

2W 字带你，深入一下线程池

小林coding 2020-12-19

前言

线程池可以说是 Java 进阶必备的知识点，也是面试中必备的考点，可能不少人看了[这篇文章](#)后能对线程池工作原理说上一二，但这还远远不够，如果碰到比较有经验的面试官再继续追问，很可能被吊打，考虑如下问题：

1. Tomcat 的线程池和 JDK 的线程池实现有啥区别，Dubbo 中有类似 Tomcat 的线程池实现吗？
2. 我司网关 dubbo 调用线程池曾经出现过这样的问题：压测时接口可以正常返回，但接口 RT 很高，假设设置的核心线程大小为 500，最大线程为 800，缓冲队列为 5000，你能从这个设置中发现出一些问题并对这些参数进行调优吗？
3. 线程池里的线程真的有核心线程和非核心线程之分？
4. 线程池被 shutdown 后，还能产生新的线程？
5. 线程把任务丢给线程池后肯定就马上返回了？
6. 线程池里的线程异常后会再次新增线程吗，如何捕获这些线程抛出的异常？
7. 线程池的大小如何设置，如何**动态设置**线程池的参数
8. 线程池的状态机画一下？
9. 阿里 Java 代码规范为什么不允许使用 Executors 快速创建线程池？
10. 使用线程池应该避免哪些问题，能否简单说下线程池的最佳实践？
11. 如何优雅关闭线程池
12. 如何对线程池进行监控

这...这就触及到
..我的知识盲区了

相信不少人看了这些问题会有些懵逼



其实这些问题的答案大多数都藏在线程池的源码里，所以深入了解线程池的源码非常重要，本章我们将会来学习一下线程池的源码，相信看完之后，以上的问题大部分都能回答，另外一些问题我们也会在文中与大家一起探讨。

本文将会从以下几个方面来介绍线程池的原理。

1. 为什么要用线程池
2. 线程池是如何工作的
3. 线程池提交任务的两种方式
4. ThreadPoolExecutor 源码剖析
5. 解答开篇的问题
6. 线程池的最佳实践
7. 总结

相信大家看完对线程池的理解会更进一步，肝文不易，看完别完了三连哦。

为什么要用线程池

在[上文](#)也提到过，创建线程有三大开销，如下：

1、其实 Java 中的线程模型是基于操作系统原生线程模型实现的，也就是说 Java 中的线程其实是基于内核线程实现的，线程的创建，析构与同步都需要进行系统调用，而系统调用需要在用户态与内核中来回切换，代价相对较高，线程的生命周期包括「线程创建时间」，「线程执行任务时间」，「线程销毁时间」，创建和销毁都需要导致系统调用。2、每个 Thread 都需要有一个内核线程的支持，也就意味着每个 Thread 都需要消耗一定的内核资源（如内核线程的栈空间），因此能创建的 Thread 是有限的，默认一个线程的线程栈大小是 1 M，有图有真相

```
➔ ~ java -XX:+UnlockDiagnosticVMOptions -XX:NativeMemoryTracking=summary -XX:+PrintNMTStatistics -version
java version "1.8.0_181"
Java(TM) SE Runtime Environment (build 1.8.0_181-b13)
Java HotSpot(TM) 64-Bit Server VM (build 25.181-b13, mixed mode)

Native Memory Tracking:

Total: reserved=5704579KB, committed=460271KB
-   Java Heap (reserved=4194304KB, committed=262144KB)
      (mmap: reserved=4194304KB, committed=262144KB)
-   Class (reserved=1066074KB, committed=13786KB)
      (classes #352)
      (malloc=9306KB #130)
      (mmap: reserved=1056768KB, committed=4480KB)
-   Thread (reserved=19535KB, committed=19535KB)
      (thread #19)
      (stack: reserved=19456KB, committed=19456KB)
      (malloc=57KB #105)
      (arena=22KB #34)
-   Code (reserved=249629KB, committed=2565KB)
      (malloc=29KB #283)
      (mmap: reserved=249600KB, committed=2536KB)
```

图中所示，在 Java 8 下，创建 19 个线程（thread #19）需要创建 19535 KB，即 1 M 左右，reserved 代表如果创建 19 个线程，操作系统保证会为其分配这么多空间（实际上并不一定分配），committed 则表示实际已分配的空间大小。

