

fengsehng

公众号：互联网学术（IT-paper）

我的微信公众号



昵称：fengsehng

园龄：1年9个月

粉丝：3

关注：1

[+加关注](#)

< 2018年9月 >

日	一	二	三	四	五	六
26	27	28	29	30	31	1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30	1	2	3	4	5	6

搜索

[博客园](#) [首页](#) [新随笔](#) [联系](#) [订阅](#) [XML](#) [管理](#)

随笔-195 评论-2 文章-0

Java程序员必备知识-多线程框架Executor详解

为什么引入Executor线程池框架

new Thread()的缺点

每次new Thread()耗费性能

调用new Thread()创建的线程缺乏管理，被称为野线程，而且可以无限制创建，之间相互竞争，会导致过多占用系统资源导致系统瘫痪。

不利于扩展，比如如定时执行、定期执行、线程中断

采用线程池的优点

重用存在的线程，减少对象创建、消亡的开销，性能佳

可有效控制最大并发线程数，提高系统资源的使用率，同时避免过多资源竞争，避免堵塞

提供定时执行、定期执行、单线程、并发数控制等功能

Executor的介绍

在Java 5之后，并发编程引入了一堆新的启动、调度和管理线程的API。

找找看 谷歌搜索

常用链接

[我的随笔](#)[我的评论](#)[我的参与](#)[最新评论](#)[我的标签](#)

最新随笔

1. ubuntu安装qq
2. Android的log日志知识点剖析
3. Hexo写博客
4. 搭建属于自己的技术博客
5. Java程序员必备知识-多线程框架Executor详解
6. Java程序员必须掌握的线程知识-Callable和Future
7. Java泛型详解
8. Java程序员的必备知识-类加载机制详解
9. 程序员必须搞清的概念-equals和hashCode的区别
10. java内存垃圾回收模型

我的标签

[博客\(1\)](#)

随笔档案(195)

- 2016年11月 (6)
- 2016年9月 (41)
- 2016年8月 (17)
- 2016年7月 (39)
- 2016年3月 (7)

Executor框架便是Java 5中引入的,

其内部使用了线程池机制，它在java.util.concurrent 包下，通过该框架来控制线程的启动、执行和关闭，可以简化并发编程的操作。因此，在Java 5之后，通过Executor来启动线程比使用Thread的start方法更好，除了更易管理，效率更好（用线程池实现，节约开销）外，还有关键的一点：有助于避免this逃逸问题——如果我们在构造器中启动一个线程，因为另一个任务可能会在构造器结束之前开始执行，此时可能会访问到初始化了一半的对象用Executor在构造器中。

Executor框架包括：线程池，Executor，Executors，ExecutorService，CompletionService，Future，Callable等。

Executors方法介绍

Executors工厂类

通过Executors提供四种线程池，newFixedThreadPool、newCachedThreadPool、newSingleThreadExecutor、newScheduledThreadPool。

1.public static ExecutorService newFixedThreadPool(int nThreads)

创建固定数目线程的线程池。

2.public static ExecutorService newCachedThreadPool()

创建一个可缓存的线程池，调用execute将重用以前构造的线程（如果线程可用）。如果现有线程没有可用的，则创建一个新线程并添加到池中。终止并从缓存中移除那些已有 60 秒钟未被使用的线程。

3.public static ExecutorService newSingleThreadExecutor()

创建一个单线程化的Executor。

4.public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize)

创建一个支持定时及周期性的任务执行的线程池，多数情况下可用来替代Timer类。

1.newFixedThreadPool创建一个可重用固定线程数的线程池，以共享的无界队列方式来运行这些线程。

2016年2月 (5)
2016年1月 (6)
2015年12月 (5)
2015年11月 (9)
2015年10月 (11)
2015年9月 (24)
2015年8月 (17)
2015年7月 (5)
2015年6月 (3)

