# 10 | 如何使用异步设计提升系统性能？

2019-08-13 李玥

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。

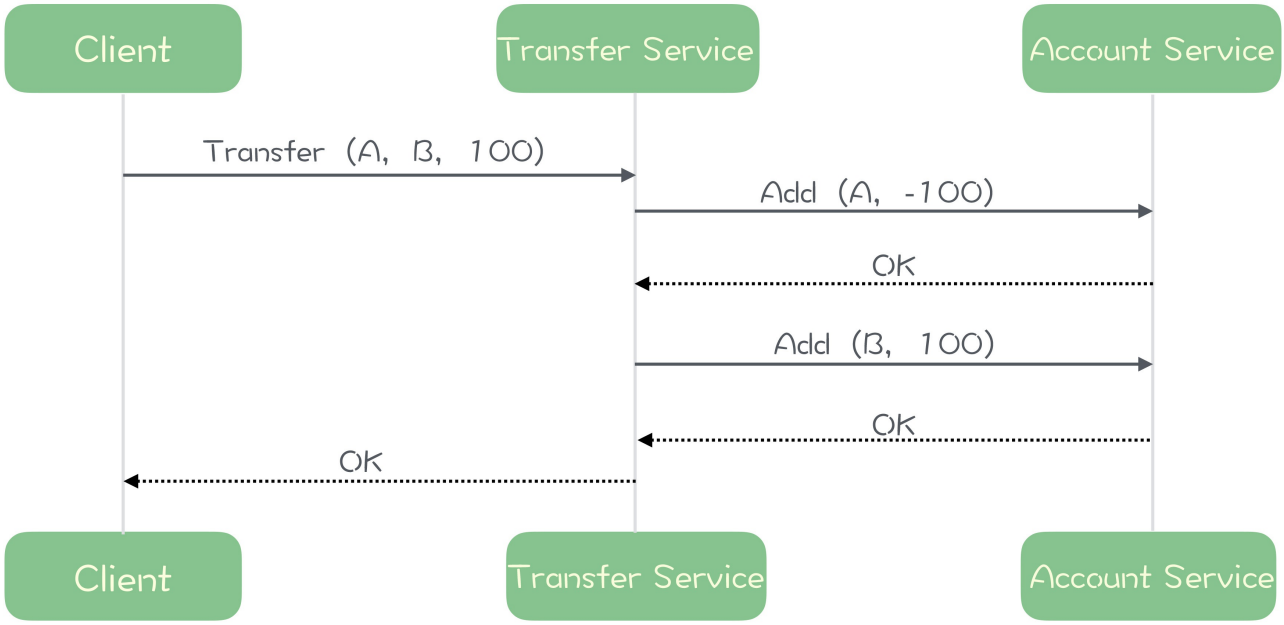
## 异步设计如何提升系统性能？

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(accountTo, 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。

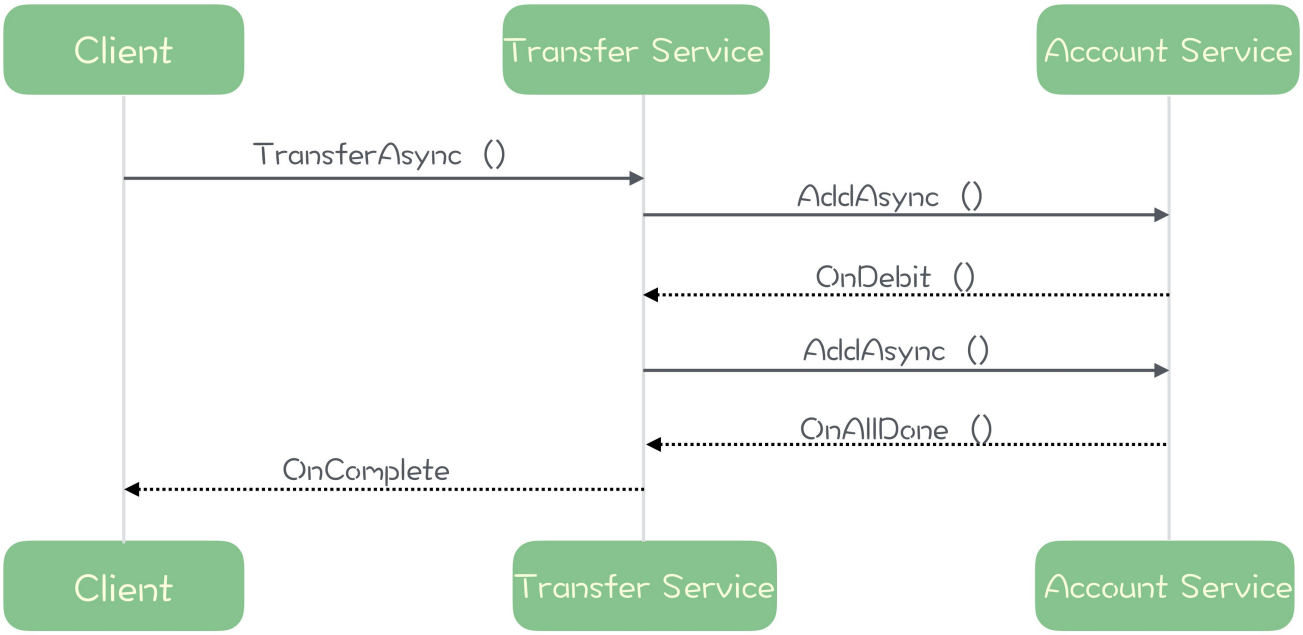
异步的实现过程相对于同步来说，稍微有些复杂。我们先定义2个回调方法：

- **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

小伟

1

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的业务逻辑，这里也没有任何等待。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。

