

02 | 架构设计的历史背景

2018-05-01 李运华 来自北京


《从0开始学架构》



理解了架构的有关概念和定义之后，今天，我会给你讲讲**架构设计的历史背景**。我认为，如果想要深入理解一个事物的本质，最好的方式就是去追寻这个事物出现的历史背景和推动因素。我们先来简单梳理一下软件开发进化的历史，探索一下软件架构出现的历史背景。

机器语言（1940 年之前）


最早的软件开发使用的是“**机器语言**”，直接使用二进制码 0 和 1 来表示机器可以识别的指令和数据。例如，在 8086 机器上完成 “ $s=768+12288-1280$ ” 的数学运算，机器码如下：

 复制代码

```
1 10110000000000000000000011
2 0000010100000000000110000
3 0010110100000000000000101
```

不用多说，不管是当时的程序员，还是现在的程序员，第一眼看到这样一串东西时，肯定是一头雾水，因为这实在是太难看懂了，这还只是一行运算，如果要输出一个“hello world”，面对几十上百行这样的 0/1 串，眼睛都要花了！

看都没法看，更何况去写这样的程序，如果不小心哪个地方敲错了，将 1 敲成了 0，例如：

 复制代码

```
1 10110000000000000000000011
2 000001010000000000110000
3 0010110000000000000000101
```


如果要找出这个程序中的错误，程序员的心里阴影面积有多大？

归纳一下，机器语言的主要问题是三难：**太难写、太难读、太难改！**

汇编语言（20 世纪 40 年代）

为了解决机器语言编写、阅读、修改复杂的问题，**汇编语言**应运而生。汇编语言又叫“**符号语言**”，用助记符代替机器指令的操作码，用地址符号（Symbol）或标号（Label）代替指令或操作数的地址。


例如，为了完成“将寄存器 BX 的内容送到 AX 中”的简单操作，汇编语言和机器语言分别如下。

 复制代码

```
1 机器语言：1000100111011000
2 汇编语言：mov ax,bx
```

相比机器语言来说，汇编语言就清晰得多了。mov 是操作，ax 和 bx 是寄存器代号，mov ax,bx 语句基本上就是“将寄存器 BX 的内容送到 AX”的简化版的翻译，即使不懂汇编，单纯看到这样一串语言，至少也能明白大概意思。

汇编语言虽然解决了机器语言读写复杂的问题，但本质上还是**面向机器**的，因为写汇编语言需要我们精确了解计算机底层的知识。例如，CPU 指令、寄存器、段地址等底层的细节。这对于程序员来说同样很复杂，因为程序员需要将现实世界中的问题和需求按照机器的逻辑进行翻译。例如，对于程序员来说，在现实世界中面对的问题是 $4 + 6 = ?$ 。而要用汇编语言实现一个简单的加法运算，代码如下：

 复制代码

```
1 .section .data
2   a: .int 10
3   b: .int 20
4   format: .asciz "%d\n"
5 .section .text
6 .global _start
7 _start:
8   movl a, %edx
9   addl b, %edx
10  pushl %edx
11  pushl $format
12  call printf
13  movl $0, (%esp)
14  call exit
```

这还只是实现一个简单的加法运算所需要的汇编程序，可以想象一下，实现一个四则运算的程序会更加复杂，更不用说用汇编写一个操作系统了！

除了编写本身复杂，还有另外一个复杂的地方在于：不同 CPU 的汇编指令和结构是不同的。例如，Intel 的 CPU 和 Motorola 的 CPU 指令不同，同样一个程序，为 Intel 的 CPU 写一次，还要为 Motorola 的 CPU 再写一次，而且指令完全不同。

高级语言（20 世纪 50 年代）

为了解决汇编语言的问题，计算机前辈们从 20 世纪 50 年代开始又设计了多个**高级语言**，最初的高级语言有下面几个，并且这些语言至今还在特定的领域继续使用。

Fortran：1955 年，名称取自“FORmula TRANslator”，即公式翻译器，由约翰·巴科斯 (John Backus) 等人发明。

LISP: 1958 年, 名称取自“ LISt Processor” , 即枚举处理器, 由约翰·麦卡锡 (John McCarthy) 等人发明。

Cobol: 1959 年, 名称取自“ Common Business Oriented Language” , 即通用商业导向语言, 由葛丽丝·霍普 (Grace Hopper) 发明。

