

10 | 如何使用异步设计提升系统性能？

李玥 · 消息队列高手课



你好，我是李玥，这一讲我们来聊一聊异步。

对于开发者来说，异步是一种程序设计的思想，使用异步模式设计的程序可以显著减少线程等待，从而在高吞吐量的场景中，极大提升系统的整体性能，显著降低时延。

因此，像消息队列这种需要超高吞吐量和超低时延的中间件系统，在其核心流程中，一定会大量采用异步的设计思想。

接下来，我们一起来通过一个非常简单的例子学习一下，使用异步设计是如何提升系统性能的。

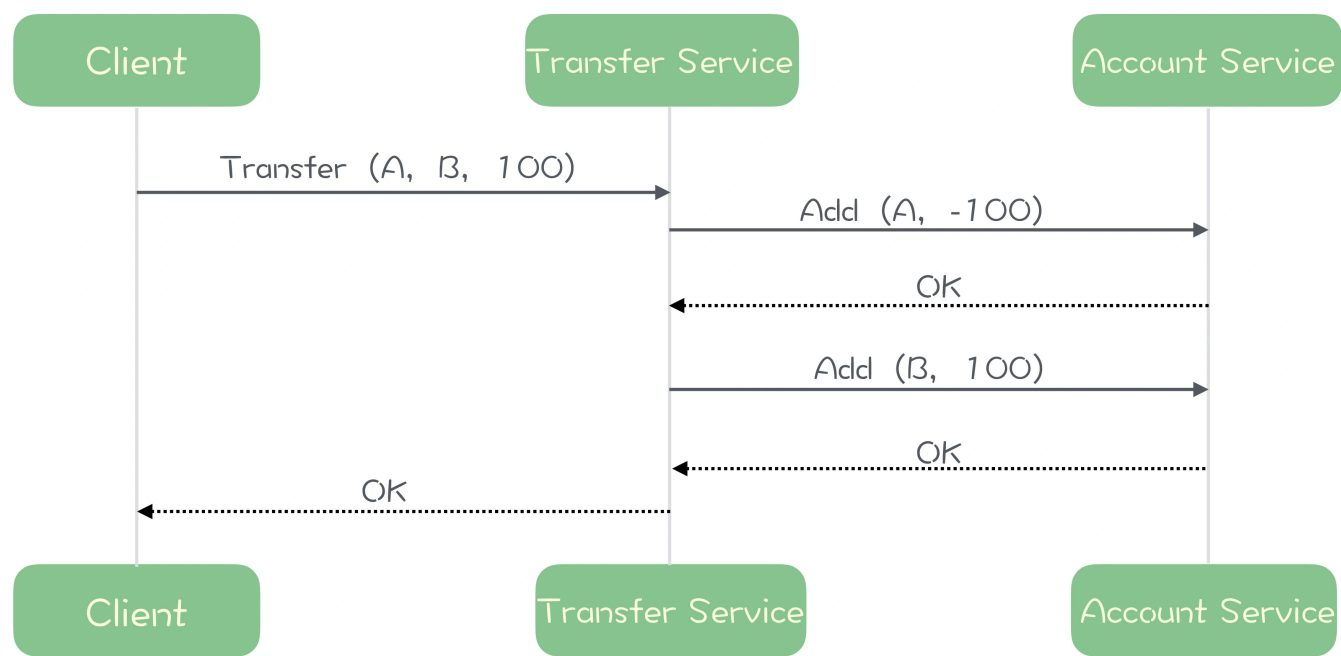
异步设计如何提升系统性能？

假设我们要实现一个转账的微服务 `Transfer(accountFrom, accountTo, amount)`，这个服务有三个参数：分别是转出账户、转入账户和转账金额。

实现过程也比较简单，我们要从账户 A 中转账 100 元到账户 B 中：

- 1. 先从 A 的账户中减去 100 元；
- 2. 再给 B 的账户加上 100 元，转账完成。

对应的时序图是这样的：



在这个例子的实现过程中，我们调用了另外一个微服务 `Add(account, amount)`，它的功能是给账户 `account` 增加金额 `amount`，当 `amount` 为负值的时候，就是扣减响应的金额。

