23 | Future：如何用多线程实现最优的“烧水泡茶”程序？

2019-04-20 王宝令

在上一篇文章《22 | Executor与线程池：如何创建正确的线程池？》中，我们详细介绍了如何创建正确的线程池，那创建完线程池，我们该如何使用呢？在上一篇文章中，我们仅仅介绍了ThreadPoolExecutor的 void execute(Runnable command) 方法，利用这个方法虽然可以提交任务，但是却没有办法获取任务的执行结果（execute()方法没有返回值）。而很多场景下，我们又都是需要获取任务的执行结果的。那ThreadPoolExecutor是否提供了相关功能呢？必须的，这么重要的功能当然需要提供了。

下面我们就来介绍一下使用ThreadPoolExecutor的时候，如何获取任务执行结果。

如何获取任务执行结果

Java通过ThreadPoolExecutor提供的3个submit()方法和1个FutureTask工具类来支持获得任务执行结果的需求。下面我们先来介绍这3个submit()方法，这3个方法的方法签名如下。

```
// 提交Runnable任务
Future<?>
  submit(Runnable task);
// 提交Callable任务
<T> Future<T>
  submit(Callable<T> task);
// 提交Runnable任务及结果引用
<T> Future<T>
  submit(Runnable task, T result);
```

你会发现它们的返回值都是Future接口，Future接口有5个方法，我都列在下面了，它们分别是**取消任务的方法cancel()、判断任务是否已取消的方法isCancelled()、判断任务是否已结束的方法isDone()以及2个获得任务执行结果的get()和get(timeout, unit)**，其中最后一个get(timeout, unit)支持超时机制。通过Future接口的这5个方法你会发现，我们提交的任务不但能够获取任务执行结果，还可以取消任务。不过需要注意的是：这两个get()方法都是阻塞式的，如果被调用的时候，任务还没有执行完，那么调用get()方法的线程会阻塞，直到任务执行完才会被唤醒。

```
// 取消任务
boolean cancel(
  boolean mayInterruptIfRunning);
// 判断任务是否已取消
boolean isCancelled();
// 判断任务是否已结束
boolean isDone();
// 获得任务执行结果
get();
// 获得任务执行结果，支持超时
get(long timeout, TimeUnit unit);
```

这3个submit()方法之间的区别在于方法参数不同，下面我们简要介绍一下。

1. 提交Runnable任务 submit(Runnable task)：这个方法的参数是一个Runnable接口，Runnable接口的run()方法是没有返回值的，所以 submit(Runnable task) 这个方法返回的Future仅可以用来断言任务已经结束了，类似于Thread.join()。
2. 提交Callable任务 submit(Callable<T> task)：这个方法的参数是一个Callable接口，它只有一

个call()方法，并且这个方法是有返回值的，所以这个方法返回的Future对象可以通过调用其get()方法来获取任务的执行结果。

3. 提交Runnable任务及结果引用 submit(Runnable task, T result)：这个方法很有意思，假设这个方法返回的Future对象是f，f.get()的返回值就是传给submit()方法的参数result。这个方法该怎么用呢？下面这段示例代码展示了它的经典用法。需要你注意的是Runnable接口的实现类Task声明了一个有参构造函数 Task(Result r)，创建Task对象的时候传入了result对象，这样就能在类Task的run()方法中对result进行各种操作了。result相当于主线程和子线程之间的桥梁，通过它主子线程可以共享数据。

```
ExecutorService executor
  = Executors.newFixedThreadPool(1);
// 创建Result对象r
Result r = new Result();
r.setAAA(a);
// 提交任务
Future<Result> future =
  executor.submit(new Task(r), r);
Result fr = future.get();
// 下面等式成立
fr === r;
fr.getAAA() === a;
fr.getXXX() === x

class Task implements Runnable{
  Result r;
  //通过构造函数传入result
  Task(Result r){
    this.r = r;
  }
  void run() {
    //可以操作result
    a = r.getAAA();
    r.setXXX(x);
  }
}
```

