1 什么是执行器框架（Executor FrameWork）

- 隔离任务的创建与运行的框架，使得开发任务可以专注于任务的逻辑

2 执行器框架的优势

- 通过使用执行器框架，开发者就可以专注于任务的逻辑，而不需要关心线程的创建、关闭等维护操作
- 执行器框架使用线程池中的线程来执行任务，可以避免不断的创建、销毁线程代理的性能消耗
- 同时，线程池框架提供了 Callable 接口，它可以使得任务有返回值

3 执行器框架的结构

类型	名称	描述
接口	Executor	最上层的接口，定义了执行任务的方法 execute
接口	ExecutorService	继承了 Executor 接口，新增了很多增强的执行任务的方法，引入了 Callable、Future 机制
抽象类	AbstractExecutorService	ExecutorService 接口的默认实现类，通过引入 RunnableFuture 接口适配了 callable、runnable 接口，使得callable 接口的实现类也可以被线程池执行
实现类	ThreadPoolExecutor	AbstractExecutorService 抽象类的实现类，是一个基础、标准的线程池实现
接口	ScheduledExecutorService	继承了ExecutorService 接口，增加了定时任务相关方法
实现类	ScheduledThreadPoolExecutor	继承了 ThreadPoolExecutor，实现了 ScheduledExecutorService中相关定时任务方法
执行器工具类	Executors	执行器的工厂方法和实用方法，创建 ExecutorService、ScheduleExecutorService、ThreadFactory、Callable 类
任务结果获取类	ExecutorCompletionService	通过它可以使得任务的发送和获取是不同的线程，并且优先获取已经完成的任务，就不会因为等待未完成的任务而阻塞了已完成的任务
实现类	ForkJoinPool	继承了 AbstractExecutorService 抽象类，专门用于 ForkJoinTask 的线程池

4 AbstractExecutorService 解析

```
public abstract class AbstractExecutorService implements ExecutorService {

    //异步方法，将 Runnable 类包装为 FutureTask 类，异步执行，立刻返回一个 Future 对象，且从Future 对象获取的结果肯定为 null
    public Future<?> submit(Runnable task) {
        if (task == null) throw new NullPointerException();
        RunnableFuture<Void> ftask = newTaskFor(task, null);//包装成新的 RUNNABLE 对象
        execute(ftask);//异步执行
        return ftask;
    }
    //异步方法，将 Runnable 类包装为 FutureTask 类，异步执行，立刻返回一个 Future 对象，且从Future 对象获取的结果肯定为 result
    public <T> Future<T> submit(Runnable task, T result) {
        if (task == null) throw new NullPointerException();
        RunnableFuture<T> ftask = newTaskFor(task, result);//包装成新的 RUNNABLE 对象
        execute(ftask);//异步执行
        return ftask;
    }
```

```
    //异步方法，将 Callable 类包装为 FutureTask 类，异步执行，立刻返回一个 Future 对象，且从
Future 对象获取的结果为 Callable 类执行的结果
    public <T> Future<T> submit(Callable<T> task) {
        if (task == null) throw new NullPointerException();
        RunnableFuture<T> ftask = newTaskFor(task);//将 Callable 对象包装成 RUNNABLE 对
象！！这样才能被 execute() 方法执行！！
        execute(ftask);//异步执行
        return ftask;
    }
    //同步方法，当 tasks 中的任务有任意一个结束时，就返回这个任务运行的结果
    public <T> T invokeAny(Collection<? extends Callable<T>> tasks) throws
InterruptedException, ExecutionException {...}
    //同步方法，当达到 timeout 时间时还没有任意任务返回则抛异常 ，否则返回第一个执行完任务的结果
    public <T> T invokeAny(Collection<? extends Callable<T>> tasks,long timeout,
TimeUnit unit)
throws InterruptedException, ExecutionException, TimeoutException {...}
    //同步方法，当 tasks 中的任务的所有任务都结束时，才返回所有任务对应的 Future 对象
    public <T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks)
throws InterruptedException{...}
    //同步方法，当达到 timeout 时间时任务还没有全部完成则抛异常，否则就返回所有任务对应的 Future 对
象
    public <T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks,long
timeout, TimeUnit unit) throws InterruptedException {...}

    //将 Runnable 类包装为 FutureTask 类，并设定这个 FutureTask 类的返回值固定为 value
    protected <T> RunnableFuture<T> newTaskFor(Runnable runnable, T value) {
        return new FutureTask<T>(runnable, value);
    }
    //将 callable 类包装为 FutureTask 类，并设定这个 FutureTask 类的返回值为 callable 的返回值
    protected <T> RunnableFuture<T> newTaskFor(Callable<T> callable) {
        return new FutureTask<T>(callable);
    }
    //几个 InvokeAny、InvokeAll 方法的核心执行逻辑
    private <T> T doInvokeAny(Collection<? extends Callable<T>> tasks,boolean timed,
long nanos){...}
}
```

