30 | 理论四：如何通过封装、抽象、模块化、中间层等解耦代码？

2020-01-10 王争

讲述：冯永吉

时长 12:51　大小 10.31M

前面我们讲到，重构可以分为大规模高层重构（简称“大型重构”）和小规模低层次重构（简称“小型重构”）。大型重构是对系统、模块、代码结构、类之间关系等顶层代码设计进行的重构。对于大型重构来说，今天我们重点讲解最有效的一个手段，那就是“解耦”。解耦的目的是实现代码高内聚、松耦合。关于解耦，我准备分下面三个部分来给你讲解。

“解耦”为何如此重要？

如何判定代码是否需要“解耦”？

如何给代码“解耦”？

话不多说，现在就让我们正式开始今天的学习吧！

“解耦”为何如此重要？

软件设计与开发最重要的工作之一就是应对复杂性。人处理复杂性的能力是有限的。过于复杂的代码往往在可读性、可维护性上都不友好。那如何来控制代码的复杂性呢？手段有很多，我个人认为，最关键的就是解耦，保证代码松耦合、高内聚。如果说重构是保证代码质量不至于腐化到无可救药地步的有效手段，那么利用解耦的方法对代码重构，就是保证代码不至于复杂到无法控制的有效手段。

我们在第 22 讲有介绍，什么是“高内聚、松耦合”。如果印象不深，你可以再去回顾一下。实际上，“高内聚、松耦合”是一个比较通用的设计思想，不仅可以指导细粒度的类和类之间关系的设计，还能指导粗粒度的系统、架构、模块的设计。相对于编码规范，它能够在更高层次上提高代码的可读性和可维护性。

不管是阅读代码还是修改代码，“高内聚、松耦合”的特性可以让我们聚焦在某一模块或类中，不需要了解太多其他模块或类的代码，让我们的焦点不至于过于发散，降低了阅读和修改代码的难度。而且，因为依赖关系简单，耦合小，修改代码不至于牵一发而动全身，代码改动比较集中，引入 bug 的风险也就减少了很多。同时，“高内聚、松耦合”的代码可测试性也更加好，容易 mock 或者很少需要 mock 外部依赖的模块或者类。

除此之外，代码“高内聚、松耦合”，也就意味着，代码结构清晰、分层和模块化合理、依赖关系简单、模块或类之间的耦合小，那代码整体的质量就不会差。即便某个具体的类或者模块设计得不怎么合理，代码质量不怎么高，影响的范围是非常有限的。我们可以聚焦于这个模块或者类，做相应的小型重构。而相对于代码结构的调整，这种改动范围比较集中的小型重构的难度就容易多了。

代码是否需要“解耦”？

那现在问题来了，我们该怎么判断代码的耦合程度呢？或者说，怎么判断代码是否符合“高内聚、松耦合”呢？再或者说，如何判断系统是否需要解耦重构呢？

间接的衡量标准有很多，前面我们讲到了一些，比如，看修改代码会不会牵一发而动全身。除此之外，还有一个直接的衡量标准，也是我在阅读源码的时候经常会用到的，那就是把模块与模块之间、类与类之间的依赖关系画出来，根据依赖关系图的复杂性来判断是否需要解耦重构。

如果依赖关系复杂、混乱，那从代码结构上来讲，可读性和可维护性肯定不是太好，那我们就需要考虑是否可以通过解耦的方法，让依赖关系变得清晰、简单。当然，这种判断还是有比较强的主观色彩，但是可以作为一种参考和梳理依赖的手段，配合间接的衡量标准一块来使用。