画外音：注意，这是在 Java 8 下的线程占用空间情况，但在 Java 11 中，对线程作了很大的优化，创建一个线程大概只需要 40 KB，空间消耗大大减少

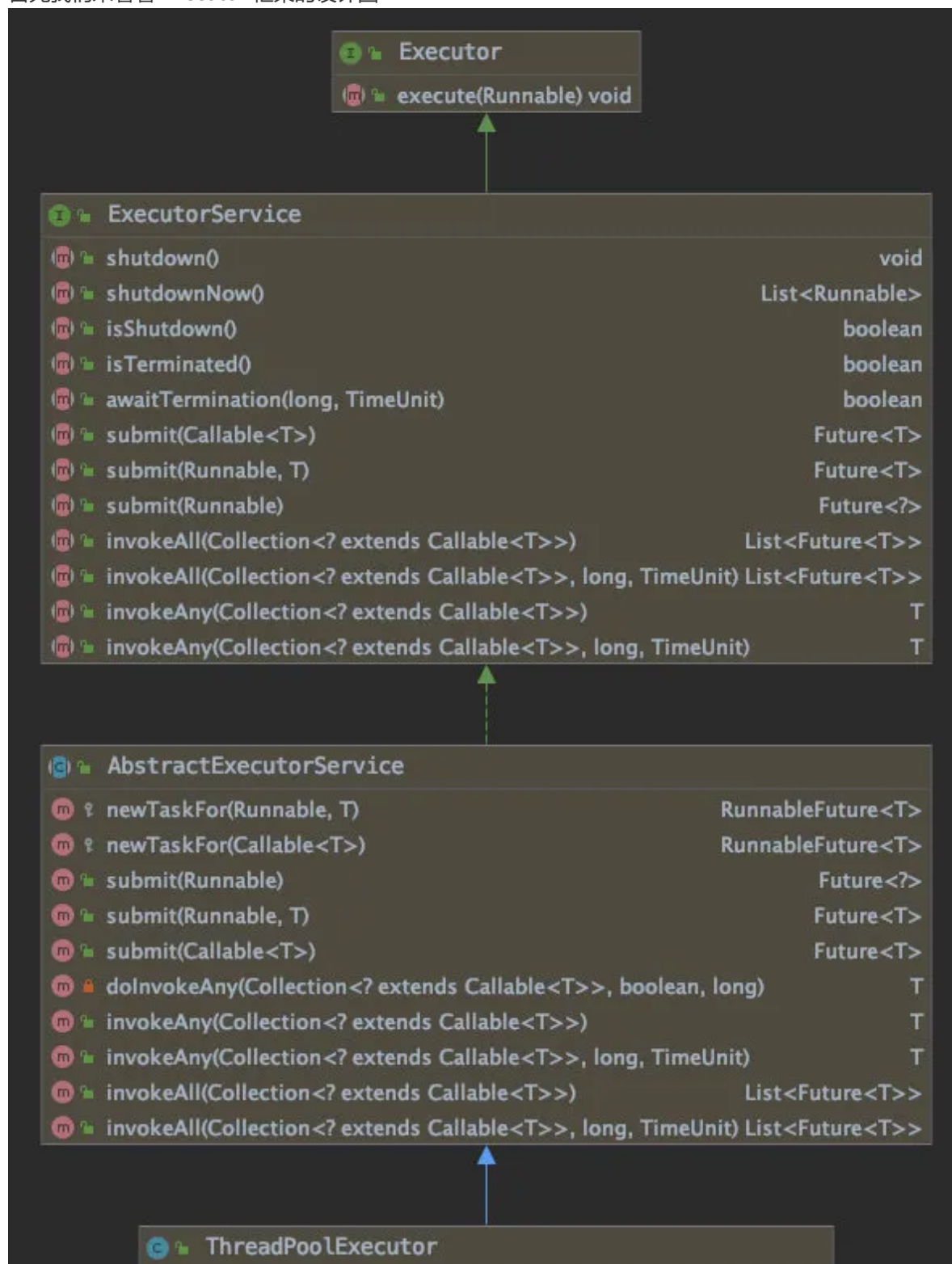
3、线程多了，导致不可忽视的上下文切换开销。

由此可见，线程的创建是昂贵的，所以必须以线程池的形式来管理这些线程，在线程池中合理设置线程大小和管理线程，以达到**合理的创建线程大小以达到最大化收益，最小化风险的目的**，对于开发人员来说，要完成任务不用关心线程如何创建，如何销毁，如何协作，只需要关心提交的任务何时完成即可，对线程的调优，监控等这些细枝末节的工作通通交给线程池来实现，所以也让开发人员得到极大的解脱！

类似线程池的这种池化思想应用在很多地方，比如数据库连接池，Http 连接池等，避免了昂贵资源的创建，提升了性能，也解放了开发人员。

ThreadPoolExecutor 设计架构图

首先我们来看看 Executor 框架的设计图



- **Executor**: 最顶层的 **Executor** 接口只提供了一个 `execute` 接口，实现了提交任务与执行任务的解耦，这个方法是最核心的，也是我们源码剖析的重点，此方法最终是由 **ThreadPoolExecutor** 实现的，
- **ExecutorService** 扩展了 **Executor** 接口，实现了终止执行器，单个/批量提交任务等方法
- **AbstractExecutorService** 实现了 **ExecutorService** 接口，实现了除 `execute` 以外的所有方法，只将一个最重要的 `execute` 方法交给 **ThreadPoolExecutor** 实现。

这样的分层设计虽然层次看起来挺多，但每一层各司其职，逻辑清晰，值得借鉴。

线程池是如何工作的

首先我们来看下如何创建一个线程池

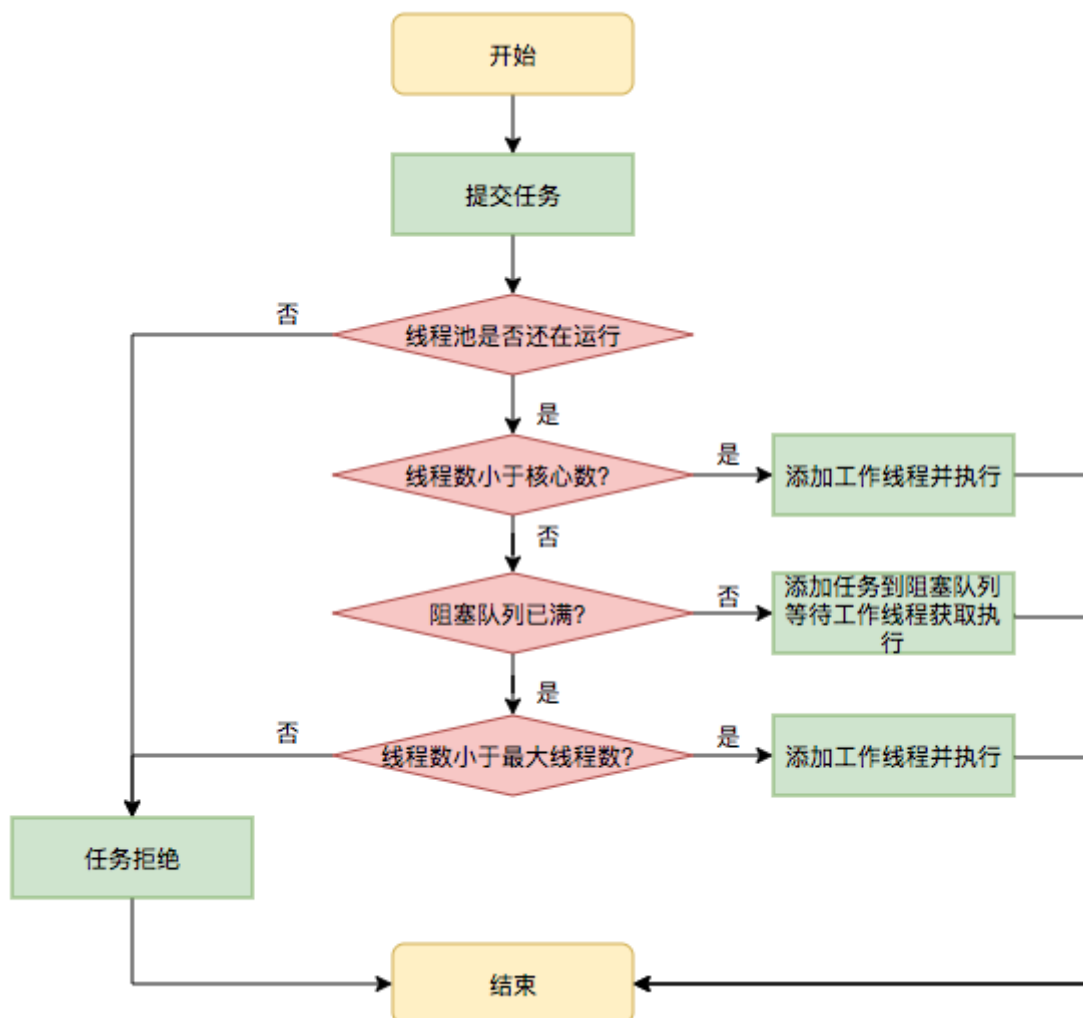
```
ThreadPoolExecutor threadPool = new ThreadPoolExecutor(10, 20, 600L,
    TimeUnit.SECONDS, new LinkedBlockingQueue<>(4096),
    new NamedThreadFactory("common-work-thread"));
// 设置拒绝策略，默认为 AbortPolicy
threadPool.setRejectedExecutionHandler(new ThreadPoolExecutor.AbortPolicy());
```

