25-CompletionService：如何批量执行异步任务？

在《23 | Future：如何用多线程实现最优的“烧水泡茶”程序？》的最后，我给你留了道思考题，如何优化一个询价应用的核心代码？如果采用“ThreadPoolExecutor+Future”的方案，你的优化结果很可能是下面示例代码这样：用三个线程异步执行询价，通过三次调用Future的get()方法获取询价结果，之后将询价结果保存在数据库中。

```
// 创建线程池
ExecutorService executor =
  Executors.newFixedThreadPool(3);
// 异步向电商S1询价
Future<Integer> f1 =
  executor.submit(
    ()->getPriceByS1());
// 异步向电商S2询价
Future<Integer> f2 =
  executor.submit(
    ()->getPriceByS2());
// 异步向电商S3询价
Future<Integer> f3 =
  executor.submit(
    ()->getPriceByS3());

// 获取电商S1报价并异步保存
executor.execute(
  ()->save(f1.get()));

// 获取电商S2报价并异步保存
executor.execute(
  ()->save(f2.get())

// 获取电商S3报价并异步保存
executor.execute(
  ()->save(f3.get())
```

上面的这个方案本身没有太大问题，但是有个地方的处理需要你注意，那就是如果获取电商S1报价的耗时很长，那么即便获取电商S2报价的耗时很短，也无法让保存S2报价的操作先执行，因为这个主线程都阻塞在了 `f1.get()` 操作上。这点小瑕疵你该如何解决呢？

估计你已经想到了，增加一个阻塞队列，获取到S1、S2、S3的报价都进入阻塞队列，然后在主线程中消费阻塞队列，这样就能保证先获取到的报价先保存到数据库了。下面的示例代码展示了如何利用阻塞队列实现先获取到的报价先保存到数据库。

```
// 创建阻塞队列
BlockingQueue<Integer> bq =
  new LinkedBlockingQueue<>();
//电商S1报价异步进入阻塞队列
executor.execute(()->
  bq.put(f1.get()));
//电商S2报价异步进入阻塞队列
executor.execute(()->
  bq.put(f2.get()));
//电商S3报价异步进入阻塞队列
executor.execute(()->
```

```
  bq.put(f3.get()));
//异步保存所有报价
for (int i=0; i<3; i++) {
  Integer r = bq.take();
  executor.execute(()->save(r));
}
```

利用CompletionService实现询价系统

不过在实际项目中，并不建议你这样做，因为Java SDK并发包里已经提供了设计精良的CompletionService。利用CompletionService不但能帮你解决先获取到的报价先保存到数据库的问题，而且还能让代码更简练。

CompletionService的实现原理也是内部维护了一个阻塞队列，当任务执行结束就把任务的执行结果加入到阻塞队列中，不同的是CompletionService是把任务执行结果的Future对象加入到阻塞队列中，而上面的示例代码是把任务最终的执行结果放入了阻塞队列中。

那到底该如何创建CompletionService呢？

CompletionService接口的实现类是ExecutorCompletionService，这个实现类的构造方法有两个，分别是：

1. `ExecutorCompletionService(Executor executor);`
2. `ExecutorCompletionService(Executor executor, BlockingQueue<Future<V>> completionQueue)`。

这两个构造方法都需要传入一个线程池，如果不指定completionQueue，那么默认会使用无界的LinkedBlockingQueue。任务执行结果的Future对象就是加入到completionQueue中。

下面的示例代码完整地展示了如何利用CompletionService来实现高性能的询价系统。其中，我们没有指定completionQueue，因此默认使用无界的LinkedBlockingQueue。之后通过CompletionService接口提供的submit()方法提交了三个询价操作，这三个询价操作将会被CompletionService异步执行。最后，我们通过CompletionService接口提供的take()方法获取一个Future对象（前面我们提到过，加入到阻塞队列中的是任务执行结果的Future对象），调用Future对象的get()方法就能返回询价操作的执行结果了。

```
// 创建线程池
ExecutorService executor =
  Executors.newFixedThreadPool(3);
// 创建CompletionService
CompletionService<Integer> cs = new
  ExecutorCompletionService<>(executor);
// 异步向电商S1询价
cs.submit(()->getPriceByS1());
// 异步向电商S2询价
cs.submit(()->getPriceByS2());
// 异步向电商S3询价
cs.submit(()->getPriceByS3());
// 将询价结果异步保存到数据库
for (int i=0; i<3; i++) {
  Integer r = cs.take().get();
  executor.execute(()->save(r));
}
```

