软件架构发展历程分享

什么是架构

计算机科学和程序设计的飞速发展，使得软件设计应用到从航空航天到日常生活的方方面面。单个人开发一段小程序的做法早就过时，大范围协作的工程化时代随即到来。随着大范围协作的效率问题和软件复杂度的爆炸式增长，管理和技术方面的各种不确定性也爆发性增加，导致软件开发的质量无法得到有效保证，周期和成本无法得到有效控制。人们一直在寻求找到这些问题的解决办法。然而 Fred Brooks 在1975年出版的软件工程圣经《人月神话》中说，没有（能解决所有问题的）银弹（ There is no silver bullet ）。自此，人们发展了项目研发过程管理来控制管理活动的不确定性，同时也发展了软件架构设计方法来控制技术方面的不确定性，进而在实践中不断的总结和改进，用于有效指导和最大程度的保障软件开发的质量、周期和成本。

架构的定义

架构（ Architecture ）一词源于建筑领域，其本身就是建筑的意思，也是体系结构的意思。维基百科英文版里对 Architecture 的解释是：规划、设计和建造建筑物的过程及产物。鉴于软件工程与建筑工程一样是一项系统的工程性工作，引入到计算机领域后，软件架构就成为了描述软件规划设计技术的专有名词。特别地，软件架构师一词在英文里，和建筑师也是同一个词（ Architect ）。

维基百科里对软件架构的定义：

> 软件架构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。软件架构师定义和设计软件的模块化，模块之间的交互，用户界面风格，对外接口方法，创新的设计特性，以及高层事物的对象操作、逻辑和流程。软件架构是一个系统的草图。软件架构描述的对象是直接构成系统的抽象组件。各个组件之间的连接则明确和相对细致地描述组件之间的通讯。在实现阶段，这些抽象组件被细化为实际的组件，比如具体某个类或者对象。在面向对象领域中，组件之间的连接通常用接口来实现。

比较公认的软件架构定义是在2000年的 ANSI/IEEE 1471 标准中定义的：

> 1. 架构过程：在系统整个生命周期中构思、定义、表达、记录、交流，验证合适实现，维护和改进架构的过程，也就是设计过程。
> 2. 架构：一个系统体现在其环境中的元素、关系的基本概念或属性，以及其设计和进化原则。

3. 架构描述：表达一个架构的工作产出物（通常指的是各种架构图和设计文档）。
4. 架构视图：通过系统的某些关注点的视角，表达一个系统的工作产出物（例如部署视图、开发视图等）。
5. 系统：包含了一个或多个进程、硬件、软件、工具与可以满足需求的人的集合。
6. 环境：决定了开发、操作、策略和其他影响系统的设置和条件。

在 UML 中，架构则被认为是系统的组织结构和相关行为。架构可被分解为通过接口互联部分的关系，以及相互作用。通过接口相互作用的部分包括类、组件和子系统。这样就可以通过 UML 的各种架构图来描述这些对象和关系，从而表达清楚一个系统的架构。

总结：软件架构是一个用于指导系统实现的草图，这个草图越详细对于系统实现的指导意义越重大，贯穿于软件的整个生命周期。在建筑领域，大楼尚未建造前，就已经存在于建筑师的脑海里；同样地，系统开始编写第一行代码之前，就已经存在于软件架构师的心里。

几个相关概念

模式（Pattern）

UML 中给出的解释更通俗易懂：模式是对于普遍问题的普遍解决方案。我们可以把一类问题的共性抽象出来，这样就可以用同样的处理办法去解决这些问题，从而形成模式，所以模式是一些经验的总结。

类库（Library）

类库是一组可复用的功能或工具的集合，应用系统通过调用它们从而达到复用功能的目的。例如，Windows 应用开发里的各种静态或动态链接库 DLL 文件，Java 开发中项目里依赖的或者 Maven 中央库里的各种 jar 包，都是 Library，比如 Apache Commons IO、HttpClient，Log4j 等。