积分与排名

积分 - 14021
排名 - 30771

最新评论

1. Re:Java程序员必须掌握的线程知识-Callable和Future

文章不严谨, 不过写的好, get(long timeout, TimeUnit unit)用来获取执行结果, 如果在指定时间内, 还没获取到结果, 就直接返回null。这里有问题, 如果指定时间没有获取到结果,

--熊猫你好

2. Re:搭建属于自己的技术博客棒!

--fengsehng

阅读排行榜

1. Java程序员必备知识-多线程框架Executor详解(8853)
2. Java程序员必须掌握的线程知识-Callable和Future(4992)
3. 搭建属于自己的技术博客(2826)
4. Hexo写博客(2337)

示例

```
ExecutorService executorService = Executors.newFixedThreadPool(5);

for (int i = 0; i < 20; i++) {
    Runnable syncRunnable = new Runnable() {
        @Override
        public void run() {
            Log.e(TAG, Thread.currentThread().getName());
        }
    };
    executorService.execute(syncRunnable);
}
```

运行结果: 总共只会创建5个线程, 开始执行五个线程, 当五个线程都处于活动状态, 再次提交的任务都会加入队列等到其他线程运行结束, 当线程处于空闲状态时会被下一个任务复用

2.newCachedThreadPool创建一个可缓存线程池, 如果线程池长度超过处理需要, 可灵活回收空闲线程

示例:

```
ExecutorService executorService = Executors.newCachedThreadPool();

for (int i = 0; i < 100; i++) {
    Runnable syncRunnable = new Runnable() {
        @Override
        public void run() {
            Log.e(TAG, Thread.currentThread().getName());
        }
    };
    executorService.execute(syncRunnable);
}
```

运行结果: 可以看出缓存线程池大小是不定值, 可以需要创建不同数量的线程, 在使用缓存型池时, 先查看池中有没有以前创建的线程, 如果有, 就复用. 如果没有, 就新建新的线程加入池中, 缓存型池子通常用于执行一些生存期很短的异步型任务

5. Android的log日志知识点剖析
(697)

评论排行榜

1. 搭建属于自己的技术博客(1)
2. Java程序员必须掌握的线程知识-Callable和Future(1)

推荐排行榜

1. Java程序员必须掌握的线程知识-Callable和Future(2)

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

schedule(Runnable command,long delay, TimeUnit unit)创建并执行在给定延迟后启用的一次性操作

示例：表示从提交任务开始计时，5000毫秒后执行

```
ScheduledExecutorService executorService = Executors.newScheduledThreadPool(5);
for (int i = 0; i < 20; i++) {
    Runnable syncRunnable = new Runnable() {
        @Override
        public void run() {
            Log.e(TAG, Thread.currentThread().getName());
        }
    };
    executorService.schedule(syncRunnable, 5000, TimeUnit.MILLISECONDS);
}
```

运行结果和newFixedThreadPool类似，不同的是newScheduledThreadPool是延时一定时间之后才执行

scheduleAtFixedRate(Runnable command, long initialDelay, long period, TimeUnit unit)

创建并执行一个在给定初始延迟后首次启用的定期操作，后续操作具有给定的周期；也就是将在 initialDelay 后开始执行，然后在initialDelay+period 后执行，接着在 initialDelay + 2 * period 后执行，依此类推

```
ScheduledExecutorService executorService = Executors.newScheduledThreadPool(5);
Runnable syncRunnable = new Runnable() {
    @Override
    public void run() {
        Log.e(TAG, Thread.currentThread().getName());
    }
};
executorService.scheduleAtFixedRate(syncRunnable, 5000, 3000, TimeUnit.MILLISECONDS);
```

scheduleWithFixedDelay(Runnable command, long initialDelay, long delay, TimeUnit unit)

创建并执行一个在给定初始延迟后首次启用的定期操作，随后，在每一次执行终止和下一次执行开始之间都存在给定的延迟

```
ScheduledExecutorService executorService = Executors.newScheduledThreadPool(5);

Runnable syncRunnable = new Runnable() {
    @Override
    public void run() {
        Log.e(TAG, Thread.currentThread().getName());
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
};

executorService.scheduleWithFixedDelay(syncRunnable, 5000, 3000, TimeUnit.MILLISECONDS);
```