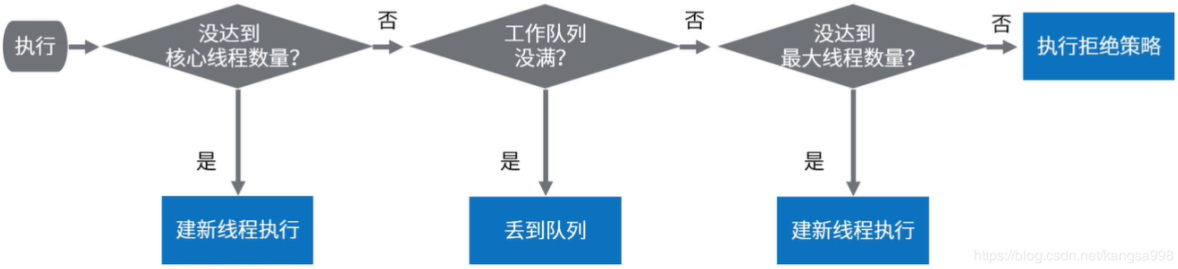
5 ThreadPoolExecutor 解析

5.1 线程池状态、线程池线程数 表示变量

```
//这个 ctl 变量很关键：它的高3位表示线程池运行的状态
//第 29 位表示线程池中的线程数
//111_*****_********_********_********：线程池为 RUNNING 状态
//000_*****_********_********_********：线程池为 SHUTDOWN 状态
//001_*****_********_********_********：线程池为 STOP 状态
//010_*****_********_********_********：线程池为 TIDYING 状态
//011_*****_********_********_********：线程池为 TERMINATED 状态
private final AtomicInteger ctl = new AtomicInteger(ctlOf(RUNNING, 0));
```

5.2 加入工作线程、工作队列、reject 的过程

- execute() 方法诠释了 加入工作线程、工作队列、reject 的逻辑
  - 如果线程池中的线程数少于核心线程数，尝试开启新的线程执行该任务。返回 true 说明任务成功被执行，返回 false，则进入后面逻辑
  - 如果线程数已经大于等于核心线程数，则新来的认为会直接尝试加入工作队列
  - 如果任务成功加入工作队列， double-check 线程池状态，防止加入任务后状态变化。如果变化了则移除任务，并走 reject 逻辑。如果没有变，但是线程数为0，则要新建一个线程
  - 如果等待队列满了，尝试开启新的线程执行，当线程数大于最大线程数时，走 reject 逻辑




```
//异步执行任务的逻辑，参数有且只有一个 Runnable 类型，说明只有 Runnable 对象才能被线程执行
//之前的 Callalbe 对象通过 FutureTask 对象包装成了 Runnable 对象，再运行的！
//所以，开启线程的方式只有一种，实现 Runnable 对象！
public void execute(Runnable command) {
    int c = ctl.get();
    //如果线程数小于核心线程数，则开启新的线程来执行 command
    if (workerCountOf(c) < corePoolSize) {
        if (addWorker(command, true))
            return;
        c = ctl.get();
    }
    //如果线程数已经大于核心线程，则先查看线程池是否处于 RUUNING 状态，如果是加入工作队列
    if (isRunning(c) && workQueue.offer(command)) {
        int recheck = ctl.get();
        //再次判断是否是 RUUNING 状态，如果不是，则移除该任务(防止加入成功后，状态变化了)，移除成功则走 reject 逻辑
        if (! isRunning(recheck) && remove(command))
            reject(command);
        else if (workerCountOf(recheck) == 0)//如果状态没有变，并且线程数为0，则新建一个线程
            addWorker(null, false);//保证有一个线程处于执行状态！
    }
    //如果等待队列满了，尝试开启新的线程执行，当线程数大于最大线程数时，走 reject 逻辑
    else if (!addWorker(command, false))
        reject(command);
}

//core：wc >= (core ? corePoolSize : maximumPoolSize)
//如果 core 为 false，并且 当前线程数达到了 最大线程数时，会返回 false，最终中 reject 逻辑
//如果 core 为 true 或 为 false 但还没有达到最大线程数时，会创建一个新的工作线程，最终这个新的工作线程会运行 runWorker() 方法！！
private boolean addWorker(Runnable firstTask, boolean core) {...}

final void runWorker(Worker w) {
    Thread wt = Thread.currentThread();
    Runnable task = w.firstTask;
    w.firstTask = null;
```

```
    w.unlock(); // allow interrupts
    boolean completedAbruptly = true;
    try {
        //关键是 getTask() 方法！！如果允许核心线程过期，则 getTask() 不会一直阻塞核心线程
        //如果不允许核心线程过期，则 getTask() 会一直阻塞核心线程，直到核心线程取到等待队列中的任务，
才返回！！！！！
        while (task != null || (task = getTask()) != null) {
            w.lock();
            try {
                beforeExecute(wt, task);
                Throwable thrown = null;
                try {
                    task.run();//去执行任务方法！！
                } finally {
                    afterExecute(task, thrown);
                }
            } finally {
                task = null;
                w.completedTasks++;
                w.unlock();
            }
        }
        completedAbruptly = false;
    } finally {
        processWorkerExit(w, completedAbruptly);
    }
}
```

