参考：http://www.cnblogs.com/dolphin0520/p/3932921.html

## 什么是线程池，为什么要使用线程池

在java中，如果每个请求到达服务器就创建一个新线程，开销是相当大的，甚至可能要比在处理实际的用户请求的时间和资源要多的多。除了创建和销毁线程的开销之外，活动的线程也需要消耗系统资源。如果在一个jvm里创建太多的线程，可能会使系统由于过度消耗内存或“切换过度”而导致系统资源不足。为了防止资源不足，服务器应用程序需要采取一些办法来限制任何给定时刻处理的请求数目，尽可能减少创建和销毁线程的次数，特别是一些资源耗费比较大的线程的创建和销毁，尽量利用已有对象来进行服务，这就是“池化资源”技术产生的原因。线程池主要用来解决线程生命周期开销问题和资源不足问题。通过对多个任务重复使用线程，线程创建的开销就被分摊到了多个任务上了，而且由于在请求到达时线程已经存在，所以消除了线程创建所带来的延迟。这样，就可以立即为请求服务，使应用程序响应更快。另外，通过适当的调整线程中的线程数目可以防止出现资源不足的情况。

## 线程池的组成部分

一个比较简单的线程池至少应包含线程池管理器、工作线程、任务列队、任务接口等部分。

**线程池管理器：**它的作用是创建、销毁并管理线程池，将工作线程放入线程池中；

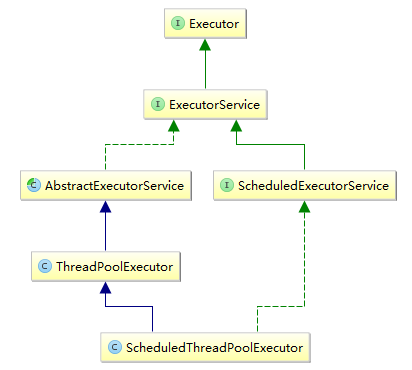
**工作线程：**它是一个可以循环执行任务的线程，在没有任务是进行等待；

**任务列队：**它的作用是提供一种缓冲机制，将没有处理的任务放在任务列队中；

**任务接口：**它是每个任务必须实现的接口，主要用来规定任务的入口、任务执行完后的收尾工作、任务的执行状态等，工作线程通过该接口调度任务的执行。

## Java中线程池的继承架构

程序启动一个新线程成本是比较高的，因为它涉及到要与操作系统进行交互。而使用线程池可以很好的提高性能，尤其是当程序中要创建大量生存期很短的线程时，更应该考虑使用线程池。线程池里的每一个线程代码结束后，并不会死亡，而是再次回到线程池中成为空闲状态，而是等待下一个对象来使用。在JDK5之前，我们必须手动实现自己的线程池，从JDK5开始，Java内置支持线程池。



### Executor

**Executor接口**是Java中线程池的一个顶级接口，在它里面只声明了一个方法execute(Runnable)，返回值为void，参数为Runnable类型，从字面意思可以理解，就是用来执行传进去的任务的，但是严格意义上讲Executor并不是一个线程池，而只是一个执行线程的工具。真正的线程池接口是ExecutorService。

Excutor接口源码：

**public interface Executor** {**void** execute(Runnable command);  
}

### ExecutorService

**ExecutorService接口**继承了Executor接口，并声明了一些方法：submit、invokeAll、invokeAny以及shutDown等方法，他是真正的线程池接口，其中该接口提供的submit方法可以执行Runnable对象或者Callable对象代表的线程。

ExcutorService接口源码：

**public interface ExecutorService extends** Executor {

**boolean** isShutdown();

**void** shutdown();List<Runnable> shutdownNow();**boolean** isTerminated();**boolean** awaitTermination(**long** timeout, TimeUnit unit) **throws** InterruptedException;

Future<?> submit(Runnable task);<T> Future<T> submit(Callable<T> task);<T> Future<T> submit(Runnable task, T result);

