目录

[一：概述 2](#_Toc385593137)

[二：软件架构 2](#_Toc385593138)

[1：运行时抽象 2](#_Toc385593139)

[2：架构元素 3](#_Toc385593140)

[3：组件 3](#_Toc385593141)

[4：连接器 3](#_Toc385593142)

[5：数据 4](#_Toc385593143)

[6：配置 4](#_Toc385593144)

[7：架构属性 4](#_Toc385593145)

[8：设计哲学 4](#_Toc385593146)

[9：架构风格 4](#_Toc385593147)

[三：基于网络应用的架构 5](#_Toc385593148)

[3.1：评估应用软件架构的设计 5](#_Toc385593149)

[3.2：性能 5](#_Toc385593150)

[3.3：网络性能 6](#_Toc385593151)

[3.4：网络效率 7](#_Toc385593152)

[3.5：可伸缩性 7](#_Toc385593153)

[3.6：可进化性 7](#_Toc385593154)

[3.7：可配置性 8](#_Toc385593155)

[3.8：可见性 8](#_Toc385593156)

[四：基于网络应用的架构风格 9](#_Toc385593157)

[4.1：分类学习法 9](#_Toc385593158)

[4.2：分层风格 9](#_Toc385593159)

# 一：概述

1：软件架构方面的研究探索的是如何以最佳的方式划分一个系统、如何标识组件、组件之间如何通信、信息如何表达、系统中的元素如何独立地进化，以及上述所有内容如何使用形式化的和非形式化的符号加以描述

2：软件方面的研究长期以来关注于对软件设计进行分类和开发设计方法学，但是却很少能够客观地评估不同的设计选择对于系统行为的影响。网络方面的研究则恰恰相反，聚焦于系统之间普通通信行为的细节和提高特殊通信技术的性能，却常常忽略了一个事实，即改变一个应用的交互风格（the interaction style）对于性能产生的影响要比改变交互所使用的通信协议更大。

# 二：软件架构

1：运行时抽象

软件架构是一个软件系统在其运行过程中某个阶段的运行时元素（run-time

elements）的抽象。一个系统可能由很多层抽象和很多个运行阶段组成，每一个

抽象和运行阶段都有自己的软件架构。

架构代表了在某个层次上系统行为的抽象，架构元素（architectural elements）可以通过自身提供给同一层其他元素的抽象接口来描述。在每一个架构元素之中，可能还存在着另一个架构，由其定义由其众多子元素构成的系统，该系统实现了由父元素的抽象接口所展示的行为。这样的架构可以递归下去直到最基本的系统元素：不能再被分解为抽象层次更低的元素

2：架构元素

软件架构是由一些架构元素（组件、连接器和数据）的配置来定义的，这些元素

之间的关系受到约束，以获得所期待的一组架构属性。

处理元素（processing elements）是执行数据转换的元素，

数据元素（data elements）是包含被使用和被转换的信息的元素，

连接元素（connecting elements）是将架构的不同部分结合在一起的粘合剂。

我将使用更加流行的术语：组件（components）和连接器（connectors）来分别表示处理元素和连接元素。

3：组件

组件是软件指令和内部状态的抽象单元，通过其接口提供数据的转换能力。

4：连接器

连接器是对于组件之间的通讯、配合或者协作进行中间斡旋的一种抽象机制

也许理解连接器的最佳方式是将它们与组件加以对比。连接器将数据元素从它的一个接口移交（transferring）到另一个接口而不改变数据，以此来支持组件之间的通信。在其内部，连接器可以包含一个由组件组成的子系统，为了移交的目的对数据进行某种转换、执行移交、然后做相反的转换并交付与原始数据相同的结果。然而，在架构所能捕获到的外部行为抽象中，可以忽略这些细节。与之相反，从外部的角度观察，组件可以（尽管并非总会这样）对数据进行转换。

5：数据

数据是组件通过连接器接收或发送的信息元素。

6：配置

配置是在系统运行期间的组件、连接器和数据之间的架构关系的结构。

7：架构属性

软件架构的架构属性集合包括了对组件、连接器和数据的选择和排列所产生的所有属性。架构属性的例子包括了由系统获得的功能属性（functional properties achieved by the system）以及非功能属性（例如，进化的容易程度、组件的可重用性、效率、动态扩展能力，这些常常被称作品质属性）