如何给代码“解耦”？

前面我们能讲了解耦的重要性，以及如何判断是否需要解耦，接下来，我们再来看一下，如何进行解耦。

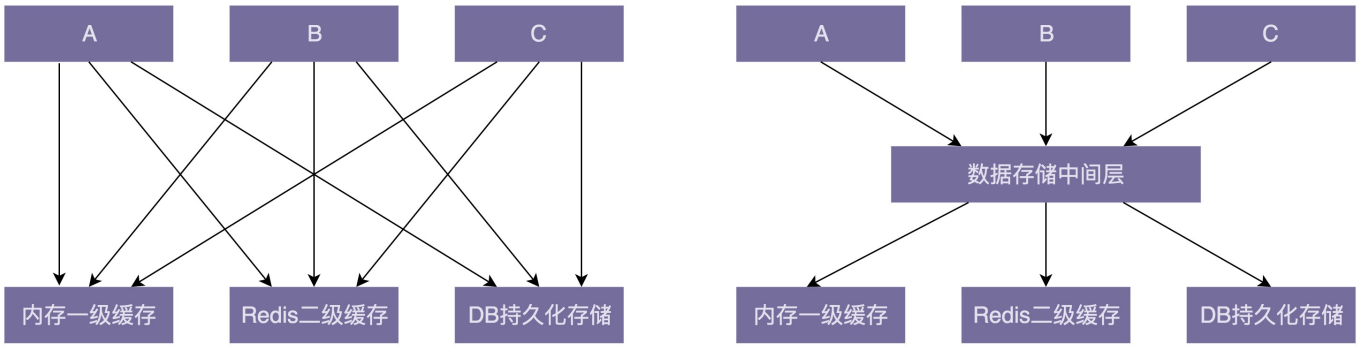
1. 封装与抽象

封装和抽象作为两个非常通用的设计思想，可以应用在很多设计场景中，比如系统、模块、lib、组件、接口、类等等的设计。封装和抽象可以有效地隐藏实现的复杂性，隔离实现的易变性，给依赖的模块提供稳定且易用的抽象接口。

比如，Unix 系统提供的 open() 文件操作函数，我们用起来非常简单，但是底层实现却非常复杂，涉及权限控制、并发控制、物理存储等等。我们通过将其封装成一个抽象的 open() 函数，能够有效控制代码复杂性的蔓延，将复杂性封装在局部代码中。除此之外，因为 open() 函数基于抽象而非具体的实现来定义，所以我们在改动 open() 函数的底层实现的时候，并不需要改动依赖它的上层代码，也符合我们前面提到的“高内聚、松耦合”代码的评判标准。

2. 中间层

引入中间层能简化模块或类之间的依赖关系。下面这张图是引入中间层前后的依赖关系对比图。在引入数据存储中间层之前，A、B、C 三个模块都要依赖内存一级缓存、Redis 二级缓存、DB 持久化存储三个模块。在引入中间层之后，三个模块只需要依赖数据存储一个模块即可。从图上可以看出，中间层的引入明显地简化了依赖关系，让代码结构更加清晰。



除此之外，我们在进行重构的时候，引入中间层可以起到过渡的作用，能够让开发和重构同步进行，不互相干扰。比如，某个接口设计得有问题，我们需要修改它的定义，同时，所有调用这个接口的代码都要做相应的改动。如果新开发的代码也用到这个接口，那开发就跟重构冲突了。为了让重构能小步快跑，我们可以分下面四个阶段来完成接口的修改。

第一阶段：引入一个中间层，包裹老的接口，提供新的接口定义。

第二阶段：新开发的代码依赖中间层提供的新接口。

第三阶段：将依赖老接口的代码改为调用新接口。

第四阶段：确保所有的代码都调用新接口之后，删除掉老的接口。

这样，每个阶段的开发工作量都不会很大，都可以在很短的时间内完成。重构跟开发冲突的概率也变小了。

3. 模块化

模块化是构建复杂系统常用的手段。不仅在软件行业，在建筑、机械制造等行业，这个手段也非常有用。对于一个大型复杂系统来说，没有人能掌控所有的细节。之所以我们能搭建出如此复杂的系统，并且能维护得了，最主要的原因就是将系统划分成各个独立的模块，让不同的人负责不同的模块，这样即便在不了解全部细节的情况下，管理者也能协调各个模块，让整个系统有效运转。

聚焦到软件开发上面，很多大型软件（比如 Windows）之所以能做到几百、上千人有条不紊地协作开发，也归功于模块化做得好。不同的模块之间通过 API 来进行通信，每个模块之间耦合很小，每个小的团队聚焦于一个独立的高内聚模块来开发，最终像搭积木一样将各个模块组装起来，构建成一个超级复杂的系统。