为什么称这些语言为“高级语言”呢? 原因在于这些语言让程序员不需要关注机器底层的低级结构和逻辑, 而只要关注具体的问题和业务即可。

还是以 $4 + 6 = ?$ 这个加法为例, 如果用 LISP 语言实现, 只需要简单一行代码即可:

```
1 (+ 4 6)
```

 复制代码

除此以外, 通过编译程序的处理, 高级语言可以被编译为适合不同 CPU 指令的机器语言。程序员只要写一次程序, 就可以在多个不同的机器上编译运行, 无须根据不同的机器指令重写整个程序。

第一次软件危机与结构化程序设计 (20 世纪 60 年代~20 世纪 70 年代)

高级语言的出现, 解放了程序员, 但好景不长, 随着软件的规模和复杂度的大大增加, 20 世纪 60 年代中期开始爆发了第一次软件危机, 典型表现有软件质量低下、项目无法如期完成、项目严重超支等, 因为软件而导致的重大事故时有发生。例如, 1963 年美国

(http://en.wikipedia.org/wiki/Mariner_1) 的水手一号火箭发射失败事故, 就是因为一行 FORTRAN 代码错误导致的。

软件危机最典型的例子莫过于 IBM 的 System/360 的操作系统开发。佛瑞德·布鲁克斯 (Frederick P. Brooks, Jr.) 作为项目主管, 率领 2000 多个程序员夜以继日地工作, 共计花费了 5000 人一年的工作量, 写出将近 100 万行的源码, 总共投入 5 亿美元, 是美国的“曼哈顿”原子弹计划投入的 1/4。尽管投入如此巨大, 但项目进度却一再延迟, 软件质量也得不到保障。布鲁克斯后来基于这个项目经验而总结的《人月神话》一书, 成了畅销的软件工程书籍。

为了解决问题，在 1968、1969 年连续召开两次著名的 NATO 会议，会议正式创造了“软件危机”一词，并提出了针对性的解决方法“软件工程”。虽然“软件工程”提出之后也曾被视为软件领域的银弹，但后来事实证明，软件工程同样无法根除软件危机，只能在一定程度上缓解软件危机。

差不多同一时间，“结构化程序设计”作为另外一种解决软件危机的方案被提了出来。艾兹赫尔·戴克斯特拉（Edsger Dijkstra）于 1968 年发表了著名的《GOTO 有害论》论文，引起了长达数年的论战，并由此产生了**结构化程序设计方法**。同时，第一个结构化的程序语言 Pascal 也在此时诞生，并迅速流行起来。

结构化程序设计的主要特点是抛弃 goto 语句，采取“自顶向下、逐步细化、模块化”的指导思想。结构化程序设计本质上还是一种面向过程的设计思想，但通过“自顶向下、逐步细化、模块化”的方法，将软件的复杂度控制在一定范围内，从而从整体上降低了软件开发的复杂度。结构化程序方法成为了 20 世纪 70 年代软件开发的潮流。

第二次软件危机与面向对象（20 世纪 80 年代）

结构化编程的风靡在一定程度上缓解了软件危机，然而随着硬件的快速发展，业务需求越来越复杂，以及编程应用领域越来越广泛，第二次软件危机很快就到来了。

第二次软件危机的根本原因还是在于软件生产力远远跟不上硬件和业务的发展。第一次软件危机的根源在于软件的“逻辑”变得非常复杂，而第二次软件危机主要体现在软件的“扩展”变得非常复杂。结构化程序设计虽然能够解决（也许用“缓解”更合适）软件逻辑的复杂性，但是对于业务变化带来的软件扩展却无能为力，软件领域迫切希望找到新的银弹来解决软件危机，在这种背景下，**面向对象的思想**开始流行起来。

面向对象的思想并不是在第二次软件危机后才出现的，早在 1967 年的 Simula 语言中就开始提出来了，但第二次软件危机促进了面向对象的发展。**面向对象真正开始流行是在 20 世纪 80 年代，主要得益于 C++ 的功劳，后来的 Java、C# 把面向对象推向了新的高峰。到现在为止，面向对象已经成为了主流的开发思想。**

虽然面向对象开始也被当作解决软件危机的银弹，但事实证明，和软件工程一样，面向对象也不是银弹，而只是一种新的软件方法而已。

软件架构的历史背景

虽然早在 20 世纪 60 年代，戴克斯特拉这位上古大神就已经涉及软件架构这个概念了，但软件架构真正流行却是从 20 世纪 90 年代开始的，由于在 Rational 和 Microsoft 内部的相关活动，软件架构的概念开始越来越流行了。

与之前的各种新方法或者新理念不同的是，“软件架构”出现的背景并不是整个行业都面临类似相同的问题，“软件架构”也不是为了解决新的软件危机而产生的，这是怎么回事呢？

卡内基·梅隆大学的玛丽·肖（Mary Shaw）和戴维·加兰（David Garlan）对软件架构做了很多研究，他们在 1994 年的一篇文章《软件架构介绍》（An Introduction to Software Architecture）中写到：

“When systems are constructed from many components, the organization of the overall system-the software architecture-presents a new set of design problems.”

