

## Java并发编程：线程池的使用

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。

### 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。

下面解释一下构造器中各个参数的含义：

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，即在没有任何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;
```

`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。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。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()
```

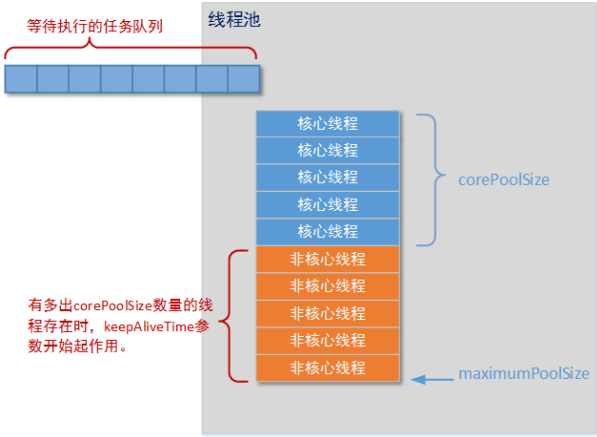
execute()方法实际上是Executor中声明的方法，在ThreadPoolExecutor进行了具体的实现，这个方法是ThreadPoolExecutor的核心方法，通过这个方法可以向线程池提交一个任务，交由线程池去执行。

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。

参数	说明
corePoolSize	线程池中核心线程数量
maximumPoolSize	线程池中最大线程数量
keepAliveTime	非核心线程存活时间
unit	keepAliveTime的时间单位
workQueue	存放任务的队列
threadFactory	用来生产线程的工厂
handler	当线程池中不能再放入任务时执行的handler

如果有一个corePoolSize为5，maximumPoolSize为10的线程池，可用下图形象展示：



这里要说明一下：所谓核心线程非核心线程只是一个数量的说明，并不是说核心线程非核心线程有本质上的不同，它们都是普通的线程而已，并且线程特性都一样，不是说核心线程有特殊标记，线程池能“认”出来这是核心线程，对其有特殊操作。

## 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。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。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：

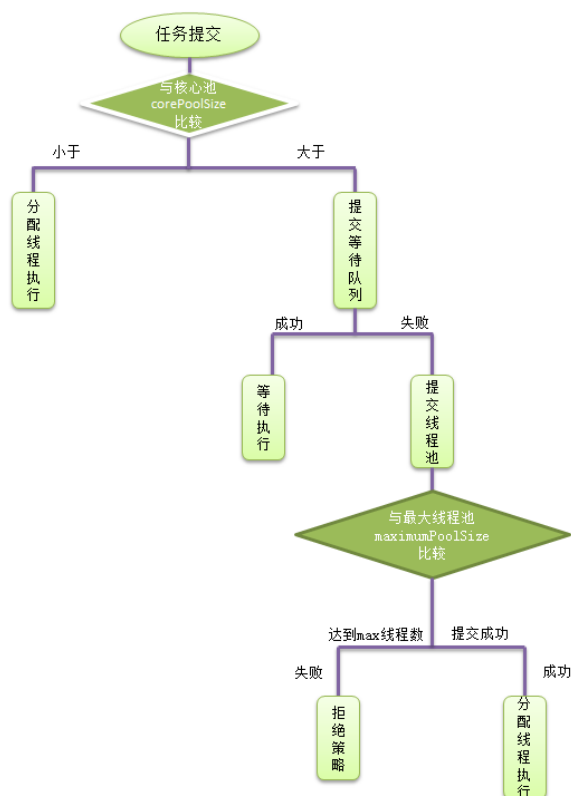
1) 首先，要清楚corePoolSize和maximumPoolSize的含义；

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 $\geq$ 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。

接下来用一个流程图来讲一讲，他究竟干了什么事。



### 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：





```

package com.demo.test;

import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

public class Test {

    public static void main(String[] args) {
        ThreadPoolExecutor executor = new ThreadPoolExecutor(
            5, //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）
            10, //线程池最大能容忍的线程数
            200, //线程存活时间
            TimeUnit.MILLISECONDS, //参数keepAliveTime的时间单位
            new ArrayBlockingQueue<Runnable>(5) //任务缓存队列，用来存放等待执行的任务
        );

        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();
    }
}

