OpenDoc Series'

架构风格与基于网络的软件架构设计

（博士论文）

作者：Roy Thomas Fielding 博士

译者：李锟、廖志刚、刘丹、杨光

原文链接：Architectural Styles and the Design of Network-based Software Architectures

作者简介：

Roy Thomas Fielding 博士是 HTTP 和 URI 等 Web 架构标准的主要设计者，Apache HTTP 服务器的主要开发者。他为 Web 架构的设计作出极其杰出的贡献，他的工作为 Web 架构奠定了坚实的基础。

译者简介：

李锟（ajaxcn.org 网站站长）、廖志刚（91yee 翻译社区负责人）、刘丹（Matrix 技术社区负责人）、杨光（《重构与模式》的译者）

致谢：

本论文是 Web 发展史上一篇非常重要的技术文献。出于社会责任感，译者认为极其有必要将它介绍给国人，使国人得以一窥 HTTP 和 URI 等 Web 架构标准背后的基本原理。基于相同的基本原理，Web 开发者能够设计并建造出最为高效的 Web 应用。因此译者发起了这一公益性的翻译项目。除了四位主要的译者以外，来自新浪公司的田乐、丁舜佳和梁晓星也参与了第 3 章的部分翻译工作，对他们的辛勤工作表示感谢。此外，国内一些专家认真地审阅了论文的译文，提出了很多中肯的评论，使得译文的质量得以保证。在此向他们表示诚挚的感谢，他们是：庄表伟、李琳骁、金尹、孟岩、骆古道、范凯、刘新生、刘江。

内容目录

论文摘要

架构风格与基于网络的软件架构设计

作者：Roy Thomas Fielding
信息与计算机科学博士
加州大学欧文分校，2000 年
博士论文答辩委员会主席：Richard N. Taylor 教授

万维网（World Wide Web）的成功，很大程度上是因为其软件架构的设计满足了 Internet 规模（Internet-scale）的分布式超媒体系统的需求。在过去的 10 年间，通过对定义 Web 架构的标准所做的一系列修改，Web 以迭代的方式不断地发展着。为了识别出 Web 需要改善的那些方面，并且避免对其进行不想要的修改，必需要有一种现代 Web 架构的模型，用来指导 Web 的设计、定义和部署。

软件架构的研究探索了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、信息如何沟通、系统的元素如何能够独立地进化，以及上述的所有东西如何能够使用形式化的和非形式化的符号加以描述。我的工作的动机是希望理解和评估基于网络的应用的架构设计，通过有原则地使用架构约束，从而从架构中获得所希望的功能、性能和社会学几方面的属性。一种架构风格是一组已命名的、协作的架构约束。

这篇论文定义了一个框架，致力于通过架构风格来理解软件架构，并且展示如何使用风格来指导基于网络的应用的架构设计。本文使用了一个对基于网络的应用的架构风格的调查，根据不同的风格在分布式超媒体的架构中所导致的架构属性，来对这些风格进行分类。然后我介绍了表述性状态转移（Representational State Transfer，REST）的架构风格，并且描述了如何使用 REST 来指导现代 Web 架构的设计和开发。

REST 强调组件交互的可伸缩性、接口的通用性、组件的独立部署、以及用来减少交互延迟、增强安全性、封装遗留系统的中间组件（intermediary components）。我描述了指导 REST 的软件工程原则和选择用来支持这些原则的交互约束，并将它们与其他架构风格的约束进行了对比。最后，我描述了从在超文本转移协议（HTTP）和统一资源标识符（URI）的标准中应用 REST，以及从这两个标准在 Web 客户端和服务器软件的后续部署等过程中学到的经验教训。

绪论

抱歉……你说的可是“屠刀”？
——摘自《建筑师讽刺剧》（The Architects Sketch）[111]

正如 Perry 和 Wolf 的预言，软件架构成为了 20 世纪 90 年代软件工程研究的焦点。由于现代软件系统的复杂性，更加有必要强调组件化的系统，其实现被划分为独立的组件，这些组件通过相互通信来执行想要完成的任务。软件架构的研究探索了如何以最佳方式划分一个系统、如何标识组件、组件之间如何通信、信息如何沟通、组成系统的元素如何能够独立地进化，以及上述的所有东西如何能够使用形式化的和非形式化的符号加以描述。

一个优秀的架构并非凭空想象。所有架构级的设计决策应该根据被设计系统的功能、行为和社会学等方面的需求来作出，这是一个原则，既适用于软件架构，同样也适用于传统的建筑架构领域。“形式追随功能”的指导方针来自从数百年失败的建筑项目中获得的经验，但是却常常被软件从业者忽视。上面引用的那句滑稽搞笑的话来自于 Monty Python 系列讽刺剧，这是当一个建筑师在面对设计一个城市公寓区的目标时，头脑里所抱有的荒诞想法。他想使用所有的现代屠宰场的组成部分来完成这个设计。这也许是他所构思过的最棒的屠宰场，但是对于预期的房客来说却谈不上舒适，他们不得不战战兢兢地行穿行在安装着旋转刀头的走廊中。

《建筑师讽刺剧》里的夸张说法也许看似荒唐可笑，但是考虑到我们是如此频繁地看到软件项目一开始就采用最新时髦的架构设计，到后来却发现系统的需求实际上并不需要这样一种架构。design-by-buzzword（按照时髦的词汇来做设计）是一种常见的现象。至少在软件行业中，很多此类行为是由于对一组特定的架构约束为什么是有用的缺乏理解。换句话说，当选择那些优秀的软件架构来重用时，这些架构背后的推理过程（reasoning），对于设计者来说并非是显而易见的。

这篇论文探索了在计算机科学的两个研究学科（软件和网络）边界上的连接点。软件研究长期以来关注软件设计的分类和对于设计方法学的开发，但是极少能够客观地评估不同的设计选择对于系统行为的影响。网络研究则恰恰相反，集中于系统之间普通的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。我的工作的动机是希望理解和评估基于网络的应用的架构设计，通过有原则地使用架构约束，从而从架构中获得所希望的功能、性能和社会学几方面的属性。当给定一个名称时，一组协作的架构约束就成为了一种架构风格。

这篇论文的前三章定义了一个通过架构风格来理解软件架构的框架，揭示了架构风格如何能够被用来指导基于网络的应用的架构设计。当将常见的架构风格应用于基于网络的超媒体的架构时，将会导致一系列架构属性，根据这些架构属性来对架构风格进行调查和分类。然后使用得到的分类来识别出一组能够改善早期万维网的架构的架构约束。

如同我们在第 4 章中所讨论的，设计 Web 的架构就必须要理解 Web 的需求。Web 是旨在成为一个 Internet 规模的分布式超媒体系统，这意味着它的内涵远远不只仅仅是地理上的分布。Internet 是跨越组织边界互相连接的信息网络。信息服务的提供商必须有能力应对无法控制（anarchic）的可伸缩性的需求和软件组件的独立部署。通过将动作控制（action controls）内嵌在从远程站点获取到的信息的表述之中，分布式超媒体为访问服务提供了一种统一的方法。因此 Web 的架构必须在如下环境中进行设计，即跨越高延迟的网络和多个可信任的边界，以大粒度的（large-grain）数据对象进行通信。

第 5 章介绍并详细描述了为分布式超媒体系统设计的表述性状态转移（REST）的架构风格。REST 提供了一组架构约束，当作为一个整体来应用时，强调组件交互的可伸缩性、

接口的通用性、组件的独立部署、以及用来减少交互延迟、增强安全性、封装遗留系统的中间组件。我描述了指导 REST 的软件工程原则和选择用来支持这些原则的交互约束，并将它们与其他架构风格的约束进行了对比。

如第 6 章中所展示的那样，在过去的 6 年间，REST 架构风格被用来指导现代 Web 的架构的设计和开发。这个工作与我所创作的超文本转移协议（HTTP）和统一资源标识符（URI）的 Internet 标准共同完成，这两个规范定义了被所有 Web 之上的组件交互所使用的通用的接口。

就像大多数真实世界中的系统一样，并非所有已部署的 Web 架构的组件都服从其架构设计中给出的每一个约束。REST 既被用来作为定义架构改进的一种方法，也被用来作为识别架构不匹配（mismatch）的一种方法。当由于无知或者疏忽，一个软件实现以违反架构约束的方式来部署时，就会发生不匹配。尽管不匹配通常无法避免，但是有可能在它们定型之前识别出它们。在第 6 章中总结了几种在现代 Web 架构中的不匹配情况，并且对它们为何会出现和它们如何偏离 REST 进行了分析。

概括来说，这篇论文对于 Internet 和计算机科学领域的软件研究作出了如下贡献：

- 一个通过架构风格来理解软件架构的框架，包括了一组相容的术语，用来描述软件架构；
- 通过当某种架构风格被应用于一个分布式超媒体系统的架构时，它将导致的架构属性，来对基于网络的应用的架构风格进行分类。
- REST，一种新颖的分布式超媒体系统的架构风格；以及
- 在设计和部署现代万维网的架构的过程中，应用和评估 REST 架构风格。

第 1 章 软件架构

尽管软件架构作为一个研究领域吸引了很多人的兴趣，但是对于什么应该被纳入架构的定义，研究者们几乎从未达成过共识。在很多情况下，这导致了在过去的研究中忽视了架构设计的一些重要的方面。本章在检查文献中现存的定义和我自己在基于网络的应用的架构方面的洞察力的基础上，定义了一套自相容（self-consistent）的软件架构术语。每个定义使用方框突出显示，随后讨论该定义如何得来，或者与相关研究进行比较。

1.1 运行时抽象

一个**软件架构**是一个软件系统在其操作的某个阶段的运行时（run-time）元素的抽象。一个系统可能由很多层抽象和很多个操作阶段组成，每个抽象和操作阶段都有自己的软件架构。

软件架构的核心是抽象原则：通过封装来隐藏系统的一些细节，从而更好地识别和支持系统的属性[117]。一个复杂的系统包含有多层的抽象，每一层抽象都有自己的架构。架构代表了在某个层次上系统行为的抽象，架构的元素被描述为提供给同层的其他元素的抽象接口[9]。在每一个元素之中，也可能还存在着另一个架构，定义了子元素的系统，这个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：它们不能再被分解为抽象层次更低的元素。

除了架构的层次，软件系统通常拥有多个操作阶段，例如启动、初始化、正常处理、重新初始化和停止。每个操作阶段都有自己的架构。例如，配置文件在启动阶段会被当作架构的一个数据元素来处理，但是在正常处理阶段则不会当作一个架构元素，因为在这个阶段这些信息已经分布到了系统中的各处。事实上，配置文件也有可能定义了正常处理阶段的架构。系统架构的一个整体描述必须既能够描述各个阶段的系统架构的行为，也能够描述在各个阶段之间的架构的迁移。

Perry 和 Wolf [105]将处理元素定义为“数据的转换”，而 Shaw 等人[118]则将组件描述为“计算和状态的所在地”。Shaw 和 Clements [122] 进一步指出：“组件是在运行时执行某种功能的软件单元。这样的例子有程序、对象、进程、过滤器。”这引出了软件架构（software architecture）和通常所说的软件结构（software structure）之间的一个重要的区别：软件架构是软件系统在运行时的抽象，而软件结构则是静态源代码的属性。尽管将源代码的模块化结构与正在运行的系统中的行为部件对应起来是有好处的，使用相同的代码部分（例如共享库）来实现独立的软件组件也有很多好处。我们将软件架构和源代码结构分离开来是为了更好的关注软件运行时的特性，这些特性不依赖于一个特定的组件实现。因此，尽管架构的设计和源代码结构的设计关系密切，它们其实是分离的设计活动。不幸的是，有些软件架构的描述并没有明确指出这个区别（例如[9]）。

1.2 元素

一个**软件架构**由一些架构元素（组件、连接器和数据）的配置来定义，这些元素之间的关系受到约束，以获得想要得到的一组架构属性。

Perry 和 Wolf [105] 对软件架构的范围和知识基础进行了全面的检查，他们提出了一个模型，将软件架构定义为一组架构元素，这些元素具有通过一组基本原理（rationale）来描述的特殊形式。架构元素包括处理、数据、连接元素。形式则由元素的属性和元素之间的关

系（即元素之上的约束）来定义。这些基本原理通过捕获选择架构风格、选择元素和形式的动机 ，为架构提供了底层的基础。

我的软件架构定义是在 Perry 和 Wolf [105]的模型基础之上的一个更加详尽的版本，但是不包括基本原理部分。尽管基本原理是软件架构研究中很重要的一个方面，尤其是在架构的描述方面，但是将它包括在软件架构的定义中，将会暗示设计文档是运行时系统的一部分。是否包括基本原理能够影响一个架构的开发，但是架构一旦建成，它将脱离其所基于的原理而独立存在。反射型的系统[80]能够根据过去的性能改变今后的行为，但是这样做是用一个更低层次的架构替换另一个更低层次的架构，而不是在那些架构中包含基本原理。

用一个类比来说明，想象一下当一个大楼的蓝图和设计计划图被烧毁了将会发生什么事情？大楼会瞬间倒塌么？不会，因为支撑着楼顶的墙体仍然保持完好无损。按照设计，一个架构拥有一组属性，允许该架构满足甚至超出系统的需求。忽视这些属性，在将来的修改中可能会违反架构的约束，就好像用一面大型窗户取代承重墙会破坏大楼结构的稳定性一样。所以，我们的软件架构定义中没有包括基本原理，而是包括了架构的属性。基本原理说明了这些属性，缺少基本原理可能会导致架构随时间的推移逐渐退化，但是基本原理本身并不是架构的一部分。

Perry 和 Wolf [105]的模型中的一个关键特征是不同类型的元素之间的区别。*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。

Garlan 和 Shaw [53]将系统的架构描述为一些计算组件和这些组件之间的交互（连接器）。这一模型是对于 Shaw 等人的模型[118]的扩展：一个软件系统的架构按照一些组件和这些组件之间的交互来定义。除了指出了系统的结构和拓扑以外，架构还显示出了想要获得的系统需求和构建系统的元素之间的对应关系。更加详细的定义可以在 Shaw 和 Garlan [121]的文章中找到。

Shaw 等人的模型的令人惊讶之处，是他们将软件架构的描述当作成是架构本身，而不是将软件的架构定义为存在于软件之中。在这个过程中，软件架构被简化为通常在大多数非形式化的架构图表中能够看到的东西：方框（组件）和直线（连接器）。数据元素和其他很多真实软件架构的动态方面都被忽略了。这样的一个模型是不足以描述基于网络的软件架构的，因为对于基于网络的应用而言，数据元素在系统中的位置和移动常常是系统行为唯一至关重要的决定因素。

1.2.1 组件

> 一个**组件**是软件指令和内部状态的一个抽象单元，通过其接口提供对于数据的转换。

组件是软件架构中最容易被识别出来的方面。在 Perry 和 Wolf [105]的定义中，处理元素被定义为提供对于数据元素的转换的组件。Garlan 和 Shaw [53]将组件简单描述为执行计算的元素。我们的定义试图更加精确地将组件和连接器之中的软件（software within connectors）区分开来。

组件是软件指令和内部状态的一个抽象单元，通过其接口提供对于数据的转换。转换的例子包括从二级存储将数据加载到内存、执行一些运算、转换为另外一种格式、使用其他数据来封装等等。每个组件的行为是架构的一部分，能够被其他组件观察到（observed）或看到（discerned） [9]。换句话说，组件应该由它为其他组件提供的接口和服务来定义，而不

是由它在接口之后的实现来定义。Parnas [101]将此定义为其他架构元素能够对该组件作出的一组假设。

1.2.2 连接器

> 一个**连接器**是对于组件之间的通讯、协调或者合作进行仲裁的一种抽象机制。

Perry 和 Wolf [105] 将连接元素模糊地描述为将架构的不同部分结合在一起的粘合剂。Shaw 和 Clements [122]提供了一个更加精确的定义：连接器是是对于组件之间的通讯、协调或者合作进行仲裁的一种抽象机制。连接器的例子包括共享的表述、远程过程调用、消息传递协议和数据流。

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

1.2.3 数据

> 一个**数据**是组件通过一个连接器接收或发送的信息元素。

上面已经提到，是否有数据元素是 Perry 和 Wolf [105]所提出的模型与大多数其他软件架构研究所提出的模型[1, 5, 9, 53, 56, 117-122, 128]之间的最大区别。Boasson [24] 批评当前的软件架构研究过于强调组件的结构和架构开发工具，他建议应该把注意力更多地放在以数据为中心的架构建模上。Jackson [67]也有相似的观点。

数据是组件通过一个连接器接收或发送的信息元素。数据的例子包括字节序列、消息、编码过的参数、以及序列化过的对象，但是不包括那些永久驻留或隐藏在组件中的信息。从架构的角度来说，一个“文件”其实是一种转换，文件系统组件从它的接口接收到的一个“文件名”数据，将该数据转换为记录在（隐藏的）内部存储系统中的字节序列。组件也能够生成数据，例如一个时钟或传感器的软件封装。

数据元素在基于网络的应用的架构中是一个无法避免的天性，这往往决定了一个特定的架构风格是否是合适的。在对移动代码设计范例（mobile code design paradigms）的比较中[50]尤其明显，在这个场景中你必须要在两种风格中做出选择：是直接与组件进行交互；还是将组件转换为一个数据元素，通过网络转移，然后转换回一个能够在本地与之交互的组件。不在架构层面上考虑数据元素，是完全不可能评估这样的一个架构的。

1.3 配置

> 一个**配置**是在系统的运行期间组件、连接器和数据之间的架构关系的结构。

Abowd 等人[1]将架构的描述定义为根据三个基本的语义类来对系统进行描述：组件——计算的所在地；连接器——定义组件之间的交互；配置——相互交互的组件和连接器的集合。可以使用多种与特定风格相关的形象化符号来可视化地展示这些概念，便于描述合法的计算和交互、以及想要得到的系统约束。

严格来说，你可能会认为一个配置等价于一组组件交互之上的特定约束。例如，Perry 和 Wolf [105] 在他们的架构形式关系（architectural form ralationships）的定义中包括了拓扑。然而，将主动的拓扑与更加通用的约束分离开，使得一个架构师更容易将主动的配置与所有合法配置可能影响的领域区分开。Medvidovic 和 Taylor [86]给出了在架构描述语言中额外用来对配置进行区分的基本原理。

1.4 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导致的所有属性。架构属性的例子包括了可以由系统获得的功能属性和非功能属性，例如：进化的相对容易程度、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（quality attributes [9]）。

属性是由架构中的一组约束所导致的。约束往往是由在架构元素的某个方面应用软件工程原则[58]来驱动的。例如，*统一管道和过滤器*（uniform pipe-and-filter）风格通过在其组件接口之上应用通用性（generality）原则——强迫组件实现单一的接口类型，从应用中获得了组件的可重用性和可配置性的品质。因此，架构约束是由通用性原则所驱动的“统一组件接口”，目的是获得两个想要得到的品质，当在架构中实现了这种风格时，这两个品质将成为可重用和可配置组件的架构属性。

架构设计的目标是创建一个包含一组架构属性的架构，这些架构属性形成了系统需求的一个超集。不同架构属性的相对重要性取决于想要得到的系统本身的特性。2.3 节检查了那些对于基于网络的应用的架构特别重要的属性。

1.5 风格

> 一种**架构风格**是一组协作的架构约束，这些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风格的架构中允许存在的元素之间的关系。

因为一种架构既包含功能属性又包含非功能属性，直接比较不同类型系统的架构，或者甚至是比较在不同环境中的相同类型的系统会比较困难。风格是一种用来对架构进行分类和定义它们的公共特征[38]的机制。每一种风格都为组件的交互提供了一种抽象，并且通过忽略架构中其余部分的偶然性细节，来捕获一种交互模式（pattern of interation）的本质特征[117]。

Perry 和 Wolf [105]将架构风格定义为对于元素类型（element types）的一种抽象和来自不同的特定架构的形式化方面（formal aspects，也许仅集中在架构的一些特定方面）。一种架构风格封装了关于架构元素的重要决策，强调对于元素和它们之间的关系的重要约束。这个定义允许风格仅仅聚焦于架构的连接器，或者组件接口的一些特定方面。

与之相反，Garlan 和 Shaw [53]、Garlan 等人[56]、Shaw 和 Clements [122]都根据各种类型的组件之间的交互模式来定义风格。明确地说，一种架构风格决定了在此风格的实例中能够使用的组件和连接器的词汇表，以及一组如何能够将它们组合在一起的约束[53]。对于架构风格的这种限制很大的视图是他们的软件架构定义的直接结果——即，将架构看作是一个形式化的描述，而不是一个正在运行的系统，这导致了仅仅基于从包含方框和直线的图表中总结出来的共享模式来进行抽象。Abowd 等人[1]更进一步，将其明确定义为一个约定的集合（collection of conventions），这个集合用来将一套架构描述解释为对于一种架构风格的定义。

新的架构能够被定义为特定风格的实例（instances）[38]。因为架构风格可能强调的是

软件架构的不同方面，一种特定的架构可能是由多种架构风格组成的。同样地，能够通过将多种基本风格组合为单个的协作风格来形成一种混合风格。

一些架构风格常常被描述为适合于所有形式的软件的“银弹”式解决方案。然而，一个好的设计师应该选择一种与正在解决的特定问题最为匹配的风格[119]。为一个基于网络的应用选择正确的架构风格必须要理解该问题领域[67]，因此需要了解应用的通信需求，知道不同的架构风格和它们所导致的特殊问题，并且有能力根据基于网络的通信的特性来预测每种交互风格的敏感度[133]。

不幸的是，使用术语“风格”来表达一组协作的约束常常会令人困惑。这种用法与“风格”一词在词典中的用法有很大的不同，后者强调设计过程的个性化。Loerke [76]专门用了一章来贬低在一个专业建筑师的工作中应该为个性化风格保留位置的想法。相反地，他将风格描述为批评家对于过去架构的视图，即应该由可选择的原料、社区的文化、本地统治者的自尊心等等因素来负责架构的风格，而不是由设计者来负责。换句话说，Loerke 认为在传统的建筑架构中，风格的真正来源是一组应用在设计上的约束，达到或复制一种特定的风格应该是设计者的最低的目标。由于将一组已命名的约束称作一种风格，使得对公共约束的特征进行沟通变得更加容易，我们将架构风格用作一种进行抽象的方法，而不是代表一种个性化的设计。