需要特别说明的是，在这段代码中，我为了使问题简化以便我们能专注于异步和性能优化，省略了错误处理和事务相关的代码，你在实际的开发中不要这样做。

1. 同步实现的性能瓶颈

首先我们来看一下同步实现，对应的伪代码如下：

复制代码

```
1 Transfer(accountFrom, accountTo, amount) {
2   // 先从accountFrom的账户中减去相应的钱数
3   Add(accountFrom, -1 * amount)
```

```
4    // 再把减去的钱数加到accountTo的账户中
5    Add(accountTo, amount)
6    return OK
7 }
```

上面的伪代码首先从 accountFrom 的账户中减去相应的钱数，再把减去的钱数加到 accountTo 的账户中，这种同步实现是一种很自然方式，简单直接。那么性能表现如何呢？接下来我们就来一起分析一下性能。

假设微服务 Add 的平均响应时延是 50ms，那么很容易计算出我们实现的微服务 Transfer 的平均响应时延大约等于执行 2 次 Add 的时延，也就是 100ms。那随着调用 Transfer 服务的请求越来越多，会出现什么情况呢？

在这种实现中，每处理一个请求需要耗时 100ms，并在这 100ms 过程中是需要独占一个线程的，那么可以得出这样一个结论：每个线程每秒钟最多可以处理 10 个请求。我们知道，每台计算机上的线程资源并不是无限的，假设我们使用的服务器同时打开的线程数量上限是 10,000，可以计算出这台服务器每秒钟可以处理的请求上限是： $10,000 \text{ (个线程)} \times 10 \text{ (次请求每秒)} = 100,000 \text{ 次每秒}$ 。

如果请求速度超过这个值，那么请求就不能被马上处理，只能阻塞或者排队，这时候 Transfer 服务的响应时延由 100ms 延长到了：排队的等待时延 + 处理时延 (100ms)。也就是说，在大量请求的情况下，我们的微服务的平均响应时延变长了。

这是不是已经到了这台服务器所能承受的极限了呢？其实远远没有，如果我们监测一下服务器的各项指标，会发现无论是 CPU、内存，还是网卡流量或者是磁盘的 IO 都空闲的很，那我们 Transfer 服务中的那 10,000 个线程在干什么呢？对，绝大部分线程都在等待 Add 服务返回结果。

也就是说，采用同步实现的方式，整个服务器的所有线程大部分时间都没有在工作，而是都在等待。

如果我们能减少或者避免这种无意义的等待，就可以大幅提升服务的吞吐能力，从而提升服务的总体性能。

2. 采用异步实现解决等待问题

接下来我们看一下，如何用异步的思想来解决这个问题，实现同样的业务逻辑。

 复制代码

```
1 TransferAsync(accountFrom, accountTo, amount, OnComplete()) {
2     // 异步从accountFrom的账户中减去相应的钱数，然后调用OnDebit方法。
3     AddAsync(accountFrom, -1 * amount, OnDebit(accountTo, amount, OnAllDone(OnCompl
4 }
5 // 扣减账户accountFrom完成后调用
6 OnDebit(accountTo, amount, OnAllDone(OnComplete())) {
7     // 再异步把减去的钱数加到accountTo的账户中，然后执行OnAllDone方法
8     AddAsync(accountTo, amount, OnAllDone(OnComplete()))
9 }
10 // 转入账户accountTo完成后调用
11 OnAllDone(OnComplete()) {
12     OnComplete()
13 }
```

细心的你可能已经注意到了，TransferAsync 服务比 Transfer 多了一个参数，并且这个参数传入的是一个回调方法 OnComplete()（虽然 Java 语言并不支持将方法作为方法参数传递，但像 JavaScript 等很多语言都具有这样的特性，在 Java 语言中，也可以通过传入一个回调类的实例来变相实现类似的功能）。

这个 TransferAsync() 方法的语义是：请帮我执行转账操作，当转账完成后，请调用 OnComplete() 方法。调用 TransferAsync 的线程不必等待转账完成就可以立即返回了，待转账结束后，TransferService 自然会调用 OnComplete() 方法来执行转账后续的工作。