我们再聚焦到代码层面。合理地划分模块能有效地解耦代码，提高代码的可读性和可维护性。所以，我们在开发代码的时候，一定要有模块化意识，将每个模块都当作一个独立的 lib 一样来开发，只提供封装了内部实现细节的接口给其他模块使用，这样可以减少不同模块之间的耦合度。

实际上，从刚刚的讲解中我们也可以发现，模块化的思想无处不在，像 SOA、微服务、lib 库、系统内模块划分，甚至是类、函数的设计，都体现了模块化思想。如果追本溯源，模块化思想更加本质的东西就是分而治之。

4. 其他设计思想和原则

“高内聚、松耦合”是一个非常重要的设计思想，能够有效提高代码的可读性和可维护性，缩小功能改动导致的代码改动范围。实际上，在前面的章节中，我们已经多次提到过这个设计思想。很多设计原则都以实现代码的“高内聚、松耦合”为目的。我们来一块总结回顾一下都有哪些原则。

单一职责原则

我们前面提到，内聚性和耦合性并非独立的。高内聚会让代码更加松耦合，而实现高内聚的重要指导原则就是单一职责原则。模块或者类的职责设计得单一，而不是大而全，那依赖它的类和它依赖的类就会比较少，代码耦合也就相应的降低了。

基于接口而非实现编程

基于接口而非实现编程能通过接口这样一个中间层，隔离变化和具体的实现。这样做的好处是，在有依赖关系的两个模块或类之间，一个模块或者类的改动，不会影响到另一个模块或类。实际上，这就相当于将一种强依赖关系（强耦合）解耦为了弱依赖关系（弱耦合）。

依赖注入

跟基于接口而非实现编程思想类似，依赖注入也是将代码之间的强耦合变为弱耦合。尽管依赖注入无法将本应该有依赖关系的两个类，解耦为没有依赖关系，但可以让耦合关系没那么紧密，容易做到插拔替换。

多用组合少用继承

我们知道，继承是一种强依赖关系，父类与子类高度耦合，且这种耦合关系非常脆弱，牵一发而动全身，父类的每一次改动都会影响所有的子类。相反，组合关系是一种弱依赖关系，这种关系更加灵活，所以，对于继承结构比较复杂的代码，利用组合来替换继承，也是一种解耦的有效手段。

迪米特法则

迪米特法则讲的是，不该有直接依赖关系的类之间，不要有依赖；有依赖关系的类之间，尽量只依赖必要的接口。从定义上，我们明显可以看出，这条原则的目的就是为了实现代码的松耦合。至于如何应用这条原则来解耦代码，你可以回过头去阅读一下第 22 讲，这里我就不赘述了。

除了上面讲到的这些设计思想和原则之外，还有一些设计模式也是为了解耦依赖，比如观察者模式，有关这一部分的内容，我们留在设计模式模块中慢慢讲解。

重点回顾

好了，今天的内容到此就讲完了。我们来一块总结回顾一下，你需要重点掌握的内容。

1.“解耦”为何如此重要?

过于复杂的代码往往在可读性、可维护性上都不友好。解耦保证代码松耦合、高内聚，是控制代码复杂度的有效手段。代码高内聚、松耦合，也就是意味着，代码结构清晰、分层模块化合理、依赖关系简单、模块或类之间的耦合小，那代码整体的质量就不会差。

2. 代码是否需要“解耦”？

间接的衡量标准有很多，比如，看修改代码是否牵一发而动全身。直接的衡量标准是把模块与模块、类与类之间的依赖关系画出来，根据依赖关系图的复杂性来判断是否需要解耦重构。

3. 如何给代码“解耦”？

给代码解耦的方法有：封装与抽象、中间层、模块化，以及一些其他的设计思想与原则，比如：单一职责原则、基于接口而非实现编程、依赖注入、多用组合少用继承、迪米特法则

等。当然，还有一些设计模式，比如观察者模式。

课堂讨论

实际上，在我们平时的开发中，解耦的思想到处可见，比如，Spring 中的 AOP 能实现业务与非业务代码的解耦，IOC 能实现对象的构造和使用的解耦。除此之外，你还能想到哪些解耦的应用场景吗？

