链滴

Java 线程池

作者：DongXiaokai0819

原文链接：https://ld246.com/article/1571631298281

来源网站：链滴

线程池

为什么要用线程池

如果有大量的任务需要并发执行，但是每个任务只需要执行很短的时间就执行完成，这样就会频繁的建->销毁线程，这样反而会浪费系统资源。

那么有没有一种办法可以让线程执行完一个任务后，不进行销毁而是转去执行其它未完成的任务，这就可以实现线程复用，而不用频繁的创建销毁线程，把时间、资源都浪费了。这种方法就是线程池。

线程池所带来的好处

- 降低资源消耗。 通过重复利用已创建的线程降低线程创建和销毁造成的消耗。
- 提高响应速度。 当任务到达时，任务可以不需要的等到线程创建就能立即执行。
- 提高线程的可管理性。 线程是稀缺资源，如果无限制的创建，不仅会消耗系统资源，还会降低系统稳定性，使用线程池可以进行统一的分配，调优和监控。

实现一个我们自己的线程池

线程池需要什么特性呢?

1. 首先需要创建一个池子（容器），来盛放我们 **已经创建好**了的线程，并且容器内的线程可以保持直存活的状态。
2. 线程必须还得可以接受外面的任务，并执行它。
3. 如果任务数大于线程池中的线程数，需要有一个队列来保存任务。

线程池实现

```
public class TestThreadPool {
    // 线程池中默认线程的个数为5
    private static int WORK_NUM = 5;
    // 队列默认任务个数为100
    private static int TASK_COUNT = 100;

    // 工作线程组
    private WorkThread[] workThreads;

    // 阻塞任务队列，作为一个缓冲
    private final BlockingQueue<Runnable> taskQueue;
    private final int worker_num;//用户在构造这个池，希望的启动的线程数

    // 创建具有默认线程个数的线程池
    public TestThreadPool() {
        this(WORK_NUM,TASK_COUNT);
    }

    // 创建线程池,worker_num为线程池中工作线程的个数
    public TestThreadPool(int worker_num,int taskCount) {
```

```
    if (worker_num<=0)
        worker_num = WORK_NUM;//默认工作线程数量
    if(taskCount<=0)
        taskCount = TASK_COUNT;//默认任务队列容量

    this.worker_num = worker_num;
    taskQueue = new ArrayBlockingQueue<>(taskCount);
    workThreads = new WorkThread[worker_num];

    for(int i=0;i<worker_num;i++) {//按照用户指定的线程数量将线程全部启动
        workThreads[i] = new WorkThread();
        workThreads[i].start();
    }
    Runtime.getRuntime().availableProcessors();
}


// 执行任务,其实只是把任务加入任务队列，什么时候执行有线程池管理器决定
public void execute(Runnable task) {
    try {
        taskQueue.put(task);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

}


// 销毁线程池,该方法保证在所有任务都完成的情况下才销毁所有线程，否则等待任务完成才销毁
public void destroy() {
    // 工作线程停止工作，且置为null
    System.out.println("ready close pool.....");
    for(int i=0;i<worker_num;i++) {
        workThreads[i].stopWorker();
        workThreads[i] = null;//help gc
    }
    taskQueue.clear();// 清空任务队列
}

// 覆盖toString方法，返回线程池信息：工作线程个数和已完成任务个数
@Override
public String toString() {
    return "WorkThread number:" + worker_num
            + "  wait task number:" + taskQueue.size();
}

/**
 * 内部类，工作线程
 */
private class WorkThread extends Thread{

    @Override
    public void run(){
        Runnable r = null;
```

```
            try {
                while (!isInterrupted()) {//判断是否有中断请求
                    r = taskQueue.take();//如果队列中没有任务，则阻塞
                    if(r!=null) {
                        System.out.println(getId()+" ready exec :"+r);
                        r.run();//直接调用run()方法，本来就在一个线程内了
                    }
                    r = null;//help gc;
                }
            } catch (Exception e) {//抛出异常后，run()方法自然结束
                // TODO: handle exception
            }
        }

        public void stopWorker() {
            interrupt();
        }
    }
}
```