8：设计哲学

作为世界上的一个元素，每一种模式都是在特定环境中的一种关系，是在那个环境中重

复出现的一种特定的力学系统。存在一种特定的空间配置，允许系统中的这些力量（forces）

为自己求解，以相互支持，最终达到一种稳定的状态。

作为模式语言中的一个元素，模式代表的是一种指导，展示出在某种有意义的环境中，

如何能够一次又一次地重复使用这种空间配置，来为特定的力学系统求解。

模式，简而言之，既是在世界上出现的一个事物，又是一种规则，这种规则告诉我们如

何创建这个事物，以及在何时必须要创建它。模式既是一个过程也是一个事物；既是对一个

活着的事物的描述，也是对生成这个事物的过程的描述

9：架构风格

架构风格是一组相互协作的架构约束，这些架构约束限制了架构元素的角色和功能，以及在任何一个遵循该架构风格的架构中允许存在的元素之间的关系。

在很多方面，与面向对象编程语言（OOPL）研究中的设计模式相比，Alexander 的模式实际上与软件架构风格拥有的共同点更多。一种架构风格，作为一组相互协作的架构约束，

应用于一个设计空间之上，以求促使系统出现所期待的架构属性。通过应用一种架构风格，

架构师通过区分不同的软件设计空间，希望结果能够更好地匹配应用中所固有的一些必须满

足的先决条件，这会导致系统的行为增强了自然的模式（natural pattern，译者注：可以与上文说的“自然地产生”关联起来），而不是与之相冲突（译者注：这段话非常符合老子“道法自然”的思想）。

# 三：基于网络应用的架构

3.1：评估应用软件架构的设计

在一棵架构风格的继承树中对架构属性所进行的评估，是特定于一个特殊应用领域之需

求的，因为特定架构约束的影响常常取决于应用的特性。例如，当我们将管道和过滤器架构风格用于要求在组件之间进行数据转换的系统中时，会产生一些积极的架构属性；但是当系统仅仅由控制消息（control messages）组成时，它只会增加系统的开销（overhead），而不会带来任何好处。因为跨不同的应用领域对架构设计进行比较用处不大，因此确保一致性的最简单的方法是使这棵继承树特定于某个领域。

3.2：性能

聚焦于基于网络应用的架构风格的主要原因之一，是因为组件交互（component

interactions）对于用户感知的性能（user-perceived performance）和网络效率（network efficiency）来说是一个决定性因素。由于架构风格影响到这些交互的特性，选择合适的架构风格能够决定基于网络应用部署的成败。

基于网络应用的性能首先取决于应用的需求，然后是所选择的交互风格，接下来是实现

架构（the realized architecture），最后是每个组件的实现。换句话说，应用软件无法回避为了实现该软件的需求而付出的基本成本；例如，如果应用软件需要数据位于系统A，并由系统B 来处理，那么该软件无法避免将数据从系统A 移动到系统B。同样地，架构无法超越其交互风格所允许的最高效率。例如，将数据从系统A 移动到系统B 的多次交互的成本不可能少于单独一次从系统A 到系统B 的交互。最后，无论架构的质量如何，交互的速度再快，也不可能比组件生产数据加上接收者消费数据所需要的总时间更快。

3.3：网络性能

网络性能这个度量手段用来描述通信的某些属性。

吞吐量（throughput）是信息（既包括应用的数据也包括通信的开销）在组件之间移交的速率。

开销（overhead）可分为初始化开销（initial setup overhead）和每次交互（都会产生的）开销（per-interaction overhead），这种区分有助于识别出能够跨多次交互共享（分摊amortization）初始化开销的连接器。

带宽（bandwidth）是在特定网络连接上可用的最大吞吐量。

可用带宽（usable bandwidth）是指应用实际可用的那部分带宽*。*

架构风格对于网络性能的影响是通过影响每个用户动作的交互数量和数据元素的粒度来实现的。鼓励小型的、强类型（strongly typed）的交互的架构风格，对于仅需在已知组件之间移交小型数据（small data transfers among known components）的应用来说会很有效率，但是会在需要移交大型数据或协商接口（large data transfer or negotiated interfaces）的应用中导致过多的开销。同样地，一种需要通过多个组件之间协作来过滤大型数据流的架构风格，在主要需要小型控制消息（small control messages）的应用中也会显得不合时宜。

3.4：网络效率

各种性能问题的影响常常与应用的分布范围有关。在局部情况（local conditions）下一种架构风格所具有的优点，在面对全局情况（global conditions）时也许却会变成缺点。因此架构风格的属性必须受到与交互距离相关的限制：在单个进程中还是在单个机器上的多个进程之间、在一个区域网（LAN）内还是分布在广域网（WAN）上。当交互跨越广域网时，还会有一个明显的额外关注点：是只涉及一个组织的交互，还是涉及多个组织的信任边界（trustboundaries）的跨互联网交互。

