  摘要： 本文主要讲了Java当中的线程池的使用方法、注意事项及其实现源码实现原理，并辅以实例加以说明,对加深Java线程池的理解有很大的帮助。

一般而言,线程池有以下几个部分:  
1.完成主要任务的一个或多个线程.  
2.用于调度管理的管理线程.  
3.要求执行的任务队列.

     首先，讲讲什么是线程池？照笔者的简单理解，其实就是**一组线程实时处于休眠状态，等待唤醒执行**。

那么为什么要有线程池这个东西呢？可以从以下几个方面来考虑：

其一、**减少在创建和销毁线程上所花的时间以及系统资源的开销** 。

其二、**将当前任务与主线程隔离，能实现和主线程的异步执行，特别是很多可以分开重复执行的任务。**

但是，开线程也不一定会带来性能上的损耗，线程池休眠也是要占用一定的内存空间，所以合理的选择线程池的大小也是有一定的依据。

**一、Executors的API介绍**

Java线程池的工厂类：Executors类,

Java类库提供了许多静态方法来创建一个线程池：

**newFixedThreadPool()**  
**说明：初始化一个指定线程数的线程池，其中corePoolSize == maxiPoolSize，使用LinkedBlockingQuene作为阻塞队列  
特点：即使当线程池没有可执行任务时，也不会释放线程。  
newCachedThreadPool()**  
**说明：初始化一个可以缓存线程的线程池，默认缓存60s，线程池的线程数可达到Integer.MAX\_VALUE，即2147483647，**内部使用SynchronousQueue作为阻塞队列；  
特点：在没有任务执行时，**当线程的空闲时间超过keepAliveTime(默认缓存时间)，会自动释放线程资源；**当提交新任务时，如果没有空闲线程，则创建新线程执行任务，会导致一定的系统开销；  
因此，使用时要注意控制并发的任务数，防止因创建大量的线程导致而降低性能。  
**newSingleThreadExecutor()**  
说明：**初始化只有一个线程的线程池，内部使用LinkedBlockingQueue作为阻塞队列。**特点：如果该线程异常结束，会重新创建一个新的线程继续执行任务，唯一的线程可以保证所提交任务的顺序执行  
**newScheduledThreadPool()**  
特定：**初始化的线程池可以在指定的时间内周期性的执行所提交的任务，在实际的业务场景中可以使用该线程池定期的同步数据。内部使用*DelayedWorkQueue()*作为阻塞队列。**

总结：除了newScheduledThreadPool的内部实现特殊一点之外，其它线程池内部都是基于**ThreadPoolExecutor**类（Executor的子类）实现的。

小结一下：在线程池中执行任务比为每个任务分配一个线程优势更多，通过重用现有的线程而不是创建新线程，可以在处理多个请求时分摊线程创建和销毁产生的巨大的开销。当请求到达时，通常工作线程已经存在，提高了响应性；通过配置线程池的大小，可以创建足够多的线程使CPU达到忙碌状态，还可以防止线程太多耗尽计算机的资源。

创建线程池基本方法：

(1)定义线程类

**[java]** [view plain](https://blog.csdn.net/evankaka/article/details/51489322) [copy](https://blog.csdn.net/evankaka/article/details/51489322)

1. **class** Handler **implements** Runnable{
2. }

(2)建立ExecutorService线程池

**[java]** [view plain](https://blog.csdn.net/evankaka/article/details/51489322) [copy](https://blog.csdn.net/evankaka/article/details/51489322)

1. ExecutorService executorService = Executors.newCachedThreadPool();

或者

**[java]** [view plain](https://blog.csdn.net/evankaka/article/details/51489322) [copy](https://blog.csdn.net/evankaka/article/details/51489322)

1. **int** cpuNums = Runtime.getRuntime().availableProcessors();  //获取当前系统的CPU 数目
2. ExecutorService executorService =Executors.newFixedThreadPool(cpuNums \* POOL\_SIZE); //ExecutorService通常根据系统资源情况灵活定义线程池大小

(3)调用线程池操作

循环操作，成为daemon,把新实例放入Executor池中

**[java]** [view plain](https://blog.csdn.net/evankaka/article/details/51489322) [copy](https://blog.csdn.net/evankaka/article/details/51489322)

1. **while**(**true**){
2. executorService.execute(**new** Handler(socket));
3. // class Handler implements Runnable{
4. 或者
5. executorService.execute(createTask(i));
6. //private static Runnable createTask(final int taskID)
7. }

execute(Runnable对象)方法其实就是对Runnable对象调用start()方法（当然还有一些其他后台动作，比如队列，优先级，IDLE timeout，active激活等）

**二、几种不同的ExecutorService线程池对象**

|  |  |
| --- | --- |
| 1.newCachedThreadPool() | -缓存型池子，先查看池中有没有以前建立的线程，如果有，就reuse.如果没有，就建一个新的线程加入池中 -缓存型池子通常用于执行一些**生存期很短的异步型任务**  因此在一些面向连接的daemon型SERVER中用得不多。 -能reuse的线程，必须是timeout IDLE内的池中线程，**缺省timeout是60s**,超过这个IDLE时长，线程实例将被终止及移出池。   注意，放入CachedThreadPool的线程不必担心其结束，**超过TIMEOUT不活动，其会自动被终止。** |
| 2.newFixedThreadPool | -newFixedThreadPool与cacheThreadPool差不多，也是能reuse就用，**但不能随时建新的线程** -其独特之处:任意时间点，最多只能有固定数目的活动线程存在，**此时如果有新的线程要建立，只能放在另外的队列中等待**，**直到当前的线程中某个线程终止直接被移出池子** -和cacheThreadPool不同，FixedThreadPool没有IDLE机制（可能也有，但既然文档没提，肯定非常长，类似依赖上层的TCP或UDP IDLE机制之类的），所以**FixedThreadPool多数针对一些很稳定很固定的正规并发线程，多用于服务器** -从方法的源代码看，cache池和fixed 池调用的是同一个底层池，只不过参数不同: **fixed池线程数固定，并且是0秒IDLE（无IDLE）** **cache池线程数支持0-Integer.MAX\_VALUE(显然完全没考虑主机的资源承受能力），60秒IDLE** |
| 3.ScheduledThreadPool | -调度型线程池 -这个池子里的线程可以**按schedule依次delay执行，或周期执行** |
| 4.SingleThreadExecutor | -单例线程，任意时间池中**只能有一个线程** -**用的是和cache池和fixed池相同的底层池，但线程数目是1-1,0秒IDLE（无IDLE）** |

**应用实例：**

**1.CachedThreadPool**

CachedThreadPool首先会按照需要创建足够多的线程来执行任务(Task)。随着程序执行的过程，有的线程执行完了任务，可以被重新循环使用时，才不再创建新的线程来执行任务。我们采用《Thinking In Java》中的例子来分析。客户端**线程和线程池之间会有一个任务队列**。当程序要关闭时，你需要注意两件事情：入队的这些任务的情况怎么样了以及正在运行的这个任务执行得如 何了。令人惊讶的是很多开发人员并没能正确地或者有意识地去关闭线程池。正确的方法有两种：**一个是让所有的入队任务都执行完毕（shutdown()）， 再就是舍弃这些任务（shutdownNow())**——这完全取决于你。比如说如果我们提交了N多任务并且希望等它们都执行完后才返回的话，那么就使用 **shutdown()**：

