

## 08 | 架构设计三原则

2018-05-15 李运华 来自北京

《从0开始学架构》



前面几期专栏，我跟你系统的聊了架构设计的主要目的是为了解决软件系统复杂度带来的问题，并分析了复杂度的来源。从今天开始，我会分两期讲讲[架构设计的 3 个原则](#)，以及架构设计原则的案例。

成为架构师是每个程序员的梦想，但并不意味着把编程做好就能够自然而然地成为一个架构师，优秀程序员和架构师之间还有一个明显的鸿沟需要跨越，这个鸿沟就是“**不确定性**”。

对于编程来说，本质上是不能存在不确定的，对于同样一段代码，不管是谁写的，不管什么时候执行，执行的结果应该都是确定的（注意：“确定的”并不等于“正确的”，有 bug 也是确定的）。而对于架构设计来说，本质上是不确定的，同样的一个系统，A 公司和 B 公司做出来的架构可能差异很大，但最后都能正常运转；同样一个方案，A 设计师认为应该这样做，B 设计师认为应该那样做，看起来好像都有道理……相比编程来说，架构设计并没有像编程语言那样的语法来进行约束，更多的时候是面对多种可能性时进行选择。

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：

是要选择业界最先进的技术，还是选择团队目前最熟悉的技术？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？

淘宝的电商网站架构很完善，我们新做一个电商网站，是否简单地照搬淘宝就可以了？

还有很多类似的问题和困惑，关键原因在于架构设计领域并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，因此架构设计有时候也会被看作一项比较神秘的工作。

业务千变万化，技术层出不穷，设计理念也是百花齐放，看起来似乎很难有一套通用的规范来适用所有的架构设计场景。但是在研究了架构设计的发展历史、多个公司的架构发展过程（QQ、淘宝、Facebook 等）、众多的互联网公司架构设计后，我发现有几个共性的原则隐含其中，这就是：**合适原则、简单原则、演化原则**，架构设计时遵循这几个原则，有助于你做出最好的选择。

## 合适原则

**合适原则宣言：“合适优于业界领先”。**

优秀的技术人员都有很强的技术情结，当他们做方案或者架构时，总想不断地挑战自己，想达到甚至优于业界领先水平是其中一个典型表现，因为这样才能够展现自己的优秀，才能在年终 KPI 绩效总结里面骄傲地写上“设计了 XX 方案，达到了和 Google 相同的技术水平”“XX 方案的性能测试结果大大优于阿里集团的 YY 方案”。

但现实是，大部分这样想和这样做的架构，最后可能都以失败告终！我在互联网行业见过“亿级用户平台”的失败案例，2011 年的时候，某个几个人规模的业务团队，雄心勃勃的提出要

做一个和腾讯 QQ（那时候微信还没起来）一拼高下的“亿级用户平台”，最后结果当然是不出所料的失败了。

为什么会这样呢？

再好的梦想，也需要脚踏实地实现！这里的“脚踏实地”主要体现在下面几个方面。

### 1. 将军难打无兵之仗

大公司的分工比较细，一个小系统可能就是一个小组负责，比如说某个通信大厂，做一个 OM 管理系统就有十几个人，阿里的中间件团队有几十个人，而大部分公司，整个研发团队可能就 100 多人，某个业务团队可能就十几个人。十几个人的团队，想做几十个人的团队的事情，而且还要做得更好，不能说绝对不可能，但难度是可想而知的。

**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

业界领先的很多方案，其实并不是一堆天才某个时期灵机一动，然后加班加点就做出来的，而是经过几年时间的发展才逐步完善和初具规模的。阿里中间件团队 2008 年成立，发展到现在已经有十年了。我们只知道他们抗住了多少次“双 11”，做了多少优秀的系统，但经历了什么样的挑战、踩了什么样的坑，只有他们自己知道！这些挑战和踩坑，都是架构设计非常关键的促进因素，单纯靠拍脑袋或者头脑风暴，是不可能和真正实战相比的。

