一，Thread和Runnable

1.Thread类是如何开启一个新的线程的

```
    public synchronized void start() {

        if (threadStatus != 0)
            throw new IllegalThreadStateException();

        group.add(this);

        boolean started = false;
        try {
            start0();
            started = true;
        } finally {
            try {
                if (!started) {
                    group.threadStartFailed(this);
                }
            } catch (Throwable ignore) {

            }
        }
    }
```

可以看到Thread类里面的start()方法调用了start0()方法，

```
private native void start0();
```

而这个start0()是一个本地方法，这个方法会调用底层c的方法来根据操作系统执行操作，windows下会创建一个新的线程，linux下会创建一个轻量级的进程。

2.实现runnable接口为什么可以开启一个新的线程?

首先来看Thread类

```
class Thread implements Runnable
```

Thread类实现了Runnable接口，那说明他一定实现了run()方法。

```
    private Runnable target;

    public Thread(Runnable target) {
        init(null, target, "Thread-" + nextThreadNum(), 0);
    }
```

其次，Thread类里面声明了一个Runnable类型的成员变量target。

```
    @Override
    public void run() {
        if (target != null) {
            target.run();
        }
    }
```

当我们在执行run()方法的时候，他会先判断我们是否传入了一个Runnable接口，如果我们传入进来了一个Runnable接口，也就是target！=null，此时我们会执行target的run()，也就是我们通过实现Runnable接口的方式创建了一个多线程。

二，Future和FutureTask的关系

1.Future

首先来看一段代码：

```
/**
 * @author yhd
 * @createtime 2020/10/4 13:41
 */
public class Main {

    public static void main(String[] args) {
        ExecutorService pool = Executors.newCachedThreadPool();
        Future<?> submit = pool.submit(new RunnableTask());
        Future submit1 = pool.submit(new CallableTask());
    }

    private  static class RunnableTask implements Runnable{
        @Override
        public void run() {

        }
    }

    private static class CallableTask implements Callable{
        @Override
        public Object call() throws Exception {
            return null;
        }
    }
}
```

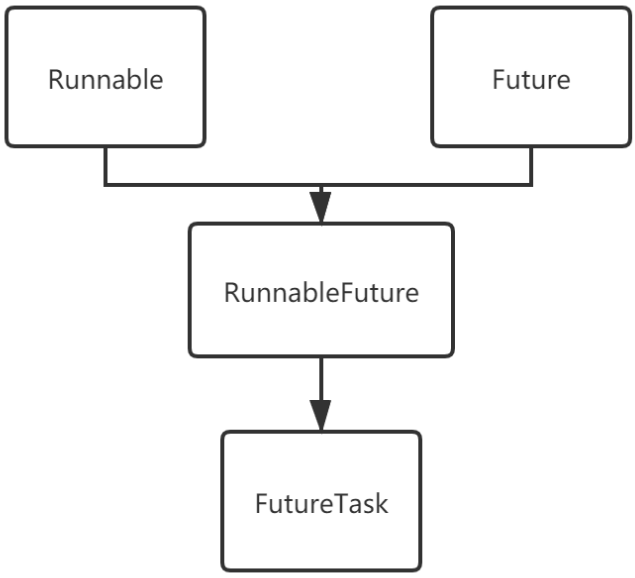
当我们调用线程池的submit()方法的时候，会给我们返回一个Future类型的对象。那么这个Future又是什么?

```
public interface Future<V> {

    //取消任务
    boolean cancel(boolean mayInterruptIfRunning);

   //是否取消了任务
    boolean isCancelled();
```

```
    //是否执行完了任务
    boolean isDone();
    //获取线程的执行结果
    V get() throws InterruptedException, ExecutionException;

   //获取结果超时
    V get(long timeout, TimeUnit unit)
        throws InterruptedException, ExecutionException, TimeoutException;
}
```