CompletionService接口说明

下面我们详细地介绍一下CompletionService接口提供的方法，CompletionService接口提供的方法有5个，这5个方法的方法签名如下所示。

其中，submit()相关的方法有两个。一个方法参数是`Callable<V> task`，前面利用CompletionService实现询价系统的示例代码中，我们提交任务就是用的它。另外一个方法有两个参数，分别是`Runnable task`和`V result`，这个方法类似于ThreadPoolExecutor的`<T> Future<T> submit(Runnable task, T result)`，这个方法在《23 | Future：如何用多线程实现最优的“烧水泡茶”程序？》中我们已详细介绍过，这里不再赘述。

CompletionService接口其余的3个方法，都是和阻塞队列相关的，take()、poll()都是从阻塞队列中获取并移除一个元素；它们的区别在于如果阻塞队列是空的，那么调用 take() 方法的线程会被阻塞，而 poll() 方法会返回 null 值。 `poll(long timeout, TimeUnit unit)` 方法支持以超时的方式获取并移除阻塞队列头部的一个元素，如果等待了 timeout unit时间，阻塞队列还是空的，那么该方法会返回 null 值。

```
Future<V> submit(Callable<V> task);
Future<V> submit(Runnable task, V result);
Future<V> take()
  throws InterruptedException;
Future<V> poll();
Future<V> poll(long timeout, TimeUnit unit)
  throws InterruptedException;
```

利用CompletionService实现Dubbo中的Forking Cluster

Dubbo中有一种叫做**Forking的集群模式**，这种集群模式下，支持**并行地调用多个查询服务，只要有一个成功返回结果，整个服务就可以返回了**。例如你需要提供一个地址转坐标的服务，为了保证该服务的高可用和性能，你可以并行地调用3个地图服务商的API，然后只要有1个正确返回了结果r，那么地址转坐标这个服务就可以直接返回r了。这种集群模式可以容忍2个地图服务商服务异常，但缺点是消耗的资源偏多。

```
geocoder(addr) {
  //并行执行以下3个查询服务，
  r1=geocoderByS1(addr);
  r2=geocoderByS2(addr);
  r3=geocoderByS3(addr);
  //只要r1,r2,r3有一个返回
  //则返回
  return r1|r2|r3;
}
```

利用CompletionService可以快速实现 Forking 这种集群模式，比如下面的示例代码就展示了具体是如何实现的。首先我们创建了一个线程池executor 、一个CompletionService对象cs和一个`Future<Integer>`类型的列表 futures，每次通过调用CompletionService的submit()方法提交一个异步任务，会返回一个Future

对象，我们把这些Future对象保存在列表futures中。通过调用 cs.take().get()，我们能够拿到最快返回的任务执行结果，只要我们拿到一个正确返回的结果，就可以取消所有任务并且返回最终结果了。

```
// 创建线程池
ExecutorService executor =
  Executors.newFixedThreadPool(3);
// 创建CompletionService
CompletionService<Integer> cs =
  new ExecutorCompletionService<>(executor);
// 用于保存Future对象
List<Future<Integer>> futures =
  new ArrayList<>(3);
//提交异步任务，并保存future到futures
futures.add(
  cs.submit(()->geocoderByS1()));
futures.add(
  cs.submit(()->geocoderByS2()));
futures.add(
  cs.submit(()->geocoderByS3()));
// 获取最快返回的任务执行结果
Integer r = 0;
try {
  // 只要有一个成功返回，则break
  for (int i = 0; i < 3; ++i) {
    r = cs.take().get();
    //简单地通过判空来检查是否成功返回
    if (r != null) {
      break;
    }
  }
} finally {
  //取消所有任务
  for(Future<Integer> f : futures)
    f.cancel(true);
}
// 返回结果
return r;
```

总结

当需要批量提交异步任务的时候建议你使用CompletionService。CompletionService将线程池Executor和阻塞队列BlockingQueue的功能融合在了一起，能够让批量异步任务的管理更简单。除此之外，CompletionService能够让异步任务的执行结果有序化，先执行完的先进入阻塞队列，利用这个特性，你可以轻松实现后续处理的有序性，避免无谓的等待，同时还可以快速实现诸如Forking Cluster这样的需求。

CompletionService的实现类ExecutorCompletionService，需要你自己创建线程池，虽看上去有些啰嗦，但好处是你可以让多个ExecutorCompletionService的线程池隔离，这种隔离性能避免几个特别耗时的任务拖垮整个应用的风险。

