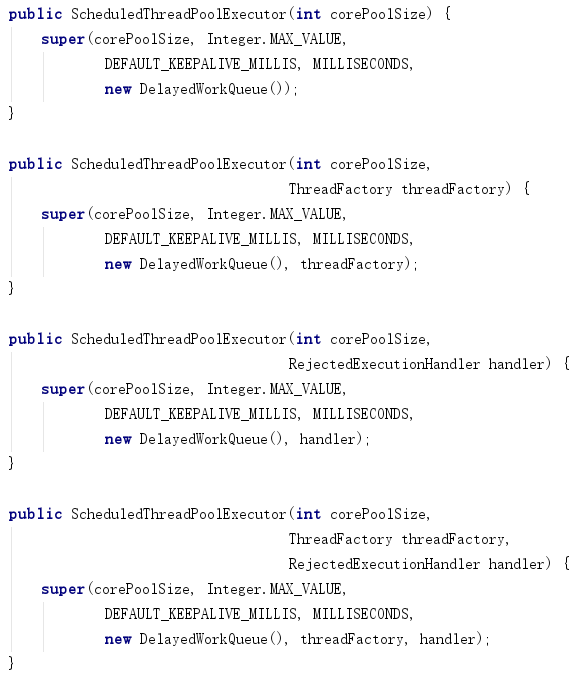
ScheduledThreadPoolExecutor是主要用来定时或者周期性执行任务的线程池，它继承了ThreadPoolExecutor并实现了ScheduledExecutorService接口：



使用ScheduledThreadPoolExecutor执行延迟任务或者周期性任务时可以直接调用上面的4个方法。调用ScheduledThreadPoolExecutor类的submit()和execute()方法不会有延迟，这两个方法直接调用的schedule()方法，只不过把延迟时间设置为0：



**构造方法**

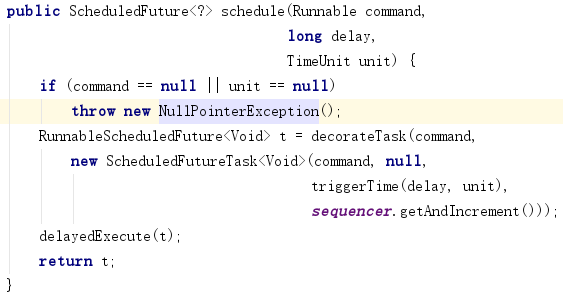


可以看到ScheduledThreadPoolExecutor的构造方法调用的是ThreadPoolExecutor，这里有两点需要注意：

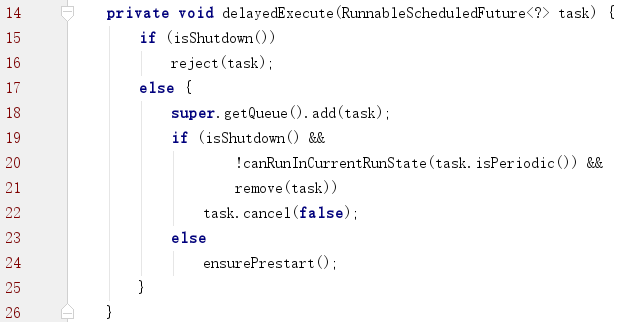
1. 它传进去的是一个DelayedWorkQueue工作队列，它是一个无界队列。
2. 它的最大线程数maximumPoolSize设置的也是无限大。由于使用的无界队列DelayedWorkQueue，所以当核心线程满了之后，再有提交的任务都会被添加进工作队列不会在开启非核心线程来执行了，所以maximumPoolSize这个值的意义也就不大了。

**源码分析**

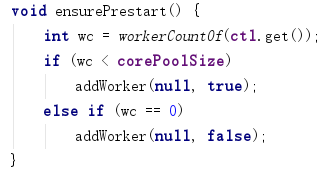
由于submit()和execute方法直接调用的schedule()方法：



比较简单，decorateTask()主要是将创建的ScheduledFutureTask对象返回回来交给delayedExecute()方法：



首先判断线程池的状态，如果线程池状态不是***RUNNING*** 则交给**handler**处理。接下来18行，首先将任务添加到无界队列中，再次判断线程池状态，如果线程池不能执行任务，则将任务从队列中移除，否则继续24行调用ensurePrestart()方法启动线程处理任务：



**总结：ScheduledThreadPoolExecutor一定是将任务先添加进任务队列，然后在执行任务。**