3.5：可伸缩性

可伸缩性表示通过主动的配置（within an active configuration），一个架构支持大量的组件或大量组件之间的交互的能力。我们能够通过以下方法来改善可伸缩性：简化组件、将服务分布到很多组件（对交互去中心化）、以及根据监视获得的信息对交互和配置加以控制。架构风格可以通过确定应用状态的位置、分布的范围以及组件之间的耦合度，来影响上述这些因素。

可伸缩性还受到以下几个因素的影响：交互的频率、组件负载（the load on a

component）随时间的分布是平均的还是存在峰值、交互是保证送达（guaranteed delivery）还是只需要尽量送达（best-effort）、一个请求是否包括同步或异步处理、以及环境是受控的还是无法控制的（即，你是否可以信任其他组件？）

3.6：可进化性

可进化性代表了能够改变一个组件实现而不会对其他组件产生负面影响的程度。组件的静态进化（static evolution）通常依赖于其实现对架构抽象的增强程度，因此这并非是任何特定架构风格所独有的。然而，如果在某种架构风格中包括了关于维护（maintenance）和应用状态位置（location of application state）的架构约束，该架构风格会对动态进化（dynamic evolution）产生影响。在分布式系统中用来从局部故障状况中恢复（recover from partial failure conditions）的相同技术[133] 也能够用于支持动态进化。

3.7：可配置性

可配置性既与可扩展性有关，也与可重用性有关，因为它是指对于组件或者组件的配置在部署之后做修改（post-deployment modification）的能力，这样组件能够使用新的服务或者新的数据元素类型。例如，管道和过滤器和按需代码两种架构风格可以分别为组件及其配置产生可配置性。

3.8：可见性

通过通用性架构约束来限制交互的接口（restricting interfaces via generality），或者提供访问功能以便于监视（providing access to monitoring），架构风格也能够影响基于网络应用中交互的可见性。在这种情况下，可见性是指一个组件对于其他两个组件之间的交互进行监视或进行中间斡旋（monitor or mediate）的能力。拥有了可见性之后，就能够通过多个交互共享的缓存来改善性能、通过分层服务来改善可伸缩性、通过反射式监视（reflective monitoring）来改善可靠性、通过允许中间组件（例如，网络防火墙）对交互做检查来改善安全性。缺乏可见性可能会导致安全问题，移动代理（mobile agent）架构风格就是一个例子。

# 四：基于网络应用的架构风格

4.1：分类学习法

建造软件的目的，是为了创造出满足或超出应用需求的系统，而不是为了创造出一种特

殊的交互拓扑或者使用一种特殊的组件类型。为系统设计所选择的架构风格，必须与这些需

求相一致，而不是相抵触。因此，应当基于这些架构风格所产生的架构属性来对架构风格进

行分类，这样才能提供有用的设计指导

4.2：分层风格



4.2.1客户-服务器（Client-Server，CS）

对于基于网络应用而言，客户-服务器风格是最为常见的的架构风格。服务器组件提供了一组服务，并监听对这些服务的请求。客户组件通过一个连接器将请求发送给服务器，希望执行一个服务。服务器可以拒绝这个请求，也可以执行这个请求并将响应发送给客户。Sinha[123]和Umar [131]对多种客户-服务器系统进行了调查。

Andrews [6]是这样描述客户-服务器组件的：客户是一个触发进程（triggering process）；服务器是一个反应进程（reactive process）。客户发送请求触发服务器作出反应。这样，客户可以自由选择启动活动的时间；然后它常常会等待，直到服务器完成对请求的处理。另一方面，服务器等待接收请求，并对请求作出反应。服务器通常是一个永不终止的进程，并且常常为多个客户提供服务。

分离关注点是在客户-服务器约束背后的原则。功能的适当分离会简化服务器组件，从而提高可伸缩性。这种简化所采用的形式通常是将所有的用户界面功能（user interfacefunctionality）移到客户组件中。只要通信的接口不发生改变，这种分离允许两种类型的组件独立地进化。

客户-服务器的基本形式并不限制应用状态在客户组件和服务器组件之间如何划分。这常常由连接器实现所使用的机制来负责，例如，远程过程调用（remote procedure call）[23]或者面向消息的中间件（message-oriented middleware）

4.2.2：分层系统（**Layered System**，**LS**）和分层**-**客户**-**服务器（**Layered-Client-Server**，**LCS**）