课后思考

本章使用CompletionService实现了一个询价应用的核心功能，后来又有了新的需求，需要计算出最低报价并返回，下面的示例代码尝试实现这个需求，你看看是否存在问题呢？

```
// 创建线程池
ExecutorService executor =
  Executors.newFixedThreadPool(3);
// 创建CompletionService
CompletionService<Integer> cs = new
  ExecutorCompletionService<>(executor);
// 异步向电商S1询价
cs.submit(()->getPriceByS1());
// 异步向电商S2询价
cs.submit(()->getPriceByS2());
// 异步向电商S3询价
cs.submit(()->getPriceByS3());
// 将询价结果异步保存到数据库
// 并计算最低报价
AtomicReference<Integer> m =
  new AtomicReference<>(Integer.MAX_VALUE);
for (int i=0; i<3; i++) {
  executor.execute(()->{
    Integer r = null;
    try {
      r = cs.take().get();
    } catch (Exception e) {}
    save(r);
    m.set(Integer.min(m.get(), r));
  });
}
return m;
```

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：

- Corner 2019-04-25 09:12:31
  1.AtomicReference<Integer>的get方法应该改成使用cas方法

2.最后筛选最小结果的任务是异步执行的，应该在return之前做同步，所以最好使用sumit提交该任务便于判断任务的完成
最后请教老师一下，第一个例子中为什么主线程会阻塞在f1.get()方法呢？ [1赞]

- 天涯煮酒 2019-04-25 09:05:02
先调用m.get()并跟r比较，再调用m.set()，这里存在竞态条件，线程并不安全 [1赞]

- 张天屹 2019-04-25 11:33:20
老师我对第一个例子还是有疑问，f.get()已经提交给了线程池执行了，为什么会说阻塞主线程呢？

- 空知 2019-04-25 11:23:25
老师,感觉开篇的阻塞队列解决future.get阻塞 存在问题,阻塞队列也是把执行get结果加到队列,然后take出来,如果线程池不够大, f1的submit 和 get占满了线程,其他线程的执行都需要等待...还是会阻塞
如果线程池足够大,原始方案就可以直接申请新的线程执行

- 朱晋君 2019-04-25 10:53:57
1.m.set(Integer.min(m.get(), r))不是原子操作
2.catch住exception后，是否需要给r一个默认值呢
3.return 等不到异步结果

另外我有1个问题
我觉得第一个例子后面3个线程异步保存，不应该阻塞在f1.get，get方法会阻塞，但是只阻塞当前线程啊

- 西行寺咕哒子 2019-04-25 10:24:16
试过返回值是2147483647，也就是int的最大值。没有等待操作完成就猴急的返回了。 m.set(Integer.min(m.get(), r)... 这个操作也不是原子操作。
试着自己弄了一下：
```
public Integer run(){
// 创建线程池
ExecutorService executor = Executors.newFixedThreadPool(3);
// 创建 CompletionService
CompletionService<Integer> cs = new ExecutorCompletionService<>(executor);
AtomicReference<Integer> m = new AtomicReference<>(Integer.MAX_VALUE);
// 异步向电商 S1 询价
cs.submit(()->getPriceByS1());
// 异步向电商 S2 询价
cs.submit(()->getPriceByS2());
// 异步向电商 S3 询价
cs.submit(()->getPriceByS3());
// 将询价结果异步保存到数据库
// 并计算最低报价
for (int i=0; i<3; i++) {
Integer r = logIfError(()->cs.take().get());
executor.execute(()-> save(r));
m.getAndUpdate(v->Integer.min(v, r));
}
return m.get();
}
```
不知道可不可行