测试类

```
public class TestMyThreadPool {
    public static void main(String[] args) throws InterruptedException {
        // 创建3个线程的线程池
        TestThreadPool t = new TestThreadPool(3,0);//线程数量为3，任务数量默认
        t.execute(new MyTask("testA"));
        t.execute(new MyTask("testB"));
        t.execute(new MyTask("testC"));
        t.execute(new MyTask("testD"));
        t.execute(new MyTask("testE"));//扔了五个任务进去
        System.out.println(t);
        Thread.sleep(10000);
        t.destroy();// 所有线程都执行完成才destory
        System.out.println(t);
    }

    // 任务类
    static class MyTask implements Runnable {

        private String name;
        private Random r = new Random();

        public MyTask(String name) {
            this.name = name;
        }

        public String getName() {
            return name;
        }

        @Override
        public void run() {// 执行任务
            try {
```

```
                Thread.sleep(r.nextInt(1000)+2000);
            } catch (InterruptedException e) {
                System.out.println(Thread.currentThread().getName()+" sleep InterruptedException:

                        +Thread.currentThread().isInterrupted());
            }
            System.out.println("任务 " + name + " 完成");
        }
    }
}
```

输出如下：

```
12 ready exec :com.kk.线程池.ch6.mypool.TestMyThreadPool$MyTask@70b9b6e1
14 ready exec :com.kk.线程池.ch6.mypool.TestMyThreadPool$MyTask@19b542ea
13 ready exec :com.kk.线程池.ch6.mypool.TestMyThreadPool$MyTask@40f53ce9//因为线程池有三个线程，所以只能先执行三个任务
WorkThread number:3  wait task number:2
任务 testB 完成//只有线程池中的线程有空闲了，才可以从阻塞队列中取出新的任务，然后执行。
13 ready exec :com.kk.线程池.ch6.mypool.TestMyThreadPool$MyTask@4e32ca87
任务 testC 完成
14 ready exec :com.kk.线程池.ch6.mypool.TestMyThreadPool$MyTask@1772bfac
任务 testA 完成
任务 testE 完成
任务 testD 完成
ready close pool.....
WorkThread number:3  wait task number:0
```

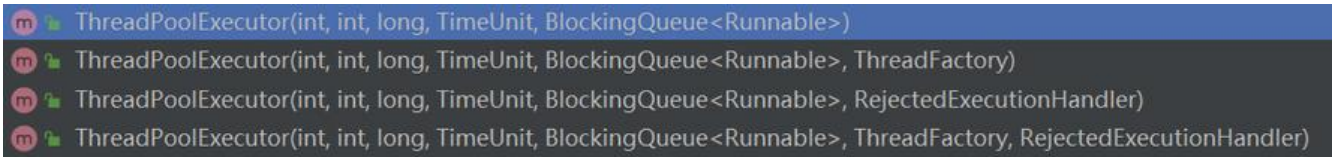
我们上述自己实现的线程池有什么问题

1. 当使用者提交的任务数量大于我们的线程池所能容纳的任务数量时，应该怎么办？

2. 我们实现的线程池，一开始就初始化了规定的线程数，而不能动态的调整，如果任务量小于线程数话，那么其它的线程就会毫无意义的阻塞着。

JDK中的线程池：ThreadPoolExecutor

ThreadPoolExecutor所提供的构造函数，以及各个参数的意义




```
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue,
                          ThreadFactory threadFactory,
                          RejectedExecutionHandler handler)
```

以下为各个参数的意义：

● int corePoolSize ： 线程池核心线程数

1. 线程池当前的线程数 < corePoolSize ，就会创建新线程
2. 线程池当前的线程数 = corePoolSize ，这个任务就会保存到BlockingQueue任务队列中去
3. 如果调用prestartAllCoreThreads()方法就会一次性的启动corePoolSize 个数的线程。

● int maximumPoolSize, 允许的最大线程数，如果线程池中的BlockingQueue任务队列也满了，线池当前线程数< maximumPoolSize时候就会再次创建新的线程，直到线程池当前线程数=maximumoolSize为止。