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

```
ExecutorService executorService = Executors.newSingleThreadExecutor();

for (int i = 0; i < 20; i++) {
    Runnable syncRunnable = new Runnable() {
        @Override
        public void run() {
            Log.e(TAG, Thread.currentThread().getName());
        }
    };
    executorService.execute(syncRunnable);
}
```

运行结果：只会创建一个线程，当上一个执行完之后才会执行第二个

ExecutorService

ExecutorService是一个接口，ExecutorService接口继承了Executor接口，定义了一些生命周期的方法。

```
public interface ExecutorService extends Executor {

    void shutdown(); // 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

    List<Runnable> shutdownNow(); // 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

    boolean isShutdown(); // 判断线程池是否已经关闭

    boolean isTerminated(); // 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 永不返回 true。

    boolean awaitTermination(long timeout, TimeUnit unit) // 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

    <T> Future<T> submit(Callable<T> task); // 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(Runnable task, T result); // 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回给定的结果。

```
Future<?> submit(Runnable 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 完成时将会返回 null
```

```
<T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks) //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。  
    throws InterruptedException;
```

```
<T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks,  
                               long timeout, TimeUnit unit) //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。  
    throws InterruptedException;
```

```
<T> T invokeAny(Collection<? extends Callable<T>> tasks) //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。  
    throws InterruptedException, ExecutionException;
```