**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

可能有人认为，业界领先的方案都是天才创造出来的，所以自己也要造一个业界领先的方案，以此来证明自己也是天才。确实有这样的天才，但更多的时候，业界领先的方案其实都是“逼”出来的！简单来说，“业务”发展到一定阶段，量变导致了质变，出现了新的问题，已有的方式已经不能应对这些问题，需要用一种新的方案来解决，通过创新和尝试，才有了业界

领先的方案。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。**

回到我前面提到的“亿级用户平台”失败的例子，分析一下原因。没有腾讯那么多的人（当然钱差得更多），没有 QQ 那样海量用户的积累，没有 QQ 那样的业务，这个项目失败其实是在一开始就注定的。注意这里的失败不是说系统做不出来，而是系统没有按照最初的目标来实现，上面提到的 3 个失败原因也全占了。

所以，真正优秀的架构都是在企业当前人力、条件、业务等各种约束下设计出来的，能够合理地将资源整合在一起并发挥出最大功效，并且能够快速落地。这也是很多 BAT 出来的架构师到了小公司或者创业团队反而做不出成绩的原因，因为没有了大公司的平台、资源、积累，只是生搬硬套大公司的做法，失败的概率非常高。

## 简单原则

**简单原则宣言：“简单优于复杂”。**

软件架构设计是一门技术活。所谓技术活，从历史上看，无论是瑞士的钟表，还是瓦特的蒸汽机；无论是莱特兄弟发明的飞机，还是摩托罗拉发明的手机，无一不是越来越精细、越来越复杂。因此当我们进行架构设计时，会自然而然地想把架构做精美、做复杂，这样才能体现我们的技术实力，也才能够将架构做成一件艺术品。

由于软件架构和建筑架构表面上的相似性，我们也会潜意识地将对建筑的审美观点移植到软件架构上面。我们惊叹于长城的宏伟、泰姬陵的精美、悉尼歌剧院的艺术感、迪拜帆船酒店的豪华感，因此，对于我们自己亲手打造的软件架构，我们也希望它宏伟、精美、艺术、豪华……总之就是不能寒酸、不能简单。

团队的压力有时也会有意无意地促进我们走向复杂的方向，因为大部分人在评价一个方案水平高低的时候，复杂性是其中一个重要的参考指标。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。

刚才我聊的这些原因，会在潜意识层面促使初出茅庐的架构师，不自觉地追求架构的复杂性。然而，“复杂”在制造领域代表先进，在建筑领域代表领先，但在软件领域，却恰恰相反，代表的是“问题”。