● long keepAliveTime, 当前线程数> corePoolSize时，线程空闲下来后，存活的时间。如果<corePolSize 即使线程空闲下来，线程也不会被杀死。

● TimeUnit unit, 存活时间的单位值

1. TimeUnit.DAYS; //天
2. TimeUnit.HOURS; //小时
3. TimeUnit.MINUTES; //分钟
4. TimeUnit.SECONDS; //秒
5. TimeUnit.MILLISECONDS; //毫秒
6. TimeUnit.MICROSECONDS; //微妙
7. TimeUnit.NANOSECONDS; //纳秒

● BlockingQueue <Runnable> workQueue, 保存任务的阻塞队列，ArrayBlockingQueue和PriorityBlockingQueue使用较少，一般使用LinkedBlockingQueue和Synchronous。线程池的排队策略与BockingQueue有关。

1. ArrayBlockingQueue; 基于数组的先进先出队列，有界
2. LinkedBlockingQueue; 基于链表的先进先出队列，无界
3. SynchronousQueue; 无缓冲的等待队列，无界
4. PriorityBlockingQueue; 基于堆的并发安全的优先级队列，无界

● ThreadFactory threadFactory, 创建线程的工厂，给新建的线程赋予名字

● RejectedExecutionHandler handler ： 当线程池的线程数达到了maximumPoolSize（所允许的大值），而且BlockingQueue任务队列也满了的情况下所采取的策略，即**饱和策略**。jdk提供了四种和策略。

1. AbortPolicy ： 直接抛出异常，默认；
2. CallerRunsPolicy：用调用者所在的线程来执行任务
3. DiscardOldestPolicy：丢弃阻塞队列里最老的任务，队列里最靠前的任务
4. DiscardPolicy ： 当前任务直接丢弃
5. 可根据实际需求，来实现自己的饱和策略，实现RejectedExecutionHandler接口即可

提交任务execute&submit

1. execute() 方法用于提交不需要返回值的任务，所以无法判断任务是否被线程池执行成功与否；

2. submit() 方法用于提交需要返回值的任务。线程池会返回一个Future类型的对象，通过这个Futur对象可以判断任务是否执行成功，并且可以通过future的get()方法来获取返回值，get()方法会阻塞前线程直到任务完成，而使用 get（long timeout，TimeUnit unit）方法则会阻塞当前线程一段时后立即返回，这时候有可能任务没有执行完。