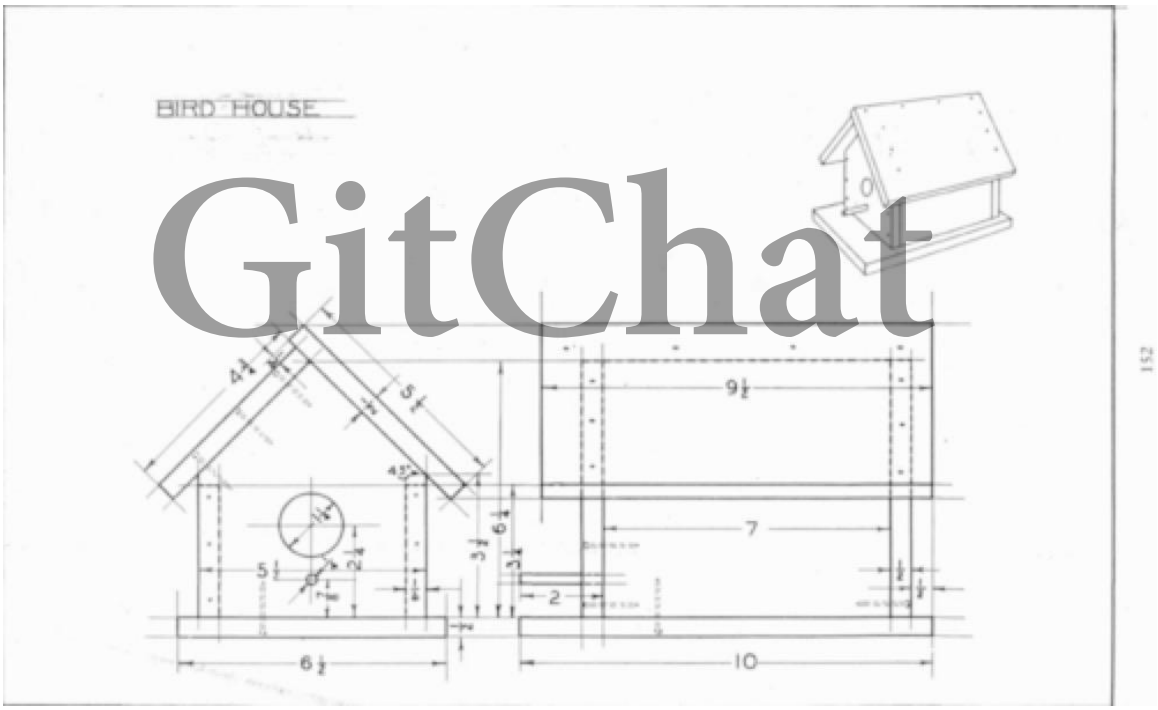
框架（Framework）

框架是基于一组类库或工具，在特定领域里根据一定的规则组合成的、开放性的应用骨架，比如 SSM/SSH 框架，更大范围来说 Dotnet Framework、JDK 都算是一种框架。

关于框架与架构的关系，Vasyl Boroviak 曾在 Stack Overflow 网站上通过两张图做了形象的对比，如下所示。

- 框架：

- 架构：



模块（Module）

模块是业务或系统的安装特定维度的一种切分，同时也可以看做是各种功能按照某种分类聚合的一种形式。例如我们的一个电商系统，可以从业务上划分为用户模块、商品模块、订单模块、支付模块、物流模块、售后模块等。另一方面，我们也可以说用户模块聚合了用户注册、用户验证等业务功能。

组件（Component）

组件是一组可以复用的业务功能的集合，包含一些对象及其行为。组件可以直接被用做业务系统的组成部分，粒度一般小于模块，也是一种功能的聚合形式。比如日志组件、

权限组件等。

服务（Service）

服务是一组对外提供业务处理能力的功能，服务需要使用明确的接口方式（比如 WebService 或 Rest 等），服务描述里应该包括约束和策略（比如参数、返回值，使用什么通讯协议和数据格式等）。

平台（Platform）

平台一般来说，是一个领域或方向上的生态系统，是很多解决方案的集大成者，提供了很多的服务、接口、规范、标准、功能、工具等。例如 J2EE 平台，包含了企业级应用开发里的各种基于 Java 语言和 JVM 虚拟机运行时的技术能力。

