ThreadPoolExecutor类

- corePoolSize：核心池的大小
- maxmumPoolSize：线程池的最大线程数
- keepAliveTime：空闲的线程可以存活的时间，线程池中的线程数大于corePoolSize，线程池进入了紧急预案状态，当线程数小于corePoolSize，keepAliveTime不起作用。
- unit：keepAliveTime参数的时间单位
- workQueue：一个阻塞队列，用来存储等待执行的任务，一般阻塞队列有以下几种选择：
    - ArrayBlockingQueue：基于数组的先进先出队列，创建时必须制定大小。
    - LinkedBlockingQueue：基于链表的先进先出队列，如果创建时没有指定大小，默认Integer.MAX_VALUE。
    - SynchronousQueue：不会保存提交的任务，而是直接创建一个新的线程来执行新的任务。
    - PriorityBlockingQueue：具有优先级的阻塞队列。
- threadFactory：线程工厂，用来创建线程。
- handler：表示拒绝处理任务所采取的策略，有以下四种取值。
    - ThreadPoolExecutor.AbortPolicy：丢弃任务并且抛出异常。
    - ThreadPoolExecutor.DiscardPolicy：丢弃任务但是不抛出异常。
    - ThreadPoolExecutor.DiscardOldestPolicy：丢弃队列最前面的任务，然后重新执行当前任务。
    - ThreadPoolExecutor.CallerRunsPolicy：由调用线程处理该任务。

线程池的状态

1.RUNNING：该状态下的线程池可以接收新的任务，并且处理阻塞队列中的任务。

2.SHUTDOWN：该状态下的线程池不接收新的任务，但是会处理阻塞队列中的任务。

3.STOP：该状态下的线程池不接收新的任务，也不会处理阻塞队列中的任务，并且会中断正在运行的任务。

4.TIDYING：该状态表示线程池对线程进行整理优化。

5.TERMINATED：该状态表示线程池停止工作。

ThreadPoolExecutor 继承了 AbstractExecutorService

AbstractExecutorService是一个抽象类，实现了ExecutorService接口

ExecutorService继承了Executor

- Executor是一个顶层接口，该接口中只声明了一个方法execute(Runnable)，返回值为void，参数是Runnable类型的，就是用来执行传入的任务。

- ExecutorService接口继承了Executor接口，并声明了一些方法：submit，invokeAll，shutDown。
- 抽象类AbstractExecutorService实现了ExecutorService接口，基本上实现了ExecutorService中声明的所有方法。
- ThreadPoolExecutor继承了AbstractExecutorService，ThreadPoolExecutor实现了全部的方法。

ThreadPoolExecutor类中常用的方法：

- execute()，是一个核心方法，通过该方法可以向线程池提交一个任务，由线程池去安排执行。
- submit()，ExecutorService接口中声明的方法，也是用来向线程池提交任务的，和execute()方法不同，可以返回执行任务的结果。
- shutdown()：关闭线程池，不会立即终止线程池，而是等待阻塞队列中的任务全部执行完毕之后在终止，同时也不会再接收新的任务。
- shutdownNow()：立即终止线程池，并中断正在执行的任务，并且清空阻塞队列，返回尚未执行的任务。

execute内部实现：

1.通过workCountof()方法获取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.如果能够成功的放入到阻塞队列中，若当前线程池是非RUNNING状态，则将该任务从阻塞队列中移除，然后执行reject()拒绝处理该任务。

如果当前线程池处于RUNNING状态，则需要再次检查线程池，如果有空闲的线程则执行该任务。

3.如果不能将任务放入阻塞队列中，说明阻塞队列已满，就通过addWorker()方法尝试创建新的线程来执行该任务，如果addWorker()方法失败，则表示当前线程池中的线程数已经达到了maxmumPoolSize，执行reject()拒绝处理该任务。

submit内部实现：

会将提交的任务封装成一个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。

总结

当提交一个任务时，线程池会创建一个新的线程去执行该任务，直到当前线程数等于corePoolSize；如果当前线程数为corePoolSize，继续提交的任务会被存入阻塞队列，等待被执行；如果阻塞队列满了，那就创建一个新的线程来执行当前任务，直到线程池中的线程数等于maxmumPoolSize，这时如果再有任务提交过来，只能执行reject()方法来拒绝处理。

实际开发中，不提倡直接实例化ThreadPoolExecutor，而是通过Executors类提供的静态方法来创建线程池：

1.创建一个固定大小的线程池，任务超过10个时，将任务放入阻塞队列中。

```
package com.southwind.thread;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class Test2 {
    public static void main(String[] args) {
        ExecutorService service =  Executors.newFixedThreadPool(10);
        for (int i = 0; i < 10; i++) {
            MyTask2 task = new MyTask2();
            service.execute(task);
        }
        service.shutdown();

//      for (int i = 0; i < 10; i++) {
//          MyTask2 task = new MyTask2();
//          new Thread(task).start();
//      }

    }
}

class MyTask2 implements Runnable{
    @Override
    public void run() {
        // TODO Auto-generated method stub
        try {
            Thread.currentThread().sleep(1000);
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
        System.out.println("...");
    }
}
```

2.创建一个缓存线程池。

newCachedThreadPool()，线程数可以达到Integer.MAX_VALUE，2147483647，内部使用SynchronousQueue作为阻塞队列。

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

3.创建单例线程池。

newSingleThreadExecutor()，线程池中只有一个线程。

```
public class Test3 {
    public static void main(String[] args) {
        ExecutorService sevice = Executors.newSingleThreadExecutor();
        for(int i = 0; i < 10; i++) {
            MyTask2 task = new MyTask2();
            sevice.execute(task);
        }
        sevice.shutdown();
    }
}
```

4.任务调度线程池。

newScheduledThreadPool()

```
public class Test4 {
    public static void main(String[] args) {
        ScheduledExecutorService service =
Executors.newScheduledThreadPool(10);
        //延迟5秒执行，每隔3秒执行一次任务
        service.scheduleAtFixedRate(new MyTask2(), 5, 3, TimeUnit.SECONDS);
        service.shutdown();
    }
}
```