下面我们再来介绍FutureTask工具类。前面我们提到的Future是一个接口，而FutureTask是一个

实实在在的工具类，这个工具类有两个构造函数，它们的参数和前面介绍的submit()方法类似，所以这里我就不再赘述了。

```
FutureTask(Callable<V> callable);
FutureTask(Runnable runnable, V result);
```

那如何使用FutureTask呢？其实很简单，FutureTask实现了Runnable和Future接口，由于实现了Runnable接口，所以可以将FutureTask对象作为任务提交给ThreadPoolExecutor去执行，也可以直接被Thread执行；又因为实现了Future接口，所以也能用来获得任务的执行结果。下面的示例代码是将FutureTask对象提交给ThreadPoolExecutor去执行。

```
// 创建FutureTask
FutureTask<Integer> futureTask
  = new FutureTask<>(()-> 1+2);
// 创建线程池
ExecutorService es =
  Executors.newCachedThreadPool();
// 提交FutureTask
es.submit(futureTask);
// 获取计算结果
Integer result = futureTask.get();
```

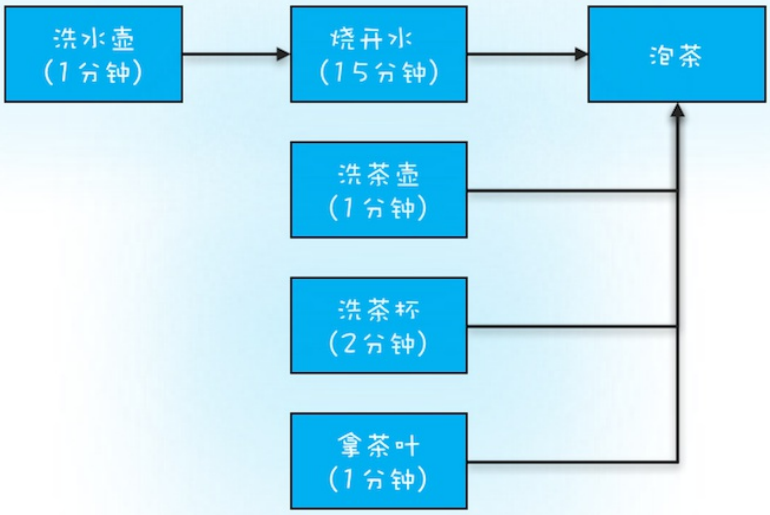
FutureTask对象直接被Thread执行的示例代码如下所示。相信你已经发现了，利用FutureTask对象可以很容易获取子线程的执行结果。

```
// 创建FutureTask
FutureTask<Integer> futureTask
  = new FutureTask<>(()-> 1+2);
// 创建并启动线程
Thread T1 = new Thread(futureTask);
T1.start();
// 获取计算结果
Integer result = futureTask.get();
```

实现最优的“烧水泡茶”程序

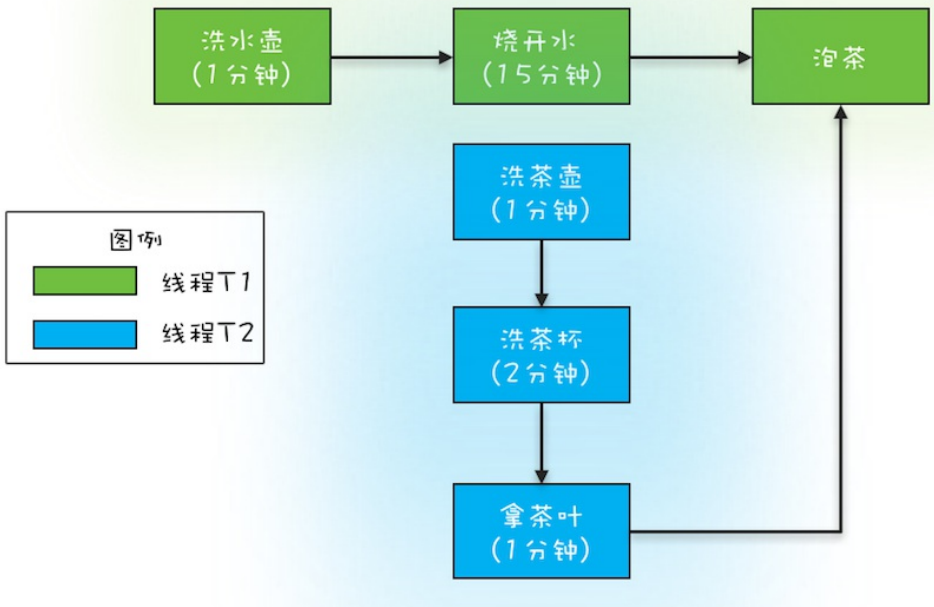
记得以前初中语文课文里有一篇著名数学家华罗庚先生的文章《统筹方法》，这篇文章里介绍了