1.6 模式和模式语言

在进行架构风格的软件工程研究的同时，面向对象编程社区一直在基于对象的（object-based）软件开发领域探索如何使用设计模式和模式语言来描述重复出现的抽象。一种设计模式被定义为一种重要的和重复出现的系统构造。一种模式语言是一个模式的系统，以一种对这些模式的应用加以指导的结构来进行组织[70]。设计模式和模式语言的概念都基于 Alexander 等人的著作[3, 4]中关于建筑架构的内容。

模式的设计空间（design space）包括了特定于面向对象编程技术的实现关注点，例如类继承和接口组合，以及架构风格所带来的高层次的设计问题[51]。在一些情况下，架构风格的描述也被称作架构模式（architectural patterns）[120]。然而，模式的一个主要的优点是它们能够将对象之间的相当复杂的交互协议描述为单个的抽象[91]，其中既包括行为的约束，也包括实现的细节。总的说来，一种模式或由多种模式集成在一起的模式语言能够被看作是实现对象之间的一组预期交互的方法。换句话说，一种模式通过遵循一种固定的设计和实现选择（implementation choices）路径，定义了一个解决问题的过程[34]。

如同软件的架构风格一样，软件模式的研究也偏离了其在建筑架构中的起源。其实，Alexander 的模式概念的核心并非是对于重复出现的架构元素的排列，而是发生在一个空间内重复出现的事件（人类的活动和情绪）的模式。Alexander 还理解到：事件的模式不能脱离于发生这些事件的空间[3]。Alexander 的设计哲学是，识别出目标文化（target culture）中公共的生活模式（pattern of life），确定哪些架构约束对于以下目的是必需的，即，对可以使期望的模式自然地产生的特定空间加以区分。这些模式存在于多个层次的抽象和所有的规模中。

作为世界上的一个元素，每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。并且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，相互支持，达到一种稳定的状态。

作为语言的一个元素，一种模式是一种指导，显示出这种空间配置如何能够一次又一次的被使用，来为特定的力学系统求解，无论在哪里存在着使其产生的环境。

模式，简而言之，既是在世界上存在的一件事物，又是关于如何创建这件事物，以及何时我们必须要创建它的规则。它既是一个过程也是一件事物；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[3]。

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更多的共同点。一种架构风格，作为一组协作的约束，应用于一个设计空间，以求促使一个系统出现所期望的架构属性。通过应用一种风格，一个架构师是在区分不同的软件设计空间，希望结果更好地匹配应用中所固有的一些必需满足的先决条件，这会导致系统行为增强了自然的模式（natural pattern），而不是与之相冲突。

1.7 视图

一种架构视图常常是特定于应用的，并且基于应用的领域而千变万化……我们已经看到架构视图解决了很多的问题，包括：与时间相关的问题（temporal issues）、状态和控制方法、数据表述、事务生命周期、安全保护、峰值要求和优雅降级（graceful degradation）。无疑，除了上述的这些视图，还存在着很多可能的视图。[70]

除了从一个系统中的很多架构，以及组成架构的很多架构风格的角度来观察以外，也有可能从很多不同的角度来观察一个架构。Perry 和 Wolf [105] 描述了三种重要的软件架构视图：处理、数据、连接。处理视图侧重于流过组件的数据流，以及组件之间连接（the connections among the components）的那些与数据相关的方面。数据视图侧重于处理的流程，而不是连接器。连接视图侧重于组件之间的关系和通信的状态。

多种架构视图在特定架构的案例研究中是司空见惯的[9]。一种架构设计方法学，4+1 视图模型[74]，使用了 5 种协作的视图来对软件架构的描述加以组织。每种视图致力于解决一组特定的关注点。

1.8 相关工作

我只在这里包括了那些定义软件架构或者描述软件架构风格的研究领域。其他的软件架构研究领域还包括架构分析技术、架构的恢复与再造（re-engineering）、架构设计的工具和环境、从规范到实现的细化、以及部署软件架构的案例研究[55]。有关风格的分类、分布式过程范例、以及中间件，这些领域的相关工作会在第 3 章中进行讨论。

1.8.1 设计方法学

大部分早期的软件架构研究都集中在设计的方法学（design methodologies）上。例如，面向对象设计[25]提倡以一种结构化的方式来解决问题，能够自然地导致基于对象的架构（或者，更确切地说，不会导致任何其他形式的架构）。最早从架构层面上强调设计的设计方法学之一是 Jackson 系统开发方法（JSD）[30]。JSD 有意将对于问题的分析结构化，这样能够导致一种组合了管道和过滤器（数据流）风格和过程控制约束（process control constraints）风格的架构风格。这些设计方法学通常只会产生一种架构风格。

人们对架构分析和开发的方法学进行了一些初步的研究。Kazman 等人使用 SAAM[68] 和 ATAM[69]的架构权衡（trade-off）分析，通过基于场景的分析对用来识别出架构的方面（architectural aspects）的设计方法进行了描述。Shaw [119]比较了一个汽车导航控制系统的多种不同的方框箭头（box-and-arrow）设计，每一种设计都使用了一种不同的设计方法学并包括了多种架构风格。

1.8.2 设计、设计模式、模式语言手册

在与传统的工程学科保持一致的同时，Shaw[117]提倡对于架构手册的开发。面向对象编程社区率先开发了设计模式的目录，例子包括“四人帮”的书籍[51] 、Coplien 和 Schmidt

[33] 的文章。

软件设计模式比架构风格更加倾向于面向特定的问题（problem-oriented）。Shaw [120]基于在[53]中描述的架构风格，提出了 8 个架构模式的例子，还包括了最适合于每一种架构的问题种类。Buschmann 等人[28] 对架构模式进行了与基于对象开发（object-based development）相同的全面的检查。这两份参考资料都是纯粹描述性的，并没有试图去比较或者展示架构模式之间的区别。

Tepfenhart 和 Cusick [129] 使用了一张二维地图来区分领域分类学、领域模型、架构风格、框架、工具包、设计模式、以及应用软件。在这张拓扑图中，设计模式是预先设计的结构，用作软件架构的的建造块（building block），而架构风格则是用来识别独立于应用领域的一种架构家族的一组操作特征。然而，它们都未能成功地定义架构本身。

1.8.3 参考模型和特定于领域的软件架构

研究者们开发出来了各种参考模型，为描述架构和显示组件之间的相互关系提供了概念框架[117]。OMG[96]开发了对象管理架构（OMA），作为代理式的分布式对象架构（brokered distributed object architectures）的参考实现，详细说明了如何定义和创建对象、客户端应用如何调用对象、如何共享和重用对象。其重点是在分布式对象的管理方面，而不是在应用中的高效的交互方面。

Hayes-Roth 等人[62] 将特定于领域的软件架构（DSSA）定义为由以下部分组成：a) 一种参考架构，为一个重大的应用领域描述了一种通用的概念框架。b) 一个包含了可重用的领域专家知识的组件库。c) 一种应用的配置方法，为满足特殊应用的需求的架构选择和配置组件。Tracz [130] 提供了一个 DSSA 的概要描述。

DSSA 项目通过将软件开发空间限制到一个满足某个领域的需求的特定的架构风格，成功地将架构决策转移到了正在运行的系统之中[88]。DSSA 的例子包括：用于航空电子学的 ADAGE [10]，用于自适应智能系统的 AIS [62]，用于导弹导航、航海、以及控制系统的 MetaH [132]。DSSA 更强调在一个公共的架构领域中的组件的重用，而不是对特定于每一个系统的架构风格的选择。

1.8.4 架构描述语言

最近发布的与软件架构有关的工作大多都是在架构描述语言（ADL）的领域里。根据 Medvidovic 和 Taylor [86]的定义，一种 ADL 是一种为明确说明软件系统的概念架构和对这些概念架构建模提供功能的语言，至少包括以下部分：组件、组件接口、连接器、以及架构配置。

Darwin 是一种声明式（declarative）的语言，它旨在成为一种通用的表示方法，来详细描述由使用不同交互机制的不同组件组成的系统的结构[81]。Darwin 的有趣之处在于，它允许详细描述分布式架构（distributed architectures）和动态组合架构（dynamically composed architectures）[82]。

UniCon [118]是一种语言和相关的工具集，用来将一组受到限制的组件和连接器组合为一个架构。Wright [5] 通过根据交互的协议来指定连接器的类型，为描述架构组件之间的交互的提供了形式化的基础。

如同设计方法学一样，ADL 经常引入一些特定的架构假设，这些假设有可能会影响到该语言描述一些架构风格的能力，并且可能会与存在于现有中间件中的假设相矛盾[38]。在一些情况下，一种 ADL 是专门为单一架构风格而设计的，这样以来就需要以丧失通用性为代价，增强它在专业化的描述和分析方面的能力。例如，C2SADEL[88]是一种为了描述以 C2 风格开发的架构而专门设计的 ADL[128]。与之相反，ACME [57]是一种试图尽量通用的

ADL，但是它所做的权衡就是，它不支持特定于风格的分析和建造实际的应用，而是聚焦于分析工具之间的交换。

1.8.5 形式化的架构模型

Abowd 等人[1]声称，根据少量的一组从架构描述的语法领域（方框直线图）到架构含义的语义领域的映射，就能够形式化地描述架构风格。然而，他们所做的假设是：架构就是描述，而不是对于一个正在运行的系统的一种抽象。

Inverardi 和 Wolf [65] 使用化学抽象机（CHAM）的表达形式，将软件架构元素建模为化学物品，在这些元素之上发生的反应受到了明确说明的规则的控制。它根据组件如何转换可用的数据元素，以及如何使用组合规则将单个转换传播到整个系统结果中，来指定组件的行为。尽管这是一个很有趣的模型，有一点仍然不明确：如何使用 CHAM 来描述任何一种其目的超出了转换一个数据流的架构形式。

Rapide [78]一种专门为定义和模拟系统架构而设计的并行的、基于事件的模拟语言。模拟器生成一组部分排序的（partially-ordered）事件，这些事件能够被用来分析内部连接之上的架构约束的一致性。Le Métayer[75]提出了一种根据图和图的语法来定义架构的表达形式。

1.9 小结

本章检查了本论文的背景。引入并形式化了一组一致的软件架构概念术语，这些术语对于避免出现文献中常见的架构和架构描述之间的混淆来说是必需的，特别是在早期的架构研究中往往未将数据作为一个重要的架构元素。最后我调查了与软件架构和架构风格相关的其他的一些研究。

后面两章通过聚焦于基于网络的应用的架构，来继续我们对于背景材料的讨论，并且描述如何使用风格来指导它们的架构设计，然后使用一种分类方法学（classification methodology）来调查公共的架构风格，这种分类方法学侧重于架构的属性，这些属性是当风格被应用于一个基于网络的超媒体架构时所导致的。

第 2 章 基于网络的应用的架构

本章通过聚焦于基于网络的应用的架构，来继续我们对于背景材料的讨论，并且描述如何使用风格来指导它们的架构设计。

2.1 范围

架构可以存在于软件系统的多个层次上。本论文检查了软件架构最高层次上的抽象，在这里组件之间的交互能够通过网络通信来实现。我们将讨论限制在基于网络的应用的架构的风格上，这样可以缩小要研究的风格之间的差异。

2.1.1 基于网络 vs. 分布式

基于网络的架构（network-based architectures）与软件架构（software architectures）的主要区别通常是：组件之间的通信仅限于消息传递（message passing）[6]或者消息传递的等价物（如果在运行时，基于组件的位置能够选择更加有效的机制的话）[128]。

Tanenbaum 和 van Renesse [127] 是这样来区分分布式系统和基于网络的系统的：分布式系统在用户看来像是普通的集中式系统，但是运行在多个独立的 CPU 之上。相反，基于网络的系统有能力跨越网络运转，但是这一点无需表达为对用户透明的方式。在某些情况下，还希望用户知道一个需要网络请求的动作和一个在他们的本地系统就能满足的动作之间的差别，尤其是当使用网络意味着额外的处理成本的时候[133]。本论文涵盖了基于网络的系统，并不仅限于那些对用户透明的系统。

2.1.2 应用软件 vs. 网络软件

对于本论文范围的另外一个限制是我们将讨论范围局限在对于应用软件的架构的讨论上，不包括操作系统、网络软件和一些仅仅为得到系统支持而使用网络的架构风格 (例如，进程控制风格[53])。应用软件代表的是一个系统的“理解业务”（business-aware）的那部分功能[131]。

应用软件的架构是对于整个系统的一种抽象，其中用户动作的目的可以被表示为一个架构的功能属性。例如，一个超媒体应用必须关注信息页面的位置、处理请求和呈现数据流。这与网络的抽象形成了对比，网络的抽象的目的是将比特从一个地点移动到另一个地点，而并不关心为何要移动这些比特。只有在应用层面上，我们才能够基于一些方面来评估设计的权衡，这些方面包括每个用户动作的交互数量、应用状态的位置、所有数据流的有效吞吐量（与单个数据流的潜在吞吐量相对应）、每个用户动作所执行的通信的广度等等。

2.2 评估应用软件架构的设计

本论文的一个目的是，为选择或创建最适合于一个特定应用领域的架构提供设计指导，请记住架构是架构设计的实现，而非设计本身。一个架构能够根据其运行时的特征来加以评估，但是我们更喜欢能够在不得不实现所有的候选架构设计之前，有一种对于这些候选架构设计的评估机制。不幸的是，众所周知，架构设计难以通过一种客观的方式来加以评估和比较。就像大多数其他创造性设计的产物一样，架构通常被展现为一件已完成的工作，就好像设计是从架构师头脑中完整地流淌出来一样。为了评估一个架构设计，我们需要检查隐藏在系统约束背后的设计基本原理，并且将从那些约束继承而来的属性与目标应用的目的进行比较。

第一个层面的评估由应用的功能需求来设定。例如，针对分布式超媒体系统的需求来对

一个进程控制架构（process control architecture）进行评估是没有意义的，因为如果架构不能正常工作，比较就没有意义。虽然这样会将一些候选架构设计排除在外，但是在大多数情况下还是会剩下很多能够满足应用的功能需求的架构设计。剩下的候选者之间的区别是对非功能需求的强调程度——每个架构都会以不同的程度支持各种识别出的系统所必需的非功能架构属性。因为架构属性是由架构约束所导致的，所以有可能通过以下方式来比较不同的架构：识别出每个架构的约束，评估每个约束所导致的一组属性，并将设计累积的属性与那些应用要求的属性进行比较。

正如上一章中所描述的，一种架构风格是一组协作的架构约束，给它取一个名称是为了便于引用。每个架构设计决策可以被看作是对一种风格的应用。因为添加的一个约束可能是从一种新的风格继承来的，我们可以将所有可能的架构风格的空间看作是一棵继承树，这棵树的根节点是“空”风格（没有约束）。当它们的约束不存在冲突时，多种风格可以进行组合，形成混合风格，最终可以形成一种代表了架构设计的完整抽象的混合风格。因此一个架构设计能够通过将约束的集合分解到一棵继承树的方法来加以分析，并且可以评估由这棵树所代表的约束的累积效果。如果我们理解了每种基本风格所导致的属性，那么遍历这棵继承树可以使我们理解设计的全部的架构属性。然后应用的特定需求就能够与设计的属性相匹配。这样比较不同的设计就变成了一件相当简单的事情：识别出哪些架构设计满足了该应用的大多数想要得到的属性。

必须意识到一种约束的效果可能会抵消一些其他的约束所带来的好处。尽管如此，一个有经验的软件架构师仍然有可能为一个特定应用领域的架构约束建造一棵这样的继承树，然后使用这棵树来评估该领域应用的很多不同的架构设计。这样，建造一棵继承树就为架构设计提供了一种评估机制。

在一棵风格的继承树中对架构属性进行的评估是特定于一个特殊的应用领域的，因为一个特定约束的影响常常取决于应用的特征。例如，管道和过滤器风格当被用在一个要求在组件之间进行数据转换的系统中时，会导致一些积极的架构属性，但是当系统仅仅由控制信息组成时，它只会增加系统的负担，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几乎没有什么用处，因此确保一致性的最简单的方法是使这棵树特定于某个领域。

设计评估往往是一个在不同的权衡之间进行选择的问题，Perry 和 Wolf [105]描述了一种明确地识别权衡的方法，给每个属性加上一个数字权重，表明它在架构中的相对重要性，这样就提供了对候选设计进行比较的规范的度量手段。然而，为了成为一个有意义的度量手段，每个权重要使用一种对于所有属性一致的客观尺度来小心地选取。实际上，这样的尺度是不存在的。与其让架构师不断调整权重直到结果匹配他们的直觉，我更愿意将所有的信息用容易看到的形式展示给架构师，通过视觉上的模式来指导架构师的直觉。这将会在下一章中演示。

2.3 关键关注点的架构属性

本节描述了用来对本论文中的架构风格进行区别和分类的架构属性。它并非想要成为一份全面的清单，我只包括了明显受到所调查的一组有限风格影响的那些属性。还有一些额外的属性，有时候也被称作软件质量（software qualities），在大多数软件工程的教科书中都会涉及到（例如[58]）。Bass 等人[9]检查了与软件架构有关的质量。

2.3.1 性能（Performance）

聚焦于基于网络的应用的风格的主要原因之一，是因为组件交互对于用户可觉察的性能（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一个基于网络的应用部署

的成败。

一个基于网络的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，然后是实现的架构，最后是每个组件的实现。换句话说，应用软件都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 A 移动到 B。同样地，一个架构无法超越其交互模式所允许的效率。例如，将数据从 A 移动到 B 的多次交互的成本不可能少于单独一次从 A 到 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一个组件实现生产数据，它的接收者消费数据所需的总时间更快。

2.3.1.1 网络性能（Network Performance）

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负载）在组件之间转移的速率。*负载*（overhead）可分为初始设置（initial setup）的负载和每次交互的负载，这种区分有助于识别能够跨多个交互共享设置负载（分摊）的连接器。*带宽*（bandwidth）是在一个特定的网络连接之上可用的最大的吞吐量。*可用带宽*（usable bandwidth）是指应用实际可用的那部分带宽。

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的数量和数据元素的粒度来实现的。一种鼓励小型的强类型（strongly typed）交互的风格，对于一个在已知组件之间转移小型数据的应用来说会很有效率，但是会在包括了大型数据转移或协商接口（negotiated interfaces）的应用中导致过多的负载。同样地，一种通过多个组件之间协调来过滤大型数据流的风格，在主要需要小型的控制消息的应用中会显得不合时宜。

2.3.1.2 用户可觉察的性能（User-perceived Performance）

用户可觉察的性能与网络性能的区别是，一个动作的性能是根据其对于使用一个应用的用户的影响来度量，而不是根据网络移动信息的速率来度量。用户可觉察的性能的主要度量手段是延迟和完成时间。

延迟（latency）是指从触发初次请求到得到第一个响应指示之间持续的时间。延迟会发生在基于网络的应用的处理过程的如下几个点上：1) 应用识别出触发动作的事件所需的时间；2) 在组件之间建立交互所需的时间；3) 在组件间传输交互数据所需的时间；4) 组件处理每个交互所需的时间；以及 5) 在应用能够呈现一个可用的结果之前，完成数据的转移和处理交互的结果所需的时间。重要的是要注意到，虽然只有在(3)和(5)中存在着真正的网络通讯，但是以上所有五点均能够受到架构风格的影响。此外，每个用户动作的多个组件交互也会增加延迟，除非它们能够并行处理。

完成时间（completion）是完成一个应用动作所花费的时间。完成时间取决于所有上述的延迟点。动作的完成时间和它的延迟之间的区别在于，延迟代表了一种应用增量处理正在接收的数据的程度。例如，一个能够在接收的同时显示一个大型图片的 Web 浏览器，与等待全部数据接收完之后才显示图片的 Web 浏览器相比，会提供好得多的用户可觉察性能，即使两者都具有相同的网络性能。

重要的是要注意到，对延迟进行优化的设计常常会产生延长完成时间的副作用，反之亦然。例如，如果算法在产生已编码的转换之前，对数据的重要部分进行采样，则对于一段数据流的压缩就能够产生更加有效率的编码。对于跨越网络转移已编码的数据来说，这会导致更短的完成时间。然而，如果压缩是在响应用户动作的过程中以一种即时（on-the-fly）的方式来执行的，在转移之前缓存大型的采样数据会产生不可接受的延迟。平衡这些权衡是很困难的，特别是当不知道接收者是更关心延迟（例如，Web 浏览器）还是更关心完成时间（例如，Web 爬虫）的时候。

2.3.1.3 网络效率（Network Efficiency）

关于基于网络的应用的一个有趣的观察是：最佳的应用性能是通过不使用网络而获得的。这基本上意味着对于一个基于网络的应用，最高效的架构风格是那些在可能的情况下，能够有效地将对于网络的使用减到最少的架构风格。可以通过重用先前的交互（缓存）、减少与用户动作相关的网络交互（复制数据和关闭连接操作）、或者通过将对数据的处理移到距离数据源更近的地方（移动代码）来减少某些交互的必要性。

各种性能问题的影响常常与应用的分布范围有关。一种风格在局部的场合下的优点当面对全局性的场合时也许会变成缺点。因此风格的属性必须受到与交互距离相关的限制：是在单个进程中还是在单个机器上的多个进程之间、是在一个区域网（LAN）内还是分布在广域网（WAN）上。是只涉及到一个组织，还是与跨 Internet 的交互相比，涉及到多个可信任的边界。

2.3.2 可伸缩性（Scalability）

可伸缩性表示在一个主动的配置中，架构支持大量的组件或大量的组件之间交互的能力。可伸缩性能够通过以下方法来改善：简化组件、将服务分布到很多组件（分散交互）、以及作为监视的结果对交互和配置进行控制。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、组件负担的分布是否平均或出现峰值、交互是必需保证送达（guaranteed delivery）还是只需要尽量送达（best-effort）、一个请求是否包括了同步或异步的处理、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？）。

2.3.3 简单性（Simplicity）

通过架构风格来导致简单性属性的主要方法是，对组件之间的功能分配应用分离关注点原则（principle of separation of concerns）。如果功能分配使得单独的组件足够简单，那么它们就更容易被理解和实现。同样地，这样的关注点分离也使得关于整体架构的推理任务变得更加容易。我选择将复杂性（complexity）、可理解性（understandability）和可验证性（verifiability）统一在简单性这个通用的属性中，是因为它们在基于网络的系统中有着密切的关联。

对架构元素应用通用性原则（principle of generality）也有助于提高简单性，因为它减少了架构中的变数。对连接器应用通用性原则就产生了中间件[22]。