```



```

package com.demo.test;

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

### 几种常见的线程池

Executors 是提供了一组工厂方法用于创建常用的 ExecutorService，分别是 FixedThreadPool, CachedThreadPool 以及 SingleThreadExecutor。这三种ThreadPoolExecutor都是调用 ThreadPoolExecutor 构造函数进行创建，区别在于参数不同。

#### 1、FixedThreadPool - 线程池大小固定，任务队列无界

下面是 Executors 类 newFixedThreadPool 方法的源码：

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

可以看到 corePoolSize 和 maximumPoolSize 设置成了相同的值，此时不存在线程数量大于核心线程数量的情

况，所以KeepAlive时间设置不会生效。任务队列使用的是不限制大小的 `LinkedBlockingQueue`，由于是无界队列所以容纳的任务数量没有上限，因此，`FixedThreadPool`的行为如下：

1) 从线程池中获取可用线程执行任务，如果没有可用线程则使用`ThreadFactory`创建新的线程，直到线程数达到`nThreads`。

2) 线程池线程数达到`nThreads`以后，新的任务将被放入队列。

`FixedThreadPool`的优点是能够保证所有的任务都被执行，永远不会拒绝新的任务；同时缺点是队列数量没有限制，在任务执行时间无限延长的这种极端情况下会造成内存问题。

例子：

```
package com.demo.threadPool;

public class MyThread extends Thread {

    private Integer num; // 正在执行的任务数
    public MyThread(Integer num) {
        this.num = num;
    }

    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName()+" 正在执行第 "+ num + "个任务");
        try {
            Thread.sleep(500); // 模拟执行任务需要耗时
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println(Thread.currentThread().getName()+" 执行完毕第 " + num + "个任务");
    }
}
```

```
package com.demo.threadPool;

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

public class FixedThreadExecutorTest {

    public static void main(String[] args) {

        //创建固定大小的线程池
        ExecutorService executor=Executors.newFixedThreadPool(2);
        ThreadPoolExecutor threadPoolExecutor = (ThreadPoolExecutor) executor;

        for(int i = 1; i <= 5; i++) {
            Thread t = new MyThread(i);
            //将线程放到池中执行
            threadPoolExecutor.execute(t);
        }
    }
}
```

```

        System.out.println("线程池中现在的线程数目是：" + threadPoolExecutor.getPoolSize() + ", 队列中正在等待执行的任务数量为：" +
            threadPoolExecutor.getQueue().size());
    }

    //关闭线程池
    threadPoolExecutor.shutdown();
}

```



运行结果：



```

线程池中现在的线程数目是：1， 队列中正在等待执行的任务数量为：0
pool-1-thread-1 正在执行第 1个任务
线程池中现在的线程数目是：2， 队列中正在等待执行的任务数量为：0
pool-1-thread-2 正在执行第 2个任务
线程池中现在的线程数目是：2， 队列中正在等待执行的任务数量为：1
线程池中现在的线程数目是：2， 队列中正在等待执行的任务数量为：2
线程池中现在的线程数目是：2， 队列中正在等待执行的任务数量为：3
pool-1-thread-2 执行完毕第 2个任务
pool-1-thread-1 执行完毕第 1个任务
pool-1-thread-2 正在执行第 3个任务
pool-1-thread-1 正在执行第 4个任务
pool-1-thread-2 执行完毕第 3个任务
pool-1-thread-1 执行完毕第 4个任务
pool-1-thread-2 正在执行第 5个任务
pool-1-thread-2 执行完毕第 5个任务

```



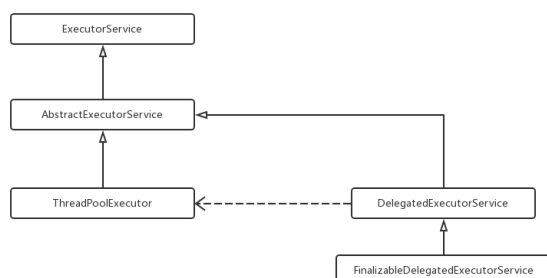
## 2、SingleThreadExecutor - 线程池大小固定为1，任务队列无界

