三大架构设计原则

可是一旦涉及“选择”，就很容易让架构师陷入两难的境地，例如：

如果选了最先进的技术后出了问题怎么办？如果选了目前最熟悉的技术，后续技术演进怎么办？

是要选择 Google 的 Angular 的方案来做，还是选择 Facebook 的 React 来做？Angular 看起来更强大，但 React 看起来更灵活？

是要选 MySQL 还是 MongoDB？团队对 MySQL 很熟悉，但是 MongoDB 更加适合业务场景？

做一个电商网站，是否简单地照搬淘宝就可以了？

没有一套通用的规范来指导架构师进行架构设计，**更多是依赖架构师的经验和直觉**

共性的原则：**合适原则、简单原则、演化原则**

#一、合适原则

合适原则宣言：“合适优于业界领先”。最后可能都以失败告终！

1. 将军难打无兵之仗：**没那么多人，却想干那么多活，是失败的第一个主要原因。**
2. 罗马不是一天建成的：**没有那么多积累，却想一步登天，是失败的第二个主要原因。**
3. 冰山下面才是关键：GFS 为何在 Google 诞生，而不是在 Microsoft 诞生？我认为 Google 有那么庞大的数据是一个主要的因素，而不是因为 Google 的工程师比 Microsoft 的工程师更加聪明。

没有那么卓越的业务场景，却幻想灵光一闪成为天才，是失败的第三个主要原因。

没有腾讯那么多的人，没有 QQ 那样海量用户的积累，没有 QQ 那样的业务。

#二、简单原则

简单原则宣言：“简单优于复杂”。

例如设计一个主备方案，如果你用**心跳**来实现，可能大家都认为这太简单了。但如果你引入 **ZooKeeper** 来做主备决策，可能很多人会认为这个方案更加“高大上”一些，毕竟 ZooKeeper 使用的是 **ZAB** 协议，而 ZAB 协议本身就很复杂。其实，真正理解 ZAB 协议的人很少（我也不懂），但并不妨碍我们都知道 ZAB 协议很优秀。

“复杂”在软件领域，却恰恰相反，代表的是“问题”。体现在两个方面：

#1. 结构的复杂性

结构复杂的系统几乎毫无例外具备两个特点：

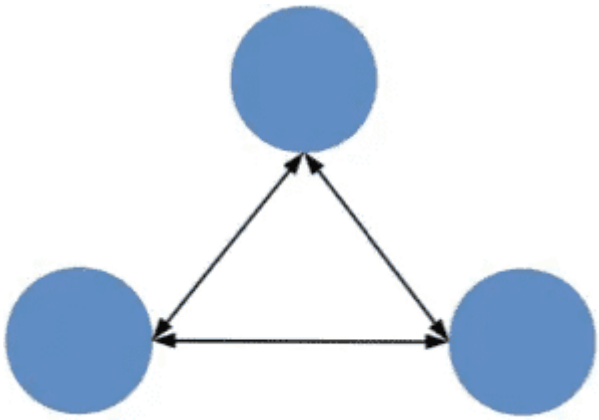
组成复杂系统的**组件数量更多**；

同时这些组件之间的**关系也更加复杂**。

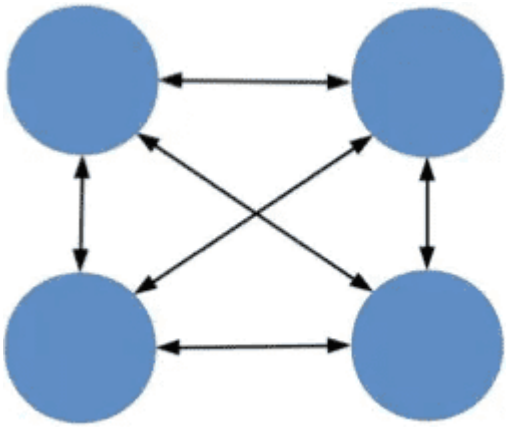
我以图形的方式来说明复杂性：

2 个组件组成的系统：

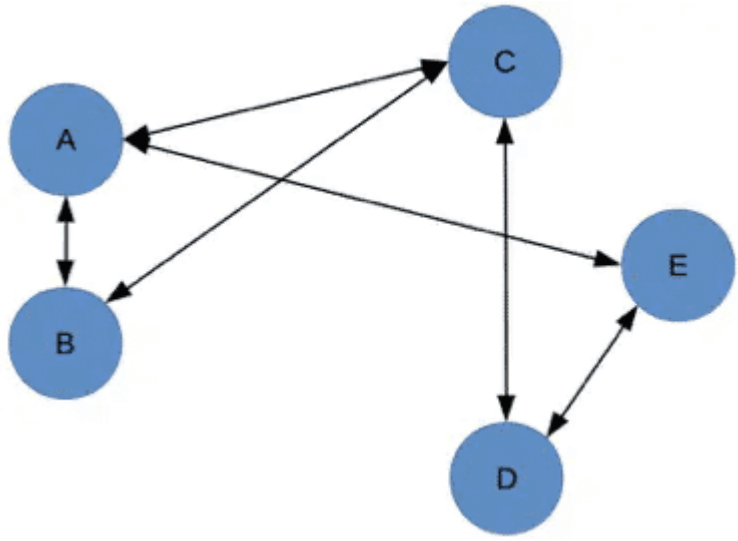
3 个组件组成的系统：



4 个组件组成的系统：



5 个组件组成的系统：



结构上的复杂性存在的第一个问题是，**组件越多，就越有可能其中某个组件出现故障**，从而导致系统故障。这个概率可以算出来，假设组件的故障率是 10%（有 10% 的时间不可用），那么有 3 个组件的系统可用性是（1-10%）×（1-10%）×（1-10%）= 72.9%，有 5 个组件的系统可用性是（1-10%）×（1-10%）×（1-10%）×（1-10%）×（1-10%）=59%，两者的可用性相差 13%。