异步的实现过程相对于同步来说，稍微有些复杂。我们先定义 2 个回调方法：

OnDebit(): 扣减账户 accountFrom 完成后调用的回调方法；

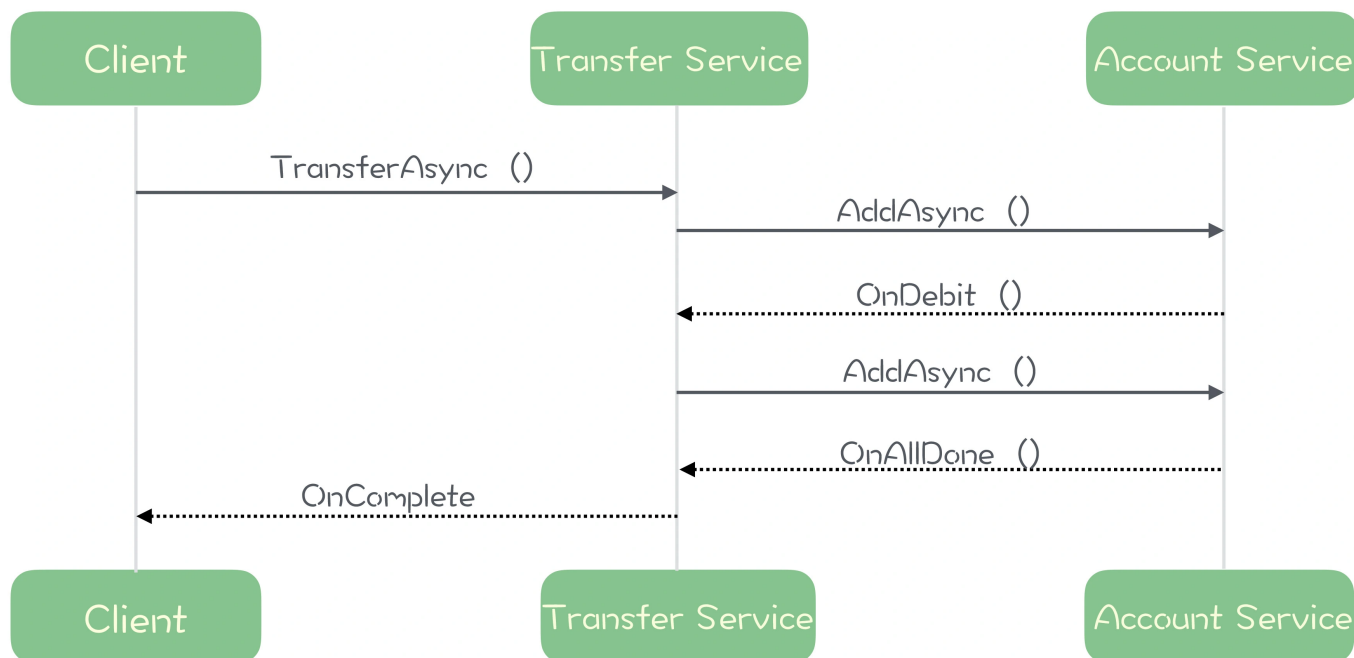
OnAllDone(): 转入账户 accountTo 完成后调用的回调方法。

整个异步实现的语义相当于：

1. 异步从 accountFrom 的账户中减去相应的钱数，然后调用 OnDebit 方法；

2. 在 OnDebit 方法中，异步把减去的钱数加到 accountTo 的账户中，然后执行 OnAllDone 方法；
3. 在 OnAllDone 方法中，调用 OnComplete 方法。

绘制成时序图是这样的：



你会发现，异步化实现后，整个流程的时序和同步实现是完全一样的，区别只是在线程模型上由同步顺序调用改为了异步调用和回调的机制。

接下来我们分析一下异步实现的性能，由于流程的时序和同步实现是一样，在低请求数量的场景下，平均响应时延一样是 100ms。在超高请求数量场景下，异步的实现不再需要线程等待执行结果，只需要个位数量的线程，即可实现同步场景大量线程一样的吞吐量。

由于没有了线程的数量的限制，总体吞吐量上限会大大超过同步实现，并且在服务器 CPU、网络带宽资源达到极限之前，响应时延不会随着请求数量增加而显著升高，几乎可以一直保持约 100ms 的平均响应时延。

