

什么是线程池？

诸如 web 服务器、数据库服务器、文件服务器和邮件服务器等许多服务器应用都面向处理来自某些远程来源的大量短小的任务。构建服务器应用程序的一个过于简单的模型是：每当一个请求到达就创建一个新的服务对象，然后在新的服务对象中为请求服务。但当有大量请求并发访问时，服务器不断的创建和销毁对象的开销很大。所以提高服务器效率的一个手段就是尽可能减少创建和销毁对象的次数，特别是一些很耗资源的对象创建和销毁，这样就引入了“池”的概念，“池”的概念使得人们可以定制一定量的资源，然后对这些资源进行复用，而不是频繁的创建和销毁。

线程池是预先创建线程的一种技术。线程池在还没有任务到来之前，创建一定数量的线程，放入空闲队列中。这些线程都是处于睡眠状态，即均为启动，不消耗 CPU，而只是占用较小的内存空间。当请求到来之后，缓冲池给这次请求分配一个空闲线程，把请求传入此线程中运行，进行处理。当预先创建的线程都处于运行状态，即预制线程不够，线程池可以自由创建一定数量的新线程，用于处理更多的请求。当系统比较闲的时候，也可以通过移除一部分一直处于停用状态的线程。

线程池的注意事项

虽然线程池是构建多线程应用程序的强大机制，但使用它并不是没有风险的。在使用线程池时需注意线程池大小与性能的关系，注意并发风险、死锁、资源不足和线程泄漏等问题。

（1）线程池大小。多线程应用并非线程越多越好，需要根据系统运行的软硬件环境以及应用本身的特点决定线程池的大小。一般来说，如果代码结构合理的话，线程数目与 CPU 数量相适合即可。如果线程运行时可能出现阻塞现象，可相应增加池的大小；如有必要可采用自适应算法来动态调整线程池的大小，以提高 CPU 的有效利用率和系统的整体性能。

（2）并发错误。多线程应用要特别注意并发错误，要从逻辑上保证程序的正确性，注意避免死锁现象的发生。

（3）线程泄漏。这是线程池应用中一个严重的问题，当任务执行完毕而线程没能返回池中就会发生线程泄漏现象。

简单线程池的设计

一个典型的线程池，应该包括如下几个部分：

- 1、线程池管理器（ThreadPool），用于启动、停用，管理线程池
- 2、工作线程（WorkThread），线程池中的线程
- 3、请求接口（WorkRequest），创建请求对象，以供工作线程调度任务的执行
- 4、请求队列（RequestQueue），用于存放和提取请求
- 5、结果队列（ResultQueue），用于存储请求执行后返回的结果

线程池管理器 通过添加请求的方法(putRequest)向请求队列(RequestQueue)添加请求, 这些请求事先需要实现请求接口, 即传递工作函数、参数、结果处理函数、以及异常处理函数。之后初始化一定数量的工作线程, 这些线程通过轮询的方式不断查看请求队列(RequestQueue), 只要有请求存在, 则会提取出请求, 进行执行。然后, 线程池管理器调用方法(poll)查看结果队列(resultQueue)是否有值, 如果有值, 则取出, 调用结果处理函数执行。通过以上讲述, 不难发现, 这个系统的核心资源在于请求队列和结果队列, 工作线程通过轮询 requestQueue 获得人物, 主线程通过查看结果队列, 获得执行结果。因此, 对这个队列的设计, 要实现线程同步, 以及一定阻塞和超时机制的设计, 以防止因为不断轮询而导致的过多 cpu 开销。在本文中 将会用python语言实现 ,python 的 Queue , 就是很好的实现了对线程同步机制。

java 线程池与五种常用线程池策略使用与解析

一.线程池

关于为什么要使用线程池久不赘述了, 首先看一下 [Java](#) 中作为线程池 Executor 底层实现类的 ThredPoolExecutor 的构造函数

```
public ThreadPoolExecutor(int corePoolSize,  
                           int maximumPoolSize,
```

```
        long keepAliveTime,

        TimeUnit unit,

        BlockingQueue<Runnable> workQueue,

        ThreadFactory threadFactory,

        RejectedExecutionHandler handler) {

    ...

}
```

其中各个参数含义如下:

corePoolSize - 池中所保存的线程数, 包括空闲线程。需要注意的是在初创建线程池时线程不会立即启动, 直到有任务提交才开始启动线程并逐渐时线程数目达到 **corePoolSize**。

若想一开始就创建所有核心线程需调用 **prestartAllCoreThreads** 方法。

maximumPoolSize - 池中允许的最大线程数。需要注意的是当核心线程满且阻塞队列也满时才会判断当前线程数是否小于最大线程数, 并决定是否创建新线程。

keepAliveTime - 当线程数大于核心时, 多于的空闲线程最多存活时间

unit - **keepAliveTime** 参数的时间单位。

workQueue - 当线程数目超过核心线程数时用于保存任务的队列。主要有 3 种类型的 **BlockingQueue** 可供选择: 无界队列, 有界队列和同步移交。将在下文中详细阐述。从参数中可以看到, 此队列仅保存实现 **Runnable** 接口的任务。

threadFactory - 执行程序创建新线程时使用的工厂。

handler - 阻塞队列已满且线程数达到最大值时所采取的饱和策略。**java** 默认提供了 4 种饱和策略的实现方式: 中止、抛弃、抛弃最旧的、调用者运行。将在下文中详细阐述。

二.可选择的阻塞队列 BlockingQueue 详解

首先看一下新任务进入时线程池的执行策略:

如果运行的线程少于 `corePoolSize`, 则 `Executor` 始终首选添加新的线程, 而不进行排队。

(如果当前运行的线程小于 `corePoolSize`, 则任务根本不会存入 `queue` 中, 而是直接运行)

如果运行的线程大于等于 `corePoolSize`, 则 `Executor` 始终首选将请求加入队列, 而不添加新的线程。

如果无法将请求加入队列, 则创建新的线程, 除非创建此线程超出 `maximumPoolSize`,

在这种情况下, 任务将被拒绝。

主要有 3 种类型的 `BlockingQueue`:

2.1 无界队列

队列大小无限制, 常用的为无界的 `LinkedBlockingQueue`, 使用该队列做为阻塞队列时要尤其当心, 当任务耗时较长时可能会导致大量新任务在队列中堆积最终导致 `OOM`。最近工作中就遇到因为采用 `LinkedBlockingQueue` 作为阻塞队列, 部分任务耗时 `80s+` 且不停有新任务进来, 导致 `cpu` 和内存飙升服务器挂掉。

2.2 有界队列

常用的有两类, 一类是遵循 `FIFO` 原则的队列如 `ArrayBlockingQueue` 与有界的

`LinkedBlockingQueue`, 另一类是优先级队列如 `PriorityBlockingQueue`。

`PriorityBlockingQueue` 中的优先级由任务的 `Comparator` 决定。

使用有界队列时队列大小需和线程池大小互相配合, 线程池较小有界队列较大时可减少内存消耗, 降低 `cpu` 使用率和上下文切换, 但是可能会限制系统吞吐量。

2.3 同步移交

如果不希望任务在队列中等待而是希望将任务直接移交给工作线程, 可使用 `SynchronousQueue` 作为等待队列。`SynchronousQueue` 不是一个真正的队列, 而是一种线程之间移交的机制。要将一个元素放入 `SynchronousQueue` 中, 必须有另一个线程正在等待接收这个元素。只有在使用无界线程池或者有饱和策略时才建议使用该队列。

2.4 几种 `BlockingQueue` 的具体实现原理

关于上述几种 `BlockingQueue` 的具体实现原理与分析将在下篇博文中详细阐述。

三. 可选的饱和策略 `RejectedExecutionHandler` 详解

JDK 主要提供了 4 种饱和策略供选择。4 种策略都做为静态内部类在 `ThreadPoolExecutor` 中进行实现。

3.1 `AbortPolicy` 中止策略

该策略是默认饱和策略。

```
public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {  
  
    throw new RejectedExecutionException("Task " + r.  
toString() +  
  
                                " rejected from "  
  
                                +  
                                e.toString());  
  
}
```

使用该策略时在饱和时会抛出 `RejectedExecutionException`(继承自 `RuntimeException`), 调用者可捕获该异常自行处理。

3.2 DiscardPolicy 抛弃策略

```
public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {  
  
    }  
}
```

如代码所示, 不做任何处理直接抛弃任务

3.3 DiscardOldestPolicy 抛弃旧任务策略

```
public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {  
  
    if (!e.isShutdown()) {  
  
        e.getQueue().poll();  
  
        e.execute(r);  
  
    }  
  
}
```

如代码, 先将阻塞队列中的头元素出队抛弃, 再尝试提交任务。如果此时阻塞队列使用 **PriorityBlockingQueue** 优先级队列, 将会导致优先级最高的任务被抛弃, 因此不建议将该种策略配合优先级队列使用。

3.4 CallerRunsPolicy 调用者运行

```
public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {  
  
    if (!e.isShutdown()) {  
  
        r.run();  
  
    }  
  
}
```

既不抛弃任务也不抛出异常，直接运行任务的 `run` 方法，换言之将任务回退给调用者来直接运行。使用该策略时线程池饱和后将由调用线程池的主线程自己来执行任务，因此在执行任务的这段时间里主线程无法再提交新任务，从而使线程池中工作线程有时间将正在处理的任务处理完成。

四.java 提供的四种常用线程池解析

在 JDK 帮助文档中，有如此一段话：

强烈建议程序员使用较为方便的 `Executors` 工厂方法 `Executors.newCachedThreadPool()`

（无界线程池，可以进行自动线程回收）、`Executors.newFixedThreadPool(int)`（固定大小线程池）

`newScheduledThreadPool` 创建一个定长线程池，支持定时及周期性任务执行。

`Executors.newSingleThreadExecutor()`（单个后台线程）它们均为大多数使用场景预定义了设置。

详细介绍一下上述四种线程池。

4.1 newCachedThreadPool

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

在 `newCachedThreadPool` 中如果线程池长度超过处理需要，可灵活回收空闲线程，若无可回收，则新建线程。

初看该构造函数时我有这样的疑惑：核心线程池为 0，那按照前面所讲的线程池策略新任务来临时无法进入核心线程池，只能进入 `SynchronousQueue` 中进行等待，而

`SynchronousQueue` 的大小为 1，那岂不是第一个任务到达时只能等待在队列中，直到第二个任务到达发现无法进入队列才能创建第一个线程？

这个问题的答案在上面讲 `SynchronousQueue` 时其实已经给出了，要将一个元素放入

`SynchronousQueue` 中，必须有另一个线程正在等待接收这个元素。因此即便

`SynchronousQueue` 一开始为空且大小为 1，第一个任务也无法放入其中，因为没有线程在等待从 `SynchronousQueue` 中取走元素。因此第一个任务到达时便会创建一个新线程执行该任务。