可以说，这个接口主要的功能就是可以获取到线程执行完成后的结果。

2.两者的关系



FutureTask可以看作是Runnable和Future两个接口的实现类

三，FutureTask源码

1.属性

```
    //表示当前任务的状态
    private volatile int state;
    //任务尚未执行
    private static final int NEW          = 0;
    //任务还未结束
    private static final int COMPLETING   = 1;
    //任务正常执行结束
    private static final int NORMAL       = 2;
    //任务发生了异常，由callable.call()向上抛出
    private static final int EXCEPTIONAL  = 3;
    //任务被取消
    private static final int CANCELLED    = 4;
    //任务正在中断
    private static final int INTERRUPTING = 5;
    //任务已经被中断
    private static final int INTERRUPTED  = 6;

    //submit(runnable/callable) 使用装饰者模式将runnable装饰成callable
    private Callable<V> callable;
```

```
    //正常情况下：用来保存call的执行结果，异常情况下：用来保存call抛出的异常
    private Object outcome; // non-volatile, protected by state reads/writes
    //执行当前任务的线程
    private volatile Thread runner;
    //因为会有很多线程去get当前任务的结果，所以 这里使用了一种数据结构 stack 头插 头取 的一个队列。
    private volatile WaitNode waiters;
```

2.WaitNode内部类

```
    static final class WaitNode {
        volatile Thread thread;
        volatile WaitNode next;
        WaitNode() { thread = Thread.currentThread(); }
    }
```

这个类里面封装着执行任务的线程和指向下一个线程的指针。

3.构造器

当我们传入一个runnable接口的时候：

```
    public FutureTask(Runnable runnable, V result) {
        this.callable = Executors.callable(runnable, result);
        this.state = NEW;       // ensure visibility of callable
    }
```

调用了Executors.callable(runnable, result);

```
    public static <T> Callable<T> callable(Runnable task, T result) {
        if (task == null)
            throw new NullPointerException();
        return new RunnableAdapter<T>(task, result);
    }
```

当任务不为空，实际上返回的是一个RunnableAdapter对象。

```
    static final class RunnableAdapter<T> implements Callable<T> {
        final Runnable task;
        final T result;
        RunnableAdapter(Runnable task, T result) {
            this.task = task;
            this.result = result;
        }
        public T call() {
            task.run();
            return result;
        }
    }
```

在这里，将runnable装饰成了callable。

4.run()

```
    //任务执行入口
    public void run() {
        /**
         * if(如果当前任务已经执行过，或者当前任务被取消了 || cas失败，当前任务被其他线程抢
占){
         *       线程就不处理了，直接返回
         * }
         */
        if (state != NEW ||
            !UNSAFE.compareAndSwapObject(this, runnerOffset,
                                         null, Thread.currentThread()))
            return;
        try {
            /**
             * 前提：当进入到这里的时候，说明任务的状态一定是NEW，而且当前线程cas抢占任务成功
             * @param callable 我们自己封装的业务逻辑 也就是我们传入进来的
callable/runnable
             * @param result 接受任务的结果
             * @param ran    true：call()执行代码成功，没抛出异常，false： call()执行代
码失败，抛出异常
             *     try{
             *         if(如果当前任务不为空，其实是为了防止空指针 || 当前任务状态为NEW，
防止当前任务被取消){
             *                 try{
             *                    执行call方法并用result来接收结果
             *                    将ran设置为true，表示call顺利执行完。
             *                 }catch{
             *                    结果设置为null
             *                    ran设置为false 表示call执行过程中发生了异常
             *                    setException(ex);  TODO
             *                 }
             *                 if(call顺利执行完){
             *                    //set就是设置结果到outcome
             *                    set(result); TODO
             *                 }
             *         }
             *      }finally{
             *                 将当前线程置为空
             *                 重新获取当前任务的状态
             *                 if(当前任务已经被中断){
             *                    handlePossibleCancellationInterrupt(s); TODO
             *                 }
             *       }
             */
            Callable<V> c = callable;
            if (c != null && state == NEW) {
                V result;
                boolean ran;
                try {
                    result = c.call();
                    ran = true;
                } catch (Throwable ex) {
                    result = null;
                    ran = false;
                    setException(ex);
                }
                if (ran)
                    set(result);
```

```
            }
        } finally {
            // runner must be non-null until state is settled to
            // prevent concurrent calls to run()
            runner = null;
            // state must be re-read after nulling runner to prevent
            // leaked interrupts
            int s = state;
            if (s >= INTERRUPTING)
                handlePossibleCancellationInterrupt(s);
        }
    }
```

