架构设计方法论

张晓宁

- 架构设计历史背景
- 架构设计的目的
- 系统复杂度来源
- 架构设计原则
- 架构师恶魔果实能力

架构设计历史背景

- 架构的价值
- 上帝视角看编程语言
- 软件危机
- 架构是什么?

架构的价值


Copyright © 2007 J.R. Heilig
All Rights Reserved
Boeing 737 MAX-8
Length 129ft 7in, baseline model.
© Gunnar J Boe / www.b737.org.uk
Copyright © 2017 Gunnar J. Boe

上帝视角看编程语言

- 机器语言：面向机器，程序员难写、难读、难改。

- 汇编语言：解决了程序员写读改难的问题，但是依然面向机器，需要精通计算机底层知识(CPU指令、寄存器等底层细节)，且不同CPU指令不同。

- 高级语言(LISP/FORTRAN/COBOL)：面向程序员，程序员不在需要关注底层细节，可以专注问题和业务，程序可以通过编译器处理后运行在不同的CPU上。

- 从实验室走向商业环境，编程语言的关注点开始逐渐从机器世界向现实世界倾斜。

第一次软件危机

- 高级语言的出现让软件系统的复杂度有了第一次的大规模增长，20世界60年代中期爆发了第一次软件危机，主要表现在质量低下、无法如期完成、严重超支、重大故障。

- 如：1963年水手一号火箭发射失败是因为一行FORTRAN代码错误导致，价值1900万美元的Bug。

- 如：IBM的OS/360操作系统，1000多程序员写了百万行代码，投入5亿美元，依然项目一再延期，质量得不到保障。佛瑞德·布鲁克斯在随后他的大作《人月神话》中曾经承认，在他管理这个项目的时候，他犯了一个价值数百万美元的错误。

- 1968年NATO正式创造了“软件危机”一词，同时提出解决方法“软件工程”。

- 1968年著名的《*Go To Statement Considered Harmful*》论文引发了长久的论战，并由此产生了自顶向下、模块化指导思想的结构化程序设计方法（派生出了面向过程的设计思想）。

- 结构化设计方法将使用if..else/for取代todo，避免面条式代码，将软件复杂度控制在一定的范围内从而整体的降低了软件开发的复杂度。从而成为20世纪70年代软件开发的潮流。

第二次软件危机

- 随着硬件快速发展（摩尔定律）和应用领域的广泛，业务需求越来越复杂，20世纪80年代迎来了第二次软件危机，其原因是软件生产力远远跟不上硬件和业务的发展。

- 第一次软件危机的复杂度来源于逻辑，第二次软件危机的复杂度来源于扩展。

- 1967年就提出的面向对象设计思想，在此历史背景下被由面向过程演进而来的C++/Java/C#推向了新的高峰。至今依然是主流的设计思想。

- 虽然面向对象降低了扩展复杂度，但事实证明和面向过程一样都不是银弹，只是一种新的软件设计思想。

架构是什么

- 20世纪60年代已经涉及软件架构的概念，20世纪90年代开始在Microsoft等大公司开始流行起来。

- 维基百科对<软件框架>的定义：框架就是基础设施，制定一套规范或者规则，大家在该规范或者规则下工作。或者说使用别人搭好的舞台来做编剧和表演。如：Spring MVC

- 维基百科对<软件架构>的定义：是有关软件整体结构与组件的抽象描述，用于指导大型软件系统各个方面的设计。如：丽呈直销系统架构

- 软件架构的出现存在历史必然性，随着软件规模的增加，系统内部耦合和逻辑复杂的问题导致修改扩展困难，进而拆分成了更多的部分，软件架构既是对各个部分组织的方法论。

“软件危机的主要原因，把它很不客气地说：在没有机器的时候，编程根本不是问题；当我们有了计算机，编程开始变成问题；而现在我们有巨大的计算机，编程就成为了一个同样巨大的问题。”

— 艾兹赫尔·戴克斯特拉，谦逊的程序员，《*Communications of the ACM*》

架构设计的目的

- 架构设计的误区

- 架构设计的目的

架构设计的误区

- 因为架构很重要，所以要做架构设计。这是一句正确的废话。

- 因为某个架构很牛逼，技术很酷炫，所以要按这个架构做。这种做法很容易让架构始终处于实验阶段，难以推广。软件架构是为了商业设计的，不是为了技术设计的。

- 强行照搬其他公司的架构。基本上很快会产生水土不服，轻则不断重构，重则推到重来。

- 为了三高，所以要做架构设计。看似靠谱的观点很可能对项目来说是灾难性的，如果项目刚上线就要面临三高真的睡觉都要笑醒了。

架构设计的目的

- 软件工程发展史就是一部软件复杂度斗争史，架构设计的主要目的是为了解决软件复杂度带来的问题。

- 需要充分的理解商业，理解需求，理解团队，找到自己的复杂度所在。

- 做到有的放矢，不贪大求全，用有限的资源解决真正的问题。

系统复杂度来源

- 高性能
- 高可用
- 扩展性
- 成本、安全、规模

高性能

- 单机高性能：源自充分利用硬件带来的复杂度，如多线程、多进程，如BIO、NIO。

- 集群高性能：源自拆分带来的复杂度，越简单的越容易做到高性能，合理拆分后还可以针对某些点按需扩展。如任务分配，分配器、连接管理、分配算法。如任务分解，按业务模块进行拆分。

