35 | 网站性能优化（下）

2019-11-29 四火

全栈工程师修炼指南 进入课程 >

讲述：四火

时长 21:12　大小 14.57M

你好，我是四火。

上一讲我们介绍了网站性能优化的基础知识，包括性能指标、关注点，以及寻找性能瓶颈的思路。那么这一讲，我们就来介绍网站性能优化的具体方法，我们将从产品和架构、后端和持久层，以及前端和网络层这样三个部分分别展开。优化的角度和方法可以说花样繁多，我在这里举一些典型的例子，希望既能给你一些内容上的介绍，进而拓宽视野，也能给你一些思考角度上的启发。

产品和架构调整

对于一个应用来说，产品和架构恰巧是两个互相对立而又相辅相成的角度。作为全栈工程师，我们当然鼓励追求细节，但是在考虑性能优化的时候，我认为还是要**优先考虑从大处着眼，而不是把大量时间花费在小处的细节提升上**，以期望获得较为明显的效果。这里的“大处”，就主要包含了产品和技术架构两个维度。

1. 同步变异步

如果页面聚合在服务端进行，那么渲染前等待的时间，在整个任务依赖树上面，取决于最慢的一个路径什么时候完成；而如果页面聚合是在客户端进行的，那么页面每一个子区域的渲染往往都可以以 Ajax 的方式独立进行，且同时进行，而母页面则可以首先展示给用户，减少用户的等待时间。

这里我还想补充一点，我们可以把同步和异步结合起来使用以获得最好的效果。比方说，用户对于网页加载的延迟是很敏锐的，但是用户对于一个页面上不同的信息，关注程度是不同的。

举例来说，一篇文章，标题可能是最先关注的，其次才是作者或是正文，至于评论、广告等等这些内容，优先级则更低。因此，我们可以让标题和正文内容的第一屏等高优先级的内容在服务端进行聚合后一并同步返回，这就省去了 Ajax 二次调用的时间开销，而次要内容或是某些生成特别耗时的内容，则可以使用异步方式在客户端单独加载。

2. 远程变本地

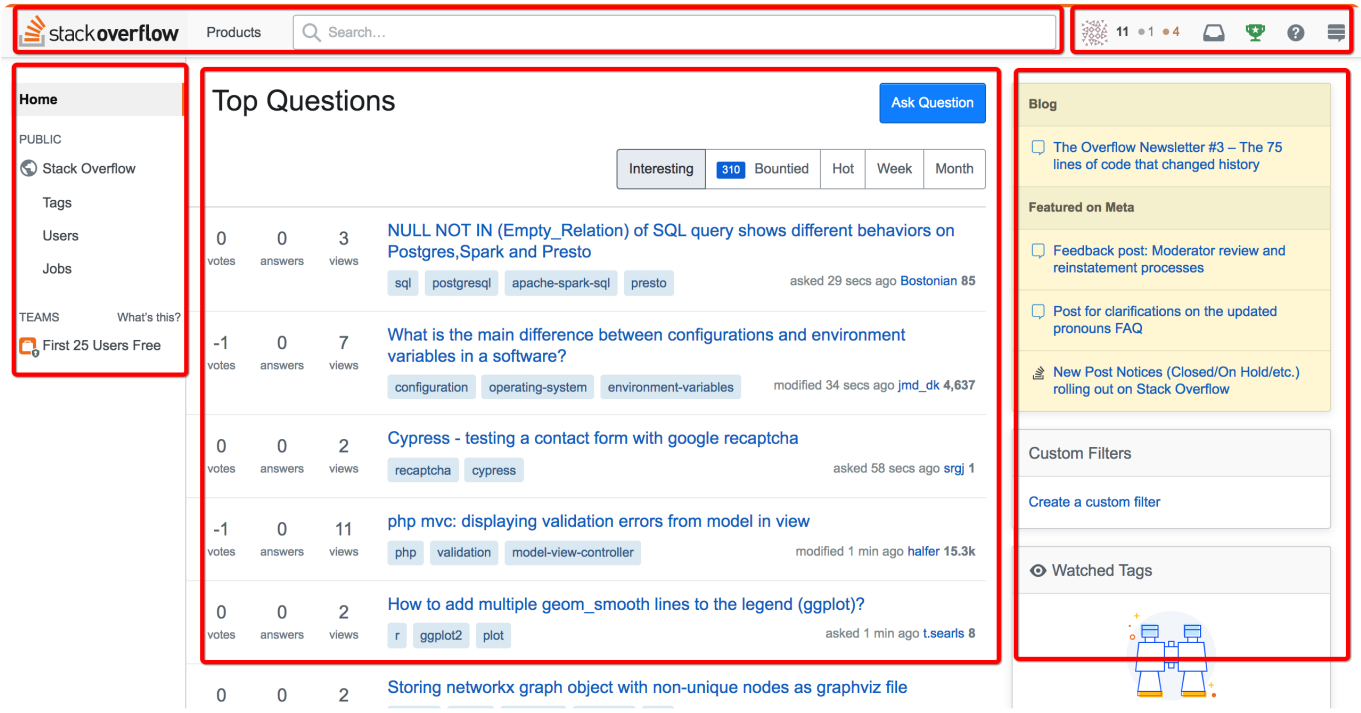
有些时候，如果能够容许牺牲一定的一致性，而将数据从远程的数据中心冗余到“本地”这样离数据使用者更近的节点，可以减少数据获取的延迟。DNS 的均衡路由就是一个例子，不同地区的用户访问同一个域名的时候，可以被定向到不同的离自己更近的节点上去；CDN 也是一个很典型的例子，静态资源可以从较近的本地节点获取。