看，这就是异步的魔力。


简单实用的异步框架: CompletableFuture

在实际开发时，我们可以使用异步框架和响应式框架，来解决一些通用的异步编程问题，简化开发。Java 中比较常用的异步框架有 Java8 内置的 [CompletableFuture](#) 和 ReactiveX 的 [RxJava](#)，我个人比较喜欢简单实用易于理解的 CompletableFuture，但是 RxJava 的功能更加强大。有兴趣的同学可以深入了解一下。


Java 8 中新增了一个非常强大的用于异步编程的类：CompletableFuture，几乎囊括了我们在开发异步程序的大部分功能，使用 CompletableFuture 很容易编写出优雅且易于维护的异步代码。

接下来，我们来看下，如何用 CompletableFuture 实现的转账服务。

首先，我们用 CompletableFuture 定义 2 个微服务的接口：

 复制代码

```
1  /**
2   * 账户服务
3   */
4  public interface AccountService {
5      /**
6       * 变更账户金额
7       * @param account 账户ID
8       * @param amount 增加的金额，负值为减少
9       */
10     CompletableFuture<Void> add(int account, int amount);
11 }
```


 复制代码

```
1  /**
2   * 转账服务
3   */
4  public interface TransferService {
5      /**
6       * 异步转账服务
7       * @param fromAccount 转出账户
8       * @param toAccount 转入账户
9       * @param amount 转账金额，单位分
10     */
```

```
11     CompletableFuture<Void> transfer(int fromAccount, int toAccount, int amount);
12 }
```

可以看到这两个接口中定义的方法的返回类型都是一个带泛型的 `CompletableFuture`，尖括号中的泛型类型就是真正方法需要返回数据的类型，我们这两个服务不需要返回数据，所以直接用 `Void` 类型就可以。

然后我们来实现转账服务：


 复制代码

```
1  /**
2   * 转账服务的实现
3   */
4  public class TransferServiceImpl implements TransferService {
5      @Inject
6      private AccountService accountService; // 使用依赖注入获取账户服务的实例
7      @Override
8      public CompletableFuture<Void> transfer(int fromAccount, int toAccount, int a
9          // 异步调用add方法从fromAccount扣减相应金额
10         return accountService.add(fromAccount, -1 * amount)
11         // 然后调用add方法给toAccount增加相应金额
12         .thenCompose(v -> accountService.add(toAccount, amount));
13     }
14 }
```

在转账服务的实现类 `TransferServiceImpl` 里面，先定义一个 `AccountService` 实例，这个实例从外部注入进来，至于怎么注入不是我们关心的问题，就假设这个实例是可用的就好了。

然后我们看实现 `transfer()` 方法的实现，我们先调用一次账户服务 `accountService.add()` 方法从 `fromAccount` 扣减响应的金额，因为 `add()` 方法返回的就是一个 `CompletableFuture` 对象，可以用 `CompletableFuture` 的 `thenCompose()` 方法将下一次调用 `accountService.add()` 串联起来，实现异步依次调用两次账户服务完整转账。

客户端使用 `CompletableFuture` 也非常灵活，既可以同步调用，也可以异步调用。

 复制代码

```

1 public class Client {
2     @Inject
3     private TransferService transferService; // 使用依赖注入获取转账服务的实例
4     private final static int A = 1000;
5     private final static int B = 1001;
6
7     public void syncInvoke() throws ExecutionException, InterruptedException {
8         // 同步调用
9         transferService.transfer(A, B, 100).get();
10        System.out.println("转账完成!");
11    }
12
13    public void asyncInvoke() {
14        // 异步调用
15        transferService.transfer(A, B, 100)
16            .thenRun(() -> System.out.println("转账完成!"));
17    }
18 }

```

在调用异步方法获得返回值 `CompletableFuture` 对象后，既可以调用 `CompletableFuture` 的 `get` 方法，像调用同步方法那样等待调用的方法执行结束并获得返回值，也可以像异步回调的方式一样，调用 `CompletableFuture` 那些以 `then` 开头的一系列方法，为 `CompletableFuture` 定义异步方法结束之后的后续操作。比如像上面这个例子中，我们调用 `thenRun()` 方法，参数就是将转账完成打印在控台上这个操作，这样就可以实现在转账完成后，在控制台打印“转账完成！”了。

