java线程池

自己实现的一个线程池

IThreadPool

```
package pool;

import java.util.List;

/**
 * @author Mingming
 * @Description
 * @Date Created in 11:29 2018/2/22
 * @Modificd By
 */
public interface IThreadPool {
   //加入任务
   void execute(Runnable task);

   //加入任务
   void execute(Runnable[] tasks);

   //加入任务
   void execute(List<Runnable> tasks);

   //销毁线程
   void destroy();
}
```

ThreadPool

```
package pool;

import java.util.List;
import java.util.concurrent.BlockingDeque;
import java.util.concurrent.LinkedBlockingDeque;
import java.util.concurrent.atomic.AtomicLong;

/**
 * @author Mingming
```

```
 * @author Minglinlin
 * @Description
 * @Date Created in 11:20 2018/2/22
 * @Modificd By
 */
@SuppressWarnings("ALL")
public class ThradPool implements IThreadPool {
    /**
     * WORKER_NUMBER默认开启线程数
     */
    static int WORKER_NUMBER = 5;

    /**
     * sumCount完成任务线程数，可见性
     */
    static volatile int sumCount = 0;

    /**
     * taskQueue任务队列
     */
    BlockingDeque<Runnable> taskQueue = new
LinkedBlockingDeque<Runnable>();

    /**
     * 线程工作组
     */
    WorkerThread[] workerThreads;

    /**
     * 原子性
     */
    static AtomicLong threadNum = new AtomicLong();

    static ThradPool thradPool;

    public ThradPool() {
        this(WORKER_NUMBER);
    }

    public ThradPool(int workerNumber) {
        WORKER_NUMBER = workerNumber;
        //开辟工作空间
        workerThreads = new WorkerThread[WORKER_NUMBER];
        //开始创建工作线程
        for (int i = 0; i < WORKER_NUMBER; i++) {
            workerThreads[i] = new WorkerThread();
            Thread thread = new Thread(workerThreads[i], "Thread-Worker" +
threadNum.incrementAndGet());

            System.out.println("初始化线程数" + (i + 1) + thread.getName());
```

```
        System.out.println("初始化线程数" + (i + 1) + thread.getName());
        thread.start();
    }
}

public static IThreadPool getThreadPool() {
    return getThreadPool(WORKER_NUMBER);
}

public static IThreadPool getThreadPool(int workerNumber) {
    if (workerNumber <= 0) {
        workerNumber = WORKER_NUMBER;
    }
    if (thradPool == null) {
        thradPool = new ThradPool(workerNumber);
    }
    return thradPool;
}

@Override
public void execute(Runnable task) {
    try {
        taskQueue.put(task);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}

@Override
public void execute(Runnable[] tasks) {
    for (Runnable task : tasks
            ) {
        execute(task);
    }
}

@Override
public void execute(List<Runnable> tasks) {
    for (Runnable task : tasks
            ) {
        execute(task);
    }
}

@Override
public void destroy() {
    while (!taskQueue.isEmpty()) {
        System.out.println("等待....");

        try {
```

```
            try {
                Thread.sleep(20);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        for (int i = 0; i < WORKER_NUMBER; i++) {
            workerThreads[i].interrupt();
            workerThreads[i] = null;
        }
        thradPool = null;
        taskQueue.clear();
    }

    @Override
    public String toString() {
        return "工作线程数量为" + WORKER_NUMBER
                + "已完成的任务数" + sumCount +
                "等待任务数量" + taskQueue.size();
    }

    class WorkerThread extends Thread {

        /**
         * isRunning 表示当前线程属于活动可用状态
         */
        private boolean isRunning = true;

        public void setWorkerFlag() {
            isRunning = false;
        }

        @Override
        public void run() {
            Runnable runnable = null;
            try {
                runnable = taskQueue.take();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            if (runnable != null) {
                runnable.run();
            }
            sumCount++;
            runnable = null;
        }
    }
}
```