这里引申出一个小技巧：有时我们可能希望线程池在没有任务的情况下销毁所有的线程，既设置线程池核心大小为 0，但又不想使用 `SynchronousQueue` 而是想使用有界的等待队列。

显然，不进行任何特殊设置的话这样的用法会发生奇怪的行为：直到等待队列被填满才会有新线程被创建，任务才开始执行。这并不是我们希望看到的，此时可通过

`allowCoreThreadTimeOut` 使等待队列中的元素出队被调用执行，详细原理和使用将会在后续博客中阐述。

4.2 `newFixedThreadPool` 创建一个定长线程池，可控制线程最大并发数，超出的线程会在队列中等待。

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

看代码一目了然了，使用固定大小的线程池并使用无限大的队列

4.3 newScheduledThreadPool 创建一个定长线程池，支持定时及周期性任务执行。

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

在来看看 `ScheduledThreadPoolExecutor` () 的构造函数

```
public ScheduledThreadPoolExecutor(int corePoolSize) {  
    super(corePoolSize, Integer.MAX_VALUE, 0, NANOSCONDS,  
        new DelayedWorkQueue());  
}
```

`ScheduledThreadPoolExecutor` 的父类即 `ThreadPoolExecutor`，因此这里各参数含义和上面一样。值得关心的是 `DelayedWorkQueue` 这个阻塞队列，在上面没有介绍，它作为静态内部类就在 `ScheduledThreadPoolExecutor` 中进行了实现。具体分析讲会在后续博客中给出，在这里只进行简单说明：`DelayedWorkQueue` 是一个无界队列，它能按一定的顺序对工作队列中的元素进行排列。因此这里设置的最大线程数 `Integer.MAX_VALUE` 没有任何意义。关于 `ScheduledThreadPoolExecutor` 的具体使用将会在后续 quartz 的周期性任务实现原理中进行进一步分析。

4.4 newSingleThreadExecutor 创建一个单线程化的线程池，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序(FIFO, LIFO, 优先级)执行。

```
public static ScheduledExecutorService newSingleThreadScheduledExecutor() {  
  
    return new DelegatedScheduledExecutorService  
        (new ScheduledThreadPoolExecutor(1));  
  
}
```

首先 new 了一个线程数目为 1 的 ScheduledThreadPoolExecutor，再把该对象传入

DelegatedScheduledExecutorService 中，看看 DelegatedScheduledExecutorService 的实现代码：

```
DelegatedScheduledExecutorService(ScheduledExecutorService executor) {  
  
    super(executor);  
  
    e = executor;  
  
}
```

在看看它的父类

```
DelegatedExecutorService(ExecutorService executor) { e =  
    executor; }
```

其实就是使用装饰模式增强了 ScheduledExecutorService (1) 的功能，不仅确保只有一个线程顺序执行任务，也保证线程意外终止后会重新创建一个线程继续执行任务。具体实现原理会在后续博客中讲解。

4.5 newWorkStealingPool 创建一个拥有多个任务队列（以便减少连接数）的线程池。

这是 jdk1.8 中新增加的一种线程池实现，先看一下它的无参实现

```
public static ExecutorService newWorkStealingPool() {  
    return new ForkJoinPool  
        (Runtime.getRuntime().availableProcessors(),  
         ForkJoinPool.defaultForkJoinWorkerThreadFactor  
Y,  
         null, true);  
}
```

返回的 `ForkJoinPool` 从 jdk1.7 开始引进，个人感觉类似于 `mapreduce` 的思想。这个线程池较为特殊，将在后续博客中给出详细的使用说明和原理。

在前面的文章中，我们使用线程的时候就去创建一个线程，这样实现起来非常简便，但是就会有一个问题：

如果并发的线程数量很多，并且每个线程都是执行一个时间很短的任务就结束了，这样频繁创建线程就会大大降低系统的效率，因为频繁创建线程和销毁线程需要时间。

那么有没有一种办法使得线程可以复用，就是执行完一个任务，并不被销毁，而是可以继续执行其他的任务？

在 **Java** 中可以通过线程池来达到这样的效果。今天我们就来详细讲解一下 **Java** 的线程池，首先我们从最核心的 `ThreadPoolExecutor` 类中的方法讲起，然后再讲述它的实现原理，接着给出了它的使用示例，最后讨论了一下如何合理配置线程池的大小。