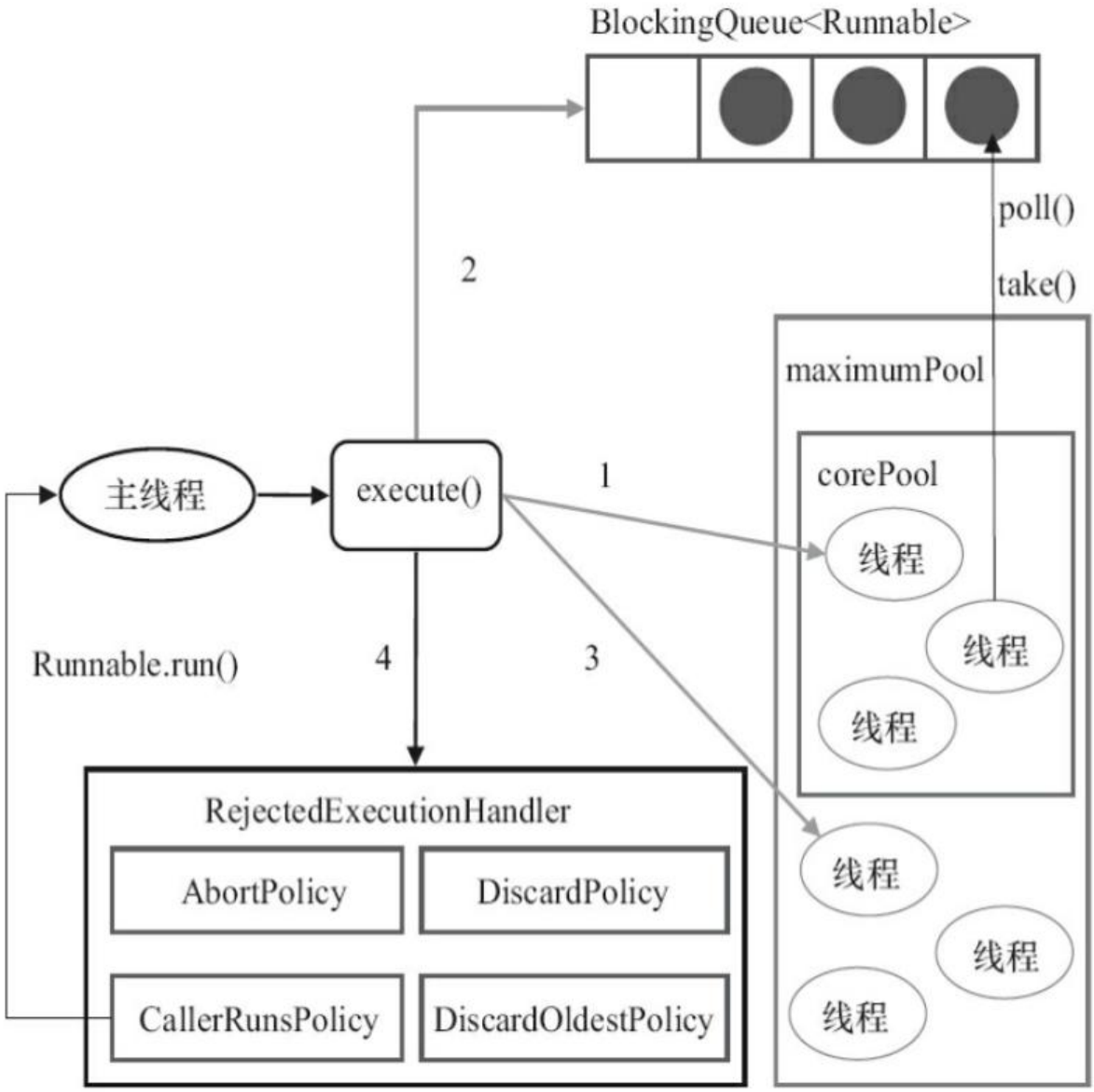
execute()方法提交任务的过程

ThreadPoolExecutor类中最核心的方法就是execute()提交方法，虽然submit()也可以提交方法，但submit()方法内部调用的还是execute()方法。

我们先看一下execute()方法的源码：

```
 public void execute(Runnable command) {
     if (command == null)
         throw new NullPointerException();
     int c = ctl.get();
     //ctl 变量被赋予了双重角色，通过高低位的不同，既表示线程池状态，又表示线程工作数目，这
一个典型的高效优化
     if (workerCountOf(c) < corePoolSize) {//如果当前线程数小于线程池核心线程数
         if (addWorker(command, true))//启动一个新的线程
             return;//成功直接返回
         c = ctl.get();
     }
     //isRunning()检查线程池是否被shuadown
     //workQueue.offer(command)往任务队列里面推任务，比较友好的入队方式，成功 return true
     if (isRunning(c) && workQueue.offer(command)) {
         int recheck = ctl.get();
         //再次进行防御性检查
         if (! isRunning(recheck) && remove(command))
             reject(command);
         else if (workerCountOf(recheck) == 0)
             addWorker(null, false);
     }
     //继续尝试添加worker，如果失败意味着已经饱和或者被shutdown
     else if (!addWorker(command, false))
         reject(command);//执行饱和策略
 }
```

下面是execute()方法的图示：



关闭线程池

- shutdownNow():设置线程池的状态为STOP，还会尝试停止正在运行或者暂停任务的线程
- shutdown()设置线程池的状态为SHUTDOWN，只会中断所有没有执行任务的线程

上述俩个方法都只是将线程池中所对应的线程进行中断请求，并不一定中断成功，所以说，Java中的程间都是相互协作的。

如何合理的配置线程池

根据任务的性质，可将任务分成三种：

- 计算密集型（CPU）
- IO密集型
- 混合型

计算密集型：加密，大数分解，正则.....，线程池的线程数应该适当设置的小一点，推荐线程池的最

线程数 = Cpu核心数+1

- 为什么+1，防止页缺失
- 机器的Cpu核心=Runtime.getRuntime().availableProcessors();