**[java]** [view plain](https://blog.csdn.net/evankaka/article/details/51489322) [copy](https://blog.csdn.net/evankaka/article/details/51489322)

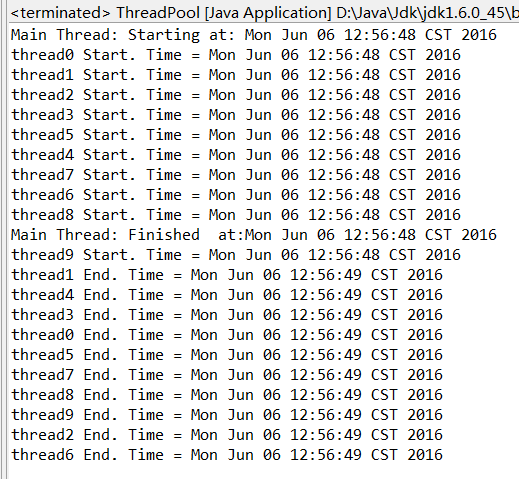
1. **import** java.util.Date;
2. **import** java.util.concurrent.ExecutorService;
3. **import** java.util.concurrent.Executors;
4. **import** java.util.concurrent.ScheduledThreadPoolExecutor;
5. **import** java.util.concurrent.TimeUnit;
7. /\*\*
8. \* 功能概要：缓冲线程池实例-execute运行
9. \*
10. \* @author linbingwen
11. \* @since  2016年5月24日
12. \*/
13. **class** Handle **implements** Runnable {
14. **private** String name;
15. **public** Handle(String name) {
16. **this**.name = "thread"+name;
17. }
18. @Override
19. **public** **void** run() {
20. System.out.println( name +" Start. Time = "+**new** Date());
21. processCommand();
22. System.out.println( name +" End. Time = "+**new** Date());
23. }
24. **private** **void** processCommand() {
25. **try** {
26. Thread.sleep(1000);
27. } **catch** (InterruptedException e) {
28. e.printStackTrace();
29. }
30. }
31. @Override
32. **public** String toString(){
33. **return** **this**.name;
34. }
35. }

验证实例：

**[java]** [view plain](https://blog.csdn.net/evankaka/article/details/51489322) [copy](https://blog.csdn.net/evankaka/article/details/51489322)

1. **public** **static** **void** testCachedThreadPool() {
2. System.out.println("Main: Starting at: "+ **new** Date());
3. ExecutorService exec = Executors.newCachedThreadPool();   //创建一个缓冲池，缓冲池容量大小为Integer.MAX\_VALUE
4. **for**(**int** i = 0; i < 10; i++) {
5. exec.execute(**new** Handle(String.valueOf(i)));
6. }
7. **exec.shutdown();  //执行到此处并不会马上关闭线程池,但之后不能再往线程池中加线程，否则会报错**
8. System.out.println("Main: Finished all threads at"+ **new** Date());
9. }

执行结果：



从上面的结果可以看出：

**1、主线程的执行与线程池里的线程分开，有可能主线程结束了，但是线程池还在运行**

**2、放入线程池的线程并不一定会按其放入的先后而顺序执行**

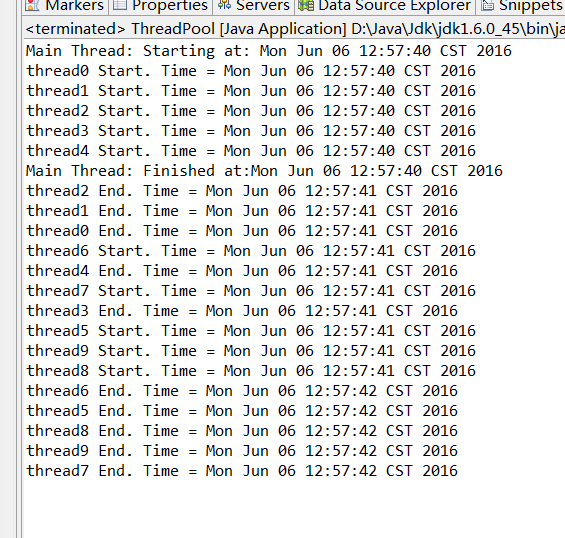
**2.FixedThreadPool**  
FixedThreadPool模式会使用一个**优先固定数目的线程来处理若干数目的任务。**规定数目的线程处理所有任务，一旦有线程处理完了任务就会被用来处理新的任务(如果有的话)。这种模式与上面的CachedThreadPool是不同的，CachedThreadPool模式下处理一定数量的任务的线程数目是不确定的。而FixedThreadPool模式下最多 的线程数目是一定的。

应用实例：

**[java]** [view plain](https://blog.csdn.net/evankaka/article/details/51489322) [copy](https://blog.csdn.net/evankaka/article/details/51489322)

1. **public** **static** **void** testFixThreadPool() {
2. System.out.println("Main Thread: Starting at: "+ **new** Date());
3. ExecutorService exec = Executors.newFixedThreadPool(5);
4. **for**(**int** i = 0; i < 10; i++) {
5. exec.execute(**new** Handle(String.valueOf(i)));
6. }
7. exec.shutdown();  //执行到此处并不会马上关闭线程池
8. System.out.println("Main Thread: Finished at:"+ **new** Date());
9. }

运行结果：



**上面创建了一个固定大小的线程池，大小为5.也就说同一时刻最多只有5个线程能运行。并且线程执行完成后就从线程池中移出。它也不能保证放入的线程能按顺序执行。这要看在等待运行的线程的竞争状态了。**

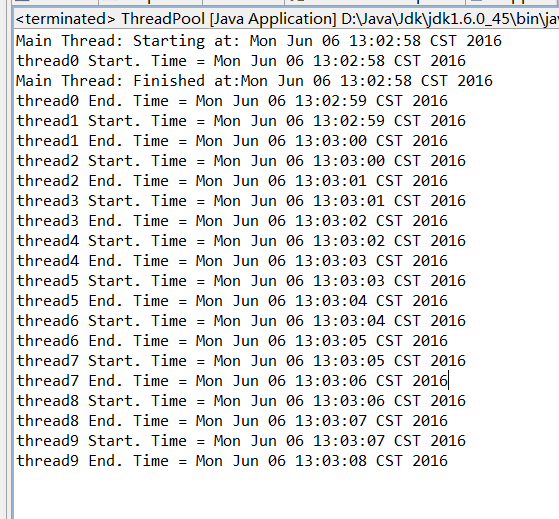
**3、newSingleThreadExecutor**

**其实这个就是创建只能运行一条线程的线程池。它能保证线程的先后顺序执行，并且能保证一条线程执行完成后才开启另一条新的线程**

**[java]** [view plain](https://blog.csdn.net/evankaka/article/details/51489322) [copy](https://blog.csdn.net/evankaka/article/details/51489322)

1. **public** **static** **void** testSingleThreadPool() {
2. System.out.println("Main Thread: Starting at: "+ **new** Date());
3. ExecutorService exec = Executors.newSingleThreadExecutor();   //创建大小为1的固定线程池
4. **for**(**int** i = 0; i < 10; i++) {
5. exec.execute(**new** Handle(String.valueOf(i)));
6. }
7. exec.shutdown();  //执行到此处并不会马上关闭线程池
8. System.out.println("Main Thread: Finished at:"+ **new** Date());
9. }

