分布式架构之系统拆分

* 系统拆分是一种特殊的重构

系统拆分是单体程序向分布式系统演变的关键一步，也是很重要的一步，拆分的好坏直接关系到未来系统的扩展性、性能、稳定性、安全性以及可维护性和可伸缩性等，拆分工作不难理解，但是如何正确拆分、有什么样的方法和原则拆分能帮助我们拆分得到一个我们理想中的系统：高可用、可扩展、可维护、可伸缩的分布式系统。

Martin Fowler的《重构 改善既有代码的设计》一书给重构的定义：在不改变代码外在行为的前提下，对代码做出修改，以改进程序的内部结构。所以拆分就是重构，是重构的一种方式。

主要从三个方面来分析系统拆分工作：拆分需求、拆分原则和拆分方法；

# 一、拆分需求

什么情况下系统才需要进行拆分? 当然并不是所有的系统都需要拆分，也并不是所有的系统都适合进行拆分，因为系统拆分是一个耗时、耗力又风险比较高的工作，在决定对系统进行拆分前，多问自己几次真的需要进行拆分吗？还有其它更好的方法吗？

拆分的需求主要来源于下面几个方面：

1、组织结构变化：从最初的一个团队成长并拆分为几个团队，团队按照业务线不同进行划分，为了减少各个业务线系统和代码间的关联和耦合性，几个团队不再可能共同向一个代码库中提交代码，必须对原有系统进行拆分，以减少工作间的干扰。

2、安全：这里所指的安全不是系统级别的安全，而是指代码或成果的安全，尤其是对于很多具有核心算法的系统，为了代码不被泄露，需要对相关系统进行模块化拆分，隔离核心功能，保护知识产权。

3、替换性：有些产品为了提供差异化的服务，需要产品具有可定制功能，根据用户的选择自由组合为一个完整的系统，比如一些模块，免费用户使用的功能与收费用户使用的功能肯定是不一样的，这就需要这些模块具有替换性，根据是免费用户还是收费用户使用不同的模块组装，这也需要对系统进行模块化拆分。

4、交付速度：单体程序最大的问题在于系统错综复杂，牵一发而动全身，也许一个小的改动就造成很多功能没办法正常工作，极大的降低了软件的交付速度，提高了测试成本，因为每次改动都需要大量的回归测试确保每个模块都能正确工作，因为我们不清楚改动会影响到什么，所以需要做大量重复工作。

5、技术需求:

1）单体程序由于技术栈固定，尤其的是比较庞大的系统，不能很方便的进行技术升级，或者说对引入新技术或框架等处于封闭状态；每种语言都有自己的特点，单体程序没有办法享受到其他语言带来的便利；映射到团队中，团队技术相对比较单一。

2）相比于基于业务的垂直拆分，基于技术的横向拆分也很重要，使用数据访问层可以很好的隐藏对数据库的直接访问、减少数据库连接数、增加数据缓存等，对于系统性能的提高具有非常明显的作用。

6、业务需求：由于业务上的某些特殊要求，比如对某个功能或模块的高可用性、高性能、可伸缩性等的要求，虽然我也可以将单体整体部署到分布式环境中实现高可用、高性能等，但是从系统维护的角度来考虑，每次改动都要重新部署所有节点，显然会增加很多潜在的风险和不稳定因素，所以我们不得不选择将这些个功能从系统中独立处理，独立部署和扩展。 如果系统有这样的需求最好还是拆分，越早拆分越好。

通过从团队、产品、交付、技术以及业务方面分析了系统拆分的需求，从更大的范围来看，拆分可以分为两种：纵向和横向。纵向拆分主要从业务角度进行，根据业务分割为不同的子系统；而横向拆分侧重于技术的分层，每个层级的技术侧重点不同，可以充分发挥和培养团队中每个人的技术特长。

# 二、拆分原则

有了拆分需求，下面就要对系统进行拆分了，面对动辄百万行代码的系统，我们从哪里下手呢？要注意哪些问题？等等一系列问题接踵而来，下面就介绍几个系统拆分的基本原则，可以帮组我们快速的开始对系统进行拆分：