<T> List<Future<T>> invokeAll(Collection<? **extends** Callable<T>> tasks) ;<T> List<Future<T>> invokeAll(Collection<? **extends** Callable<T>> tasks, **long** timeout, TimeUnit unit);<T> T invokeAny(Collection<? **extends** Callable<T>> tasks);<T> T invokeAny(Collection<? **extends** Callable<T>> tasks, **long** timeout, TimeUnit unit);  
}

### AbstractExecutorService

AbstractExecutorService抽象类实现了ExecutorService接口，基本实现了ExecutorService中声明的方法。

### ThreadPoolExecutor

**ThreadPoolExecutor类**继承了AbstractExecutorService，它是线程池ExecutorService的默认实现，是线程池中最核心的一个类。

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

#### 构造器参数说明

从上面的代码可以得知，ThreadPoolExecutor继承了AbstractExecutorService类，并提供了多个构造器，事实上，通过观察每个构造器的源码具体实现，发现其他构造器最终都是调用的下面这个构造器进行的初始化工作的：

**public** ThreadPoolExecutor(

**int** corePoolSize,

**int** maximumPoolSize,

**long** keepAliveTime,

TimeUnit unit,

BlockingQueue<Runnable> workQueue,

ThreadFactory threadFactory,

RejectedExecutionHandler handler)

下面解释下一下构造器中各个参数的含义：

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

**maximumPoolSize：**线程池最大线程数，它表示在线程池中最多能创建多少个线程；

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

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

TimeUnit.DAYS; //天

TimeUnit.HOURS; //小时

TimeUnit.MINUTES; //分钟

TimeUnit.SECONDS; //秒

TimeUnit.MILLISECONDS; //毫秒

TimeUnit.MICROSECONDS; //微妙

TimeUnit.NANOSECONDS; //纳秒

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

ArrayBlockingQueue;

LinkedBlockingQueue;

SynchronousQueue;

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

**threadFactory：**线程工厂，主要用来创建线程；

**handler：**表示当拒绝处理任务时的策略，有以下四种取值：

ThreadPoolExecutor.AbortPolicy:丢弃任务并抛出RejectedExecutionException异常。

ThreadPoolExecutor.DiscardPolicy：也是丢弃任务，但是不抛出异常。

ThreadPoolExecutor.DiscardOldestPolicy：丢弃队列最前面的任务，然后重新尝试执行任务（重复此过程）

ThreadPoolExecutor.CallerRunsPolicy：由调用线程处理该任务

#### **一些重要的方法**

Summit是线程池中一个比较重要的方法，后面会对其进行介绍。

shutdown()和shutdownNow()是用来关闭线程池的。还有很多其他的方法：比如：getQueue() 、getPoolSize() 、getActiveCount()、getCompletedTaskCount()等获取与线程池相关属性的方法，有兴趣的朋友可以自行查阅API。

#### **\*Runnable和Callable的区别**

Runnable和Callable接口的源码如下：

**public interface** Runnable {**public abstract void** run();  
}

**public interface** Callable<V> {V call() **throws** Exception;  
}

(1)Callable规定的方法是call(),Runnable规定的方法是run()  
(2)Callable的任务执行后有返回值，而Runnable的任务是没有返回值得的  
(3)call() 可以抛出受检查的异常，而run()不能抛出受检查的异常，因此使用run()方法时，只能使用try、catch来捕获异常。

我们在调用submit方法向线程池提交任务的时候，通常要将任务封装为Runnable或Callable的对象，然后交由线程池去执行。那这里会有一个问题，什么时候使用Runnable，什么时候使用Callable呢？

我们知道使用线程池时需要使用submit方法来向线程池提交任务，其实，线程池最终是通过execute()方法来执行任务的，而execute()方法是在Executor接口中声明的，它的声明如下：

**void** execute(Runnable command);

可以看到该方法只接受Runnable的对象，那么当我们将任务封装为Callable对象时，线程池是如何将其转为Runnable的对象，然后调用execute()来执行任务的呢？这里我们看下submit()方法的源码：