知乎社区编程领域优秀问题回答者 ze ran 说：

> 库是工具箱。
>
> 框架是一套通用的解决方案。
>
> 架构是高度抽象的需求，是系统中的不变量。
>
> 平台是所有可能做的事的集合。

从软件的生命周期看架构设计

设计期

在设计期，软件作为一个成品还不存在，所以我们可以称之为概念形态。此时架构师、产品经理或需求分析师等人员利用自己的经验能力，对系统的业务需求进行分析、拆解、抽象，形成业务文档和技术文档，以及技术验证代码等。这个阶段，架构设计工作是重中之重，其中包括：

- 系统分拆：如何把系统拆解为不同的子系统、模块、业务单元；
- 技术选型：使用什么样的基础技术框架或脚手架；
- 技术验证：确定核心技术难点如何解决，检验能否满足期望指标；
- 接口规范：系统的内部不同部分以何种形式确定接口契约和数据通信；
- 集成方式：系统与外部其他业务系统如何进行集成；
- 技术规范：如何规范开发、测试、部署和运维的技术标准性；
- 部署方案：系统如何进行物理部署，需要多少机器、什么配置，对网络有什么要求；
- 运维方案：系统如何进行技术性运维，如何日常监控、预警报警；

- ……

这个阶段总结一下就是：业务为要，架构先行（包括业务架构和技术架构）。

实现期

这个阶段主要是编码与测试，准备部署上线，是软件从代码到最终的生产系统的过程，我们可以称之为代码形态。此阶段需要考虑的技术类工作包括：

- 确保各项技术规范和技术指标的执行落地，保障高质量的代码；
- 指导研发人员和解决各类技术问题，提升研发团队效率；
- 制定测试的技术性方案和基准，自动化、性能、安全等；
- 配合准备部署环境，运维实施方案落地等。

运行期

这个阶段系统上线、验收通过，已经初步稳定，然后进入维护阶段，成为了设计期架构设计草图的一个可用实例，我们可以称之为实例形态。此时需要考虑：

- 发布上线相关基础性工作，包括是否使用持续集成（CI)、自动化发布等技术；
- 运维基础性工作，自动化运维，监控等相关技术。

架构的形式与特点

设计文档和代码

我们一般说的架构既包括架构的设计过程，也包括设计的产出物，一般可以包括各类设计文档、设计图，也可以包括一些技术验证代码、Demo 或者其他相关程序。文档的目的在于准确记录我们的思维产物，在软件尚未实现时，作为指导蓝图，尽量精确的描述清楚软件。在软件的实现过程中，可能随时随着我们的深入研究，根据具体情况对文档做出局部的一些调整和修改。文档作为结项或交接的一部分，也是整个软件项目的产出物的一部分，成为公司 IT 资产的有机组成部分。

架构服务于业务

正如19世纪的伟大建筑师路易斯•沙利文（Louis Sullivan）倡导的建筑设计著名格言：“功能决定形式（Form follows function）”，软件架构首先是要服务于业务功能的。

架构影响研发团队的组织形式

业务拆分的方法和技术框架的选择必然会影响到研发团队的组织形式。业务拆分的越细致，越有利于我们更好的对项目的各项指标量化计算，更精确的估计工时和成本，从而指导我们每个小组应该分配多少资源，使用什么样的协同和任务确认形式。并且随着项目的推进，计划与实际情况之间的匹配程度也随时可以进一步精确调整，进而影响到我们应该对每一块任务的投入资源进行动态调整。

架构存在于每一个系统

每一个已经实现并运行的系统，都是特定架构设计的载体。有些系统对应的架构，有详细的设计文档来描述；有些系统的设计文档，残缺不全，甚至还因为在系统的发展变化的同时，文档没有更新，导致设计文档与实际系统不符；有些系统干脆就没有设计文档。但是这些系统，都是基于一定的架构来创建的。

每种架构都有特定的架构风格

每种架构方式，每个具体系统内所体现的架构设计，都是可以被工程师们理解，进而提炼出来一些架构思想和设计原则，这些思想和原则就是这种架构方式的风格。依据这些风格，我们可以将各种架构方式，进行分门别类，从而进一步讨论每种架构风格的特点。