IO密集型：读取文件，数据库连接，网络通讯, 线程池线程数应适当大一点，推荐线程池的最大线程数 = 机器的Cpu核心数*2,

混合型：当俩种任务的规模差不多的时候，应该将混合型的任务尽量拆分成计算密集型和IO密集型，果某一种的任务规模远大于另外一种任务的规模的时候，拆分的意义不大。

队列的选择上，应该使用有界，无界队列可能会导致内存溢出，OOM

Java 并发类库提供的线程池有哪几种？ 分别有什么特点?

Executors 目前提供了 5 种不同的线程池创建配置：

- newCachedThreadPool()，它是一种用来处理大量短时间工作任务的线程池，具有几个鲜明特点它会试图缓存线程并重用，当无缓存线程可用时，就会创建新的工作线程；如果线程闲置的时间超过 0 秒，则被终止并移出缓存；长时间闲置时，这种线程池，不会消耗什么资源。其内部使用 SynchronusQueue 作为工作队列。
- newSingleThreadExecutor()，它的特点在于工作线程数目被限制为 1，操作一个无界的工作队列所以它保证了所有任务的都是被顺序执行，最多会有一个任务处于活动状态，并且不允许使用者改动程池实例，因此可以避免其改变线程数目。
- newFixedThreadPool(int nThreads)，重用指定数目（nThreads）的线程，其背后使用的是无界工作队列，任何时候最多有 nThreads 个工作线程是活动的。这意味着，如果任务数量超过了活动队数目，将在工作队列中等待空闲线程出现；如果有工作线程退出，将会有新的工作线程被创建，以补指定的数目 nThreads。
- newSingleThreadScheduledExecutor() 和 newScheduledThreadPool(int corePoolSize)，创建是个 ScheduledExecutorService，可以进行定时或周期性的工作调度，区别在于单一工作线程还是个工作线程。
- newWorkStealingPool(int parallelism)，基于ForkJoinPool实现的线程池，Java 8 才加入这个创方法，其内部会构建ForkJoinPool,利用Work-Stealing算法，并行的处理任务，不保证处理顺序

这里详细说一下俩个Scheduled线程池：

- newSingleThreadScheduledExecutor：只包含一个线程，只需要单个线程执行周期任务，保证顺的执行各个任务
- newScheduledThreadPool 可以包含多个线程的，线程执行周期任务，适度控制后台线程数量的候

方法说明：

```
public ScheduledFuture<?> schedule(Runnable command,
                                   long delay,
                                   TimeUnit unit)
```

schedule()方法，任务只执行一次，还可以延时执行。

```
public ScheduledFuture<?> scheduleAtFixedRate(Runnable command,
                                              long initialDelay,
```

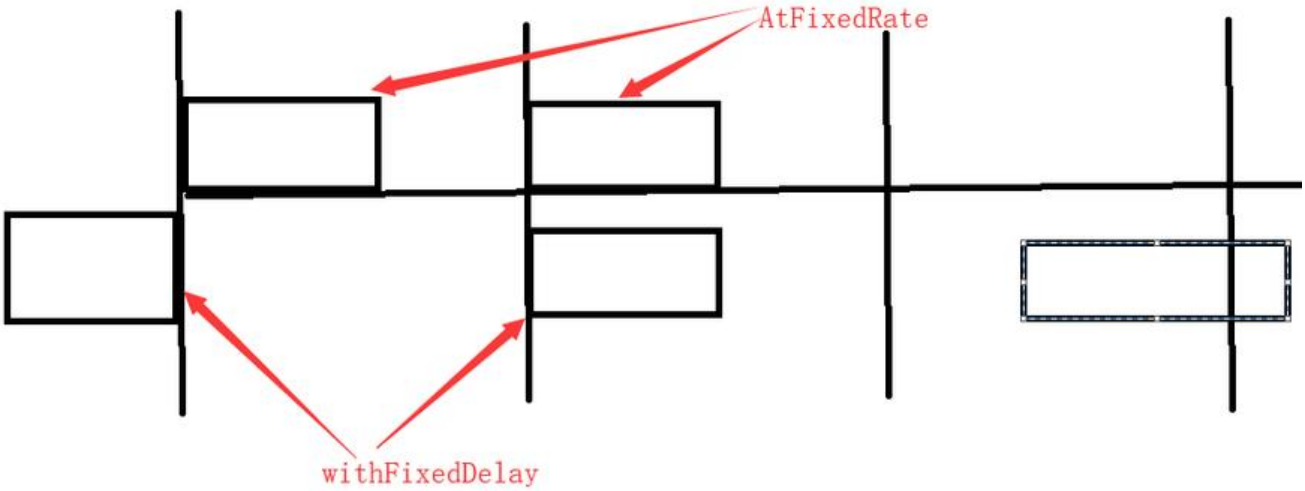
```
                                        long period,
                                        TimeUnit unit)
```