```
<T> T invokeAny(Collection<? extends Callable<T>> tasks,  
                long timeout, TimeUnit unit)  
    throws InterruptedException, ExecutionException, TimeoutException;  
}
```

ExecutorService接口继承自Executor接口, 它提供了更丰富的实现多线程的方法, 比如, ExecutorService提供了关闭自己的方法, 以及可为跟踪一个或多个异步任务执行状况而生成 Future 的方法。可以调用ExecutorService的shutdown () 方法来平滑地关闭 ExecutorService, 调用该方法后, 将导致ExecutorService停止接受任何新的任务且等待已经提交的任务执行完成(已经提交的任务会分两类: 一类是已经在执行的, 另一类是还没有开始执行的), 当所有已经提交的任务执行完毕后将会关闭ExecutorService。因此我们一般用该接口来实现和管理多线程。

ExecutorService的生命周期包括三种状态：运行、关闭、终止。创建后便进入运行状态，当调用了shutdown () 方法时，便进入关闭状态，此时意味着ExecutorService不再接受新的任务，但它还在执行已经提交了的任务，当素有已经提交了的任务执行完后，便到达终止状态。如果不调用shutdown () 方法，ExecutorService会一直处在运行状态，不断接收新的任务，执行新的任务，服务器端一般不需要关闭它，保持一直运行即可。

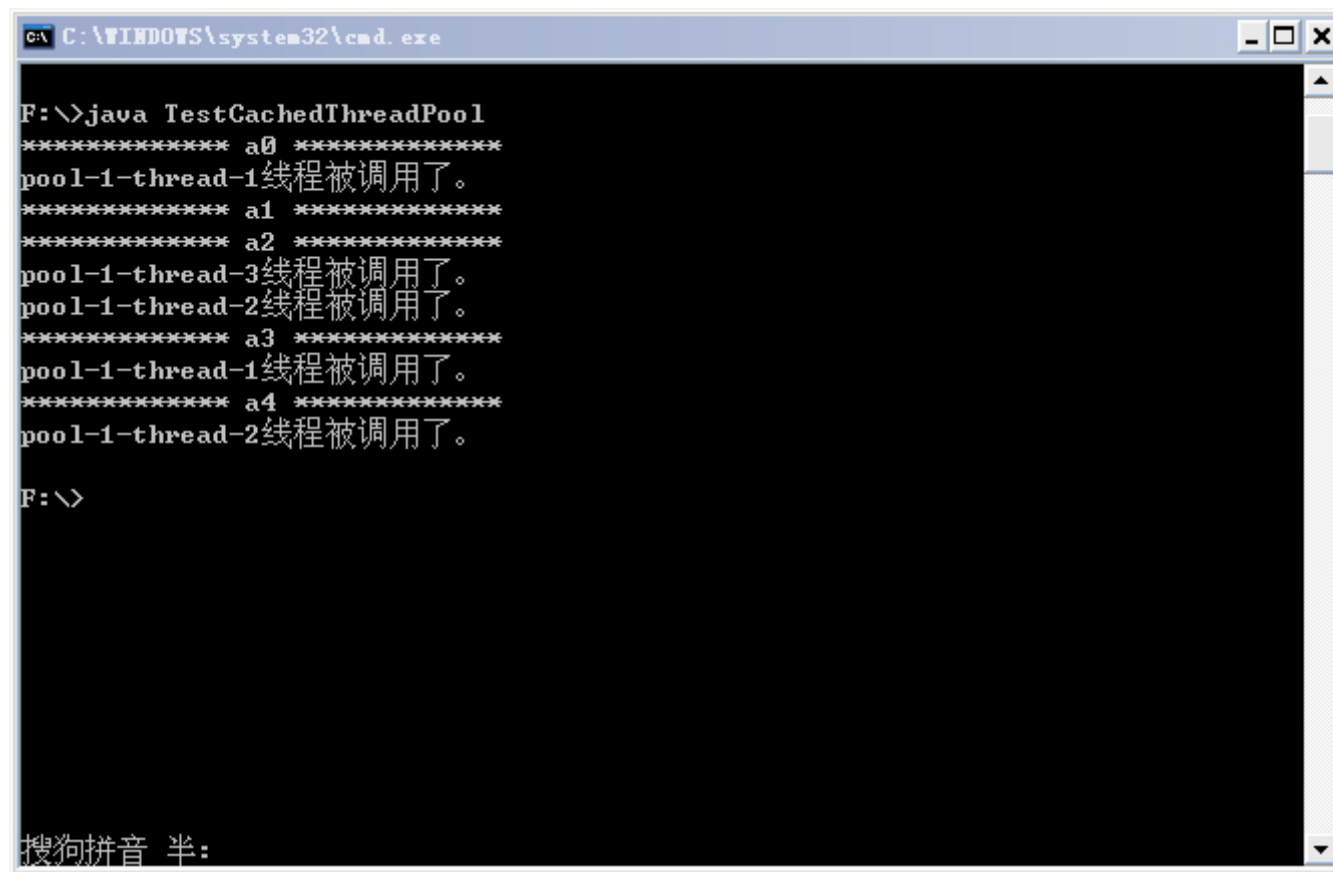
Executor执行Runnable任务

一旦Runnable任务传递到execute () 方法，该方法便会自动在一个线程上执行。下面是Executor执行Runnable任务的示例代码：

```
public class TestCachedThreadPool{
    public static void main(String[] args){
        ExecutorService executorService = Executors.newCachedThreadPool();
        // ExecutorService executorService = Executors.newFixedThreadPool(5);
        // ExecutorService executorService = Executors.newSingleThreadExecutor();
        for (int i = 0; i < 5; i++){
            executorService.execute(new TestRunnable());
            System.out.println("***** a" + i + " *****");
        }
        executorService.shutdown();
    }
}

class TestRunnable implements Runnable{
    public void run(){
        System.out.println(Thread.currentThread().getName() + "线程被调用了。");
    }
}
```

结果



```
C:\WINDOWS\system32\cmd.exe

F:\>java TestCachedThreadPool
***** a0 *****
pool-1-thread-1线程被调用了。
***** a1 *****
***** a2 *****
pool-1-thread-3线程被调用了。
pool-1-thread-2线程被调用了。
***** a3 *****
pool-1-thread-1线程被调用了。
***** a4 *****
pool-1-thread-2线程被调用了。