简单翻译一下：随着软件系统规模的增加，计算相关的算法和数据结构不再构成主要的设计问题；当系统由许多部分组成时，整个系统的组织，也就是所说的“软件架构”，导致了一系列新的设计问题。

这段话很好地解释了“软件架构”为何先在 Rational 或者 Microsoft 这样的大公司开始逐步流行起来。因为只有大公司开发的软件系统才具备较大规模，而只有规模较大的软件系统才会面临软件架构相关的问题，例如：

系统规模庞大，内部耦合严重，开发效率低；

系统耦合严重，牵一发而动全身，后续修改和扩展困难；

系统逻辑复杂，容易出问题，出问题后很难排查和修复。

软件架构的出现有其历史必然性。20 世纪 60 年代第一次软件危机引出了“结构化编程”，创造了“模块”概念；20 世纪 80 年代第二次软件危机引出了“面向对象编程”，创造了“对象”概念；到了 20 世纪 90 年代“软件架构”开始流行，创造了“组件”概念。我们可

以看到，“模块”“对象”“组件”本质上都是对达到一定规模的软件进行拆分，差别只是在于随着软件的复杂度不断增加，拆分的粒度越来越粗，拆分的层次越来越高。

《人月神话》中提到的 IBM 360 大型系统，开发时间是 1964 年，那个时候结构化编程都还没有提出来，更不用说软件架构了。如果 IBM 360 系统放在 20 世纪 90 年代开发，不管是质量还是效率、成本，都会比 1964 年开始做要好得多，当然，这样的话我们可能就看不到《人月神话》了。

小结

今天我为你回顾了软件开发进化的历史，以及软件架构出现的历史背景，从历史发展的角度，希望你深入了解架构设计的本质有所帮助。

这就是今天的全部内容，留一道思考题给你吧。为何结构化编程、面向对象编程、软件工程、架构设计最后都没有成为软件领域的银弹？

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！）

© 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。

精选留言 (243)



公号-技术夜未眠

2018-05-01

2018年5月1日心得

在古代的狼人传说中，只有用银质子弹（银弹）才能制服这些异常凶残的怪兽。在软件开发活动中，“银弹”特指人们渴望找到用于制服软件项目这头难缠的“怪兽”的“万能钥匙”。

软件开发过程包括了分析、设计、实现、测试、验证、部署、运维等多个环节。从IT技术的发展历程来看，先辈们在上述不同的环节中提出过很多在当时看来很先进的方法与理念。但是，这些方法、理念在摩尔定律、业务创新、技术发展面前都被一一验证了以下观点：我们

可以通过诸多方式去接近“银弹”，但很遗憾，软件活动中没有“银弹”。

布鲁克斯发表《人月神话》三十年后，又写了《设计原本》。他认为一个成功的软件项目的最重要因素就是设计，架构师、设计师需要在业务需求和IT技术中寻找到一个平衡点。个人觉得，对这个平衡点的把握，就是架构设计中的取舍问题。而这种决策大部分是靠技术，但是一定程度上也依赖于架构师的“艺术”，技术可以依靠新工具、方法论、管理模式去提升，但是“艺术”无法量化，是一种权衡。

软件设计过程中，模块、对象、组件本质上是对一定规模软件在不同粒度和层次上的“拆分”方法论，软件架构是一种对软件的“组织”方法论。一分一合，其目的是为了软件研发过程中的成本、进度、质量得到有效控制。但是，一个成功的软件设计是要适应并满足业务需求，同时不断“演化”的。设计需要根据业务的变化、技术的发展不断进行“演进”，这就决定了这是一个动态活动，出现新问题，解决新问题，没有所谓的“一招鲜”。