**public** Future<?> submit(Runnable task) {  
 **if** (task == **null**) **throw new** NullPointerException();  
 RunnableFuture<Void> ftask = newTaskFor(task, **null**);  
 execute(ftask);  
 **return** ftask;  
}

**public** <T> Future<T> submit(Callable<T> task) {  
 **if** (task == **null**) **throw new** NullPointerException();  
 RunnableFuture<T> ftask = newTaskFor(task);  
 execute(ftask);  
 **return** ftask;  
}

从上面的代码我们发现，不管任务是封装为Callable还是Runnable的对象，最终都是先将task包装为一个RunnableFuture对象，然后再调用execute(Runnable command);方法。我们再来看看RunnableFuture的源码实现：

**public interface** RunnableFuture<V> **extends** Runnable, Future<V> {**void** run();  
}

RunnableFuture继承了Runnable和Future接口，所以它便同时拥有了这两个接口的特性。

现在，我们再回到刚刚那个问题，什么时候使用Runnable，什么时候使用Callable来封装线程池任务，当执行的任务不需要有返回值，我们可以使用Runnable来封装任务，但是如果任务执行完成后，需要返回对应的结果（比如使用线程池开启多个任务去查询一些服务接口）时，我们就使用Callable来封装任务。另外，如果，使用Runnable封装任务，然后在通过Future接口的get方法来获取返回值的话，那么返回值都为null，如：

Future<?> future = pool.submit(**new** RunnableTask(**"这是一个任务"**));  
**try** {  
 *// 如果是使用Runnable封装任务，那么这里获取的返回值都是null，并且future.get()方法会阻塞* System.***out***.println(future.get());  
} **catch** (Exception e) {e.printStackTrace();}

#### **ScheduledThreadPoolExecutor**

ScheduledThreadPoolExecutor继承了ThreadPoolExecutor，并实现了ScheduledExecutorService接口，它是周期性任务调度的类实现。

## Executors类

要配置一个线程池是比较复杂的，尤其是对于线程池的原理不是很清楚的情况下，很有可能配置的线程池不是较优的，因此在Executors类里面提供了一些静态工厂，生成一些常用的线程池。

* **newScheduledThreadPool：**创建一个大小无限的线程池。此线程池支持定时以及周期性执行任务的需求。
* **newSingleThreadExecutor：**创建一个单线程的线程池。这个线程池只有一个线程在工作，也就是相当于单线程串行执行所有任务。如果这个唯一的线程因为异常结束，那么会有一个新的线程来替代它。此线程池保证所有任务的执行顺序按照任务的提交顺序执行。
* **newFixedThreadPool：**创建固定大小的线程池。每次提交一个任务就创建一个线程，直到线程达到线程池的最大大小。线程池的大小一旦达到最大值就会保持不变，如果某个线程因为执行异常而结束，那么线程池会补充一个新线程。
* **newCachedThreadPool：**创建一个可缓存的线程池。如果线程池的大小超过了处理任务所需要的线程，那么就会回收部分空闲（60秒不执行任务）的线程，当任务数增加时，此线程池又可以智能的添加新线程来处理任务。此线程池不会对线程池大小做限制，线程池大小完全依赖于操作系统（或者说JVM）能够创建的最大线程大小。

## Future接口

Future 表示异步计算的结果，它提供了检查计算是否完成的方法，以等待计算的完成，并获取计算的结果。计算完成后只能使用 get 方法来获取结果，如有必要，计算完成前可以阻塞此方法。取消则由 cancel 方法来执行。还提供了其他方法，以确定任务是正常完成还是被取消了。一旦计算完成，就不能再取消计算。如果为了可取消性而使用 Future 但又不提供可用的结果，则可以声明 Future<?> 形式类型、并返回 null 作为底层任务的结果。Future就是对于具体的Runnable或者Callable任务的执行结果进行取消、查询是否完成、获取结果。必要时可以通过get方法获取执行结果，该方法会阻塞直到任务返回结果。也就是说Future提供了三种功能：