F:\>
```

Executor执行Callable任务

在Java 5之后，任务分两类：一类是实现了Runnable接口的类，一类是实现了Callable接口的类。两者都可以被ExecutorService执行，但是Runnable任务没有返回值，而Callable任务有返回值。并且Callable的call()方法只能通过ExecutorService的submit(Callable task)方法来执行，并且返回一个Future，是表示任务等待完成的Future。

下面给出一个Executor执行Callable任务的示例代码：

```
public class CallableDemo{
    public static void main(String[] args){
        ExecutorService executorService = Executors.newCachedThreadPool();
        List<Future<String>> resultList = new ArrayList<Future<String>>();
```

```
//创建10个任务并执行
for (int i = 0; i < 10; i++){
    //使用ExecutorService执行Callable类型的任务，并将结果保存在future变量中
    Future<String> future = executorService.submit(new TaskWithResult(i));
    //将任务执行结果存储到List中
    resultList.add(future);
}

//遍历任务的结果
for (Future<String> fs : resultList){
    try{
        while(!fs.isDone()); //Future返回如果没有完成，则一直循环等待，直到Future返回完成
        System.out.println(fs.get()); //打印各个线程（任务）执行的结果
    } catch (InterruptedException e){
        e.printStackTrace();
    } catch (ExecutionException e){
        e.printStackTrace();
    } finally{
        //启动一次顺序关闭，执行以前提交的任务，但不接受新任务
        executorService.shutdown();
    }
}

}

}

class TaskWithResult implements Callable<String>{
    private int id;

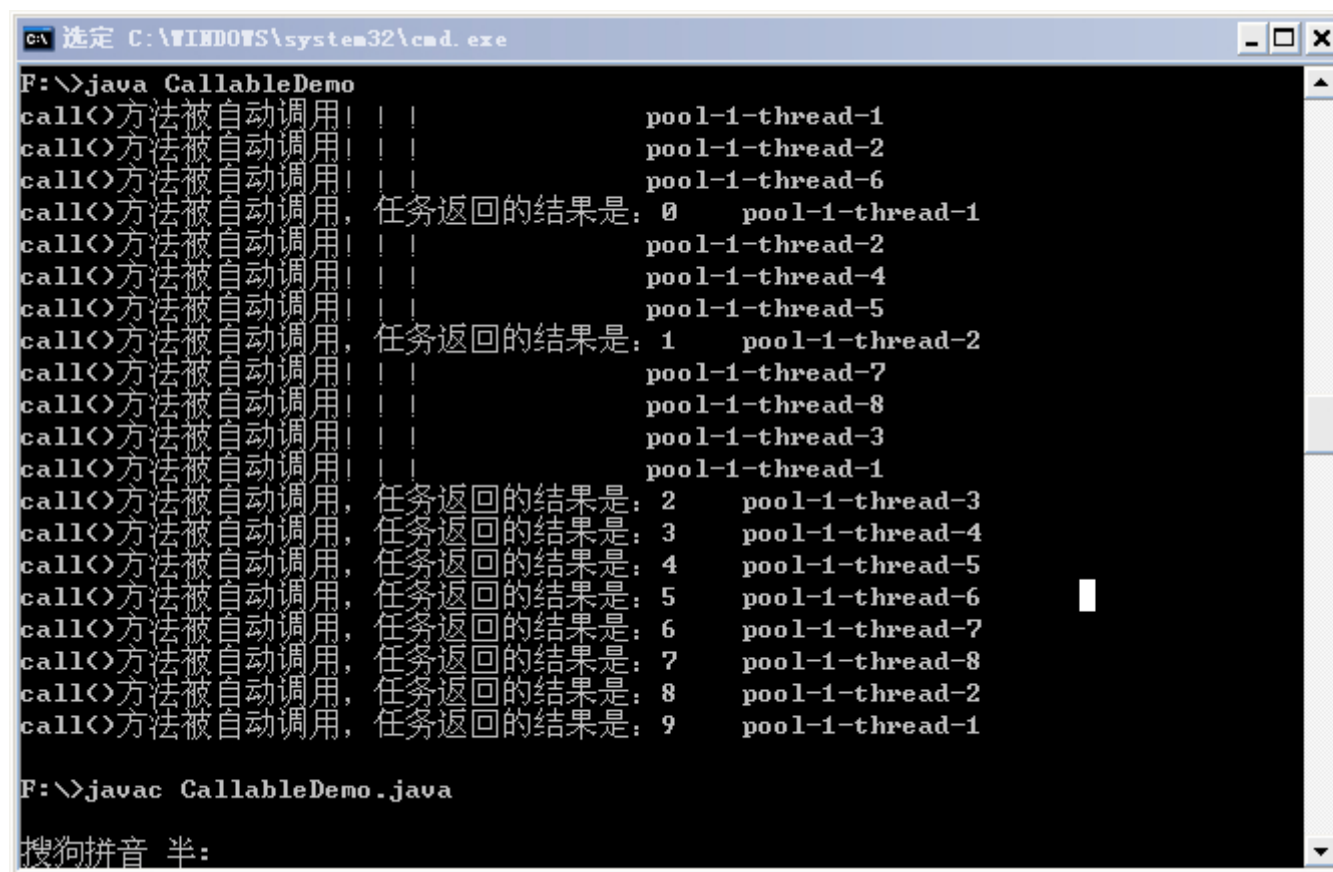
    public TaskWithResult(int id){
        this.id = id;
    }