小结

简单的说，异步思想就是，**当我们要执行一项比较耗时的操作时，不去等待操作结束，而是给这个操作一个命令：“当操作完成后，接下来去执行什么。”**

使用异步编程模型，虽然并不能加快程序本身的速度，但可以减少或者避免线程等待，只用很少的线程就可以达到超高的吞吐能力。

同时我们也需要注意到异步模型的问题：相比于同步实现，异步实现的复杂度要大很多，代码的可读性和可维护性都会显著的下降。虽然使用一些异步编程框架会在一定程度上简化异步开发，但是并不能解决异步模型高复杂度的问题。

异步性能虽好，但一定不要滥用，只有类似在像消息队列这种业务逻辑简单并且需要超高吞吐量的场景下，或者必须长时间等待资源的地方，才考虑使用异步模型。如果系统的业务逻辑比较复杂，在性能能够满足业务需求的情况下，采用符合人类自然的思路且易于开发和维护的同步模型是更加明智的选择。

思考题

课后给你留 2 个思考题：

第一个思考题是，我们实现转账服务时，并没有考虑处理失败的情况。你回去可以想一下，在异步实现中，如果调用账户服务失败时，如何将错误报告给客户端？在两次调用账户服务的 Add 方法时，如果某一次调用失败了，该如何处理才能保证账户数据是平的？

第二个思考题是，在异步实现中，回调方法 OnComplete() 是在什么线程中运行的？我们是否能控制回调方法的执行线程数？该如何做？欢迎在留言区写下你的想法。

感谢阅读，如果你觉得这篇文章对你有帮助的话，也欢迎把它分享给你的朋友。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

精选留言 (65)



小伟 置顶

2019-08-22

个人的一点想法：

异步回调机制的本质是通过减少线程等待时占用的CPU时间片，来提供CPU时间片的利用率。

具体做法是用少数线程响应业务请求，但处理时这些线程并不真正调用业务逻辑代码，而是简单的把业务处理逻辑扔到另一个专门执行业务逻辑代码的线程后就返回了，故不会有任何等待（CPU时间片浪费）。专门执行业务逻辑的线程可能会由于IO慢导致上下文切换而浪费一些CPU时间片，但这已经不影响业务请求的响应了，而业务逻辑执行完毕后会去把回调处理逻辑再扔到专门执行回调业务逻辑的线程中，这时的执行业务逻辑线程的使命已完成，线程返回，然后会去找下一个需要执行业务逻辑，这里没有任何等待。回调业务处理线程也是同理。

以上于《摩登时代》里的卓别林很像，每个人只做自己的那点事（卓别林只拧螺丝）。有的线程

只负责响应请求(放螺丝)，有的线程只负责执行业务逻辑(拧螺丝)，有的线程只负责执行回调代码(敲螺丝)，完成后就返回并继续执行下一个相同任务(拧完这个螺丝再找下一个需要拧的螺丝)，没有相互依赖的等待(放螺丝的不等螺丝拧好就直接放下一个螺丝)。

有利就有弊，分开后是不用等别人了，但想知道之前的步骤是否已经做好了就难了。比如螺丝没有拧紧就开始敲，会导致固定不住。如果发现螺丝没拧好，敲螺丝的人就要和工头说这块板要返工，螺丝取下，重新放，重新拧，之后才能敲。

个人感觉把关联性且无需长时间等待的操作(如大量磁盘或网络IO)打包成同步，其他用异步，这样可以在规避CPU时间片浪费的同时兼顾了一致性，降低了补偿的频率和开销。

作者回复: 人工置顶 📌

共 25 条评论 >

👍 134



笑傲流云

2019-08-13

个人的思路，欢迎老师点评下哈。

- 1, 调用账户失败，可以在异步callBack里执行通知客户端的逻辑；
- 2, 如果是第一次失败，那后面的那一步就不用执行了，所以转账失败；如果是第一次成功但是第二次失败，首先考虑重试，如果转账服务是幂等的,可以考虑一定次数的重试，如果不能重试，可以考虑采用补偿机制，undo第一次的转账操作。
- 3, CompletableFuture默认是在ForkjoinPool commonpool里执行的，也可以指定一个Executor线程池执行，借鉴guava的ListenableFuture的时间，回调可以指定线程池执行，这样就能控制这个线程池的线程数目了。