运行结果：



其实它也等价于以下：

**[java]** [view plain](https://blog.csdn.net/evankaka/article/details/51489322) [copy](https://blog.csdn.net/evankaka/article/details/51489322)

1. ExecutorService exec = Executors.newFixedThreadPool(1);

**4、newScheduledThreadPool**

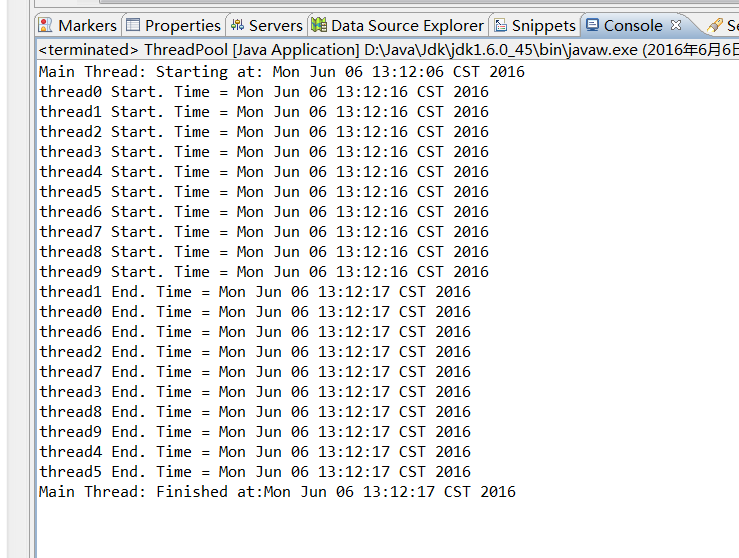
这是一个计划线程池类，它能设置线程执行的先后间隔及执行时间等，功能比上面的三个强大了一些。

以下实例：

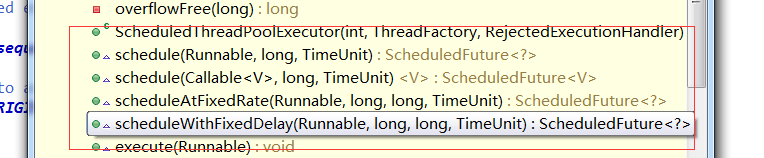
**[java]** [view plain](https://blog.csdn.net/evankaka/article/details/51489322) [copy](https://blog.csdn.net/evankaka/article/details/51489322)

1. **public** **static** **void** testScheduledThreadPool() {
2. System.out.println("Main Thread: Starting at: "+ **new** Date());
3. ScheduledThreadPoolExecutor  exec = (ScheduledThreadPoolExecutor) Executors.newScheduledThreadPool(10);   //创建大小为10的线程池
4. **for**(**int** i = 0; i < 10; i++) {
5. exec.schedule(**new** Handle(String.valueOf(i)), 10, TimeUnit.SECONDS);//延迟10秒执行
6. }
7. exec.shutdown();  //执行到此处并不会马上关闭线程池
8. **while**(!exec.isTerminated()){
9. //wait for all tasks to finish
10. }
11. System.out.println("Main Thread: Finished at:"+ **new** Date());
12. }

实现每个放入的线程延迟10秒执行。  
结果：



ScheduledThreadPoolExecutor的定时方法主要有以下四种：



下面将主要来具体讲讲scheduleAtFixedRate和scheduleWithFixedDelay

**scheduleAtFixedRate 按指定频率周期执行某个任务**  
public ScheduledFuture<?> scheduleAtFixedRate(Runnable command,   
long initialDelay,   
long period,   
TimeUnit unit);   
command：执行线程  
initialDelay：初始化延时  
period：两次开始执行最小间隔时间  
unit：计时单位  
  
**scheduleWithFixedDelay 周期定时执行某个任务/按指定频率间隔执行某个任务(注意)**  
public ScheduledFuture<?> scheduleWithFixedDelay(Runnable command,   
long initialDelay,   
long delay,   
TimeUnit unit);   
command：执行线程  
initialDelay：初始化延时  
period：前一次执行结束到下一次执行开始的间隔时间（间隔执行延迟时间）  
unit：计时单位

使用实例：

**[java]** [view plain](https://blog.csdn.net/evankaka/article/details/51489322) [copy](https://blog.csdn.net/evankaka/article/details/51489322)

1. **class** MyHandle **implements** Runnable {
3. @Override
4. **public** **void** run() {
5. System.out.println(System.currentTimeMillis());
6. **try** {
7. Thread.sleep(1 \* 1000);
8. } **catch** (InterruptedException e) {
9. // TODO Auto-generated catch block
10. e.printStackTrace();
11. }
12. }
14. }

**1.按指定频率周期执行某个任务**

下面实现每隔2秒执行一次，注意，如果上次的线程还没有执行完成，那么会阻塞下一个线程的执行。即使线程池设置得足够大。

**[java]** [view plain](https://blog.csdn.net/evankaka/article/details/51489322) [copy](https://blog.csdn.net/evankaka/article/details/51489322)

1. /\*\*
2. \* 初始化延迟0ms开始执行，每隔2000ms重新执行一次任务
3. \* @author linbingwen
4. \* @since  2016年6月6日
5. \*/
6. **public** **static** **void** executeFixedRate() {
7. ScheduledExecutorService executor = Executors.newScheduledThreadPool(10);
8. executor.scheduleAtFixedRate(
9. new MyHandle(),
10. 0,
11. 2000,
12. TimeUnit.MILLISECONDS);
13. }

**间隔指的是连续两次任务开始执行的间隔。对于scheduleAtFixedRate方法，当执行任务的时间大于我们指定的间隔时间时，它并不会在指定间隔时开辟一个新的线程并发执行这个任务。而是等待该线程执行完毕。**

**2、按指定频率间隔执行某个任务**

**[java]** [view plain](https://blog.csdn.net/evankaka/article/details/51489322) [copy](https://blog.csdn.net/evankaka/article/details/51489322)

1. /\*\*
2. \* 以固定延迟时间进行执行
3. \* 本次任务执行完成后，需要延迟设定的延迟时间，才会执行新的任务
4. \*/
5. **public** **static** **void** executeFixedDelay() {
6. ScheduledExecutorService executor = Executors.newScheduledThreadPool(10);
7. executor.scheduleWithFixedDelay(
8. **new** MyHandle(),
9. 0,
10. 2000,
11. TimeUnit.MILLISECONDS);
12. }

**间隔指的是连续上次执行完成和下次开始执行之间的间隔。**

**3.周期定时执行某个任务**  
  
周期性的执行一个任务，可以使用下面方法设定每天在固定时间执行一次任务。

**[java]** [view plain](https://blog.csdn.net/evankaka/article/details/51489322) [copy](https://blog.csdn.net/evankaka/article/details/51489322)

