31 凭票取餐—Future模式详解

更新时间：2019-12-12 09:40:27

> 与有肝胆人共事，从无字句处读书。
>
> ——周恩来

从本节开始，我们进入新的一章学习，同时也是最后一章的学习。我们从如何实现一个线程开始学起，学习了并发的问题和解决办法，学习了线程池等工具的使用，学习了各种并发容器。本章将会讲解实际开发中经常会用到的多线程设计模式及其在 JDK 中的实现和应用。

本节我们要学习的是 Future 模式。我们先来看一个例子，假如你中午要出去买一份午餐打包带回家，并且要去超市买一管牙膏，应该怎么做才会时间最短？当然是点好外卖，然后去超市买牙膏，等你回来看外卖是否已经做好了，如果做好了，拿小票取餐。如果还没好，那就继续等待，等做好后取餐回家。

如果程序不使用多线程实现的话，那么主线程就会阻塞在外卖加工过程上，直到午餐做好，才能去超市买东西。但如果我们采用多线程，可以点餐后马上去超市买牙膏，同时有新的线程加工你的午餐。今天我们来学习一种新的多线程应用模式 Future，解决起类似问题就容易多了。

1、Future 模式介绍

我们先不着急讲解 Future，先来回顾下之前我们讲解的 Thread 和 runnable，实现多线程的方式是新起线程运行 run 方法，但是 run 方法有个缺陷是没有返回值，并且主线程也并不知道新的线程何时运行完毕。上文的例子，我们不但需要做饭的线程返回午餐，并且主线程需要知道午餐已经好了。使用我们之前学习知识，通过 wait、notify 和共享资源也可以实现，但会比较复杂。其实 JDK 提供了非常方便的工具就是 Future。Future 持有要运行的任务，以及任务的结果。主线程只要声明了 Future 对象，并且启动新的线程运行他。那么随时能通过 Future 对象获取另外线程运行的结果。

接下来我们看看 Future 如何实现例子中的场景。

2、Future 使用

上述例子的代码如下：

```
public class Client {
    public static void main(String[] args) throws ExecutionException, InterruptedException {

        FutureTask<String> cookTask = new FutureTask<>(new Callable<String>() {
            @Override
            public String call() throws Exception {
                Thread.sleep(3000);
                return "5斤的龙虾";
            }
        });

        Long startTime = System.currentTimeMillis();

        System.out.println("我点了5斤的龙虾。");
        new Thread(cookTask).start();

        System.out.println("我去买牙膏。");
        TimeUnit.SECONDS.sleep(2);
        System.out.println("我买到牙膏了！");

        String lunch = cookTask.get();
        System.out.println("我点的"+lunch+"已经OK了！");

        Long userTime = (System.currentTimeMillis() - startTime)/1000;
        System.out.println("我一共用了"+userTime+"秒买午餐并且买牙膏。");
    }
}
```

代码中先了一个 FutureTask 对象，称之为 cookTask。顾名思义，这个 task 是用来做饭的。可以看到构造方法中传入 Callable 的实现。实现的 call 方法中模拟做饭用了 3 秒钟。

主线程运行后，先点了 5 斤的龙虾，然后一个新的线程就开始去执行 cookTask 了。等会儿，到这里你一定会问，Thread 构造方法需要传入 Runnable 的实现啊？没错，FutureTask 实现了 Runnable 接口。FutureTask 的 run 方法实际执行的是 Callable 的 call 方法。那么新的线程 start 后，实际做饭的逻辑会被执行：自线程 sleep3 秒后返回 “5 斤的龙虾”。

主线程在启动做饭的自线程后继续向下执行，去买牙膏。这里 sleep 两秒，模拟买牙膏的时间消耗。

买到牙膏接下来的一行代码 String lobster = cookTask.get (); 重点说一下，此时分两种情况：

1. cookTask 运行的线程已经结束了，那么可以直接取到运行的结果赋值给 lunch；
2. cookTask 运行的线程还没有执行结束，此时主线程会阻塞，直到能取得运行结果。

cookTask 就是你的购物小票，只要你没弄丢，随时能去取你的午饭。

程序最后计算了整个过程的执行时间。由于采用了多线程并发，所以执行时间应该等于耗时最长的那个任务。这个例子中做龙虾 3 秒 > 买牙膏 2 秒，所以总共耗时 3 秒，输出如下：

```
我点了5斤的龙虾
我去买牙膏
我买到牙膏了！
我点的5斤的龙虾已经OK了
我一共用了3秒买午餐并且买牙膏
```

