[多线程 ForkJoinPool](https://www.cnblogs.com/lixuwu/p/7979480.html)

背景：ForkJoinPool的优势在于，可以充分利用多cpu，多核cpu的优势，把一个任务拆分成多个“小任务”，把多个“小任务”放到多个处理器核心上并行执行；当多个“小任务”执行完成之后，再将这些执行结果合并起来即可。这种思想值得学习。

Java7 提供了ForkJoinPool来支持将一个任务拆分成多个“小任务”并行计算，再把多个“小任务”的结果合并成总的计算结果。

ForkJoinPool是ExecutorService的实现类，因此是一种特殊的线程池。

使用方法：创建了ForkJoinPool实例之后，就可以调用ForkJoinPool的submit(ForkJoinTask<T> task) 或invoke(ForkJoinTask<T> task)方法来执行指定任务了。

其中ForkJoinTask代表一个可以并行、合并的任务。ForkJoinTask是一个抽象类，它还有两个抽象子类：RecusiveAction和RecusiveTask。其中RecusiveTask代表有返回值的任务，而RecusiveAction代表没有返回值的任务。

下面的UML类图显示了ForkJoinPool、ForkJoinTask之间的关系

Java7 提供了ForkJoinPool来支持将一个任务拆分成多个“小任务”并行计算，再把多个“小任务”的结果合并成总的计算结果。

ForkJoinPool是ExecutorService的实现类，因此是一种特殊的线程池。

使用方法：创建了ForkJoinPool实例之后，就可以调用ForkJoinPool的submit(ForkJoinTask<T> task) 或invoke(ForkJoinTask<T> task)方法来执行指定任务了。

其中ForkJoinTask代表一个可以并行、合并的任务。ForkJoinTask是一个抽象类，它还有两个抽象子类：RecusiveAction和RecusiveTask。其中RecusiveTask代表有返回值的任务，而RecusiveAction代表没有返回值的任务。

下面的UML类图显示了ForkJoinPool、ForkJoinTask之间的关系

A picture containing screenshot

Description automatically generated

因为我的电脑是i7处理器，一共8个cpu，观察线程的名称可以发现，8个cpu都在运行。

分析

在Java 7中引入了一种新的线程池：ForkJoinPool。

它同ThreadPoolExecutor一样，也实现了Executor和ExecutorService接口。它使用了一个无限队列来保存需要执行的任务，而线程的数量则是通过构造函数传入，如果没有向构造函数中传入希望的线程数量，那么当前计算机可用的CPU数量会被设置为线程数量作为默认值。

ForkJoinPool主要用来使用分治法(Divide-and-Conquer Algorithm)来解决问题。典型的应用比如快速排序算法。

这里的要点在于，ForkJoinPool需要使用相对少的线程来处理大量的任务。

比如要对1000万个数据进行排序，那么会将这个任务分割成两个500万的排序任务和一个针对这两组500万数据的合并任务。以此类推，对于500万的数据也会做出同样的分割处理，到最后会设置一个阈值来规定当数据规模到多少时，停止这样的分割处理。比如，当元素的数量小于10时，会停止分割，转而使用插入排序对它们进行排序。

那么到最后，所有的任务加起来会有大概2000000+个。问题的关键在于，对于一个任务而言，只有当它所有的子任务完成之后，它才能够被执行。

所以当使用ThreadPoolExecutor时，使用分治法会存在问题，因为ThreadPoolExecutor中的线程无法像任务队列中再添加一个任务并且在等待该任务完成之后再继续执行。而使用ForkJoinPool时，就能够让其中的线程创建新的任务，并挂起当前的任务，此时线程就能够从队列中选择子任务执行。