1. /\*\*
2. \* 每天晚上9点执行一次
3. \* 每天定时安排任务进行执行
4. \*/
5. **public** **static** **void** executeEightAtNightPerDay() {
6. ScheduledExecutorService executor = Executors.newScheduledThreadPool(1);
7. **long** oneDay = 24 \* 60 \* 60 \* 1000;
8. **long** initDelay  = getTimeMillis("21:00:00") - System.currentTimeMillis();
9. initDelay = initDelay > 0 ? initDelay : oneDay + initDelay;
11. executor.scheduleAtFixedRate(
12. **new** MyHandle(),
13. initDelay,
14. oneDay,
15. TimeUnit.MILLISECONDS);
16. }
18. /\*\*
19. \* 获取指定时间对应的毫秒数
20. \* @param time "HH:mm:ss"
21. \* @return
22. \*/
23. **private** **static** **long** getTimeMillis(String time) {
24. **try** {
25. DateFormat dateFormat = **new** SimpleDateFormat("yy-MM-dd HH:mm:ss");
26. DateFormat dayFormat = **new** SimpleDateFormat("yy-MM-dd");
27. Date curDate = dateFormat.parse(dayFormat.format(**new** Date()) + " " + time);
28. **return** curDate.getTime();
29. } **catch** (ParseException e) {
30. e.printStackTrace();
31. }
32. **return** 0;
33. }

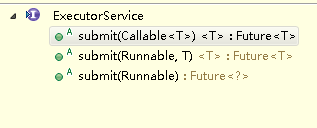
**三、线程池一些常用方法**

**1、submit()**

       将线程放入线程池中，除了使用execute，也可以使用submit（有返回值），它们两个的区别是一个使用有返回值，一个没有返回值。submit的方法很适应于生产者-消费者模式，通过和Future结合一起使用，**可以起到如果线程没有返回结果，就阻塞当前线程等待线程池结果返回**。

它主要有三种方法：

一般用第一种比较多

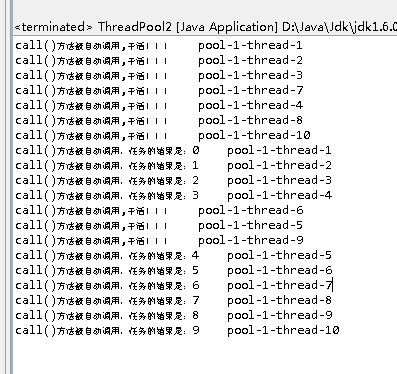


如下实例。注意，**submit中的线程要实现接口Callable**

**[java]** [view plain](https://blog.csdn.net/evankaka/article/details/51489322) [copy](https://blog.csdn.net/evankaka/article/details/51489322)

1. **package** com.func.axc.executors;
3. **import** java.util.ArrayList;
4. **import** java.util.List;
5. **import** java.util.concurrent.Callable;
6. **import** java.util.concurrent.ExecutionException;
7. **import** java.util.concurrent.ExecutorService;
8. **import** java.util.concurrent.Executors;
9. **import** java.util.concurrent.Future;
11. /\*\*
12. \* 功能概要：缓冲线程池实例-submit运行
13. \*
14. \* @author linbingwen
15. \* @since  2016年5月25日
16. \*/
17. **class** TaskWithResult **implements** Callable<String> {
18. **private** **int** id;
20. **public** TaskWithResult(**int** id) {
21. **this**.id = id;
22. }
24. /\*\*
25. \* 任务的具体过程，一旦任务传给ExecutorService的submit方法，则该方法自动在一个线程上执行。
26. \*
27. \* @return
28. \* @throws Exception
29. \*/
30. **public** String call() **throws** Exception {
31. System.out.println("call()方法被自动调用,干活！！！             " + Thread.currentThread().getName());
32. //一个模拟耗时的操作
33. **for** (**int** i = 999999; i > 0; i--) ;
34. **return**"call()方法被自动调用，任务的结果是：" + id + "    " + Thread.currentThread().getName();
35. }
36. }
38. **public** **class** ThreadPool2 {
39. **public** **static** **void** main(String[] args) {
40. ExecutorService executorService = Executors.newCachedThreadPool();
41. List<Future<String>> resultList = **new** ArrayList<Future<String>>();
43. //创建10个任务并执行
44. **for** (**int** i = 0; i < 10; i++) {
45. //使用ExecutorService执行Callable类型的任务，并将结果保存在future变量中
46. **Future<String> future = executorService.submit(new TaskWithResult(i));**
47. //将任务执行结果存储到List中
48. resultList.add(future);
49. }
50. //启动一次顺序关闭，执行以前提交的任务，但不接受新任务。如果已经关闭，则调用没有其他作用。
51. executorService.shutdown();
53. //遍历任务的结果
54. **for** (Future<String> fs : resultList) {
55. **try** {
56. System.out.println(fs.get());     //打印各个线程（任务）执行的结果
57. } **catch** (InterruptedException e) {
58. e.printStackTrace();
59. } **catch** (ExecutionException e) {
60. e.printStackTrace();
61. } **finally** {
63. }
64. }
65. }
66. }

结果如下：



从上面可以看到，输出结果的依次的。说明每次get都 阻塞了的。

看了下它的源码，其实它最终还是调用 了execute方法

**[java]** [view plain](https://blog.csdn.net/evankaka/article/details/51489322) [copy](https://blog.csdn.net/evankaka/article/details/51489322)

1. **public** <T> Future<T> submit(Callable<T> task) {
2. **if** (task == **null**) **throw** **new** NullPointerException();
3. RunnableFuture<T> ftask = newTaskFor(task);
4. execute(ftask);
5. **return** ftask;
6. }

**2、execute()**

表示往线程池添加线程，有可能会立即运行，也有可能不会。无法预知线程何时开始，何时线束。

主要源码如下：

**[java]** [view plain](https://blog.csdn.net/evankaka/article/details/51489322) [copy](https://blog.csdn.net/evankaka/article/details/51489322)

1. **public** **void** execute(Runnable command) {
2. **if** (command == **null**)
3. **throw** **new** NullPointerException();
4. **if** (poolSize >= corePoolSize || !addIfUnderCorePoolSize(command)) {
5. **if** (runState == RUNNING && workQueue.offer(command)) {
6. **if** (runState != RUNNING || poolSize == 0)
7. ensureQueuedTaskHandled(command);
8. }
9. **else** **if** (!addIfUnderMaximumPoolSize(command))
10. reject(command); // is shutdown or saturated
11. }
12. }

**3、shutdown()**

通常放在execute后面。如果调用 了这个方法，一方面，表明当前线程池已不再接收新添加的线程，新添加的线程会被拒绝执行。**另一方面，表明当所有线程执行完毕时，回收线程池的资源。注意，它不会马上关闭线程池！**

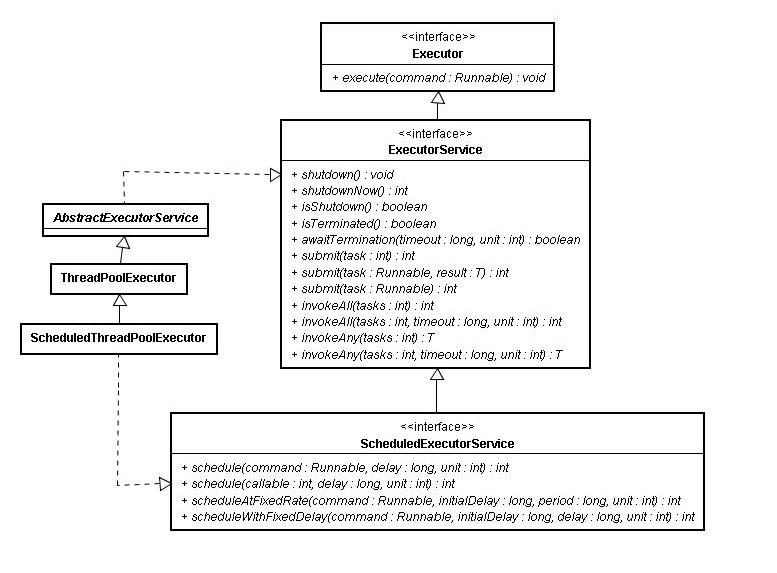
**4、shutdownNow()**

不管当前有没有线程在执行，马上关闭线程池！**这个方法要小心使用，要不可能会引起系统数据异常！**

**四、ThreadPoolExecutor技术内幕**

经过上面的过程，基本上可以掌握线程池的一些基本用法。下面再来看看JAVA中线程池的源码实现。

首先是其继承关系如下：



通过观察上面四种线程池的源码：

如：newFixedThreadPool

**[java]** [view plain](https://blog.csdn.net/evankaka/article/details/51489322) [copy](https://blog.csdn.net/evankaka/article/details/51489322)