作者回复: 👍👍👍

共 11 条评论 >

👍 93



senekis

2019-08-13

老师，我一直有一个困惑，就是想不明白为何异步可以节省线程。每次发起一个异步调用不都会创建一个新的线程吗？我理解了好几次，感觉只是异步处理线程在等待时可以让出时间片给其他线程运行啊？一直想不明白这个问题，困扰了好久，求老师解惑。

作者回复: 太多的线程会造成频繁的cpu上下文切换，你可以想象一下，假设你的小公司只有8台电脑，你雇8个程序员一直不停的工作显然是效率最高的。考虑到程序员要休息不可能连轴转，雇佣24个人，每天三班倒，效率也还行。

但是，你要雇佣10000个人，他们还是只能用这8台电脑，大部分时间不都浪费在换人、交接工作上吗？

共 30 条评论 >

👍 68



付永强

2019-08-13

老师可能里面过多提到线程这两个字，所以很多人把异步设计理解成节约线程，其实李玥老师这里想说明的是异步是用来提高CPU的利用率，而不是节省线程。

异步编程是通过分工的方式，是为了减少了CPU因线程等待的可能，让CPU一直处于工作状态。换句话说，如果我们能想办法减少CPU空闲时间，我们的计算机就可以支持更多的线程。

其实线程是一个抽象概念，我们从物理层面理解，就是单位时间内把每毫核分配处理不同的任务，从而提高单位时间内CPU的利用率。

作者回复: 🍊🍊🍊

线程就是为了能自动分配CPU时间片而生的。

共 6 条评论 >

👍 60



我丢了一只小凳子

2020-02-19

老师，有一点不太懂，异步转账时，假如专门几个线程（Threads_quest）处理转账请求，其他的线程处理账户增减金额逻辑，虽然大量请求来的时候，Threads_quest 这几个线程可以接受请求之后扔给其他线程处理增减金额，但是由于请求量过大，不也会导致其他线程处理变慢吗，导致完整的处理也变慢

作者回复: 首先，你要理解，为什么请求多了程序会变慢这个事儿。

计算的资源，比如说CPU、磁盘IO，它的处理能力是恒定的，都不会因为请求量大而“变慢”。

比如说CPU执行一次加法，任何情况下耗时都是差不多的。

我们所看到的请求量大“系统变慢”的现象，一定是因为某一种资源忙，达到了瓶颈。比如说，一个单核CPU，每做一次加法需要0.1秒，那它每秒最多做10次加法。

一下子100个程序同时都来请求CPU做一次加法，这个CPU就需要10秒才能算完。对于这个CPU，它并没变慢，仍然是每秒做10次加法。

但是对于某一个请求CPU的程序来说，它看到的现象是，我让CPU做了一次加法，它八秒才做完，看起来就是“变慢”了。

所以，程序变慢一定是因为某一个资源忙，遇到了瓶颈。同步程序因为线程数量的限制，它的瓶颈往往是线程数量。并不能发挥服务器的全部处理能力。异步程序不需要那么多线程，所以可以发挥出服务器的全部处理能力，直到把CPU或者磁盘IO打满，所以要快得多。

共 5 条评论>

👍 53



mark

2019-09-20

go 语言的本质就是 用同步的方式，写出异步的代码。

每个用户过来直接启动一个 go routine ，可以同时启动上百万的协程
go processConn()

而go 的内部 只启动少量线程，io 的等待全部做了异步处理。



👍 27



沧浪之水

2020-04-28

一直有个困惑：异步可以解决io等待的问题，我感觉只是把等待的线程从业务线程转到异步框架线程了。最后的结果是，总有一个线程在io等待。不是业务线程就是异步线程。从整个机器来考虑，都是在消耗线程资源。 那怎么理解使用异步能够提升吞吐量呢，因为业务总是要有返回数据的。 数据在异步线程中等待，并不节约时间。吞吐量的卡点还是io时间。



