02 架构设计的历史背景

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：

```
10110000000000000000011
00000101000000000110000
00101101000000000000101
```

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：

```
10110000000000000000011
00000101000000000110000
00101100000000000000101
```

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。

```
机器语言：1000100111011000
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：

```
.section .data
  a: .int 10
  b: .int 20
  format: .asciz "%d\n"
.section .text
.global _start
_start:
  movl a, %edx
  addl b, %edx
  pushl %edx
  pushl $format
  call printf
  movl $0, (%esp)
  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：

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，希望对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？

上一页 下一页