欢迎在留言区写下你的思考和答案，和同学一起交流和分享。如果有收获，也欢迎你把这篇文章分享给你的朋友。

上一篇　29 | 理论三：什么是代码的可测试性？如何写出可测试性好的代码？

下一篇　31 | 理论五：让你最快速地改善代码质量的20条编程规范（上）

精选留言 (31)

写留言

Geek_Zjy

2020-01-10

必须留个言，倾诉倾诉。
昨天晚上就因为看争哥直播，3岁儿子把 mac 的屏给我弄碎了，这一下子看直播的代价也太惨重了，5千多。
重点是我还只看了个开头o(╥﹏╥)o

4 19

下雨天
2020-01-10

消息队列，事件监听实现了被观察者和观察者的解耦！

展开

9

李小四
2020-01-10

设计模式_30
作业
消息队列，作为观察者模式的的代表，极大程度地实现了解耦，也在很大程度上解决了资源有限时的高并发崩溃。
我认为API的使用也算是一种解耦吧，将客户端与服务端，将不同模块的服务可以高效配...

展开

1 6

Jeff.Smile
2020-01-10

重构是术与道的结合，道为重构的思路，指南。术是具体的手段！

1 2

Ken张云忠
2020-01-10

实际上，在我们平时的开发中，解耦的思想到处可见，比如，Spring 中的 AOP 能实现业务与非业务代码的解耦，IOC 能实现对象的构造和使用的解耦。
除此之外，你还能想到哪些解耦的应用场景吗？
解耦是人类应对复杂性问题的有效手段,解耦的核心是拆分,横向可以拆分出不同的模块,纵向可以拆分出不同的工序,然后就有了人类的大分工协作,分工协作可以把大规模的人有效组...

展开

2

王涛
2020-01-10

代码解耦的第二种方式，中间层。

上层代码都依赖中间层代码，中间层也是使用基于借口而非实现编程。
抽象出中间层肯定是好的，但这样是否也会带来另一个问题: 中间层接口变动必然会影响所有上层代码调用，接口的影响面是否是变大了？如果是的话，下一步有该怎么优化呢?

展开

2

辣么大
2020-01-10

docker 通过容器打包应用，解耦应用和运行平台。

展开

1

桂城老托尼
2020-01-10

通过消息中间件实现的生产与消费的解耦;
通过SPI回调实现的主流程与个性化编排实现的解耦;
同步调用改为异步调用理论上也算调用与被调用的解耦;

展开

1

lyshrine
2020-01-13

依赖注入是不是也算是组合?

展开

饭粒
2020-01-13

Linux 虚拟文件系统解耦系统调用和具体的文件系统实现；TCP/IP 网络协议分层。

L
2020-01-13

rabbitmq等消息队列，采用了观察者模式，一定程度上实现了解耦

番茄炒西红柿
2020-01-12

解耦这是我突然想到现在推行的事件驱动编程，本身就是解耦思想的一种体现

平风造雨
2020-01-11

消息列队，事件驱动，都是典型的解耦。

展开

Frank
2020-01-11

目前能想到的解耦手段有以下两个：
1. 引入消息中间件实现业务系统之间的解耦；
2. 分层思想，如MVC思想，网络的OSI七层参考模型；

展开

黄林晴
2020-01-10

打卡✓

展开

编程界的小学生
2020-01-10

中间件也可以解耦，MVC架构也是解耦。

展开

此鱼不得水
2020-01-10

大佬多来一些例子哈哈

展开

Jxin
2020-01-10

1.防腐层，解耦本地服务和远程服务的api依赖。但这不过是中间层的一种落地范式，其核心原则也属于设计原则中的迪米乐和接口隔离（由api客服端实现）。

2.事件机制，解决跨服务的事务操作依赖。常规范式就是mq了。（进程级的事件机制也

有，但分布式场景很少用，毕竟mq能做的进程级事件不一定能做比如消息持久化，进程...

展开

DullBird

2020-01-10

Mq消息通知，数据缓存通过mysql binlog监听更新, 数据的领域层serivce，和业务service之间的调用关系，spring security中authentication实现和Session的解耦

展开

睡觉

2020-01-10

之前做过一个电商系统。交易需要记录在区块链上。当时采用的就是解耦的思想。电商系统还是负责原有的业务。通过rpc将交易数据传递到区块链服务进行入链业务。