看下其构造方法签名如下

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

    // 省略代码若干
}
```

要理解这些参数具体代表的意义，必须清楚线程池提交任务与执行任务流程，如下



图片来自美团技术团队

步骤如下

1、corePoolSize: 如果提交任务后线程还在运行，当线程数小于 corePoolSize 值时，**无论线程池中的线程是否忙碌**，都会创建线程，并把任务交给此新创建的线程进行处理，如果线程数少于等于 corePoolSize，那么这些线程不会回收，除非将 allowCoreThreadTimeOut 设置为 true，**但一般不这么干**，因为频繁地创建销毁线程会极大地增加系统调用的开销。

- 2、workQueue: 如果线程数大于核心数 (corePoolSize) 且小于最大线程数 (maximumPoolSize), 则会将任务先丢到阻塞队列里, 然后线程自己去阻塞队列中拉取任务执行。
- 3、maximumPoolSize: 线程池中最大可创建的线程数, 如果提交任务时队列满了且线程数未到达这个设定值, 则会创建线程并执行此次提交的任务, 如果提交任务时队列满了但线程数已经到达了, 此时说明已经超出了线程池的负载能力, 就会执行拒绝策略, 这也好理解, 总不能让源源不断地任务进来把线程池给压垮了吧, 我们首先要保证线程池能正常工作。
- 4、RejectedExecutionHandler: 一共有以下四种拒绝策略
- AbortPolicy: 丢弃任务并抛出异常, 这也是默认策略;
 - CallerRunsPolicy: 用调用者所在的线程来执行任务, 所以开头的问题「线程把任务丢给线程池后肯定就马上返回了?」我们可以回答了, 如果用的是 CallerRunsPolicy 策略, 提交任务的线程 (比如主线程) 提交任务后并不能保证马上就返回, 当触发了这个 reject 策略不得不亲自来处理这个任务。
 - DiscardOldestPolicy: 丢弃阻塞队列中靠最前的任务, 并执行当前任务。
 - DiscardPolicy: 直接丢弃任务, 不抛出任何异常, 这种策略只适用于不重要的任务。
- 5、keepAliveTime: 线程存活时间, 如果在此时间内超出 corePoolSize 大小的线程处于 idle 状态, 这些线程会被回收
- 6、threadFactory: 可以用此参数设置线程池的命名, 指定 defaultUncaughtExceptionHandler (有啥用, 后文阐述), 甚至可以设定线程为守护线程。
- 现在问题来了, 该如何合理设置这些参数呢。

首先来看线程大小设置

<<Java 并发编程实战>>告诉我们应该分两种情况

1. 针对 CPU 密集型的任务, 在有 Ncpu 个处理器的系统上, 当线程池的大小为 Ncpu + 1 时, 通常能实现最优的利用率, +1 是因为当计算密集型线程偶尔由于缺页故障或其他原因而暂停工作时, 这个"额外"的线程也能确保 CPU 的时钟周期不会被浪费, 所谓 CPU 密集, 就是线程一直在忙碌, 这样将线程池的大小设置为 Ncpu + 1 避免了线程的上下文切换, 让线程时刻处于忙碌状态, 将 CPU 的利用率最大化。
2. 针对 IO 密集型的任务, 它也给出了如下计算公式

$$N_{cpu} = \text{number of CPUs}$$

$$U_{cpu} = \text{target CPU utilization}, 0 \leq U_{cpu} \leq 1$$

$$\frac{W}{C} = \text{ratio of wait time to compute time}$$

The optimal pool size for keeping the processors at the desired utilization is :

$$N_{threads} = N_{cpu} * U_{cpu} * (1 + \frac{W}{C})$$

这些公式看看就好, 实际的业务场景中基本用不上, 这些公式太过理论化了, 脱离业务场景, 仅可作个理论参考, 举个例子, 你说 CPU 密集型任务设置线程池大小为 N + 1 个, 但实际上在业务中往往不只设置一个线程池, 这种情况套用的公式就懵逼了



有点意思！说下去

再来看 workQueue 的大小设置

由上文可知，如果最大线程大于核心线程数，当且仅当核心线程满了且 workQueue 也满的情况下，才会新增新的线程，也就是说如果 workQueue 是无界队列，那么当线程数增加到 corePoolSize 后，永远不会再新增新的线程了，也就是说此时 maximumPoolSize 的设置就无效了，也无法触发 RejectedExecutionHandler 拒绝策略，任务只会源源不断地填充到 workQueue，直到 OOM。



震惊

所以 workQueue 应该为有界队列，至少保证在任务过载的情况下线程池还能正常工作，那么哪些是有界队列，哪些是无界队列呢。

有界队列我们常用的以下两个

- `LinkedBlockingQueue`: 链表构成的有界队列，按先进先出（FIFO）的顺序对元素进行排列，但注意在创建时需指定其大小，否则其大小默认为 `Integer.MAX_VALUE`，相当于无界队列了
- `ArrayBlockingQueue`: 数组实现的有界队列，按先进先出（FIFO）的顺序对元素进行排列。

无界队列我们常用 `PriorityBlockingQueue` 这个优先级队列，任务插入的时候可以指定其权重以让这些任务优先执行，但这个队列很少用，原因很简单，线程池里的任务执行顺序一般是平等的，如果真有必须某些类型的任务需要优先执行，大不了再开个线程池好了，将不同的任务类型用不同的线程池隔离开来，也是合理利用线程池的一种实践。

说到这我相信大家应该能回答开头的问题「阿里 Java 代码规范为什么不允许使用 Executors 快速创建线程池？」，最常见的是以下两种创建方式

```

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

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

```

image-20201109002227476

newCachedThreadPool 方法的最大线程数设置成了 Integer.MAX_VALUE，而 newSingleThreadExecutor 方法创建 workQueue 时 LinkedBlockingQueue 未声明大小，相当于创建了无界队列，一不小心就会导致 OOM。

threadFactory 如何设置

一般业务中会有多个线程池，如果某个线程池出现了问题，定位是哪一个线程出问题很重要，所以为每个线程池取一个名字就很有必要了，我司用的 dubbo 的 NamedThreadFactory 来生成 threadFactory，创建很简单

```
new NamedThreadFactory("demo-work")
```

它的实现还是很巧妙的，有兴趣地可以看看它的源码，每调用一次，底层有个计数器会加一，会依次命名为「demo-work-thread-1」，「demo-work-thread-2」，「demo-work-thread-3」这样递增的字符串。

在实际的业务场景中，一般很难确定 corePoolSize，workQueue，maximumPoolSize 的大小，如果出问题了，一般来说只能重新设置一下这些参数再发布，这样往往需要耗费一些时间，美团的[这篇文章](#)给出了让人眼前一亮的解决方案，当发现问题（线程池监控告警）时，动态调整这些参数，可以让这些参数实时生效，能在发现问题时及时解决，确实是个很好的思路。

线程池提交任务的两种方式

线程池创建好了，该怎么给它提交任务，有两种方式，调用 execute 和 submit 方法，来看下这两个方法的方法签名

```

// 方式一: execute 方法
public void execute(Runnable command) {
}

// 方式二: ExecutorService 中 submit 的三个方法
<T> Future<T> submit(Callable<T> task);
<T> Future<T> submit(Runnable task, T result);
Future<?> submit(Runnable task);

```

区别在于调用 execute 无返回值，而调用 submit 可以返回 Future，那么这个 Future 能到底能干啥呢，看它的接口

```

public interface Future<V> {

    /**

```

```

    * 取消正在执行的任务，如果任务已执行或已被取消，或者由于某些原因不能取消则返回 false
    * 如果任务未开始或者任务已开始但可以中断（mayInterruptIfRunning 为 true），则
    * 可以取消/中断此任务
    */
    boolean cancel(boolean mayInterruptIfRunning);

    /**
     * 任务在完成前是否已被取消
     */
    boolean isCancelled();

    /**
     * 正常的执行完流程流程，或抛出异常，或取消导致的任务完成都会返回 true
     */
    boolean isDone();

    /**
     * 阻塞等待任务的执行结果
     */
    V get() throws InterruptedException, ExecutionException;

