有道云链接：https://note.youdao.com/s/J6OZk2d6
课程代码：

线程池执行任务的具体流程是怎样的?

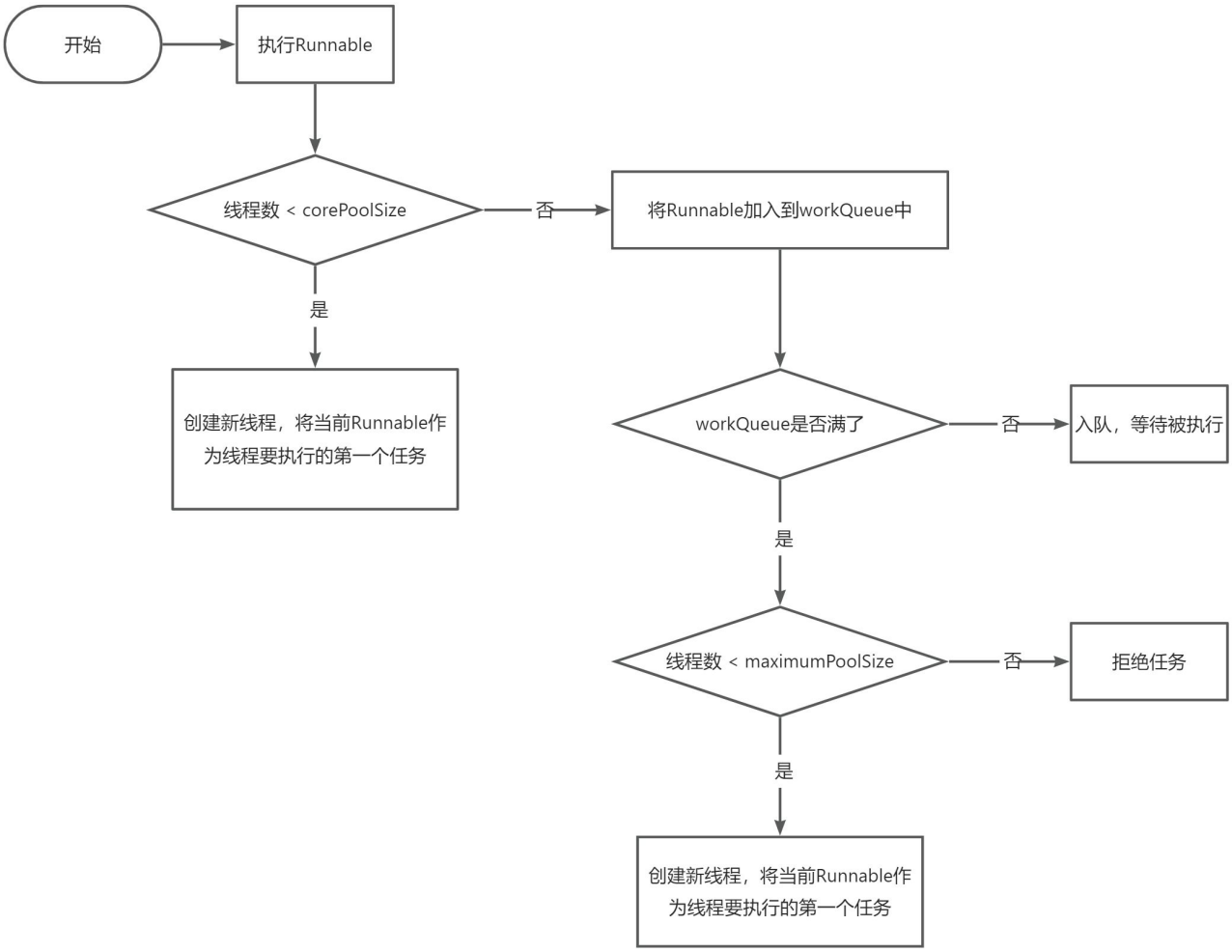
ThreadPoolExecutor中提供了两种执行任务的方法：

1. void execute(Runnable command)
2. Future<?> submit(Runnable task)

实际上submit中最终还是调用的execute()方法，只不过会返回一个Future对象，用来获取任务执行结果：

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

execute(Runnable command)方法执行时会分为三步：



注意：提交一个Runnable时，不管当前线程池中的线程是否空闲，只要数量小于核心线程数就会创建新线程。

注意：ThreadPoolExecutor相当于是非公平的，比如队列满了之后提交的Runnable可能会比正在排队的Runnable先执行。

线程池的五种状态是如何流转的?