5.setException（）

```
    /**
     * 如果当前任务执行过程中发生异常，就会执行这个方法
     * if(使用cas的方式将当前任务的状态设置为完成中成功){
     *     outcome来接收异常
     *     将当前任务的状态设置为发生异常
     *     finishCompletion();  TODO
     * }
     * @param t call向上抛出的异常
     */
    protected void setException(Throwable t) {
        if (UNSAFE.compareAndSwapInt(this, stateOffset, NEW, COMPLETING)) {
            outcome = t;
            UNSAFE.putOrderedInt(this, stateOffset, EXCEPTIONAL); // final state
            finishCompletion();
        }
    }
```

6.set()

```
    /**
     * if(cas的方式设置当前任务的状态为完成中){
     *     outcome接收任务的结果
     *     设置当前任务的状态为正常完成
     *     finishCompletion();  TODO
     * }
     * @param v 任务的结果
     */
    protected void set(V v) {
        if (UNSAFE.compareAndSwapInt(this, stateOffset, NEW, COMPLETING)) {
            outcome = v;
            UNSAFE.putOrderedInt(this, stateOffset, NORMAL); // final state
            finishCompletion();
        }
    }
```

7.handlePossibleCancellationInterrupt（）

```
    /**
     *
```

```
     * @param s 当前任务的状态
     *          if(如果当前任务的状态为被打断){
     *              自旋判断如果当前线程的状态为被打断{
     *                      让执行当前任务的线程释放cpu的抢占
     *              }
     *          }
     *
     */
    private void handlePossibleCancellationInterrupt(int s) {

        if (s == INTERRUPTING)
            while (state == INTERRUPTING)
                Thread.yield(); // wait out pending interrupt

    }
```

8.get()

```
    /**
     * 获取当前任务的状态
     * if(状态小于等于未完成){
     *      awaitDone  TODO
     * }
     * //如果当前任务已经完成
     * return report(s);  TODO
     * @return
     * @throws InterruptedException
     * @throws ExecutionException
     */
    public V get() throws InterruptedException, ExecutionException {
        int s = state;
        if (s <= COMPLETING)
            s = awaitDone(false, 0L);
        return report(s);
    }
```