    /**
     * 阻塞等待任务的执行结果，不过这里指定了时间，如果在 timeout 时间内任务还未执行完成，
     * 则抛出 TimeoutException 异常
     */
    V get(long timeout, TimeUnit unit)
        throws InterruptedException, ExecutionException, TimeoutException;
}

```

可以用 Future 取消任务，判断任务是否已取消/完成，甚至可以阻塞等待结果。

submit 为啥能提交任务（Runnable）的同时也能返回任务（Future）的执行结果呢

```

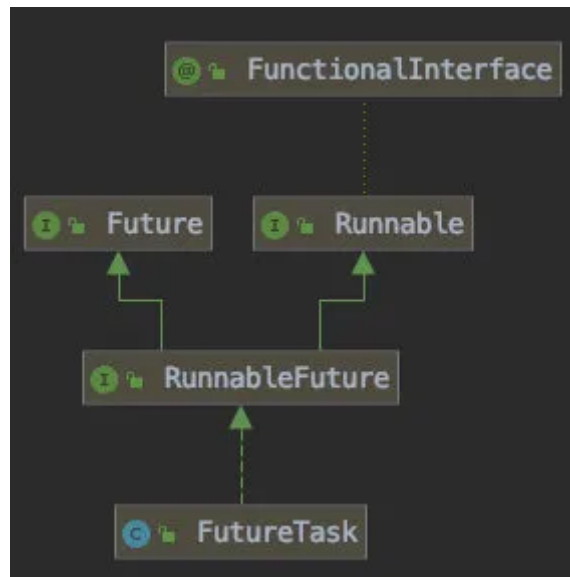
public Future<?> submit(Runnable task) {
    if (task == null) throw new NullPointerException();
    RunnableFuture<Void> ftask = newTaskFor(task, null);
    execute(ftask);
    return ftask;
}

public <T> Future<T> submit(Runnable task, T result) {
    if (task == null) throw new NullPointerException();
    RunnableFuture<T> ftask = newTaskFor(task, result);
    execute(ftask);
    return ftask;
}

public <T> Future<T> submit(Callable<T> task) {
    if (task == null) throw new NullPointerException();
    RunnableFuture<T> ftask = newTaskFor(task);
    execute(ftask);
    return ftask;
}

```


原来在最后执行 `execute` 前用 `newTaskFor` 将 `task` 封装成了 `RunnableFuture`, `newTaskFor` 返回了 `FutureTask` 这个类, 结构图如下



可以看到 `FutureTask` 这个接口既实现了 `Runnable` 接口, 也实现 `Future` 接口, 所以在提交任务的同时也能利用 `Future` 接口来执行任务的取消, 获取任务的状态, 等待执行结果这些操作。

`execute` 与 `submit` 除了是否能返回执行结果这一区别外, 还有一个重要区别, 那就是使用 `execute` 执行如果发生了异常, 是捕获不到的, 默认会执行 `ThreadGroup` 的 `uncaughtException` 方法 (下图数字 2 对应的逻辑)

```
public void uncaughtException(Thread t, Throwable e) {
    if (parent != null) {
        parent.uncaughtException(t, e);
    } else {
        Thread.UncaughtExceptionHandler ueh =
            Thread.getDefaultUncaughtExceptionHandler();
        if (ueh != null) {
            ueh.uncaughtException(t, e); 1
        } else if (!(e instanceof ThreadDeath)) {
            System.err.print("Exception in thread \""
                + t.getName() + "\" "); 2
            e.printStackTrace(System.err);
        }
    }
}
```

所以如果你想监控执行 `execute` 方法时发生的异常, 需要通过 `threadFactory` 来指定一个 `UncaughtExceptionHandler`, 这样就会执行上图中的 1, 进而执行 `UncaughtExceptionHandler` 中的逻辑,如下所示:

```
//1.实现一个自己的线程池工厂
ThreadFactory factory = (Runnable r) -> {
    //创建一个线程
    Thread t = new Thread(r);
    //给创建的线程设置UncaughtExceptionHandler对象 里面实现异常的默认逻辑
    t.setUncaughtExceptionHandler((Thread thread1, Throwable e) -> {
        // 在此设置统计监控逻辑
        System.out.println("线程工厂设置的exceptionHandler" + e.getMessage());
    });
};
```

```

        return t;
    };

    // 2.创建一个自己定义的线程池，使用自己定义的线程工厂
    ExecutorService service = new ThreadPoolExecutor(1, 1, 0,
        TimeUnit.MILLISECONDS, new LinkedBlockingQueue(10), factory);

    //3.提交任务
    service.execute(()->{
        int i=1/0;
    });

```

执行以上逻辑最终会输出「线程工厂设置的exceptionHandler/ by zero」,通过这样的方式就能通过设置的 defaultUncaughtExceptionHandler 来执行我们的监控逻辑了。

如果用 submit，如何捕获异常呢，当我们调用 future.get 就可以捕获

```

Callable testCallable = xxx;
Future future = executor.submit(myCallable);
try {
    future.get(3);
} catch (InterruptedException e) {
    e.printStackTrace();
} catch (ExecutionException e) {
    e.printStackTrace();
}

```

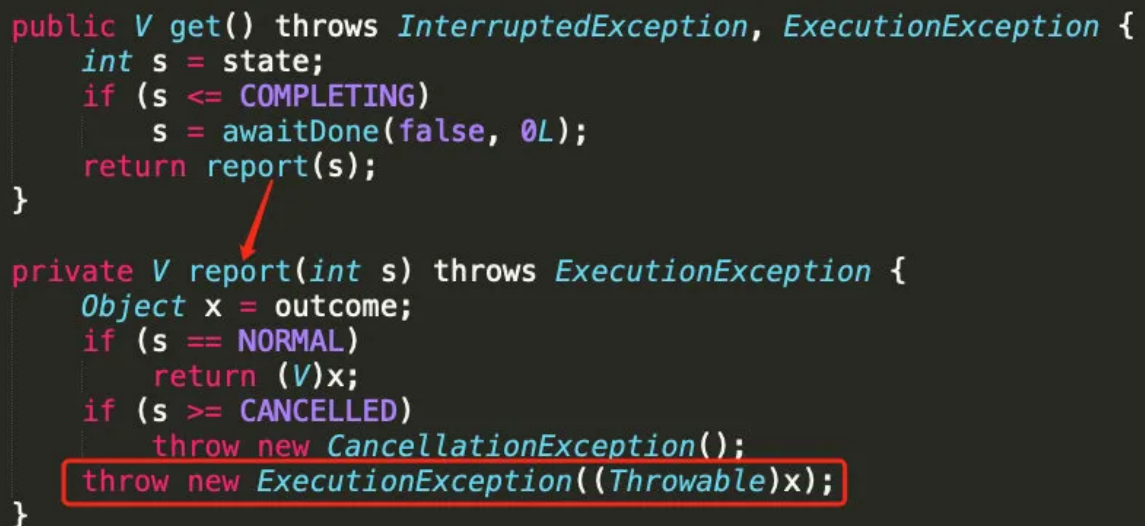
那么 future 为啥在 get 的时候才捕获异步呢，因为在执行 submit 时抛出异常后此异常被保存了起来，而在 get 的时候才被抛出