软件领域的复杂性体现在两个方面：

### 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；

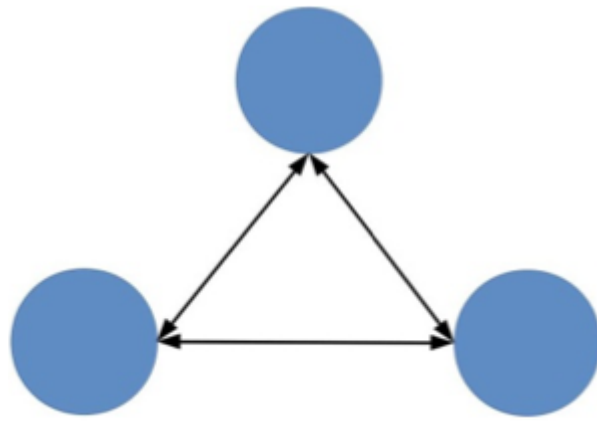
同时这些组件之间的关系也更加复杂。

我以图形的方式来说明复杂性：

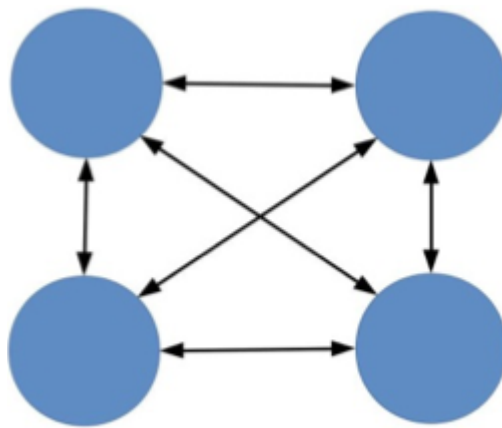
2 个组件组成的系统：



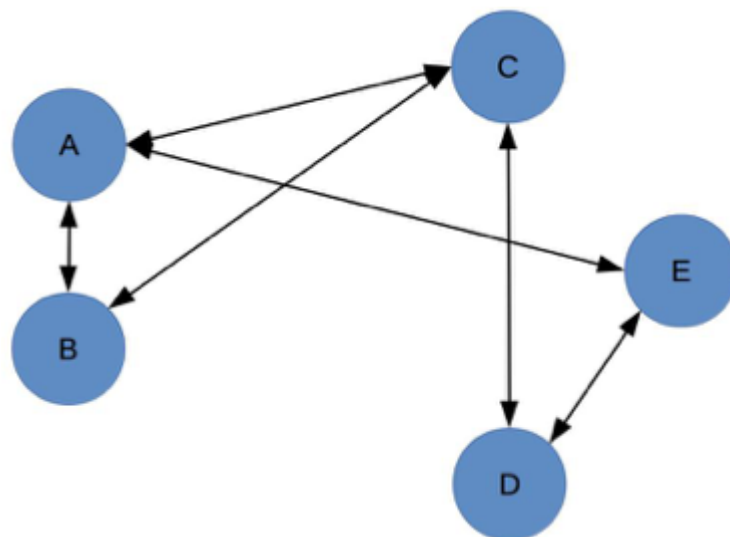
3 个组件组成的系统：



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  $(1-10\%) \times (1-10\%) \times (1-10\%) = 72.9\%$ ，有 5 个组件的系统可用性是  $(1-10\%) \times (1-10\%) \times (1-10\%) \times (1-10\%) \times (1-10\%) = 59\%$ ，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。

逻辑复杂几乎会导致软件工程的每个环节都有问题，假设现在淘宝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。

线上运行出现故障，几十个人扑上去定位和处理，一间小黑屋都装不下所有人，整个办公区闹翻天。

.....

不用多说，肯定谁都无法忍受这样的场景。

但是，为什么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。

综合前面的分析，我们可以看到，无论是结构的复杂性，还是逻辑的复杂性，都会存在各种问题，所以架构设计时如果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。

《UNIX 编程艺术》总结的 KISS (Keep It Simple, Stupid!) 原则一样适应于架构设计。

## 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。

例如，软件架构描述的是一个软件系统的结构，包括各个模块，以及这些模块的关系；建筑架构描述的是一幢建筑的结构，包括各个部件，以及这些部件如何有机地组成成一幢完美的建筑。