一个烧水泡茶的例子，文中提到最优的工序应该是下面这样：



烧水泡茶最优工序

下面我们用程序来模拟一下这个最优工序。我们专栏前面曾经提到，并发编程可以总结为三个核心问题：分工、同步和互斥。编写并发程序，首先要做的就是分工，所谓分工指的是如何高效地拆解任务并分配给线程。对于烧水泡茶这个程序，一种最优的分工方案可以是下图所示的这样：用两个线程T1和T2来完成烧水泡茶程序，T1负责洗水壶、烧开水、泡茶这三道工序，T2负责洗茶壶、洗茶杯、拿茶叶三道工序，其中T1在执行泡茶这道工序时需要等待T2完成拿茶叶的工序。对于T1的这个等待动作，你应该可以想出很多种办法，例如Thread.join()、CountDownLatch，甚至阻塞队列都可以解决，不过今天我们用Future特性来实现。



下面的示例代码就是用这一章提到的Future特性来实现的。首先，我们创建了两个FutureTask——ft1和ft2，ft1完成洗水壶、烧开水、泡茶的任务，ft2完成洗茶壶、洗茶杯、拿茶叶的任务；这里需要注意的是ft1这个任务在执行泡茶任务前，需要等待ft2把茶叶拿来，所以ft1内部需要引用ft2，并在执行泡茶之前，调用ft2的get()方法实现等待。

```
// 创建任务T2的FutureTask
FutureTask<String> ft2
  = new FutureTask<>(new T2Task());
// 创建任务T1的FutureTask
FutureTask<String> ft1
  = new FutureTask<>(new T1Task(ft2));
// 线程T1执行任务ft1
Thread T1 = new Thread(ft1);
T1.start();
// 线程T2执行任务ft2
Thread T2 = new Thread(ft2);
T2.start();
// 等待线程T1执行结果
System.out.println(ft1.get());

// T1Task需要执行的任务：
// 洗水壶、烧开水、泡茶
class T1Task implements Callable<String>{
  FutureTask<String> ft2;
  // T1任务需要T2任务的FutureTask
  T1Task(FutureTask<String> ft2){
    this.ft2 = ft2;
  }
  @Override
  String call() throws Exception {
    System.out.println("T1:洗水壶...");
    TimeUnit.SECONDS.sleep(1);

    System.out.println("T1:烧开水...");
    TimeUnit.SECONDS.sleep(15);
    // 获取T2线程的茶叶
```

```
    // 获取T2线程的茶叶
    String tf = ft2.get();
    System.out.println("T1:拿到茶叶:"+tf);

    System.out.println("T1:泡茶...");
    return "上茶:" + tf;
  }
}
// T2Task需要执行的任务:
// 洗茶壶、洗茶杯、拿茶叶
class T2Task implements Callable<String> {
  @Override
  String call() throws Exception {
    System.out.println("T2:洗茶壶...");
    TimeUnit.SECONDS.sleep(1);

    System.out.println("T2:洗茶杯...");
    TimeUnit.SECONDS.sleep(2);

    System.out.println("T2:拿茶叶...");
    TimeUnit.SECONDS.sleep(1);
    return "龙井";
  }
}
// 一次执行结果：
T1:洗水壶...
T2:洗茶壶...
T1:烧开水...
T2:洗茶杯...
T2:拿茶叶...
T1:拿到茶叶:龙井
T1:泡茶...
上茶:龙井
```