```

public V get() throws InterruptedException, ExecutionException {
    int s = state;
    if (s <= COMPLETING)
        s = awaitDone(false, 0L);
    return report(s);
}

private V report(int s) throws ExecutionException {
    Object x = outcome;
    if (s == NORMAL)
        return (V)x;
    if (s >= CANCELLED)
        throw new CancellationException();
    throw new ExecutionException((Throwable)x);
}

```



关于 execute 和 submit 的执行流程 why 神的[这篇文章](#)写得非常透彻，我就不拾人牙慧了，建议大家好好品品，收获会很大！

ThreadPoolExecutor 源码剖析

前面铺垫了这么多，终于到了最核心的源码剖析环节了。

对于线程池来说，我们最关心的是它的「状态」和「可运行的线程数量」，一般来说我们可以选择用两个变量来记录，不过 Doug Lea 只用了一个变量 (ctl) 就达成目的了，我们知道变量越多，代码的可维护性就越差，也更容易出 bug, 所以只用一个变量就达成了两个变量的效果，这让代码的可维护性大大提高，那么他是怎么设计的呢

```
// ThreadPoolExecutor.java
public class ThreadPoolExecutor extends AbstractExecutorService {
    private final AtomicInteger ctl = new AtomicInteger(ctlOf(RUNNING, 0));
    private static final int COUNT_BITS = Integer.SIZE - 3;
    private static final int CAPACITY = (1 << COUNT_BITS) - 1;

    // 结果: 111 00000000000000000000000000000000
    private static final int RUNNING = -1 << COUNT_BITS;
    // 结果: 000 00000000000000000000000000000000
    private static final int SHUTDOWN = 0 << COUNT_BITS;
    // 结果: 001 00000000000000000000000000000000
    private static final int STOP = 1 << COUNT_BITS;
    // 结果: 010 00000000000000000000000000000000
    private static final int TIDYING = 2 << COUNT_BITS;
    // 结果: 011 00000000000000000000000000000000
    private static final int TERMINATED = 3 << COUNT_BITS;

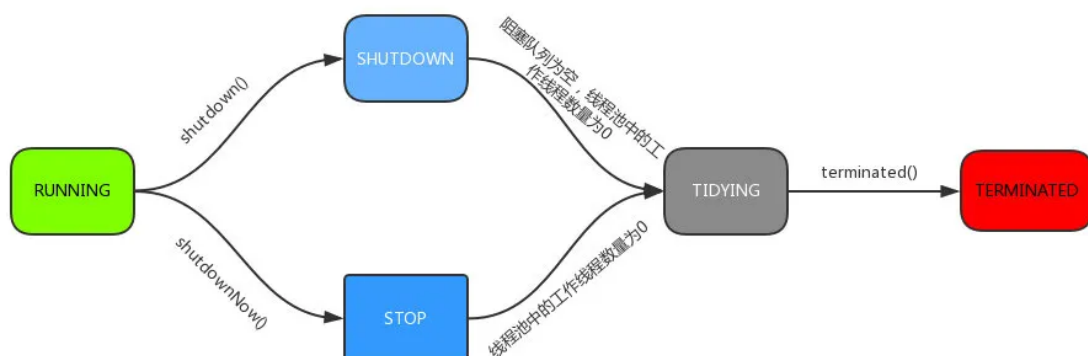
    // 获取线程池的状态
    private static int runStateOf(int c) { return c & ~CAPACITY; }
    // 获取线程数量
    private static int workerCountOf(int c) { return c & CAPACITY; }
}

```

可以看到，ctl 是一个原子类的 Integer 变量，有 32 位，低 29 位表示线程数量，29 位最大可以表示 $(2^{29}-1)$ (大概 5 亿多)，足够记录线程大小了，如果未来还是不够，可以把 ctl 声明为 AtomicLong，高 3 位用来表示线程池的状态，3 位可以表示 8 个线程池的状态，由于线程池总共只有五个状态，所以 3 位也是足够了，线程池的五个状态如下

- RUNNING: 接收新的任务，并能继续处理 workQueue 中的任务
- SHUTDOWN: 不再接收新的任务，不过能继续处理 workQueue 中的任务
- STOP: 不再接收新的任务，也不再处理 workQueue 中的任务，并且会中断正在处理任务的线程
- TIDYING: 所有的任务都完结了，并且线程数量 (workCount) 为 0 时即为此状态，进入此状态后会调用 terminated() 这个钩子方法进入 TERMINATED 状态
- TERMINATED: 调用 terminated() 方法后即为此状态

线程池的状态流转及触发条件如下



有了这些基础，我们来分析下 execute 的源码

```
public void execute(Runnable command) {
    if (command == null)
        throw new NullPointerException();
    int c = ctl.get();

```

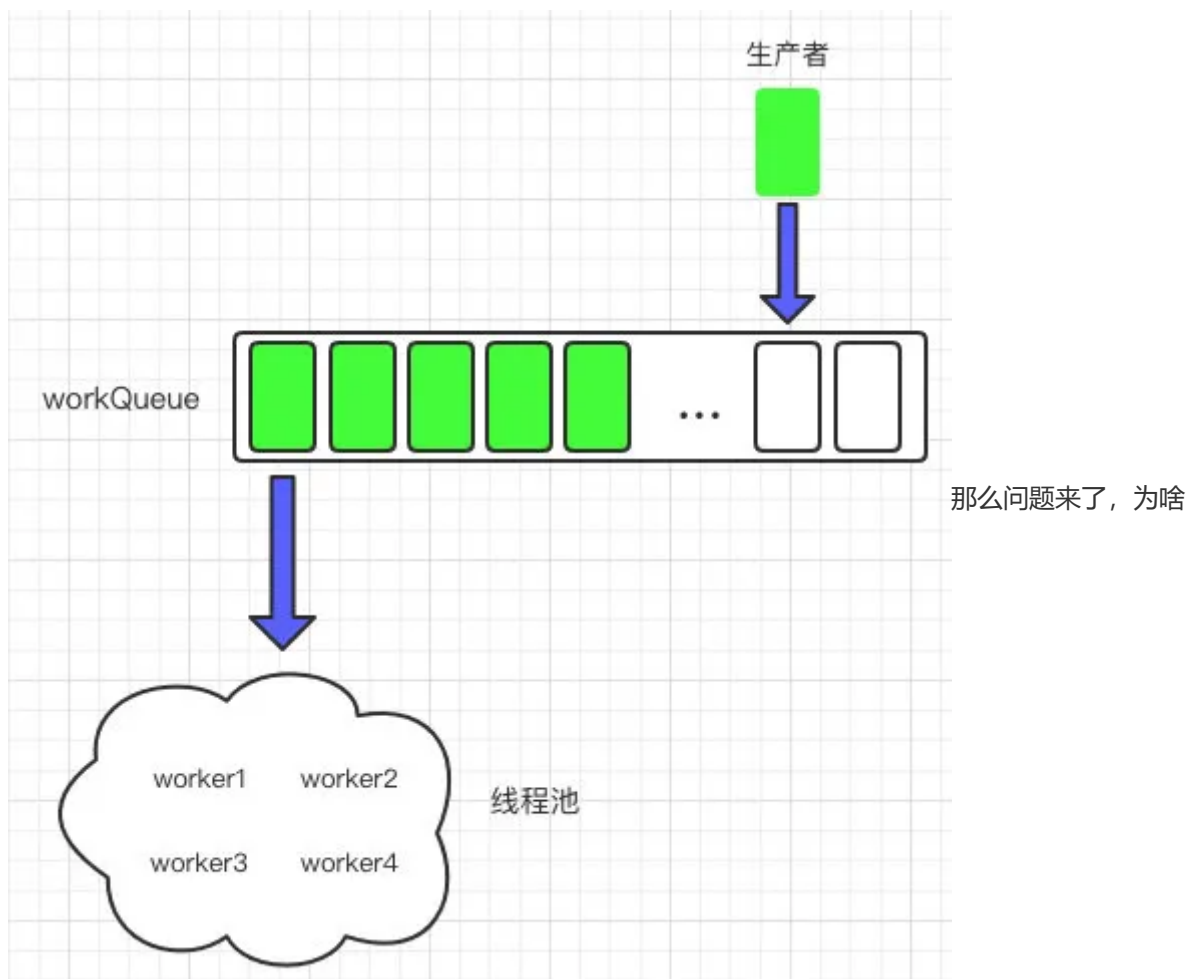
```