然而，字面意思上的相似性却掩盖了一个本质上的差异：建筑一旦完成（甚至一旦开建）就不可再变，而软件却需要根据业务的发展不断地变化！

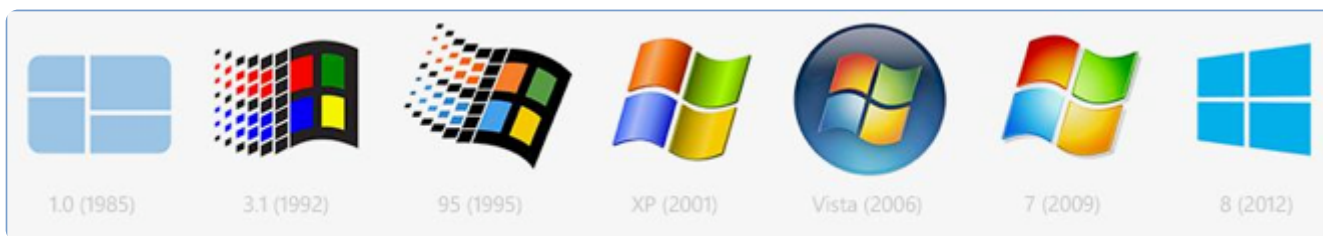
古埃及的吉萨大金字塔，4000 多年前完成的，到现在还是当初的架构。

中国的明长城，600 多年前完成的，现在保存下来的长城还是当年的结构。

美国白宫，1800 年建成，200 年来进行了几次扩展，但整体结构并无变化，只是在旁边的空地扩建或者改造内部的布局。











对比一下，我们来看看软件架构。

Windows 系统的发展历史：



如果对比 Windows 8 的架构和 Windows 1.0 的架构，就会发现它们其实是两个不同的系统了！

Android 的发展历史：

 <p>Android 1.6</p> <p>Donut</p>	 <p>Android 2.0</p> <p>Eclair</p>	 <p>Android 2.2</p> <p>Froyo</p>	 <p>Android 2.3</p> <p>Gingerbread</p>	 <p>Android 3.0</p> <p>Honeycomb</p>
 <p>Android 4.0</p> <p>Ice Cream Sandwich</p>	 <p>Android 4.1</p> <p>Jelly Bean</p>	 <p>Android 4.4</p> <p>KitKat</p>	 <p>Android 5.0</p> <p>Lollipop</p>	 <p>Android 6.0</p> <p>Marshmallow</p>

(<http://www.dappworld.com/wp-content/uploads/2015/09/Android-History-Dappworld.jpg>)

同样，Android 6.0 和 Android 1.6 的差异也很大。

**对于建筑来说，永恒是主题；而对于软件来说，变化才是主题。**软件架构需要根据业务的发展而不断变化。设计 Windows 和 Android 的人都是顶尖的天才，即便如此，他们也不可能在 1985 年设计出 Windows 8，不可能在 2009 年设计出 Android 6.0。

如果没有把握“软件架构需要根据业务发展不断变化”这个本质，在做架构设计的时候就很容易陷入一个误区：试图一步到位设计一个软件架构，期望不管业务如何变化，架构都稳如磐石。

为了实现这样的目标，要么照搬业界大公司公开发表的方案；要么投入庞大的资源和时间来做各种各样的预测、分析、设计。无论哪种做法，后果都很明显：投入巨大，落地遥遥无期。更让人沮丧的是，就算跌跌撞撞拼死拼活终于落地，却发现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：**

首先，生物要适应当时的环境。

其次，生物需要不断地繁殖，将有利的基因传递下去，将不利的基因剔除或者修复。

第三，当环境变化时，生物要能够快速改变以适应环境变化；如果生物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：

首先，设计出来的架构要满足当时的业务需要。

其次，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。

第三，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。

架构师在进行架构设计时需要牢记这个原则，时刻提醒自己不要贪大求全，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，快速落地以满足业务需要，然后在运行过程中不断完善架构，不断随着业务演化架构。

即使是大公司的团队，在设计一个新系统的架构时，也需要遵循演化的原则，而不应该认为团队人员多、资源多，不管什么系统上来就要一步到位，因为业务的发展和变化是很快的，不管多牛的团队，也不可能完美预测所有的业务发展和变化路径。

## 小结

今天我为你讲了面对“不确定性”时架构设计的三原则，分别是合适优于业界领先、简单优于复杂、演化优于一步到位，希望对你有所帮助。

这就是今天的全部内容，留一道思考题给你吧。我讲的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。

欢迎你把答案写到留言区，和我一起讨论。相信经过深度思考的回答，也会让你对知识的理解更加深刻。（编辑乱入：精彩的留言有机会获得丰厚福利哦！）

## 精选留言 (155)



公号-技术夜未眠

2018-05-15

今日得到

架构即决策。架构需要面向业务需求，并在各种资源（人、财、物、时、事）约束条件下去做权衡、取舍。而决策就会存在不确定性。采用一些高屋建瓴的设计原则有助于去消除不确定，去逼近解决问题的最优解。