3. 页面静态化

对于为什么要从大处着眼，我来举一个我在项目中经历过的例子。在我们的一个网站项目，模板中使用了大量的 OGNL 表达式，在做了 profiling 之后，发现 OGNL 表达式占用了相当比例的 CPU，而将其改成 EL 表达式等其它方式，这样做确实可以降低这些 CPU 的使用。这件事情其实没有错，但是这个处理的优先级应该往下放，因为这样的优化，可能只会带来 10% 左右的最终性能提升。

但是另一方面，我们逐渐引入了页面静态化技术，对于几个关键页面，它直接带来了 300% 到 800% 的性能提升，这就让前面的页面模板的调整显得无足轻重了。性能优化就是这样，并非简单的“一分耕耘一分收获”，有时候我们能找到一些优化的办法，看起来效果很明显，但是调整的代价却不大。

关于页面静态化，我来举一个 StackOverflow 的例子：



你看这个 StackOverflow 的页面，我们就可以按照页面静态化的原则来分析：把它划分为几个不同的区域，每个区域都可以具备自己特有的一致性要求，无论是页面还是数据，都可以单独做不同条件下的缓存。

比方说，正上方和左侧区域是不会变的，这些内容可以写在静态的 HTML 页面模板里，可以说是完全“静态”的；中间的主问题区域，则可能是定期或按一定规则刷新的，相当于在过期时间内也是静态的；而右上角和用户相关的数据，则需要每次页面访问实时生成，以便让不同的用户看到特有的属于他自己的内容，也就是说，这部分内容是完全“动态”的。

页面静态化的实践中，我们可以将页面解耦成不同的部分：

> 从产品的角度来定义每一个部分允许的一致性窗口，比如有的数据是一小时更新一次，数据可以不用非常准确，而有的则是要实时，数据要求非常准确；

而技术角度，对于每一部分不同的一致性要求，依赖于缓存的特性，也就是空间换时间，我们可以让每个部分进行分别管理，最终聚合起来（页面聚合的方式请参见 [第 09 讲]）。

在极端的情况下，整个页面是可以直接完全被缓存起来，甚至直接预先生成，而做到全页面静态化的，比如一些几乎不存在个性内容的静态博客站点就是如此。

当然，还有许多常见的其它架构上的设计，起着提高网站性能的作用，没有展开介绍，是因为它们已经被介绍过，或者是大家普遍比较熟悉了。比如说，对反向代理、集群和负载分担的使用，这些技术我们分别在 [第 28 讲] 和 [第 29 讲] 介绍过，如今几乎所有的大型网站都使用它们来组网。

后端和持久层优化

1. 串行变并行

道理上很简单，串行的逻辑，在没有依赖限制的情况下，可以并行执行。后端的逻辑如果需要执行多项操作，那么如果没有依赖，或者依赖项满足的情况下，可以立即执行，而不必一个一个挨个等待依次完成。Spring 的 @Async 注解，可以比较方便地将普通的 Java 方法调用变成异步进行的，用这种方法可以同时执行数个互不依赖的方法。