2.3.4 可修改性（Modifiability）

可修改性是对于应用的架构所作的修改的容易程度。可修改性能够被进一步分解为在下面所描述的可进化性、可扩展性、可定制性、可配置性和可重用性。基于网络的系统的一个特殊的关注点是动态的可修改性[98]，它要求在对一个已部署的应用做修改时，无需停止和重新启动整个系统。

即使有可能建造一个完美地匹配用户需求的软件系统，那些需求也会随时间发生变化，就像社会的变化一样。因为基于网络的应用中的组件可能跨多个组织边界来分布，系统必须准备好应对逐渐的和片段的修改，一些旧的组件实现将会与一些新的组件实现共存，而不会妨碍新的组件实现使用它们的扩展功能。

2.3.4.1 可进化性（Evolvability）

进化性代表了一个组件实现能够被改变而不会对其他组件产生负面影响的程度。组件的静态进化通常依赖于其实现是否加强了架构的抽象，因此这并非是任何特定架构风格所独有的。然而，动态进化能够受到风格的影响，如果该风格包括了对于维护（maintenance）和应用状态的位置（location of application state）的约束。在分布式系统中用来从局部故障（partial failure）状况中恢复（recover from partial failure conditions）的相同技术 [133] 也能够被用来支持动态进化。

2.3.4.2 可扩展性（Extensibility）

可扩展性被定义为将功能添加到一个系统中的能力[108]。动态可扩展性意味着功能能够被添加到一个已部署的系统中，而不会影响到系统的其他部分。导致可扩展性的方法是在一个架构中减少组件之间的耦合，就像在基于事件的集成（event-based intergration）的例子中展示的那样。

2.3.4.3 可定制性（Customizability）

可定制性是指临时性地规定一个架构元素的行为的能力，然后该元素能够提供一种非常规的服务。一个组件是可定制的，是指一个客户能够扩展该组件的服务，而不会对该组件的其他客户产生影响[50]。支持可定制性的风格也可能会提高简单性和可扩展性，这是因为通过仅仅直接实现最常用的服务，允许客户端来定义不常用的服务，服务组件的尺寸和复杂性将会降低。可定制性是通过远程求值（remote evaluation）风格和按需代码（code-on-demand）风格所导致的一种架构属性。

2.3.4.4 可配置性（Configurability）

可配置性既与可扩展性有关，也与可重用性有关，因为它是指在部署后（post-deployment）对于组件，或者对于组件配置的修改，这样组件能够使用新的服务或者新的数据元素类型。管道和过滤器风格和按需代码风格是两个可以分别为组件配置和组件带来可配置性的例子。

2.3.4.5 可重用性（Reusability）

可重用性是应用的架构的一个属性，如果一个应用的架构中的组件、连接器或数据元素能够在不做修改的情况下在其他应用中重用，那么该架构就具有可重用性。在架构风格中导致可重用性的主要机制是降低组件之间的耦合（对于其他组件的标识的了解）和强制使用通用的组件接口。统一管道和过滤器风格是这种约束（译者注：即，强制使用通用的组件接口）的例子。

2.3.5 可见性（Visibility）

风格也能够通过限制必须使用通用性的接口，或者提供访问监视功能的方法，来影响基于网络的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仲裁的能力。可见性能够通过以下方式改善性能：交互的共享缓存、通过分层服务提供可伸缩性、通过反射式监视（reflective monitoring）提供可靠性、以及通过允许中间组件（例如，网络防火墙）对交互做检查提供安全性。移动代理风格（mobile agent style）是缺乏可见性可能会导致安全问题的一个例子。

这种对于术语“可见性”的使用方法与 Ghezzi 等人[58]的使用方法存在着区别，他们所

说的可见性是对开发过程而言的，而非对产品而言的。

2.3.6 可移植性（Portability）

如果软件能够在不同的环境下运行，软件就是可移植的[58]。会导致可移植性属性的风格包括那些将代码和代码所要处理的数据一起移动的风格，例如虚拟机和移动代理（mobile agent）风格；以及那些限制只能使用标准格式的数据元素的风格。

2.3.7 可靠性（Reliability）

从应用的架构角度来说，可靠性可以被看作当在组件、连接器或数据之中出现部分故障时，一个架构容易受到系统层面故障影响的程度。架构风格能够通过以下方法提高可靠性：避免单点故障、增加冗余、允许监视、以及用可恢复的动作来缩小故障的范围。

2.4 小结

通过聚焦于基于网络的应用的架构，并且描述架构风格如何能够被用来指导这些架构的设计，本章检查了本论文的范围。本章还定义了架构属性的集合，这些架构属性将在本论文的剩余部分中被用来对架构风格进行比较和评估。

下一章将会在一个分类框架中，调查一些常见的基于网络的应用的架构风格。当将架构风格应用于一种作为原型的基于网络的超媒体系统的架构时，将会导致一系列架构属性。这个分类框架根据这些架构属性，来对每种架构风格进行评估。

第 3 章 基于网络的架构风格

本章在一个分类框架中，调查一些常见的基于网络的应用的架构风格。当将架构风格应用于一种作为原型的基于网络的超媒体系统的架构时，将会导致一系列架构属性。这个分类框架根据这些架构属性，来对每种架构风格进行评估。

3.1 分类方法学

建造软件的目的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——而是为了创造出满足或者超出应用之需求的系统。为系统设计而选择的架构风格必须要与这些需求相一致，而不是相抵触。因此，为了提供有用的设计指导，应当基于这些架构风格所导致的架构属性来对架构风格进行分类。

3.1.1 选择哪些架构风格来进行分类

这个分类中包括的架构风格集合并不是包罗万象的，并没有包括所有可能的基于网络的应用的架构风格。事实上，只要将一个架构约束添加到任意一种被调查的风格中，就能够形成一种新的风格。我的目标是描述一组有代表性的风格样本（特别是那些在软件架构文献中已经确定的风格），并且提供一个框架，使得其他风格一旦被开发出来就能够被添加到这个分类中。

我有意排除了那些与被调查的风格结合后，形成的基于网络的应用并不能为通信或交互属性带来任何增强的风格。例如，黑板架构风格（blackboard architectural style）[95]由一个中央仓库和一组在这个仓库上适时地执行操作的组件（知识源）组成。黑板架构可以通过分布组件的方式被扩展成为基于网络的系统，但是这样一种扩展的属性完全基于支持分布（distribution）的交互风格——通过基于事件的集成（event-based integration）来通知、按照客户-服务器方式来轮询、或者仓库的复制。因此，尽管这种混合风格具备网络能力，但是将它包括到分类中却没有增加任何的价值。

3.1.2 风格所导致的架构属性

我的分类使用每种风格所导致的架构属性的相对变化，作为一种方法来展示当每种风格被应用于分布式超媒体系统时所产生的效果。请注意，正如在 2.2 节中所描述的，对于某个特定属性的一种风格进行评估，依赖于正在被研究的系统的交互类型。架构属性是相对的，添加一种架构约束可能会增强，也可能会削弱一个特定的属性，或者在增强属性的一个方面的同时削弱属性的另一个方面。同样，增强一个属性也可能会导致削弱另一个属性。

尽管我们对于架构风格的讨论包括了那些广泛使用的基于网络的系统，但我们对于每种风格的评估是以它对于我们所研究的单一软件类型——基于网络的超媒体系统的架构所产生的影响为基础的。聚焦于一种特定的软件类型，使我们可以使用一个系统设计师评估那些优点的相同方法，识别出一种风格超越其他风格的优点。由于我们并不打算宣称任何单一的风格对于所有的软件类型都是普适的，因此通过限制我们的评估的焦点，简单地减少了我们需要评估的风格的数量。为其他的应用类型评估相同的风格对于未来的研究来说，是一个开放的领域。

3.1.3 可视化

我使用一张风格和架构属性的比较表作为分类的主要可视化方式。表中的数值表明了特定行中的风格对于列中的架构属性的相对影响。减号（-）表示消极影响，加号（+）表示积

极影响，加减号（±）表示影响的性质依赖于问题领域的某个方面。尽管这是一个对于下面各节中的细节的一个粗略的简化，但它确实能够表明一种风格所带来（或者忽略）的架构属性。

另一种替代的可视化方法是使用一张对架构风格进行分类的基于属性的继承关系图表。各种风格根据它们之间的继承关系来分类，风格之间的弧线显示出得到或失去的架构属性。这张图表的起点是“空”风格（没有约束）。有可能直接从对于架构风格的描述衍生出这样一张图表。

3.2 数据流风格（Data-flow Styles）

表 3-1：评估基于网络的超媒体系统的数据流风格

风格	继承	网络性能	用户可觉察性能	效率	可伸缩性	简单性	可进化性	可扩展性	可定制性	可配置性	可重用性	可见性	可移植性	可靠性
PF			±			+	+	+		+	+			
UPF	PF	-	±			++	+	+		++	++	+		

3.2.1 管道和过滤器（Pipe and Filter，PF）

在管道和过滤器风格中，每个组件（过滤器）从其输入端读取数据流并在其输出端产生数据流，通常对输入流应用一种转换并增量地处理它们，以使输出在输入被完全处理完之前就能够开始[53]。这种风格也被称作单路数据流网络（one-way data flow network）[6]。这里的架构约束是一个过滤器必须完全独立于其他的过滤器（零耦合）：它不能与其他过滤器在其上行和下行数据流接口分享状态、控制线程或标识[53]。

Abowd 等人[1]使用 Z 语言为管道和过滤器风格提供了广泛的形式化描述。用于图像处理的 *Khoros* 软件开发环境[112]提供了一个使用此风格建造的一系列应用的优秀例子。

Garlan 和 Shaw [53]描述了管道和过滤器风格的几个有利的属性。首先，PF 允许设计者将系统全部的输入/输出看作个别的过滤器行为的简单组合（简单性）。其次，PF 支持重用：任何两个过滤器都能够被连接（hooked）在一起，只要它们允许数据在它们之间传输（可重用性）。第三，PF 系统能够很容易地被维护和增强：新的过滤器能够被添加到现有的系统中（可扩展性）；而旧的过滤器能够被改进后的过滤器所替代（可进化性）。第四，PF 允许一些特定类型的专门性分析（可验证性），例如吞吐量和死锁分析。最后，PF 天生支持并发执行（用户可觉察的性能）。

PF 风格的缺点包括：通过长的管道时会导致延迟的增加；如果一个过滤器不能增量地处理它的输入，那么将进行批量的串行处理，此时不允许进行任何交互。一个过滤器无法与环境进行交互，因为它不知道哪个特定的输出流与哪个特定的输入流共享一个控制器。如果正在解决的问题不适合使用数据流模式，这些属性会降低用户可觉察的性能。

PF 风格有一个很少被提到的方面：存在着一个潜在的“看不见的手”——即，为了建立整个应用而安排过滤器的配置。一个由多个过滤器组成的网络通常在每次投入使用之前就已经安排好了：允许应用基于手头的任务和数据流的特性来指定过滤器组件的配置（可配置性）。这个控制功能被看作系统的一个独立的操作阶段，因此是一个独立的架构，即使系统

的各阶段是密不可分的。

3.2.2 统一管道和过滤器（Uniform Pipe and Filter，UPF）

统一管道和过滤器风格在 PF 风格的基础上，添加了一个约束，即所有过滤器必须具有相同的接口。这种风格的主要实例出现在 Unix 操作系统中，其中过滤器进程具有由一个字符输入数据流（stdin）和两个字符输出数据流（stdout 和 stderr）组成的接口。通过限定使用这个接口，就能够随意排列组合独立开发的过滤器，从而形成新的应用。这也简化了理解一个特定的过滤器如何工作的任务。

统一接口的一个缺点是：如果数据需要被转换为它的原始格式或者从它的原始格式转换为特定的格式，这个约束可能会降低网络性能。

3.3 复制风格（Replication Styles）

表 3-2：评估基于网络的超媒体系统的复制风格

风格	继承	网络性能	用户可觉察性能	效率	可伸缩性	简单性	可进化性	可扩展性	可定制性	可配置性	可重用性	可见性	可移植性	可靠性
RR			++		+									+
$	RR		+	+	+	+								

3.3.1 复制仓库（Replicated Repository，RR）

基于复制仓库风格[6]的系统通过利用多个进程提供相同的服务，来改善数据的可访问性（accessibility of data）和服务的可伸缩性（scalability of service）。这些分散的服务器交互为客户端制造出只有一个集中的服务的“幻觉”。主要的例子包括诸如 XMS [49]这样的分布式文件系统和 CVS[www.cyclic.com]这样的远程版本控制系统。

RR 风格的主要优点在于改善了用户可觉察的性能，实现途径是通过减少处理正常请求的延迟，并在主服务器故障或有意的线下漫游（intentional roaming off the network）时支持离线操作（disconnected operation）。在这里，简单性是不确定的，因为 RR 风格允许不关心网络（network-unaware）的组件能够透明地操作本地的复制数据，这补偿了复制所导致的复杂性。维护一致性是 RR 风格的主要关注点。

3.3.2 缓存（Cache，$）

复制仓库风格的一种变体是缓存风格：复制个别请求的结果，以便可以被后面的请求重用。这种形式的复制最常出现在潜在的数据集远远超出单个客户端的容量的情况下，例如在 WWW[20]中，或者在不必完全访问仓库的地方。可以执行延迟复制（lazy replication）：当复制一个请求的尚未缓存的响应数据时，根据引用的局部性（locality of reference）和兴趣的趋同性（commonality of interest），将有用的数据项复制到缓存中以备稍后重用。还可以执行执行主动复制（active replication）：基于预测到的请求来预先获取可缓存的数据项。

与复制仓库风格相比，缓存风格对于用户可觉察性能的改善较少，因为没有命中缓存的

请求会更多，只有近期被访问过的数据才能被离线操作使用。另一方面，缓存风格实现起来要容易得多，不需要复制仓库风格那么多的处理和存储，而且由于只有当数据被请求时才会传输数据，因此缓存风格更加高效。缓存风格当与客户-无状态-服务器风格（client- stateless-server style）结合后就成为了一种基于网络的架构风格。

3.4 分层风格（Hierarchical Styles）

表 3-3: 评估基于网络的超媒体系统的分层风格

风格	继承	网络性能	用户可觉察性能	效率	可伸缩性	简单性	可进化性	可扩展性	可定制性	可配置性	可重用性	可见性	可移植性	可靠性
CS					+	+	+							
LS			-		+		+				+		+	
LCS	CS+LS		-		++	+	++				+		+	
CSS	CS	-			++	+	+					+		+
C$SS	CSS+$	-	+	+	++	+	+					+		+
LC$SS	LCS+C$SS	-	±	+	+++	++	++				+	+	+	+
RS	CS			+	-	+	+					-		
RDA	CS			+	-	-						+		-

3.4.1 客户-服务器（Client-Server，CS）

客户-服务器风格在基于网络的应用的架构风格中最为常见。服务器组件提供了一组服务，并监听对这些服务的请求。客户端组件通过一个连接器将请求发送到服务器，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回客户端。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：一个客户是一个触发进程；一个服务器是一个反应进程。客户端发送请求触发服务器的反应。这样，客户端可以在它所选择的时间启动活动；然后它通常等待直到请求的服务完成处理。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端提供服务。

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接口（译者注：即用户界面）功能移到客户端组件中。只要接口不发生改变，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如何在客户端组件和服务器组件之间划分。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）[131]。

3.4.2 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”的风格，但是它在基于网络的系统中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。

分层系统通过对相邻的外部层之外的所有层隐藏内部层，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，降低了用户可觉察的性能[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。一个代理组件作为一个或多个客户端组件的共享服务器（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。一个网关组件在客户端或代理看起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”（inner- layer）服务器。这些额外的中间组件添加了很多个层，用来为系统添加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端组件来处理[6]。

3.4.3 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）。从客户端发到服务器的每个请求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 3 个架构属性。可见性的改善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。可靠性的改善是因为这些约束简化了从部分故障中恢复的任务[133]。可伸缩性的改善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，通过增加在一系列请求中发送的重复数据（每次交互的开销），可能会降低网络性能。

3.4.4 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）。一个缓存在客户端和服务器之间扮演一个仲裁者：早先请求的响应能够（如果它们被认为是可缓存的）被重用，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。有效地利用此风格的实例系统是 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可觉察的性能。

3.4.5 分层-客户-缓存-无状态-服务器（Layered-Client-Cache-Stateless-Server，LC$SS）

分层-客户-缓存-无状态-服务器风格通过添加代理和/或网关组件，继承了分层-客户-服务器风格和客户-缓存-无状态-服务器风格。使用此风格的范例系统是 Internet 域名系统（DNS）。

LC$SS 风格的优点和缺点是 LCS 风格和 C$SS 风格的优点和缺点集合。然而，请注意我们不能将 CS 风格的贡献计算两次，因为如果贡献来自相同的祖先的话，那么其优点是不可叠加的。

3.4.6 远程会话（Remote Session，RS）

远程会话风格是客户-服务器风格的一种变体，它试图使客户端组件（而非服务器组件）的复杂性最小化或者使得它们的可重用性最大化。每个客户端在服务器上启动一个会话，然后调用服务器的一系列服务，最后退出会话。应用状态被完全保存在服务器上。这种风格通常在以下场合中使用：想要使用一个通用的客户端（generic client）（例如 TELNET[106]）或者通过一个模仿通用客户端的接口（例如 FTP [107]）来访问远程服务。

远程会话风格的优点是：集中维护在服务器的接口更加容易；当对功能进行扩展时，减少了已部署的客户端中的不一致问题；如果交互利用了服务器上扩展的会话上下文，它能够提高效率。它的缺点是：由于要在服务器上保存应用状态，降低了服务器的可伸缩性；因为监视程序必须要知道服务器的完整状态，降低了交互的可见性。

3.4.7 远程数据访问（Remote Data Access，RDA）

远程数据访问风格[131]是客户-服务器风格的一种变体，它将应用状态分布在客户端和服务器上。客户端以一种标准的格式发送一个数据库查询（例如 SQL）请求到服务器，服务器分配一个工作空间并执行这个查询，这可能会导致一个巨大的结果集。客户端能够在结果集上进行进一步操作（例如表连接）或者每次获取结果的一部分。客户端必须了解服务的数据结构，以便建造依赖于该结构的查询。

远程数据访问风格的优点是：一个巨大的数据集能够在服务器端通过多次迭代的方式逐渐减少，而不需要通过网络传输完整的数据集，从而改善了效率；通过使用一种标准的查询语言，从而改善了可见性。这种风格的缺点是：客户端必须像服务器实现那样理解相同的数据库操作概念（因此缺少简单性）；而且在服务器上保存应用的上下文，降低了可伸缩性。由于部分故障会导致工作空间处于未知状态 ，可靠性也蒙受了损失。尽管能够使用事务机制（例如，二次提交）来修正可靠性的问题，但是代价是增加了复杂性和交互的开销。

3.5 移动代码风格（Mobile Code Styles）

移动代码风格使用移动性（mobility）来动态地改变在处理过程与数据源或结果目的地之间的距离。Fuggetta 等人[50]全面地检查过这些风格。为了考虑不同组件的位置，他们在架构层面引入了一种站点抽象（site abstraction），作为主动配置的一部分。位置（location）概念的引入，使得在设计的层面对组件之间的交互开销进行建模成为了可能。特别是，当与包括了网络通信的交互的开销作比较时，共享同一位置的组件之间的交互开销的成本被认为是可以忽略不计的。通过改变自己的位置，一个组件可以改善接近性（proximity）和它的交互的质量，减少交互开销从而提高效率和用户可觉察的性能。

在所有的移动代码风格中，一个数据成员被动态地转换为一个组件。为了确定一个特定的行为是否需要移动性，Fuggetta 等人[50]使用了一种分析方法：将代码尺寸（code's size）

作为一个数据成员，与在正常的数据转移中所节省的部分作比较。如果软件架构的定义排除了数据元素，那么从架构的观点来建模将是不可能的。

表 3-4：评估基于网络的超媒体系统的移动代码风格

风格	继承	网络性能	用户可觉察性能	效率	可伸缩性	简单性	可进化性	可扩展性	可定制性	可配置性	可重用性	可见性	可移植性	可靠性
VM						±		+				-	+	
REV	CS+VM			+	-	±		+	+			-	+	-
COD	CS+VM		+	+	+	±		+		+		-		
LCODC$SS	LC$SS+COD	-	++	++	+4+	+±+	++	+		+	+	±	+	+
MA	REV+COD		+	++		±		++	+	+		-	+	

3.5.1 虚拟机（Virtual Machine，VM）

所有移动代码风格的基础是虚拟机（或解释器）风格[53]。代码必须以某种方式来执行，首选的方式是在一个满足了安全性和可靠性关注点的受控环境中执行，而这正是虚拟机风格所提供的。虚拟机风格本身并不是基于网络的风格，但是它通常在客户-服务器风格（REV 和 COD 风格）中与一个组件结合在一起使用。

虚拟机通常被用作脚本语言的引擎，包括像 Perl[134]这样的通用语言和像 PostScript[2]这样的与特定任务相关的语言。虚拟机带来的主要好处是在一个特定平台上分离了指令（instruction）和实现（implementation）（可移植性），并且使可扩展性变得容易。因为难以简单地通过查看代码来了解可执行代码将要做什么事情，因此降低了可见性。同时由于需要对求值环境（evaluation environment）进行管理，也降低了简单性，但在一些情况下可以通过简化静态的功能（static functionality）得到补偿。

3.5.2 远程求值（Remote Evaluation，REV）

远程求值风格[50]来源于客户-服务器风格和虚拟机风格，一个客户端组件必须要知道如何来执行一个服务，但缺少执行此服务所必需的资源（CPU 周期、数据源等等），这些资源恰好位于一个远程站点上。因此，客户端将如何执行服务的代码发送给远程站点上的一个服务器组件，服务器组件使用可用的资源来执行代码，然后将执行结果发送回客户端。这种远程求值风格假设将要被执行的代码是处在一种受保护的环境中，这样除了那些正在被使用的资源外，它不会影响到相同服务器的其他客户端。

远程求值风格的优点包括：能够定制服务器组件的服务，这改善了可扩展性和可定制性；当代码能够使它的动作适应于服务器内部的环境（而不是客户端做出一系列交互来做同样的事情）时，能够得到更好的效率（译者注：即通过一次交互，直接发送一段代码到服务器上执行，并且将结果返回）；由于需要管理求值的环境，降低了简单性，但在一些情况下可以通过简化静态的服务器功能得到补偿。可伸缩性降低了，但是可以通过服务器对执行环境的管理（杀掉长期运行的代码，或当资源紧张时杀掉大量消耗资源的代码）加以改善，但是管