### 1 合适原则

架构无优劣，但存合适性。“汝之蜜糖，吾之砒霜”；架构一定要匹配企业所在的业务阶段；不要面向简历去设计架构，高大上的架构不等于适用；削足适履与打肿充胖都不符合合适原则；所谓合适，一定要匹配业务所处阶段，能够合理地将资源整合在一起并发挥出最大功效，并能够快速落地。

### 2 简单原则

“我没有时间写一封短信，所以只好写一封长信”。其实，简单比复杂更加困难。面对系统结构、业务逻辑和复杂性，我们可以编写出复杂的系统，但在软件领域，复杂代表的是“问题”。架构设计时如果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。但是，事实上，当软件系统变得太复杂后，就会有人换一个思路进行重构、升级，将它重新变得简单，这也是软件开发的大趋势。简单原则是一个朴素且伟大的原则，Google的MapReduce系统就采用了分而治之的思想，而背后就是将复杂问题转化为简单问题的典型案例。

### 3 演化原则

大到人类社会、自然生物，小到一个细胞，似乎都遵循这一普世原则，软件架构也不例外。业务在发展、技术在创新、外部环境在变化，这一切都是在告诫架构师不要贪大求全，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，快速落地以满足业务需要，然后在运行过程中不断完善架构，不断随着业务演化架构。怀胎需要十月，早一月或晚一月都很危险。



@漆~心endless

2018-05-17

架构设计三原则；  
合适原则最适合；  
简单原则不简单；  
演化原则需推进；  
如若脱离三原则；  
老板生气你苦逼。

作者回复：确认过眼神，你是油菜花的人！！😂😂😂

共 2 条评论 >

👍 70



石同享

2018-05-15

合适原则是不是可理解为：确定了一定的系统复杂度之后，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。这样和之前的内容都可以联系起来了。

作者回复：是的，我的思路就是这样的，三个原则是一体的，三个原则与架构设计的目的也是一脉相承的

共 2 条评论 >

👍 43



桌小洛

2018-08-11

总结的很好，架构设计原则：

合适原则：有多大的脚（复杂度），穿多大码的鞋。BAT或者业界领先的架构对很多团队和公司来说都是大码的鞋而已，穿上不合适的鞋，团队必定步履蹒跚，很难走的很远，还有可能摔倒（项目失败）。

但是如果能根据自己的业务需求，对BAT或者业界领先的架构进行仔细调研，进行删减，重新取其一部分组合成适合自己的架构，也是非常好的方法。

简单原则：我感觉改成简洁原则更合适，简单+整洁。360行皆是艺术，架构也是一门艺术。一个复杂的系统，如果能用一个简洁的架构来实现，完全相当于一个艺术品。相反，如果一个普通的系统，反而被设计成了一个错综复杂的复杂架构，相当于做了一个糟粕品。

演化原则：通过 简化设计 + 重构 来保证架构的与时俱进。不要前期就对架构进行过度设计，毕竟无论你前期怎么设计，总会有你意想不到的变化产生，唯一不变的就是变化。等你的脚长大了，再去穿大码的鞋。

作者回复：你的解读很形象👍👍



👍 34



**narry**

2018-05-15

个人感觉合适原则是最重要的，它决定了对简单原则和演化原则的判断，没有以合适为基础，很难判断简单是否能满足业务的需求，演化的起点在哪里



👍 24



**何磊**

2018-05-15

三大原则：合适原则，简单原则，演进原则。

最重要的莫过于合适原则，如果不合适，无意欲削足适履，团队感觉难受，业务不稳定等，因此架构设计的第一原则一定是合适，合适当前的业务，合适当前的团队，合适当前的成本（时间与资本）

