10-如何使用异步设计提升系统性能？

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。

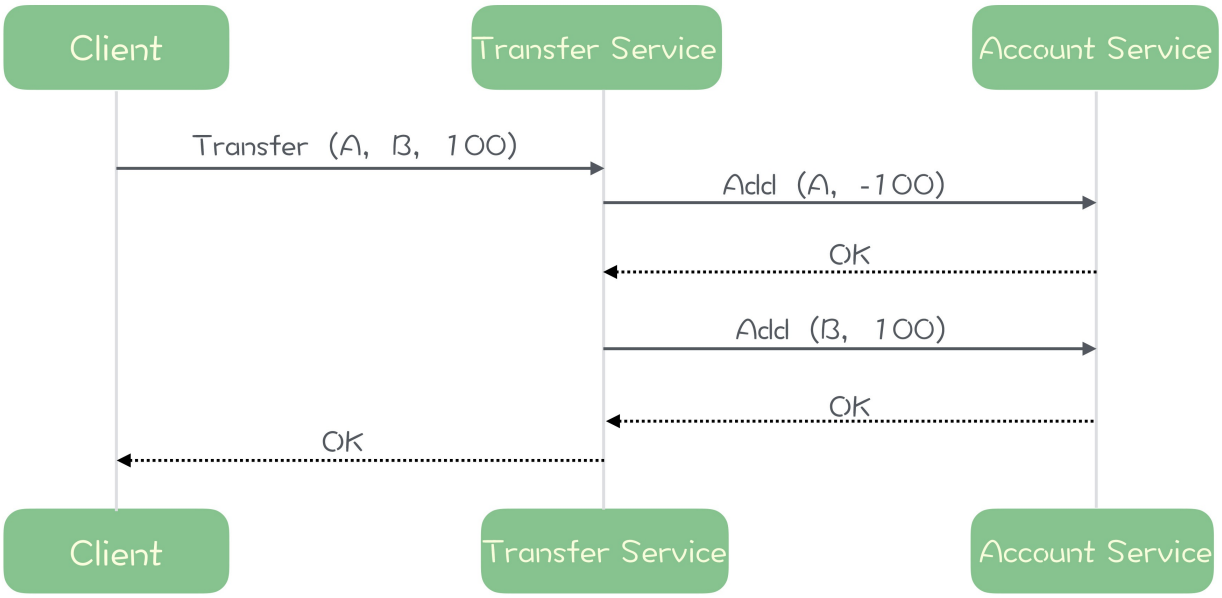
异步设计如何提升系统性能？

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：

```
Transfer(accountFrom, accountTo, amount) {
  // 先从accountFrom的账户中减去相应的钱数
  Add(accountFrom, -1 * amount)
  // 再把减去的钱数加到accountTo的账户中
  Add(accountFrom, amount)
  return OK
}
```

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 （个线程）* 10（次请求每秒） = 100,000 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。

```
TransferAsync(accountFrom, accountTo, amount, OnComplete()) {
  // 异步从accountFrom的账户中减去相应的钱数，然后调用OnDebit方法。
  AddAsync(accountFrom, -1 * amount, OnDebit(accountTo, amount, OnAllDone(OnComplete())))
}
// 扣减账户accountFrom完成后调用
OnDebit(accountTo, amount, OnAllDone(OnComplete())) {
  //  再异步把减去的钱数加到accountTo的账户中，然后执行OnAllDone方法
  AddAsync(accountTo, amount, OnAllDone(OnComplete()))
}
// 转入账户accountTo完成后调用
OnAllDone(OnComplete()) {
  OnComplete()
}
```