理功能本身会导致与部分故障和可靠性相关的难题。然而，最大的限制是，由于客户端发送代码而不是标准化的查询，因此缺乏可见性。如果服务器无法信任客户端，缺乏可见性会导致明显的部署问题。

3.5.3 按需代码（Code on Demand，COD）

在按需代码风格[50]中，一个客户端组件知道如何访问一组资源，但不知道如何处理它们。它向一个远程服务器发送对于如何处理资源的代码的请求，接收这些代码，然后在本地执行这些代码。

按需代码风格的优点包括：能够为一个已部署的客户端添加功能，改善了可扩展性和可配置性；当代码能够使它的动作适应于客户端的环境，并在本地与用户交互而不是通过远程交互时，能够得到更好的用户可觉察性能和效率。由于需要管理求值环境，降低了简单性，但在一些情况下可以通过简化静态的客户端功能得到补偿。由于服务器将工作交给了客户端（否则将消耗服务器的资源），从而改善了服务器的可伸缩性。像远程求值风格一样，最大的限制是由于服务器发送代码而不是简单的数据，因此缺乏可见性。如果客户端无法信任服务器，缺乏可见性会导致明显的部署问题。

3.5.4 分层-按需代码-客户-缓存-无状态-服务器（Layered-Code-on-Demand-Client-Cache-Stateless-Server，LCODC$SS）

作为一些架构如何互补的例子，考虑将按需代码风格添加到上面讨论过的分层-客户-缓存-无状态-服务器风格上。因为代码被看作不过是另一种数据元素，因此这并不会妨碍 LC$SS 风格的优点。该风格的一个例子是 HotJava Web 浏览器[java.sun.com]，它允许 applet 和协议扩展作为有类型的媒体（typed media）来下载。

LCODC$SS 风格的优点和缺点正是 COD 风格和 LC$SS 风格的优点和缺点的组合。我们将进一步讨论 COD 风格和其他 CS 风格的组合，不过这个调查并非是想要包括所有可能的组合。

3.5.5 移动代理（Mobile Agent，MA）

在移动代理风格[50]中，一个完整的计算组件，与它的状态、必需的代码、执行任务所需的数据一起被移动到远程站点。该风格可以看作来源于远程求值风格和按需代码风格，因为移动性是同时以这两种方式工作的。

移动代理风格超越那些已经在 REV 风格和 COD 风格中描述过的优点的主要优点是：对于选择在何时移动代码而言，具有更大的灵活性（dynamism）。当一个应用根据推算决定移动到另一个地点，以减少在该应用和它希望处理的下一组数据之间的距离，此时它可以在一个地点正处于处理信息的中途（译者注：即不必等待信息完全处理完）。此外，因为应用状态每次都是在单一地点，所以减少了由局部故障引起的可靠性问题。

3.6 点对点风格（Peer-to-Peer Styles）

表 3-5：评估基于网络的超媒体系统的点对点风格

风格	继承	网络性能	用户可觉察性能	效率	可伸缩性	简单性	可进化性	可扩展性	可定制性	可配置性	可重用性	可见性	可移植性	可靠性
EBI				+	--	±	+	+		+	+	-		-
C2	EBI+LCS		-	+		+	++	+		+	++	±	+	±
DO	CS+CS	-		+			+	+		+	+	-		-
BDO	DO+LCS	-	-				++	+		+	++	-	+	

3.6.1 基于事件的集成（Event-based Integration，EBI）

基于事件的集成风格也被称作隐式调用（implicit invocation）风格或者事件系统（event system）风格，它通过除去了解连接器接口的标识（identity on the connector interface）的必要性，降低了组件之间的耦合。此风格不是直接调用另一个组件，而是一个组件能够发布（或广播）一个或者多个事件。在事件发布后，系统中的其他组件能够注册对于某些事件类型的兴趣，由系统本身来调用所有已注册的组件[53]。该风格的例子包括：Smalltalk-80 中的 MVC 范例[72]，以及很多软件工程环境的集成机制，包括 Field [113]、SoftBench[29]和 Polylith[110]。

通过使添加侦听事件的新组件变得容易（可扩展性）、鼓励使用通用的事件接口和集成机制（可重用性）、允许组件被替换而不会影响其他组件的接口（可进化性），基于事件的集成风格为可扩展性、可重用性和可进化性提供了强有力的支持。如同管道和过滤器系统一样，将组件放在事件接口上——这一“看不见的手”需要高层次的配置架构。大多数 EBI 系统也将显式调用（explicit invocation）作为交互的一种补充形式 [53]。对于由数据监视支配，而不是由数据获取支配的应用，EBI 通过除去轮询式交互（polling interactions）的必要性，能够提高效率。

EBI 系统的基本形式由一个事件总线组成，所有的组件通过这个总线侦听它们所感兴趣的事件。当然，这会立即导致有关通知的数量、由通知引发其他组件的广播从而导致的事件风暴、在通知传送系统中的单点故障等方面的可伸缩性问题。这些问题能够通过以损害简单性为代价，使用分层系统和事件过滤来加以改善。

EBI 系统的另一个缺点是：难以预料一个动作将会产生什么样的响应（缺乏可理解性），事件通知并不适合交换大粒度的数据[53]，而且也不支持从局部故障中恢复。

3.6.2 C2

C2 架构风格[128]直接支持大粒度的重用，并且通过加强底层独立性（substrate independence），支持系统组件的灵活组合。它通过将基于事件的集成风格和分层-客户-服务器风格相结合来达到这些目标。异步通知消息向下传送，异步请求消息向上传送，这是组件之间通信的唯一方式。这加强了对高层依赖的松散耦合（服务请求可以被忽略），并且与底层实现了零耦合（不知道使用了通知），从而改善了对于整个系统的控制，又没有丧失 EBI 的大多数优点。

通知是对于组件中的状态变化的公告。C2 并不对在通知中应该包括什么内容加以限制：一个标志、一个状态 delta 改变量、或者一个完整的状态表述都是有可能的。一个连接器的首要职责是消息的路由和广播；它的第二个职责是消息过滤。引入对于消息的分层过滤，解决了 EBI 系统的可伸缩性问题，同时也改善了可进化性和可重用性。包括了监视能力的重量级连接器能够被用来改善可见性和减少局部故障所导致的可靠性问题。

3.6.3 分布式对象（Distributed Objects，DO）

分布式对象风格将系统组织为结对进行交互的组件的集合。一个对象是一个实体，这个实体封装了一些私有的状态信息或数据、操作数据的一组相关的操作或过程、以及一个可能存在的控制线程，这种封装使得它们能够被整体地看作单个的单元[31]。通常，一个对象的状态对于所有其他对象而言，是完全隐藏和受到保护的。检查或修改对象状态的唯一方法是对该对象的一个公共的、可访问的操作发起请求或调用。这样就为每个对象创建了一个良好定义的接口，在对象的操作实现和它的状态信息保持私有的同时，公开操作对象的规格，这样做改善了可进化性。

一个操作可以调用可能位于其他对象之上的操作。这些操作也同样可以调用其他对象之上的操作，以此类推。一个相关联的调用链被称作一个动作（action）[31]。状态分布于对象之间，这对使状态尽可能保持最新而言是有利的，但是不利之处是难以获得系统活动的总的视图（缺乏可见性）。

一个对象为了与另一个对象交互，它必须知道另一个对象的标识。当一个对象的标识改变时，它必须对所有显式调用它的其他对象做修改[53]。这就必需要有一些控制器对象来负责维护系统的状态，以完成应用的需求。分布式对象系统的核心问题包括：对象管理、对象交互管理和资源管理[31]。

分布式对象系统被设计用来隔离正在被处理的数据，因此该风格通常不支持数据流。然而，当它和移动代理风格相结合时，可以为对象提供更好的移动性。

3.6.4 被代理的分布式对象（Brokered Distributed Objects，BDO）

为了降低对象标识的影响，现代分布式对象系统通常使用一种或更多种中间风格（intermediary styles）来辅助通信。这包括基于事件的集成风格和被代理的客户/服务器（brokered client/server）风格[28]。被代理的分布式对象风格引入了名称解析组件——其目的是将该组件接收到的客户端请求中一个通用的服务名称解析为一个能够满足该请求的对象的特定名称，并使用这个特定名称来答复客户端。尽管它改善了可重用性和可进化性，但额外的间接层要求额外的网络交互，这降低了效率和用户可觉察的性能。

被代理的分布式对象系统目前受到两个标准的控制：OMG[97]所开发的 CORBA 行业标准和 ISO/IEC [66]所开发的开放分布式处理（ODP）的国际标准。

尽管分布式对象引起了非常多的兴趣，然而与大多数其他的基于网络的架构风格相比，这样一类架构风格能够提供的优点很少，它们最适合被使用在包括了对已封装服务（例如硬件设备）的远程调用的应用中，在这些应用中，效率和网络交互的频率并不是很值得关注。

3.7 局限

每一种架构风格都推崇在组件之间的一种特定的交互类型。当组件跨广域网（wide-area network）分布时，对于网络的使用或误用会严重影响应用的可用性。通过以架构风格对于架构属性的影响（尤其是对于一个分布式超媒体系统的基于网络的应用的性能的影响）来刻画架构，我们得以有能力选择更加适合于此类应用的软件设计。然而，对于所选择的分类方

法，这里存在着一些局限。

第一个局限是这里的评估是特别为分布式超媒体的需求而量身定制的。例如，如果通信的内容是细粒度的控制消息，那么管道和过滤器风格的很多优良品质就不复存在；而且如果通信需要用户的交互，管道和过滤器风格根本就不适用。同样地，如果对客户端请求的响应完全没有被缓存，那么分层缓存风格只会增加延迟，而不会带来任何好处。这种类型的区别并没有出现在这个分类中，只能在对于每种风格的讨论中非形式化地加以探讨。我相信这个局限能够通过对每一种类型的通信问题创建单独的分类表格来加以解决。问题领域的例子包括：大粒度的数据获取、远程信息监视、搜索、远程控制系统、以及分布式处理。

第二个局限是对于架构属性的分组。在一些情况下，识别出一个架构属性所导致的一些特殊的方面，例如可理解性和可验证性，要比将它们笼统地混在简单性的标题下更好。尤其是对于那些有可能以损失可理解性为代价改善可验证性的风格而言。然而，将一个有很多抽象概念的属性作为单个度量手段也是有价值的，因为我们并不想使这个分类过于特殊化，以至于不存在影响相同属性类别的两种风格。一种解决方案就是在分类中既包括特殊的属性，也包括概括的属性。

尽管如此，这些最初的调查和分类，对于任何可能解决这些局限的更进一步的分类来说，是一个必需的先决条件。

3.8 相关工作

3.8.1 架构风格和模式的分类方法

与本章最直接相关的研究领域是架构风格和架构级模式（architecture-level patterns）的识别和分类。

Shaw [117]描述了一些架构风格，后来 Garlan 和 Shaw[53]对这些风格进行了扩展。Shaw 和 Clements[122]提出了这些风格的初步分类，Bass 等人[9]重复了他们的工作，其中使用了以控制和数据（control and data issues）为坐标轴的二维的、表格化的分类策略，按照以下的功能类别加以组织：在风格中使用哪种组件和连接器；在组件之间如何共享、分配和转移控制；数据如何通过系统来进行通信；数据和控制如何交互；何种类型的推理机制是与该风格相兼容的。这种分类方法的主要目的是标识风格的特征，而不是帮助对风格进行比较。它总结出一组少量的“经验法则”，作为一种形式的设计指导。

与本章不同的是，Shaw 和 Clements[122]的分类并没有为应用软件的设计师提供一种有用的方法，来帮助他们对于设计进行评估。问题是建造软件的目的并不是建造一种具有特殊的形状、拓扑或者组件类型的系统，以这种方式来对分类加以组织，并不能够帮助设计师找到符合他们的需要的架构风格。它也混淆了风格之间的本质区别和那些只具有次要重要性的其他问题，并且模糊了风格之间的来源关系。进一步讲，它并没有将焦点放在任何特殊的架构类型上，例如基于网络的应用。最后，它无法描述出风格能够如何组合和将它们组合之后的效果。

Buschmann 和 Meunier[27]描述了一种根据抽象的粒度、功能、结构原则来组织模式的分类方案（classification scheme）。根据抽象的粒度（granularity of abstraction）将模式划分为三个分类：架构框架（architectural frameworks□ 用于架构的模板）、设计模式（design patterns）和习惯用法（idioms）。他们的分类解决的一些问题与本文所解决的问题相同，例如分离关注点和导致架构属性的结构原则（structural principles），但是仅仅覆盖了两种这里所描述的架构风格。他们的分类后来又被 Buschmann 等人[28]进行了相当大的扩展，后面的这份文献讨论了广泛得多的的架构模式，以及它们与软件架构的关系。

Zimmer[137]使用一个以设计模式之间的关系为基础的图表，来组织设计模式，这使得

理解 Gamma 等人[51]的目录中的模式的全部结构更加容易。然而，被分类的模式并不是架构模式，分类仅仅是排他性地基于起源或使用关系，而不是基于架构属性。

3.8.2 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、一个分布式程序中的各种过程（过滤器、客户端、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）的复制工人（replicated worker）。然而，他从在单个任务上互操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]提供了一个对于隐式调用研究的调查，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，提供了对基于事件的集成机制的调查。Rosenblum 和 Wolf[114]调查了一个用于 Internet 规模的事件通知的设计框架。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。

Fuggetta 等人[50]提供了对于移动代码范例的一个彻底的调查和分类。本章建立在他们的工作之上并进行了扩展：我将移动代码风格与其他基于网络的风格进行了比较，并将它们放在单一的框架和架构定义集合之中。

3.8.3 中间件

Bernstein[22]将中间件定义为包括了标准编程接口和协议的分布式系统服务。这些服务被称为中间件，是因为它们扮演了一个位于操作系统和网络软件之上、特定行业的应用软件之下的中间层。Umar[131]提供了对于中间件的广泛的分析。

关于中间件的架构研究聚焦于在现成的（off-the-shelf）中间件中集成组件的问题和影响。Di Nitto 和 Rosenblum[38]描述了对于中间件和预定义组件（predefined components）的使用如何影响正在开发的系统的架构，以及反过来，对于特定架构的选择如何限制了对于中间件的选择。Dashofy 等人[35]讨论了以 C2 风格来使用中间件。

Garlan 等人[56]指出了在现成的组件中的一些架构假设，检查了创建者在创建用于架构设计的 Aesop 工具[54]的过程中重用子系统时存在的问题。他们将问题分类为能够造成架构不匹配的四个主要的假设：组件的特性（nature of components）、连接器的特性（nature of connectors）、全局架构的结构（global architectural structure）、以及构建过程（construction process）。

3.9 小结

本章在一个分类框架中，提供了一个对基于网络的应用的常见架构风格的调查。当将架构风格应用于一种作为原型的基于网络的超媒体系统的架构时，将会导致一系列架构属性，这个分类框架根据这些架构属性，来对每种架构风格进行评估。在下面的表 3-6 列出了全部的分类。

下一章使用从这个调查和分类中所获得的洞察力，推导出开发和评估一种架构风格的方法，用来对改进现代万维网架构的设计加以指导。

表 3-6：评估总结

风格	继承	网络性能	用户可觉察性能	效率	可伸缩性	简单性	可进化性	可扩展性	可定制性	可配置性	可重用性	可见性	可移植性	可靠性
PF			±			+	+	+		+	+			
UPF	PF	-	±			++	+	+		++	++	+		
RR			++		+									+
$	RR		+	+	+	+								
CS					+	+	+							
LS			-		+		+				+		+	
LCS	CS+LS		-		++	+	++				+		+	
CSS	CS	-			++	+	+					+		+
C$SS	CSS+$	-	+	+	++	+	+					+		+
LC$SS	LCS+C$SS	-	±	+	+++	++	++				+	+	+	+
RS	CS			+	-	+	+					-		
RDA	CS			+	-	-						+		-
VM						±		+				-	+	
REV	CS+VM			+	-	±		+	+			-	+	-
COD	CS+VM		+	+	+	±		+		+		-		
LCODC$SS	LC$SS+COD	-	++	++	+4+	+±+	++	+		+	+	±	+	+
MA	REV+COD		+	++		±		++	+	+		-	+	
EBI				+	--	±	+	+		+	+	-		-
C2	EBI+LCS		-	+		+	++	+		+	++	±	+	±
DO	CS+CS	-		+			+	+		+	+	-		-
BDO	DO+LCS	-	-				++	+		+	++	-	+	

第 4 章 设计 Web 架构：问题与洞察力

本章介绍了万维网架构的需求和一些问题，这些问题是在对万维网的关键通信协议进行设计和对提议的改进进行评估的过程中遇到的。从对基于网络的超媒体系统的架构风格的调查和分类过程中获得的洞察力，使我有能力推导出开发一种架构风格的方法，用来对改进现代万维网架构的设计加以指导。

4.1 万维网应用领域的需求

Berners-Lee[20]写到：“Web 的主要目的是旨在成为一种共享的信息空间（a shared information space），人们和机器都可以通过它来进行沟通。”我们需要的是一种人们用来保存和构造他们自己的信息的方式，无论信息在性质上是永久的还是短暂的，这样信息对于他们自己和其他人都是可用的，并且能够引用和构造由其他人保存的信息，而不必每个人都保持和维护一份本地的副本。

这个系统最初所希望的最终用户是分布在世界各地，通过 Internet 连接的各个大学和政府的高能物理研究实验室。他们的机器是各种不同种类的终端、工作站、服务器和超级计算机的大杂烩，所以他们所需要的操作系统软件和文件格式也是一个“大杂烩”。信息的范围涉及到从个人的研究笔记到组织机构的电话列表等方面。建造一个这样的系统所面对的挑战是：为这些结构化的信息提供统一的、一致的接口；这些信息可以在尽可能多的平台上获得；当新的人和新的组织加入到这个项目（译者注：即 Web）时可进行增量的部署。

4.1.1 低门槛

参与创建和构造信息是自愿的，因此采用“低门槛”策略是十分必要的。这种策略被应用于 Web 架构的所有使用者：阅读者、创作者和应用的开发者。

超媒体被选择作为用户接口是因为它的简单性和通用性：能够使用相同的接口而无须考虑信息的来源，超媒体关系（链接）的灵活性使得能够对其进行无限的构造，对于链接的直接操作允许在信息内部建立复杂的关系，来引导阅读者浏览整个应用。因为通过一个搜索接口访问大型数据库中的信息，常常要比通过浏览方式来访问更加容易，所以 Web 也结合了这种能力：通过将用户输入的数据提供给服务，然后呈现超媒体形式的结果，来执行简单的查询。

对于创作者而言，首要的需求是整个系统的部分可用性（partial availability）必须不至于妨碍对于内容的创作。超文本的创作语言（hypertext authoring language）必须是简单的，能够使用现有的编辑工具来创建。无论是否是直接连接到 Internet，都期待创作者能以这种格式将其创作内容保存为个人的研究笔记，因此一些被引用的信息尚不可用（无论是暂时性的还是永久性的）这一事实不能妨碍对可用信息的阅读和创作。因为类似的原因，必须能够在所引用的目标信息可用之前创建对于该信息的引用。因为创作者被鼓励在对信息源的开发中进行合作，因此无论引用的形式是 e-mail 地址还是在会议中写在餐巾纸的背面，这些引用都必须是容易沟通的。

出于应用开发者的利益，简单性也是一个目标。由于所有的协议都被定义为文本，所以通信能够被观察，并且能够使用现有的网络工具来对通信进行交互式的测试。这使得尽管缺少标准，但是协议还是能够尽早地得到应用。

4.1.2 可扩展性

简单性使得部署一个分布式系统的最初实现成为了可能，可扩展性使得我们避免了永远

陷入已部署系统的局限之中。即使有可能建造一个完美地匹配用户需求的软件系统，那些需求也会随时间发生变化，就像社会的变化一样。如果一个系统想要像 Web 那样“长命”，它就必须做好应对变化的准备。

4.1.3 分布式超媒体

超媒体（hypermedia）是由应用控制信息（application control information）来定义的，这些控制信息内嵌在信息的表述之中，或者作为信息的表述之上的一层。分布式超媒体允许在远程地点存储表述和控制信息。由于它的这个特性，一个分布式超媒体系统中的用户动作需要将大量的数据从其存储地转移到其使用地。这样，Web 架构必须被设计为支持大粒度的数据转移。

用户可觉察的延迟（在选择一个链接和呈现可用的结果之间的时间）对于超媒体交互的可用性而言是高度敏感的。因为 Web 的信息源是跨越整个 Internet 分布的，这种架构必须使网络交互（在数据转移协议中的往返时间）最小化。

4.1.4 Internet 规模

Web 是旨在成为一个 Internet 规模的分布式超媒体系统，这意味着它的内涵远远不只仅仅是地理上的分布。Internet 是跨越组织边界互相连接的信息网络。信息服务的提供商必需能够应对无法控制的可伸缩性和软件组件的独立部署两方面的需求。

4.1.4.1 无法控制的可伸缩性

大多数软件系统创建时都有一个隐含的假设：整个系统处在一个实体的控制之下，或者至少参与到系统中的所有实体都是向着一个共同的目标行动，而不是有着各自不同的目标。这样的假设当系统在 Internet 上开放地运行时，无法安全地满足。无法控制的可伸缩性是指，因为架构元素可能会与在它们的组织的控制范围之外的元素进行通信，它们需要在遭遇到以下情况时仍然能够继续正常运行：无法预测的负载量、特殊的不良格式或恶意构造的数据。该架构必须要服从于加强可见性和可伸缩性的机制。

无法控制的可伸缩性需求被应用于所有的架构元素。不能期待客户端保持所有服务器的信息，也不能期待服务器跨多个请求保持状态的信息。超媒体数据元素不能保持 “回退指针”（back-pointers，即引用每个数据元素的标识符），因为对一个资源的引用的数量与对此信息感兴趣的人数是成正比的（译者注：即，不可能为每一个人都保持一个回退指针）。特别是有报道价值的信息能够导致“闪电式拥塞”：当可获得此信息的消息传遍世界时，网站的访问量会出现突发的尖峰。