```

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

```

这个工厂方法中使用无界LinkedBlockingQueue，并且将线程数设置成1，除此以外还使用FinalizableDelegatedExecutorService类进行了包装。这个包装类的主要目的是为了屏蔽ThreadPoolExecutor中动态修改线程数量的功能，仅保留ExecutorService中提供的方法。虽然是单线程处理，一旦线程因为处理异常等原因终止的时候，ThreadPoolExecutor会自动创建一个新的线程继续进行工作。



**SingleThreadExecutor 适用于在逻辑上需要单线程处理任务的场景，同时无界的LinkedBlockingQueue保证新任务都能够放入队列，不会被拒绝；缺点和FixedThreadPool相同，当处理任务无限等待的时候会造成内存问题。**

例子：



```
package com.demo.threadPool;

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

public class SingleThreadExecutorTest {

    public static void main(String[] args) {
        //创建一个单线程的线程池
        ExecutorService executor= Executors.newSingleThreadExecutor();

        for(int i = 1; i <= 5; i++) {
            Thread t = new MyThread(i);
            //将线程放到池中执行
            executor.execute(t);
        }

        //关闭线程池
        executor.shutdown();
    }
}
```



运行结果：



```
pool-1-thread-1 正在执行第 1个任务
pool-1-thread-1 执行完毕第 1个任务
pool-1-thread-1 正在执行第 2个任务
pool-1-thread-1 执行完毕第 2个任务
pool-1-thread-1 正在执行第 3个任务
pool-1-thread-1 执行完毕第 3个任务
pool-1-thread-1 正在执行第 4个任务
pool-1-thread-1 执行完毕第 4个任务
pool-1-thread-1 正在执行第 5个任务
pool-1-thread-1 执行完毕第 5个任务
```



### 3、CachedThreadPool - 线程池无限大（MAX INT），等待队列长度为1

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

SynchronousQueue是一个只有1个元素的队列，入队的任务需要一直等待直到队列中的元素被移出。核心线程数是0，意味着所有任务会先入队列；最大线程数是Integer.MAX\_VALUE，可以认为线程数量是没有限制



的。KeepAlive时间被设置成60秒，意味着在没有任务的时候线程等待60秒以后退

出。**CachedThreadPool对任务的处理策略是提交的任务会立即分配一个线程进行执行，线程池中线程数量会随着任务数的变化自动扩张和缩减，在任务执行时间无限延长的极端情况下会创建过多的线程。**

例子：



```
package com.demo.threadPool;

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

public class CachedThreadExecutorTest {

    public static void main(String[] args) {

        //创建一个可缓存的线程池
        ExecutorService executor=Executors.newCachedThreadPool();
        ThreadPoolExecutor threadPoolExecutor = (ThreadPoolExecutor) executor;

        for(int i = 1; i <= 5; i++) {
            Thread t = new MyThread(i);
            //将线程放到池中执行
            threadPoolExecutor.execute(t);
            System.out.println("线程池中现在的线程数目是：" + threadPoolExecutor.getPoolSize() + "， 队列中正在等待执行的任务数量为：" + threadPoolExecutor.getQueue().size());
        }