👍 9



linqw

2019-08-13

尝试回答课后习题，老师有空帮忙看下哦

思考题一、如果在异步实现中，如果调用账户服务失败，可以以账单的形式将转账失败的记录下来，比如客户在转账一段时间后

查看账单就可以知道转账是否成功，只要保证转账失败，客户的钱没有少就可以。两次调用账户服务，感觉可以这样写

/**

* 转账服务的实现

*/

```

public class TransferServiceImpl implements TransferService {
    @Inject
    private AccountService accountService; // 使用依赖注入获取账户服务的实例
    @Override
    public CompletableFuture<Void> transfer(int fromAccount, int toAccount, int amount) {
        // 异步调用 add 方法从 fromAccount 扣减相应金额
        return accountService.add(fromAccount, -1 * amount).exceptionally(add(fromAccount, amount))
            .thenCompose(v -> accountService.add(toAccount, amount)).exceptionally(add(toAccount, -1 * amount));
    }
}

```

思考题二、在异步实现中，回调方法OnComplete()可以在另一个线程池执行，比如rocketmq中异步实现，

再异步发送消息时，会将封装一个ResponseFuture包含回调方法、通道、请求消息id，将其请求id做为key，ResponseFuture做为value放入map中

等响应返回时，根据请求id从map中获取ResponseFuture，然后将ResponseFuture中的回调方法封装成任务放入到线程池中执行。然后执行

特定的回调方法。CompletableFuture有点需要注意的是，在不同的业务中需创建各自的线程池，否则都是共用ForkJoinPool。

写下对异步的理解，如果同步一个请求线程需要等待处理结果完，才可以处理其他请求，这样的话会导致如果请求多，创建很多线程

但是这些线程大部分都是等待处理结果，如果有后续的请求，没有其他线程及时处理会导致延迟（等待线程时间+处理时间），

会出现机器的cpu、磁盘、内存都不高（因为等待的线程是不占CPU的）很多请求超时之类的情况。异步的话，就是让线程调用处理接口就直接返回

不用等待处理结果，后续的处理结果可以用回调的形式来处理。如果对那些不需要实时拿到结果的业务就很适合，可以提高整个系统的吞吐率

作者回复: 总结的非常好!

有一点需要改进一下，转账服务的实现中，异常处理的部分，还是需要先检查再补偿，否则有可能出现重复补偿的情况。



Better me

2019-08-13

对于思考题:

1、应该可以通过程式事物来保证数据的完整性。如何将错误结果返回给客户端，感觉这边和老师上次答疑网关如何接收服务端秒杀结果有点类似，通过方法回调，在回调方法中保存下转账成功或失败

2、在异步实现中，回调方法 `OnComplete()` 在执行 `OnAllDone()` 回调方法的那个线程，可以通过一个异步线程池去控制回调方法的线程数，如Spring中的`async`就是通过结合线程池来实现异步调用

看了两遍才稍微有点思路，老师有空看看

作者回复: 🍊🍊🍊



👍 8



timmy21

2019-08-21

我有一个困惑，客户端发起一个请求转账服务，此时转账服务会启动一个线程A处理该请求，然后转账再使用线程池异步调用账户服务。但是线程A还是存在，并等待处理结果的。我的问题来了，如果有10万个转账请求，转账服务还是最少开启10万个线程A的吧？

作者回复: 线程A在异步调用完账户服务后就可以结束了，不需要等待，响应可以由其他的线程负责返回给调用者，由于这个过程中不涉及任何具体的业务逻辑，是非常快的，可以认为瞬间就结束了。所以并不需要很多的线程。



👍 5



王建坤

2020-07-07

老师你好，`CompletableFuture`这种回调底层还是forkjoin框架，forkjoin对于io这种操作还是会阻塞线程，而且`CompletableFuture`默认线程数是与cpu核数一样的。在现在容器化的场景下，Cpu核数都不会很多（一般都是个位数），那么使用`CompletableFuture`是执行io操作是不是会更早的无响应？因为个位数的线程很快就都被阻塞了。

作者回复: 这里说一下我的理解:

`CompletableFuture`还不能等完全同于`ForkJoin`。

可以简单的理解为

CompletableFuture.then() 等于 Fork

CompletableFuture.get() 等于 Join

但不是所有场景下，CompletableFuture都需要用get()结束的。也就是说，有时候是不需要调用阻塞的get()方法的。

另外，虽然CompletableFuture 默认使用 ForkJoinPool，但你完全可以给它提供一个自定义的执行器。



👍 4



xfan

2019-10-31

这一篇写的有点歧义，我觉得举的例子主要是网络IO方面的情况，其实发送请求本身就是串行的，不论是回调还是阻塞发送，其实都是串行的，不同的是：同步等待这种情况是利用了当前线程栈，而异步发送时创建了新线程发送，而抛弃了原有线程。从CPU时间片利用效率来说，这两者没有区别，而且异步可能会更消耗CPU时间片，因为创建线程需要。再者网络IO时间也不会减少，吞吐量也不会增加，还是走了两趟。如果真的需要减少时间增加吞吐，那需要计算好结果发送给ADD，比如 $-100 + 80 = -20$ 最后发送直接发 -20 即可。