以下是本文的目录大纲：

一.**Java** 中的 `ThreadPoolExecutor` 类

二.深入剖析线程池实现原理

三.使用示例

四.如何合理配置线程池的大小

若有不正之处请多多谅解，并欢迎批评指正。

请尊重作者劳动成果，转载请标明原文链接：

<http://www.cnblogs.com/dolphin0520/p/3932921.html>

一. Java 中的 **ThreadPoolExecutor** 类

`java.util.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。

在 `ThreadPoolExecutor` 类中提供了四个构造方法：

```
public class ThreadPoolExecutor extends AbstractExecutorService {
    .....
    public ThreadPoolExecutor(int corePoolSize, int
maximumPoolSize, long keepAliveTime, TimeUnit unit,
        BlockingQueue<Runnable> workQueue);

    public ThreadPoolExecutor(int corePoolSize, int
maximumPoolSize, long keepAliveTime, TimeUnit unit,
        BlockingQueue<Runnable> workQueue, ThreadFactory
threadFactory);

    public ThreadPoolExecutor(int corePoolSize, int
maximumPoolSize, long keepAliveTime, TimeUnit unit,
        BlockingQueue<Runnable>
workQueue, RejectedExecutionHandler handler);

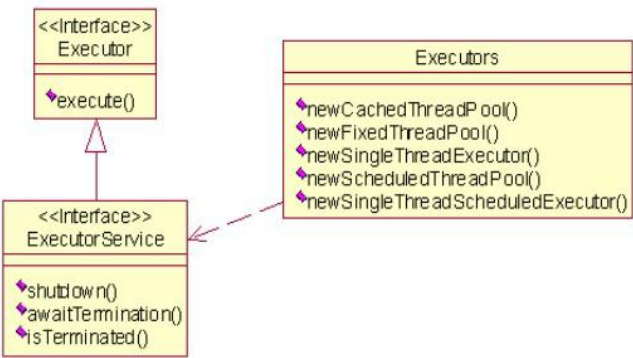
    public ThreadPoolExecutor(int corePoolSize, int
maximumPoolSize, long keepAliveTime, TimeUnit unit,
        BlockingQueue<Runnable> workQueue, ThreadFactory
threadFactory, RejectedExecutionHandler handler);
    ...
}
```

从上面的代码可以得知，`ThreadPoolExecutor` 继承了 `AbstractExecutorService` 类，并提供了四个构造器，事实上，通过观察每个构造器的源码具体实现，发现前面三个构造器都是调用的第四个构造器进行的初始化工作。

Java 类库中提供的线程池简介：

java 提供的线程池更加强大，相信理解线程池的工作原理，看类库中的线程池就不会感到陌生了。

- java.util.concurrent包提供了现成的线程池的实现。



JDK 类库中的线程池的类框图

Executor接口表示线程池，它的execute(Runnable task)方法用来执行Runnable类型的任务。Executor的子接口

ExecutorService中声明了管理线程池的一些方法，比如用于关闭线程池的shutdown()方法等。

Executors类中包含一些静态方法，它们负责生成各种类型的线程池ExecutorService实例。

Executors 类的生成 ExecutorService 实例的静态方法

Executors 类的静态方法	创建的 ExecutorService 线程池的类型
newCachedThreadPool()	在有任务时才创建新线程，空闲线程被保留 60 秒。
newFixedThreadPool(int nThreads)	线程池中包含固定数目的线程，空闲线程会一直保留。参数 nThreads 设定线程池中线程的数目。
newSingleThreadExecutor()	线程池中只有一个工作线程，它依次执行每个任务。
newScheduledThreadPool(int corePoolSize)	线程池能按时间计划来执行任务，允许用户设定计划执行任务的时间。参数 corePoolSize 设定线程池中线程的最小数目。当任务较多，线程池可能会创建更多的工作线程来执行任务。
newSingleThreadScheduledExecutor()	线程池中只有一个工作线程，它能按时间计划来执行任务。

下面解释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；

Java 架构学习群: 895244712

- **maximumPoolSize**: 线程池最大线程数, 这个参数也是一个非常重要的参数, 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
```

具体参数的配置与线程池的关系将在下一节讲述。



Java 架构学习群: 895244712

从上面给出的 `ThreadPoolExecutor` 类的代码可以知道, `ThreadPoolExecutor` 继承了 `AbstractExecutorService`, 我们来看一下 `AbstractExecutorService` 的实现:

```
public abstract class AbstractExecutorService implements
ExecutorService {

    protected <T> RunnableFuture<T> newTaskFor(Runnable runnable, T
value) { };
    protected <T> RunnableFuture<T> newTaskFor(Callable<T> callable)
{ };
    public Future<?> submit(Runnable task) { };
    public <T> Future<T> submit(Runnable task, T result) { };
    public <T> Future<T> submit(Callable<T> task) { };
    private <T> T doInvokeAny(Collection<? extends Callable<T>> tasks,
                                boolean timed, long nanos)
        throws InterruptedException, ExecutionException,
TimeoutException {
    };
    public <T> T invokeAny(Collection<? extends Callable<T>> tasks)
        throws InterruptedException, ExecutionException {
    };
    public <T> T invokeAny(Collection<? extends Callable<T>> tasks,
                            long timeout, TimeUnit unit)
        throws InterruptedException, ExecutionException,
TimeoutException {
    };
    public <T> List<Future<T>> invokeAll(Collection<? extends
Callable<T>> tasks)
        throws InterruptedException {
    };
    public <T> List<Future<T>> invokeAll(Collection<? extends
Callable<T>> tasks,
                                        long timeout, TimeUnit unit)
        throws InterruptedException {
    };
}
```

`AbstractExecutorService` 是一个抽象类, 它实现了 `ExecutorService` 接口。

Java 架构学习群: 895244712

我们接着看 `ExecutorService` 接口的实现:

```
public interface ExecutorService extends Executor {

    void shutdown();
    boolean isShutdown();
    boolean isTerminated();
    boolean awaitTermination(long timeout, TimeUnit unit)
        throws InterruptedException;
    <T> Future<T> submit(Callable<T> task);
    <T> Future<T> submit(Runnable task, T result);
    Future<?> submit(Runnable task);
    <T> List<Future<T>> invokeAll(Collection<? extends Callable<T>>
tasks)
        throws InterruptedException;
    <T> List<Future<T>> invokeAll(Collection<? extends Callable<T>>
tasks,
                                long timeout, TimeUnit unit)
        throws InterruptedException;