个人感觉把关联性强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。

2019-08-22

作者回复

人工置顶

2019-08-23

笑傲流云

17

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。

2019-08-13

作者回复

2019-08-14

senekis

10

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。

2019-08-13

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、交接工作上了吗？

2019-08-14

付永强 5

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cup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。

2019-08-13

作者回复

线程就是为了能自动分配CPU时间片而生的。

2019-08-14

linqw 3

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)
)
// 然后调用 add 方法给 toAccount 增加相应金额
.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

2019-08-13

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。

2019-08-14

谢清 2

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

2019-08-13

Better me 2

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

2019-08-13

作者回复

[illegible]

2019-08-14

蓝魔、

1

老师，转账例子代码中给转入账号加钱写错了吧

2019-08-13

作者回复

感谢指正，我尽快让编辑小姐姐改正。

2019-08-14

godtrue

0

课后思考及问题

1：什么是同步？

大望：安红嫁给我吧—等待回复，然后准备婚礼

2：什么是异步？

大望：安红嫁给我吧—然后准备婚礼，回复不嫁，则哭死，嫁，则婚礼事宜也准备好了

3：同步和异步的区别？

同步—正常的人思维，简单直接，但会被等待阻塞

异步—想要更快，不等待回复，直接进行下一步，等待通知是否OK，若OK，则皆大欢喜，提高了速度，否则要想好怎么收场，有点费事

4：常说异步比同步快，那她快在哪里？

异步快就快在不等慢人，所以，合作起来烦人一点

同步更易合作，代价就是要等一等肉喇叭几的人

想要走的更快就一个人走，想要走的更远就一群人走—异步单独走确实更快，但异步还想一群人走，那就容易出问题啦！

5：异步是怎么实现的？

这个需要老师回答一下，感觉文中只是讲了明面上异步大概怎么实现的，水下的动作没有讲。比如：异步说是发个命令执行某个动作，那发给谁了？那个动作什么时候执行完，总需要人监控着吧？不然谁知道啥时候执行完？另外，即使执行完了，什么时候通知？怎么通知？通知谁？如果通知她的时候，她正在忙怎么办？是等待还是打断强行通知？另外，我觉得问题的根源在于，处理速度有差异，有了阶级什么事就需要分出个三六九等，就需要兼顾平等、效率，因为无论如何大家是有速度差的，有都生活在一块不得不分工协作。

希望，老师就以上问题，能在细致的回答一下，多谢！

6：异步的使用原则

1：异步虽快，也不要贪多哟—能不用，就不用

2：前后结果有依赖，不可用

3：复杂系统，不建议用，会增加维护成本
4：尽量坚守第一条

2019-08-22

作者回复

回答你的问题，异步具体是怎么实现的？

安红.嫁给我吧(然后安红你要干啥看这里());

func 然后安红你要干啥看这里()
if(不同意）
给我打个电话让我哭死吧。
else
准备婚礼，摆酒，拍婚纱照balabala.....

也就是说后面的事儿都丢给安红来做了。

2019-08-22

timmy21 0

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？

2019-08-21

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。

2019-08-22

DFighting 0

单个线程，其实同步和异步都没啥区别，但是多线程情况下，因为只有一个CPU，处理每个阻塞线程的询问请求都会涉及到不同线程上下文的切换乃至用户态、核心态的转换，这才是最耗时的。

2019-08-20

缺点就是太吊 0

我在想，转账操作是原子操作，我们可不可以直接通过amountService定义转账方法，通过数据库事务保证，amountService.transfer(fromAccount, toAccount, money);transerService只是简单委托accountService

2019-08-20

作者回复

这样做是可以的。

2019-08-21

Alexdown 0

老师文章末尾留的问题都不错，希望老师在下一篇文章结尾处给出老师的答案供参考，谢谢

2019-08-15

monalisali 0

老师，问个问题：如何CPU和内存占有率都很高，用异步可以解决吗？

2019-08-14

作者回复

这个问题异步解决不了。如果说真的cpu或者磁盘占用率达到100%了，并且你的代码逻辑没什么问题，那这就是程序的极限了。

2019-08-16

monalisali 0

思考题一：

可以写一个兜底函数类似与catch，所有异常都走到这个函数中。然后通过传入catch的参数来判断错误类型，并决定后续操作

转入方法也类似，先判断下转出方法是否成功，成功了再执行。

2019-08-14

海罗沃德 0

老师能否对比一下异步和StreamingData，据说StreamingData可以让线程使用率更高，效率比异步处理也更高，这是怎么实现的？

2019-08-14

作者回复

你说的“StreamingData“指的是Lambda表达式还是流计算，或是其他什么技术呢？能具体说一下吗？

2019-08-16

Liam 0

异步实现里面还是要用线程池限制一下线程数吧，否则没有达到减少线程的效果

2019-08-14

作者回复

是的，一般都会使用线程池。

2019-08-14

DAV 0

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?

2019-08-13

作者回复

可以继续学习后面的课程，我们会有相应的源码分析。

2019-08-14

川杰

0

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？

2019-08-13

作者回复

第一，OnDebit()和OnAllDone()可以在同一个线程中执行。没必要每个回调方法都开启一个新的线程。

第二，由于不需要等待，执行每个异步方法的耗时会非常短。

第三，可以使用线程池来避免反复创建销毁线程的开销，所以只需要很少的线程。

最后一个问题，“通知转账成功”这个业务逻辑，不一定非得在接收请求的那个线程里面执行，可以直接在OnAllDone()里面通知转账成功。

2019-08-14

许童童

0

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

2019-08-13

作者回复

2019-08-14