1. 判断任务是否完成
2. 能够中断任务
3. 能够获取任务执行结果

**public interface** Future<V> {  
 *//试图取消对此任务的执行* **boolean** cancel(**boolean** mayInterruptIfRunning);  
 *//如果在任务正常完成前将其取消，则返回 true* **boolean** isCancelled();  
 *//如果任务已完成，则返回 true* **boolean** isDone();  
 *//如有必要，等待计算完成，然后获取其结果* V get();  
 *//如有必要，最多等待为使计算完成所给定的时间之后，获取其结果（如果结果可用）* V get(**long** timeout, TimeUnit unit);  
}

#### **FutureTask**

FutureTask是Future接口的一个实现类，我们可以继承该接口来实现一些，额外的操作。例如，当我们调用submit方法，向线程池提交多个任务时，我们无法知道哪个任务是先被执行的，因为我们可以将任务进一步封装为FutureTask的子类，该类中有一个空实现方法done()，该方法就是为了就给子类实现的，该方法会在线程执行任务的调用，我们可以通过实现该方法将任务加到一个队列中，以此来判断任务执行的先后顺序，例如：

**private class** QueueingFutureTask **extends** FutureTask<V> {  
 QueueingFutureTask(Callable<V> task) {  
 **super**(task);  
 }  
  
 *//开始执行任务的时候调用该方法* @Override  
 **protected void** done() {  
 *//按任务的执行顺序将任务放到队列中* **completionQueue**.add(**this**);  
 }  
}

#### **Future接口的局限和CompletableFuture类**

我们知道 Future 接口提供了方法来检测异步计算是否已经结束（使用isDone 方法），等待异步操作结束，以及获取计算的结果。但是这些特性还不足以让你编写简洁的并发代码。比如，我们很难表述 Future 结果之间的依赖性；从文字描述上这很简单，“当长时间计算任务完成时，请将该计算的结果通知到另一个长时间运行的计算任务，这两个计算任务都完成后，将计算的结果与另一个查询操作结果合并”。但是，使用 Future 中提供的方法完成这样的操作又是另外一回事。这也是我们需要更具描述能力的特性的原因，比如下面这些。

* 将两个异步计算合并为一个——这两个异步计算之间相互独立，同时第二个又依赖于第一个的结果。
* 等待 Future 集合中的所有任务都完成。
* 仅等待 Future 集合中最快结束的任务完成（有可能因为它们试图通过不同的方式计算同一个值），并返回它的结果。
* 通过编程方式完成一个 Future 任务的执行（即以手工设定异步操作结果的方式）。
* 应对 Future 的完成事件（即当 Future 的完成事件发生时会收到通知，并能使用 Future计算的结果进行下一步的操作，不只是简单地阻塞等待操作的结果）。

CompletableFuture 类，实现了 Future 接口，它java8中为解决上面的问题而设计的，它利用Java 8的新特性以更直观的方式将上述需求都变为可能。 Stream 和 CompletableFuture 的设计都遵循了类似的模式：它们都使用了Lambda表达式以及流水线的思想。从这个角度，你可以说CompletableFuture 和 Future 的关系就跟 Stream 和 Collection 的关系一样。详细的介绍请参数《java8实战》一书。

## 深入剖析线程池实现原理

### 线程池状态

在ThreadPoolExecutor中定义了一个volatile变量，另外定义了几个static final变量表示线程池的各个状态，如下：

**volatile int runState**;  
**static final int *RUNNING*** = 0;  
**static final int *SHUTDOWN*** = 1;  
**static final int *STOP*** = 2;  
**static final int *TERMINATED*** = 3;

runState表示当前线程池的状态，它是一个volatile变量用来保证线程之间的可见性。下面的几个static final变量表示runState可能的几个取值。