    <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;
}
```

而 `ExecutorService` 又是继承了 `Executor` 接口,我们看一下 `Executor` 接口的实现:

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

到这里,大家应该明白了 `ThreadPoolExecutor`、`AbstractExecutorService`、`ExecutorService` 和 `Executor` 几个之间的关系了。

`Executor` 是一个顶层接口，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；

然后 `ExecutorService` 接口继承了 `Executor` 接口，并声明了一些方法：`submit`、`invokeAll`、`invokeAny` 以及 `shutdown` 等；

抽象类 `AbstractExecutorService` 实现了 `ExecutorService` 接口，基本实现了 `ExecutorService` 中声明的所有方法；

然后 `ThreadPoolExecutor` 继承了类 `AbstractExecutorService`。

在 `ThreadPoolExecutor` 类中有几个非常重要的方法：

```
execute()  
submit()  
shutdown()  
shutdownNow()
```

`execute()`方法实际上是 `Executor` 中声明的方法，在 `ThreadPoolExecutor` 进行了具体的实现，这个方法是 `ThreadPoolExecutor` 的核心方法，通过这个方法可以向线程池提交一个任务，交由线程池去执行。

`submit()`方法是在 `ExecutorService` 中声明的方法，在 `AbstractExecutorService` 就已经有了具体的实现，在 `ThreadPoolExecutor` 中并没有对其进行重写，这个方法也是用来向线程池提交任务的，但是它和 `execute()`方法不同，它能够返回任务执行的结果，去看 `submit()`方法的实现，会发现它实际上还是调用的 `execute()`方法，只不过它利用了 `Future` 来获取任务执行结果（`Future` 相关内容将在下一篇讲述）。

`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。

二.深入剖析线程池实现原理

在上一节我们从宏观上介绍了 `ThreadPoolExecutor`，下面我们来深入解析一下线程池的具体实现原理，将从下面几个方面讲解：

1.线程池状态

2.任务的执行

3.线程池中的线程初始化

4.任务缓存队列及排队策略

5.任务拒绝策略

6.线程池的关闭

7.线程池容量的动态调整

1.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:

```
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。

2.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;    //线程存活时间
```

```
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**（**10+4**）。

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。

Java 架构学习群: 895244712

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。

下面我们进入正题, 看一下任务从提交到最终执行完毕经历了哪些过程。

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:

```
public void execute(Runnable command) {
    if (command == null)
        throw new NullPointerException();
    if (poolSize >= corePoolSize || !addIfUnderCorePoolSize(command)) {
        if (runState == RUNNING && workQueue.offer(command)) {
            if (runState != RUNNING || poolSize == 0)
                ensureQueuedTaskHandled(command);
        }
        else if (!addIfUnderMaximumPoolSize(command))
            reject(command); // is shutdown or saturated
    }
}
```

上面的代码可能看起来不是那么容易理解, 下面我们一句一句解释:

首先, 判断提交的任务 `command` 是否为 `null`, 若是 `null`, 则抛出空指针异常;

接着是这句, 这句要好好理解一下:

```
if (poolSize >= corePoolSize || !addIfUnderCorePoolSize(command))
```

由于是或条件运算符, 所以先计算前半部分的值, 如果线程池中当前线程数不小于核心池大小, 那么就会直接进入下面的 `if` 语句块了。

如果线程池中当前线程数小于核心池大小, 则接着执行后半部分, 也就是执行:

```
addIfUnderCorePoolSize(command)
```

如果执行完 `addIfUnderCorePoolSize` 这个方法返回 `false`, 则继续执行下面的 `if` 语句块, 否则整个方法就直接执行完毕了。

Java 架构学习群: 895244712

如果执行完 `addIfUnderCorePoolSize` 这个方法返回 `false`, 然后接着判断:

```
if (runState == RUNNING && workQueue.offer(command))
```

如果当前线程池处于 `RUNNING` 状态, 则将任务放入任务缓存队列; 如果当前线程池不处于 `RUNNING` 状态或者任务放入缓存队列失败, 则执行:

```
addIfUnderMaximumPoolSize(command)
```

如果执行 `addIfUnderMaximumPoolSize` 方法失败, 则执行 `reject()`方法进行任务拒绝处理。

回到前面:

```
if (runState == RUNNING && workQueue.offer(command))
```

这句的执行, 如果说当前线程池处于 `RUNNING` 状态且将任务放入任务缓存队列成功, 则继续进行判断:


```
if (runState != RUNNING || poolSize == 0)
```

这句判断是为了防止在将此任务添加进任务缓存队列的同时其他线程突然调用 `shutdown` 或者 `shutdownNow` 方法关闭了线程池的一种应急措施。如果是这样就执行:

```
ensureQueuedTaskHandled(command)
```

进行应急处理, 从名字可以看出是保证 添加到任务缓存队列中的任务得到处理。

我们接着看 2 个关键方法的实现: `addIfUnderCorePoolSize` 和 `addIfUnderMaximumPoolSize`:

```
  
  
private boolean addIfUnderCorePoolSize(Runnable firstTask) {  
    Thread t = null;  
    final ReentrantLock mainLock = this.mainLock;  
    mainLock.lock();  
    try {  
        if (poolSize < corePoolSize && runState == RUNNING)  
            t = addThread(firstTask);           //创建线程去执行  
firstTask 任务  
    } finally {  
        mainLock.unlock();  
    }  
}
```

```
    if (t == null)
        return false;
    t.start();
    return true;
}
```