有些朋友可能知道，Amazon 是 SOA 架构（Service-Oriented Architecture，面向服务架构）最早的践行者。贝索斯在 2002 年的时候，就开始要求 Amazon 内部的所有服务，都只能以 Web 接口的形式暴露出来，供其他团队调用，而每个服务，都由专门的团队维护。如今，访问亚马逊的一个商品页，借助 SOA 架构，后台都要调用成百上千个开放的服务，你可以想象，如果这些调用是串行进行的，页面的加载时间将难以想象。

2. 数据库索引创建

凡是提到关系数据库的优化，索引的创建往往是很容易想到的优化方法之一。而对于某些支持半结构化数据存储的非关系数据库，往往也对索引存在有限的支持。

索引的创建，最常见的一个原因，是为了在查询的时候，显著减少消耗的时间。由于索引是单独以 B 树或者是 B+ 树等变种来存储的，而我们知道这样的数据结构查询的速度可以达到 log(n)，和原表相比，索引在检索时的数据的读取量又很小，因此查询速度的提升往往是立竿见影的。

但是索引创建并非是没有代价的，在关系数据库中，索引作为原表中索引列数据的一份冗余，维护它自然是有开销的，每当索引数据增、删、改发生的时候，索引也需要相应地发生变化。

3. 数据库表拆分

数据库表拆分也是一个很常见的优化方式，这里的拆分分为横向（水平）拆分和纵向（垂直）拆分两种。

横向拆分，指的是把业务意义上的同一个表，拆分到不同的数据库节点或不同子表中，也就是说，这些节点中的表结构都是一样的，当然，存储的数据是不一样的。

我们前面介绍过的 Sharding 和 Partitioning 就是属于这一类型。那么，在查询或修改的时候，我们怎么知道数据在哪台机器上呢?

这就可以根据主键做 hash 映射或者范围映射来找到相应的节点，再继续进行操作了。Hash 映射本专栏已经介绍过了，而范围映射也很常见，比如用户的交易数据，3 月份的数据一个表，4 月份的数据一个表，这就是使用时间来做范围条件的一个例子。

再来说说垂直拆分，说的是把一个表拆成多个表，甚至拆到多个库中，这时的拆分是按照不同列来进行的，拆分出的表结构是完全不一样的，表和表之间通常使用外键关联。比如有这样一个文件表：

ID	TITLE	CREATION_DATE	DESC	CONTENT
1	秋日图	2019-11-05	一幅秋景。	...

各列分别为：唯一 ID、标题、创建日期、描述，以及以 BLOB 格式存储的文件具体内容（CONTENT）。

在这种情况下，如果执行全表扫描的查询，在文件非常大且记录数非常多的情况下，执行的过程会非常慢。这是因为一般的关系数据库是行数据库，数据是一行一行读取的（关于行数据库和列数据库的区别和原理，你可以回看 [第 25 讲]），磁盘一次读取一块，这一块内包含若干行。那么由于 CONTENT 列往往非常大，每次读取只能读到非常少的行，因此需

要读取很多次才能完成全表扫描。这种情况下，我们就可以做这样的拆表优化，把这个表拆成如下两个：

ID	TITLE	CREATION_DATE	DESC
1	秋日图	2019-11-05	一幅秋景。

ID	CONTENT
1	...

你看，第一个表一下子就瘦身下来了，这个表没有了那个最大的 BLOB 对象组成的列，在全表扫描进行查询的时候，就可以比较快地进行，而当找到了相应的 ID 并需要取出 CONTENT 的时候，再根据 ID 到第二个表里面去查询出具体需要的文件来。

你可能注意到了，这种优化的动机和前提有这样两个：

查询无法单纯地走索引完成，而需要进行全表扫描或部分表扫描；

某列或某几列占用空间巨大，而它们却并不需要参与关系查询。

当这两个条件都符合的时候，我们就可以考虑垂直拆分（纵向拆表）了。

4. 悲观锁变乐观锁

在关系数据库中，如果我们提到了“锁”，就意味着我们想让数据库某条数据的写操作变得“安全”，换言之，当我们需要根据某些条件而对数据进行更改的时候，不会受到并发的其它写操作的影响，而丧失正确性或完整性。在使用“锁”来实现的时候，有悲观锁和乐观锁两种实现。