总结

利用Java并发包提供的Future可以很容易获得异步任务的执行结果，无论异步任务是通过线程池ThreadPoolExecutor执行的，还是通过手工创建子线程来执行的。Future可以类比为现实世界里

的提货单，比如去蛋糕店订生日蛋糕，蛋糕店都是先给你一张提货单，你拿到提货单之后，没有必要一直在店里等着，可以先去干点其他事，比如看场电影；等看完电影后，基本上蛋糕也做好了，然后你就可以凭提货单领蛋糕了。

利用多线程可以快速将一些串行的任务并行化，从而提高性能；如果任务之间有依赖关系，比如当前任务依赖前一个任务的执行结果，这种问题基本上都可以用Future来解决。在分析这种问题的过程中，建议你用有向图描述一下任务之间的依赖关系，同时将线程的分工也做好，类似于烧水泡茶最优分工方案那幅图。对照图来写代码，好处是更形象，且不易出错。

课后思考

不久前听说小明要做一个询价应用，这个应用需要从三个电商询价，然后保存在自己的数据库里。核心示例代码如下所示，由于是串行的，所以性能很慢，你来试着优化一下吧。

```
// 向电商S1询价，并保存
r1 = getPriceByS1();
save(r1);
// 向电商S2询价，并保存
r2 = getPriceByS2();
save(r2);
// 向电商S3询价，并保存
r3 = getPriceByS3();
save(r3);
```

欢迎在留言区与我分享你的想法，也欢迎你在留言区记录你的思考过程。感谢阅读，如果你觉得这篇文章对你有帮助的话，也欢迎把它分享给更多的朋友。

精选留言

vector 9

最近使用CompletableFuture工具方法以及lamda表达式比较多，语言语法的变化带来编码效率的提升真的很大。

2019-04-21

aroll 6

建议并发编程课程中的Demo代码，尽量少使用System.out.println, 因为其实现有使用隐式锁，一些情况还会有锁粗化产生

2019-04-20

作者回复

好建议

2019-04-20

Asanz 4

不是不建议使用 Executors 创建线程池了吗？？？

2019-04-21

linqw 3

课后习题，老师帮忙看下哦

```
public class ExecutorExample {
private static final ExecutorService executor;
```

```
static {executor = new ThreadPoolExecutor(4, 8, 1, TimeUnit.SECONDS, new ArrayBlockingQueue<Runnable>(1000), runnable -> null, (r, executor) -> {//根据业务降级策略});
}
static class S1Task implements Callable<String> {
@Override
public String call() throws Exception {return getPriceByS1();}}
static class S2Task implements Callable<String> {
@Overridepublic String call() throws Exception {return getPriceByS2();}}
static class S3Task implements Callable<String> {@Override public String call() throws Exception {return getPriceByS3();}}
static class SaveTask implements Callable<Boolean> {private List<FutureTask<String>> futureTasks; public SaveTask(List<FutureTask<String>> futureTasks) {this.futureTasks = futureTasks;
}
@Override
public Boolean call() throws Exception {
for (FutureTask<String> futureTask : futureTasks) {
String data = futureTask.get(10, TimeUnit.SECONDS);
saveData(data);
}
return Boolean.TRUE;
}
}
private static String getPriceByS1() {
return "fromDb1";
}
private static String getPriceByS2() {
return "fromDb2";
}
private static String getPriceByS3() {
return "fromDb3";
}
private static void saveData(String data) {
//save data to db
}
public static void main(String[] args) {
S1Task s1Task = new S1Task();FutureTask<String> st1 = new FutureTask<>(s1Task);S2Task s2Task = new S2Task();FutureTask<String> st2 = new FutureTask<>(s2Task);S3Task s3Task = new S3Task();FutureTask<String> st3 = new FutureTask<>(s3Task);List<FutureTask<String>> futureTasks = Lists.<FutureTask<String>>newArrayList(st1, st2, st3);FutureTask<Boolean> saveTask = new FutureTask<>(new SaveTask(futureTasks));executor.submit(st1);executor.submit(st2);executor.submit(st3);executor.submit(saveTask);}}
```