这个是 `addIfUnderCorePoolSize` 方法的具体实现,从名字可以看出它的意图就是当低于核心池大小时执行的方法。下面看其具体实现,首先获取到锁,因为这地方涉及到线程池状态的变化,先通过 `if` 语句判断当前线程池中的线程数目是否小于核心池大小,有朋友也许会有疑问:前面在 `execute()` 方法中不是已经判断过了吗,只有线程池当前线程数目小于核心池大小才会执行 `addIfUnderCorePoolSize` 方法的,为何这地方还要继续判断?原因很简单,前面的判断过程中并没有加锁,因此可能在 `execute` 方法判断的时候 `poolSize` 小于 `corePoolSize`,而判断完之后,在其他线程中又向线程池提交了任务,就可能导致 `poolSize` 不小于 `corePoolSize` 了,所以需要在这个地方继续判断。然后接着判断线程池的状态是否为 `RUNNING`,原因也很简单,因为有可能在其他线程中调用了 `shutdown` 或者 `shutdownNow` 方法。然后就是执行

```
t = addThread(firstTask);
```

这个方法也非常关键,传进去的参数为提交的任务,返回值为 `Thread` 类型。然后接着在下面判断 `t` 是否为空,为空则表明创建线程失败(即 `poolSize >= corePoolSize` 或者 `runState` 不等于 `RUNNING`),否则调用 `t.start()` 方法启动线程。



我们来看一下 `addThread` 方法的实现:

```
private Thread addThread(Runnable firstTask) {
    Worker w = new Worker(firstTask);
    Thread t = threadFactory.newThread(w); //创建一个线程,执行任务
    if (t != null) {
        w.thread = t; //将创建的线程的引用赋值为 w 的成员变量
        workers.add(w);
        int nt = ++poolSize; //当前线程数加 1
        if (nt > largestPoolSize)
            largestPoolSize = nt;
    }
    return t;
}
```



在 `addThread` 方法中, 首先用提交的任务创建了一个 `Worker` 对象, 然后调用线程工厂 `threadFactory` 创建了一个新的线程 `t`, 然后将线程 `t` 的引用赋值给了 `Worker` 对象的成员变量 `thread`, 接着通过 `workers.add(w)` 将 `Worker` 对象添加到工作集当中。

下面我们看一下 `Worker` 类的实现:

```
private final class Worker implements Runnable {
    private final ReentrantLock runLock = new ReentrantLock();
    private Runnable firstTask;
    volatile long completedTasks;
    Thread thread;
    Worker(Runnable firstTask) {
        this.firstTask = firstTask;
    }
    boolean isActive() {
        return runLock.isLocked();
    }
    void interruptIfIdle() {
        final ReentrantLock runLock = this.runLock;
        if (runLock.tryLock()) {
            try {
                if (thread != Thread.currentThread())
                    thread.interrupt();
            } finally {
                runLock.unlock();
            }
        }
    }
    void interruptNow() {
        thread.interrupt();
    }

    private void runTask(Runnable task) {
        final ReentrantLock runLock = this.runLock;
        runLock.lock();
        try {
            if (runState < STOP &&
                Thread.interrupted() &&
                runState >= STOP)
                boolean ran = false;
            beforeExecute(thread, task); //beforeExecute 方法是
ThreadPoolExecutor 类的一个方法, 没有具体实现, 用户可以根据
```


//自己需要重载这个方法和后面的 afterExecute 方法来进行一些统计信息, 比如某个任务的执行时间等

```
        try {
            task.run();
            ran = true;
            afterExecute(task, null);
            ++completedTasks;
        } catch (RuntimeException ex) {
            if (!ran)
                afterExecute(task, ex);
            throw ex;
        }
    } finally {
        runLock.unlock();
    }
}

public void run() {
    try {
        Runnable task = firstTask;
        firstTask = null;
        while (task != null || (task = getTask()) != null) {
            runTask(task);
            task = null;
        }
    } finally {
        workerDone(this);    //当任务队列中没有任务时, 进行清理工作
    }
}
}
```



它实际上实现了 **Runnable** 接口, 因此上面的 **Thread t = threadFactory.newThread(w);**效果跟下面这句的效果基本一样:

```
Thread t = new Thread(w);
```

相当于传进去了一个 **Runnable** 任务, 在线程 **t** 中执行这个 **Runnable**。

既然 **Worker** 实现了 **Runnable** 接口, 那么自然最核心的方法便是 **run()**方法了:



```
public void run() {
    try {
        Runnable task = firstTask;
        firstTask = null;
        while (task != null || (task = getTask()) != null) {
            runTask(task);
            task = null;
        }
    } finally {
        workerDone(this);
    }
}
```



从 `run` 方法的实现可以看出，它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()` 去取新的任务来执行，那么去哪里取呢？自然是从任务缓存队列里面去取，`getTask` 是 `ThreadPoolExecutor` 类中的方法，并不是 `Worker` 类中的方法，下面是 `getTask` 方法的实现：

```
Runnable getTask() {
    for (;;) {
        try {
            int state = runState;
            if (state > SHUTDOWN)
                return null;
            Runnable r;
            if (state == SHUTDOWN) // Help drain queue
                r = workQueue.poll();
            else if (poolSize > corePoolSize || allowCoreThreadTimeOut)
                //如果线程数大于核心池大小或者允许为核心池线程设置空闲时间，
                //则通过 poll 取任务，若等待一定的时间取不到任务，则返回 null
                r = workQueue.poll(keepAliveTime,
                    TimeUnit.NANOSECONDS);
            else
                r = workQueue.take();
            if (r != null)
                return r;
            if (workerCanExit()) { //如果没取到任务，即 r 为 null，
                //则判断当前的 worker 是否可以退出
            }
        }
    }
}
```

```
        if (runState >= SHUTDOWN) // Wake up others
            interruptIdleWorkers(); //中断处于空闲状态的
worker
        return null;
    }
    // Else retry
} catch (InterruptedException ie) {
    // On interruption, re-check runState
}
}
}
```