架构需要不断的发展演进

随着计算机软硬件的不断发展，软件架构思想也在不断的发展变化。另一方面，软件为其提供业务处理和服务能力的每个具体行业领域也在不断发展变化，业务处理流程、参与角色、业务形式不断的推陈出新。这就要求我们在系统架构设计时，保持终身学习的精神，持续吸收新思想新知识，保持贴近一线业务群体，随时因地制宜，调整架构设计，采取最适合当下场景的解决方案。

架构的目标与方法

明确软件系统架构的一些通用目标，可以使我们更明确如何考虑架构的方向；而了解架构的方法和方法论，则让我们可以知道从哪些角度可以比较全面的描述清楚一个系统的架构设计。

可控性与拆分

对于复杂问题的简化处理，一个简单办法就是分而治之。按一定的粒度把目标问题进行分解，可以非常有效的提升目标的可控性，使得目标变得更加可以量化、进而优化。

系统按照合适的粒度拆分成不同模块的过程，我们一般称为模块化。模块化也是软件工程化的基础。在这个基础上才能够实现分工合作。

复用性与抽象

复用性一直是软件设计领域的一个很重要的指标。复用的一个关键是我们对于现有具体问题的抽象，找到各种不同问题中存在的不变性，进而作为一种通用结构来统一处理。

拆成是把整体变成很多局部，再对局部分开对待和研究其性质。反过来，我们按照高内聚的指导思想把一些紧密联系的功能聚合后，打包成一个可以整体复用的部分，这就是组件，这个过程就是组件化。通过组件化，我们可以得到抽象再组合出来很多业务组件。这样，在更大粒度上实现了功能的复用。

非功能性需求九维目标

（1）高性能

系统必须满足预期的性能目标，在并发用户数（Concurrent Users）、并发事务数（Transactions per Second，TPS）、吞吐量（Throughout）等指标方面达到预估值，支撑使用人群的正常使用操作。

（2）可靠性

业务系统直接影响到用户的经营和管理，因此必须是可靠的。

（3）稳定性

软件系统必须是能够在用户的使用周期内长期稳定运行的。这要求系统具有一定的容错能力。

（4）可用性

可用性是指系统在指定时间内的提供服务能力的概率值。我们一般采取集群、分布式等手段提升系统的可用性。高可用性是目前系统架构设计方面的一个热点。

（5）安全性

用户的业务数据是具有非常高的商业价值，如果被泄露或篡改将会带来重大损失。安全性是软件系统的一个重要的指标，也是架构设计的一个重要目标。

（6）灵活性

软件系统应该具备满足不同特点的用户群和目标市场的能力，更灵活。

（7）易用性

软件系统必须拥有较好的用户体验，便于用户使用。

（8）可扩展性

业务和技术都在不断的发展变化，软件系统需要随时根据变化扩展改造的能力。

（9）可维护性

软件系统的维护包括修复现有的错误，以及将新的需求和改进添加到已有系统。因此一个易于维护的系统对于用户提出的问题或改进，可以及时的实现高效的反馈和响应支持，同时有效降低维护成本。

基于这些目标，经常有人说：“架构是系统非功能性需求的解决办法的集合”。

架构的不同风格

典型的企业级应用系统或者互联网应用系统一般都是通过 Web 提供一组业务服务能力。这类系统包括提供给用户操作的、运行于浏览器中、具有 UI 的业务逻辑展示和输入部分，运行于服务器端、用后端编程语言构建的业务逻辑处理部分，以及用于存储业务数据的关系数据库或其他类型的存储软件。

根据软件系统在运行期的表现风格和部署结构，我们可以粗略地将其划分为两大类。

（1）整个系统的所有功能单元，整体部署到同一个进程（所有代码可以打包成1个或多个文件），我们可以称之为“单体架构”（Monolithic Architecture）；