所谓悲观锁（Pessimistic Locking），指的是数据库从“最坏”的角度考虑，所以它会先使用排它锁锁定相应的行，进行相应的读判断和写操作，一旦成功了，再提交变更并释放锁资源。**在使用悲观锁锁定相应行的过程中，如果有其它的写操作，是无法同时进行的，而只能等待。**且看这样一组基于 SQL 的例子，它用于将用户的积分更新：

```
select POINTS from USERS where ID=1001 for update;
... (省略，计算得出积分需要变更为 123)
update USERS set POINTS=123 where ID=1001;
```

我来简单做个说明：

第一行，使用 “for update” 这个技巧来锁定 ID 为 1001 的记录，查询出当前的积分；

第二行，业务逻辑得到积分需要如何变更，假如说得出的结果是积分需要变更为 123；

第三行，执行积分变更（这里假设事务在 update 之后是配置为自动提交的）；

这样一来，如果有两条请求同时想执行以上逻辑，那么第一条请求可以执行成功，而第二条会一直等在那里，直到第一条执行完成，它再去执行。这种方式就保证了锁机制的有效性。

下面再来说说乐观锁（Optimistic Locking）。它和悲观锁正好相反，这种情况假定“大多数”的操作发生锁冲突的概率较小，使用一个当前版本号，来表示当前记录的版本。**乐观锁方式下，不需要使用显示的加锁、提交这样的操作，但缺点是一旦发生冲突，整个过程要重来。**我们可以把前面的步骤变成下面这样：

```
select POINTS, VERSION from USERS where ID=1001;
... (省略，计算得出积分需要变更为 123)
update USERS set POINTS=123 and VERSION=VERSION+1 where ID=1001 and VERSION=1;
```

第一行，在读取积分的时候，也一并读取到了当前的版本号，假设版本号是 1；

第二行，业务逻辑得到积分需要如何变更，假如说得出的结果是积分需要变更为 123；

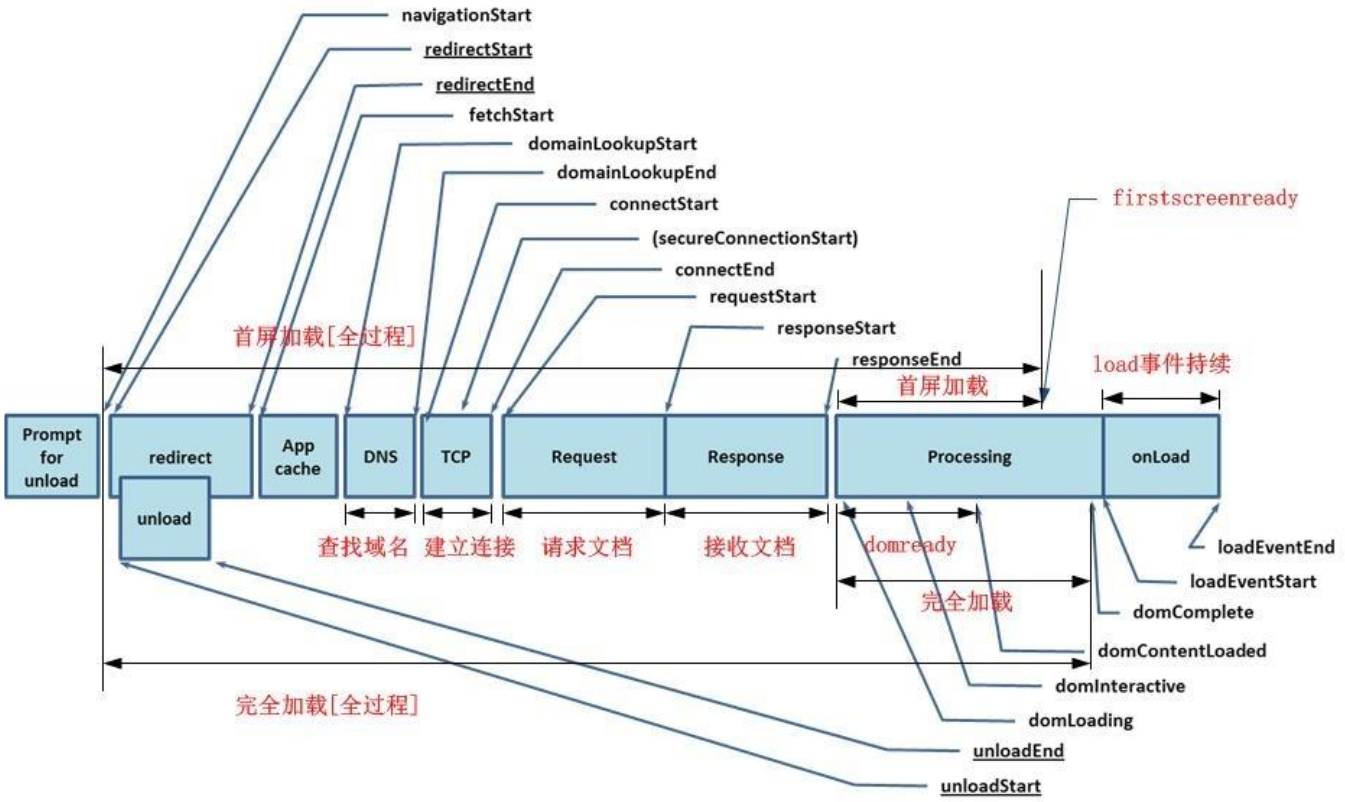
第三行，更新积分为 123 并自增版本号，但是条件是版本号为 1。之后需要检测这条 update 语句影响的代码行数：如果影响的行数为 1，说明更新成功，程序结束；如果影响的行数为 0，说明在第一行读取数据以后，记录发生了变更，需要重新执行整个过程。