在 `getTask` 中, 先判断当前线程池状态, 如果 `runState` 大于 `SHUTDOWN` (即为 `STOP` 或者 `TERMINATED`), 则直接返回 `null`。

如果 `runState` 为 `SHUTDOWN` 或者 `RUNNING`, 则从任务缓存队列取任务。

如果当前线程池的线程数大于核心池大小 `corePoolSize` 或者允许为核心池中的线程设置空闲存活时间, 则调用 `poll(time,timeUnit)`来取任务, 这个方法会等待一定的时间, 如果取不到任务就返回 `null`。

然后判断取到的任务 `r` 是否为 `null`, 为 `null` 则通过调用 `workerCanExit()`方法来判断当前 `worker` 是否可以退出, 我们看一下 `workerCanExit()`的实现:

```
private boolean workerCanExit() {
    final ReentrantLock mainLock = this.mainLock;
    mainLock.lock();
    boolean canExit;
    //如果 runState 大于等于 STOP, 或者任务缓存队列为空了
    //或者 允许为核心池线程设置空闲存活时间并且线程池中的线程数目大
    于 1
    try {
        canExit = runState >= STOP ||
            workQueue.isEmpty() ||
            (allowCoreThreadTimeOut &&
                poolSize > Math.max(1, corePoolSize));
    } finally {
        mainLock.unlock();
    }
    return canExit;
}
```



也就是说如果线程池处于 **STOP** 状态、或者任务队列已为空或者允许为核心池线程设置空闲存活时间并且线程数大于 1 时, 允许 **worker** 退出。如果允许 **worker** 退出, 则调用 **interruptIdleWorkers()**中断处于空闲状态的 **worker**, 我们看一下 **interruptIdleWorkers()**的实现:



```
void interruptIdleWorkers() {  
    final ReentrantLock mainLock = this.mainLock;  
    mainLock.lock();  
    try {  
        for (Worker w : workers) //实际上调用的是 worker 的  
interruptIfIdle() 方法  
            w.interruptIfIdle();  
    } finally {  
        mainLock.unlock();  
    }  
}
```



从实现可以看出, 它实际上调用的是 **worker** 的 **interruptIfIdle()**方法, 在 **worker** 的 **interruptIfIdle()**方法中:



```
void interruptIfIdle() {  
    final ReentrantLock runLock = this.runLock;  
    if (runLock.tryLock()) { //注意这里, 是调用 tryLock() 来获取锁的,  
因为如果当前 worker 正在执行任务, 锁已经被获取了, 是无法获取到锁的  
//如果成功获取了锁, 说明当前 worker 处  
于空闲状态  
        try {  
            if (thread != Thread.currentThread())  
                thread.interrupt();  
        } finally {  
            runLock.unlock();  
        }  
    }  
}
```



这里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。

我们再看 `addIfUnderMaximumPoolSize` 方法的实现,这个方法的实现思想和 `addIfUnderCorePoolSize` 方法的实现思想非常相似,唯一的区别在于 `addIfUnderMaximumPoolSize` 方法是在线程池中的线程数达到了核心池大小并且往任务队列中添加任务失败的情况下执行的:



```
private boolean addIfUnderMaximumPoolSize(Runnable firstTask) {
    Thread t = null;
    final ReentrantLock mainLock = this.mainLock;
    mainLock.lock();
    try {
        if (poolSize < maximumPoolSize && runState == RUNNING)
            t = addThread(firstTask);
    } finally {
        mainLock.unlock();
    }
    if (t == null)
        return false;
    t.start();
    return true;
}
```



看到没有,其实它和 `addIfUnderCorePoolSize` 方法的实现基本一模一样,只是 `if` 语句判断条件中的 `poolSize < maximumPoolSize` 不同而已。

到这里,大部分朋友应该对任务提交给线程池之后到被执行的整个过程有了一个基本的了解,下面总结一下:

- 1) 首先,要清楚 `corePoolSize` 和 `maximumPoolSize` 的含义;
- 2) 其次,要知道 `Worker` 是用来起到什么作用的;
- 3) 要知道任务提交给线程池之后的处理策略,这里总结一下主要有 4 点:

- 如果当前线程池中的线程数目小于 `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 $\geq \text{corePoolSize}$ ，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。

3. 线程池中的线程初始化

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。

4. 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：

- 1) `ArrayBlockingQueue`：基于数组的先进先出队列，此队列创建时必须指定大小；
- 2) `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`；
- 3) `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。

5.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

6.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()` 和 `setMaximumPoolSize()`，

- `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。

三.使用示例

前面我们讨论了关于线程池的实现原理，这一节我们来看一下它的具体使用：



```
public class Test {
    public static void main(String[] args) {
        ThreadPoolExecutor executor = new ThreadPoolExecutor(5, 10, 200,
            TimeUnit.MILLISECONDS,
            new ArrayBlockingQueue<Runnable>(5));

        for(int i=0;i<15;i++){
            MyTask myTask = new MyTask(i);
            executor.execute(myTask);
            System.out.println("线程池中线程数目:
"+executor.getPoolSize()+"，队列中等待执行的任务数目: "+
            executor.getQueue().size()+"，已执行玩别的任务数目:
"+executor.getCompletedTaskCount());
        }
        executor.shutdown();
    }
}

class MyTask implements Runnable {
    private int taskNum;

    public MyTask(int num) {
        this.taskNum = num;
    }

    @Override
    public void run() {
        System.out.println("正在执行 task "+taskNum);
        try {
            Thread.currentThread().sleep(4000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("task "+taskNum+"执行完毕");
    }
}
```