架构元素的安全性和它们的运行平台也成为了一个重大的关注点。多个组织边界意味着在任何通信中都可能会存在的多个信任边界。中间应用（intermediary applications□ 例如防火墙）应该能够检查应用的交互，并且阻止交互去做那些超越本组织的安全策略之外的事情。应用的交互的参与者应该要么假设接收到的任何信息都是不可信的，要么在确认信息可信之前要求一些额外的认证。这要求该架构有能力沟通与认证数据和授权控制有关的信息。然而，因为认证降低了可伸缩性，架构的默认操作应该被限制在不需要可信数据（trusted data）的动作上：即一组具有良好定义的语义的安全操作。

4.1.4.2 独立部署

多个组织边界也意味着系统必须准备好应对逐渐的和片段的修改，一些旧的组件实现将会与一些新的组件实现共存，而不会妨碍新的组件实现使用它们的扩展功能。现有的架构元

素需要被设计为考虑到以后将会添加新的架构功能。同样地，旧的实现需要易于识别，这样遗留的行为能够被封装起来，而不会对新的架构元素带来不利的影响。架构作为一个整体，必须被设计为易于以一种部分的、迭代的方式来部署架构元素，因为强制以一种整齐划一的方式来部署是不可能的。

4.2 问题

到了 1993 年末，很明显已经不仅仅是研究者对 Web 感兴趣了。Web 首先被小型研究团体所采用，然后被传播到校区宿舍、俱乐部、个人主页、以及发布校园信息的各个科系。当个人开始对那些令他们着迷的话题发布个人信息时，这种社会化的网络效应使得网站以指数的速度增长，一直持续至今。虽然对于 Web 在商业上的兴趣才刚刚开始，但是很明显在国际性的范围发布信息的能力对于商业来说具有无法抵挡的诱惑力。

尽管为 Web 的成功而欢欣鼓舞，但 Internet 开发者社区开始担心 Web 使用的快速增长率，伴随早期 HTTP 的一些糟糕的网络特性，将会很快超越 Internet 基础设施的容量，并且导致全面的崩溃。Web 应用的交互性质的变化更加恶化了这种情况。尽管最初的协议是为单个的请求响应对（request-response pairs）而设计的，新的站点使用了越来越多的内嵌图片（in-line images）作为网页内容的一部分，这导致了不同的浏览交互模式（interaction profile）。已部署的架构在对可扩展性、共享缓存、中间组件的支持等方面存在着严重的局限，这使得开发解决增长率问题的特别解决方案非常困难。同时，软件市场中的商业竞争导致了新的提议和一些有时候与 Web 协议相矛盾的提议层出不穷。

在 Internet 工程工作组（IETF）中形成了三个工作组，为 Web 的三个主要的标准而工作：URI、HTTP 和 HTML。这些工作组的主要任务是定义在现有的早期 Web 架构中被公共地、一致地实现的架构通信的子集，并且识别出在这个架构中存在的问题，然后指定一组标准来解决那些问题。这给我们带来了一个挑战：我们如何将一组新的功能引入到一个已经被广泛部署的架构中，以及如何确保新功能的引入不会对那些使 Web 成功的架构属性带来不利的甚至是毁灭性的影响。

4.3 推导方法（Approach）

早期的 Web 架构基于一些可靠的原则：分离关注点、简单性、通用性，但是缺乏对于架构的描述和基本原理。其设计基于一组非形式化的超文本笔记[14]、两份早期的面向用户社区的论文[12, 13]、以及已归档的 Web 开发者社区邮件列表（www-talk@info.cern.ch）中的讨论。然而，事实上，对于早期 Web 架构的真正描述，出现在 libwww（用于客户端和服务端的 CERN 协议库）和 Mosic（NCSA 开发的浏览器客户端）的实现中，以及与它们互操作的一些其他的实现中。

一种架构风格能够被用来定义 Web 架构背后的原则，这样这些原则对于未来的架构就是可见的了。如同在第 1 章中所讨论的那样，一种风格是一组已命名的架构元素之上的约束，它导致了一组架构想要得到的属性。因此，我的推导方法的第一步，就是识别出那些负责导致想要得到的属性的一组存在于早期 Web 架构中的约束。

> 假设一：在 WWW 架构背后的设计基本原理能够通过一种由应用于 Web 架构中的元素之上的约束组成的架构风格来描述。

为了扩展在架构实例上所导致的属性，可以在一种架构风格上应用额外的约束。我的推导方法的下一步是识别出在一个 Internet 规模的分布式超媒体系统中想要得到的属性，然后

选择额外的会导致那些属性的架构风格，将它们与早期的 Web 约束相结合，形成一种新的、混合的现代 Web 架构的架构风格。

> 假设二：能够为 WWW 架构风格添加约束，从而获得更好地反映一个现代 Web 架构想要得到的属性的新的混合风格。

使用新的架构风格作为指导，我们能够对被提议的扩展与针对风格中的约束对 Web 架构所做的修改进行比较。存在冲突表明这个提议会违反一个或多个在 Web 背后的设计原则。在一些情况下，一旦新的功能被使用时，通过要求提供一个特定的指示，能够去除存在的冲突。对于影响一个响应的默认可缓存能力（cacheability）的 HTTP 扩展而言，这样做是有效的。对于严重的冲突，例如改变交互风格，要么使用更加有益于 Web 风格的设计来替代相同的功能，要么告知提议人将此功能实现为与 Web 并行运行的单独的架构。

> 假设三：修改 Web 架构的提议能够与更新后的 WWW 架构风格进行比较和分析，以便在部署之前识别出存在的冲突。

修订后的协议标准是根据新的架构风格的指导来编写的。最后，如同修订后的协议标准中定义的那样，更新后的 Web 架构通过参与到基础设施（infrastructure）和中间件软件（middleware software）的开发过程中来进行部署，它们组成了大多数的 Web 应用。这包括了我直接参与 Apache HTTP 服务器项目和 libwww-perl 客户端库的软件开发而得到的直接经验，以及通过为 W3C 的 libwww 和 jigsaw 项目、Netscape Navigator、Lynx、MSIE 这三种浏览器、还有一大堆其他实现的开发者提供建议而得到的间接经验，这些建议是 IETF 演讲的一部分。

尽管我是以单一的顺序来描述这个推导方法的，但是它实际上是以一种无顺序的、迭代的方式来应用的。在过去的六年中，我一直在构建模型、为架构风格添加约束、通过客户端和服务端软件的实验性扩展来测试这些约束对于 Web 协议标准的影响。同样地，其他人也曾建议为架构添加某些功能，这些功能超出了我的当前模型风格的范围，但是并不与该风格相冲突，这导致我回过头去修订架构的约束，以便更好地反映改进后的架构。我的目标就是总是维持一个一致的、正确的、反映出我所希望的 Web 架构应该如何运转的模型，这样就能够使用它来指导定义适当行为的协议标准，而不是创建一种仅仅局限于当工作开始之初所设想到的那些约束的人造模型。

4.4 小结

本章提出了万维网的架构需求，以及在设计和评估被提议的对万维网关键通信协议的改进的过程中所遇到的问题。这里的挑战是：开发一个用来设计架构改进的方法，使得能够在改进部署之前对它们进行评估。我的方法是使用一种架构风格来定义和改进在 Web 架构背后的设计基本原理，使用风格作为严格的测试，在被提议的扩展部署之前，对这些扩展加以验证，并且通过将修订后的架构直接应用在已经创建了 Web 基础设施的软件开发项目中，来部署修订后的架构。

下一章介绍并详细描述了为分布式超媒体系统设计的表述性状态转移（REST）架构风格，该风格被开发出来，用来代表现代 Web 应该如何运转的模型。REST 提供了一组架构约束，当作为一个整体来应用时，强调组件交互的可伸缩性、接口的通用性、组件的独立部署、以及用来减少交互延迟、增强安全性、封装遗留系统的中间组件。

第 5 章 表述性状态转移（REST）

本章介绍并详细描述了为分布式超媒体系统设计的表述性状态转移（REST）架构风格，描述了指导 REST 的软件工程原则和选择用来支持这些原则的交互约束，并将它们与其他架构风格的约束进行了对比。REST 是从第 3 章描述的几种基于网络的架构风格中衍生出来的一种混合架构风格，并且添加了一些额外的约束，用来定义统一的连接器接口。我使用第 1 章中的软件架构框架来定义 REST 的架构元素，并检查原型架构的过程样本、连接器和数据视图。

5.1 推导 REST

Web 架构背后的设计基本原理，能够被描述为由一组应用于架构中元素之上的约束组成的架构风格。当将每个约束添加到进化中的风格时，会产生一些影响。通过检查这些影响，我们就能够识别出 Web 的约束所导致的属性。然后就能够应用额外的约束来形成一种新的架构风格，这种风格能够更好地反映出现代 Web 架构所期待的属性。本节通过简述 REST 作为架构风格的推导过程，提供了关于 REST 的总体概览，后面各节将会详细描述组成 REST 风格的各种特定约束。

5.1.1 从“空”风格开始

无论是建筑还是软件，人们对架构设计的过程都有着两种常见的观点。第一种观点认为设计师一切从零开始——一块空的石板、白板、或画板——并使用熟悉的组件建造出一个架构，直到该架构满足希望的系统需求为止。第二种观点则认为设计师从作为一个整体的系统需求出发，此时没有任何约束，然后增量地识别出各种约束，并将它们应用于系统的元素之上，以便对设计空间加以区分，并允许影响到系统行为的力量（forces）与系统协调一致，自然地流动。第一种观点强调创造性和无限的想象力，而第二种观点则强调限制和对系统环境的理解。REST 是使用后一种过程发展而成的。随着增量地应用一组约束，已应用的约束会将架构的过程视图区分开，图 5-1 至 5 -8 以图形化的方式依次描述了这个过程。

“空”风格（图 5-1）仅仅是一个空的约束集合。从架构的观点来看，空风格描述了一个组件之间没有明显边界的系统。这就是我们描述 REST 的起点。

图 5-1：“空”风格

5.1.2 客户-服务器

首先被添加到我们的混合风格中的约束来自 3.4.1 小节描述的客户-服务器架构风格（图 5-2）。客户-服务器约束背后的原则是分离关注点。通过分离用户接口和数据存储这两个关注点，我们改善了用户接口跨多个平台的可移植性；同时通过简化服务器组件，改善了系统的可伸缩性。然而，对于 Web 来说，最重要的是这种关注点的分离允许组件独立地进化，从而支持多个组织领域的 Internet 规模的需求。

图 5-2：客户-服务器风格

5.1.3 无状态

我们接下来再为客户-服务器交互添加一个约束：通信必须在本质上是无状态的，如 3.4.3 小节中的客户-无状态-服务器（CSS）风格那样，因此从客户到服务器的每个请求都必须包含理解该请求所必需的所有信息，不能利用任何存储在服务器上的上下文，会话状态因此要全部保存在客户端。

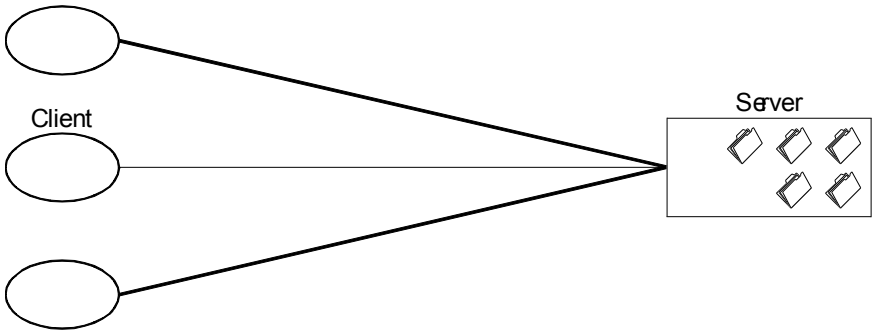

图 5-3：客户-无状态-服务器风格

这个约束导致了可见性、可靠性和可伸缩性三个架构属性。改善了可见性是因为监视系统不必为了确定一个请求的全部性质而去查看该请求之外的多个请求。改善了可靠性是因为它减轻了从局部故障[133]中恢复的任务量。改善了可伸缩性是因为不必在多个请求之间保存状态，从而允许服务器组件迅速释放资源，并进一步简化其实现，因为服务器不必跨多个请求管理资源的使用。

与大多数架构上抉择一样，无状态这一约束反映出设计上的权衡。其缺点是：由于不能将状态数据保存在服务器上的共享上下文中，因此增加了在一系列请求中发送的重复数据（每次交互的开销），可能会降低网络性能。此外，将应用状态放在客户端还降低了服务器对于一致的应用行为的控制，因为这样一来，应用就得依赖于跨多个客户端版本（译者注：例如多个浏览器窗口）的语义的正确实现。

5.1.4 缓存

为了改善网络的效率，我们添加了缓存约束，从而形成了 3.4.4 小节描述的客户-缓存-无状态-服务器风格（图 5-4）。缓存约束要求一个请求的响应中的数据被隐式地或显式地标记为可缓存的或不可缓存的。如果响应是可缓存的，那么客户端缓存就可以为以后的相同请求重用这个响应的数据。

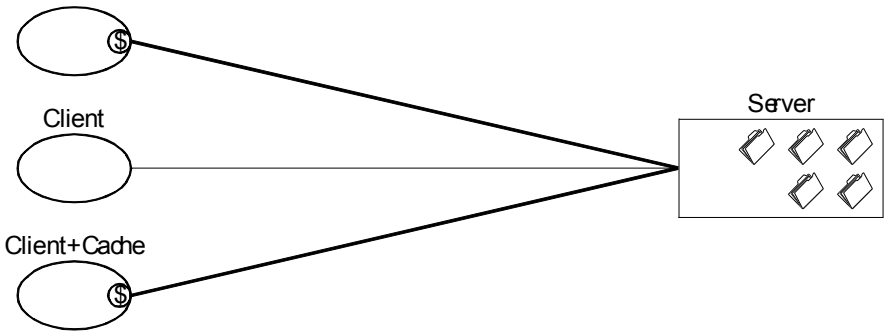

图 5-4：客户-缓存-无状态-服务器风格

添加缓存约束的好处在于，它们有可能部分或全部消除一些交互，从而通过减少一系列交互的平均延迟时间，来提高效率、可伸缩性和用户可觉察的性能。然而，付出的代价是，如果缓存中陈旧的数据与将请求直接发送到服务器得到的数据差别很大，那么缓存会降低可靠性。

早期的 Web 架构，如图 5-5 所示[11]，是通过客户-缓存-无状态-服务器的约束集合来定义的。也就是说，1994 年之前的 Web 架构的设计基本原理聚焦于在 Internet 上交换静态文档的无状态的客户-服务器交互。交互的通信协议仅包含了对非共享缓存的初步支持，但是并没有限定接口要对所有的资源提供一组一致的语义。相反，Web 依赖于使用一个公共的客户-服务器实现库（CERN 的 libwww）来维护 Web 应用之间的一致性。

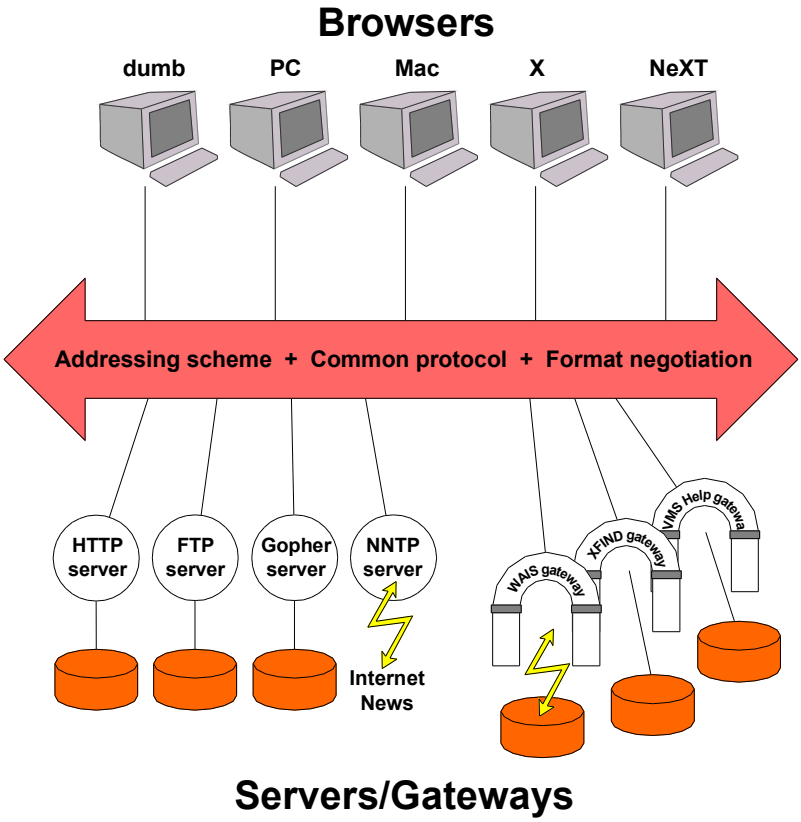

图 5-5：早期 WWW 的架构图

Web 实现的开发者早已经超越了这种早期的设计。除了静态的文档之外，请求还能够识别出动态生成响应的服务，例如图像地图（image-maps）[Kevin Hughes]和服务器端脚本（server-side scripts）[Rob McCool]。人们也以代理[79]和共享缓存[59]的形式开展了对中间组件的研究，但是为了使中间组件能够可靠地通信，还需要对现有的协议进行扩展。以下几小节描述了添加到 Web 架构风格中的约束，以便用来对形成现代 Web 架构的扩展加以指导。

5.1.5 统一接口

使 REST 架构风格区别于其他基于网络的架构风格的核心特征是，它强调组件之间要有一个统一的接口（图 5-6）。通过在组件接口上应用通用性的软件工程原则，整体的系统架构得到了简化，交互的可见性也得到了改善。实现与它们所提供的服务是解耦的，这促进了独立的可进化性。然而，付出的代价是，统一接口降低了效率，因为信息都使用标准化的形式来转移，而不能使用特定于应用的需求的形式。REST 接口被设计为可以高效地转移大粒度的超媒体数据，并针对 Web 的常见情况做了优化，但是这也导致了该接口对于其他形式的架构交互并不是最优的。

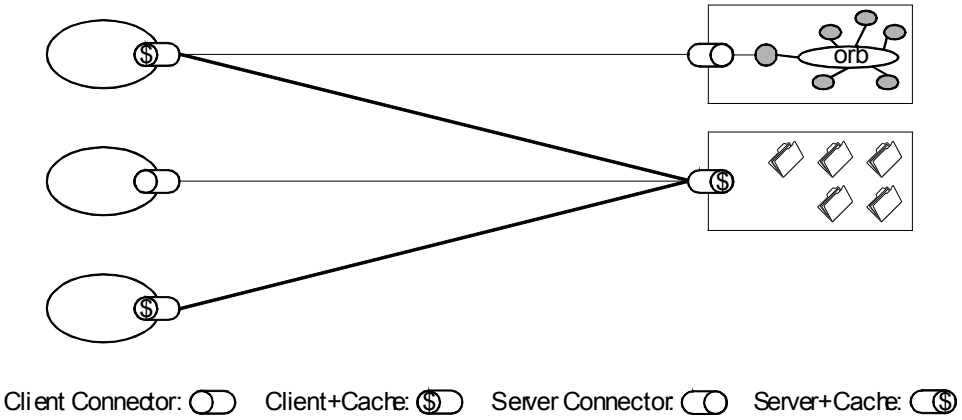

图 5-6：统一-客户-缓存-无状态-服务器风格

为了获得统一的接口，需要有多个架构约束来指导组件的行为。REST 由四个接口约束来定义：资源的识别（identification of resources）、通过表述对资源执行的操作、自描述的消息（self-descriptive messages）、以及作为应用状态引擎的超媒体。这些约束将在 5.2 节中讨论。

5.1.6 分层系统

为了进一步改善与 Internet 规模的需求相关的行为，我们添加了分层的系统约束（图 5-7）。正如 3.4.2 小节中所描述的那样，分层系统风格通过限制组件的行为（即，每个组件只能“看到”与其交互的紧邻层），将架构分解为若干等级的层。通过将组件对系统的知识限制在单一层内，为整个系统的复杂性设置了边界，并且提高了底层独立性。我们能够使用层来封装遗留的服务，使新的服务免受遗留客户端的影响，通过将不常用的功能转移到一个共享的中间组件中，从而简化组件的实现。中间组件还能够通过支持跨多个网络和处理器的负载均衡，来改善系统的可伸缩性。

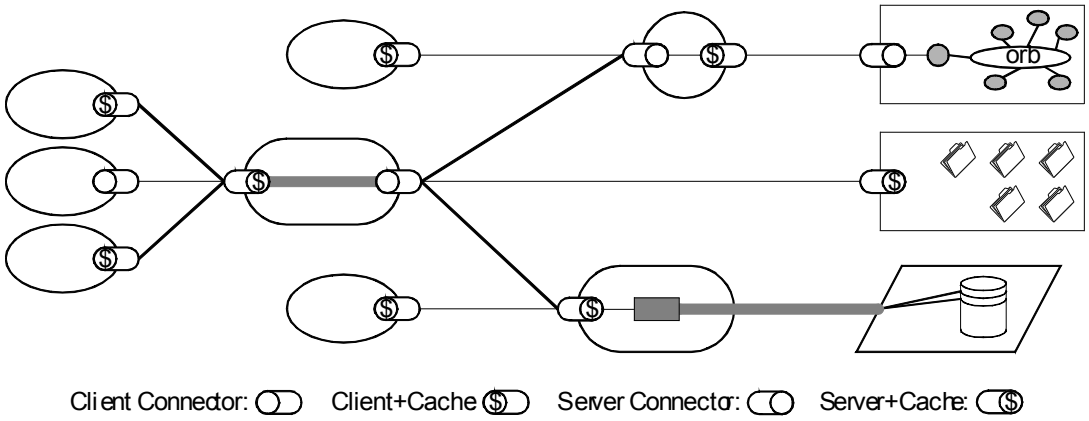

图 5-7：统一-分层-客户-缓存-无状态-服务器风格

分层系统的主要缺点是：增加了数据处理的开销和延迟，因此降低了用户可觉察的性能[32]。对于一个支持缓存约束的基于网络的系统来说，可以通过在中间层使用共享缓存所获得的好处来弥补这一缺点。在组织领域的边界设置共享缓存能够获得显著的性能提升[136]。这些中间层还允许我们对跨组织边界的数据强制执行安全策略，例如防火墙所要求的那些安全策略[79]。