        // 如果当前线程数少于核心线程数（corePoolSize），无论核心线程是否忙碌，都创建线程，直到达到 corePoolSize 为止
        if (workerCountOf(c) < corePoolSize) {
            // 创建线程并将此任务交给 worker 处理（此时此任务即 worker 中的 firstTask）
            if (addWorker(command, true))
                return;
            c = ctl.get();
        }

        // 如果线程池处于 RUNNING 状态，并且线程数大于 corePoolSize 或者
        // 线程数少于 corePoolSize 但创建线程失败了，则将任务丢进 workQueue 中
        if (isRunning(c) && workQueue.offer(command)) {
            int recheck = ctl.get();
            // 这里需要再次检查线程池是否处于 RUNNING 状态，因为在任务入队后可能线程池状态会发生变化，（比如调用了 shutdown 方法等），如果线程状态发生了变化了，则移除此任务，执行拒绝策略
            if (!isRunning(recheck) && remove(command))
                reject(command);
            // 如果线程池在 RUNNING 状态下，线程数为 0，则新建线程加速处理 workQueue 中的任务
            else if (workerCountOf(recheck) == 0)
                addWorker(null, false);
        }
        // 这段逻辑说明线程数大于 corePoolSize 且任务入队失败了，此时会以最大线程数（maximumPoolSize）为界来创建线程，如果失败，说明线程数超过了 maximumPoolSize，则执行拒绝策略
        else if (!addWorker(command, false))
            reject(command);
    }
}

```

从这段代码中可以看到，创建线程是调用 addWorker 实现的，在分析 addWorker 之前，有必要简单提一下 Worker，线程池把每一个执行任务的线程都封装为 Worker 的形式，取名为 Worker 很形象，线程池的本质是生产者-消费者模型，生产者不断地往 workQueue 中丢 task, workQueue 就像流水线一样不断地输送着任务，而 worker（工人）不断地取任务来执行



要把线程封装到 worker 中呢，线程池拿到 task 后直接丢给线程处理或者让线程自己去 workQueue 中处理不就完了？

将线程封装为 worker 主要是为了更好地管理线程的中断

来看下 Worker 的定义

```
// 此处可以看出 worker 既是一个 Runnable 任务，也实现了 AQS（实际上是用 AQS 实现了一个独占锁，这样由于 worker 运行时会上锁，执行 shutdown, setCorePoolSize, setMaximumPoolSize 等方法时会试着中断线程（interruptIdleWorkers），在这个方法中断方法中会先尝试获取 worker 的锁，如果不成功，说明 worker 在运行中，此时会先让 worker 执行完任务再关闭 worker 的线程，实现优雅关闭线程的目的）
```

```
private final class worker
    extends AbstractQueuedSynchronizer
    implements Runnable
{
    private static final long serialVersionUID = 6138294804551838833L;

    // 实际执行任务的线程
    final Thread thread;
    // 上文提到，如果当前线程数少于核心线程数，创建线程并将提交的任务交给 worker 处理，此时 firstTask 即为此提交的任务，如果 worker 从 workQueue 中获取任务，则 firstTask 为空

    Runnable firstTask;
    // 统计完成的任务数
    volatile long completedTasks;

    worker(Runnable firstTask) {
        // 初始化为 -1，这样在线程运行前（调用 runworker）禁止中断，在 interruptIfStarted() 方法中会判断 getState() >= 0
    }
}
```



```

        setState(-1);
        this.firstTask = firstTask;

        // 根据线程池的 threadFactory 创建一个线程，将 worker 本身传给线程（因为
worker 实现了 Runnable 接口）
        this.thread = getThreadFactory().newThread(this);
    }

    public void run() {
        // thread 启动后会调用此方法
        runWorker(this);
    }

    // 1 代表被锁住了，0 代表未锁
    protected boolean isHeldExclusively() {
        return getState() != 0;
    }

    // 尝试获取锁
    protected boolean tryAcquire(int unused) {
        // 从这里可以看出它是一个独占锁，因为当获取锁后，cas 设置 state 不可能成功，这
里我们也能明白上文中将 state 设置为 -1 的作用，这种情况下永远不可能获取得锁，而 worker 要被中
断首先必须获取锁
        if (compareAndSetState(0, 1)) {
            setExclusiveOwnerThread(Thread.currentThread());
            return true;
        }
        return false;
    }

    // 尝试释放锁
    protected boolean tryRelease(int unused) {
        setExclusiveOwnerThread(null);
        setState(0);
        return true;
    }

    public void lock()          { acquire(1); }
    public boolean tryLock()    { return tryAcquire(1); }
    public void unlock()        { release(1); }
    public boolean isLocked()   { return isHeldExclusively(); }

    // 中断线程，这个方法会被 shutdownNow 调用，从中可以看出 shutdownNow 要中断线程不
需要获取锁，也就是说如果线程正在运行，照样会给你中断掉，所以一般来说我们不用 shutdownNow 来中断
线程，太粗暴了，中断时线程很可能在执行任务，影响任务执行
    void interruptIfStarted() {
        Thread t;
        // 中断也是有条件的，必须是 state >= 0 且 t != null 且线程未被中断
        // 如果 state == -1，不执行中断，再次明白了为啥上文中 setState(-1) 的意义
        if (getState() >= 0 && (t = thread) != null && !t.isInterrupted()) {
            try {
                t.interrupt();
            } catch (SecurityException ignore) {
            }
        }
    }
}

```

通过上文对 Worker 类的分析，相信大家不难理解 **将线程封装为 worker 主要是为了更好地管理线程的中断** 这句话。

理解了 Worker 的意义，我们再来看 addWorker 的方法

```
private boolean addWorker(Runnable firstTask, boolean core) {
    retry:
    for (;;) {
        int c = ctl.get();

        // 获取线程池的状态
        int rs = runStateOf(c);

        // 如果线程池的状态 >= SHUTDOWN，即为 SHUTDOWN, STOP, TIDYING, TERMINATED 这四个状态，只有一种情况有可能创建线程，即线程状态为 SHUTDOWN，且队列非空时，firstTask == null 代表创建一个不接收新任务的线程（此线程会从 workQueue 中获取任务再执行），这种情况下创建线程是为了加速处理完 workQueue 中的任务
        if (rs >= SHUTDOWN &&
            ! (rs == SHUTDOWN &&
                firstTask == null &&
                ! workQueue.isEmpty()))
            return false;

        for (;;) {
            // 获取线程数
            int wc = workerCountOf(c);
            // 如果超过了线程池的最大 CAPACITY（5 亿多，基本不可能）
            // 或者 超过了 corePoolSize（core 为 true）或者 maximumPoolSize（core 为 false）时
            // 则返回 false
            if (wc >= CAPACITY ||
                wc >= (core ? corePoolSize : maximumPoolSize))
                return false;
            // 否则 CAS 增加线程的数量，如果成功跳出双重循环
            if (compareAndIncrementWorkerCount(c))
                break retry;
            c = ctl.get(); // Re-read ctl