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。

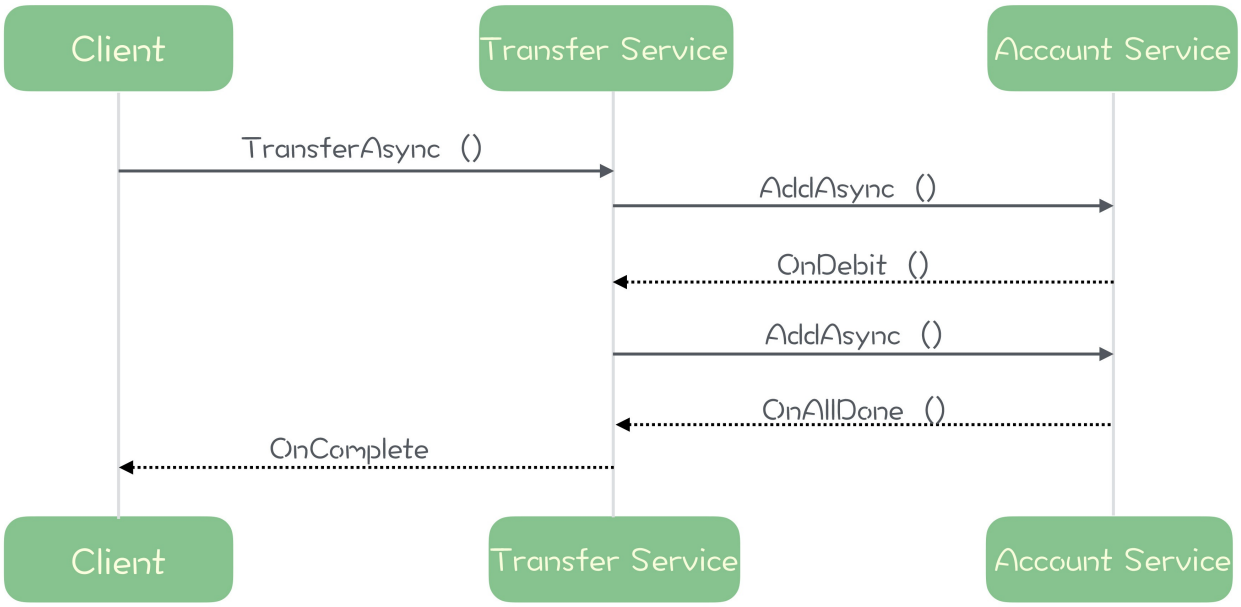
异步的实现过程相对于同步来说，稍微有些复杂。我们先定义2个回调方法：

- **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，几乎囊获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：

```
/**
 * 账户服务
 */
public interface AccountService {
    /**
     * 变更账户金额
     * @param account 账户ID
     * @param amount 增加的金额，负值为减少
     */
    CompletableFuture<Void> add(int account, int amount);
}
```

```
/**
 * 转账服务
 */
public interface TransferService {
    /**
     * 异步转账服务
     * @param fromAccount 转出账户
     * @param toAccount 转入账户
     * @param amount 转账金额，单位分
     */
    CompletableFuture<Void> transfer(int fromAccount, int toAccount, int amount);
}
```

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CompletableFeture，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：

```
/**
 * 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)
      // 然后调用add方法给toAccount增加相应金额
      .thenCompose(v -> accountService.add(toAccount, amount));
    }
}
```

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CompletableFeture对象，可以用CompletableFeture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。

```
public class Client {
    @Inject
    private TransferService transferService; // 使用依赖注入获取转账服务的实例
    private final static int A = 1000;
    private final static int B = 1001;

    public void syncInvoke() throws ExecutionException, InterruptedException {
        // 同步调用
        transferService.transfer(A, B, 100).get();
        System.out.println("转账完成! ");
    }

    public void asyncInvoke() {
        // 异步调用
        transferService.transfer(A, B, 100)
                .thenRun(() -> System.out.println("转账完成! "));
    }
}
```

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，在性能足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。

精选留言：

- senekis 2019-08-13 08:33:16

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。 [4赞]

- 笑傲流云 2019-08-13 13:24:31
  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。 [2赞]

- 不会爬树的熊 2019-08-13 09:45:24
  第一个问题: 俺觉着吧，账户服务那噶如果能增加一个查询服务，超时+调用失败都可以通过查询确认当前账户状态，延后再决定下面儿咋处理。

  第二个问题:就是老师基础篇聊的幂等方案么+请求顺序的方案么

  谢谢老师给点评 [1赞]

- 蓝魔丶 2019-08-13 08:47:09
  老师，转账例子代码中给转入账号加钱写错了吧 [1赞]

- DAV 2019-08-13 15:39:18
  请教一下，在整个消息队列的场景里面怎么融合异步调用？举例，A发送消息到消息队列，消费进程处理后如何通过回调形式返回结果给A?

- Better me 2019-08-13 14:35:11
  对于思考题:
  1、应该可以通过编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

- 川杰 2019-08-13 12:27:07
  老师，请教个问题，吞吐量增加可以理解，因为请求发生后就直接返回了，从而避免了后续等待的延时；但是，以今天内容为例：
  1、TransferAsync请求发生，直接返回，并开启新线程处理OnDebit函数；
  2、OnDebit处理完毕，开启新线程处理OnAllDone函数；
  3、OnAllDone函数处理完毕；
  那么，从宏观来看，线程数量是不是要比同步多很多？
  还有一个问题，调用方如何得知转账成功？前台开启一个新线程去轮询吗？

- 许童童 2019-08-13 11:46:51
  如何将错误报告给客户端?
  javascript中用.catch捕获异常

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？
事务补偿

- 亚洲舞王.尼古拉斯.赵四 2019-08-13 11:27:48
  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？

- 广训 2019-08-13 09:34:07
  accountService如果本身是自己处理逻辑，那将其放入一个事务中就解决部分失败。如果调用三方服务，就比较麻烦。需要把两步操作都留存，失败的列表每天都要处理。

  一般都有专门的异步线程池来运行异步task任务，比如spring 的async。

- Jxin 2019-08-13 09:28:55
  1.提个问题，为什么要1w个线程？java的线程模型不是1:1的吗，实际在跑的线程仅有核数*2，1w这个数量不是造成了内存浪费和上下文切换成本吗。
  2.异步异常回传，并传回当前操作绑定在线程本地空间的事务实例。只有当两次都成功才提交两个add的事务。（事务不能声明，只能手动开启和提交了）

- A:春哥大魔王 2019-08-13 09:19:46
  jdk1.8之前没有completablefuture应该用什么搞异步呢？

- 盛 2019-08-13 09:11:01
  其实异步减少的是等待时间。关于第一个思考题：调用失败是不是就可以抛出error，直接不执行第一步；代码中是否少了相应的查询操作进行核对啊，否则万一就根本没有执行成功呢。操作完成后相应的查询核对应当时候放到了后面吧：有问题再进行相应的类似于回滚之类的操作。
  第二个问题oncomplete()在什么线程中执行这个不是很清楚：不过我觉得可以通过线程去控制回调方法的执行线程数，具体实现方式就暂时还没想到。

- HadesFX 2019-08-13 09:00:24
  1.可以增加异步调用后的返回值用来判断是否成功。
  2.应该是存在一个默认线程池，使用线程池中的线程，可能存在一个配置自定义线程池的方法用来定制控制数量。
  不知道理解的对不对。

- 业余草 2019-08-13 08:47:40
  异步虽好，但使用场景有限！

- 白小白 2019-08-13 08:28:11
  嗯……第一遍有点懵……我再来两遍……再回答思考问题……