（2）整个系统的功能单元分散到不同的进程，然后由多个进程共同提供不同的业务能力，我们称之为“分布式架构”（Distributed Architecture）。

再结合软件系统在整个生命周期的特点，我们可以进一步区分不同的架构风格。

对于单体架构，我们根据设计期和开发实现期的不同模式和划分结构，可以分为：

- 简单单体模式：代码层面没有拆分，所有的业务逻辑都在一个项目（Project）里打包成一个二进制的编译后文件，通过这个文件进行部署，并提供业务能力；
- MVC 模式：系统内每个模块的功能组件按照不同的职责划分为模型（Model）、视图（View）、控制器（Controller）等角色，并以此来组织研发实现工作；
- 前后端分离模式：将前后端代码耦合的设计改为前端逻辑和后端逻辑独立编写实现的处理模式；
- 组件模式：系统的每一个模块拆分为一个子项目（SubProject），每个模块独立编译打包成一个组件，然后所有需要的组件一起再部署到同一个容器里；
- 类库模式：A 系统需要复用 B 系统的某些功能，这时可以直接把 B 系统的某些组件作为依赖库，打包到 A 系统来使用。

对于分布式架构，我们根据设计期的架构思想和运行期的不同结构，可以分为：

- 面向服务架构（Service Oriented Architecture）：以业务服务的角度和服务总线的方式（一般是 WebService 与 ESB）考虑系统架构和企业 IT 治理；
- 分布式服务架构（Distributed Service Architecture）：基于去中心化的分布式服务框架与技术，考虑系统架构和服务治理；
- 微服务架构（MicroServices Architecture）：微服务架构可以看做是面向服务架构和分布式服务架构的拓展，使用更细粒度的服务（所以叫微服务）和一组设计准则

来考虑大规模的复杂系统架构设计。

也有人把如上的各个架构风格总结为四个大的架构发展阶段：

（1）单体架构阶段
（2）垂直架构阶段
（3）SOA 架构阶段
（4）微服务架构阶段

这种分类跟前述的方式并没有多大区别。我们接下来详细介绍其中的一些重要架构风格。

单体架构：简单单体模式

简单单体模式是最简单的架构风格，所有的代码全都在一个项目中。这样研发团队的任何一个人都可以随时修改任意的一段代码，或者增加一些新的代码。开发人员也可以只在自己的电脑上就可以随时开发、调试、测试整个系统的功能。也不需要额外的一些依赖条件和准备步骤，我们就可以直接编译打包整个系统代码，创建一个可以发布的二进制版本。这种方式对于一个新团队的创立初期，需要迅速开始从0到1，抓住时机实现产品最短时间推向市场，可以省去各种额外的设计，直接上手干活，争取了时间，因而是非常有意义的。

但是这种方式对于一个系统的长期稳定发展确实有很多坏处的。

首先，简单单体模式的系统存在代码严重耦合的问题。所有的代码都在一起，就算是按照 package 来切分了不同的模块，各不同模块的代码还是可以直接相互引用，这就导致了系统内的对象间依赖关系混乱，修改一处代码，可能会影响一大片的功能无法正常使用。

第二，简单单体模式的系统变更对部署影响大，并且这个问题是所有的单体架构系统都存在的问题。系统作为一个单体部署，每次发布的部署单元就是一个新版本的整个系统，系统内的任何业务逻辑调整都会导致整个系统的重新打包，部署、停机、再重启，进而导致了系统的停机发布时间较长。每次发布上线都是生产系统的重大变更，这种部署模式大大提升了系统风险，降低了系统的可用性。

第三，简单单体模式的系统影响开发效率。如果一个使用 Java 的简单单体项目代码超过100万行，那么在一台笔记本电脑上修改了代码后执行自动编译，可能需要等待十分钟以上，并且内存可能不够编译过程使用，这是非常难以忍受的。

第四，简单单体模式打包后的部署结构可能过于庞大，导致业务系统启动很慢，进而也会影响系统的可用性。这一条也是所有单体架构的系统都有的问题。

第五，扩展性受限，也是所有单体架构的一个问题。如果任何一个业务存在性能问题，那么都需要考虑多部署几个完整的实例的集群，或者再加上负载均衡设备，才能保证整个系统的性能可以支撑用户的使用。

所以，简单单体模式比较适用于规模较小的系统，特别是需要快速推出原型实现，以质量换速度的场景。

单体架构：MVC 模式

MVC 也是一个非常常见的3层（3-Tier）结构架构模式，它把每个模块划分为模型层（Model Layer）、视图层（View Layer）、控制器层（Controller Layer）等部分。