还有许许多多其它的后端和持久层的优化通用技术，这里就不展开了。比如缓存的应用，互联网应用有句话叫做“缓存为王”，缓存的本质就是空间换时间，在 [第 21 讲] 中我们

曾经仔细聊过这部分的内容。再比如，应用之外，我们还经常需要从应用宿主和操作系统等角度来考虑，对于这部分，我在扩展阅读中我给出了一些材料供你阅读。

前端和网络层优化

当我们思考前端优化的时候，和后端一样，我们可以考虑连接、下载、解析、加载、渲染等等整个过程，先从大局上对整个时间消耗的分布有一个把握，下图来自这篇文章。



1. 减少请求次数

这是进行许多前端优化的共同目标之一，有不少全栈工程师日常在进行的实践，都遵循了这一条原则。

> 文本资源：CSS 压缩、JavaScript 压缩。有两个静态资源的后期处理方式我们经常结合起来使用，一个是压缩，一个是混淆。前者将多个文件压缩成一个，目的是减少大小，而更重要的是减少请求数；后者则是将变量使用无意义的名称替代，目的就是让产品代码“难懂”，减少代码实际意义的泄露。
>
> 图像资源：CSS Sprites 和 Inline Image。前者又叫做雪碧图、精灵图，是将网站所用的多张图片拼成一张，这样多张图片只需要下载一次，而使用 CSS 中的

background-image 和 background-position 在目标位置选择显示哪一张图片。后者则是干脆将二进制的图片使用 Base64 算法序列化成文本，直接嵌入在原始页面上。

缓存控制头部。即 Cache-Control 头，这部分我们在 [第 21 讲] 已经介绍过；还有 Etag 头，浏览器会把它发送给服务端用于鉴别目标资源是否发生了更改，如果没有更改，一个 304 响应会返回；以及 Expires 头，是服务端来指定过期时间的。

2. 减少渲染次数

CSS 或者 DOM 的变化都会引发渲染，而渲染是由单独的线程来进行的，这个过程会阻塞用户的操作。对于较大的页面、DOM 较多的页面，浏览器渲染会占用大量的 CPU 并带来明显的停顿时间。

渲染其实包括两种，一种是 reflow，就是页面元素的位置、间隔等等发生更改，这个工作是由 CPU 完成的；另一种叫做 repaint，基本上就是当颜色等发生变更的时候就需要重新绘制，这个工作是由 GPU 完成的。

因此，如果我们能够减少反复、多次，或是无意义的渲染，就可以在一定程度上为 Web 应用提速，特别是 reflow。那么对于这方面优化的其中一个思路，就是合并操作，即可以合并多个 DOM 操作为一次进行，或是合并单个 DOM 的多次操作为一次进行（React 或者 Vue.js 的 Virtual DOM 技术就借鉴了这种思路）。