    /**
```

```
* 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  
*/
```

```
public String call() throws Exception {  
    System.out.println("call()方法被自动调用!!! " + Thread.currentThread().getName());  
    //该返回结果将被Future的get方法得到  
    return "call()方法被自动调用，任务返回的结果是：" + id + " " +  
    Thread.currentThread().getName();  
}  
}
```

某次执行结果如下：



```
C:\>选定 C:\WINDOWS\system32\cmd.exe  
F:\>java CallableDemo  
call()方法被自动调用!!! pool-1-thread-1  
call()方法被自动调用!!! pool-1-thread-2  
call()方法被自动调用!!! pool-1-thread-6  
call()方法被自动调用，任务返回的结果是：0 pool-1-thread-1  
call()方法被自动调用!!! pool-1-thread-2  
call()方法被自动调用!!! pool-1-thread-4  
call()方法被自动调用!!! pool-1-thread-5  
call()方法被自动调用，任务返回的结果是：1 pool-1-thread-2  
call()方法被自动调用!!! pool-1-thread-7  
call()方法被自动调用!!! pool-1-thread-8  
call()方法被自动调用!!! pool-1-thread-3  
call()方法被自动调用!!! pool-1-thread-1  
call()方法被自动调用，任务返回的结果是：2 pool-1-thread-3  
call()方法被自动调用，任务返回的结果是：3 pool-1-thread-4  
call()方法被自动调用，任务返回的结果是：4 pool-1-thread-5  
call()方法被自动调用，任务返回的结果是：5 pool-1-thread-6  
call()方法被自动调用，任务返回的结果是：6 pool-1-thread-7  
call()方法被自动调用，任务返回的结果是：7 pool-1-thread-8  
call()方法被自动调用，任务返回的结果是：8 pool-1-thread-2  
call()方法被自动调用，任务返回的结果是：9 pool-1-thread-1  
  
F:\>javac CallableDemo.java  
搜狗拼音 半:
```

从结果中可以同样可以看出，submit也是首先选择空闲线程来执行任务，如果没有，才会创建新的线程来执行任务。另外，需要注意：如果Future的返回尚未完成，则get () 方法会阻塞等待，直到Future完成返回，可以通过调用isDone () 方法判断Future是否完成了返回。

自定义线程池

自定义线程池，可以用ThreadPoolExecutor类创建，它有多个构造方法来创建线程池，用该类很容易实现自定义的线程池，这里先贴上示例程序：