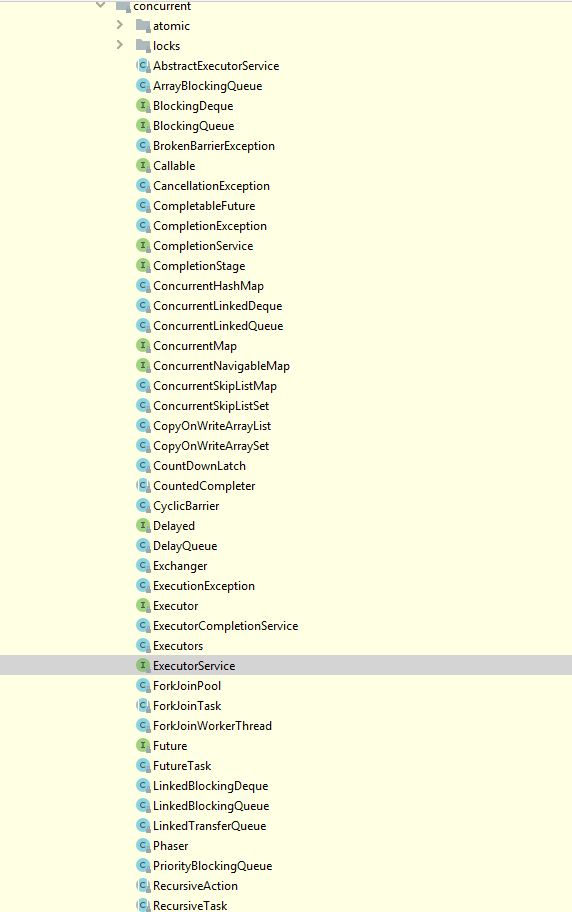
以上程序的关键是fork()和join()方法。在ForkJoinPool使用的线程中，会使用一个内部队列来对需要执行的任务以及子任务进行操作来保证它们的执行顺序。

那么使用ThreadPoolExecutor或者ForkJoinPool，会有什么性能的差异呢？

首先，使用ForkJoinPool能够使用数量有限的线程来完成非常多的具有父子关系的任务，比如使用4个线程来完成超过200万个任务。但是，使用ThreadPoolExecutor时，是不可能完成的，因为ThreadPoolExecutor中的Thread无法选择优先执行子任务，需要完成200万个具有父子关系的任务时，也需要200万个线程，显然这是不可行的。

ps：ForkJoinPool在执行过程中，会创建大量的子任务，导致GC进行垃圾回收，这些是需要注意的。

[Java并发编程之java.util.concurrent包下常见类的使用](https://www.cnblogs.com/miketwais/p/java_util_concurrent.html)



[\*\*线程池详解（包括Future和FutureTask）](https://blog.csdn.net/yancychas/article/details/97396366?utm_medium=distribute.pc_relevant.none-task-blog-BlogCommendFromMachineLearnPai2-9.nonecase&depth_1-utm_source=distribute.pc_relevant.none-task-blog-BlogCommendFromMachineLearnPai2-9.nonecase)

## Java 四种线程池的使用

<https://juejin.im/post/59df0c1af265da432f301c8d>

**1，线程池的作用**  
线程池作用就是限制系统中执行线程的数量。   
根据系统的环境情况，可以自动或手动设置线程数量，达到运行的最佳效果。   
少了浪费了系统资源，多了造成系统拥挤效率不高。   
用线程池控制线程数量，其他线程排 队等候。   
一个任务执行完毕，再从队列的中取最前面的任务开始执行。   
若队列中没有等待进程，线程池的这一资源处于等待。   
当一个新任务需要运行时，如果线程池 中有等待的工作线程，就可以开始运行了；否则进入等待队列。   
**2，为什么要用线程池?**  
1.减少了创建和销毁线程的次数，每个工作线程都可以被重复利用，可执行多个任务。   
2.可以根据系统的承受能力，调整线程池中工作线线程的数目，防止因为消耗过多的内存，而把服务器累趴下(每个线程需要大约1MB内存，线程开的越多，消耗的内存也就越大，最后死机)。   
Java里面线程池的顶级接口是Executor，但是严格意义上讲Executor并不是一个线程池，而只是一个执行线程的工具。真正的线程池接口是ExecutorService。   
**3，比较重要的几个类**

     类   
    描述

**ExecutorService**  
    真正的线程池接口。

**ScheduledExecutorService**  
    能和Timer/TimerTask类似，解决那些需要任务重复执行的问题。

**ThreadPoolExecutor**  
    ExecutorService的默认实现。

**ScheduledThreadPoolExecutor**  
    继承ThreadPoolExecutor的ScheduledExecutorService接口实现，周期性任务调度的类实现。

**4，new Thread的弊端**

1. public class TestNewThread {
3. public static void main(String[] args) {
4. new Thread(new Runnable() {
6. @Override
7. public void run() {
8. System.out.println("start");
9. }
10. }).start();
11. }
12. }