分层系统约束和统一接口约束相结合，导致了与统一管道和过滤器风格（3.2.2 小节）类似的架构属性。尽管 REST 的交互是双向的，但是超媒体交互的大粒度的数据流每一个都能够被当作一个数据流网络来处理，其中包括一些有选择地应用在数据流上的过滤器组件，以便在数据传递的过程中对它的内容进行转换[26]。在 REST 中，中间组件能够主动地转换消息的内容，因为这些消息是自描述的，并且其语义对于中间组件是可见的。

5.1.7 按需代码

我们为 REST 添加的最后的约束来自于 3.5.3 小节中描述的按需代码风格（图 5-8）。通过下载并执行 applet 形式或脚本形式的代码，REST 允许对客户端的功能进行扩展。这样，通过减少必须被预先实现的功能的数目，简化了客户端的开发。允许在部署之后下载功能代码也改善了系统的可扩展性。然而，这也降低了可见性，因此它只是 REST 的一个可选的约束。

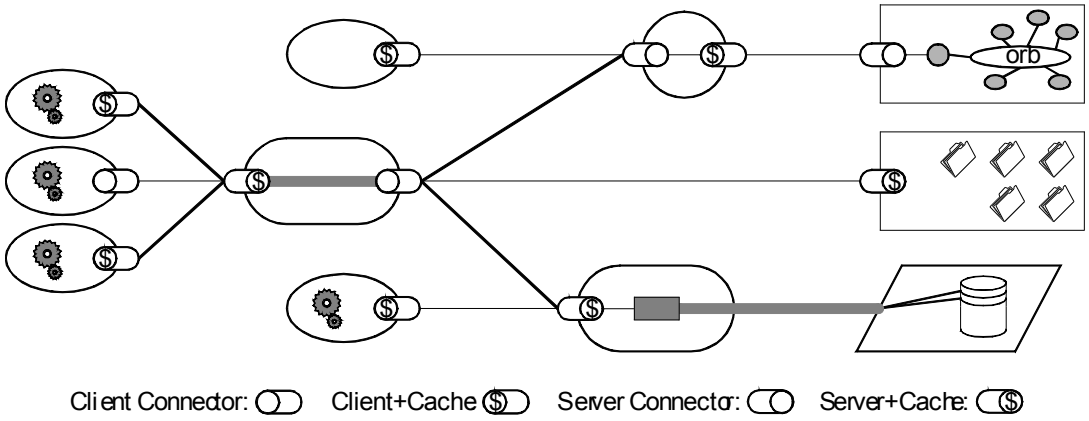

图 5-8：REST 风格

可选的约束的想法似乎有些矛盾。然而，在设计一个包含多个组织边界的系统的架构时，

它确实是有用的。这意味着只有当已知对于整个系统的某些领域有效的情况下，架构才会从可选的约束得到好处（或蒙受损失）。例如，如果已知一个组织中的所有客户端软件都支持 Java applet[45]，那么该组织中的服务就能够构造为可以通过下载 Java 类来增强客户端的功能，以便从可选的约束得到好处。然而，与此同时，该组织的防火墙可能会阻止转移来自外部资源的 Java applet，因此对于 Web 的其余部分来说，这些客户端似乎是不支持按需代码的。一个可选的约束允许我们设计在一般的场合下支持期待的行为的架构，但是我们需要理解，这些行为可能在某些环境中无法使用。

5.1.8 风格推导小结

REST 由一组选择用来在候选架构上导致想要得到的属性的架构约束组成。尽管这些约束每一个都能够独立加以考虑，但是根据它们在通用的架构风格中的来源来对它们进行描述，使得我们理解选择它们背后的基本原理更加容易。图 5-9 根据第 3 章中调查过的基于网络的架构风格图形化地描述了 REST 约束的来源。

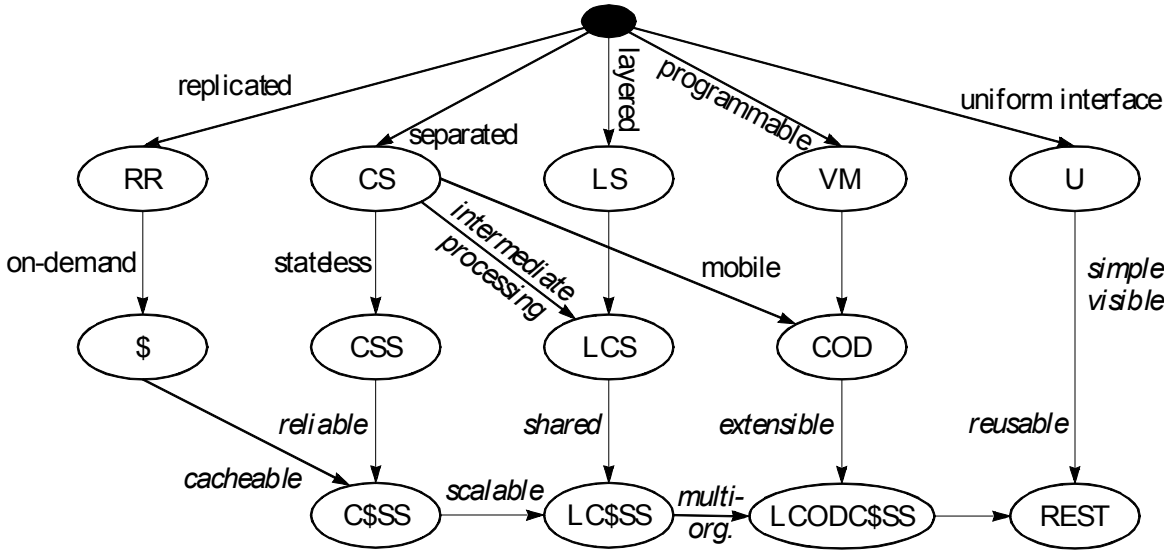

图 5-9：REST 所继承的风格约束

5.2 REST 架构的元素

表述性状态转移（REST）风格是对分布式超媒体系统中的架构元素的一种抽象。REST 忽略了组件实现和协议语法的细节，以便聚焦于以下几个方面：组件的角色、组件之间的交互之上的约束、组件对重要数据元素的解释。REST 包括了一组对于定义 Web 架构基础的组件、连接器和数据的基本约束，因此它代表了基于网络的应用的行为的本质。

5.2.1 数据元素（Data Elements）

在分布式对象风格[31]中，所有的数据都封装在数据的处理组件之中，并且被数据的处理组件隐藏起来。与分布式对象不同的是，架构的数据元素的性质和状态是 REST 的一个关键的方面。这一设计的基本原理能够在分布式超媒体的特性中看到。当选择了一个链接后，该链接所指向的信息需要从它的存储地移动到它的使用地，后者在大多数情况下是一个人类阅读者。这与很多其他的分布式处理模型[6,50]不同，在这些模型中，有可能而且通常也是更高效的做法是将“处理代理”（例如，可移动的代码、存储过程、搜索表达式等等）移动

到数据所在地，而不是将数据移动到其处理器的所在地。

一个分布式超媒体系统的架构师仅拥有三种基本的选项：1）在数据的所在地对数据进行呈现，并向接收者发送一个固定格式的映象（a fixed-format image）；2）将数据和呈现引擎封装起来并将两者一起发送给接收者；或者，3）发送原始数据和一些描述数据类型的元数据，这样接收者就能够选择它们自己的呈现引擎。

每一种选项都有其优点和缺点。第一种选项对应于传统的客户-服务器风格[31]，它使得与数据的真实性质有关的所有信息都被隐藏在数据发送者之中，防止了其他组件对数据结构作出假设，并且简化了客户端的实现。然而，它也严重限制了接收者的功能，并且将大部分处理负担都放在了发送者一边，这导致了可伸缩性的问题。第二种选项对应于可移动对象（mobile object）风格[50]，它提供了信息的隐藏，同时还可以通过唯一的呈现引擎来支持对于数据的专门处理。但是，它将接收者的功能限制在了引擎所能预测的范围之内，并且可能会大幅增加需要转移的数据量。第三种选项允许发送者保持简单性和可伸缩性，同时还使得需要转移的数据最小化。但是，它丧失了数据隐藏的优点，要求发送者和接收者都必须理解相同的数据类型。

通过聚焦于共享对于带有元数据的数据类型的理解，但是限制暴露为标准接口的操作的范围，REST 提供的是所有三种选项的一个混合体。REST 组件通过以一种数据格式转移资源的表述来进行通信，该格式与一组进化中的标准数据类型之一相匹配，可以基于接收者的能力和期待的内容、以及资源的性质来动态地选择不同的表述（译者注：不同的表述，体现在使用不同数据格式）。表述与其原始来源格式相同，还是由来源衍生但使用不同的格式，这些信息被隐藏在了接口的背后。可移动对象风格的好处通过以下方式来获得：发送一个表述，这个表述由一个封装过的呈现引擎（例如：Java[45]）的标准数据格式中的指令组成。REST 因此获得了客户-服务器风格的分离关注点的好处，而不存在服务器的可伸缩性问题，它允许通过一个通用的接口来隐藏信息，从而支持封装和服务的进化，并且可以通过下载功能引擎（feature-engine）来提供一组不同的功能。

REST 的数据元素总结于表 5-1。

表 5-1 REST 的数据元素

数据元素	现代 Web 实例
资源	一个超文本引用意图指向的概念上的目标
资源标识符	URL、URN
表述	HTML 文档、JPEG 图片
表述元数据	媒体类型、最后修改时间
资源元数据	源链接、alternates、vary
控制数据	if-modified-since、cache-control

5.2.1.1 资源和资源标识符（Resources and Resource Identifiers）

REST 对于信息的核心抽象是*资源*。任何能够被命名的信息都能够作为一个资源：一份文档或一张图片、一个与时间相关的服务（例如，“洛杉矶今日的天气”）、一个其他资源的集合、一个非虚拟的对象（例如，人）等等。换句话说，任何可能作为一个创作者的超文本引用的目标的概念都必须符合资源的定义。一个资源是到一组实体的概念上的映射，而不是在任何特定时刻与该映射相关联的实体本身。

更精确地说，资源 *R* 是一个随时间变化的成员函数 $M_R(t)$，该函数将时间 *t* 映射到等价的一个实体或值的集合，集合中的值可能是*资源的表述*和/或*资源的标识符*。一个资源可以映射到空集，这允许在一个概念的任何实现存在之前引用这个概念——这一观念对于 Web 之前的大多数超文本系统来说比较陌生[61]。一些资源在它们被创建后的任何时刻来检查，它们都对应着相同的值的集合，从这个意义上说它们是静态的。其他的资源所对应的值则会随时间而频繁地变化。对于一个资源来说，唯一必须是静态的是映射的语义，因为语义才是区别资源的关键。

例如，“一篇学术论文的创作者首选的版本”是一个其值会经常变化的映射；相反，到“X 会议学报中发表的论文”的映射则是静态的。它们是两个截然不同的资源，即使某个时刻它们可能会映射到相同的值。这种区别是必要的，这样两个资源就能够被独立地标识和引用。软件工程领域中一个类似的例子是版本控制系统的源代码文件的单独标识，这些标识可以是：“最新版本”、“版本号 1.2.7”，或“包含有 Orange 功能实现的修订版本”。

对资源的这一抽象的定义使得 Web 架构的核心功能得以实现。首先，它通过包含了很多信息来源而没有人为地通过类型或实现对它们加以区分，从而实现了通用性。其次，它允许引用到表述的延迟绑定（late binding），从而支持基于请求的性质来进行内容协商。最后，它允许一个创作者引用一个概念而不是引用此概念的某个单独的表述，从而使得当表述改变时无须修改所有的现有链接（假设创作者使用了正确的标识符）。

REST 使用一个*资源标识符*来标识组件之间交互所涉及到的特定资源。REST 连接器提供了访问和操作资源的值集合的一个通用的接口，而无须关心其隶属函数（membership function）是如何定义的，或者处理请求的软件是何种类型。由命名权威（naming authority）来为资源分配资源标识符，使得引用资源成为可能。由同样的命名权威来负责维护映射的语义有效性（例如，确保隶属函数不会改变）。

传统的超文本系统[61]通常只在一个封闭的或局部的环境中运行，它们使用随信息的变化而改变的唯一节点或文档标识符，并依赖链接服务器（link server）以独立于内容的方式来维护引用[135]。因为集中式的链接服务器无法满足 Web 的超大规模和多个组织领域的需求，所以 REST 采用了其他的方式——依赖资源的创作者来选择最符合被标识的概念本质的资源标识符。很自然，标识符的质量常常与为保持其有效性所花费的金钱成正比，因此随着暂时性的（或未被良好支持的资源）信息移动或消失，会导致出现破损的链接。

5.2.1.2 表述（Representations）

REST 组件通过以下方式在一个资源上执行动作：使用一个表述来捕获资源的当前的或预期的状态、在组件之间传递该表述。一个*表述*是一个字节序列，以及描述这些字节的*表述元数据*。表述的其他常用但不够精确的名称包括：文档、文件、HTTP 消息实体、实例或变量。

表述由数据、描述数据的元数据、以及（有时候存在的）描述元数据的元数据组成（通常用来验证消息的完整性）。元数据以名称-值对的形式出现，其中的名称对应于一个定义值的结构和语义的标准。响应消息可以同时包括表述元数据和*资源元数据*（关于资源的信息，并不特定于所提供的表述）。

*控制数据*定义了组件之间的消息的用途，例如被请求的动作或响应的含义。它也被用来提供请求的参数，以及覆盖某些连接元素（connecting elements）的默认行为。例如，可以使用包括在请求或响应消息中的控制数据来修改缓存的行为。

依赖于消息中的控制数据，一个特定的表述可能表示的是被请求资源当前的状态或预期的状态，或者某个其他资源的值，例如一个客户端查询表单中的输入数据的表述，或通过一个响应返回的某种出错状况的表述。例如，对一个资源的远程创作需要创作者将资源的表述

发送到服务器，这就为该资源创建了一个以后的请求可以获取的值。如果一个资源在特定时刻的值集合由多个表述组成，可以使用内容协商来选择将包括在一个特定消息中的最佳表述。

表述的数据格式被称为一种*媒体类型*[48]。一个表述能够被包括在一个消息中，并由接收者根据消息的控制数据和媒体类型的性质来做处理。媒体类型有些是用来做自动处理的，有些是用来呈现给用户来查看的，还有少数是可以同时用于两种用途的。组合的媒体类型能够被用来将多个表述封装在单个消息之中。

媒体类型的设计能够直接影响到一个分布式超媒体系统的用户可觉察的性能。在接收者能够开始对表述做呈现之前必须要接收到的任何数据都会增加交互的延迟。一种数据格式如果将最重要的呈现信息放在前面，允许正在接收剩余的信息的同时对最初的信息进行增量地呈现，这样将导致的用户可觉察的性能要比那些必须在呈现开始前接收全部信息的数据格式好得多。

例如，即使在网络性能相同的情况下，能够在接收信息的同时增量地呈现大型 HTML 文档的 Web 浏览器所提供的用户可觉察性能，要比那些在呈现之前要等待整个文档完全接收的浏览器好得多。需要注意的是，表述的呈现能力也会受到对于内容的选择的影响。如果动态尺寸的表格的尺寸及其内嵌的对象必须要在它们能够被呈现之前确定，那么它们出现在一个超媒体页面的显示区域中就会增加显示该表格的延迟。

5.2.2 连接器（Connectors）

如表 5-2 所总结的那样，REST 使用多种不同的连接器类型来对访问资源和转移资源表述的活动进行封装。连接器代表了一个组件通信的抽象接口，通过提供清晰的关注点分离、并且隐藏资源的底层实现和通信机制，从而改善了架构的简单性。接口的通用性也使得组件的可替换性成为了可能：如果用户对系统的访问仅仅是通过一个抽象的接口，那么接口的实现就能够被替换，而不会对用户产生影响。由于组件的网络通信是由一个连接器来管理的，所以在多个交互之间能够共享信息，以便提高效率和响应能力。

表 5-2 REST 的连接器

连接器	现代 Internet 实例
客户端	libwww、libwww-perl
服务器	libwww、Apache API、NSAPI
缓存	浏览器缓存、Akamai 缓存网络
解析器（resolver）	绑定（DNS 查找库）
隧道（tunnel）	SOCKS、HTTP 连接之后的 SSL

所有的 REST 交互都是无状态的。也就是说，无论之前有任何其他请求，每个请求都包含了连接器理解该请求所必需的全部信息。这个约束能够实现四个功能：1）它使得连接器无需保存请求之间的应用状态，从而降低了物理资源的消耗并改善了可伸缩性；2）它允许对交互进行并行处理，处理机制无需理解交互的语义；3）它允许中间组件孤立地查看并理解一个请求，当需要对服务作出动态安排时，这是必需要满足的；4）它强制每个请求都必须包含可能会影响到一个已缓存响应的可重用性的所有信息。

连接器接口与过程调用有些类似，但是在参数和结果的传递方式上有着重要的区别。其传入参数由请求的控制数据、一个表示请求的目标的资源标识符、以及一个可选的表述组成。其传出参数由响应的控制数据、可选的资源元数据、以及一个可选的表述组成。从一种抽象

的观点来看，调用是同步的，但是传入参数和传出参数都可以作为数据流来传递。换句话说，处理可以在完全知道参数的值（译者注：即数据流的全部数据）之前进行，从而避免了对于大量数据转移进行批量处理而产生的延迟。

主要的连接器类型是客户端和服务器。两者之间的本质区别是：*客户端*通过发送请求来发起通信；*服务器*侦听连接并对请求作出响应，以便为其服务提供访问的途径。一个组件可能包括客户端和服务器两种连接器。

第三种连接器类型是*缓存*连接器，可以位于客户端或服务器连接器的接口处，以便保存当前交互的可缓存的响应，这样它们就能够被以后的请求交互来重用。客户端可以使用缓存来避免重复的网络通信，服务器可以使用缓存来避免重复执行生成响应的处理，这两种情况都可以减小交互的延迟。一个缓存通常在使用它的连接器的地址空间内实现。

某些缓存连接器是共享的，这意味着它所缓存的响应可以被最初获得该响应的客户端以外的其他客户端所使用。共享缓存可以有效地减少受欢迎的服务器的负载出现“闪电拥塞”，的几率，尤其是当缓存以分等级的形式来安排，以便覆盖大量用户的时候，例如在公司的 Intranet 中、Internet 服务提供商的用户、或共享国家网络主干的大学。然而，如果被缓存的响应不能与新的请求原本应该获得的响应相匹配的话，共享缓存就会导致出错。REST 试图在透明的缓存行为和高效地使用网络这两个想要得到的属性之间取得平衡，而不是假设无论何时都需要绝对的透明性。

一个缓存有能力确定一个响应的可缓存性（cacheability），因为接口是通用的而不是特定于每个资源的。在默认情况下，数据获取请求（retrieval request）的响应是可缓存的，其他类型的请求的响应是不可缓存的。如果请求中包含了某种形式的用户认证信息，或者响应明确表示它不应该被共享，那么该响应就只能被缓存在非共享缓存中。一个组件能够通过包括控制数据来覆盖这些默认的行为，使得交互变成是可缓存的、不可缓存的、或者仅在有限时间内是可缓存的。

一个*解析器*负责将部分或完整的资源标识符翻译成创建组件间连接所需的网络地址信息。例如，大多数 URI 都包括一个 DNS 主机名，作为一种机制来标识该资源的命名权威。为了发起一个请求，一个 Web 浏览器会从 URI 中提取出主机名，并利用 DNS 解析器来获得该权威的 Internet 协议（IP）地址。另一个例子是某些识别模式（例如 URN[124]）要求一个中间组件将一个永久标识符翻译为一个更加短暂的地址，以便访问被标识的资源。使用一个或多个中间解析器（intermediate resolvers）能够通过增加间接层的方式来延长资源引用的寿命，尽管这样做会增加请求的延迟。

连接器类型的最后一种形式是*隧道*，它简单地跨连接的边界对通信进行中继，例如一个防火墙或更低层的网关。隧道作为 REST 的一部分来建模，而不是作为网络基础设施的一部分来进行抽象，唯一的原因是某些 REST 组件可能会动态地从主动的组件行为切换到一个通道。主要的例子是当响应一个 CONNECT 方法请求时，HTTP 代理会切换到一个隧道[71]，从而允许其客户使用一种不同的协议（例如 TLS）来直接与不支持代理的远程服务器通信。当两端终止通信时，隧道就会消失。

5.2.3 组件（Components）

REST 组件根据它们在整个的应用动作（application action）中的角色来进行分类，总结于表 5-3。

表 5-3 REST 的组件

组件	现代 Web 实例
来源服务器（origin server）	Apache httpd、微软 IIS
网关（gateway）	Squid、CGI、反向代理
代理（proxy）	CERN 代理、Netscape 代理、Gauntlet
用户代理（user agent）	Netscape Navigator、Lynx、MOMspider

一个*用户代理*使用一个客户端连接器发起请求，并成为响应的最终接收者。最常见的例子是一个 Web 浏览器，它提供了对信息服务的访问途径，并且根据应用的需要呈现服务的响应。

一个*来源服务器*使用一个服务器连接器管理被请求资源的名字空间。来源服务器是其资源表述的权威数据来源，并且必须是任何想要修改资源的值的请求的最终接收者。每个来源服务器都为其服务提供了一个以资源的层次结构形式出现的通用的接口。资源的实现细节被隐藏在这一接口的背后。

为了支持转发，可能还要对请求和响应进行转换，中间组件同时扮演了客户端和服务器两种角色。一个*代理*组件是由客户端选择的中间组件，用来为其他的服务、数据转换、性能增强（performance enhancement）、或安全保护（security protection）提供接口封装。一个*网关*（也叫作反向代理）组件是由网络或来源服务器强加的中间组件，用来为其他的服务、数据转换、性能增强，或安全增强（security enforcement）提供接口封装。需要注意的是，代理和网关之间的区别是，何时使用代理是由客户端来决定的。

5.3 REST 架构的视图

现在我们已经孤立地了解了 REST 的架构元素，我们能够使用架构视图[105]来描述这些元素如何协作以形成一个架构。为了展示 REST 的设计原则，需要使用三种视图——过程视图、连接器视图、数据视图。

5.3.1 过程视图（Process View）

架构的过程视图的主要作用是，通过展示数据在系统中的流动路径，得出组件之间的交互关系。不幸的是，一个真实系统的交互通常会涉及到大量的组件，导致整体的视图因受到细节的干扰而模糊不清。图 5-10 提供了一个基于 REST 的架构的过程视图，其中包括了对三个并行请求的处理。