scheduleAtFixedRate() 提交固定时间间隔的任务

```
public ScheduledFuture<?> scheduleWithFixedDelay(Runnable command,
                                        long initialDelay,
                                        long delay,
                                        TimeUnit unit)
```

scheduleWithFixedDelay() 提交固定延时间隔执行的任务

scheduleAtFixedRate()与scheduleWithFixedDelay()的区别



scheduleAtFixedRate()任务超时怎么办?

不强制停止任务，若任务超时，则超时后也等待任务执行完成，然后等待的任务立即开始执行。

run()方法内出现异常怎么办?

建议在提交给ScheduledThreadPoolExecutor的任务一定要catch异常。如果没有try catch住，下次时任务就不会执行了，就挂掉了

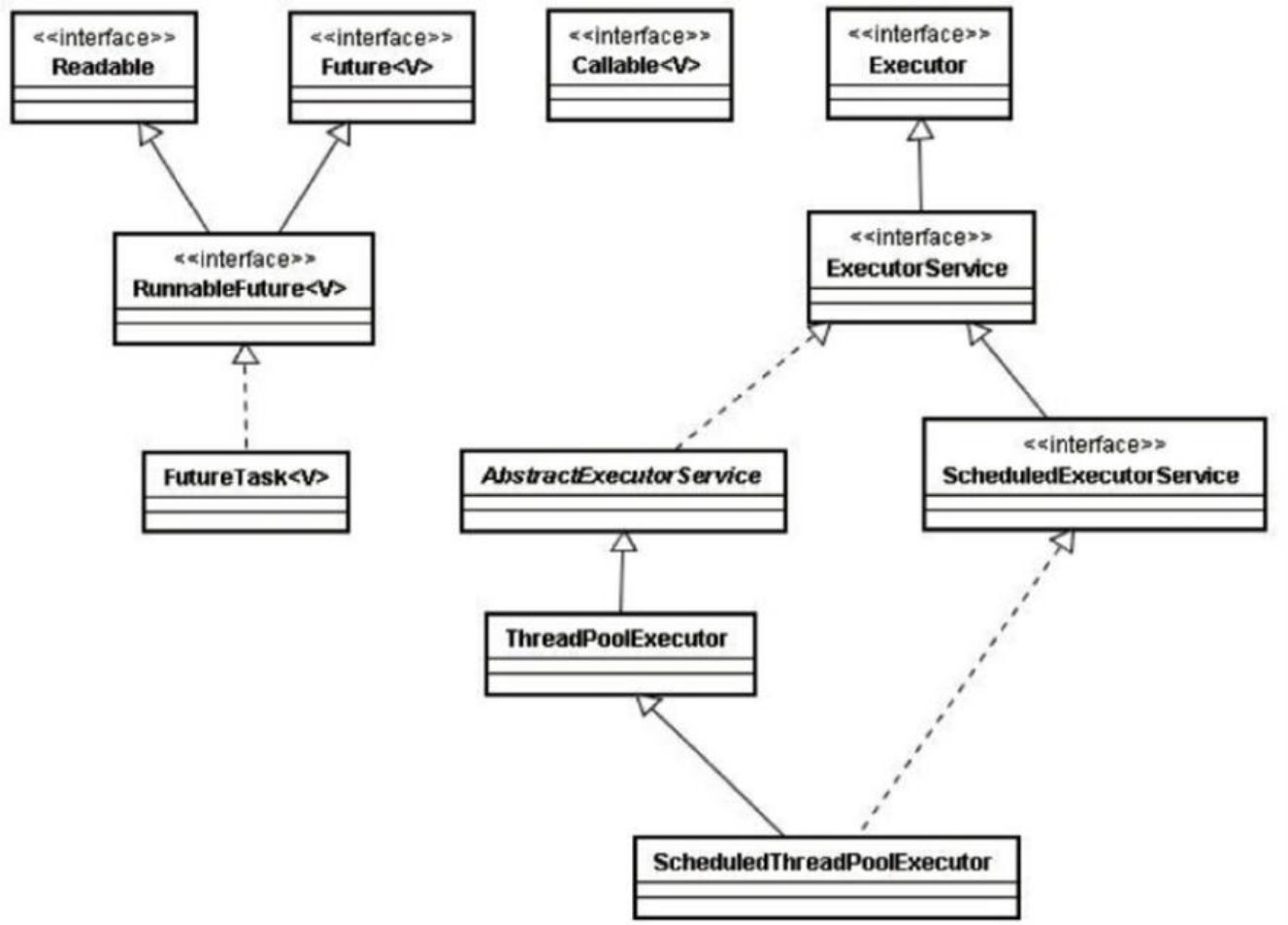
建议