            // 如果线程运行状态发生变化，跳到外层循环继续执行
            if (runStateOf(c) != rs)
                continue retry;
            // 说明是因为 CAS 增加线程数量失败所致，继续执行 retry 的内层循环
        }
    }

    boolean workerStarted = false;
    boolean workerAdded = false;
    worker w = null;
    try {
        // 能执行到这里，说明满足增加 worker 的条件了，所以创建 worker，准备添加进线程池中执行任务
        w = new Worker(firstTask);
        final Thread t = w.thread;
        if (t != null) {
            // 加锁，是因为下文要把 w 添加进 workers 中，workers 是 HashSet，不是线程安全的，所以需要加锁予以保证

```

```

        final ReentrantLock mainLock = this.mainLock;
        mainLock.lock();
        try {
            // 再次 check 线程池的状态以防执行到此步时发生中断等
            int rs = runStateOf(ctl.get());
            // 如果线程池状态小于 SHUTDOWN（即为 RUNNING），
            // 或者状态为 SHUTDOWN 但 firstTask == null（代表不接收任务，只是创建线
            程处理 workQueue 中的任务），则满足添加 worker 的条件
            if (rs < SHUTDOWN ||
                (rs == SHUTDOWN && firstTask == null)) {
                // 如果线程已启动，显然有问题（因为创建 worker
                后，还没启动线程呢），抛出异常
                if (t.isAlive())
                    throw new IllegalStateException();
                workers.add(w);
                int s = workers.size();

                // 记录最大的线程池大小以作监控之用
                if (s > largestPoolSize)
                    largestPoolSize = s;
                workerAdded = true;
            }
        } finally {
            mainLock.unlock();
        }

        // 说明往 workers 中添加 worker 成功，此时启动线程
        if (workerAdded) {
            t.start();
            workerStarted = true;
        }
    }
} finally {
    // 添加线程失败，执行 addWorkerFailed 方法，主要做了将 worker 从 workers 中移
    除，减少线程数，并尝试着关闭线程池这样的操作
    if (! workerStarted)
        addWorkerFailed(w);
}
return workerStarted;
}

```

从这段代码我们可以看到多线程下情况的不可预料性，我们发现在满足条件情况下，又对线程状态重新进行了 check,以防期间出现中断等线程池状态发生变更的操作，这也给我们以启发：多线程环境下的各种临界条件一定要考虑到位。

执行 addWorker 创建 worker 成功后，线程开始执行了 (t.start())，由于在创建 Worker 时，将 Worker 自己传给了此线程，所以启动线程后，会调用 Worker 的 run 方法

```

public void run() {
    runWorker(this);
}

```

可以看到最终会调用 runWorker 方法，接下来我们分析下 runWorker 方法

```

final void runWorker(Worker w) {
    Thread wt = Thread.currentThread();

```

```

Runnable task = w.firstTask;
w.firstTask = null;
// unlock 会调用 tryRelease 方法将 state 设置成 0, 代表允许中断, 允许中断的条件上文我
们在 interruptIfStarted() 中有提过, 即 state >= 0
w.unlock();
boolean completedAbruptly = true;
try {
    // 如果在提交任务时创建了线程, 并把任务丢给此线程, 则会先执行此 task
    // 否则从任务队列中获取 task 来执行 (即 getTask() 方法)
    while (task != null || (task = getTask()) != null) {
        w.lock();

        // 如果线程池状态为 >= STOP (即 STOP, TIDYING, TERMINATED ) 时, 则线程应该
中断
        // 如果线程池状态 < STOP, 线程不应该中断, 如果中断了 (Thread.interrupted()
返回 true, 并清除标志位), 再次判断线程池状态 (防止在清除标志位时执行了 shutdownNow() 这样的
方法), 如果此时线程池为 STOP, 执行线程中断
        if ((runStateAtLeast(ctl.get(), STOP) ||
            (Thread.interrupted() &&
             runStateAtLeast(ctl.get(), STOP))) &&
            !wt.isInterrupted())
            wt.interrupt();
        try {
            // 执行任务前, 子类可实现此钩子方法作为统计之用
            beforeExecute(wt, task);
            Throwable thrown = null;
            try {
                task.run();
            } catch (RuntimeException x) {
                thrown = x; throw x;
            } catch (Error x) {
                thrown = x; throw x;
            } catch (Throwable x) {
                thrown = x; throw new Error(x);
            } finally {
                // 执行任务后, 子类可实现此钩子方法作为统计之用
                afterExecute(task, thrown);
            }
        } finally {
            task = null;
            w.completedTasks++;
            w.unlock();
        }
    }
    completedAbruptly = false;
} finally {
    // 如果执行到这只有两种可能, 一种是执行过程中异常中断了, 一种是队列里没有任务了, 从这
里可以看出线程没有核心线程与非核心线程之分, 哪个任务异常了或者正常退出了都会执行此方法, 此方法会
根据情况将线程数-1
    processWorkerExit(w, completedAbruptly);
}
}

```

来看看 processWorkerExit 方法是咋样的

```

private void processWorkerExit(Worker w, boolean completedAbruptly) {
    // 如果异常退出, cas 执行线程池减 1 操作

```

```

        if (completedAbruptly)
            decrementWorkerCount();

        final ReentrantLock mainLock = this.mainLock;
        mainLock.lock();
        try {
            completedTaskCount += w.completedTasks;
            // 加锁确保线程安全地移除 worker
            workers.remove(w);
        } finally {
            mainLock.unlock();
        }

        // woker 既然异常退出，可能线程池状态变了（如执行 shutdown 等），尝试着关闭线程池
        tryTerminate();

        int c = ctl.get();

        // 如果线程池处于 STOP 状态，则如果 woker 是异常退出的，重新新增一个 woker，如果是正常
        // 退出的，在 wokerQueue 为非空的条件下，确保至少有一个线程在运行以执行 wokerQueue 中的任务
        if (runStateLessThan(c, STOP)) {
            if (!completedAbruptly) {
                int min = allowCoreThreadTimeOut ? 0 : corePoolSize;
                if (min == 0 && !workQueue.isEmpty())
                    min = 1;
                if (workerCountOf(c) >= min)
                    return; // replacement not needed
            }
            addWorker(null, false);
        }
    }
}

```

接下来我们分析 woker 从 workQueue 中取任务的方法 getTask

```

private Runnable getTask() {
    boolean timedOut = false; // Did the last poll() time out?

    for (;;) {
        int c = ctl.get();
        int rs = runStateOf(c);

        // 如果线程池状态至少为 STOP 或者
        // 线程池状态 == SHUTDOWN 并且任务队列是空的
        // 则减少线程数量，返回 null，这种情况下上文分析的 runWorker 会执行
        // processWorkerExit 从而让获取此 Task 的 woker 退出
        if (rs >= SHUTDOWN && (rs >= STOP || workQueue.isEmpty())) {
            decrementWorkerCount();
            return null;
        }

        int wc = workerCountOf(c);

        // 如果 allowCoreThreadTimeOut 为 true，代表任何线程在 keepAliveTime 时间内处
        // 于 idle 状态都会被回收，如果线程数大于 corePoolSize，本身在 keepAliveTime 时间内处于
        // idle 状态就会被回收
        boolean timed = allowCoreThreadTimeOut || wc > corePoolSize;
    }
}

```