- 模型层：代表业务数据实体部分；
- 视图层：代表前端的展示部分；
- 控制器层：代表请求分发，处理调度部分。

更一般地，我们可以添加数据操作层（Data Access Layer）等，形成一个 N 层（N-Tier）结构模型。

整个系统由多个模块组成，每个模块又由这种不同的部分组成。这样一来，我们就把整个系统拆解成了很多粒度较小的零件。这种方式之所以流行开来，主要是因为：

- 简单，直观，可以非常有效的上手，作为 Web 系统设计的一般方法，这一点是 MVC 模式被普及的基础条件；
- 经过上面过程的横割竖切，我们已经把系统拆解为一个个的小单元，非常有利于分配开发工作，投入大量的工程师进行大规模的工程化开发，这一点非常重要，在软件行业高速发展的今天，适合工程化的方式才是最具有生产力的办法；
- 不同的模块和分层结构，本身就可以作为一个开发层面的子项目拆分结构，这样我们就可以把系统拆分成多个不同的子项目；
- 基于 MVC 模式，又陆续发展了 ORM 等简化数据操作层的技术与框架，以及相应的代码生成工具等，极大的提供了软件开发效率。

当然，MVC 模式也存在定义不够明确，对于简单的业务场景拆解过细导致复杂度增加等问题，需要在实践中不断摸索和总结应用经验。基于单体架构下的 MVC 模式依然解决不了单体架构本身存在的问题，特别是对于可用性和扩展性的影响。

单体架构：前后端分离模式

传统的 Web 系统都是 BS 结构的，一般的 JSP 或页面标签 Tag 技术、后端 FreeMaker 或 Velocity 等模板技术导致 HTML/CSS/JavaScript 等前端技术与后端的处理逻辑和数据耦合到一起，这种方式明显不符合现代工程化的专业领域细分原则。特别是随着富网络应用程序（Rich Internet Application，RIA）概念的兴起，Ajax 和 JQuery 框架，前端 UI 组件技术的大行其道，程序员们在浏览器端写了很多逻辑处理和界面处理的 JavaScript 代码。后来越来越多的业务逻辑需要在浏览器端实现，前端技术逐渐发展到了一个百花齐放的阶段，特别是近年来基于 NodeJS 前端技术栈的成功应用，最终使前端技术成为了一个与后端技术领域并驾齐驱的领域。

其实早在2006年，ExtJS 作为当时的前端解决方案集大成者，已经实现了前端代码逻辑和界面全部都由 JavaScript 完成，后端只提供基于 URL 的 JSON 数据接口。这样 Web 系统

就由原来的 BS 系统，变成了提供 UI 和交互的前端 B 系统，提供数据接口的后端 S 系统，从而达到了前后端分离的目标。自从，越来越多的系统采用前后端分离的模式，并且进一步影响了研发团队的组成：前端团队负责前端系统开发，后端团队负责后端系统开发，两个团队一起制定前后端系统的数据接口。

只要数据接口保持稳定不变，那么前后端系统可以各自独立发展和维护。这一条准则不仅仅是单体架构独有的，所有的 Web 系统都可以按照这种方式进行设计。前后端分离模式一直影响到现在的系统架构方法，成为了当下的一种最佳实践。目前最主流的三种前端开发框架（React、AngularJS、Vue），都遵循着这种设计理念。

分布式架构：面向服务架构（SOA）

- 服务与SOA

面向服务架构（SOA）是一种建设企业 IT 生态系统的架构指导思想。SOA 的关注点是服务。服务最基本的业务功能单元，由平台中立性的接口契约来定义。通过将业务系统服务化，可以将不同模块解耦，各种异构系统间可以轻松实现服务调用、消息交换和资源共享。

（1）从宏观的视角来看，不同于以往的孤立业务系统，SOA 强调整个企业 IT 生态环境是一个大的整体。整个 IT 生态中的所有业务服务构成了企业的核心 IT 资源。各系统的业务拆解为不同粒度和层次的模块和服务，服务可以组装到更大的粒度，不同来源的服务可以编排到同一个处理流程，实现非常复杂的集成场景和更加丰富的业务功能。