1、业务优先：每个系统天然都会按业务功能分成多个模块，每个模块又包含许多业务相关的功能，在系统拆分时，我们就可以优先考虑按照业务边界进行切割，切割完成后再真的会每个模块进行拆解，循序渐进，逐渐迭代深入，最终完成系统的拆解。这个过程类似庖丁解牛，要找到关节之处下刀，方能事半功倍。

2、循序渐进：系统拆分过程中包含两个非常重要的工作：拆分和测试。二者缺一不可，并且二者是并行进行的，一定要边拆分边测试。每一步拆分完成都要保证系统功能是完整的，保证系统的测试是完整的。拆分有小步前进，如此以来犯错的几率就会减少，这就是循序渐进的重要性，不要累积了很多步骤以后才进行测试，由于代码已经改变了几个版本，你就会发现一些代码已很难恢复到原来的样子，这就造成一些Bug已经无法修复了，这时候要么更改程序逻辑，这当然需要你对业务非常的了解，但是一般情况下我是不敢下手。

所有拆分时一定要与测试并行，完成一步测试一步，发布一步，这样循环下去；我想大部分的人跟我一样会有这样的想法：我宁愿去重写一遍也不愿意修改老的代码，因为要理解原来的代码真的很难，即使有详尽的描述文档，你会保证不会遗漏一个细节吗？

3、兼顾技术：系统不能为了分布式而分布式，系统拆分的代价相当昂贵；当然如果有拆分的需要，我们也不能白白浪费这么好的学习机会：

重构：拆分过程不仅仅是业务梳理的过程，也是系统进行重构的过程。通过系统的重构我们可以使用一些法则或者模式让代码更清晰更有效率。

分层：拆分可以让系统分解为许多功能单一的系统，这些系统可以根据需要使用不同的技术和架构进行实现，可以让熟悉不同技术的人做不同的事，工作更高效，产品质量也可以提高。比如那些熟悉HTML, JavaScript等前端技术的人可以充实UI等方面的开发，而那些JAVA,C++方面的专家就可以把精力放在服务端程序的开发上，而那些熟悉数据库技术的人就可以话更多精力在数据库的优化上，术业有专攻，合适的人做合适的事。

4、可靠测试：“重构之前，首先检查自己是否有一套可靠的测试机制”，这是Martin Fowler在《重构》这本书中说到的，它同样对系统拆分有效。拆分是在对系统进行大手术，每一次的改动都要保证系统保持原来的行为不变。我非常喜欢自动化测试脚本，每当我提交完的我的修改时，会自动触发一系列的测试，使得我有足够的信心进行下一步的拆分或重构，不至于在错误的道路上越走越远，以至于找不到回来的路。测试与拆分如影随形，每一步都要有足够的测试。没有测试的拆分和重构我真的不敢想象结果会是什么样子。

# 三、拆分方法

前面介绍了那么多，都在为拆分做准备，如何进行拆分这才是重点所在。拆分的内容分为数据拆分和系统拆分。数据拆分主要指数据库的拆分和数据访问对象的拆分。在系统拆分过程中，一般情况下我会先进行数据拆分，难后再进行系统拆分，这种至下而上的操作我会有信心。

**数据拆分**

首先对数据库按照业务进行拆分，把业务相关表放到一个新的数据库中，但是保持上层系统结构不变（如下图）。



数据库的拆分相对于业务的拆分来说比较容易，对系统的影响也容易判断；拆分数据库时，对于一些公共数据即：既出现在A系统又出现在B系统或者其他系统的具有公共属性的数据表，可以分两步进行处理：

第一步、在A、B系统中保留两份相同的数据，注意这里指的是具有公共属性的表，不包括那些与业务相关的数据表，虽然业务数据也会被其他模块引用。拆分过程一定要把握好业务边界，只有定义清晰的业务边界才能拆分出具有清晰定义的模块，这一点非常重要。