5.3 线程状态处理几种方法

```
//使线程池的状态 CAS 为 SHUTDOWN，并且尝试中断所有工作线程
//此方法会立即返回，并不会等待已提交的任务都完成
//注意：如果线程不响应中断，则并没有效果
public void shutdown() {...}

//使线程池的状态 CAS 为 STOP，并强制中断所有工作线程，并返回被中断的线程
//注意：如果线程不响应中断，则并没有效果
public List<Runnable> shutdownNow(){...}

public boolean isShutdown(){...}//判断线程是否处于 RUNNING 状态
public boolean isTerminating(){...}//判断线程是否处于 正在 Terminating 但是没有 Terminated
的状态
public boolean isTerminated(){...}//判断线程是否处于 Terminated 的状态

//当超过 timeout 时，线程池还没有处于 TERMINATED 的状态，则返回 false，否则只要一旦到 TERMINATED
的状态 就返回 true
public boolean awaitTermination(long timeout, TimeUnit unit) throws
InterruptedException{..}
```

5.4 构造函数参数解析

变量	描述
corePoolSize	核心线程数
maximumPoolSize	最大线程数
keepAliveTime	线程空闲的最大时间，到了会被销毁
unit	keepAliveTime 对应的时间单位
workQueue	工作队列的类型，即阻塞队列的类型
threadFactory	线程工厂，用于创建线程
handler	执行 reject 的逻辑
allowCoreThreadTimeOut	使得核心线程也会过期，不在构造函数参数中！

5.5 拒绝策略解析

- 当线程数达到最大后，如果还有任务过了，就可能会被拒绝，这是就会走拒绝逻辑
- 如果在构造函数中没有指定 拒绝策略，则会使用默认拒绝策略：AbortPolicy
- ThreadPoolExecutor 也提供了另外 3 中拒绝策略
- 同时开发者可以自定义拒绝策略

```
//AbortPolicy：直接抛 Rejected 异常
public static class AbortPolicy implements RejectedExecutionHandler {}
//DiscardPolicy：什么都不做，任务也不会执行了，客户也不知道有个任务被拒绝了
public static class DiscardPolicy implements RejectedExecutionHandler {}
//减小工作队列的大小，重新执行该任务，这是该任务就可能放入工作队列中了
public static class DiscardOldestPolicy implements RejectedExecutionHandler {}
//转为同步执行该方法（当前线程直接执行 run() 方法）
public static class CallerRunsPolicy implements RejectedExecutionHandler {}
```