结构上的复杂性存在的第二个问题是，**某个组件改动，会影响关联的所有组件**，这些被影响的组件同样会继续递归影响更多的组件。还以上面图中 5 个组件组成的系统为例，组件 A 修改或者异常时，会影响组件 B/C/E，D 又会影响 E。这个问题会影响整个系统的开发效率，因为一旦变更涉及外部系统，需要协调各方统一进行方案评估、资源协调、上线配合。

结构上的复杂性存在的第三个问题是，**定位一个复杂系统中的问题总是比简单系统更加困难**。首先是组件多，每个组件都有嫌疑，因此要逐一排查；其次组件间的关系复杂，有可能**表现故障的组件并不是真正问题的根源**。

#2. 逻辑的复杂性

意识到结构的复杂性后，我们的第一反应可能就是“**降低组件数量”，毕竟组件数量越少，系统结构越简**。最简单的结构当然就是整个系统只有一个组件，即系统本身，所有的功能和逻辑都在这一个组件中实现。

不幸的是，这样做是行不通的，原因在于除了结构的复杂性，还有逻辑的复杂性，即如果某个组件的逻辑太复杂，一样会带来各种问题。

逻辑复杂的组件，一个典型特征就是**单个组件承担了太多的功能**。以电商业务为例，常见的功能有：商品管理、商品搜索、商品展示、订单管理、用户管理、支付、发货、客服……把这些功能全部在一个组件中实现，就是典型的逻辑复杂性。

将这些功能全部在单一的组件中实现，可以想象一下这个恐怖的场景：

系统会很庞大，可能是上百万、上千万的代码规模，“clone”一次代码要 30 分钟。

几十、上百人维护这一套代码，某个“菜鸟”不小心改了一行代码，导致整站崩溃。

需求像雪片般飞来，为了应对，开几十个代码分支，然后各种分支合并、各种分支覆盖。

产品、研发、测试、项目管理不停地开会讨论版本计划，协调资源，解决冲突。

版本太多，每天都要上线几十个版本，系统每隔 1 个小时重启一次。

复杂的电路就意味更强大的功能，而复杂的架构却有很多问题呢？根本原因在于电路一旦设计好后进入生产，就不会再变，复杂性只是在设计时带来影响；而一个软件系统在投入使用后，后续还有源源不断的需求要实现，因此要不断地修改系统，复杂性在整个系统生命周期中都有很大影响。

功能复杂的组件，另外一个典型特征就是采用了**复杂的算法**。复杂算法导致的问题主要是难以理解，进而导致难以实现、难以修改，并且出了问题**难以快速解决**。

以 ZooKeeper 为例，ZooKeeper 本身的功能主要就是选举，为了实现分布式下的选举，采用了 **ZAB** 协议，所以 ZooKeeper **功能虽然相对简单，但系统实现却比较复**杂。相比之下，etcd 就要简单一些，因为 etcd 采用的是 Raft 算法，相比 ZAB 协议，Raft 算法更加容易理解，更加容易实现。

简单的方案和复杂的方案都可以满足需求，最好选择简单的方案。

#三、演化原则

演化原则宣言：“演化优于一步到位”。

软件架构从字面意思理解和建筑结构非常类似，事实上“架构”这个词就是建筑领域的专业名词，维基百科对“软件架构”的定义中有一段话描述了这种相似性：

从和目的、主题、材料和结构的联系上来说，软件架构可以和建筑物的架构相比拟。

对于软件来说，变化才是主题。根据业务的发展而不断变化，很多预测和分析都是不靠谱的。

考虑到软件架构需要根据业务发展不断变化这个本质特点，**软件架构设计其实更加类似于大自然“设计”一个生物，通过演化让生物适应环境，逐步变得更加强大：**

1、**生物要适应当时的环境**。

2、**有利的基因传递下去，将不利的基因剔除或者修复**。

3、**无法调整就被自然淘汰**；新的生物会保留一部分**原来被淘汰生物的基因**。

#软件架构设计同样是类似的过程：

1、设计出来的架构要**满足当时的业务需要**。

2、架构要**不断**地在实际应用过程中**迭代**，保留优秀的设计，修复有缺陷的设计，改正错误的设计，去掉无用的设计，使得架构逐渐完善。

3、**当业务发生变化时**，架构要**扩展、重构，甚至重写**；代码也许会重写，但有价值的经验、教训、逻辑、设计等（类似生物体内的基因）却可以在新架构中延续。

不要贪大求全，或者盲目照搬大公司的做法。应该认真分析当前业务的特点，明确业务面临的主要问题，设计合理的架构，在运行过程中不断完善架构，不断随着业务演化架构。

#小结

这三条架构设计原则是否每次都要全部遵循？是否有优先级？谈谈你的理解，并说说为什么。

#评论：

合适优于先进>演化优于一步到位>简单优于复杂

适应当前需要是首位的，连当前需求都满足不了谈不到其他。架构整体发展是要**不断演进**的，在这个大前提下，尽量追求简单，但也有该复杂的时候，就要复杂，比如生物从单细胞一直演化到如今，复杂是避免不了的，

合适原则：确定了复杂度之后，能承受其包括性能，可用性，可拓展性，成本，安全方面的最小代价解(**简单原则**)，而演化原则是对上述系统的**迭代优化**。

#参考文章

- https://www.jianshu.com/p/d77b29ddd12b