```
public class ThreadPoolTest{
    public static void main(String[] args){
        //创建等待队列
        BlockingQueue<Runnable> bqueue = new ArrayBlockingQueue<Runnable>(20);
        //创建线程池，池中保存的线程数为3，允许的最大线程数为5
        ThreadPoolExecutor pool = new ThreadPoolExecutor(3,5,50,TimeUnit.MILLISECONDS,bqueue);
        //创建七个任务
        Runnable t1 = new MyThread();
        Runnable t2 = new MyThread();
        Runnable t3 = new MyThread();
        Runnable t4 = new MyThread();
        Runnable t5 = new MyThread();
        Runnable t6 = new MyThread();
        Runnable t7 = new MyThread();
        //每个任务会在一个线程上执行
        pool.execute(t1);
        pool.execute(t2);
        pool.execute(t3);
        pool.execute(t4);
        pool.execute(t5);
        pool.execute(t6);
        pool.execute(t7);
        //关闭线程池
        pool.shutdown();
    }
}
```

```
class MyThread implements Runnable{  
    @Override  
    public void run() {  
        System.out.println(Thread.currentThread().getName() + "正在执行。。。");  
        try{  
            Thread.sleep(100);  
        }catch(InterruptedException e){  
            e.printStackTrace();  
        }  
    }  
}
```

参考

<http://gold.xitu.io/entry/57cbaf667db2a2007895256e>

http://blog.csdn.net/ns_code/article/details/17465497

<http://www.infoq.com/cn/articles/executor-framework-thread-pool-task-execution-part-01>

<http://www.cnblogs.com/limingluzhu/p/4858776.html>

我的微信二维码如下，欢迎交流讨论

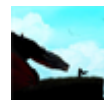


欢迎关注《IT面试题汇总》微信订阅号。每天推送经典面试题和面试心得技巧

微信订阅号二维码如下:



```
<script type="text/javascript"> (function () {('pre.prettyprint code').each(function () { var lines =  
(this).text().split('\n').length; var numbering =  
( '< ul/ >').addClass('pre - numbering').hide();(this).addClass('has-numbering').parent().append(  
numbering); for(i = 1; i <= lines; i++)$numbering.append('$(< li/ >').text(i));numbering.fadeIn(  
1700); }); }); </script>
```

[好文要顶](#)[关注我](#)[收藏该文](#)

fengsehng

关注 - 1

粉丝 - 3

[+加关注](#)

0

0

« 上一篇: [Java程序员必须掌握的线程知识-Callable和Future](#)

» 下一篇: [搭建属于自己的技术博客](#)

posted on 2016-11-09 21:10 [fengsehng](#) 阅读(8854) 评论(0) [编辑](#) [收藏](#)

[刷新评论](#) [刷新页面](#) [返回顶部](#)

注册用户登录后才能发表评论, 请 [登录](#) 或 [注册](#), [访问](#)网站首页。



最新IT新闻:

- [美团基石投资者敲定 除了腾讯还有四家知名基金](#)
 - [谷歌一盘棋：推出无下载可试玩功能，招安小游戏为App输血](#)
 - [炒币故事：想要一夜暴富结果债台高筑 却仍希望翻盘](#)
 - [EOS的超级竞选者们：节衣缩食准备“过冬”](#)
 - [滴滴代驾司机工作期间身亡 120万保单变成了1万？](#)
- » [更多新闻...](#)



最新知识库文章:

- [如何招到一个靠谱的程序员](#)
 - [一个故事看懂“区块链”](#)
 - [被踢出去的用户](#)
 - [成为一个有目标的学习者](#)
 - [历史转折中的“杭派工程师”](#)
- » [更多知识库文章...](#)

Powered by: [博客园](#) 模板提供: [沪江博客](#) Copyright ©2018 fengsehng