《阿里巴巴Java开发手册》中强制线程池不允许使用 Executors 去创建，而是通过 ThreadPoolExecutor 的方式，这样的处理方式让写的同学更加明确线程池的运行规则，规避资源耗尽的风险

Executors 返回线程池对象的弊端如下:

- FixedThreadPool 和 SingleThreadExecutor： 允许请求的队列长度为 Integer.MAX_VALUE，能堆积大量的请求，从而导致OOM。
- CachedThreadPool 和 ScheduledThreadPool： 允许创建的线程数量为 Integer.MAX_VALUE 可能会创建大量线程，从而导致OOM。

Executor框架



● Executor 是一个基础接口，内部只提供了一个方法，如下，设计该接口的初衷是将任务提交与任执行细节解耦。

```
void execute(Runnable command);
```

● ExecutorService 则更加完善，不仅提供 service 的管理功能，比如 shutdown()、invokeAll() 等法，也提供了更加全面的提交任务机制，如返回Future而不是void的submit()方法

```
<T> Future<T> submit(Callable<T> task);
```

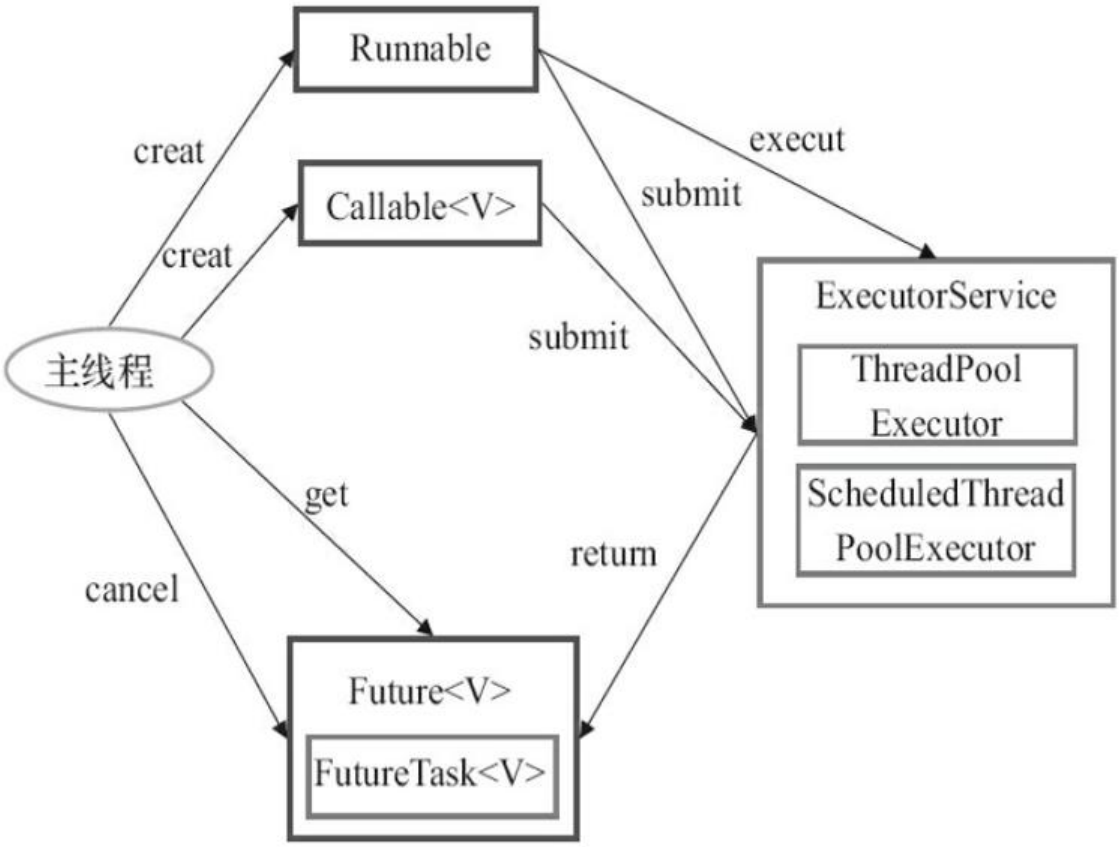
● AbstractExecutorService是一个抽象类，它实现了ExecutorService接口。