作者回复

没问题，就是有点复杂，代码还可以精简一下

2019-04-23

圆圜

2

你这个不对啊，应该是executeservice.submit t2furturetask，不能直接提交t2

2019-04-22

undifined

2

课后题：

可以用 Future

```
ExecutorService threadPoolExecutor = Executors.newFixedThreadPool(3);
Future<R> future1 = threadPoolExecutor.submit(Test::getPriceByS1);
Future<R> future2 = threadPoolExecutor.submit(Test::getPriceByS2);
Future<R> future3 = threadPoolExecutor.submit(Test::getPriceByS3);
R r1 = future1.get();
R r2 = future2.get();
R r3 = future3.get();
```

也可以用 CompletableFuture

```
CompletableFuture<R> completableFuture1 = CompletableFuture.supplyAsync(Test::getPriceByS1);
CompletableFuture<R> completableFuture2 = CompletableFuture.supplyAsync(Test::getPriceByS2);
CompletableFuture<R> completableFuture3 = CompletableFuture.supplyAsync(Test::getPriceByS3);
CompletableFuture.allOf(completableFuture1, completableFuture2, completableFuture3)
.thenAccept(System.out::println);
```

老师这样理解对吗 谢谢老师

2019-04-20

liu

1

future是阻塞的等待。发起任务后，做其他的工作。做完后，从future获取处理结果，继续进行后面的任务

2019-04-25

捞鱼的搬砖奇

1

Future的get()是拿到任务的执行结果不吧。为什么又说是拿到方法的入参了。

2019-04-21

QQ怪

1

老师，在提交 Runnable 任务及结果引用的例子里面的x变量是什么？

2019-04-20

作者回复

任意的东西，想成数字0也行

2019-04-20

QQ怪 1

在实际项目中应用已经应用到了Feture,但没有使用线程池，没有那么优雅，所以算是get到了

2019-04-20

张三 1

打卡。感觉很神奇，之前完全不会用。学的知识太陈旧了，继续学习。

2019-04-20

henry 1

现在是在主线程串行完成3个询价的任务，执行第一个任务，其它2个任务只能等待执行，如果要提高效率，这个地方需要改进，可以用老师今天讲的futuretask，三个询价任务改成futuretask并行执行，效率会提高

2019-04-20

作者回复

2019-04-20

张天屹 1

我不知道是不是理解错老师意思了，先分析依赖有向图，可以看到三条线，没有入度>1的节点那么启动三个线程即可。

图：

s1询价 -> s1保存

s2询价 -> s2保存

s3询价 -> s3保存

代码：

```
new Thread(() -> {
r1 = getPriceByS1();
save(r1);
}).start();
new Thread(() -> {
r2 = getPriceByS2();
save(r2);
}).start();
new Thread(() -> {
r3 = getPriceByS3();
save(r3);
}).start();
```