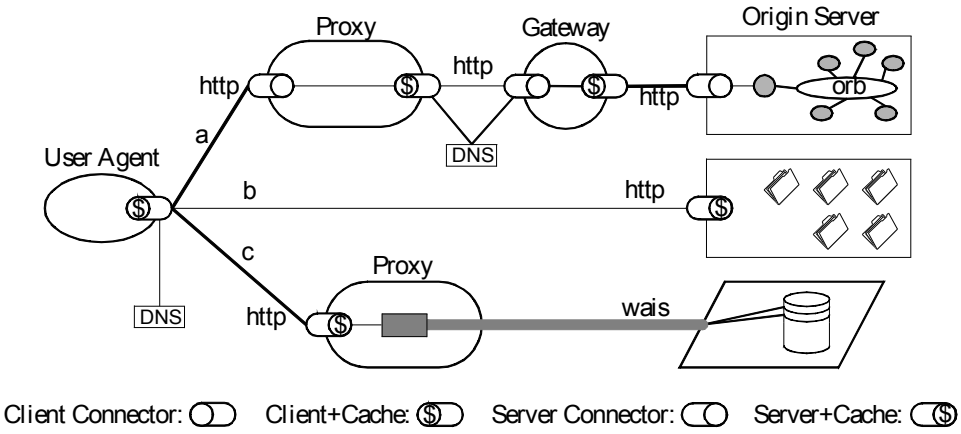

图 5-10：一个基于 REST 的架构的过程视图

一个用户代理正处在三个并行交互（a、b 和 c）的中途。用户代理的客户端连接器缓存无法满足请求，因此它根据每个资源标识符的属性和客户端连接器的配置，将每个请求路由到资源的来源。请求(a)被发送到一个本地代理，代理随后访问一个通过 DNS 查找发现的缓存网关，该网关将这个请求转发到一个能够满足该请求的来源服务器，服务器的内部资源由一个封装过的对象请求代理（object request broker）架构来定义。请求(b)直接发送到一个来源服务器，它能够通过自己的缓存来满足这个请求。请求(c)被发送到一个代理，它能够直接访问 WAIS（一种与 Web 架构分离的信息服务），并将 WAIS 的响应翻译为一种通用的连接器接口能够识别的格式。每一个组件只知道与它们自己的客户端或服务器连接器的交互；整个过程拓扑是我们的视图的产物。

REST 的客户-服务器关注点分离简化了组件的实现、降低了连接器语义的复杂性、改善了性能调优的效率、并且提高了纯服务器组件（pure server components）的可伸缩性。分层系统的约束允许在通信的不同地点引入中间组件——代理、网关、防火墙——而无须改变组件之间的接口，从而允许它们来辅助通信的转译（communication translation），或通过大规模的、共享的缓存来改善性能。REST 通过强制消息具有自描述性（请求之间的交互是无状态的、使用标准的方法和媒体类型来表达语义和交换信息、响应可以明确地表明其可缓存性）来支持中间组件的处理。

由于组件之间是动态连接的，因此它们对于一个特定的应用动作的排列和功能的性质与管道和过滤器风格类似。尽管 REST 组件通过双向的数据流来通信，但是每个方向的处理是独立的，因此容易受到数据流转换器（过滤器）的影响。通用的连接器接口允许基于每个请求或响应的属性将组件放置到数据流上。

服务也可以通过使用复杂的中间组件的层次结构和多个分布式来源服务器来实现。REST 的无状态本质允许每个交互独立于其他的交互，使其无须了解整体的组件拓扑结构（因为这对于一个 Internet 规模的架构来说是不可能的任务），并且允许组件要么作为目的地要么作为中间组件，由每个请求的目标来动态决定。连接器只需要在它们的通信范围期间知道彼此的存在即可，尽管它们可能会出于性能原因而对其他组件的存在和能力进行缓存。

5.3.2 连接器视图（Connector View）

架构的连接器视图集中于组件之间的通信机制。对一个基于 REST 的架构而言，我们对定义通用资源接口的约束尤其感兴趣。

客户端连接器检查资源标识符，以便为每个请求选择一个合适的通信机制。例如，一个客户端可以被配置为当资源的标识符表明其为一个本地资源时，连接到一个特定的代理组件（或许是作为一个注释过滤器）。同样地，客户端也可以被配置为拒绝对于标识符的某些子集的请求。

REST 并不限制通信只能使用一种特殊的协议，但是它会限制组件之间的接口，因此也

限制了交互的范围和在组件之间可能作出的有关实现的假设（implementation assumption）。例如，Web 的主要转移协议是 HTTP，但是 REST 架构也包括了对来自 Web 出现之前就已存在的网络服务器的资源，包括 FTP[107]、Gopher[7]和 WAIS[36]的无缝地访问。与那些服务的交互被限制为只能使用 REST 连接器的语义。为了获得连接器语义的单一的、通用的接口的好处，这一约束牺牲了其他架构的一些好处，例如像 WAIS 这样的相关性反馈协议（relevance feedback protocol）的有状态交互的好处。作为回报，通用的接口使得通过单个代理访问多个服务成为了可能。如果一个应用需要另一个架构的额外能力，它能够将其作为一个单独并行运行的系统来实现并调用那些能力，这与 Web 架构如何使用“telnet”和“mailto”资源是类似的。

5.3.3 数据视图（Data View）

一个架构的数据视图展示了信息在组件之间流动时的应用状态。因为 REST 被明确定位于分布式信息系统，它将一个应用看作是一种信息（information）和控制（control）的聚合体，用户可以通过这个聚合体执行他们想要完成的任务。例如，在一个在线字典上查找单词是一个应用、在一个虚拟博物馆里面观光是一个应用、为准备一门考试而复习课堂笔记也是一个应用。每个应用都定义了其底层系统的目标，可以针对这些目标来测量系统的性能。

组件之间的交互以动态改变尺寸的消息的形式来进行。小粒度的或中等粒度的消息用来控制交互的语义，但是应用的大部分工作需要通过包含一个完整的资源表述的大粒度消息来完成。请求的语义的最常见形式是获取资源的一个表述（例如 HTTP 中的“GET”方法），通常我们可以对其进行缓存以便以后重用。

REST 将所有的控制状态（control state）都浓缩在从交互的响应中接收到的表述之中。其目的是通过使服务器无须维护当前请求之外的客户端状态，从而改善服务器的可伸缩性。一个应用状态因此由以下几方面来定义：悬而未决的请求（pending requests）、相连接的组件（有些可能是过滤被缓冲的数据）的拓扑结构、连接器上活跃的请求、请求响应中表述的数据流、以及当用户代理接收到这些表述时对表述的处理。

无论何时当没有未完成的请求时（即，没有悬而未决的请求，并且所有当前请求集合的响应都已经被完全接收到，或者已经接收到了足够的数据，可以将其看作一个表述的数据流），一个应用就达到了一种稳定的状态。对于一个浏览器应用而言，这种状态对应于一个“网页”，包括了主要的表述和辅助的表述，例如内嵌的图片、内嵌的 applet、以及样式表。应用的稳定状态的重要性在于，它会同时影响用户可觉察的性能和网络请求流量的峰值。

一个浏览器应用的用户可觉察性能由稳定状态之间的延迟（从选择一个网页上的超媒体链接到下一个网页的可用信息呈现出来所需的时间）来决定。因此，浏览器性能的优化主要集中在降低这种通信的延迟上面。

因为基于 REST 的架构主要通过转移资源的表述来进行通信，所以延迟会同时受到通信协议的设计和表述数据格式的设计两方面的影响。当响应数据正在被接收时增量地呈现这些数据的能力，是由媒体类型的设计和每个表述中的布局信息（内嵌对象的视觉尺寸）的有效性来决定的。

我们观察到的一个有趣的事实是：最高效的网络请求是那些不使用网络的请求。换句话说，重用已缓存的响应结果的能力能够显著地改善应用的性能。尽管由于查找所带来的开销，使用一个缓存会为每个单独的请求增加一点延迟，但是请求的平均延迟会大幅降低，即使是在只有较小百分比的请求命中了缓存的情况下。

应用的下一个控制状态位于第一个被请求的资源的表述之中，因此获得第一个请求是一件需要优先完成的事情。REST 交互能够通过“先响应后思考”的协议来加以改进。换句话说，假设有两个协议，在第一个协议中，为了在发送一个内容响应之前做一些功能协商之类

的事情 ，每个用户动作都需要多次交互；在第二个协议中，先发送最有可能是最佳响应的内容，然后如果第一个响应无法满足客户端的需要，再提供一列替代选项给客户端来获取。第一个协议在感觉上会更慢一些。

用户代理负责控制和保存应用状态，并且这些状态可以由来自多个服务器的表述组成。除了使服务器免除了存储状态所带来的可伸缩性问题以外，这还允许用户直接操作状态（例如，Web 浏览器的历史信息）、预测状态的变化（例如，链接地图和表述的预先获取）、以及从一个应用跳转到另一个应用（例如，书签和 URI 栏目的对话框）。

因此，REST 的模型应用是一个引擎，它通过检查和选择当前的表述集合中的状态跃迁选项，从一个状态移动到下一个状态。毫不奇怪，这与一个超媒体浏览器的用户接口完全匹配。然而，REST 风格并不假设所有应用都是浏览器。事实上，通用的连接器接口对服务器隐藏了应用的细节，因此各种形式的用户代理都是等价的，无论是为一个索引服务执行信息获取任务的自动化机器人，还是查找匹配特定查询标准的数据的私人代理，或者是忙于巡视破损的引用或被修改的内容的维护爬虫（译者注：即执行维护任务的网络爬虫）。

5.4 相关工作

Bass 等人[9]在其文献中用了一章篇幅来介绍万维网的架构，但是他们的描述仅包括了 CERN/W3C 开发的 libwww（客户端和服务器库）和 Jigsaw 软件中的实现架构。这些实现是由熟悉 Web 架构的设计和基本原理的人们开发出来的，尽管它们反映出了很多 REST 的设计约束，但是真正的 WWW 架构是独立于任何单一实现的。现代 Web 是由它的标准的接口和协议来定义的，而不是由在软件的一个特定部分中如何实现这些接口和协议来定义的。

REST 风格来源于很多先前存在的分布式处理范例[6, 50]、通信协议、以及软件领域。虽然 REST 的组件交互被结构化为一个分层的客户-服务器风格，但是额外添加的通用的资源接口的约束使得替换中间组件和通过中间组件执行检查成为了可能。虽然请求和响应看上去像是远程调用风格，但是 REST 消息的目标是一个概念上的资源，而不是一个实现的标识符。

有一些研究尝试将 Web 架构建模为一种分布式文件系统的形式（例如，WebNFS）或者建模为一种分布式对象系统[83]。然而，他们排除了多种不同的 Web 资源类型或者实现策略，仅仅是因为“不感兴趣”，而实际上这些被排除的内容会使得这些模型之下的假设变得无效。REST 运作得很好，因为它并不将资源的实现局限于某些特定的预定义模型中，从而允许每个应用选择一种与它们自己的需求最匹配的实现，并支持在不影响用户的情况下对实现进行替换。

将资源的表述发送给消费组件（consuming components），这种交互方法与基于事件的集成（EBI）风格有些相似。其关键的区别是：EBI 风格是基于推模型的。包含状态的组件（等价于 REST 中的来源服务器）在状态改变时产生一个事件，无论事实上有没有组件对该事件感兴趣或在监听该事件。在 REST 风格中，消费组件通常需要自己去拉表述。尽管当单个客户端希望监视单个资源时，这种方式的效率要低一些（译者注：因为客户端需要对服务器做轮询），但是 Web 的规模使得我们不可能实现一种无节制的推模型。

在 Web 中有原则地使用包含了清晰的组件、连接器和表述概念的 REST 风格，这种方法与 C2 架构风格[128]有着密切的联系。C2 风格通过聚焦于结构化地使用连接器以获得底层独立性，支持开发分布式的、动态的应用。C2 应用依赖于状态改变的异步通知和请求消息。与其他基于事件的方案一样，C2 在名义上是基于推模型的，尽管 C2 架构也可以通过只在接收到请求时才发出通知，以 REST 的拉风格来运作。然而，C2 风格缺乏 REST 的中间组件友好的约束，例如通用的资源接口、保证无状态的交互，以及对于缓存的内在支持。

5.5 小结

本章介绍了为分布式超媒体系统设计的表述性状态转移（REST）架构风格。REST 提供了一组架构约束，当作为一个整体来应用时，强调组件交互的可伸缩性、接口的通用性、组件的独立部署、以及用来减少交互延迟、增强安全性、封装遗留系统的中间组件。我描述了指导 REST 的软件工程原则和选择用来支持这些原则的交互约束，并将它们与其他架构风格的约束进行了对比。

下一章通过从将 REST 应用于现代 Web 架构的设计、规范和部署的过程中学到的经验和教训，介绍了对于 REST 架构的一个评估。这些工作包括创作当前的超文本转移协议（HTTP/1.1）和统一资源标识符（URI）的 Internet 标准跟踪规范（standard-track specification），以及通过 libwww-perl 客户端协议库和 Apache HTTP 服务器来实现这个架构。

第 6 章 经验与评估

自从 1994 年以来，REST 架构风格就被用来指导现代 Web 架构的设计和开发。本章描述了在创作超文本转移协议（HTTP）和统一资源标识符（URI）的 Internet 标准（这两个规范定义了在 Web 上进行交互的所有组件使用的通用接口）的过程中，以及将这些技术部署在 libwww-perl 客户端库、Apache HTTP 服务器项目、以及协议标准的其他实现的过程中，应用 REST 所学到的经验和教训。

6.1 Web 标准化

如第 4 章所描述的那样，开发 REST 的动机是为 Web 应该如何运转创建一种架构模型，使之成为 Web 协议标准的指导框架。REST 被用来描述想要得到的 Web 架构，帮助识别出现有的问题，对各种替代方案进行比较，并且保证协议的扩展不会违反使 Web 成功的那些核心约束。这项工作是作为 Internet 工程工作组（IETF）和万维网协会（W3C）定义 Web 架构的标准（HTTP、URI 和 HTML）的工作的一部分来完成的。

我参与 Web 标准的开发过程开始于 1993 年晚期，当时我开发了 libwww-perl 协议库，作为 MOMspider[39]的客户端连接器接口。在那个时候，Web 的架构由一组非形式化的超文本节点[14]来描述，两篇早期的介绍性论文[12,13]草拟了超文本规范，展现出了一些提议的 Web 功能（一些已经被实现了），分布于世界各地的 WWW 项目的参与者使用公开的 www-talk 邮件列表进行非正式的讨论。当与各种 Web 的实现相比时，每一个规范都显得相当过时，这主要是因为在 Mosaic 图形化浏览器[NCSA]出现之后 Web 的快速进化。一些试验性的扩展被添加到 HTTP 中以支持 HTTP 代理，但是协议的大部分内容都假设在用户代理和来源 HTTP 服务器或一个到遗留系统的网关之间是一个直接的连接。在这个架构中，并不知道有缓存、代理或网络爬虫的存在，甚至是在它们的实现已经存在并且正常运行的情况下。还有很多其他的扩展被提议应该包括在下一版本的协议中。

与此同时，来自行业内的压力也不断增长，要求对 Web 接口协议的某个版本或某些版本进行标准化。Berners-Lee[20]组建了 W3C，对于 Web 的架构进行研究，并且为编写 Web 标准和参考实现提供创作资源，但是标准化本身是由 Internet 工程工作组[www.ietf.org]及其 URI、HTTP 和 HTML 工作组来掌管的。由于我在开发 Web 软件方面的经验，我被首先遴选出来创作相对 URL（Relative URL）的规范[40]，后来又与 Henrik Frystyk Nielsen 共同创作了 HTTP/1.0 规范[19]，然后我成为了 HTTP/1.1 的主要的架构师，并且最终创作了形成 URI 通用语法[21]标准的 URL 规范的修订版。

REST 的第一版开发于 1994 年 10 月和 1995 年 8 月之间，起初是作为当我编写 HTTP/1.0 规范和最初的 HTTP/1.1 建议时，用来沟通各种 Web 概念的一种方法。它在随后的 5 年中以迭代的方式不断改进，并且被应用于各种 Web 协议标准的修订版和扩展之中。REST 最初被称作“HTTP 对象模型”，但是那个名称常常引起误解，使人们误以为它是一个 HTTP 服务器的实现模型。这个名称“表述性状态转移”是有意唤起人们对于一个良好设计的 Web 应用如何运转的印象：一个由网页组成的网络（一个虚拟状态机），用户通过选择链接（状态转移）在应用中前进，导致下一个页面（代表应用的下一个状态）被转移给用户，并且呈现给他们，以便他们来使用。

REST 并非是想要捕获 Web 协议标准的所有可能的使用方法。仍然存在着一些与分布式超媒体系统的应用模型不匹配的 HTTP 应用和 URI 应用。然而，重要的是 REST 确实能够完全捕获一个分布式超媒体系统的那些被认为是 Web 的行为和性能需求的核心的方面，这样在这个模型中对行为进行优化，将能够导致在已部署的 Web 架构中得到最适宜的行为。换句话说，REST 是为常见的情况优化过的，这样它所应用于 Web 架构上的那些约束也同样是

为常见的情况优化过的。

6.2 将 REST 应用于 URI

统一资源标识符（URI）既是 Web 架构的最简单的元素，也是最重要的元素。URI 还有很多的名称：WWW 地址、通用文档标识符、通用资源标识符[15]、以及最后出现的统一资源定位器（URL）和统一资源名称（URN）的组合。除了它的名称以外，URI 的语法自从 1992 年以来维持相对稳定。然而，Web 地址的规范也定义了我们所称之为的“资源”的概念的范围和语义，这个概念自从早期的 Web 架构以来发生了变化。REST 被用来为 URI 标准[21]定义术语“资源”，也被用来定义通过它们的表述来操作资源的通用接口的全部语义。

6.2.1 重新定义资源

早期 Web 架构将 URI 定义为文档的标识符。创作者得到的指导是按照网络上一个文档的地点来定义标识符。然后能够使用 Web 协议来获取那个文档。然而，有很多理由可以证实，这个定义并不是很令人满意。首先，它暗示创作者正在标识被转移的内容，这意味着任何时候当内容改变了，这个标识符都应该改变。其次，存在着很多地址对应于一个服务，而不是一个文档——创作者可能是有意将读者引导到那个服务，而不是引导到来自预先访问那个服务而获取到的特定的结果。最后，存在着一些地址在某段时间内没有对应一个文档，例如当文档尚不存在，或者当地址仅仅被用来进行命名，而不是被用来进行定位和获取信息时。

在 REST 中对于“资源”的定义基于一个简单的前提：标识符的改变应该尽可能很少发生。因为 Web 使用内嵌的标识符，而不是链接服务器，创作者需要一个标识符，这个标识符能够紧密地匹配他们想要通过一个超媒体引用来表达的语义，允许这个引用保持静态，甚至是在访问该引用所获得的结果可能会随时间而变化的情况下。REST 达到了这个目标，通过将一个资源定义为创作者想要标识的语义，而不是对应于创建这个引用时的那些语义的值。然后留给创作者来保证所选择的这个标识符确实真正标识出了他所想要表达的语义。

6.2.2 操作影子（Manipulating Shadows）

将“*资源*”定义为一个 URI 标识了一个概念，而不是标识了一个文档，这给我们带来了另一个问题：一个用户如何访问、操作或转移一个概念，使得他们在选择了一个超文本链接后能够得到一些有用的东西。REST 通过定义在被标识的资源的“*表述*”之上执行的操作，而不是在资源本身之上执行的操作回答了这个问题。一个来源服务器维护着从资源的标识符到每个资源相对应的表述集合的映射，因此可以通过由资源标识符定义的通用接口转移资源的表述来操作一个资源。

REST 对于资源的定义来源于 Web 的核心需求：独立创作跨多个可信任域的互相连接的超文本。强制接口的定义与接口的需求相匹配会使得协议似乎含糊不清，但这仅仅是因为被操作的接口仅仅是一个接口，而不是一个实现。这些协议是与一个应用动作的意图密切相关的，但是接口背后的机制必须要确定该意图如何来影响底层实现中资源到表述的映射。

这里所隐藏的信息是关键的软件工程原则之一，也就是 REST 使用统一接口的动机。因为客户端被限制为只能对资源的表述执行操作，而不是直接访问资源的实现，因此资源的实现可以以任何命名权威所希望的形式来建造，而不会影响到使用资源的表述的客户端。此外，如果当资源被访问时，存在着资源的多个表述，可以使用一个内容选择算法来动态地选择一个最适合客户端能力的表述。当然，其缺点就是对资源进行远程创作不像对文件进行远程创作那么直接。

6.2.3 远程创作（Remote Authoring）

通过 Web 的统一接口执行远程创作的挑战在于：能够被客户端获取到的表述与服务器端所使用的保存、生成或获取表述内容的机制之间是相互分离的。一个单独的服务器可以将它的名字空间的一部分映射到一个文件系统，文件系统随后映射到一个 i 节点，随后再映射到一个磁盘位置，但是这些底层机制提供了一种将一个资源与一组表述相关联的方法，而不是标识资源本身。很多不同的资源能够映射到相同的表述，而其他资源可能完全没有映射到的表述。

为了对一个现有的资源进行创作，创作者必须首先获得特定资源的 URI：绑定到目标资源的处理器底层表述上的 URI 集合。一个资源并不总是映射到单个的文件，但是所有非静态的资源来自于某些其他资源，通过跟踪继承树，一个创作者能够最终找到所有必须编辑的资源，以便修改一个资源的表述。这些相同的原则适用于以任何形式继承而来的表述，无论它是来自内容协商、脚本、servlet、托管的配置（managed configurations）、翻译（versioning）等等。

资源并不是存储对象（storage object）。资源并不是一种服务器用来处理存储对象的机制。资源是一种概念上的映射——服务器接收到标识符（标识这个映射），将它应用于当前的映射实现（mapping implementation，通常是与特定集合相关的树的深度遍历和/或哈希表的组合）上，以发现当前负责处理该资源的处理器实现，然后处理器实现基于请求的内容选择适当的动作+响应。所有这些特定于实现的问题都隐藏在 Web 接口之后，它们的性质无法由仅能够通过 Web 接口访问资源的客户端来作出假设。