线程池有五种状态：

- RUNNING：**会**接收新任务并且**会**处理队列中的任务
- SHUTDOWN：**不会**接收新任务并且**会**处理队列中的任务
- STOP：**不会**接收新任务并且**不会**处理队列中的任务，并且会中断在处理的任务（注意：一个任务能不能被中断得看任务本身）
- TIDYING：所有任务都终止了，线程池中也没有线程了，这样线程池的状态就会转为TIDYING，一旦达到此状态，就会调用线程池的terminated()
- TERMINATED：terminated()执行完之后就会转变为TERMINATED

这五种状态并不能任意转换，只会有以下几种转换情况：

1. RUNNING -> SHUTDOWN：手动调用shutdown()触发，或者线程池对象GC时会调用finalize()从而调用

shutdown()

2. (RUNNING or SHUTDOWN) -> STOP：调用shutdownNow()触发，如果先调shutdown()紧着调shutdownNow()，就会发生SHUTDOWN -> STOP
3. SHUTDOWN -> TIDYING：队列为空并且线程池中没有线程时自动转换
4. STOP -> TIDYING：线程池中没有线程时自动转换（队列中可能还有任务）
5. TIDYING -> TERMINATED：terminated()执行完后就会自动转换

线程池中的线程是如何关闭的?

我们一般会使用thread.start()方法来开启一个线程，那如何停掉一个线程呢?

Thread类提供了一个stop()，但是标记了@Deprecated，为什么不推荐用stop()方法来停掉线程呢?

因为stop()方法太粗暴了，一旦调用了stop()，就会直接停掉线程，但是调用的时候根本不知道线程刚刚在做什么，任务做到哪一步了，这是很危险的。

这里强调一点，stop()会释放线程占用的synchronized锁（不会自动释放ReentrantLock锁，这也是不建议用stop()的一个因素）。

```
package com.zhouyu;

import java.util.concurrent.locks.ReentrantLock;

/**
 * 作者：周瑜大都督
 */
public class ThreadTest {

    static int count = 0;
    static final Object lock = new Object();
    static final ReentrantLock reentrantLock = new ReentrantLock();

    public static void main(String[] args) throws InterruptedException {

        Thread thread = new Thread(new Runnable() {
            public void run() {
```

```
//                synchronized (lock) {
                reentrantLock.lock();
                    for (int i = 0; i < 100; i++) {
                        count++;
                        try {
                            Thread.sleep(1000);
                        } catch (InterruptedException e) {
                            throw new RuntimeException(e);
                        }
                    }
//                }
                reentrantLock.unlock();
            }
        });

        thread.start();

        Thread.sleep(5*1000);

        thread.stop();
//
//        Thread.sleep(5*1000);

        reentrantLock.lock();
        System.out.println(count);
        reentrantLock.unlock();

//        synchronized (lock) {
//            System.out.println(count);
//        }


    }
}

```

所以，我们建议通过自定义一个变量，或者通过中断来停掉一个线程，比如：

```java
public class ThreadTest {

    static int count = 0;
    static boolean stop = false;

    public static void main(String[] args) throws InterruptedException {

        Thread thread = new Thread(new Runnable() {
            public void run() {

                for (int i = 0; i < 100; i++) {
                    if (stop) {
                        break;
                    }

                    count++;
                    try {
                        Thread.sleep(1000);
                    } catch (InterruptedException e) {
                        throw new RuntimeException(e);
                    }
                }
            }
        });

        thread.start();

        Thread.sleep(5 * 1000);

        stop = true;

        Thread.sleep(5 * 1000);


        System.out.println(count);


    }
}
```

不同点在于，当我们把stop设置为true时，线程自身可以控制到底要不要停止，何时停止，同样，我们可以调用thread的interrupt()来中断线程：

```
public class ThreadTest {

    static int count = 0;
    static boolean stop = false;

    public static void main(String[] args) throws InterruptedException {

        Thread thread = new Thread(new Runnable() {
            public void run() {

                for (int i = 0; i < 100; i++) {
                    if (Thread.currentThread().isInterrupted()) {
                        break;
                    }

                    count++;
                    try {
                        Thread.sleep(1000);
                    } catch (InterruptedException e) {
                        break;
                    }
                }
            }
        });

        thread.start();

        Thread.sleep(5 * 1000);

        thread.interrupt();

        Thread.sleep(5 * 1000);


```

```
35          System.out.println(count);
36
37
38      }
39  }
```

不同的地方在于，线程sleep过程中如果被中断了会接收到异常。

讲了这么多，其实线程池中就是通过interrupt()来停止线程的，比如shutdownNow()方法中会调用：

```
1  void interruptIfStarted() {
2      Thread t;
3      if (getState() >= 0 && (t = thread) != null && !t.isInterrupted()) {
4          try {
5              t.interrupt();
6          } catch (SecurityException ignore) {
7          }
8      }
9  }
```

线程池为什么一定得是阻塞队列?

