02 | 架构设计的历史背景 李运华 2018-05-01 讲述: 黄洲君 大小: 5.08M 时长: 11:05 理解了架构的有关概念和定义之后,今天,我会给你讲讲<mark>架构设计的历史背景</mark>。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: 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 国 复制代码 1 1011000000000000000000011 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。 目 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 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: 目 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 ル&씨ㅋ T V=: 쓰ㅣ께/시기/リ, 제不/미니이 떠ㅁ不센, /(而ഠ미누 비ㅣV时바다. 目 复制代码 1 (+ 4 6) 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 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." 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? 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!) 更多学习推荐 **一般客大学** 架构技不囬试题 快速攻克架构师技术面试 限时免费 🖁 仅限 99 名 © 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。 Geek_8cc8c5 由作者筛选后的优质留言将会公开显示,欢迎踊跃留言。 提交留言 0/2000字 Ctrl + Enter 发表 精选留言(199) 公号-技术夜未眠 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 (1) 1 JA(1): 170 IN 4F/M L 751 / 1/14/UN ECTISEVENE/1) LAC ECTI / U/MATIU ; AC 11 IA/M 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 2018-05-01 9 603 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,谢谢备备 1 2018-05-02 **167** cruise

> felix 变化才是唯一的不变, 所以银弹不会存在 2018-05-02 Wp Up 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 2018-05-01 **1** 49 李志博 软件开发的结果在于人, 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 **1** 37 2018-05-01 xuan "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 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;

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

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

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

23

(21

1 21

6 7

...

6 5

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.

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 😃 😃 📥

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。

结构化 面向对象等主要是来提高生产率 可靠性和简洁性

感觉像是看大佬们在华山论剑般,评论相当精彩

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。

2018-05-01

可变性:一般有如下几种情况:

2018-05-02

Alspadger

2018-05-01

crazyone

2018-05-06

yoummg yoummg

作者的用心令人敬佩。

2018-05-01

超赞凸,

闭嘴

的功力展现60%就行。

上继续去探索

2018-05-02

的原因,会在具体解决问题的时候做到有的放矢。

2018-07-08

■ KingPoker

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。

2018-05-01

□ □ 1

▼ ▼ 1

▼ ▼ 1

▼ ▼ 1

▼ ▼ 1

▼ ▼ 1

▼ ▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

▼ 1

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

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

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

程! **候鸟归来的季节**技术在不断发展,新的业务需求催生新的技术,没有银弹
2018-05-01

□ 6

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

作者回复: 这是提炼出来的,为了写这一篇,我写了2~3周,如果觉得意犹未尽,可以在这个基础

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

所有的解决方案都是为了解决某一类问题, 既然是某一类问题势必是术业有专攻, 从辩证的角度也很好 理解, 事物都有两面性, 不存在某种方案可以解决所有问题而没有任何弊端, 有弊端自然会出现针对某

个场景更好的解决方案。而且,世界在变化,针对现实世界解决问题的软件业务逻辑也势必不断变化,所以只有针对多数场景暂时的银弹,而没有永久的。未来还有更好的
2018-05-01

Welton
技术在不断发展和更新,没有一劳永逸的事物能够代替不断变化的大千世界需求!