我觉得这里不需要future,除非询价和保存之间还有别的计算工作

2019-04-20

作者回复

用线程池就用到了

2019-04-20

潭州太守

0

老师CompletableFuture.orTimeout是不是不会阻塞主线程

2019-06-04

路阳

0

```
public class ExecutorExample {

private static final ExecutorService executor = Executors.newFixedThreadPool(4);

public static void main(String[] args) {
FutureTask<String> st1 = new FutureTask<>(ExecutorExample::getPriceByS1);
FutureTask<String> st2 = new FutureTask<>(ExecutorExample::getPriceByS2);
FutureTask<String> st3 = new FutureTask<>(ExecutorExample::getPriceByS3);
Runnable saveTask = () -> {
List<FutureTask<String>> list = new ArrayList<FutureTask<String>>() {{
add(st1);
add(st2);
add(st3);
}};
while (!list.isEmpty()) {
Iterator<FutureTask<String>> it = list.iterator();
while (it.hasNext()) {
FutureTask<String> ftask = it.next();
if (ftask.isDone()) {
try {
saveData(ftask.get());
it.remove();
} catch (InterruptedException e) {
e.printStackTrace();
} catch (ExecutionException e) {
e.printStackTrace();
}
}
}
}
};

executor.submit(st1);
executor.submit(st2);
executor.submit(st3);
```

```
executor.submit(saveTask);

}

private static String getPriceByS1() {
return "fromDb1";
}

private static String getPriceByS2() {
return "fromDb2";
}

private static String getPriceByS3() {
return "fromDb3";
}

private static void saveData(String data) {
//save data to db
System.out.println("save data " + data);
}
}
```

2019-05-28

三木禾 0

这个可以用生产消费者模式啊

2019-05-18

 0

分别提交三个futuretask给线程池，然后最后分别get出结果，统一进行保存数据库

2019-04-30

Sunqc 0

老师，你所说的订蛋糕，我这样理解对吗，把任务提交给线程池就是让蛋糕店做蛋糕；去看电影就是主线程做其他事，提货单是对应调用future的get

2019-04-30

作者回复

理解的对

2019-04-30

右耳听海 0

回答思考，这三个任务如果没有结果依赖，直接用线程池提交三个任务应该就可以并行了吧

2019-04-26

Cancer 0

```
static class S1QueryTask implements Callable<Double>{
public Double call() throws Exception {
Thread.sleep(1000);//模拟查询时间
return Double.valueOf(10f);
}
}
static class S2QueryTask implements Callable<Double>{
public Double call() throws Exception {
Thread.sleep(2000);//模拟查询时间
return Double.valueOf(20f);
}
}
static class S3QueryTask implements Callable<Double>{
public Double call() throws Exception {
Thread.sleep(3000);//模拟查询时间
return Double.valueOf(30f);
}
}
static class SaveTask implements Callable<Double>{
final FutureTask<Double>[] queryfts;
private SaveTask(FutureTask<Double>[] queryfts) {
this.queryfts = queryfts;
}
private Double save(Double combRst) throws InterruptedException{
Thread.sleep(500);//模拟保存时间
return combRst;
}
public Double call() throws Exception {
Double combRst = new Double(0f);
for(FutureTask<Double> queryft : queryfts) {
Double rst = queryft.get();
if(rst != null) {
combRst += rst;
}
}
return save(combRst);
}
}
public static void main(String[] args) {
FutureTask<Double>[] queryfts = new FutureTask[] {new FutureTask<Double>(new S1QueryT
ask()),new FutureTask<Double>(new S2QueryTask()),new FutureTask<Double>(new S3Quer
yTask())};
FutureTask<Double> saveft3 = new FutureTask<Double>(new SaveTask(queryfts));
```

```
ExecutorService executor = Executors.newFixedThreadPool(4);
long start = System.currentTimeMillis();
for(FutureTask<Double> queryft : queryfts) {
executor.submit(queryft);
}
executor.submit(saveft3);
try {
Double combRst = saveft3.get();
long end = System.currentTimeMillis();
if(combRst != null) {
System.out.println("保存成功,合并结果：" + combRst);
}else {
System.out.println("保存失败");
}
System.out.println("耗时：" + (end - start) + "ms");
} catch (InterruptedException e) {
//按需处理
} catch (ExecutionException e) {
//按需处理
} catch (TimeoutException e) {
//按需处理
}finally {
executor.shutdown();
}
}
```

2019-04-24