        //关闭线程池
        threadPoolExecutor.shutdown();
    }
}
```



运行结果：



```
线程池中现在的线程数目是：1， 队列中正在等待执行的任务数量为：0
pool-1-thread-1 正在执行第 1个任务
线程池中现在的线程数目是：2， 队列中正在等待执行的任务数量为：0
pool-1-thread-2 正在执行第 2个任务
线程池中现在的线程数目是：3， 队列中正在等待执行的任务数量为：0
pool-1-thread-3 正在执行第 3个任务
线程池中现在的线程数目是：4， 队列中正在等待执行的任务数量为：0
pool-1-thread-4 正在执行第 4个任务
线程池中现在的线程数目是：5， 队列中正在等待执行的任务数量为：0
pool-1-thread-5 正在执行第 5个任务
pool-1-thread-4 执行完毕第 4个任务
pool-1-thread-5 执行完毕第 5个任务
pool-1-thread-2 执行完毕第 2个任务
pool-1-thread-3 执行完毕第 3个任务
pool-1-thread-1 执行完毕第 1个任务
```



三种ExecutorService特性总结

类型	核心线程数	最大线程数	Keep Alive 时间	任务队列	任务处理策略
FixedThreadPool	固定大小	固定大小（与核心线程数相同）	0	LinkedBlockingQueue	线程池大小固定，
SingleThreadExecutor	1	1	0	LinkedBlockingQueue	与 FixedThreadF
CachedThreadPool	0	Integer.MAX_VALUE	1分钟	SynchronousQueue	线程池的数量无限

总结：

1. newSingleThreadExecutor

创建一个单线程的线程池。这个线程池只有一个线程在工作，也就是相当于单线程串行执行所有任务。如果这个唯一的线程因为异常结束，那么会有一个新的线程来替代它。此线程池保证所有任务的执行顺序按照任务的提交顺序执行。

2.newFixedThreadPool

创建固定大小的线程池。每次提交一个任务就创建一个线程，直到线程达到线程池的最大大小。线程池的大小一旦达到最大值就会保持不变，如果某个线程因为执行异常而结束，那么线程池会补充一个新线程。

3. newCachedThreadPool

创建一个可缓存的线程池。如果线程池的大小超过了处理任务所需要的线程，那么就会回收部分空闲（60秒不执行任务）的线程，当任务数增加时，此线程池又可以智能的添加新线程来处理任务。此线程池不会对线程池大小做限制，线程池大小完全依赖于操作系统（或者说JVM）能够创建的最大线程大小。

4.newScheduledThreadPool

创建一个大小无限的线程池。此线程池支持定时以及周期性执行任务的需求。

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。

补充：Java阻塞队列

队列以一种先进先出的方式管理数据，阻塞队列（BlockingQueue）是一个支持两个附加操作的队列，这两个附加的操作是：当从队列中获取或者移除元素时，如果队列为空，需要等待，直到队列不为空；同时如果向队列中添加元

素时，此时如果队列无可利用空间，也需要等待。在多线程进行合作时，阻塞队列是很有用的工具。

**生产者-消费者模式：**阻塞队列常用于生产者和消费者的场景，生产者线程可以定期的把中间结果存到阻塞队列中，而消费者线程把中间结果取出并在将来修改它们。队列会自动平衡负载，如果生产者线程集运行的比消费者线程集慢，则消费者线程集在等待结果时就会阻塞；如果生产者线程集运行的快，那么它将等待消费者线程集赶上来。

## Java中的阻塞队列

java.util.concurrent包提供了几种不同形式的阻塞队列，如数组阻塞队列ArrayBlockingQueue、链表阻塞队列LinkedBlockingQueue、优先级阻塞队列PriorityBlockingQueue和延时队列DelayQueue等，下面简单介绍一下这几个阻塞队列：

**数组阻塞队列：**ArrayBlockingQueue是一个由数组支持的有界阻塞队列，内部维持着一个定长的数据缓冲队列（该队列由数组构成），此队列按照先进先出（FIFO）的原则对元素进行排序，在构造时需要给定容量。ArrayBlockingQueue内部还保存着两个整形变量，分别标识着队列的头部和尾部在数组中的位置。

对于数组阻塞队列，可以选择是否需要公平性，所谓公平访问队列是指阻塞的所有生产者线程或消费者线程，当队列可用时，可以按照阻塞的先后顺序访问队列，即先阻塞的生产者线程，可以先往队列里插入元素，先阻塞的消费者线程，可以先从队列里获取元素。通常，公平性会使你在性能上付出代价，只有在的确非常需要的时候再使用它。

我们可以使用以下代码创建一个公平的阻塞队列：

```
ArrayBlockingQueue fairQueue = new ArrayBlockingQueue(1000, true);
```

数组阻塞队列的公平性是使用可重入锁实现的，其构造函数代码如下：