9.awaitDone()

```
    /**
     * deadline=0 说明不带超时
     * 引用当前线程封装成WaitNode对象
     * @param timed 是否设置了超时
     * @param nanos 超时时间
     * @return
     * @throws InterruptedException
     */
    private int awaitDone(boolean timed, long nanos)
        throws InterruptedException {
        //deadline=0 说明不带超时
        final long deadline = timed ? System.nanoTime() + nanos : 0L;
        //引用当前线程封装成WaitNode对象
        WaitNode q = null;
        //当前线程是否进入获取结果的等待队列
        boolean queued = false;
        //自旋
```

```
        for (;;) {
            //如果当前线程被其他线程用中断的方式唤醒
            //这种唤醒方式会将Thread的中断标记位设置为false
            if (Thread.interrupted()) {
                //当前线程node出队
                removeWaiter(q);
                //get方法抛出中断异常
                throw new InterruptedException();
            }

            //假设当前线程是被其它线程 使用unpark(thread) 唤醒的话。会正常自旋，走下面逻辑。

            //获取当前任务的状态
            int s = state;
            //条件成立说明当前任务已经有结果了 可能正常执行结束，也可能抛出异常结束
            if (s > COMPLETING) {
                //条件成立说明已经为当前线程创建过node，
                // 此时需要将node设置位null help GC
                if (q != null)
                    q.thread = null;
                //直接返回当前的状态
                return s;
            }
            //如果任务尚未执行完，接着等
            else if (s == COMPLETING) // cannot time out yet
                Thread.yield();
            //第一次自旋的时候，尚未给当前线程创建waitNode对象，此时就需要位当前线程创建waitNode对象
            else if (q == null)
                q = new WaitNode();
            //第二次自旋，创建好了对象还没有入队，cas的方式入队
            else if (!queued)
                queued = UNSAFE.compareAndSwapObject(this, waitersOffset,
            //第三次自旋 ，判断是否设置超时时间
  q.next = waiters, q);
            else if (timed) {
                nanos = deadline - System.nanoTime();
                if (nanos <= 0L) {
                    removeWaiter(q);
                    return state;
                }
                LockSupport.parkNanos(this, nanos);
            }
            //如果没设置超时时间
            //当前get操作的线程就会被park了。  线程状态会变为 WAITING状态，相当于休眠了..
            //除非有其它线程将你唤醒  或者 将当前线程 中断。
            else
                LockSupport.park(this);
        }
    }
```

10.report()

```
    private V report(int s) throws ExecutionException {
        Object x = outcome;
        //如果任务正常执行结束
        if (s == NORMAL)
            //返回结果
            return (V)x;
        //如果任务被取消或者中断了
        if (s >= CANCELLED)
            //抛出异常
            throw new CancellationException();
        //否则就说明，任务发生了异常，直接将异常往上抛
        throw new ExecutionException((Throwable)x);
    }
```

11.finishCompletion()

```
    private void finishCompletion() {
        //q指向waiters 链表的头结点。
        for (WaitNode q; (q = waiters) != null;) {
            //使用cas设置 waiters 为 null 是因为怕 外部线程使用 cancel 取消当前任务 也会触发finishCompletion方法。 小概率事件。
            if (UNSAFE.compareAndSwapObject(this, waitersOffset, q, null)) {
                //自旋
                for (;;) {
                    //获取当前节点封装的线程
                    Thread t = q.thread;
                    //如果当前节点的线程不为空
                    if (t != null) {
                        //help GC
                        q.thread = null;
                        //唤醒当前节点对应的线程
                        LockSupport.unpark(t);
                    }
                    //next 当前节点的下一个节点
                    WaitNode next = q.next;
                    //如果已经到了最后一个节点，跳出自旋
                    if (next == null)
                        break;
                    q.next = null; // unlink to help gc
                    q = next;
                }
                break;
            }
        }

        done();
        //将callable 设置为null helpGC
        callable = null;        // to reduce footprint
    }
```

11.cancel()

```
    /**
     * 取消当前任务
     * if(
```

```
     *       !(
     *           state == NEW 成立 表示当前任务处于运行中 或者 处于线程池 任务队列中..
     *           &&
     *           修改状态成功
     * )
     * ){
     *       说明不能取消，直接返回false。
     * }
     * 否则执行下面的逻辑代码
     * （此时的前提条件，任务正在运行中或处于等待队列 && 修改状态成功）
     * try{
     *     if(尝试去打断成功){
     *
     *         try{
     *              （
     *             t=null 当前线程正处于等待队列
     *             t!=null 当前线程正在执行中
     *             ）
     *             if(如果当前线程正在执行){
     *                 给runner线程一个中断信号..
     *                 如果你的程序是响应中断 会走中断逻辑..
     *                 假设你程序不是响应中断的..啥也不会发生。
     *             }
     *         }finally{
     *             设置线程状态为中断
     *         }
     *     }
     * }finally{
     *     唤醒获取结果的线程
     * }
     * 返回 取消成功
     * @param mayInterruptIfRunning
     * @return
     */
    public boolean cancel(boolean mayInterruptIfRunning) {
        if (!(state == NEW &&
              UNSAFE.compareAndSwapInt(this, stateOffset, NEW,
                  mayInterruptIfRunning ? INTERRUPTING : CANCELLED)))
            return false;
        try {    // in case call to interrupt throws exception
            if (mayInterruptIfRunning) {
                try {
                    Thread t = runner;
                    if (t != null)
                        t.interrupt();
                } finally { // final state
                    UNSAFE.putOrderedInt(this, stateOffset, INTERRUPTED);
                }
            }
        } finally {
            finishCompletion();
        }
        return true;
    }
```