- 美美 2019-04-25 10:21:58
  第一个例子 我感觉f1.get()没有阻塞主线程啊 f1.get()是在线程池里异步执行的啊

- 黄海峰 2019-04-25 10:08:08
  我实际测试了第一段代码，确实是异步的，f1.get不会阻塞主线程。。。

```
public static void main(String[] args) {
ExecutorService executor = Executors.newFixedThreadPool(3);
Future<Integer> f1 = executor.submit(()->getPriceByS1());
Future<Integer> f2 = executor.submit(()->getPriceByS2());
Future<Integer> f3 = executor.submit(()->getPriceByS3());

executor.execute(()-> {
try {
save(f1.get());
} catch (InterruptedException e) {
e.printStackTrace();
} catch (ExecutionException e) {
e.printStackTrace();
}
});
executor.execute(()-> {
try {
save(f2.get());
} catch (InterruptedException e) {
e.printStackTrace();
} catch (ExecutionException e) {
e.printStackTrace();
}
});
executor.execute(()-> {
try {
save(f3.get());
} catch (InterruptedException e) {
e.printStackTrace();
} catch (ExecutionException e) {
e.printStackTrace();
}
});
}

private static Integer getPriceByS1() {
try {
Thread.sleep(10000);
} catch (InterruptedException e) {
e.printStackTrace();
}
return 1;
}
```

```
private static Integer getPriceByS2() {
try {
Thread.sleep(1000);
} catch (InterruptedException e) {
e.printStackTrace();
}
return 2;
}
private static Integer getPriceByS3() {
try {
Thread.sleep(1000);
} catch (InterruptedException e) {
e.printStackTrace();
}
return 3;
}
private static void save(Integer i) {
System.out.println("save " + i);
}
```

- 美美 2019-04-25 10:06:18
  第一个例子 f1.get()是线程池里异步执行的 为啥会阻塞主线程

- 刘章周 2019-04-25 09:58:15
  m.get()和m.set()不是原子性操作，正确代码是:do{int expect = m.get();int min= Integer.min(expect,r);}while(!m.compareAndSet(expect,min))。老师，是这样吗？

- Zach_ 2019-04-25 09:54:24
  1.思考题: 这是个先判断后执行的复合操作，多线程下依赖上一个线程写入的结果。这里的m.set()方法应该不是原子操作，应该保证set的原子性吧

  2.问题: 老师，这里的cs.take().get()方法我有个疑问:

  a.当执行某个询价任务的过程中抛出了异常，这个异常是不是在Future<T>里面啊？
  是通过cs.take().get()里的get()方法取到异常了吗？
  还是直接在阻塞get()的过程中抛出来了啊？

  b.如果是阻塞在get()这里，此时任务线程被interrupt()了，那我们是不是就获取不到询价结果而是在get()过程中捕获了一个异常啊?

  或者评论区的童鞋回答一下我也很感谢啊！

- 张三 2019-04-25 09:45:02
  打卡。

- 黄海峰 2019-04-25 09:37:40
  老师，有个地方不理解。。第一段代码中这个f1.get不是在线程池里执行的吗？为何会阻塞了主线程？

  //获取电商 S1 报价并异步保存
  executor.execute(

()->save(f1.get()));

- 渔夫 2019-04-25 09:28:29
m.set(Integer.min(m.get(), r));

这个逻辑执行应该互斥，但是 set/get 之间无法形成原子操作，应该使用 getAndUpdate 来完成该逻辑

- 张天屹 2019-04-25 09:27:51
我觉得问题出在return m这里需要等待三个线程执行完成，但是并没有。
...
AtomicReference<Integer> m = new AtomicReference<>(Integer.MAX_VALUE);
CountDownLatch latch = new CountDownLatch(3);
for(int i=0; i<3; i++) {
executor.execute(()->{
Integer r = null;
try {
r = cs.take().get();
} catch(Exception e) {}
save(r);
m.set(Integer.min(m.get(), r));
latch.countDown();
});
latch.await();
return m;
}

- 周治慧 2019-04-25 09:16:07
存在问题，在执行executor.execute的时候多个线程是非阻塞的异步执行，可能还没等到线程执行完的时候就直接返回结果了，大部分情况会出现是integer的最大值。改进的办法是在遍历时去取阻塞队列中的值后再执行set操作，因为在get取阻塞队列中的值的过程是一个阻塞，最后在利用线程池的非阻塞异步操作去保存结果。

- undifined 2019-04-25 09:05:02
老师 用 CompletionService 和用 CompletionFuture 查询，然后用 whenComplete 或者 thenAcceptEither 这些方法的区别是什么，我觉得用 CompletionFuture 更直观些；
老师可以在下一讲的时候说一下上一讲的思考题正确答案吗，谢谢老师

- 苏志辉 2019-04-25 08:57:03
m.set和get存在静态条件不是原子的，可能存在设置和不是最小值

- 空空空空 2019-04-25 08:55:50
算低价的时候是用三个不同的线程去计算，是异步的，因此可能算出来并不是预期的结果
老师，这样理解对吗？

- 郑晨Cc 2019-04-25 02:46:53
executor.execute（Callable）提交任务是非阻塞的 return m；很大概率返回 Integer.Maxvalue，而且老师为了确保返回这个max还特意加入了save这个阻塞的方法