1. **public** **static** ExecutorService newFixedThreadPool(**int** nThreads) {
2. **return** **new** ThreadPoolExecutor(nThreads, nThreads,
3. 0L, TimeUnit.MILLISECONDS,
4. **new** LinkedBlockingQueue<Runnable>());
5. }

如：newCachedThreadPool

**[java]** [view plain](https://blog.csdn.net/evankaka/article/details/51489322) [copy](https://blog.csdn.net/evankaka/article/details/51489322)

1. **public** **static** ExecutorService newCachedThreadPool() {
2. **return** **new** ThreadPoolExecutor(0, Integer.MAX\_VALUE,
3. 60L, TimeUnit.SECONDS,
4. **new** SynchronousQueue<Runnable>());
5. }

如：newSingleThreadExecutor

**[java]** [view plain](https://blog.csdn.net/evankaka/article/details/51489322) [copy](https://blog.csdn.net/evankaka/article/details/51489322)

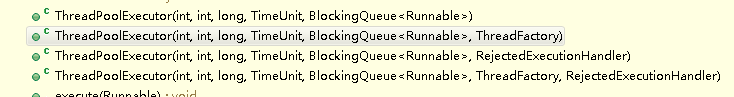
1. **public** **static** ExecutorService newSingleThreadExecutor() {
2. **return** **new** FinalizableDelegatedExecutorService
3. (**new** ThreadPoolExecutor(1, 1,
4. 0L, TimeUnit.MILLISECONDS,
5. **new** LinkedBlockingQueue<Runnable>()));
6. }

可以发现，其实它们调用的都是同一个接口**ThreadPoolExecutor方法**，只不过传入参数不一样而已。下面就来看看这个神秘的ThreadPoolExecutor。  
首先来看看它的一些基本参数：

**[java]** [view plain](https://blog.csdn.net/evankaka/article/details/51489322) [copy](https://blog.csdn.net/evankaka/article/details/51489322)

1. **public** **class** ThreadPoolExecutor **extends** AbstractExecutorService {
3. //运行状态标志位
4. **volatile** **int** runState;
5. **static** **final** **int** RUNNING    = 0;
6. **static** **final** **int** SHUTDOWN   = 1;
7. **static** **final** **int** STOP       = 2;
8. **static** **final** **int** TERMINATED = 3;
10. //线程缓冲队列，当线程池线程运行超过一定线程时并满足一定的条件，待运行的线程会放入到这个队列
11. **private** **final** BlockingQueue<Runnable> workQueue;
12. //重入锁，更新核心线程池大小、最大线程池大小时要加锁
13. **private** **final** ReentrantLock mainLock = **new** ReentrantLock();
14. //重入锁状态
15. **private** **final** Condition termination = mainLock.newCondition();
16. //工作都set集合
17. **private** **final** HashSet<Worker> workers = **new** HashSet<Worker>();
18. //线程执行完成后在线程池中的缓存时间
19. **private** **volatile** **long**  keepAliveTime;
20. //核心线程池大小
21. **private** **volatile** **int**   corePoolSize;
22. //最大线程池大小
23. **private** **volatile** **int**   maximumPoolSize;
24. //当前线程池在运行线程大小
25. **private** **volatile** **int**   poolSize;
26. //当缓冲队列也放不下线程时的拒绝策略
27. **private** **volatile** RejectedExecutionHandler handler;
28. //线程工厂，用来创建线程
29. **private** **volatile** ThreadFactory threadFactory;
30. //用来记录线程池中曾经出现过的最大线程数
31. **private** **int** largestPoolSize;
32. //用来记录已经执行完毕的任务个数
33. **private** **long** completedTaskCount;
35. ................
36. }

初始化线程池大小 有以下四种方法：



从源码中可以看到其实最终都是调用了以下的方法：

**[java]** [view plain](https://blog.csdn.net/evankaka/article/details/51489322) [copy](https://blog.csdn.net/evankaka/article/details/51489322)

1. **public** ThreadPoolExecutor(**int** corePoolSize,
2. **int** maximumPoolSize,
3. **long** keepAliveTime,
4. TimeUnit unit,
5. BlockingQueue<Runnable> workQueue,
6. ThreadFactory threadFactory,
7. RejectedExecutionHandler handler) {
8. **if** (corePoolSize < 0 ||
9. maximumPoolSize <= 0 ||
10. maximumPoolSize < corePoolSize ||
11. keepAliveTime < 0)
12. **throw** **new** IllegalArgumentException();
13. **if** (workQueue == **null** || threadFactory == **null** || handler == **null**)
14. **throw** **new** NullPointerException();
15. **this**.corePoolSize = corePoolSize;
16. **this**.maximumPoolSize = maximumPoolSize;
17. **this**.workQueue = workQueue;
18. **this**.keepAliveTime = unit.toNanos(keepAliveTime);
19. **this**.threadFactory = threadFactory;
20. **this**.handler = handler;
21. }

这里很简单，就是设置一下各个参数，并校验参数是否正确，然后抛出对应的异常。

接下来我们来看看最重要的方法execute，其源码如下：

**[java]** [view plain](https://blog.csdn.net/evankaka/article/details/51489322) [copy](https://blog.csdn.net/evankaka/article/details/51489322)

1. **public** **void** execute(Runnable command) {
2. **if** (command == **null**)
3. **throw** **new** NullPointerException();
4. **if** (poolSize >= corePoolSize || !addIfUnderCorePoolSize(command)) { //  判断1
5. **if** (runState == RUNNING && workQueue.offer(command)) { // 判断2
6. **if** (runState != RUNNING || poolSize == 0)  //  判断3
7. ensureQueuedTaskHandled(command);
8. }
9. **else** **if** (!addIfUnderMaximumPoolSize(command)) //  判断4
10. reject(command); // is shutdown or saturated
11. }
12. }

笔者在上面加了点注释。下面我们一个一个判断来看

首先判断1

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

（1）当poolSize >= corePoolSize 不成立时，表明当前线程数小于核心线程数目，左边返回fasle.接着执行右边判断!addIfUnderCorePoolSize(command)

它做了如下操作

**[java]** [view plain](https://blog.csdn.net/evankaka/article/details/51489322) [copy](https://blog.csdn.net/evankaka/article/details/51489322)