执行一个异步任务你还只是如下new Thread吗？   
那你就out太多了，new Thread的弊端如下：   
1.每次new Thread新建对象性能差。   
2.线程缺乏统一管理，可能无限制新建线程，相互之间竞争，及可能占用过多系统资源导致死机或oom。   
3.缺乏更多功能，如定时执行、定期执行、线程中断。   
相比new Thread，Java提供的四种线程池的好处在于：   
1.重用存在的线程，减少对象创建、消亡的开销，性能佳。   
2.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。   
3.提供定时执行、定期执行、单线程、并发数控制等功能。

**四种线程池**  
Java通过Executors提供四种线程池，分别为：   
**1,newCachedThreadPoo**  
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，若无可回收，则新建线程。   
**2,newFixedThreadPool**  
创建一个定长线程池，可控制线程最大并发数，超出的线程会在队列中等待。   
**3,newScheduledThreadPool**  
创建一个定长线程池，支持定时及周期性任务执行。   
**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。

下面我们再来分析一下submit()和execute()的以及shutdown()和shutdownNow()的区别

* submit()，提交一个线程任务，可以接受回调函数的返回值吗，适用于需要处理返回着或者异常的业务场景
* execute()，执行一个任务，没有返回值
* shutdown()，表示不再接受新任务，但不会强行终止已经提交或者正在执行中的任务
* shutdownNow()，对于尚未执行的任务全部取消，正在执行的任务全部发出interrupt()，停止执行

## 五种线程池的适应场景

* newCachedThreadPool：用来创建一个可以无限扩大的线程池，适用于服务器负载较轻，执行很多短期异步任务。
* newFixedThreadPool：创建一个固定大小的线程池，因为采用无界的阻塞队列，所以实际线程数量永远不会变化，适用于可以预测线程数量的业务中，或者服务器负载较重，对当前线程数量进行限制。
* newSingleThreadExecutor：创建一个单线程的线程池，适用于需要保证顺序执行各个任务，并且在任意时间点，不会有多个线程是活动的场景。
* newScheduledThreadPool：可以延时启动，定时启动的线程池，适用于需要多个后台线程执行周期任务的场景。
* newWorkStealingPool：创建一个拥有多个任务队列的线程池，可以减少连接数，创建当前可用cpu数量的线程来并行执行，适用于大耗时的操作，可以并行来执行

## RejectedExecutionHandler 线程池四种拒绝任务策略

《Java线程池》：任务拒绝策略

在没有分析线程池原理之前先来分析下为什么有任务拒绝的情况发生。

这里先假设一个前提：线程池有一个任务队列，用于缓存所有待处理的任务，正在处理的任务将从任务队列中移除。因此在任务队列长度有限的情况下就会出现新任务的拒绝处理问题，需要有一种策略来处理应该加入任务队列却因为队列已满无法加入的情况。另外在线程池关闭的时候也需要对任务加入队列操作进行额外的协调处理。

RejectedExecutionHandler提供了四种方式来处理任务拒绝策略

1、直接丢弃（DiscardPolicy）

2、丢弃队列中最老的任务(DiscardOldestPolicy)。

3、抛异常(AbortPolicy)

4、将任务分给调用线程来执行(CallerRunsPolicy)。

这四种策略是独立无关的，是对任务拒绝处理的四中表现形式。最简单的方式就是直接丢弃任务。但是却有两种方式，到底是该丢弃哪一个任务，比如可以丢弃当前将要加入队列的任务本身（DiscardPolicy）或者丢弃任务队列中最旧任务（DiscardOldestPolicy）。丢弃最旧任务也不是简单的丢弃最旧的任务，而是有一些额外的处理。除了丢弃任务还可以直接抛出一个异常（RejectedExecutionException），这是比较简单的方式。抛出异常的方式（AbortPolicy）尽管实现方式比较简单，但是由于抛出一个RuntimeException，因此会中断调用者的处理过程。除了抛出异常以外还可以不进入线程池执行，在这种方式（CallerRunsPolicy）中任务将有调用者线程去执行。

既然有四种拒绝策略可以选择，那么线程池的[默认拒绝策略](https://blog.csdn.net/suifeng629/article/details/98884972)是什么呢？查看

java.util.concurrent.ThreadPoolExecutor类的源码，我们可以看到：

*/\*\*  
 \* The default rejected execution handler  
 \*/***private static final** RejectedExecutionHandler ***defaultHandler*** =  
 **new** AbortPolicy();