5.6 Worker 类解析

- Worker实现了 工作队列中任务的执行逻辑

```
private final class Worker extends AbstractQueuedSynchronizer implements Runnable {
    final Thread thread;// Worker 内资了一个 thread 用来执行 新的任务和工作队列中的任务

    Worker(Runnable firstTask) {
        setState(-1); //设置
        this.firstTask = firstTask;
        this.thread = getThreadFactory().newThread(this);//thread 中加入 Runnable 对象
（自身）
    }
    //通过 Worker 中的线程执行新的任务和工作队列中的任务
    public void run() {runWorker(this);}
}
//使 thread 工作线程  从队列中反复获取任务并执行它们
//不明白 为什么 要继承 AQS 来实现锁功能，不是只会由 它内部的 thread 线程才会去执行 runworker 嘛，为
什么还要加锁
```

```
final void runWorker(Worker w) {}
```

6 ScheduledThreadPoolExecutor 解析

6.1 ScheduledThreadPoolExecutor 常用方法解析

```
public class ScheduledThreadPoolExecutor extends ThreadPoolExecutor implements
ScheduledExecutorService {
    //设置延迟 delay 时间的定时任务
    public <V> ScheduledFuture<V> schedule(callable,delay,unit) {
        //
        ScheduleFutureTask s = ScheduleFutureTask<V>(callable,triggerTime(delay,unit));
        //decorateTask 默认返回 s
        RunnableScheduledFuture<V> t = decorateTask(callable,s);
        //关键方法！
        delayedExecute(t);
        return t;
    }
    //设置延迟 initialDelay 时间的定时任务，且每次任务完成后，过 period 时间又会重新执行
    public ScheduledFuture<?> scheduleAtFixedRate(command,initialDelay,period,unit) {
        ScheduledFutureTask<Void> sft =
            new ScheduledFutureTask<Void>(command,null,triggerTime(initialDelay,
unit),unit.toNanos(period));
        RunnableScheduledFuture<Void> t = decorateTask(command, sft);
        sft.outerTask = t;// outerTask 参数 保证了任务的 定期执行！
        delayedExecute(t);
        return t;
    }
    //设置延迟 initialDelay 时间的定时任务，并且每隔 delay 时间执行一次，不管上次任务有没有执行完
    public ScheduledFuture<?> scheduleWithFixedDelay(Runnable command,
                                                     long initialDelay,
                                                     long delay,
                                                     TimeUnit unit) {
        ScheduledFutureTask<Void> sft =
            new ScheduledFutureTask<Void>(command,
                                          null,
                                          triggerTime(initialDelay, unit),
                                          unit.toNanos(-delay));
        RunnableScheduledFuture<Void> t = decorateTask(command, sft);
        sft.outerTask = t;
        delayedExecute(t);
        return t;
    }

    //----------重写了 ThreadPoolExecutor 中的 execute 方法 和 AbstractExecutorService 中
的 submit 方法，所有方法内部都调用了 sechedule 方法，即创建一个 定时任务，并立即执行
    public void execute(Runnable command) {
        schedule(command, 0, NANOSECONDS);
    }
    public Future<?> submit(Runnable task) {
        return schedule(task, 0, NANOSECONDS);
    }
```

```
    public <T> Future<T> submit(Runnable task, T result) {
        return schedule(Executors.callable(task, result), 0, NANOSECONDS);
    }
    public <T> Future<T> submit(Callable<T> task) {
        return schedule(task, 0, NANOSECONDS);
    }

}
```