（2）从研发的视角来看，系统的复用可以从以前代码级的粒度，扩展到业务服务的粒度；能够快速应对业务需求和集成需求的变更。

（3）从管理的角度来看，SOA 从更高的层次对整个企业 IT 生态进行统一的设计与管理，对消息处理与服务调用进行监控，优化资源配置，降低系统复杂度和综合成本，为业务流程梳理和优化提供技术支撑。

- SOA 落地方式

SOA的落地方式与水平，跟企业 IT 特点、服务能力和发展阶段直接相关。目前常见的落地方式主要有分布式服务化和集中式管理两种。

（1）分布式服务化

互联网类型的企业，业务与技术发展快，数据基数与增量都大，并发访问量高，系统间依赖关系复杂、调用频繁，分布式服务化与服务治理迫在眉睫。通过统一的服务化技术手段，进一步实现服务的注册与寻址、服务调用关系查找、服务调用与消息处理监控、服务质量与服务降级等等。现有的一些分布式服务化技术有 Dubbo（基于 Java）、Finagle（基于 Scala）和 ICE（跨平台）等。

（2）集中式管理

传统企业的 IT 内部遗留系统包袱较重，资源整合很大一部分是需要打通新旧技术体系的任督二脉，所以更偏重于以 ESB 作为基础支撑技术，以整合集成为核心，将各个新旧系统的业务能力逐渐的在 ESB 容器上聚合和集成起来。比较流行的商业 ESB 有 IBM 的 WMB 和 Oracle 的 OSB，开源 ESB 有 Mule、ServiceMix/WSO2 ESB、JBoss ESB 和 OpenESB。

一方面，集中式管理的 SOA，其优势在于管理和集成企业内部各处散落的业务服务能力，同时一个明显的不足在于其中心化的架构方法，并不能解决各个系统自己内部的问题。另一方面，随着自动化测试技术、轻量级容器技术等相关技术的发展，分布式服务技术越来越像微服务架构方向发展。

分布式架构：微服务架构（MSA）

James Lewis 和 Martin Fowler 给微服务架构的定义如下：

> 微服务架构风格，以实现一组微服务的方式来开发一个独立的应用系统的方法。其中每个小微服务都运行在自己的进程中，一般采用 HTTP 资源 API 这样轻量的机制相互通信。这些微服务围绕业务功能进行构建，通过全自动化的部署方式来进行独立部署。这些微服务可以使用不同的语言来编写，也可以使用不同的数据存储技术，并且基于最低限度的集中管理。

同时 Martin Fowler 总结有如下 9 个特性：

- 组件以服务的形式提供
- 围绕业务功能进行组织
- 产品而不是项目
- 强化终端与弱化管道
- “去中心化”地治理技术署
- “去中心化”地管理数据
- “基础设施”自动化
- “容错”设计
- “演进式”设计

微服务架构可以看作是一种 SOA 的发展实现，其将 SOA 中原本可能聚合在同一个系统内的多个服务组件拆分到各自独立的系统进程。

微服务架构的优势在于：

- 更加彻底的组件化，系统内部各个组件之间解耦的比较干脆，单个系统的规模小很多；
- 可以组建每个服务独立的维护团队，利于各自团队独立的开发和维护；
- 每个微服务独立部署，只要服务间的接口稳定，各系统可以相互之间互不干扰的独立发展；
- 微服务架构使得每个服务本身可以独立的扩展，性能出现瓶颈，优化或增加这个服务的配置即可。

微服务架构的劣势也很明显，拆分的过细则要求自动化测试能力必须跟得上。一个全流程的测试跨8~10个系统是所有测试人员的恶魔。

问题的排查，数据的一致性保障，系统的监控等等，都会因为拆分的太细，复杂度大幅度增加。如果代码或设计上有一个修改涉及到多个不同的微服务，那么在团队之间协调配合成本也会增加。对旧系统的微服务架构和组件化改造也是一个比较大的问题。

在组件化上所做的任何工作的成功度，取决于软件与组件的匹配程度。但是合理的组件边界应该如何确定，这是非常困难的，演进式的处理方式是我们觉得比较合理和靠谱的。因此，Martin Fowler 也建议，不要一上来就以微服务架构作为系统设计的起点。相反地，要用一个单块系统作为起点，并保持其模块化。当这个单块系统出现了问题后，再将其分解为微服务。