第二步、处理公共数据，主要有两种方式：

1. 建立公共服务模块

将这部分数据连同业务拆分为独立的服务模块，通过API对外提供服务，这部分工作需要重头开始构建，所以在系统拆分或重构的过程中，不一定只是对原有功能进行修补和优化，有时候就需要实现一些原来不存在但又很基础的功能。如果将系统拆分工作看作是一个主干，那构建公共服务模块的工作就是一些分支，拆分的过程需要处理大量的公共功能，因为**拆分就是要将不相关的功能分开，把相关的功能合并的过程**。做完这些公共模块的工作还是要回到主干继续下面的工作。

1. 对于不需要抽象为公共服务的数据，可以保持一定的数据冗余，但是冗余的数据拷贝数不要太多，这会对数据同步提出很高的要求；根据数据的不同使用不同的数据同步机制确保各个模块中数据的一致性，至于要实现强一致性还是最终一致性要根据业务需求来定，不同的数据库在数据一致性方面的都有相关的实现，这里不再赘述。

第三步：建立数据服务层，在拆分后数据库基础之上建立一层数据服务层，对数据库访问进行封装和隐藏，提供通用的数据访问接口。

**系统拆分**

完成数据拆分后，就可以开始业务的拆分；拆分过程中需要坚持一个基本原则：高内聚、低耦合。拆分的实际工作就是解耦，下面主要介绍基于Proxy-Façade模式的系统拆分方法：

1）找出当前模块中对其他模块的关联调用或引用，在本模块中建立一层代理（Proxy），用于代理对其他模块的访问，所有的外部访问都需要通过这些代理对象实现。

2）找出所有外部模块对该模块内的类和对象的调用或引用，同样在该模块内建立一些Façade类，其他模块对该模块的访问都需要动过Façade类， Façade类隐藏了模块的内部实现细节。



为什么使用Proxy-Façade模式？Proxy-Façade模式的优点在哪里？

系统拆分最重要的目标就是要实现服务化，隐藏模块内部的实现细节，以接口形式提供服务和使用服务，实现高内聚低耦合的目的。Proxy-Façade模式可以很好的满足这一目标，Proxy将模块与外部模块的通信进行封装和隐藏，使用方不用关心对方的业务实现，通过Proxy就像直接与相应的模块在通信；Façade隐藏了模块的内部实现细节，对外提供友好的接口。Proxy-Façade模式不仅可以隐藏业务细节，还可以隐藏通讯实现，使用方无需关心服务是本地调用还是通过RPC的远程调用。Proxy-Façade模式不仅实现业务的高内聚而且也能够保证模块间的低耦合。

到此我们实现了系统模块的数据解耦和业务解耦，但是工作还没有结束， 第三：通过代理对象我们实现了模块间的松耦合调用，逐渐的我们会发现，代理对象越来越复杂，因为原来系统内的相互调用我们没有任何规划，想怎么实例化一个对象就怎么实例化一个对象，想要生成某种格式的数据就生成某种格式的数据，因为实在系统内部所以非常的随意。 是时候整理这些调用了，根据系统特点建立多层次的调用关系，构建基础数据接口，然后在此之上进行组合为复杂的数据结构，不要把所有功能或方法都实现在一个代理类中。这件事情说起来很简单，但是实际操作过程中会有很多的问题，比如有的调用多为查询的、有的调用多为更新的等等，需要针对不同的情况因地制宜，没有一个统一的模板可以提供。接口的设计也是一门大学问。 经过上面三个步骤的处理，我们就实现了使用代理对象的模块间的解耦，但是实际目前的系统还没有完全解耦，因为我们还有代理对象的存在，代理对象会出现在使用者的代码中，并且代理对象只是将原来的调用集中管理，代理对象内部还是与服务端程序紧密结合，所以模块间还是存在耦合性，比如对原来的接口进行修改，增加一个参数，结果就是我需要修改模块内部的服务类，修改代理类，修改使用者的代码，如何消除这种耦合呢？其实这就服务化要解决的问题，我们将代理对象作为facade对象，对外提供一致接口，隐藏内部实现细节；可以采用服务注册和发现的方式实现模块间的解耦，模块作为服务的提供者对外提供服务，开放服务接口，又作为服务使用者，使用其他模块的服务和组合成更复杂的服务。