执行结果:



正在执行 task 0
线程池中线程数目: 1, 队列中等待执行的任务数目: 0, 已执行玩别的任务数目: 0
线程池中线程数目: 2, 队列中等待执行的任务数目: 0, 已执行玩别的任务数目: 0
正在执行 task 1
线程池中线程数目: 3, 队列中等待执行的任务数目: 0, 已执行玩别的任务数目: 0
正在执行 task 2
线程池中线程数目: 4, 队列中等待执行的任务数目: 0, 已执行玩别的任务数目: 0
正在执行 task 3
线程池中线程数目: 5, 队列中等待执行的任务数目: 0, 已执行玩别的任务数目: 0
正在执行 task 4
线程池中线程数目: 5, 队列中等待执行的任务数目: 1, 已执行玩别的任务数目: 0
线程池中线程数目: 5, 队列中等待执行的任务数目: 2, 已执行玩别的任务数目: 0
线程池中线程数目: 5, 队列中等待执行的任务数目: 3, 已执行玩别的任务数目: 0
线程池中线程数目: 5, 队列中等待执行的任务数目: 4, 已执行玩别的任务数目: 0
线程池中线程数目: 5, 队列中等待执行的任务数目: 5, 已执行玩别的任务数目: 0
线程池中线程数目: 6, 队列中等待执行的任务数目: 5, 已执行玩别的任务数目: 0
正在执行 task 10
线程池中线程数目: 7, 队列中等待执行的任务数目: 5, 已执行玩别的任务数目: 0
正在执行 task 11
线程池中线程数目: 8, 队列中等待执行的任务数目: 5, 已执行玩别的任务数目: 0
正在执行 task 12
线程池中线程数目: 9, 队列中等待执行的任务数目: 5, 已执行玩别的任务数目: 0
正在执行 task 13
线程池中线程数目: 10, 队列中等待执行的任务数目: 5, 已执行玩别的任务数目: 0
正在执行 task 14
task 3 执行完毕
task 0 执行完毕

```
task 2 执行完毕
task 1 执行完毕
正在执行 task 8
正在执行 task 7
正在执行 task 6
正在执行 task 5
task 4 执行完毕
task 10 执行完毕
task 11 执行完毕
task 13 执行完毕
task 12 执行完毕
正在执行 task 9
task 14 执行完毕
task 8 执行完毕
task 5 执行完毕
task 7 执行完毕
task 6 执行完毕
task 9 执行完毕
```



从执行结果可以看出, 当线程池中线程的数目大于 5 时, 便将任务放入任务缓存队列里面, 当任务缓存队列满了之后, 便创建新的线程。如果上面程序中, 将 **for** 循环中改成执行 20 个任务, 就会抛出任务拒绝异常了。

不过在 **java doc** 中, 并不提倡我们直接使用 **ThreadPoolExecutor**, 而是使用 **Executors** 类中提供的几个静态方法来创建线程池:

```
Executors.newCachedThreadPool();           //创建一个缓冲池, 缓冲池容量大
小为 Integer.MAX_VALUE
Executors.newSingleThreadExecutor();       //创建容量为 1 的缓冲池
Executors.newFixedThreadPool(int);         //创建固定容量大小的缓冲池
```

下面是这三个静态方法的具体实现:



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

```
        return new FinalizableDelegatedExecutorService
            (new ThreadPoolExecutor(1, 1,
                                    0L, TimeUnit.MILLISECONDS,
                                    new
LinkedBlockingQueue<Runnable>()));
    }
    public static ExecutorService newCachedThreadPool() {
        return new ThreadPoolExecutor(0, Integer.MAX_VALUE,
                                       60L, TimeUnit.SECONDS,
                                       new SynchronousQueue<Runnable>());
    }
}
```



从它们的具体实现来看，它们实际上也是调用了 `ThreadPoolExecutor`，只不过参数都已配置好了。

`newFixedThreadPool` 创建的线程池 `corePoolSize` 和 `maximumPoolSize` 值是相等的，它使用的 `LinkedBlockingQueue`；

`newSingleThreadExecutor` 将 `corePoolSize` 和 `maximumPoolSize` 都设置为 1，也使用的 `LinkedBlockingQueue`；

`newCachedThreadPool` 将 `corePoolSize` 设置为 0，将 `maximumPoolSize` 设置为 `Integer.MAX_VALUE`，使用的 `SynchronousQueue`，也就是说来了任务就创建线程运行，当线程空闲超过 60 秒，就销毁线程。

实际中，如果 `Executors` 提供的三个静态方法能满足要求，就尽量使用它提供的三个方法，因为自己去手动配置 `ThreadPoolExecutor` 的参数有点麻烦，要根据实际任务的类型和数量来进行配置。

另外，如果 `ThreadPoolExecutor` 达不到要求，可以自己继承 `ThreadPoolExecutor` 类进行重写。

四.如何合理配置线程池的大小

本节来讨论一个比较重要的话题：如何合理配置线程池大小，仅供参考。

一般需要根据任务的类型来配置线程池大小：

如果是 CPU 密集型任务，就需要尽量压榨 CPU，参考值可以设为 $N_{CPU}+1$

如果是 IO 密集型任务，参考值可以设置为 $2*N_{CPU}$

当然，这只是一个参考值，具体的设置还需要根据实际情况进行调整，比如可以先将线程池大小设置为参考值，再观察任务运行情况和系统负载、资源利用率来进行适当调整。