```
public ArrayBlockingQueue(int capacity, boolean fair) {
    if (capacity <= 0)
        throw new IllegalArgumentException();
    this.items = new Object[capacity];
    lock = new ReentrantLock(fair);
    notEmpty = lock.newCondition();
    notFull = lock.newCondition();
}
```

**链表阻塞队列：**LinkedBlockingQueue是基于链表的有界阻塞队列，内部维持着一个数据缓冲队列（该队列由链表构成），此队列按照先进先出的原则对元素进行排序。当生产者往队列中放入一个数据时，队列会从生产者手中获取数据，并缓存在队列内部，而生产者立即返回；只有当队列缓冲区达到最大值缓存容量时（可以通过LinkedBlockingQueue的构造函数指定该值），才会阻塞生产者队列，直到消费者从队列中消费掉一份数据，生产者线程将会被唤醒，反之对于消费者这端的处理也基于同样的原理。需要注意的是，如果构造一个LinkedBlockingQueue对象，而没有指定其容量大小，LinkedBlockingQueue会默认一个类似无限大小（Integer.MAX\_VALUE）的容量，这样的话，如果生产者的速度一旦大于消费者的速度，也许还没有等到队列满阻塞产生，系统内存就有可能已经被消耗殆尽了。

LinkedBlockingQueue之所以能够高效的处理并发数据，是因为其对于生产者端和消费者端分别采用了独立的锁来控制数据同步，这也意味着在高并发的情况下生产者和消费者可以并行地操作队列中的数据，以此来提高整个队列的并发性能。

**优先级阻塞队列：**PriorityBlockingQueue是一个支持优先级排序的无界阻塞队列，默认情况下元素采取自然顺序排列，也可以通过构造函数传入的Comparator对象来决定。在实现PriorityBlockingQueue时，内部控制线程同步的锁采用的是公平锁。需要注意的是PriorityBlockingQueue并不会阻塞数据生产者，而只是在没有可消费的数据时阻塞数据的消费者，因此使用的时候要特别注意，生产者生产数据的速度绝对不能快于消费者消费数据的速度，否则时

间一长，会最终耗尽所有的可用堆内存空间。

**延时队列：DelayQueue**是一个支持延时获取元素的使用**优先级队列**实现的**无界阻塞队列**。队列中的元素必须实现Delayed接口和Comparable接口（用以指定元素的顺序），也就是说**DelayQueue**里面的元素必须有public void compareTo(To)和long getDelay(TimeUnit unit)方法存在；在创建元素时可以指定多久才能从队列中获取当前元素，只有在延迟期满时才能从队列中提取元素。

**SynchronousQueue**：SynchronousQueue是一种无界、无缓冲的阻塞队列，可以认为SynchronousQueue是一个缓存值为1的阻塞队列，但是SynchronousQueue内部并没有数据缓存空间，数据是在配对的生产者和消费者线程之间直接传递的。可以这样来理解：SynchronousQueue是一个传球手，SynchronousQueue不存储数据元素，队列头元素是第一个排队要插入数据的线程，而不是要交换的数据，SynchronousQueue负责把生产者线程处理的数据直接传递给消费者线程，生产者和消费者互相等待对方，握手，然后一起离开。SynchronousQueue的吞吐量高于LinkedBlockingQueue 和 ArrayBlockingQueue。

## 任务阻塞队列

当线程池创建的线程数量大于 corePoolSize 后，新来的任务将会加入到堵塞队列（workQueue）中等待有空闲线程来执行。workQueue的类型为BlockingQueue，通常可以取下面三种类型：

- 1、ArrayBlockingQueue：基于数组的FIFO队列，是有界的，创建时必须指定大小
- 2、LinkedBlockingQueue：基于链表的FIFO队列，是无界的，默认大小是 Integer.MAX\_VALUE
- 3、synchronousQueue:一个比较特殊的队列，虽然它是无界的，但它不会保存任务，每一个新增任务的线程必须等待另一个线程取出任务，也可以把它看成容量为0的队列。

所有 BlockingQueue 都可用于传输和保持提交的任务。可以使用此队列与池大小进行交互：

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，则任务根本不会存放，添加到queue中，而是直接抄家伙（thread）开始运行）如果运行的线程等于或多于 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。

排队有三种通用策略：

**直接提交。**工作队列的默认选项是 SynchronousQueue，它将任务直接提交给线程而不保持它们。在此，如果不存在可用于立即运行任务的线程，则试图把任务加入队列将失败，因此会构造一个新的线程。此策略可以避免在处理可能具有内部依赖性的请求集时出现锁。直接提交通常要求无界 maximumPoolSizes 以避免拒绝新提交的任务。当命令以超过队列所能处理的平均数连续到达时，此策略允许无界线程具有增长的可能性。

**无界队列。**使用无界队列（例如，不具有预定义容量的 LinkedBlockingQueue）将导致在所有 corePoolSize 线程都忙时新任务在队列中等待。这样，创建的线程就不会超过 corePoolSize。（因此，maximumPoolSize 的值也就无效了。）当每个任务完全独立于其他任务，即任务执行互不影响时，适合于使用无界队列；例如，在 Web 页服务器中。这种排队可用于处理瞬态突发请求，当命令以超过队列所能处理的平均数连续到达时，此策略允许无界线程具有增长的可能性。

**有界队列。**当使用有限的 maximumPoolSizes 时，有界队列（如 ArrayBlockingQueue）有助于防止资源耗尽，但是可能较难调整和控制。队列大小和最大池大小可能需要相互折衷：使用大型队列和小型池可以最大限度地降低 CPU 使用率、操作系统资源和上下文切换开销，但是可能导致人工降低吞吐量。如果任务频繁阻塞（例如，如果它们是 I/O 边界），则系统可能为超过您许可的更多线程安排时间。使用小型队列通常要求较大的池大小，CPU 使用率较高，但是可能遇到不可接受的调度开销，这样也会降低吞吐量。

**BlockingQueue**的选择。

**例子一：使用直接提交策略，也即SynchronousQueue。**

首先SynchronousQueue是无界的，也就是说他存储任务的能力是没有限制的，但是由于该Queue本身的特性，在某

**次添加元素后必须等待其他线程取走后才能继续添加。**在这里不是核心线程便是新创建的线程，但是我们试想一样下，下面的场景。

我们使用一下参数构造ThreadPoolExecutor：