3. 减少 JavaScript 阻塞

JavaScript 阻塞，本质上是由于 JavaScript 解释执行是单线程所造成的，阻塞期间浏览器拒绝响应用户的操作。同步的 Ajax 调用会引发阻塞（一直阻塞到响应返回），耗时的 JavaScript 代码执行也会引起阻塞。

我们通过将大的工作分裂成多次执行（可以通过每次具备一定间隔时间的回调来实现），每次执行后主动让出执行线程，这样每次就可以只阻塞一小会儿，以显著减少 JavaScript 阻塞对用户造成的影响；而对于一些独立的耗时操作，可以引入 Web Worker 分配单独运行的线程来完成。

4. 文本消息压缩

对于页面这样的文本内容，通过配置 Web 容器的 gzip 压缩，可以获得很好的压缩比，从而减小消息体的大小，减少消息传输的时间，代价是压缩和解压缩需要消耗 CPU 时间。

这种优化的本质是时间换空间，而前面介绍过的缓存本质则是空间换时间，二者比起来，刚好是相反的。可有趣的是，它们的目的却是一致的，都是为了缩短用户访问的时间，都是为了减少延迟。

对于前端的优化，技巧比较零散，可能有不少朋友会想起那最著名的 35 条军规（中文译文有不少，比如这里），这篇文章是如今很多前端技能优化学习首先要阅读的“老文章”了。

总结思考

今天我从产品和架构、后端和持久化，以及前端和网络层三个角度，结合一些具体的技巧，向你介绍了一些常见的网站性能优化方法。网站的性能优化是一个大课题，希望你在学完上一讲和这一讲之后，你能从前到后比较全面地去分析和思考问题。

下面我来提两个问题吧：

> 你在项目中是否做过性能优化的工作，能否介绍一下你都进行了哪些有效的优化实践呢？
>
> 文中介绍了数据库表拆分的两种方式，水平拆分和垂直拆分，它们都带来了显而易见的好处。可是，我们总是需要辩证地去看待一项技术，你能说出它们会带来哪些坏处吗？

扩展阅读

> 文中提到了 Amazon 对于 SOA 的实践，你可以阅读这篇文章简单了解一下这个故事。
>
> 本文主要讲的还是应用层面的调优，没有介绍虚拟机、容器等性能优化和操作系统的性能优化。如果你对它们感兴趣的话，我在这里推荐两个材料。关于 JVM 调优，可以参看 Java 应用性能调优实践这篇，而操作系统层面的性能优化，你可以从 Linux 性能调优指南这个材料中找感兴趣的阅读。
>
> 从 Webkit 内部渲染机制出发，谈网站渲染性能优化，这篇文章是从浏览器的机制这个角度来讲性能优化的，推荐一读。

文中介绍了 reflow 和 repaint，对于这方面的优化可以阅读 reflow 和 repaint 引发的性能问题这篇文章。

© 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。

上一篇 34 | 网站性能优化（上）

下一篇 36 | 全栈开发中的算法（上）

精选留言 (3)

写留言

_CountingStars
2019-11-29

如果使用了 HTTP/2 协议 “ 减少请求次数” 这条优化建议里把多文件压缩成一个可以不用做了，甚至做了会使用性能变慢。

展开

作者回复: 是的，你说的很对。实际上，HTTP/2 就是对于之前 1.1 版本“填坑”的问题，提高了性能，还解决了安全性等问题。

靠人品去赢

2019-11-29

老哥，你这文章真的稳。

展开


leslie

2019-11-29

数据库如果需要做垂直拆分不少都是早期的设计上的坑：一口气吃成胖子或者没想到后期的扩展性，水平拆分倒是一种常态-数据量过大造成的。垂直拆分其实最被动：涉及到大量的代码修改；相对比而言水平拆分的代价就小许多。

Javascripe的坑有时同样蛮大的：故而其实如何合理使用任何一个部件需要整体的思考，否则经常容易把某个部件用爆了。

展开

作者回复: 关于拆分，我想补充的是，拆分有时候也是不可避免的，本来设计就是一个循序渐进的过程，最开始确实要避免过度设计。到后来随着业务的发展再来拆分也未必是一件坏事。