线程池中的线程在运行过程中，执行完创建线程时绑定的第一个任务后，就会不断的从队列中获取任务并执行，那么如果队列中没有任务了，线程为了不自然消亡，就会阻塞在获取队列任务时，等着队列中有任务过来就会拿到任务从而去执行任务。

通过这种方法能最终确保，线程池中能保留指定个数的核心线程数，关键代码为：

```
1  try {
2      Runnable r = timed ?
3          workQueue.poll(keepAliveTime, TimeUnit.NANOSECONDS) :
4          workQueue.take();
5      if (r != null)
6          return r;
```

```
    timedOut = true;
} catch (InterruptedException retry) {
    timedOut = false;
}
```

某个线程在从队列获取任务时，会判断是否使用超时阻塞获取，我们可以认为非核心线程会poll()，核心线程会take()，非核心线程超过时间还没获取到任务后面就会自然消亡了。

线程发生异常，会被移出线程池吗?

答案是会的，那有没有可能核心线程数在执行任务时都出错了，导致所有核心线程都被移出了线程池?

在源码中，当执行任务时出现异常时，最终会执行processWorkerExit()，执行完这个方法后，当前线程也就自然消亡了，但是！processWorkerExit()方法中会额外再新增一个线程，这样就能维持住固定的核心线程数。

Tomcat是如何自定义线程池的?

Tomcat中用的线程池为org.apache.tomcat.util.threads.ThreadPoolExecutor，注意类名和JUC下的一样，但是包名不一样。

Tomcat会创建这个线程池：

```
public void createExecutor() {
    internalExecutor = true;
    TaskQueue taskqueue = new TaskQueue();
    TaskThreadFactory tf = new TaskThreadFactory(getName() + "-exec-", daemon, getThreadPriority());
    executor = new ThreadPoolExecutor(getMinSpareThreads(), getMaxThreads(), 60, TimeUnit.SECONDS,taskqueue, tf);
    taskqueue.setParent( (ThreadPoolExecutor) executor);
}
```

注入传入的队列为TaskQueue，它的入队逻辑为：

```
public boolean offer(Runnable o) {
    //we can't do any checks
    if (parent==null) {
        return super.offer(o);
        }

        //we are maxed out on threads, simply queue the object
        if (parent.getPoolSize() == parent.getMaximumPoolSize()) {
        return super.offer(o);
        }

        //we have idle threads, just add it to the queue
        if (parent.getSubmittedCount()<=(parent.getPoolSize())) {
        return super.offer(o);
        }

        //if we have less threads than maximum force creation of a new thread
        if (parent.getPoolSize()<parent.getMaximumPoolSize()) {
        return false;
        }

        //if we reached here, we need to add it to the queue
        return super.offer(o);
}
```

特殊在：

- 入队时，如果线程池的线程个数等于最大线程池数才入队
- 入队时，如果线程池的线程个数小于最大线程池数，会返回false，表示入队失败

这样就控制了，Tomcat的这个线程池，在提交任务时：

1. 仍然会先判断线程个数是否小于核心线程数，如果小于则创建线程
2. 如果等于核心线程数，会入队，但是线程个数小于最大线程数会入队失败，从而会去创建线程

所以随着任务的提交，会优先创建线程，直到线程个数等于最大线程数才会入队。

当然其中有一个比较细的逻辑是：在提交任务时，如果正在处理的任务数小于线程池中的线程个数，那么也会直接入队，而不会去创建线程，也就是上面源码中getSubmittedCount的作用。

线程池的核心线程数、最大线程数该如何设置?

我们都知道，线程池中有两个非常重要的参数：

1. corePoolSize：核心线程数，表示线程池中的常驻线程的个数
2. maximumPoolSize：最大线程数，表示线程池中能开辟的最大线程个数

那这两个参数该如何设置呢?