当创建线程池后，初始时，线程池处于RUNNING状态；

如果调用了shutdown()方法，则线程池处于SHUTDOWN状态，此时线程池不能够接受新的任务，它会等待所有任务执行完毕；

如果调用了shutdownNow()方法，则线程池处于STOP状态，此时线程池不能接受新的任务，并且会去尝试终止正在执行的任务；

当线程池处于SHUTDOWN或STOP状态，并且所有工作线程已经销毁，任务缓存队列已经清空或执行结束后，线程池被设置为TERMINATED状态。

### 任务的执行

在了解将任务提交给线程池到任务执行完毕整个过程之前，我们先来看一下ThreadPoolExecutor类中其他的一些比较重要成员变量：

**private final** BlockingQueue<Runnable> **workQueue**; *//任务缓存队列，用来存放等待执行的任务***private final** ReentrantLock **mainLock** = **new** ReentrantLock(); *//线程池的主要状态锁，对线程池状态（比如线程池大小、runState等）的改变都要使用这个锁***private final** HashSet<Worker> **workers** = **new** HashSet<Worker>(); *//用来存放工作集***private volatile long keepAliveTime**; *//线程存货时间***private volatile boolean allowCoreThreadTimeOut**; *//是否允许为核心线程设置存活时间***private volatile int corePoolSize**; *//核心池的大小（即线程池中的线程数目大于这个参数时，提交的任务会被放进任务缓存队列）***private volatile int maximumPoolSize**; *//线程池最大能容忍的线程数***private volatile int poolSize**; *//线程池中当前的线程数***private volatile** RejectedExecutionHandler **handler**; *//任务拒绝策略***private volatile** ThreadFactory **threadFactory**; *//线程工厂，用来创建线程***private int largestPoolSize**; *//用来记录线程池中曾经出现过的最大线程数***private long completedTaskCount**; *//用来记录已经执行完毕的任务个数*

每个变量的作用都已经标明出来了，这里要重点解释一下corePoolSize、maximumPoolSize、largestPoolSize三个变量。corePoolSize在很多地方被翻译成核心池大小，其实我的理解这个就是线程池的大小。举个简单的例子：

假如有一个工厂，工厂里面有10个工人，每个工人同时只能做一件任务。因此只要当10个工人中有工人是空闲的，来了任务就分配给空闲的工人做；当10个工人都有任务在做时，如果还来了任务，就把任务进行排队等待；如果说新任务数目增长的速度远远大于工人做任务的速度，那么此时工厂主管可能会想补救措施，比如重新招4个临时工人进来；然后就将任务也分配给这4个临时工人做；如果说着14个工人做任务的速度还是不够，此时工厂主管可能就要考虑不再接收新的任务或者抛弃前面的一些任务了。当这14个工人当中有人空闲时，而新任务增长的速度又比较缓慢，工厂主管可能就考虑辞掉4个临时工了，只保持原来的10个工人，毕竟请额外的工人是要花钱的。

这个例子中的corePoolSize就是10，而maximumPoolSize就是14。也就是说corePoolSize就是线程池大小，maximumPoolSize在我看来是线程池的一种补救措施，即任务量突然过大时的一种补救措施。不过为了方便理解，在本文后面还是将corePoolSize翻译成核心池大小。largestPoolSize只是一个用来起记录作用的变量，用来记录线程池中曾经有过的最大线程数目，跟线程池的容量没有任何关系。

　　在ThreadPoolExecutor类中，最核心的任务提交方法是execute()方法，虽然通过submit也可以提交任务，但是实际上submit方法里面最终调用的还是execute()方法，所以我们只需要研究execute()方法的实现原理即可。另外在线程池实现中有一个非常巧妙的设计方式，假如我们来设计线程池，可能会有一个任务分派线程，当发现有线程空闲时，就从任务缓存队列中取一个任务交给空闲线程执行。但是在这里，并没有采用这样的方式，因为这样会要额外地对任务分派线程进行管理，无形地会增加难度和复杂度，这里直接让执行完任务的线程去任务缓存队列里面取任务来执行。