一个分层系统是按照层次来组织的，每一层为在其之上的层提供服务，并且使用在其之下的层所提供的服务[53]。尽管分层系统被看作一种“单纯”的架构风格，但是在基于网络的系统中，对分层系统的使用仅限于与客户-服务器风格相结合，形成分层-客户-服务器风格。

内部层对相邻外部层之外的所有层而言，是被隐藏起来的。通过这样做，分层系统减少了跨越多层的耦合，从而改善了可进化性和可重用性。分层系统的例子包括分层通信协议的处理，例如，TCP/IP 和OSI 协议栈[138]，以及硬件接口库。分层系统的主要缺点是它们增加了处理数据的开销和延迟，降低了用户感知的性能[32]。

分层-客户-服务器风格在客户-服务器风格的基础上添加了代理（proxy）组件和网关（gateway）组件。对客户组件而言，代理组件是一个共享服务器（a shared server），它接收请求并进行可能的转换后将其转发给服务器。网关组件在客户组件或代理组件看来像是一个正常的服务器，但是事实上它将请求进行可能的转换后转发给了它的“内部层”服务器（innerlayer servers）。这些额外的居间斡旋组件为系统添加了多个层，可以用来实现诸如负载均衡和安全性检查这样的功能。

基于分层-客户-服务器风格的架构在信息系统文献[131]中常常被称为两层、三层或者多层架构。

LCS 风格也可以作为在大规模分布式系统中管理标识信息（managing identity）的一种解决方案，在这样的系统中了解所有服务器的完整信息是代价高昂的。相反，服务器被组织为多个层次，这样那些很少被用到的服务可以由中间组件来处理，而不是直接由每个客户组件来处理[6]

4.2.3：客户-无状态-服务器（Client-Stateless-Server，CSS）

客户-无状态-服务器风格源自客户-服务器风格，并且添加了额外的约束：在服务器组件

之上不允许有会话状态（session state）。从客户发给服务器的每个请求中，都必须包含理解请求所必需的全部信息，不能利用任何保存在服务器上的上下文（context），会话状态应全部保存在客户端。

这些约束改善了可见性、可靠性和可伸缩性等三个架构属性。改善了可见性是因为监视

系统再也不必为了确定请求的全部性质而查看多个请求的数据。改善了可靠性是因为这些约

束使得从部分故障中恢复更加简单[133]。改善了可伸缩性是因为不必保存多个请求之间的

状态，允许服务器组件迅速释放资源并进一步简化其实现。

客户-无状态-服务器风格的缺点是：因为我们不能将状态数据保存在服务器上的共享上下文中，增加了在一系列请求中发送的重复数据（每次交互的开销），这样做可能会降低络

性能。

4.2.4：客户-缓存-无状态-服务器（Client-Cache-Stateless-Server，C$SS）

客户-缓存-无状态-服务器风格来源于客户-无状态-服务器风格和缓存风格（通过添加缓存组件）。缓存在客户和服务器之间进行中间斡旋：它能够重用早先请求的响应（如果缓存组件认为它们是可缓存的），以响应稍后的相同请求，如果将该请求转发到服务器，得到的响应可能与缓存中已有的响应相同。有效地利用此架构风格的范例系统是Sun 微系统公司的NFS[115]。

添加缓存组件的好处是，它们有可能部分或全部消除一些交互，从而提高效率和用户感知的性能。

4.2.5：分布式系统和编程范例

Andrews[6]调查了分布式程序中的过程（processes in a distributed program ）如何通过消息传递来进行交互。他定义了并发程序、分布式程序、分布式程序中的各种过程（过滤器、客户、服务器、对等体（peers））、交互范例（interaction paradigms）、以及通信频道。交互范例代表了软件架构风格中与通信相关的方面。他描述了通过过滤器（管道和过滤器）网络的单向数据流、客户-服务器、心跳（heartbeat）检测、探测/回应（probe/echo）、广播、标记传递（token passing）、复制服务器（replicated server）、以及带有任务包（bag of tasks）的复制工作者（replicated worker）。然而，他是从在单个任务（a single task）上协作的多个过程的观点来进行表述，而不是通用的基于网络的架构风格。

Sullivan 和Notkin[126]调查了与隐式调用相关的研究（implicit invocation research），并且描述了对隐式调用的应用，以改善软件工具套件的可进化性品质（evolution quality）。Barrett等人[8]通过建造一个用来进行比较的框架，然后查看某些系统如何符合此框架，调查了基于事件的集成机制。Rosenblum 和Wolf[114]调查了一个用于互联网规模的事件通知（Internet-scale event notification）的设计框架。所有这些都是与EBI 风格的范围和需求相关的，而没有为基于网络的系统提供解决方案。