线程池的默认拒绝策略为AbortPolicy，即丢弃任务并抛出RejectedExecutionException异常。

[Future与FutureTask的区别与联系](https://blog.csdn.net/f641385712/article/details/83546803?utm_medium=distribute.pc_relevant.none-task-blog-BlogCommendFromMachineLearnPai2-4.nonecase&depth_1-utm_source=distribute.pc_relevant.none-task-blog-BlogCommendFromMachineLearnPai2-4.nonecase)

Future模式简述

传统单线程环境下，调用函数是同步的，必须等待程序返回结果后，才可进行其他处理。 Futrue模式下，调用方式改为异步。

Futrue模式的核心在于：充分利用主函数中的等待时间，利用等待时间处理其他任务，充分利用计算机资源。

所谓异步调用其实就是实现一个可无需等待被调用函数的返回值而让操作继续运行的方法。在 Java 语言中，简单的讲就是另启一个线程来完成调用中的部分计算，使调用继续运行或返回，而不需要等待计算结果。但调用者仍需要取线程的计算结果。

JDK5新增了Future接口，用于描述一个异步计算的结果。虽然 Future 以及相关使用方法提供了异步执行任务的能力，但是对于结果的获取却是很不方便，只能通过阻塞或者轮询的方式得到任务的结果。阻塞的方式显然和我们的异步编程的初衷相违背，轮询的方式又会耗费无谓的 CPU 资源，而且也不能及时地得到计算结果。

### Future

Future就是对于具体的Runnable或者Callable任务的执行结果进行取消、查询是否完成、获取结果等操作。必要时可以通过get方法获取执行结果，该方法会阻塞直到任务返回结果。

Future类位于java.util.concurrent包下，它是一个接口：

从上面方法的注释可以看出，Futrue提供了三种功能：

1）判断任务是否完成；

2）能够中断任务；

3）能够获取任务执行结果。（最为常用的）

因为Future只是一个接口，所以是无法直接用来创建对象使用的，因此就有了下面的FutureTask（JDK8以前唯一实现类）。

### FutureTask

public class FutureTask<V> implements RunnableFuture<V>

public interface RunnableFuture<V> extends Runnable, Future<V>

由此看出FutureTask它既可以作为Runnable被线程执行，又可以作为Future得到Callable的返回值。

FutureTask提供了2个构造器：

* FutureTask实现了Runnable，因此它既可以通过Thread包装来直接执行，也可以提交给ExecuteService来执行。
* FutureTask实现了Futrue可以直接通过get()函数获取执行结果，该函数会阻塞，直到结果返回。

总结

Futrue的使用和FutrueTask的使用，没有本质的区别。所以…

<https://blog.csdn.net/xxj_jing/article/details/84835476>

<https://blog.csdn.net/u010994966/article/details/103188858>

<https://blog.csdn.net/suifeng629/article/details/98884972>

## Java-五种线程池，四种拒绝策略，三种阻塞队列

三种阻塞队列：

BlockingQueue<Runnable> workQueue = null;

workQueue = new ArrayBlockingQueue<>(5);//基于数组的先进先出队列，有界

workQueue = new LinkedBlockingQueue<>();//基于链表的先进先出队列，无界

workQueue = new SynchronousQueue<>();//无缓冲的等待队列，无界

四种拒绝策略：

RejectedExecutionHandler rejected = null;

rejected = new ThreadPoolExecutor.AbortPolicy();//默认，队列满了丢任务抛出异常

rejected = new ThreadPoolExecutor.DiscardPolicy();//队列满了丢任务不异常

rejected = new ThreadPoolExecutor.DiscardOldestPolicy();//将最早进入队列的任务删，之后再尝试加入队列

rejected = new ThreadPoolExecutor.CallerRunsPolicy();//如果添加到线程池失败，那么主线程会自己去执行该任务

五种线程池：

ExecutorService threadPool = null;

threadPool = Executors.newCachedThreadPool();//有缓冲的线程池，线程数 JVM 控制

threadPool = Executors.newFixedThreadPool(3);//固定大小的线程池

threadPool = Executors.newScheduledThreadPool(2);

threadPool = Executors.newSingleThreadExecutor();//单线程的线程池，只有一个线程在工作

threadPool = new ThreadPoolExecutor();//默认线程池，可控制参数比较多

【1】

<https://blog.csdn.net/u014378181/article/details/80940767?utm_medium=distribute.pc_relevant.none-task-blog-BlogCommendFromMachineLearnPai2-1.nonecase&depth_1-utm_source=distribute.pc_relevant.none-task-blog-BlogCommendFromMachineLearnPai2-1.nonecase>