- 业务关联：追求更好的用户体验，满足业务快速增长。

高可用

- 核心就是数据不丢，业务不停。

- 内在因素的硬件故障和软件BUG，外在因素断电、地震，使得单点系统本质上无法做到高可用。

- 高可用的本质就是冗余，高可用的复杂度也来自冗余。

- 高可用中的关键难点多来自于存储高可用，如跨地域同步延迟是无法突破的物理定律。如通过冗余来实现高可用系统，状态决策本质上就不可能做到完成正确。

- 业务关联：业务的根本所在，公司的实力象征。

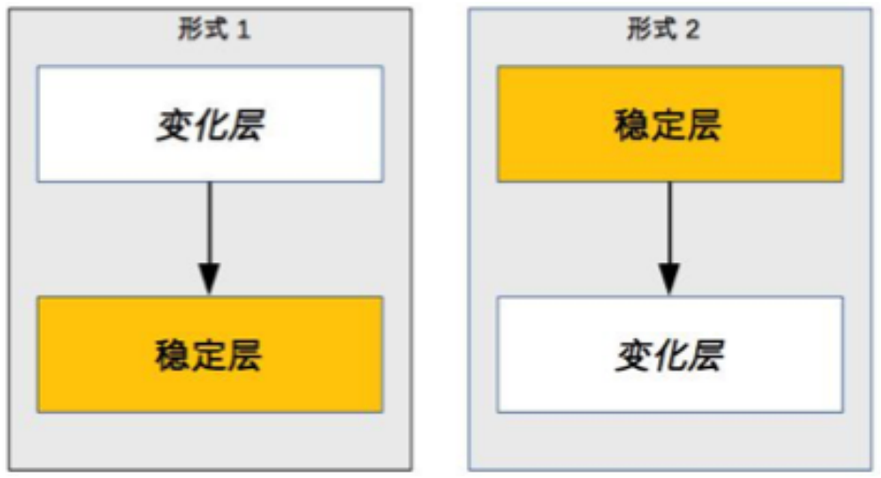
扩展性

- 现代商业环境的快速变化和一些不可描述的原因，带来了需求的不断变化，新需求的不断产生，可扩展性在应对变化中显得尤为重要。

- 面向对象的设计思想缓解了扩展性带来的问题，设计模式更进一步的总结了解决不同扩展性问题时使用的套路。

- 变化无法预测，因为“唯一不变的就是变化”，所以我们能做的就是最大程度的识别变化区，封装变化。

- 业务关联：快速响应业务变化。

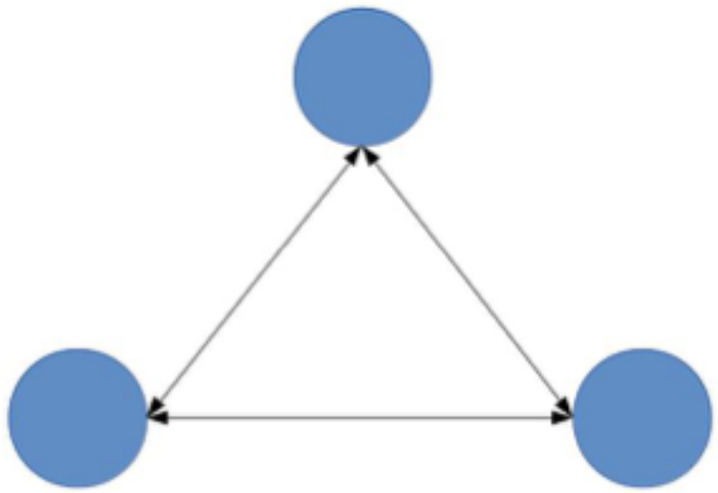


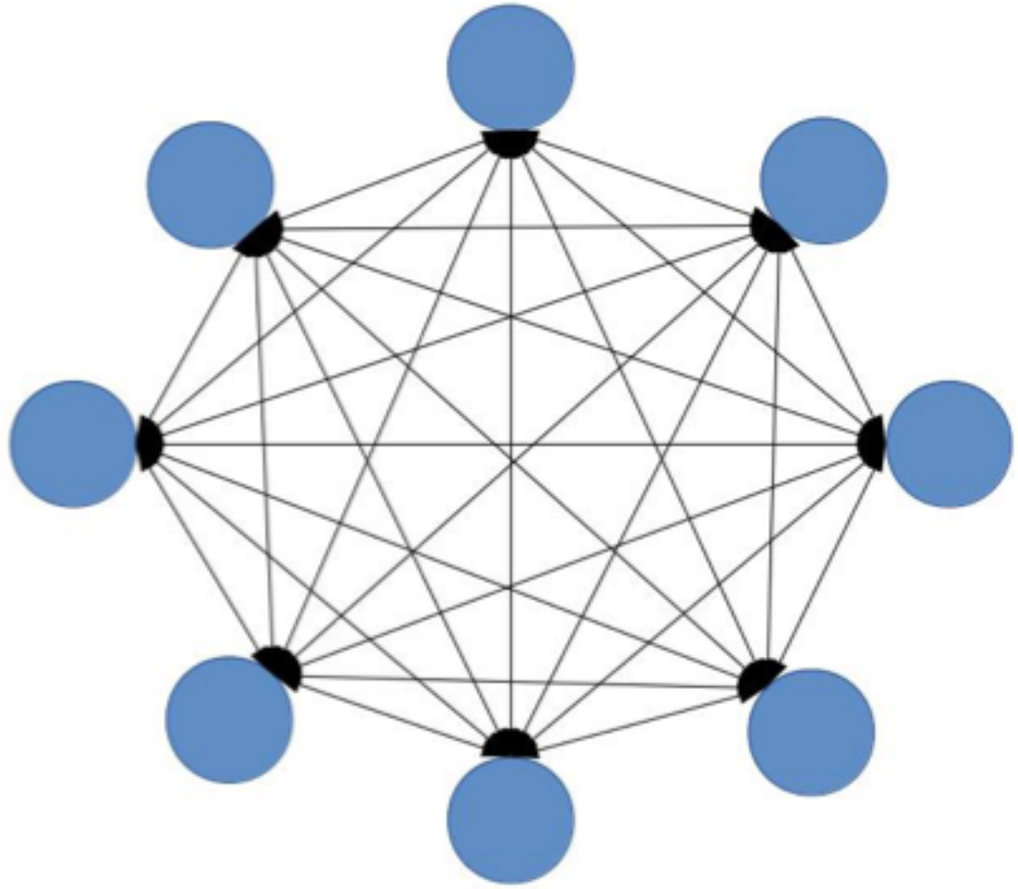
成本、安全、规模

- 成本有运行成本和开发成本两部分，项目初期运行成本(几十台)基本不成问题，开发成本尤为重要，甚至决定了团队的是否能生存下去。随着业务规模的增长成本热点转向运行成本(万台)，引入新的技术或创新来降低运行成本变的尤为重要。

- 安全有功能安全和架构安全，功能安全主要有xss/sql注入等漏洞导致，需要引入代码静态检查和在线监测工具来最大程度避免，架构安全主要有ddos等暴力手段，需要引入如腾讯大禹此类防火墙来做流量清洗。