其二简单原则，这是一个相对原则，是在合适的基础上进行选择最简单方案，绝不能孤立，并且简单是自己业务的对比，比如：当前淘宝的架构每次迭代，他们选择一个简单方案，但他们的简单不意味着我们的简单。

关于演进原则，系统一定变化的、生长的，但是他们的起跑点肯定不同，比如大公司造的系统都是富二代，他会从微服务开始演进，十个人的小团队会从单体应用演进。

对于上面三个原则，演进原则其实我觉得考不考虑，他都存在，只要你的业务继续，不过好的架构有助于可扩展性，让后续演进更丝滑般流畅，不好的架构到了某个阶段只能重来。

作者回复：恰恰“演进原则”很多人不知道，总想一步到位，过度设计

共 2 条评论 >

👍 17



**Loy**

2018-05-18

发现这三个选择放在爱情上，也完全没毛病

作者回复: 爱情怎么演化? 😊😊

共 5 条评论 >

👍 15



**SHLOMA**

2018-05-21

题外话, 如何说服领导, 合适大于先进 😊

作者回复: 人手不够, 工期太长, 成本太高, 没有合适的人才..... 😊

共 3 条评论 >

👍 13



**查理**

2018-05-17

演进很重要, 很多人都喜欢过度设计, 简单的东西搞复杂。其实以后系统怎样变化很多过度的预测都不准, 还不如让系统在一开始保持精简一点, 根据需求慢慢演进

作者回复: 赞同, 预测太长没有意义, 也预测不准

共 4 条评论 >

👍 13



**Geek\_5420ac**

2021-01-11

深有体会, 刚来公司发现现有的项目写的跟shi一样, 一直跟组长提出重构, 虽然拖了大半年但还是重构了, 到了自己负责重构的搭建时, 又发现其实之前框架的很多逻辑其实是优于我设计的, 最后设计出来的发现其实也不是自己想象中的那么完美, 因为设计的同时既要保证原有功能不能丢失, 而且还有考虑到后续的拓展, 这两者在我看来一般是矛盾的, 很难在两者中间取舍, 说到底还是自己接触的太少了, 加油干饭人!

作者回复: 你刚来就觉得项目要重构, 有点操之过急了, 很多时候看起来不合理其实很可能是你不熟悉而已 😊

共 3 条评论 >

👍 10



**Forrest Li**

2018-07-04

面对不确定性, 架构师始终要做出一个选择, 而三个原则遵循着解决问题的思路, 提供了选择



的依据。首先是合适，能够解决问题，其次是从合适中挑选简单的、能hold住的方案，最后，不要指望一个方案能解决所有问题，总会有弊端、不足，在碰到未来无法预料的情况时再做调整就是。变是永远不变的。

作者回复: 仅此一家，别无分店的三原则 😊



👍 7



**钰澜一付晓岩**

2018-05-17

我们之前花了多年时间进行了一次全范围的架构升级，几乎重做了所有主力系统，这个过程中，我的体会其实跟三原则非常接近，尤其是演化原则，大公司的复杂系统永远不可能一步到位，甚至较长的工期本就会导致原有设计在最终实现时就已经落后，复杂系统的架构只能是演化迭代的，这其中即有技术因素也有业务因素，有工艺的复杂也有人的复杂；简单原则，实际上是对单一职责的追求，但是对大型系统而言，单一职责也会带来通讯的复杂，调用的复杂，大型系统清晰的逻辑分层并不容易实现，如果涉及多方协同施工，单一职责更难贯彻，尽管大家都认可，但是实现起来并不容易；合适原则非常正确，不过可惜的是，它经常不是甲方的原则。这三原则对于架构设计而言非常重要，但是保证它发挥作用的是架构师能够获得充分的保障履行职责。