```
new ThreadPoolExecutor(
    2, 3, 30, TimeUnit.SECONDS,
    new SynchronousQueue<Runnable>(),
    new RecorderThreadFactory("CookieRecorderPool"),
    new ThreadPoolExecutor.CallerRunsPolicy());
```

当核心线程已经有2个正在运行。

- 1、此时继续来了一个任务（A），根据前面介绍的“如果运行的线程等于或多于 corePoolSize，则 Executor 始终首选将请求加入队列，而不添加新的线程。”，所以A被添加到queue中。
- 2、又来了一个任务（B），且核心2个线程还没有忙完，OK，接下来首先尝试1中描述，但是由于使用的SynchronousQueue，所以一定无法加入进去。
- 3、此时便满足了上面提到的“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。”，所以必然会新建一个线程来运行这个任务。
- 4、暂时还可以，但是如果这三个任务都还没完成，连续来了两个任务，第一个添加入queue中，后一个呢？queue中无法插入，而线程数达到了maximumPoolSize，所以只好执行异常策略了。

所以在使用SynchronousQueue通常要求maximumPoolSize是无界的，这样就可以避免上述情况发生（如果希望限制就直接使用有界队列）。对于使用SynchronousQueue的作用jdk中写的很清楚：**此策略可以避免在处理可能具有内部依赖性的请求集时出现锁。**

什么意思？如果你的任务A1，A2有内部关联，A1需要先运行，那么先提交A1，再提交A2，当使用SynchronousQueue我们可以保证，A1必定先被执行，在A1么有被执行前，A2不可能添加入queue中。

## 例子二：使用无界队列策略，即LinkedBlockingQueue

这个就拿newFixedThreadPool来说，根据前文提到的规则：

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。那么当任务继续增加，会发生什么呢？

如果运行的线程等于或多于 corePoolSize，则 Executor 始终首选将请求加入队列，而不添加新的线程。OK，此时任务变加入队列之中了，那什么时候才会添加新线程呢？

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。这里就很有意思了，可能会出现无法加入队列吗？不像SynchronousQueue那样有其自身的特点，对于无界队列来说，总是可以加入的（资源耗尽，当然另当别论）。换言之，永远也不会触发产生新的线程！corePoolSize大小的线程数会一直运行，忙完当前的，就从队列中拿任务开始运行。所以要防止任务疯长，比如任务运行的时间比较长，而添加任务的速度远远超过处理任务的时间，而且还不断增加，不一会儿就爆了。

## 例子三：有界队列，使用ArrayBlockingQueue。

这个是最为复杂的使用，所以JDK不推荐使用也有些道理。与上面的相比，最大的特点便是可以防止资源耗尽的情况发生。

举例来说，请看如下构造方法：

```
new ThreadPoolExecutor(
    2, 4, 30, TimeUnit.SECONDS,
    new ArrayBlockingQueue<Runnable>(2),
    new RecorderThreadFactory("CookieRecorderPool"),
```