6.2 ScheduledThreadPoolExecutor 工作原理伪源码解析

```
private void delayedExecute(RunnableScheduledFuture<?> task) {
    if (isShutdown())
        reject(task);
    else {
        super.getQueue().add(task);
        //如果这时线程池 shutdown 了，并且设置了 shoudown 后不能执行的值，则不执行任务，直接退出
        if (isShutdown() &&
            !canRunInCurrentRunState(task.isPeriodic()) &&
            remove(task))
            task.cancel(false);
        else
            //关键
            ensurePrestart();
    }
}

void ensurePrestart() {
    int wc = workerCountOf(ctl.get());
    //走 addWorker 流程，和ThreadPoolExecutor 的区别在于 getTask() 方法中调用的 阻塞队列的出队列方法不同！！！
    if (wc < corePoolSize)
        addWorker(null, true);
    else if (wc == 0)
        addWorker(null, false);
}

// addWorker 内部方法调用流程
addWorker() {
    runWorker() {
        getTask() {
//这一行代码是关键：如果核心线程也可以失效则走 poll，否则走 take
//并且，SchedulePool 内部使用的是 DelayQueue，它的 poll 和 take 方法 是 实现 ScheulePool 的核心！
Runnable r = timed ?workQueue.poll(keepAliveTime, TimeUnit.NANOSECONDS) :
                     workQueue.take();
        }
    }
}

//你会发现，delayQueue 内部使用了 优先队列(下面的 q 参数)，来保证第一个节点的 delay 值是最小的！！
workQueue.take() {
    if (delay <= 0)
```

```
            return q.poll();//如果等待队列的第一个节点已经到期了，则执行 poll() 方法
        else
            await();//否则等待，这里有个关键的地方，第一个节点必须是等待时间最短的节点
    }

    //核心中的核心
    poll() {
        if (size == 0)
            return null;
        int s = --size;//队列中最后一个节点的索引
        modCount++;
        E result = (E) queue[0];
        E x = (E) queue[s];//去除最后一个节点
        queue[s] = null;//移除数组最后一个
        if (s != 0)
            //将 x 节点设置为第一个节点，并重新建堆（用对排序来保证第一个元素的 delay 值是最小
    的！！！！！！！）
            siftDown(0, x);
        return result;
    }
```

7 Executors 解析

方法	描述
newFixedThreadPool(nThreads, threadFactory)	创建固定线程数的线程池(无界队列)
newSingleThreadExecutor(threadFactory)	FixedPool nThreads 为 1的情况
newCachedThreadPool(threadFactory)	当任务到时如果有线程在等待任务则直接执行，如果没有则新建一个工作线程来执行(maxPoolSize 为最大值)
newScheduledThreadPool(corePoolSize, threadFactory)	定时、定期任务线程池
defaultThreadFactory()	返回一个默认的线程工厂
PrivilegedThreadFactory()	通过 java 的 AccessController.doPrivileged 创建 run() 方法
callable(Runnable task, T result)	通过 runnable 对象创建 callable 对象的方法

8 ExecutorCompletionService 解析

8.1 ExecutorCompletionService 核心

- 实现了 FutureTask 的子类，并重写了 done() 方法
- 当 Callable 任务完成后，会调用 set() 方法中的 done() 方法，最终将执行完的任务放入 completionQueue

```
private class QueueingFuture extends FutureTask<Void> {
    QueueingFuture(RunnableFuture<V> task) {
        super(task, null);
        this.task = task;
    }
    protected void done() { completionQueue.add(task); }
    private final Future<V> task;
}
```

8.2 构造函数解析

- 构造函数中必须传入一个 Executor 类，用来执行任务
- completionQueue 可以没有，默认为 LinkedBlockingQueue

```
public ExecutorCompletionService(Executor executor,
                                 BlockingQueue<Future<V>> completionQueue) {
    if (executor == null || completionQueue == null)
        throw new NullPointerException();
    this.executor = executor;
    this.aes = (executor instanceof AbstractExecutorService) ?
        (AbstractExecutorService) executor : null;
    this.completionQueue = completionQueue;
}
```

8.3 可用方法

```
public Future<V> submit(Callable<V> task) {
    if (task == null) throw new NullPointerException();
    RunnableFuture<V> f = newTaskFor(task);
    //关键：通过 QueueingFuture 保证了 RunnableFuture，使得可以调用 done() 方法
    executor.execute(new QueueingFuture(f));
    return f;
}
//主要是将 runnable 对象包装为 callable
public Future<V> submit(Runnable task, V result) {
    if (task == null) throw new NullPointerException();
    RunnableFuture<V> f = newTaskFor(task, result);
    executor.execute(new QueueingFuture(f));
    return f;
}

//------- 核心方法 -------这个类的实现也是为了这个方法服务的！
//通过这个方法，可以获取已完成的任务，这样就不会因为等待未完成的任务，而使得已完成任务一直没有被处理
public Future<V> take() throws InterruptedException {
    return completionQueue.take();
}
public Future<V> poll() {
    return completionQueue.poll();
}
public Future<V> poll(long timeout, TimeUnit unit)
    throws InterruptedException {
```

```
        return completionQueue.poll(timeout, unit);
    }
```

9 ForkJoinPool 解析

- 留待以后分析。。。

参考

jdk1.8u171

java 的 AccessController.doPrivileged使用