以上只是针对设计领域的银弹讨论，放眼到软件全生命周期，银弹问题会更加突出。

小到一个软件开发团队，大到一个行业，没有银弹，但是“行业最佳实践”可以作为指路明灯，这个可以有。

作者回复: 赞，666，你已经提前帮我做了后面相关内容的预热了👍👍

共 14 条评论 >

👍 780



narry

2018-05-02

软件开发最本质的挑战有两个:复杂和变更，而软件的价值是保证业务的响应力，而与之相对的是开发资源的有限，而各种的软件开发方法论，也都是在研究有限的资源下，如何应对着两个挑战，寻找平衡点，实现业务目标，因为是在寻找平衡点，就说明是有取舍的，所以就没有所谓的银弹的存在

作者回复: 回答的很好，作者也受到了启发，谢谢👍👍

共 4 条评论 >

👍 221



cruise

2018-05-01

从哲学角度来说，是不存在银弹的。任何技术或方法都不是独立来看的，要综合其它各种相关因素来考虑的。因此对别人来说可能是银弹的，对你来说可能是个炸弹了。架构设计也是一样

的，不能脱离业务、公司实际情况、人员配置、经费预算、时间投入等等与技术本身无关的因素，但却又是影响，甚至决定架构设计方向的因素。因此说没有最好，只有更合适。



👍 108



felix

2018-05-02

变化才是唯一的不变，所以银弹不会存在

作者回复：言简意赅，抓住了核心本质，“银弹”产生于一定的历史背景和大环境，而历史和环境总是会变化的

共 2 条评论 >

👍 81



Up

2018-05-01

作者这个问题是否在考验，读者认真看了这篇文章没有？我认为文章的软件发展历史正是答案，软件工程归根结底是为各行各业的需求服务的，而随着需求的复杂度越来越高，用户的要求越来越高，软件也越复杂，形态也在不断变化，所以没有一种方法论能称得上是银弹，只能说某一种方法论适合某一种需求。这也正是架构师存在的意义，去选择合适的技术，如果有银弹，还要架构师干嘛！以上只是个人见解！

作者回复：你已经看穿一切👍👍

确实是想通过介绍历史来启发大家思考



👍 67



李志博

2018-05-01

软件开发的結果在于人，而不在于方法论，面向对象，设计模式，架构，这些概念的推出距离现在，好几十年了吧，可真正理解透彻的能有多少呢，就算有像作者这样理解透彻的，还在一线开发的能有多少.....阿里的p9难道还在一线写代码嘛.....最终写代码的人还是理解不到位的我们，技术强的，写的项目能多撑两年，但是复杂到一定程度，没有良好关系架构指导，都是坑

作者回复：其实不一定要P9才要理解到位呢，我2014年就写了《面向对象葵花宝典》，那时我还在写代码的哦，其实我现在也写代码，不写代码很多技术没法确切理解，我现在写demo代码比较多，例如用golang写个简单的区块链，用java写个reactor等

共 3 条评论 >

👍 42



crazyone

2018-05-06

感觉像是看大佬们在华山论剑般，评论相当精彩

共 1 条评论 >

👍 31



Alspadger

2018-05-01

因为设计者都是站在当时的业务瓶颈下考虑问题的，因为你不可预测当业务发展的一定程度后，又会遇到怎么样的技术瓶颈。也就是所谓的技术支撑业务发展，业务推动技术发展。



👍 29



xuan

2018-05-02

“No Silver Bullet”的原文是：“没有任何技术或管理上的进展，能够独立地许诺十年内使生产率、可靠性或简洁性获得数量级上的进步。”之所以这样说，是因为软件的根本困难（Essence，包括复杂度、一致性、可变性、不可见性）

复杂度:规模上, 软件实体可能比任何由人类创造的其他实体更复杂, 因为没有任何两个软件部分是相同的

一致性:软件的变化必须遵循一系列接口标准规范,有些情况下它的变化就是要兼容;

可变性:一般有如下几种情况:

1.当客户喜欢用某个功能或者某个功能能解决他的某些问题时,他会针对这方面提出很多优化该功能的需求点

2.硬件或者其他配件的升级变化 必须升级现有软件平台

不可见性:软件不存在一种空间形态 可以通过一张图

或者其他载体来可视化展示 ,不能通过地图 电路设计图等来全面展示.

由于这几个点的变化，导致系统越来越臃肿,从而导致管理成本上升,沟通困难,可靠性逐年下降等等；而结构化 面向对象等主要是来提高生产率 可靠性和简洁性

作者回复: 没有看过《人月神话》的程序员不能成为好的架构师 😊😊👍👍



👍 26



Mark Yao

2018-05-02

软件本身的复杂度难以度量，随时间和规模发展，原有的解决方案很快难适应，人们就不断总结经验模式和设计解决新困难的办法，但是不管什么样的架构设计都是在尽量满足适应我们可

能遇到的问题的解决方案，不是解决问题方案。生活中我们的应用从单体到主备再到集群、分布式、微服务最后到最新的Service Mesh，这些其实都是解决和改善、完善、优化我们在软件开发遇到的问题。There is no silver bullet.