加入我调整买牙膏需要 10 秒，那么输出则如下：

```
我点了5斤的龙虾
我去买牙膏
我买到牙膏了！
我点的5斤的龙虾已经OK了
我一共用了10秒买午餐并且买牙膏
```

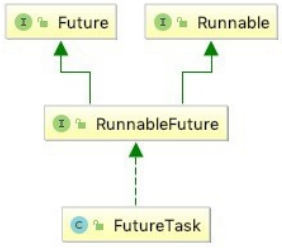
总共耗时 10 秒。

现在我们想一下，假如单线程串行执行，点完午餐必须等待午餐做好了，才能去买牙膏。那么永远耗时都是 2 者之和。采用并发执行后，仅为时间较长的那个任务的时间。

由于我们调用 Future 的 get 方法后主线程就开始阻塞了，所以我们应该在真正需要使用 Future 对象的返回结果时才去调用，充分利用并发的特性来提升程序性能。

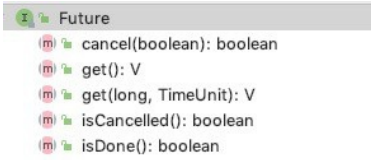
3、Future 源码解析

Future 是一个接口，而 FutrueTask 则是他的实现，我们看一下它们的继承关系：



FutureTask 不但实现了 Future 而且实现了 Runnable 接口。这也是为什么它能作为参数传入 Thread 构造方法。

Runnable 接口我们讲过，里面只有一个 run 方法，用于被 Thread 调用。我们看一下 Future 接口有哪些方法：



cancel 用于尝试取消任务。

get 用于等待并获取任务执行结果。带时间参数的 get 方法只会等待指定时间长度。

isCancelled 返回任务在完成前是否已经被取消。

isDone 返回任务是否完成。

我们用到最多的就是 get 方法，获取任务的执行结果。

3.1 FutureTask 构造方法

```
public FutureTask(Callable<V> callable) {
    if (callable == null)
        throw new NullPointerException();
    this.callable = callable;
    this.state = NEW;       // ensure visibility of callable
}
```

需要传入 Callable 的实现，Callable 是一个接口，定义了 call 方法，返回 V 类型。

然后定义了 FutureTask 的状态为 NEW。FutrueTask 定义了如下状态：

```
private static final int NEW          = 0;
private static final int COMPLETING   = 1;
private static final int NORMAL       = 2;
private static final int EXCEPTIONAL  = 3;
private static final int CANCELLED    = 4;
private static final int INTERRUPTING = 5;
private static final int INTERRUPTED  = 6;
```

通过字面我们很容易理解其含义。

3.2 run 方法解析

FutrueTask 实现了 Runnbale 接口，所以 Thread 运行后实际上执行的是 FutrueTask 的 run 方法。我们要想了解 Future 的实现原理，那么就应该从它的 run 方法开始入手。

```
public void run() {
  //如果此时状态不为NEW直接结束
  //如果为NEW，但是CAS操作把本线程写入为runner时，发现runner已经不为null，那么也直接结束
    if (state != NEW ||
        !UNSAFE.compareAndSwapObject(this, runnerOffset,
                                     null, Thread.currentThread()))
        return;
    try {
      //取得Callable对象
        Callable<V> c = callable;
        if (c != null && state == NEW) {
            V result;
            boolean ran;
            try {
              //运行Callable对象的call方法，并且取得返回值。
                result = c.call();
                ran = true;
            } catch (Throwable ex) {
                result = null;
                ran = false;
                setException(ex);
            }
          //如果call方法成功执行结束，那么把执行结果设置给成员变量outcome;
            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);
    }
}
```

核心逻辑就是执行运行 Callable 对象的 call 方法，把返回结果写入 outcome。outcome 用来保存计算结果。

保存计算结果则是通过 set 方法。

3.3 set 方法解析

set 方法代码如下：

```
protected void set(V v) {
  //状态还是NEW，保存计算结果给outcome
    if (UNSAFE.compareAndSwapInt(this, stateOffset, NEW, COMPLETING)) {
        outcome = v;
      //更新状态为NORMAL
        UNSAFE.putOrderedInt(this, stateOffset, NORMAL); // final state
      //唤醒等待的线程
        finishCompletion();
    }
}
```