任务提交给线程池之后的处理策略主要有以下4点：

* 如果当前线程池中的线程数目<corePoolSize，则每来一个任务，就会创建一个线程去执行这个任务；
* 如果当前线程池中的线程数目>=corePoolSize，则每来一个任务，会尝试将其添加到任务缓存队列当中，若添加成功则该任务会等待空闲线程将其取出去执行 , 若添加失败（一般来说是任务缓存队列已满），则会尝试创建新的线程去执行这个任务；
* 如果当前线程池中的线程数目达到maximumPoolSize，则会采取任务拒绝策略进行处理；
* 如果线程池中的线程数量大于 corePoolSize时，如果某线程空闲时间超过keepAliveTime，线程将被终止，直至线程池中的线程数目不大于corePoolSize；如果允许为核心池中的线程设置存活时间，那么核心池中的线程空闲时间超过keepAliveTime，线程也会被终止。

### 线程池中的线程初始化

任务提交到线程池后，线程池首先执行的是通过构造器传进来的任务firstTask，在调用runTask()执行完firstTask之后，在while循环里面不断通过getTask()去任务缓存队列里面取新的任务来执行。默认情况下，创建线程池之后，线程池中是没有线程的，需要提交任务之后才会创建线程。在实际中如果需要线程池创建之后立即创建线程，可以通过以下两个方法办到：

* prestartCoreThread()：初始化一个核心线程
* prestartAllCoreThreads()：初始化所有核心线程

下面是这2个方法的实现：  
**public boolean** prestartCoreThread() {  
 **return** addIfUnderCorePoolSize(**null**); *//注意传进去的参数是null*}  
**public int** prestartAllCoreThreads() {  
 **int** n = 0;  
 **while** (addIfUnderCorePoolSize(**null**))*//注意传进去的参数是null* ++n;  
 **return** n;  
}

注意上面传进去的参数是null，根据第2小节的分析可知如果传进去的参数为null，则最后执行线程会阻塞在getTask方法中的r = workQueue.take();即等待任务队列中有任务。

### 任务缓存队列及排队策略

在前面我们多次提到了任务缓存队列，即workQueue，它用来存放等待执行的任务。workQueue的类型为BlockingQueue<Runnable>，通常可以取下面三种类型：

* ArrayBlockingQueue：基于数组的先进先出队列，此队列创建时必须指定大小；
* LinkedBlockingQueue：基于链表的先进先出队列，如果创建时没有指定此队列大小，则默认为Integer.MAX\_VALUE；
* synchronousQueue：这个队列比较特殊，它不会保存提交的任务，而是将直接新建一个线程来执行新来的任务。

### 任务拒绝策略

当线程池的任务缓存队列已满并且线程池中的线程数目达到maximumPoolSize，如果还有任务到来就会采取任务拒绝策略，通常有以下四种策略：

ThreadPoolExecutor.AbortPolicy:丢弃任务并抛出RejectedExecutionException异常。

ThreadPoolExecutor.DiscardPolicy：也是丢弃任务，但是不抛出异常。

ThreadPoolExecutor.DiscardOldestPolicy：丢弃队列最前面的任务，然后重新尝试执行任务（重复此过程）

ThreadPoolExecutor.CallerRunsPolicy：由调用线程处理该任务

### 线程池的关闭

ThreadPoolExecutor提供了两个方法，用于线程池的关闭，分别是shutdown()和shutdownNow()，其中：

shutdown()：不会立即终止线程池，而是要等所有任务缓存队列中的任务都执行完后才终止，但再也不会接受新的任务

shutdownNow()：立即终止线程池，并尝试打断正在执行的任务，并且清空任务缓存队列，返回尚未执行的任务

### 7.线程池容量的动态调整

ThreadPoolExecutor提供了动态调整线程池容量大小的方法：

* setCorePoolSize：设置核心池大小
* setMaximumPoolSize：设置线程池最大能创建的线程数目大小

当上述参数从小变大时，ThreadPoolExecutor进行线程赋值，还可能立即创建新的线程来执行任务。