ThreadTest

```
package pool;


import java.util.ArrayList;
import java.util.List;

/**
 * @author Mingming
 * @Description
 * @Date Created in 12:10 2018/2/22
 * @Modificd By
 */
public class ThreadPoolTest {
    public static void main(String[] args) {
        //获取线程池
        IThreadPool t = ThradPool.getThreadPool(20);

        List<Runnable> taskList = new ArrayList<Runnable>();
        for (int i = 0; i < 10; i++) {
            taskList.add(new Task());
        }
        //执行任务
        t.execute(taskList);
        System.out.println(t);
        //销毁线程
        t.destroy();
        System.out.println(t);
    }

    static class Task implements Runnable {

        private static volatile int i = 1;

        @Override
        public void run() {
            System.out.println("当前处理的线程:" +
Thread.currentThread().getName() + " 执行任务" + (i++) + " 完成");
        }
    }
}
```

这个线程池的实现，有一个问题。就是所有的任务执行完成之后，不能终止线程。应该是在执行take方法的时候阻塞了。现在还没有找到解决方法。

Java中的ThreadPoolExecutor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。

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);
    ...
}
```

构造器中的参数含义

> - *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，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
```

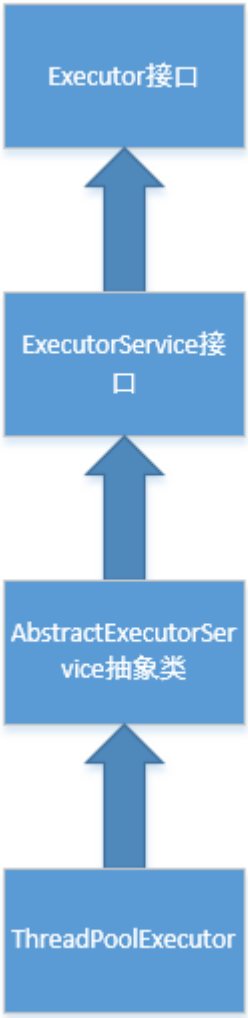
- *workQueue：一个阻塞队列，用来存储等待执行的任务，这个参数的选择也很重要，会对线程池的运行过程产生重大影响，一般来说，这里的阻塞队列有以下几种选择*

```
ArrayBlockingQueue;
LinkedBlockingQueue;
SynchronousQueue;
```

- *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
```

ThreadPoolExecutor的继承和实现关系



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。

```
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

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

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

shutdown()和shutdownNow()是用来关闭线程池的。

深入剖析线程池实现原理

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. 线程池关闭
7. 线程池容量的状态调整

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

在了解任务提交给线程池到任务执行完毕整个过程之前，先看一下ThreadPoolExecutor类中的一些比较重要的成员变量。

```
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 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。

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：

jdk1.8中的实现

```
public void execute(Runnable command) {
    if (command == null)
```

```
        if (command == null)
            throw new NullPointerException();
        int c = ctl.get();
        if (workerCountOf(c) < corePoolSize) {
            if (addWorker(command, true))
                return;
            c = ctl.get();
        }
        if (isRunning(c) && workQueue.offer(command)) {
            int recheck = ctl.get();
            if (! isRunning(recheck) && remove(command))
                reject(command);
            else if (workerCountOf(recheck) == 0)
                addWorker(null, false);
        }
        else if (!addWorker(command, false))
            reject(command);
    }
```

之前的实现：

```
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
    }
}
```

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。

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();
```

```
    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吃大小时执行的方法。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;
```

```
                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：

```
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 &&
```

```
            (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>=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，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()+"
```

```
            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.SingleThreadExecutor()//单个线程的线程池、即线程池中每次只有一个线程工作，单线程串行执行任务
Executors.newFixedThreadExecutor(n)//固定数量的线程池，每提交一个任务就是一个线程，直到达到线程池的最大数量，然后后面进入等待队列直到前面的任务完成。
Executors.newCacheThreadExecutor()//(推荐使用)可缓存线程池，当线程池超过了处理任务所需的线程，那么就会回收一部分空闲（一般是60秒无执行）的线程，当有任务来的时候又智能的添加线程执行。
```

Executors.newCachedThreadPool()

下面是这四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 neSingleThreadExecutor() {
    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。