1. **private** **boolean** addIfUnderCorePoolSize(Runnable firstTask) {
2. Thread t = **null**;
3. **final** ReentrantLock mainLock = **this**.mainLock;//加锁
4. mainLock.lock();
5. **try** {
6. **if** (poolSize < corePoolSize && runState == RUNNING)//线程池在运行且当前线程小于核心线程（外面已做了一次相同的判断，确保和外面的一样）
7. t = addThread(firstTask);//加入线程
8. } **finally** {
9. mainLock.unlock();
10. }
11. **return** t != **null**;
12. }

发现它又调用addTread

**[java]** [view plain](https://blog.csdn.net/evankaka/article/details/51489322) [copy](https://blog.csdn.net/evankaka/article/details/51489322)

1. **private** Thread addThread(Runnable firstTask) { //调用这个方法之前加锁
2. Worker w = **new** Worker(firstTask);//线程包装成一个work
3. Thread t = threadFactory.newThread(w);//线程工厂从work创建线程
4. **boolean** workerStarted = **false**;
5. **if** (t != **null**) {
6. **if** (t.isAlive()) // 线程应该是未激活状态
7. **throw** **new** IllegalThreadStateException();
8. w.thread = t;
9. workers.add(w);//全局set添加一个work
10. **int** nt = ++poolSize;//当前运行线程数目加1
11. **if** (nt > largestPoolSize)
12. largestPoolSize = nt;
13. **try** {
14. t.start();//注意，这里线程执行了，但是其实真正调用的是<span style="font-family:Arial, Helvetica, sans-serif;">Worker类的run方法！！！！！！！！！</span>
15. workerStarted = **true**;
16. }
17. **finally** {
18. **if** (!workerStarted)
19. workers.remove(w);
20. }
21. }
22. **return** t;
23. }

**其实Work是真实去调用线程方法的地方,它是对Thread类的一个包装，每次Thread类调用其start方法时，就会调用到work的run方法。**其代码如下，

**[java]** [view plain](https://blog.csdn.net/evankaka/article/details/51489322) [copy](https://blog.csdn.net/evankaka/article/details/51489322)

1. **private** **void** runTask(Runnable task) { //真正发起线程方法的地方
2. **final** ReentrantLock runLock = **this**.runLock;
3. runLock.lock();
4. **try** {

**[java]** [view plain](https://blog.csdn.net/evankaka/article/details/51489322) [copy](https://blog.csdn.net/evankaka/article/details/51489322)

1. **if** ((runState >= STOP || //判断的判断
2. (Thread.interrupted() && runState >= STOP)) &&
3. hasRun)
4. thread.interrupt();
6. **boolean** ran = **false**;
7. beforeExecute(thread, task);//处理前
8. **try** {
9. task.run();//执行真正的原始线程的run方法
10. ran = **true**;
11. afterExecute(task, **null**);//处理后
12. ++completedTasks;
13. } **catch** (RuntimeException ex) {
14. **if** (!ran)
15. afterExecute(task, ex);
16. **throw** ex;
17. }
18. } **finally** {
19. runLock.unlock();
20. }
21. }
23. //这里执行线程的方法
24. **public** **void** run() {
25. **try** {
26. hasRun = **true**;
27. Runnable task = firstTask;
28. firstTask = **null**;
29. **while** (task != **null** || (task = getTask()) != **null**) {
30. runTask(task);
31. task = **null**;
32. }
33. } **finally** {
34. workerDone(**this**);
35. }
36. }

发现要执行一个线程真的很不容易，如果addIfUnderCorePoolSize返回true，刚表明成功添加一条线程，并调用了其start方法，那么整个调用到此结束。**如果返回fasle.那么就进入判断2.**

（2）**当poolSize >= corePoolSize成立时，整个判断返回true。接着执行判断2**

判断2

**[java]** [view plain](https://blog.csdn.net/evankaka/article/details/51489322) [copy](https://blog.csdn.net/evankaka/article/details/51489322)

1. **if** (runState == RUNNING && workQueue.offer(command)) { // 判断2

**如果当前线程池在运行状态，并且将当前线程加入到缓冲队列中。workQueue的offer是一个非阻塞方法。如查缓冲队列满了的话，返回为false.否则返回true;**

**如果上面两个都 为true，表明线程被成功添加到缓冲队列中，并且当前线程池在运行。进入判断3**

判断3

**[java]** [view plain](https://blog.csdn.net/evankaka/article/details/51489322) [copy](https://blog.csdn.net/evankaka/article/details/51489322)

1. **if** (runState != RUNNING || poolSize == 0)
2. ensureQueuedTaskHandled(command);

当线程被加入到线程池中，进入判断3.如果这时线程池没有在运行或者运行的线程为为0。那么就调用ensureQueuedTaskHandled，它做的其实是判断下是否在拒绝这个线程的执行。

**[java]** [view plain](https://blog.csdn.net/evankaka/article/details/51489322) [copy](https://blog.csdn.net/evankaka/article/details/51489322)

1. **private** **void** ensureQueuedTaskHandled(Runnable command) {
2. **final** ReentrantLock mainLock = **this**.mainLock;
3. mainLock.lock();
4. **boolean** reject = **false**;
5. Thread t = **null**;
6. **try** {
7. **int** state = runState;
8. **if** (state != RUNNING && workQueue.remove(command)) //线程池没有在运行，且缓冲队列中有这个线程
9. reject = **true**;
10. **else** **if** (state < STOP &&
11. poolSize < Math.max(corePoolSize, 1) &&
12. !workQueue.isEmpty())
13. t = addThread(**null**);
14. } **finally** {
15. mainLock.unlock();
16. }
17. **if** (reject)
18. reject(command); //根据拒绝策略处理线程
19. }

判断4

**[java]** [view plain](https://blog.csdn.net/evankaka/article/details/51489322) [copy](https://blog.csdn.net/evankaka/article/details/51489322)

1. **else** **if** (!addIfUnderMaximumPoolSize(command))
2. reject(command); // is shutdown or saturated

**在判断2为false时执行，表明当前线程池没有在运行或者该线程加入缓冲队列中失败，那么就会尝试再启动下该线程，如果还是失败，那就根据拒绝策略来处理这个线程。其源码如下：**

**[java]** [view plain](https://blog.csdn.net/evankaka/article/details/51489322) [copy](https://blog.csdn.net/evankaka/article/details/51489322)

1. **private** **boolean** addIfUnderMaximumPoolSize(Runnable firstTask) {
2. Thread t = **null**;
3. **final** ReentrantLock mainLock = **this**.mainLock;
4. mainLock.lock();
5. **try** {
6. **if** (poolSize < maximumPoolSize && runState == RUNNING) //如果当前运行线程数目 小于最大线程池大小 并且 线程池在运行，那么启动该线程
7. t = addThread(firstTask);
8. } **finally** {
9. mainLock.unlock();
10. }
11. **return** t != **null**;
12. }

一般调用这个方法是发生在缓冲队列已满了，那么线程池会尝试直接启动该线程。当然，它要保存当前运行的poolSize一定要小于maximumPoolSize。否则，最后。还是会拒绝这个线程！

以上大概就是整个线程池启动一条线程的整体过程。

**总结：**

**ThreadPoolExecutor中，包含了一个任务缓存队列和若干个执行线程，任务缓存队列是一个大小固定的缓冲区队列，用来缓存待执行的任务，执行线程用来处理待执行的任务。每个待执行的任务，都必须实现Runnable接口，每个Handler都要执行线程调用其run()方法，完成相应任务。**