我们对线程池负责执行的任务分为三种情况：

1. CPU密集型任务，比如找出1-1000000中的素数
2. IO密集型任务，比如文件IO、网络IO
3. 混合型任务

CPU密集型任务的特点时，线程在执行任务时会一直利用CPU，所以对于这种情况，就尽可能避免发生线程上下文切换。

比如，现在我的电脑只有一个CPU，如果有两个线程在同时执行找素数的任务，那么这个CPU就需要额外的进行线程上下文切换，从而达到线程并行的效果，此时执行这两个任务的总时间为：

任务执行时间*2+线程上下文切换的时间

而如果只有一个线程，这个线程来执行两个任务，那么时间为：

任务执行时间*2

所以对于CPU密集型任务，线程数最好就等于CPU核心数，可以通过以下API拿到你电脑的核心数：

```
Runtime.getRuntime().availableProcessors()
```

只不过，为了应对线程执行过程发生缺页中断或其他异常导致线程阻塞的请求，我们可以额外在多设置一个线程，这样当某个线程暂时不需要CPU时，可以有替补线程来继续利用CPU。

所以，对于CPU密集型任务，我们可以设置线程数为：**CPU核心数+1**

我们在来看IO型任务，线程在执行IO型任务时，可能大部分时间都阻塞在IO上，假如现在有10个CPU，如果我们只设置了10个线程来执行IO型任务，那么很有可能这10个线程都阻塞在了IO上，这样这10个CPU就都没活干了，所以，对于IO型任务，我们通常会设置线程数为：**2*CPU核心数**

不过，就算是设置为了**2*CPU核心数**，也不一定是最佳的，比如，有10个CPU，线程数为20，那么也有可能这20个线程同时阻塞在了IO上，所以可以再增加线程，从而去压榨CPU的利用率。

通常，如果IO型任务执行的时间越长，那么同时阻塞在IO上的线程就可能越多，我们就可以设置更多的线程，但是，线程肯定不是越多越好，我们可以通过以下这个公式来进行计算：
线程数 = CPU核心数 *（ 1 + 线程等待时间 / 线程运行总时间 ）

- 线程等待时间：指的就是线程没有使用CPU的时间，比如阻塞在了IO
- 线程运行总时间：指的是线程执行完某个任务的总时间

我们可以利用jvisualvm抽样来估计这两个时间：

图中表示，在刚刚这次抽样过程中，run()总共的执行时间为538948ms，利用了CPU的时间为86873ms，所以没有利用CPU的时间为538948ms-86873ms。

所以我们可以计算出：
线程等待时间 = 538948ms-86873ms
线程运行总时间 = 538948ms

所以：线程数 = 8 *（ 1 + （538948ms-86873ms） / 538948ms ）= 14.xxx

所以根据公式算出来的线程为14、15个线程左右。

按上述公式，如果我们执行的任务IO密集型任务，那么：线程等待时间 = 线程运行总时间，所以：
线程数 = CPU核心数 *（ 1 + 线程等待时间 / 线程运行总时间 ）
= CPU核心数 *（ 1 + 1 ）

= CPU核心数 * 2

以上只是理论，实际工作中情况会更复杂，比如一个应用中，可能有多个线程池，除开线程池中的线程可能还有很多其他线程，或者除开这个应用还是一些其他应用也在运行，所以实际工作中如果要确定线程数，最好是压测。

比如我写了一个：

```
@RestController
public class ZhouyuController {

    @GetMapping("/test")
    public String test() throws InterruptedException {
        Thread.sleep(1000);
        return "zhouyu";
    }

}
```

这个接口会执行1s，我现在利用apipost来压：

这是在Tomcat默认最大200个线程的请求下的压测结果。

当我们把线程数调整为500：

```
server.tomcat.threads.max=500
```

发现执行效率提高了一倍，假如再增加线程数到1000：

性能就降低了。

总结，我们再工作中，对于：

1. CPU密集型任务：CPU核心数+1，这样既能充分利用CPU，也不至于有太多的上下文切换成本
2. IO型任务：建议压测，或者先用公式计算出一个理论值（理论值通常都比较小）
3. 对于核心业务（访问频率高），可以把核心线程数设置为我们压测出来的结果，最大线程数可以等于核心线程数，或者大一点点，比如我们压测时可能会发现500个线程最佳，但是600个线程时也还行，此时600就可以为最大线程数
4. 对于非核心业务（访问频率不高），核心线程数可以比较小，避免操作系统去维护不必要的线程，最大线程数可以设置为我们计算或压测出来的结果。