```

// worker 应该被回收的几个条件，这个比较简单，就此略过
if ((wc > maximumPoolSize || (timed && timedOut))
    && (wc > 1 || workQueue.isEmpty())) {
    if (compareAndDecrementWorkerCount(c))
        return null;
    continue;
}

try {
    // 阻塞获取 task，如果在 keepAliveTime 时间内未获取任务，说明超时了，此时
    timedOut 为 true
    Runnable r = timed ?
        workQueue.poll(keepAliveTime, TimeUnit.NANOSECONDS) :
        workQueue.take();
    if (r != null)
        return r;
    timedOut = true;
} catch (InterruptedException retry) {
    timedOut = false;
}
}
}

```

经过以上源码剖析，相信我们对线程池的工作原理理解得八九不离十了，再来简单过一下其他一些比较有用的方法，开头我们提到线程池的监控问题，我们看一下可以监控哪些指标

- `int getCorePoolSize()`: 获取核心线程数。
- `int getLargestPoolSize()`: 历史峰值线程数。
- `int getMaximumPoolSize()`: 最大线程数(线程池线程容量)。
- `int getActiveCount()`: 当前活跃线程数
- `int getPoolSize()`: 当前线程池中的线程总数
- `BlockingQueue getQueue()` 当前线程池的任务队列，据此可以获取积压任务的总数，
`getQueue.size()`

监控思路也很简单，开启一个定时线程 `ScheduledThreadPoolExecutor`，定期对这些线程池指标进行采集，一般会采用一些开源工具如 Grafana + Prometheus + MicroMeter 来实现。

如何实现核心线程池的预热

使用 `prestartAllCoreThreads()` 方法，这个方法会一次性创建 `corePoolSize` 个线程，无需等到提交任务时才创建，提交创建好线程的话，一有任务提交过来，这些线程就可以立即处理。

如何实现动态调整线程池参数

- `setCorePoolSize(int corePoolSize)` 调整核心线程池大小
- `setMaximumPoolSize(int maximumPoolSize)`
- `setKeepAliveTime()` 设置线程的存活时间

解答开篇的问题

其它问题基本都在源码剖析环节回答了，这里简单说下其他问题

1、Tomcat 的线程池和 JDK 的线程池实现有啥区别, Dubbo 中有类似 Tomcat 的线程池实现吗? Dubbo 中一个叫 `EagerThreadPool` 的东西, 可以看看它的使用说明

```
17 package org.apache.dubbo.common.threadpool.support.eager;
18
19
20 import ...
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39 * EagerThreadPool
40 * When the core threads are all in busy,
41 * create new thread instead of putting task into blocking queue.
42
43 public class EagerThreadPool implements ThreadPool {
44
45     @Override
46     public Executor getExecutor(URL url) {
47         String name = url.getParameter(THREAD_NAME_KEY, DEFAULT_THREAD_NAME);
48         int cores = url.getParameter(CORE_THREADS_KEY, DEFAULT_CORE_THREADS);
49         int threads = url.getParameter(THREADS_KEY, Integer.MAX_VALUE);
50         int queues = url.getParameter(QUEUES_KEY, DEFAULT_QUEUES);
51         int alive = url.getParameter(ALIVE_KEY, DEFAULT_ALIVE);
```

从注释里可以看出, 如果核心线程都处于 busy 状态, 如果有新的请求进来, `EagerThreadPool` 会选择先创建线程, 而不是将其放入任务队列中, 这样可以更快地响应这些请求。

Tomcat 实现也是与此类似的, 只不过稍微有所不同, 当 Tomcat 启动时, 会先创建 `minSpareThreads` 个线程, 如果经过一段时间收到请求时这些线程都处于忙碌状态, 每次都会以 `minSpareThreads` 的步长创建线程, 本质上也是为了更快地响应处理请求。具体的源码可以看它的 `ThreadPool` 实现, 这里就不展开了。

2、我司网关 dubbo 调用线程池曾经出现过这样的问题: 压测时接口可以正常返回, 但接口 RT 很高, 假设设置的核心线程大小为 500, 最大线程为 800, 缓冲队列为 5000, 你能从这个设置中发现出一些问题并对这些参数进行调优吗? 这个参数明显能看出问题来, 首先任务队列设置过大, 任务达到核心线程后, 如果再有请求进来会先进入任务队列, 队列满了之后才创建线程, 创建线程也是需要不少开销的, 所以我们后来把核心线程设置成了与最大线程一样, 并且调用 `prestartAllCoreThreads()` 来预热核心线程, 就不用等请求来时再创建线程了。

线程池的几个最佳实践

1、线程池执行的任务应该是互相独立的, 如果互相依赖的话, 可能导致死锁, 比如下面这样的代码

```
ExecutorService pool = Executors
    .newSingleThreadExecutor();
pool.submit(() -> {
    try {
        String qq=pool.submit(()->"QQ").get();
        System.out.println(qq);
    } catch (Exception e) {
    }
});
```

2、核心任务与非核心任务最好能用多个线程池隔离开来

曾经我们业务上就出现这样的故障: 突然很多用户反馈短信收不到了, 排查才发现发短信是在一个线程池里, 而另外的定时脚本也是用的这个线程池来执行任务, 这个脚本一分钟可能产生几百上千条任务, 导致发短信的方法在线程池里基本没机会执行, 后来我们用了两个线程池把发短信和执行脚本隔离开来解决了问题。

3、添加线程池监控，动态设置线程池

如前文所述，线程池的各个参数很难一次性确定，既然难以确定，又要保证发现问题后及时解决，我们就需要为线程池增加监控，监控队列大小，线程数量等，我们可以设置 3 分钟内比如队列任务一直都是满了的话，就触发告警，这样可以提前预警，如果线上因为线程池参数设置不合理而触发了降级等操作，可以通过动态设置线程池的方式来实时修改核心线程数，最大线程数等，将问题及时修复。

总结

本文详细剖析了线程池的工作原理，相信大家对其工作机制应该有了较深入的了解，也对开头的几个问题有了较清楚的认识，本质上设置线程池的目的是为了利用有效的资源最大化性能，最小化风险，同时线程池的使用本质上是为了更好地为用户服务，据此也不难明白 Tomcat, Dubbo 要另起炉灶来设置自己的线程池了。

巨人的肩膀

- <https://dzone.com/articles/how-much-memory-does-a-java-thread-take>
 - <https://segmentfault.com/a/1190000021047279>
 - <https://www.cnblogs.com/trust-freedom/p/6681948.html>
 - 深入理解线程池 <https://tinyurl.com/y675j928>
 - 有的线程它死了，于是它变成一道面试题 <https://mp.weixin.qq.com/s/wrTVGLDvhE-eb5lhygWEqQ>
 - Java 并发编程实战
 - Java线程池实现原理及其在美团业务中的实践: <https://mp.weixin.qq.com/s/baYuX8aCwQ9PP6k7TDI2Ww>
 - 线程池异常处理详解，一文搞懂！ <https://www.cnblogs.com/ncy1/articles/11629933.html>
-

推荐阅读

[点个外卖，我把「软中断」搞懂了](#)

[读者问：小林怎么学操作系统和计算机网络呀？](#)

[读者问：小林你的 500 张图是怎么画的？](#)

[读者问：小林你能分享做公众号的经验吗？](#)