**ThreadPoolExecutor对象初始化时，不创建任何执行线程，当有新任务进来时，才会创建执行线程。**

**构造ThreadPoolExecutor对象时，需要配置该对象的核心线程池大小和最大线程池大小：**

**当目前执行线程的总数小于核心线程大小时，所有新加入的任务，都在新线程中处理**

**当目前执行线程的总数大于或等于核心线程时，所有新加入的任务，都放入任务缓存队列中**

**当目前执行线程的总数大于或等于核心线程，并且缓存队列已满，同时此时线程总数小于线程池的最大大小，那么创建非核心线程，加入线程池中，协助处理新的任务。**

**当所有线程都在执行，线程池大小已经达到上限，并且缓存队列已满时，就rejectHandler拒绝新的任务**

**五、自定义线程池**

再来看看它的方法

**[java]** [view plain](https://blog.csdn.net/evankaka/article/details/51489322) [copy](https://blog.csdn.net/evankaka/article/details/51489322)

1. **public** ThreadPoolExecutor(**int** corePoolSize,//核心线程大小
2. **int** maximumPoolSize,//最大线程大小
3. **long** keepAliveTime,//线程缓存时间
4. TimeUnit unit,//前面keepAlive
5. BlockingQueue<Runnable> workQueue,//缓存队列
6. ThreadFactory threadFactory,//线程工大
7. RejectedExecutionHandler handler)//拒绝策略

block queue有以下几种实现：

**[java]** [view plain](https://blog.csdn.net/evankaka/article/details/51489322) [copy](https://blog.csdn.net/evankaka/article/details/51489322)

1. ArrayBlockingQueue : 有界的数组队列
2. LinkedBlockingQueue : **可支持有界/无界的队列，使用链表实现**
3. PriorityBlockingQueue : 优先队列，可以针对任务排序
4. SynchronousQueue : 队列长度为1的队列，和Array有点区别就是：client thread提交到block queue会是一个阻塞过程，直到有一个worker thread连接上来poll task。当线

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

**[java]** [view plain](https://blog.csdn.net/evankaka/article/details/51489322) [copy](https://blog.csdn.net/evankaka/article/details/51489322)

1. ThreadPoolExecutor.AbortPolicy:丢弃任务并抛出RejectedExecutionException异常。
2. ThreadPoolExecutor.DiscardPolicy：也是丢弃任务，但是不抛出异常。
3. 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）
4. ThreadPoolExecutor.CallerRunsPolicy：由调用线程处理该任务

比如定义如下一个线程池：

**[java]** [view plain](https://blog.csdn.net/evankaka/article/details/51489322) [copy](https://blog.csdn.net/evankaka/article/details/51489322)

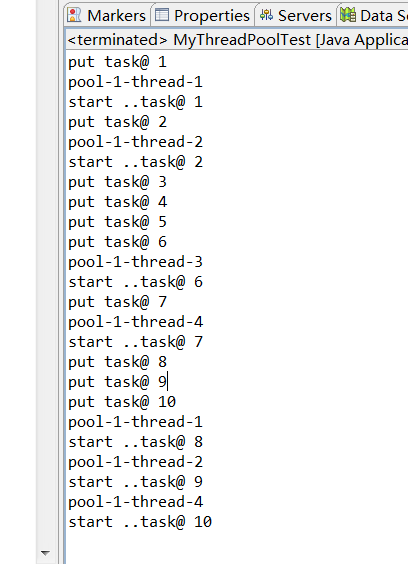
1. ThreadPoolExecutor threadPool = **new** ThreadPoolExecutor(2, 4, 3,TimeUnit.SECONDS, **new** ArrayBlockingQueue<Runnable>(3),Executors.defaultThreadFactory(),**new** ThreadPoolExecutor.DiscardOldestPolicy());

这里核心线程数为2，最大线程数为4，线程缓存时间为3秒，缓冲队列的容量设置为3。线程工厂设置为默认

下面是一个具体实例：

**[java]** [view plain](https://blog.csdn.net/evankaka/article/details/51489322) [copy](https://blog.csdn.net/evankaka/article/details/51489322)

1. **package** com.func.axc.executors;
3. **import** java.io.Serializable;
4. **import** java.util.concurrent.ArrayBlockingQueue;
5. **import** java.util.concurrent.Executors;
6. **import** java.util.concurrent.ThreadPoolExecutor;
7. **import** java.util.concurrent.TimeUnit;
9. /\*\*
10. \* 功能概要：
11. \*
12. \* @author linbingwen
13. \* @since 2016年6月7日
14. \*/
15. **public** **class** MyThreadPoolTest {
17. **private** **static** **int** produceTaskSleepTime = 2;
18. **private** **static** **int** produceTaskMaxNumber = 10;
20. **public** **static** **void** main(String[] args) {
21. // 构造一个线程池
22. ThreadPoolExecutor threadPool = **new** ThreadPoolExecutor(2, 4, 3,TimeUnit.SECONDS, **new** ArrayBlockingQueue<Runnable>(3),Executors.defaultThreadFactory(),**new** ThreadPoolExecutor.DiscardOldestPolicy());
24. **for** (**int** i = 1; i <= produceTaskMaxNumber; i++) {
25. **try** {
26. // 产生一个任务，并将其加入到线程池
27. String task = "task@ " + i;
28. System.out.println("put " + task);
29. threadPool.execute(**new** ThreadPoolTask(task));
30. // 便于观察，等待一段时间
31. Thread.sleep(produceTaskSleepTime);
32. } **catch** (Exception e) {
33. e.printStackTrace();
34. }
35. }
36. }
37. }
39. /\*\*
40. \* 线程池执行的任务
41. \*/
42. **class** ThreadPoolTask **implements** Runnable, Serializable {
43. **private** **static** **final** **long** serialVersionUID = 0;
44. **private** **static** **int** consumeTaskSleepTime = 2000;
45. // 保存任务所需要的数据
46. **private** Object threadPoolTaskData;
48. ThreadPoolTask(Object tasks) {
49. **this**.threadPoolTaskData = tasks;
50. }
52. **public** **void** run() {
53. // 处理一个任务，这里的处理方式太简单了，仅仅是一个打印语句
54. System.out.println(Thread.currentThread().getName());
55. System.out.println("start .." + threadPoolTaskData);
57. **try** {
58. // //便于观察，等待一段时间
59. Thread.sleep(consumeTaskSleepTime);
60. } **catch** (Exception e) {
61. e.printStackTrace();
62. }
63. threadPoolTaskData = **null**;
64. }
66. **public** Object getTask() {
67. **return** **this**.threadPoolTaskData;
68. }
69. }