```
new ThreadPoolExecutor.CallerRunsPolicy());
```

假设，所有的任务都永远无法执行完。

对于首先来的A,B来说直接运行，接下来，如果来了C,D，他们会被放到queue中，如果接下来再来E,F，则增加线程运行E，F。但是如果再来任务，队列无法再接受了，线程数也到达最大的限制了，所以就会使用拒绝策略来处理。

总结：

## BlockingQueue

**ArrayBlockingQueue**：基于数组的阻塞队列，在内部维护了一个定长数组，以便缓存队列中的数据对象。并没有实现读写分离，也就意味着生产和消费不能完全并行。是一个有界队列

**LinkedBlockingQueue**：基于列表的阻塞队列，在内部维护了一个数据缓冲队列（由一个链表构成），实现采用分离锁（读写分离两个锁），从而实现生产者 and 消费者操作的完全并行运行。是一个无界队列，

**SynchronousQueue**：没有缓存的队列，生存者生产的数据直接会被消费者获取并消费。若没有数据就直接调用出栈方法则会报错。

## 三种队列使用场景

**newFixedThreadPool** 线程池采用的队列是LinkedBlockingQueue。其优点是无界可缓存，内部实现读写分离，并发的处理能力高于ArrayBlockingQueue。

**newCachedThreadPool** 线程池采用的队列是SynchronousQueue。其优点就是无缓存，接收到的任务均可直接处理，再次强调，慎用！

并发量不大，服务器性能较好，可以考虑使用SynchronousQueue。

并发量较大，服务器性能较好，可以考虑使用LinkedBlockingQueue。

并发量很大，服务器性能无法满足，可以考虑使用ArrayBlockingQueue。系统的稳定最重要。

## 案例：

若有Thread1、Thread2、Thread3、Thread4四条线程分别统计C、D、E、F四个盘的大小，所有线程都统计完毕交给Thread5线程去做汇总，应当如何实现



```
package com.demo.threadPool;

import java.io.File;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.concurrent.Callable;
import java.util.concurrent.CompletionService;
import java.util.concurrent.ExecutorCompletionService;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

/**
 * 若有Thread1、Thread2、Thread3、Thread4四条线程分别统计C、D、E、F四个盘的大小，所有线程都统计完毕交给Thread5线程去做汇总，应当如何实现？
 * 思考：汇总，说明要把四个线程的结果返回给第五个线程，若要线程有返回值，推荐使用callable。Thread和Runnable都没返回值
 */
public class ITDragonThreads {

    public static void main(String[] args) throws Exception {
        // 无缓冲无界线程池
        ExecutorService executor = Executors.newFixedThreadPool(8);
```



```

// 相对ExecutorService, CompletionService可以更精确和简便地完成异步任务的执行
CompletionService<Long> completion = new ExecutorCompletionService<Long>(executor);
//Future<Long> f = null;
CountWorker countWorker = null;
//long total = 0;
for (int i = 0; i < 4; i++) { // 四个线程负责统计
    countWorker = new CountWorker(i);
    completion.submit(countWorker);
    //f = executor.submit(countWorker);
    //total += f.get();
}
// 关闭线程池
executor.shutdown();
// 主线程相当于第五个线程, 用于汇总数据
long total = 0;
for (int i = 0; i < 4; i++) {
    total += completion.take().get();
}
System.out.println(total / 1024 / 1024 / 1024 + "G");
}
}

class CountWorker implements Callable<Long>{
    private Integer type;
    public CountWorker() {
    }
    public CountWorker(Integer type) {
        this.type = type;
    }

    @Override
    public Long call() throws Exception {
        ArrayList<String> paths = new ArrayList<>(Arrays.asList("c:", "d:", "e:", "f:"));
        return countDiskSpace(paths.get(type));
    }

    // 统计磁盘大小
    private Long countDiskSpace (String path) {
        File file = new File(path);
        long totalSpace = file.getTotalSpace();
        System.out.println(path + " 总空间大小 : " + totalSpace / 1024 / 1024 / 1024 + "G");
        return totalSpace;
    }
}

```



运行结果：