## 常用的线程池模式以及不同线程池的使用场景

### newCachedThreadPool：

底层：返回ThreadPoolExecutor实例，corePoolSize为0；maximumPoolSize为Integer.MAX\_VALUE；keepAliveTime为60L；unit为TimeUnit.SECONDS；workQueue为SynchronousQueue(同步队列)

通俗：当有新任务到来，则插入到SynchronousQueue中，由于SynchronousQueue是同步队列，因此会在池中寻找可用线程来执行，若有可以线程则执行，若没有可用线程则创建一个线程来执行该任务；若池中线程空闲时间超过指定大小，则该线程会被销毁。

适用：执行很多短期异步的小程序或者负载较轻的服务器

### newFixedThreadPool：

底层：返回ThreadPoolExecutor实例，接收参数为所设定线程数量nThread，corePoolSize为nThread，maximumPoolSize为nThread；keepAliveTime为0L(不限时)；unit为：TimeUnit.MILLISECONDS；WorkQueue为：new LinkedBlockingQueue<Runnable>() 无界阻塞队列

通俗：创建可容纳固定数量线程的池子，每个线程的存活时间是无限的，当池子满了就不在添加线程了；如果池中的所有线程均在繁忙状态，对于新任务会进入阻塞队列中(无界的阻塞队列)

适用：执行长期的任务，性能好很多

### newSingleThreadExecutor:

底层：FinalizableDelegatedExecutorService包装的ThreadPoolExecutor实例，corePoolSize为1；maximumPoolSize为1；keepAliveTime为0L；unit为：TimeUnit.MILLISECONDS；workQueue为：new LinkedBlockingQueue<Runnable>() 无界阻塞队列

通俗：创建只有一个线程的线程池，且线程的存活时间是无限的；当该线程正繁忙时，对于新任务会进入阻塞队列中(无界的阻塞队列)

适用：一个任务一个任务执行的场景

### NewScheduledThreadPool:

底层：创建ScheduledThreadPoolExecutor实例，corePoolSize为传递来的参数，maximumPoolSize为Integer.MAX\_VALUE；keepAliveTime为0；unit为：TimeUnit.NANOSECONDS；workQueue为：new DelayedWorkQueue() 一个按超时时间升序排序的队列

通俗：创建一个固定大小的线程池，线程池内线程存活时间无限制，线程池可以支持定时及周期性任务执行，如果所有线程均处于繁忙状态，对于新任务会进入DelayedWorkQueue队列中，这是一种按照超时时间排序的队列结构

适用：周期性执行任务的场景

线程池任务执行流程：

当线程池小于corePoolSize时，新提交任务将创建一个新线程执行任务，即使此时线程池中存在空闲线程。

当线程池达到corePoolSize时，新提交任务将被放入workQueue中，等待线程池中任务调度执行

当workQueue已满，且maximumPoolSize>corePoolSize时，新提交任务会创建新线程执行任务

当提交任务数超过maximumPoolSize时，新提交任务由RejectedExecutionHandler处理

当线程池中超过corePoolSize线程，空闲时间达到keepAliveTime时，关闭空闲线程

当设置allowCoreThreadTimeOut(true)时，线程池中corePoolSize线程空闲时间达到keepAliveTime也将关闭

备注：

一般如果线程池任务队列采用LinkedBlockingQueue队列的话，那么不会拒绝任何任务（因为队列大小没有限制），这种情况下，ThreadPoolExecutor最多仅会按照最小线程数来创建线程，也就是说线程池大小被忽略了。

如果线程池任务队列采用ArrayBlockingQueue队列的话，那么ThreadPoolExecutor将会采取一个非常负责的算法，比如假定线程池的最小线程数为4，最大为8所用的ArrayBlockingQueue最大为10。随着任务到达并被放到队列中，线程池中最多运行4个线程（即最小线程数）。即使队列完全填满，也就是说有10个处于等待状态的任务，ThreadPoolExecutor也只会利用4个线程。如果队列已满，而又有新任务进来，此时才会启动一个新线程，这里不会因为队列已满而拒接该任务，相反会启动一个新线程。新线程会运行队列中的第一个任务，为新来的任务腾出空间。

这个算法背后的理念是：该池大部分时间仅使用核心线程（4个），即使有适量的任务在队列中等待运行。这时线程池就可以用作节流阀。如果挤压的请求变得非常多，这时该池就会尝试运行更多的线程来清理；这时第二个节流阀—最大线程数就起作用了。