● Java标准类库提供了几种线程池实现：ThreadPoolExecutor、ScheduledThreadPoolExecutor、FrkJoinPool。上述三个线程池的设计特点主要是在于其高度的可调节性和灵活性，以尽量满足复杂多的应用场景。

● Executors 则从简化使用的角度，为我们提供了各种方便的创建几种已设计好的线程池的静态方法。

Executor框架基本使用流程

原文链接：Java 线程池



CompletionService

● CompletionService只是一个接口，来弥补Executor框架不能完全保证任务执行异步性的短板。

● CompletionService实际上可以看做是Executor和BlockingQueue的结合体。CompletionService接收到要执行的任务时，通过类似BlockingQueue的put和take获得任务执行的结果。CompletionSevice的一个实现是ExecutorCompletionService，ExecutorCompletionService把具体的计算任务交Executor完成。

场景

现在有个场景，就是有一堆需要返回值的任务，所以我们想用多线程来提高效率，我们想到了Executo框架的submit()方法，那么如何保存从异步任务中返回的值呢

这里我们用一个队列来存储，提交给线程池的任务（FutureTask）

```
BlockingQueue<Future<Integer>> queue = new LinkedBlockingQueue<Future<Integer>>();
```

然后我们像队列里面扔任务。

```
for (int i = 0; i < TOTAL_TASK; i++) {
    Future<Integer> future = pool.submit(new WorkTask("ExecTask" + i));
    queue.add(future);//i=0 先进队列，i=1的任务跟着进
}
```

但是这样解决的话，会有一个问题，我们从任务队列里面取出任务并get()这个任务的返回值时

```
queue.take().get()
```

get()方法会产生阻塞，我们没法保证先执行的任务先执行完成，所以这样做显然是不行的。

用CompletionService解决上述问题

我们只需要将创建的线程池传入CompletionService构造方法内（包装一下）。

```
CompletionService<Integer> completionService = new ExecutorCompletionService<>(pool);
```

然后我们可以completionService 既可以当成线程池pool来提交任务，也可以当成任务队列BlockingQueue来用。

```
completionService.submit(new WorkTask("ExecTask" + i));//提交任务
...
completionService.take().get();//获取返回结果
```

CompletionService在内部将先执行完成的任务结果放入BlockingQueue中，所以我们取的时候是按任务完成时间的顺序取结果的。

参考：

- 极客时间《Java核心技术36讲》 https://time.geekbang.org/column/intro/82
- https://blog.csdn.net/weixin_28760063/article/details/81266152
- https://snailclimb.top/JavaGuide/#/