```

e: 总空间大小 : 400G
f: 总空间大小 : 230G
d: 总空间大小 : 299G
c: 总空间大小 : 118G
1050G

```

分类: [Java多线程](#)

好文要顶

关注我

收藏该文



平凡希

关注 - 1

粉丝 - 353

+加关注

0

推荐

0

反对

« 上一篇：[Java多线程之ReentrantLock与Condition](#)

» 下一篇：[设计模式：抽象工厂模式](#)

posted @ 2017-10-24 14:36 平凡希 阅读(1186) 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](#)。**

【推荐】超50万VC++源码: 大型组态工控、电力仿真CAD与GIS源码库！

【福利】校园拼团福利，腾讯云1核2G云服务器10元/月！

【大赛】2018首届“顶天立地”AI开发者大赛

腾讯云

新注册用户域名抢购**1元起**

.com首年28元 .cn首年19元

立即抢购

.cn .com .xyz

#### 最新IT新闻:

- 摩拜单车宣布将在全国实行零门槛免押 发布摩拜助力车
  - 《我不是药神》大热 阿里影业大涨12% 北京文化连续3日涨停
  - 新一轮“黑车”治理开启 无资质网约车进入“严打周期”
  - Steam公布2018上半年畅销游戏榜：90款游戏上榜
  - 不让三星独美：LG开始准备折叠屏手机 国产厂商要跟进
- » 更多新闻...

#### 最新知识库文章:

- 从Excel到微服务
  - 如何提升你的能力？给年轻程序员的几条建议
  - 程序员的那些反模式
  - 程序员的宇宙时间线
  - 突破程序员思维
- » 更多知识库文章...

历史上的今天:  
2016-10-24 Spring系列之AOP实现的两种方式

昵称：平凡希  
园龄：7年6个月  
粉丝：353  
关注：1  
[+加关注](#)

2018年7月						
日	一	二	三	四	五	六
24	25	26	27	28	29	30
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	31	1	2	3	4

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](#)

随笔分类

[Java IO\(5\)](#)

[Java NIO\(7\)](#)

[java8\(3\)](#)

[Java多线程\(20\)](#)

java基础(11)
Java集合(8)
Java虚拟机(9)
Linux
Mybatis(9)
mysql(15)
redis(9)
spring(8)
SpringMVC(8)
设计模式(5)

随笔档案
2018年6月 (1)
2018年4月 (1)
2018年1月 (3)
2017年12月 (2)
2017年11月 (10)
2017年10月 (3)
2017年9月 (6)
2017年8月 (9)
2017年7月 (6)
2017年6月 (12)
2017年5月 (3)
2017年4月 (3)
2017年3月 (24)

2017年2月 (14)
2017年1月 (2)
2016年12月 (5)
2016年11月 (4)
2016年10月 (7)
2016年9月 (5)
2016年8月 (1)
2016年7月 (6)

最新评论

1. Re:Java中的值传递和引用传递	
我是一个java小白看您的博客写的确实很专业，但是我对这个引用传递有我一点看法，如果说引用传递是传递变量的地址算是值传递的话，那么指针也算是一种值传递吧，java中是没有指针的，但是传递地址实际上和指.....	
	---色即是空
2. Re:SpringMVC+Spring+Mybatis框架集成	
大神啊，已收藏	
	--lslb
3. Re:SpringMVC工作原理	
终于看懂了	
	--一位不愿透露姓名的网友
4. Re:SpringMVC+Spring+Mybatis框架集成	
mark下!谢谢分享	
	--wizard_Q
5. Re:SpringMVC工作原理	
nice	

阅读排行榜

- 1. SpringMVC工作原理(139623)
- 2. springmvc请求参数获取的几种方法(87383)
- 3. Spring系列之Spring常用注解总结(86852)
- 4. mysql 递归查询(41487)
- 5. 深入理解Java中的String(28612)

评论排行榜

- 1. Spring系列之Spring常用注解总结(16)
- 2. SpringMVC工作原理(11)
- 3. 深入理解Java中的String(10)
- 4. java集合框架综述(7)
- 5. springmvc请求参数获取的几种方法(6)

推荐排行榜

- 1. Spring系列之Spring常用注解总结(42)
- 2. SpringMVC工作原理(33)
- 3. 深入理解Java中的String(12)
- 4. java集合框架综述(10)
- 5. springmvc请求参数获取的几种方法(7)