线程池今天就说到这里，总结如下：

## jdk1.5引入Executor线程池框架，通过它把任务的提交和执行进行解耦，只需要定义好任务，然后提交给线程池，而不用关心该任务是如何执行、被哪个线程执行，以及什么时候执行。

# ****初始化线程池（4种）****

## ****简介：****

**Java线程池的工厂类：Executors类,**

初始化4种类型的线程池：

**newFixedThreadPool()**  
说明：初始化一个指定线程数的线程池，其中corePoolSize == maxiPoolSize，使用LinkedBlockingQuene作为阻塞队列  
特点：即使当线程池没有可执行任务时，也不会释放线程。  
**newCachedThreadPool()**  
说明：初始化一个可以缓存线程的线程池，默认缓存60s，线程池的线程数可达到Integer.MAX\_VALUE，即2147483647，内部使用SynchronousQueue作为阻塞队列；  
特点：在没有任务执行时，当线程的空闲时间超过keepAliveTime，会自动释放线程资源；当提交新任务时，如果没有空闲线程，则创建新线程执行任务，会导致一定的系统开销；  
因此，使用时要注意控制并发的任务数，防止因创建大量的线程导致而降低性能。  
**newSingleThreadExecutor()**  
说明：初始化只有一个线程的线程池，内部使用LinkedBlockingQueue作为阻塞队列。  
特点：如果该线程异常结束，会重新创建一个新的线程继续执行任务，唯一的线程可以保证所提交任务的顺序执行  
**newScheduledThreadPool()**  
特定：初始化的线程池可以在指定的时间内周期性的执行所提交的任务，在实际的业务场景中可以使用该线程池定期的同步数据。

总结：除了newScheduledThreadPool的内部实现特殊一点之外，其它线程池内部都是基于**ThreadPoolExecutor**类（Executor的子类）实现的。

## ****ThreadPoolExecutor内部具体实现：****

ThreadPoolExecutor类构造器语法形式：

ThreadPoolExecutor（corePoolSize,maxPoolSize,keepAliveTime,timeUnit,workQueue,threadFactory,handle);

方法参数：  
　**corePoolSize：核心线程数**  
**maxPoolSize：最大线程数**  
**keepAliveTime：线程存活时间（在corePore<\*<maxPoolSize情况下有用）**  
**timeUnit：存活时间的时间单位**  
**workQueue：阻塞队列（用来保存等待被执行的任务）**

注：关于workQueue参数的取值,**JDK提供了4种阻塞队列类型供选择：**  
　　      ArrayBlockingQueue：基于数组结构的有界阻塞队列，按FIFO排序任务；  
　　      inkedBlockingQuene：基于链表结构的阻塞队列，按FIFO排序任务，吞吐量通常要高于

            SynchronousQuene：一个不存储元素的阻塞队列，每个插入操作必须等到另一个线程调用移除操作，否则插入操作一直处于阻塞状态，吞吐量通常要高于ArrayBlockingQuene；

　　      PriorityBlockingQuene：具有优先级的无界阻塞队列；

**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**

 注： 当线程池的饱和策略，当阻塞队列满了，且没有空闲的工作线程，如果继续提交任务，必须采取一种策略处理该任务，线程池提供了4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

　　　　当然也可以根据应用场景实现***RejectedExecutionHandler***接口，自定义饱和策略，如记录日志或持久化存储不能处理的任务。

# ****线程池的状态（5种）****

其中AtomicInteger变量ctl的功能非常强大：利用低29位表示线程池中线程数，通过高3位表示线程池的运行状态：  
1、**RUNNING**：-1 << COUNT\_BITS，即高3位为111，该状态的线程池会接收新任务，并处理阻塞队列中的任务；  
2、**SHUTDOWN**： 0 << COUNT\_BITS，即高3位为000，该状态的线程池不会接收新任务，但会处理阻塞队列中的任务；  
3、**STOP** ： 1 << COUNT\_BITS，即高3位为001，该状态的线程不会接收新任务，也不会处理阻塞队列中的任务，而且会中断正在运行的任务；  
4、**TIDYING** ： 2 << COUNT\_BITS，即高3位为010，该状态表示线程池对线程进行整理优化；  
5、**TERMINATED**： 3 << COUNT\_BITS，即高3位为011，该状态表示线程池停止工作；

# ****向线程池提交任务（2种）****

有两种方式：

**Executor.execute(Runnable command);**

**ExecutorService.submit(Callable<T> task);**

## ****execute()内部实现****

1.首次通过workCountof()**获知当前线程池中的线程数**，

**如果小于corePoolSize, 就通过addWorker()创建线程并执行该任务；**

**否则，将该任务放入阻塞队列；**

2. 如果能成功将任务放入阻塞队列中,

如果当前线程池**是非RUNNING状态**，则将该任务从**阻塞队列中移除**，然后执行**reject()处理该任务**；

如果当前线程池**处于RUNNING状态**，则需要再次检查线程池（因为可能在上次检查后，有线程资源被释放），**是否有空闲的线程；如果有则执行该任务**；

3、如果不能将任务放入阻塞队列中,说明阻塞队列已满；那么将通过**addWoker()尝试创建一个新的线程去执行这个任务；如果addWoker()执行失败，说明线程池中线程数达到maxPoolSize,则执行reject()处理任务；**

## ****sumbit()内部实现****

会将提交的Callable任务会被封装成了一个**FutureTask对象**

**FutureTask类实现了Runnable接口，这样就可以通过Executor.execute()提交**FutureTask到线程池中等待被执行，最终执行的是**FutureTask的run方法**；

**比较：**

 两个方法都可以向线程池提交任务，**execute()方法的返回类型是void**，它定义在Executor接口中, **而submit()方法可以返回持有计算结果的Future对象**，它定义在ExecutorService接口中，它扩展了Executor接口，其它线程池类像ThreadPoolExecutor和ScheduledThreadPoolExecutor都有这些方法。

public class RunnableTestMain {  
  
    public static void main(String[] args) {  
        ExecutorService pool = Executors.newFixedThreadPool(2);  
          
        /\*\*  
         \* execute(Runnable x) 没有返回值。可以执行任务，但无法判断任务是否成功完成。  
         \*/  
        pool.execute(new RunnableTest("Task1"));   
          
        /\*\*  
         \* submit(Runnable x) 返回一个future。可以用这个future来判断任务是否成功完成。请看下面：  
         \*/  
        Future future = pool.submit(new RunnableTest("Task2"));  
          
        try {  
            if(future.get()==null){//如果Future's get返回null，任务完成  
                System.out.println("任务完成");  
            }  
        } catch (InterruptedException e) {  
        } catch (ExecutionException e) {  
            //否则我们可以看看任务失败的原因是什么  
            System.out.println(e.getCause().getMessage());  
        }  
  
    }  
  
}

1、接收的参数不一样

**2、submit有返回值，而execute没有**

用到返回值的例子，比如说我有很多个做validation的task，我希望所有的task执行完，然后每个task告诉我它的执行结果，是成功还是失败，如果是失败，原因是什么。  
然后我就可以把所有失败的原因综合起来发给调用者。

个人觉得cancel execution这个用处不大，很少有需要去取消执行的。

而最大的用处应该是第二点。  
**3、submit方便Exception处理**  
意思就是如果你在你的task里会抛出checked或者unchecked exception，  
而你又希望外面的调用者能够感知这些exception并做出及时的处理，那么就需要用到submit，通过捕获Future.get抛出的异常。

**线程池的关闭（2种）**

　　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

# 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()，

# ****总结：****

**线程池中的核心线程数，当提交一个任务时，线程池创建一个新线程执行任务，直到当前线程数等于corePoolSize；如果当前线程数为corePoolSize，继续提交的任务被保存到阻塞队列中，等待被执行；如果阻塞队列满了，那就创建非核心线程执行当前任务；直到线程池中的线程数达到maxPoolSize,这时再有任务来，只能执行reject()处理该任务；**

注：如果执行了线程池的prestartAllCoreThreads()方法，线程池会提前创建并启动所有核心线程。