作者回复: 回答正确👍



👍 20



淡云天

2018-05-02

解空间是建立在问题空间之上的，问题空间的扩展速度远超解空间时，就会架空解空间。而这时就需要新的、适应问题空间扩展速度的解空间来担当这个阶段的银弹。这一点类似于宏观物理学和量子物理学，只不过物理学几百年的进化之路，计算机只用了二十年就走完了。。。



👍 16



yoummg

2018-07-08

作者的用心令人敬佩。

为什么现在我们在谈“架构”，他不是平白无故产生的，他是在一定的背景下产生的。更好地理解他产生的原因，会在具体解决问题的时候做到有的放矢。

直到现在才看明白，what，why，how。这真是一个认清事物最本质的三步。👍👍👍

作者回复: 你已经洞悉天机👍👍😄 整个专栏思路就是这样的



👍 15



KingPoker

2018-05-01

推荐一本书「伟大的计算原理」，把计算机的本质问题描述的很透彻，也给我有一些全新的认知。



👍 12



闭嘴

2018-05-02

感觉作者对整个软件行业有比较深入的了解。就是内容太少。还没看就没了。希望后面的文章多来一点干货。让我这种小白能够学习到一点实质的东西。能够解决项目问题的一些东西。希望大神能够把自己的功力展现60%就行。

作者回复: 这是提炼出来的, 为了写这一篇, 我写了2~3周, 如果觉得犹未尽, 可以在这个基础上继续去探索



👍 7



带刺的温柔

2018-05-01

软件架构是为了解决大规模开发时遇到的效率、复杂及扩展性问题。听老师所说让我对架构认知又更加清晰落地。但是对拆分粒度越来越粗, 层次越来越高理解的还是不够, 其实与我的一些开发习惯是相悖的, 一般我会尽可能拆分细来保证后期的扩展性。不知道老师我是哪里理解偏差了

共 1 条评论 >

👍 7



强

2018-05-02

关于银弹, 我想从另外一个角度聊聊。上学时候, 老师(c++标准编写者)跟我们分享的一思考题: 软件究竟属于工程行业还是偏艺术(或工艺)行业。前几十年, 软件从业者基本是努力将其往工程化发展, 像硬件制造一样可控, 高度复用, 流水生产。经过这些年的发展, 工程化基本未实现(否则码农就和生产线工人一样, 工资不会越来越高), 现在越来越多人思考, 也行软件更多是艺术行业。既然是艺术类, 自然就无银弹的说法



👍 6



阿罗

2018-05-01

超赞👍,

饱含认识论和系统论的理论知识与软件实践知识。太难得了, 非大集成者无以为之。我买过最值的课程!



👍 6



候鸟归来的季节

2018-05-01

技术在不断发展, 新的业务需求催生新的技术, 没有银弹



👍 6



Geek_92f9aa

2020-10-29

一个答案解决所有问题：“因为熵增定律”。

而熵增的表现其实就是变化。

那如何克服这一变化？

同样是一句话概括：“生命以负熵为食”。

即在生物界，生命通过已知的信息完成外界能量到自身的转移，这个过程虽然逃不过熵增定律，但通过加速外界的熵增实现了生命自身熵的不变，生物因此得以维持自身状态不变(即活着，没死)

文章说到的架构的历史，其实就是一个对抗熵增的生命演化史。软件本身没有生命，所以要依靠人来实现自身状态维持。

即如果我们将软件和人看成一个整体，那么其状态即是可维持的，所以这就是银弹。而如果将人从这个整体中剥离出去，软件就失去了生命力，无法永远维持自身状态，再牛逼的设计也不可能成为银弹，除非让其拥有对抗熵增的能力，那样的话软件也是有生命的。从这点来看，人工智能极有可能成为一个新生物，届时再也不需要程序员了，恩，人也不需要了，哈哈，细思极恐。

作者回复：别担心，你我有生之年应该还不会被人工智能干掉??????

共 2 条评论 >

👍 5



Will

2018-05-05

怎么可能出现一套方法能解决一万年后出现的问题呢？
可能时间是“银弹”吧。



👍 5