四，思路整理

1.首先明确一点，不管我们传入的是Runnable还是Callable接口，他最终用的都是Callable，Runnable接口会被他通过装饰者模式封装为Callable。

2.一个任务执行的入口其实就是run方法。

3.进入run方法，首先他会判断当前任务是否已经被执行过或者当前线程通过cas的方式并没有抢到执行当前任务的机会。如果是的话，说明已经有线程正在执行或者执行完了当前的任务，直接返回即可。

4.接下来他会判断当前任务是否为空或者当前任务的状态，其实判断状态就是为了判断当前任务是不是被取消了。如果任务不为空并且没有被取消，他会执行call方法（实际上就是我们自己的业务代码）并将结果设置到result将任务的是否被执行状态改成true表示顺利执行完。

5.call方法执行过程中如果发生异常了，会将结果设置为null，是否被顺利执行的状态为设置为false，表示执行过程发生了异常。

6.接下来，它会将异常信息封装到outcome，然后设置但该你任务的状态为异常结束，并自旋的方式唤醒所有等待队列中等待获取结果的线程。

7.如果call正常执行完了，他会用cas的方式设置当前任务的完成状态为完成中，如果设置成功，outcome来接收任务的结果，设置当前任务的状态为正常完成状态，并且唤醒所有等待结果的线程。

8.最终它会将执行当前任务的线程设置为null，并判断，如果当前任务的状态是被打断中或者已经被打断，他就会自旋判断如果当前线程的状态为被打断，让执行当前任务的线程释放cpu。

9.get方法是获取当前任务的结果。首先他会获取当前任务的状态，如果状态小于等于未完成首先他会通过自旋的方式，第一次自旋尚未给当前线程创建waitNode对象，此时就需要位当前线程创建waitNode对象。第二次自旋，创建好了对象还没有入队，cas的方式入队。第三次自旋，判断是否设置超时时间，如果没设置超时时间，当前get操作的线程就会被park了。 线程状态会变为 WAITING状态，相当于休眠了..除非有其它线程将你唤醒 或者 将当前线程 中断。他会获取当前线程的任务状态，如果任务还没有完成，释放cpu接着等。如果任务已经完成，此时需要将node设置位null help GC，直接返回当前的状态。除此之外还要判断，如果当前线程被其他线程用中断的方式唤醒，这种唤醒方式会将Thread的中断标记位设置为false，当前线程出队，get方法抛出中断异常。

10.如果状态表示已经有结果，会执行report方法。

11.report方法，如果任务正常执行结束，返回结果，如果任务被取消或者中断了，抛出异常，如果任务执行过程中发生异常结束了，返回异常。

12.最后就是任务的取消方法 cancel。他会先判断state == NEW 成立 表示当前任务处于运行中 或者 处于线程池 任务队列中..并且cas修改状态成功，他就会尝试取打断。

13.如果尝试打断成功，给runner线程一个中断信号..如果你的程序是响应中断 会走中断逻辑..假设你程序不是响应中断的..啥也不会发生。最后，设置线程状态为中断，唤醒获取结果的线程。