作者回复: 架构师再牛气，首先得要有领导的信任 😊



👍 6



**十里坡剑神**

2021-01-03

非互联网行业的程序猿路过。架构设计得是否复杂，是否高大上，关系到能申请到多少项目预算甚至能否立项

作者回复: 哈哈，之前有朋友也说过，如果是做外包项目，架构设计无法遵守三原则，必须要高大上；如果在某些事业单位做项目，架构设计的越复杂越好，拿到的经费多。

共 3 条评论 >

👍 5



**t7ink**

2019-02-28

我的理解是：  
在合适原则的基础上保持可演化的最简单的架构。





4

**木木**

2018-05-15

合适优先级最高吧，反正一切脱离业务需求的架构设计都是耍流氓



4

**天外来客~**

2020-10-23

我们公司是做产品的，系统微服务化。由于业务的特殊性，客户要求不能上云，不能外网，必须部署在自己的服务器。问题来了，客户的服务器通常就一两台，配置也有限，而面对十几二十个微服务，性能问题尤为突出。这种场景，该如何优化设计

作者回复: 2台服务器还上啥微服务啊？大一统它不香嘛??????

共 2 条评论 &gt;



3

**易燃易爆炸**

2018-06-26

作者回复 架构师再牛气，首先得要有领导的信任😁 201...

极客时间版权所有: <https://time.geekbang.org/column/article/7071?device=geekTime.ios>

哈哈，看到作者这句话，深感这才是第一重要的。个人感觉合适最重要，本着实事求是的态度，解决问题为导向，不求高大上，但是最各种体系都要熟悉，知道优缺点，了解具体试用场景，可能这是最好的状态吧。

作者回复: 不能说第一重要，但确实挺重要，第一重要的是架构师要真懂技术😁



3

**lufeng**

2018-06-06

我们既要仰望星空（演进），又要脚踏实地（合适，简单），so 演进是一种sense，这个要求对业务有深入的理解和对技术趋势有独到的见解，合适和简单是两个维度，就是功能和质量要求达标的情况下尽量简单，因为越简单的东西越稳定、性能，可用性和可扩展性相当于复杂系统要好。

作者回复: 油菜花😄😄先脚踏实地，再仰望星空



👍 3



**Tony**

2018-05-30

第一次了解架构三原则

初出茅庐时候有次和老大1-1面谈

直接甩给老大几个问题：为什么不用spring?为啥不用ibatis? 为啥还在用SP?

看完这篇，现在回头想想老大给我的回复，姜还是老的辣.....

当时也是拍脑袋想到为何不用业界流行的框架重构我们的系统，其实没有最好的框架，只有合适的框架，只要能够简单的解决系统面临的业务复杂度，架构组优先会选择公司现有的框架另外提到架构演绎，我的看法应该是偿还技术债务，公司目前推行的是敏捷开发，敏捷开发的價值就是快速可靠的持续交付，往往team实践时候优先考虑如何在现有框架基础上快速实现业务需求

长期以往一个组件的功能就非常复杂了，功能上容易牵一发而动全身，所以这时候不得不让team停顿下，解决现有的技术债务，从而让复杂的组件从功能上解耦



👍 3



**shark**

2021-03-12

华哥666，一语道破天机，三个原则相辅相成。合适是先决条件，选择一个不合适的架构，一开始就把简单个抹了，不合适的架构虽然可以演讲，但是这跟让一条鱼去陆地生活一样，虽然从进化的角度是可行的，但是成本大的吓人。简单就是承上启下，越简单越轻量的越容易适配，同时越简单也越容易演化。演化则是适合的升级，业务是不断的升级变化，就像环境不断的变化，物种不选择进化就会被淘汰一样，架构演化其实也是一个不断适合化的过程。

作者回复: 为了得出这3条原则，过程持续了5年时间，其实我原来整理了10多条，最后发现其它基本都可以用这三条推断出来。

当然，这个过程中见过的和踩过的坑那就更多了。



👍 2