👍 3



长期规划

2019-09-23

同步会有等待，导致线程没活干，CPU利用率低，但创建的大量线程占着内存等资源。如果换成异步，将一个任务切成多个片段，切点是IO阻塞的地方。维护一个线程池，当执行每一个片段时，从线程池取线程。这样，同样多任务，用更少线程数就可以，线程少了，线程等待时间少，利用率提高了。

共 1 条评论 >

👍 3



小祺

2019-09-06

老师，你用CompletableFuture这种方式跟我自己用一个固定大小的线程池去submit, 然后返回Future再get有什么本质的区别吗？吞吐量上呢？

作者回复：性能上没有区别。

区别是代码结构更清晰简单，易于维护。



业余草

2019-08-13

异步虽好，但使用场景有限！



tongZi

2020-07-16

看到评论区大家的评论，发表一下自己的一些小观点，望指正

1、client调用transfer service，transfer然后异步调用account service；

transfer和client保持连接的线程大大减少了。

由于使用异步调用，达到线程上线时，减少了线程的创建等待，但是真正占用cpu时间片，的线程可以更多了；

像文中老师说的“服务器 CPU、网络带宽资源达到极限之前”，可以处理更多的任务，提高了cpu使用率



团子

2020-04-21

老师，即使转账和账户服务都是异步，但是只是在我们代码端的异步，如果add方法设计到io传输，那completeablefuture中线程还是要block的把

作者回复: 结合Netty或者NIO，是可以做到全异步的，不需要线程在阻塞等待响应。



小祺

2019-09-06

老师，我理解异步是可以解决请求超时的问题，但是像文中举例这种转账操作，转出转入两个操作是前后依赖的没法并行，那么这种前后依赖的任务使用异步跟同步又有什么区别呢？

另外，当10万请求过来之后，虽然用了异步可以瞬间返回，但是其实几万个请求对象在CompletableFuture内部线程池内部还是排队啊，所以最后来的请求还是要等很久才能被执行到。那么既然同步or异步都需要排队，异步究竟快在哪里了呢？

作者回复: 第一个问题，转入转出这两个操作不需要串行，是可以并行的。甚至执行顺序都没什么要求。我们唯一要保证的是这两个操作在一个事务中执行，“要么都成功，要么都失败”，就可以了。

你这个场景是在调用方（转账服务）异步，而服务提供方（账户服务）还是同步服务的情况下，才会出现。

你仔细看一下我们的异步设计，服务提供方提供的也是异步服务，那调用账户服务也是一瞬间就完成了，这样就不会出现你说的“几万个请求对象在CompletableFuture内部线程池内部还是排队”的情况了。



👍 2



谢清

2019-08-13

学习了，一点思路，欢迎老师点评

第一个问题：

两次add方法保持最终一致性，第一次add失败不在调用第二次，告知客户转账失败；第一次成功调第二次失败，告知用户：转账进行中，转账对象收款中；可设置补偿策略，还是失败的话，转账后台人工介入补偿，还是不行则人工还原账户金额并告知用户：转账失败

第二个问题：

笑傲流云的答案不错。结合前面课程，用流量控制也可实现



👍 2



亚洲舞王.尼古拉斯赵...

2019-08-13

1.老师，能否解释一下为什么“使用异步编程模型之后，在少量请求之下，时延依旧是100ms，但是在大量请求之下，异步的实现不需要等待线程的执行结果”？少量请求不也不需要等待吗

2.如果使用异步方式实现，我的onComplete()方法在另一个线程里执行，我怎么通知我的客户端我执行成功还是失败呢？

作者回复: A1: 是的，异步方式下少量和大量请求都不需要等待执行结果。

A2: 在onComplete()方法中通知。



👍 2