- 规模主要是业务功能和数据规模的量变带来的质变，业务功能规模复杂度计算公式 <复杂度=功能数量+功能之间的连接数>，大数据行业的火热已经说明了数据规模带来的复杂度。





架构设计原则

- 合适原则
- 简单原则
- 演进原则

合适原则

- 合适优于业界领先：xx方案是Google级的水平，实际使用场景每天20万PV。

- 资源永远是有限的，当10个人要干100个人的活的时候就已经输了。

- 优秀的系统是经过无数的磨练产生的，一步登天的事情并不存在。

- Google的系统之所以顶级的，是因为现有了顶级的场景，才会有顶级的系统。

- 优秀的架构是在企业当前各种限制条件下，合理的将资源最大程度的发挥。

简单原则

- 简单优于复杂：软件架构的复杂并不等于精美，复杂也并不等于水平高
- 组件越多，就越有可能其中某个组件出现故障。
- 某个组件的改动，可能影响关联的多个组件。
- 定位一个复杂系统的问题只会更复杂。
- 优秀的架构就像武林高手，对招中尽量避免不必要的动作。

演进原则

- 演进优于一步到位：对于建筑来说，永恒是主题；对于软件来说，变化才是主题。
- 软件架构的一步到位的结果可能是永远到不了位。
- 面向问题演进，面向业务演进。
- 优秀架构的演进就像生物的进化。

架构师恶魔果实能力

- 理解商业，理解技术，做技术和商业的桥梁
- 复杂度识别
- 权衡，取舍，抑制技术至上的冲动
- 做一个朴实的架构

“如果软件设计真的有银弹，那我相信是演进。”

—程序员小张

丽呈现在各个系统面临的软件复杂度是什么?