如果没有被取消则会保存计算结果 v 到 outcome。然后更新最终状态为 NORMAL。最后调用 finishCompletion 方法唤醒阻塞的线程。代码如下：

```
private void finishCompletion() {
    // assert state > COMPLETING;
  //遍历等待线程，结束等待
    for (WaitNode q; (q = waiters) != null;) {
        if (UNSAFE.compareAndSwapObject(this, waitersOffset, q, null)) {
            for (;;) {
              //结束等待线程的挂起
                Thread t = q.thread;
                if (t != null) {
                    q.thread = null;
                    LockSupport.unpark(t);
                }
              //如果没有下一个等待线程，那么结束循环
                WaitNode next = q.next;
                if (next == null)
                    break;
                q.next = null; // unlink to help gc
                q = next;
            }
            break;
        }
    }
  //全部完成后回调FutrueTask的done方法。done方法为空，可以由子类实现。
    done();
  //清除callable
    callable = null;        // to reduce footprint
}
```

3.4 get 方法解析

get 方法用于获取任务的返回值，如果还没有执行完成，则会阻塞，代码如下：

```
public V get() throws InterruptedException, ExecutionException {
    //获取当前Task的状态
    int s = state;
    //如果还没有完成，则阻塞等待完成
    if (s <= COMPLETING)
        s = awaitDone(false, 0L);
  //获取任务执行的返回结果
    return report(s);
}
```

我们先来看 awaitDone 的代码：

```
private int awaitDone(boolean timed, long nanos)
    throws InterruptedException {
  //计算等待截止时长
    final long deadline = timed ? System.nanoTime() + nanos : 0L;
    WaitNode q = null;
    boolean queued = false;
    for (;;) {
      //当前线程如果被打断，则不再等待。从等待链表中移除
        if (Thread.interrupted()) {
            removeWaiter(q);
            throw new InterruptedException();
        }
   //取得目前的状态
        int s = state;
      //如果已经执行完成，清空q节点保存的线程
        if (s > COMPLETING) {
            if (q != null)
                q.thread = null;
            return s;
        }
      //如果正在执行，让出CPU执行权
        else if (s == COMPLETING) // cannot time out yet
            Thread.yield();
      //没有进入以上分支，运行到此分支，这说明此线程确实需要开始等待了，
      //那么如果还未为此线程建立关联的等待节点，则进行创建。
        else if (q == null)
            q = new WaitNode();
      //通过CAS把此线程的等待node加入到连表中。失败的话，下次循环若能运行到此分支，会继续添加。
        else if (!queued)
            queued = UNSAFE.compareAndSwapObject(this, waitersOffset,
                                                 q.next = waiters, q);
      //如果设置了超时，检查是否超时。超时的话结束等待。 否则挂起超时时长
      //如果没有设置超时时长，则永久挂起
      //回到上面的finishCompletion方法，等到task执行完成后会执行LockSupport.unpark(t)，结束阻塞。
      else if (timed) {
            nanos = deadline - System.nanoTime();
            if (nanos <= 0L) {
                removeWaiter(q);
                return state;
            }
            LockSupport.parkNanos(this, nanos);
        }
        else
            LockSupport.park(this);
    }
}
```

最后我们看一下 report 方法：

```
private V report(int s) throws ExecutionException {
  //获取执行结果
    Object x = outcome;
  //NORMAL为正常结束，那么直接把X转型后返回
    if (s == NORMAL)
        return (V)x;
  //如果任务被取消了，则抛出异常
    if (s >= CANCELLED)
        throw new CancellationException();
    throw new ExecutionException((Throwable)x);
}
```

outcome 保存的就是任务的执行结果。根据此时的状态，选择返回执行结果还是抛出取消的异常。

最后我们总结下 FutureTask 的代码：

1、FutureTask 实现 Runnable 和 Future 接口；

2、在线程上运行 FutureTask 后，run 方法被调用，run 方法会调用传入的 Callable 接口的 call 方法；

3、拿到返回值后，通过 set 方法保存结果到 outcome，并且唤醒所有等待的线程；

4、调用 get 方法获取执行结果时，如果没有执行完毕，则进入等待，直到 set 方法调用后被唤醒。

下图示意了两个线程运行 task 和 get 时的程序逻辑：

4、总结

Future 模式在实际开发中有着大量的应用场景。比如说微服务架构中，需要调用不同服务接口获取数据，但是接口调用间并无依赖关系，那么可以通过 FutureTask 并发调用，然后再执行后续逻辑。如果我们采用串行的方式，则需要一个接口返回后，再调用下一个接口。FutreTask 需要结合 Callable 接口使用，示例代码中为了让大家显示的看到 Callable 接口，所以采用匿名对象的方式。实际使用中我们可以使用 lambda 表达式来简化代码，如下：

```
FutureTask<String> cookTask = new FutureTask<>(() -> {
    Thread.sleep(3000);
    return "5斤的龙虾";
})
```

}

30 限量供应，不好意思您来晚了
—Semaphore详解

32 请按到场顺序发言—
Completion Service详解