例如，考虑在以下场景中将会发生的事情。随着一个网站的用户量的增长，决定将旧的基于 XOS 平台的 Brand X 服务器替换为一个新的运行于 FreeBSD 之上的 Apache 服务器。磁盘存储硬件被替换掉了、操作系统被替换掉了、HTTP 服务器也被替换掉了、也许为所有内容生成响应的方法也被替换掉了。尽管如此，不需要改变的是 Web 的接口：如果设计正确，新服务器上的名字空间可以完全镜象原先老服务器的名字空间，这意味着从客户端（它仅仅知道资源，而不知道它们是如何实现的）的观点来看，除了改善了的站点的健壮性，什么变化也没有发生。

6.2.4 将语义绑定到 URI

正如上面所提到的，一个资源能够拥有多个标识符。换句话说，可以存在两个或更多个不同的 URI，当用来访问服务器时，具有相同的语义。也有可能有两个 URI，在访问服务器时导致使用相同的机制，然而两个 URI 标识的是两个不同的资源，因为它们并不意味着相同的事物。

对于设置资源标识符和用表述组装那些资源的动作而言，语义是一个副产品。服务器或客户端软件绝对不需要知道或理解 URI 的含义——它们仅仅扮演一个管道，通过这个管道，资源的创建者（一个作为命名权威的人）能够将表述与通过 URI 标识的语义关联起来。换句话说，在服务器端没有资源，仅仅是通过由资源定义的抽象接口提供答案的机制。这看起来似乎很奇怪，但是这正是使得 Web 跨越如此众多的不同实现的关键所在。

按照用来组成已完成产品的组件来定义事物，是每一个工程师的天性。Web 却并非是以这种方式运作的。基于在一个应用动作期间每个组件的角色，Web 架构由在组件之间的通信模型之上的约束组成。这防止了组件对于每件事物作出超越资源抽象的假设，因此隐藏了抽象接口任何一端的真实的机制。

6.2.5 REST 在 URI 中的不匹配

就像大多数真实世界中的系统一样，并非所有已部署的 Web 架构的组件都服从 Web 架构设计中给出的每一个约束。REST 既被用来作为一个定义架构改进的方法，也被用来识别架构的不匹配。当由于无知或者疏忽，一个软件实现以违反架构约束的方式来部署时，就会发生不匹配。尽管不匹配通常无法避免，但是有可能在它们定型之前识别出它们。

尽管 URI 的设计与 REST 的标识符的架构概念相匹配，单单依靠语法却不足以迫使命名权威按照资源模型来定义他们自己的 URI。一种形式的滥用是在由一个超媒体响应形式的表述（a hypermedia response representation）所引用的所有的 URI 中包括标识当前用户的信息。这样内嵌的用户 id 能够被用来维护服务器端会话的状态，通过记录用户的动作来跟踪他们的行为，或者跨多个动作携带用户的首选项（例如 Hyper-G 网关[84]）。尽管如此，由于违反了 REST 的约束，这些系统会导致共享缓存变得效率低下，这降低了服务器的可伸缩性，并且在一个用户与其他用户共享那些引用时会导致不希望的结果。

另一个与 REST 的资源接口的冲突发生在当软件试图将 Web 看作一个分布式文件系统的时候。因为文件系统暴露出了它们的信息的实现，有工具能够跨越多个站点对这些信息做镜象，来作为一种负载均衡和使内容以更接近用户的方式重新分布的方法。然而，他们能够这样做仅仅是因为文件拥有一组固定的语义（一个命名的字节序列），能够很容易地被复制。与之相反，试图将一个 Web 服务器的内容以文件的形式做镜象将会失败，因为资源接口并非总是匹配一个文件系统的语义，而且资源的表述中同时包括有数据和元数据（这对于一个表述的语义来说是非常重要的）。Web 服务器的内容能够在远程站点上被复制，但是应该仅仅复制完整的服务器机制和配置，或者有选择地仅仅复制那些其表述已知为静态的资源（例如，与 Web 站点相联系的缓存网络通过将特定的资源表述复制到整个 Internet 的边缘，以降低延迟和减轻来源的服务器的负担）。

6.3 将 REST 应用于 HTTP

超文本转移协议（HTTP）在 Web 架构中有一个特殊的角色，既作为在 Web 组件之间通信的主要的应用级协议，也作为特别为转移资源的表述而设计的唯一的协议。与 URI 不同，需要做大量的修改才能使 HTTP 能够支持现代的 Web 架构。HTTP 实现的开发者对于采纳提议的增强很保守，因此在对 HTTP 的扩展能够被部署之前，需要对扩展加以证实，并且要受到对于标准的评论的影响。REST 被用来识别出现有的 HTTP 实现的问题，指定一个能够与 HTTP/1.0 协议[19]互操作的协议子集，分析提议的 HTTP/1.1[42]的扩展，并且提供部署 HTTP/1.1 的动机和驱动因素。

由 REST 识别出的在 HTTP 中的关键问题领域包括：为新协议版本的部署制订计划、将对消息的解析与 HTTP 的语义以及底层的传输层（TCP）分离开、明确区分权威的和非权威的响应、对于缓存的细粒度的控制、以及协议的各种无法自描述的方面。REST 也被用来为基于 HTTP 的 Web 应用的性能建模，并且预测作为持久连接和内容协商的这些扩展对 Web 应用产生的影响。最后，REST 被用来对标准化的 HTTP 扩展的范围加以限制，仅限于那些适合于此架构模型的扩展，而不是允许误用 HTTP 的应用也同样地对标准产生影响。

6.3.1 可扩展性

REST 的一个主要的目标是在一个已部署的架构中支持逐渐的和片段的修改。我们对 HTTP 做了修改，通过引入版本控制的需求，并且扩展每个协议的语法元素的规则来支持这个目标。

6.3.1.1 协议版本控制

HTTP 是一个协议家族，通过主的和次的版本号来区分。该家族成员共享“HTTP”这个名称，这主要是因为当与一个基于“http”这个 URI 名字空间的服务直接通信时，它对应于期待的协议。一个连接器必须服从包括于每一个消息[90]中的 HTTP-version 协议元素之上的约束。

一个消息的 HTTP-version 代表了发送者对于协议的支持能力，以及正在发送的消息的总的兼容性（主版本号）。这允许客户端使用一个简化的（HTTP/1.0）功能子集发送一个正常的 HTTP/1.1 请求，同时向接收者表明它有能力支持完全的 HTTP/1.1 通信。换句话说，它提供了一个在 HTTP 协议范围内进行试探性的协议协商的形式。一个请求/响应链之上的每一个连接都能够在它所支持的最佳协议级别上执行操作，而不管作为这个链条一部分的某些客户端或服务器。

协议的意图是服务器应该总是使用它能够理解的协议的最高次版本，以及与客户端的请求消息相同的主版本来作出响应。其限制就是服务器不能使用那些高级别协议的可选功能，这些功能是禁止被发送给一个旧版本的客户端的。协议中不存在无法与相同主版本的所有其他次版本共同使用的必需功能（译者注：协议的功能分为必需功能和可选功能两类，必需功能是相同的主版本必须要支持的），否则将是一种对于协议所作的不兼容（incompatible）的改变，从而要求通信的双方不得不改变协议的主版本。能够依赖于次版本号改变的 HTTP 功能，仅限于那些被通信中的直接邻居解释的功能，因为 HTTP 并不要求整个请求/响应链中的所有中间组件都使用相同的版本。

这些规则的存在协助了多个协议修订版的部署，并且防止了 HTTP 架构师遗忘掉协议的部署是其设计的一个重要方面。这些规则是通过使得对于协议兼容的改变和不兼容的改变容易区别来做到这一点的。兼容的改变很容易部署，对于协议接受能力的差异能够在协议流（protocol stream）中进行沟通。不兼容的改变难以部署，因为它们在协议流能够开展通信之前，必须做一些工作来确定协议的接受能力。

6.3.1.2 可扩展的协议元素

HTTP 包括了很多单独的名字空间，每一个都有不同的约束，但是不能限制可扩展性是所有名字空间的共同需求。一些名字空间由单独的 Internet 标准来管理，并且由多个协议共享（例如，URI 模式（URI schemes）[21]、媒体类型（media types）[48]、MIME 头信息字段名（MIME header field names）[47]、字符集值（charset values）、语言标签（language tags））。而其他的名字空间由 HTTP 来管理，包括：方法名称、响应状态码、非 MIME 头信息字段名、以及在标准的 HTTP 头信息字段中的值。既然早期的 HTTP 没有为在这些名字空间中的改变如何来部署定义一组一致的规则，这就成为了规范工作需要面对的头等问题之一。

HTTP 请求的语义通过请求方法的名称来表示。任何时候当一组可标准化的语义能够被客户端、服务器和任何它们之间的中间组件共享时，则允许对方法进行扩展。不幸的是，早期的 HTTP 扩展，明确地说，即 HEAD 方法，使得对于一个 HTTP 响应消息的解析依赖于要知道请求方法的语义。这导致了部署上的困难：如果接收者需要在一个方法能够被一个中间组件安全地转发之前知道它的语义，那么所有的中间组件都必须在新的方法能够被部署之前进行更新（译者注：中间组件需要进行更新以理解新的方法，无法简单地将其转发）。

通过将解析和转发 HTTP 消息的规则与新的 HTTP 协议元素的相关语义分离开，这个部署问题得到了解决。例如，只有在 HEAD 方法中 Content-Length 头信息字段才能够表示消息体长度之外的含义，没有新的方法能够改变对消息长度的计算。GET 和 HEAD 也是仅有的两个方法，在其中有条件的请求头信息字段具有刷新缓存的语义，而对于所有其他的方法，

请求头信息字段的含义是需要满足的一个前提（a precondition）。

同样地，HTTP 需要一个通用的规则来解释新的响应状态码，这样新的响应能够进行部署而不会严重损害老的客户端。因此我们扩大了这个规则，规定每个状态码属于一个类别，通过三位十进制数的第一位数字来表示：100-199 表示消息中包含一个临时的信息响应，200-299 表示请求成功，300-399 表示请求需要被重定向到另一个资源，400－499 表示客户端发生了一个不应该重复的错误，500-599 表示服务器端遇到了一个错误，但是客户端稍后可以得到一个更好的响应（或者通过某个其他服务器）。如果接收者不理解一个消息中的状态码的特定语义，那么它们必须将该状态码按照与同一类别中状态码为 x00 时相同的方式来处理。就像是方法名称的规则一样，这个可扩展性的规则在当前的架构上添加了一个需求，这样架构就能够预见到未来的改变。改变因此能够被部署在一个现有架构之上，而无须害怕出现不利的组件反应（adverse component reactions）。

6.3.1.3 升级

在 HTTP/1.1 中新增的 Upgrade 头信息字段，通过允许客户端在一个旧的协议流中表达它希望使用一个更好的协议，从而减少了部署不兼容的改变的难度。Upgrade 就是特别设计来支持有选择地将 HTTP/1.x 替换为其他的，可能对一些任务更有效率的未来的协议。这样，HTTP 不仅仅支持内部的可扩展性，也可以在一个活跃的连接期间完全将其自身替换为其他的协议。如果服务器支持改进的协议，并且希望进行切换，它简单地响应一个 101 状态码并且继续，就好像请求是在那个升级的协议中接收到的一样。

6.3.2 自描述的消息

组件之间的 REST 约束的消息（REST constrains messages）是自描述的，以便支持中间组件对于交互的处理。然而，早期 HTTP 的一些方面并不是自描述的，包括缺乏在请求中对于主机的标识，无法依照语法来区分消息控制数据和表述元数据，无法区分仅仅想要立即与对方连接的控制数据和想要发送给所有接收者的元数据，缺乏对于必需扩展（mandatory extensions）和使用分层的编码来描述表述的元数据的支持。

6.3.2.1 主机

早期 HTTP 设计中一个最糟糕的错误是，决定不发送作为一个请求消息的目标的完整的 URI，而是仅仅发送那些没有用于建立连接的部分。其所做的假设是：一个服务器将基于连接的 IP 地址和 TCP 端口得知它自己的命名权威。然而，这个假设没有预测到多个命名权威可能存在于单个服务器上，随着 Web 和域名（http 的 URL 名字空间中命名权威的基础）以指数的速率增长，远远超出了新的 IP 地址的供应量，这成为了一个非常紧急的问题。

为 HTTP/1.0 和 HTTP/1.1 而定义和部署的解决方案是：在一个请求消息的 Host 头信息字段中包括目标 URL 的主机信息。部署这个功能被认为是如此重要，以至于 HTTP/1.1 规范要求服务器拒绝任何没有包括一个 Host 字段的 HTTP/1.1 请求。得到的结果就是：现在存在很多大型的 ISP 服务器，可以在单个 IP 地址上运行数以万计的基于名字的虚拟主机网站。

6.3.2.2 分层的编码

HTTP 为了描述表述的元数据，继承了多用途 Internet 邮件扩展（MIME）[47]的语法。MIME 没有定义分层的媒体类型，而是倾向于在 Content-Type 字段值中仅仅包括最外层媒体类型的标签。然而，这妨碍了一个接收者在不对该层次进行解码的情况下确定一个已编码消息的性质。一个早期的 HTTP 扩展通过在 Content-Encoding 字段中分别列出外层媒体类型的

编码，并且将最内层媒体类型的标签放在 Content-Type 中解决了这个问题。这是一个糟糕的设计决策，因为它改变了 Content-Type 的语义却没有改变它的字段名称，当旧的用户代理遇到这个扩展时会导致产生混淆。

一种更好的方案是继续将 Content-Type 看作是最外层的媒体类型，并且在那种类型中使用一个新的字段来描述内嵌的类型。不幸的是，在第一个扩展的缺点被识别出之前，它已经被部署了。

REST 确实识别出了另外一层编码的需求：将那些编码通过一个连接器添加到一个消息上，从而改善了消息在网络上的可转移性（transferability）。这个新的层叫做一个转移编码（transfer-encoding，引用在 MIME 中一个类似的概念），允许为转移而对消息进行编码，而不是意味着表述生来就是已编码的。转移编码能够被转移代理（transfer agents）因为某种原因添加或者移除，而不会改变表述的语义。

6.3.2.3 语义独立性

如上面所描述的，对于 HTTP 消息的解析与它的语义相分离。消息的解析（包括发现和将头信息字段融合在一起）与对头信息字段内容的解析过程完全分离。以这种方式，中间组件能够迅速处理和转发 HTTP 消息，并且扩展能够在不会破坏现有解析器的情况下进行部署。

6.3.2.4 传输独立性

早期的 HTTP，包括大多数的 HTTP/1.0 实现，使用了底层的传输协议作为表示响应消息结束的方法。一个服务器通过关闭 TCP 连接来表明响应消息的结束。不幸的是，这导致了在协议中出现了一个严重的故障状况：一个客户端没有办法区分一个完成的响应和一个因为某种网络故障而被截断的响应。为了解决这个问题，Content-Length 头信息字段在 HTTP/1.0 中被重新定义了，以表示消息体的字节长度（只要能够预先知道它的长度），并且还将“chunked”（分块）转移编码引入到了 HTTP/1.1 中。

chunked 编码允许一个表述在它的生成阶段（当头信息字段被发送时）尺寸是未知的，而它的边界通过一系列能够在被发送前单独设置尺寸的分块来描述。它也允许元数据在消息的末尾发送，支持在最初消息被生成的时候创建可选的元数据，而不会增加响应的延迟（译者注：因为这些可选的元数据是在消息的末尾）。

6.3.2.5 尺寸限制

一个对于应用层协议的灵活性来说时常出现的障碍，是在协议的参数上过度指定尺寸限制。尽管在协议的实现中总是存在着一些实际的限制（例如，可用的金钱），在协议中指定这些限制，却会将所有的应用限制在相同的上限内，而不管它们的具体实现有何需求。结果常常是一个最小公分母式的协议，无法超越其最初创建者的设想进行很大的扩展。

在 HTTP 协议中并没有限制 URI 的长度、头信息字段的长度、表述的长度、或者任何由一列栏目组成的字段值的长度。尽管老的 Web 客户端对于超过 255 个字符组成的 URI 的处理存在着众所周知的问题，注意到在 HTTP 规范中存在这个问题就足够了，而不是要求所有的服务器都受到这样的限制。没有指定一个协议的最大值的原因是在一个可控制的环境中（例如，在一个 Intranet 中）的应用能够通过替换旧的的组件来避免那些限制。

尽管我们并不需要发明人造的限制，HTTP/1.1 确实需要定义一组合适的响应状态码，指示出何时一个特定的协议元素对于一个服务器的处理来说太长了。需要添加以下这些响应状态码：请求的 URI 太长、头信息字段太长、消息体太长。不幸的是，客户端没有办法向服务器表明它可能会存在资源限制。当资源有限的设备（例如 PDA）在试图不通过一个与特定设备相关的中间组件对通信进行调整的情况下使用 HTTP 时，这会导致出现问题。

6.3.2.6 缓存控制

因为 REST 努力在高效率的、低延迟的行为和想要得到的语义上透明的缓存行为之间取得平衡，它允许应用来确定缓存的需求，而不是将该需求硬编码在协议本身之中，这对于 HTTP 来说是至关重要的。对于协议来说，最重要的事情是完全地和精确地描述正在被转移的数据，使得没有应用会受到愚弄，误以为存在某个事物，实际上存在另一个事物。HTTP/1.1 通过添加 Cache-Control、Age、Etag 和 Vary 几个头信息字段来达到这个目标。

6.3.2.7 内容协商

所有资源都会将一个请求（由方法、标识符、请求头信息和有时存在的一个表述组成）映射到一个响应（由一个状态码、响应头信息字段和有时存在的一个表述组成）。当一个 HTTP 请求映射到在服务器端的多个表述时，服务器可以参与到与客户端的内容协商中，以便确定哪一个表述最适合客户端的需求。这实际上是一个“内容选择”的过程，而不是协商。

尽管有一些已部署的内容协商的实现，但是它们并没有被包括在 HTTP/1.0 规范之中，因为在该规范发布之时，并不存在可互操作（interoperable）的实现子集。这部分是由于在 NCSA Mosaic 中的糟糕实现，它在每个请求的头信息字段中发送 1KB 的首选项信息，而不管资源是否是可协商的[125]。因为在全部 URI 中只有远少于 0.01%的 URI 对于内容是可协商的，结果是增加了请求的延迟，而几乎什么也没有得到，这导致了后来的浏览器完全忽视了 HTTP/1.0 在内容协商方面的功能。

抢先式（服务器驱动（server-driven）的）协商发生在以下情况：当服务器根据请求头信息字段的值或正常的请求参数（方法/标识符/状态码）之外的某事物，为某种特殊的请求方法/标识符/状态码的组合改变响应的表述。当这种情况发生时，客户端需要得到通知，这样当语义透明时一个缓存就能够知道，要为将来的请求使用一个特殊的已缓存响应，而一个用户代理一旦知道协商的信息对于接收到的响应的影响，就能够提供比正常情况下发送的更加详细的首选项。HTTP/1.1 为这个目的引入了 *Vary* 头信息字段。Vary 简单地列出了请求头信息字段的维度（译者注：即可以进行协商的头信息字段的列表），这样响应就能够根据这些信息发生改变。

在抢先式协商中，用户代理告诉服务器它有能力接收什么，然后假定服务器选择了最适合于用户代理所声称的能力的表述。然而，这是一个不容易处理的问题，因为它不仅需要知道用户代理能够接收什么，也需要知道它对于所能接收的每个功能的支持程度如何，以及用户想要使用表述的目的是什么。例如，一个在屏幕上查看一张图片的用户也许想要一个简单位图形式的表述，但是使用相同浏览器的相同的用户实际上可能更想要的是一个 PostScript 形式的表述（如果他是想要将图片发送到一个打印机）。这也要依赖于用户根据他们自己个人偏爱的内容正确地配置了他们的浏览器。简而言之，一个服务器几乎无法有效地利用抢先式协商，但是这是早期 HTTP 所定义的自动化内容选择的唯一形式。

HTTP/1.1 添加了反作用式（用户代理驱动（agent-driven）的）协商的概念。在这种情况下，当一个用户代理请求一个被协商的资源时，服务器响应一组可用表述的列表。用户代理随后能够根据它自己的能力和目的，选择一个最佳的表述。关于可用表述的信息可以用以下方式来提供：通过一个单独的表述（例如一个 300 响应）、在响应的数据（例如，有条件的 HTML）中、或者作为一个“最有可能”的响应的附录。最后的方式对于 Web 来说是最佳的方式，因为一个额外的交互仅仅在用户代理确定其他选项中的一个会更好时，才会成为是必需的。反作用协商只是正常的浏览器模型的一个自动化的反映（automated reflection），这意味着它能够充分利用 REST 的所有性能上的优势。

抢先式协商和反作用式协商都需要解决如何对表述维度（译者注：即可用表述的列表）的实际性质（即，如何表明浏览器支持 HTML 表格但不支持 INSERT 元素）进行沟通的难题。

不过，反作用式协商有着明显的优点：不必在每个请求中都发送首选项、具有更多的上下文信息、当面对替换选项时可以使用这些信息作出选择，而不会影响到缓存。

第三种形式的协商，透明式协商[64]，是特许一个中间缓存扮演一个用户代理（agent），来代表其他的用户代理，由这个中间缓存来选择一个更好的表述，并且发起请求来获取那个表述。这个请求有可能该缓存内部通过另一个缓存命中来满足，因此有可能不必发送额外的网络请求。然而，通过这样做，它们执行了服务器驱动的协商，因此必须添加合适的 Vary 信息，这样其他的外部缓存（outbound cache）才不会产生混淆。

6.3.3 性能

HTTP/1.1 聚焦于在组件之间改善通信的语义，但是对于用户可觉察的性能也有一些改善，虽然它受到了与 HTTP/1.0 的语法相兼容的需求的限制。

6.3.3.1 持久连接

尽管早期 HTTP 的每个连接单个请求/响应的行为实现起来很简单，但是它导致了对于底层 TCP 传输机制低效率的使用，因为每次交互都要重新建立连接的开销，以及 TCP 的拥塞控制慢启动（slow-start congestion control）算法[63, 125]的特性。作为其结果，出现了一些提议的扩展，在单个连接中组合多个请求和响应。

第一个提议是定义一组新的方法，在单个消息中封装多个请求（MGET、MHEAD 等等），并且将响应作为一个多个部分（multipart）的 MIME 类型返回。这个建议遭到了拒绝了，因为它违反了 REST 的几个约束。首先，客户端在第一个请求能够被发送到网络之前，需要知道它想要打包的所有的请求，因为一个请求体必需根据设置在最初请求的头信息字段中的一个 content-length 字段集合的信息来进行长度分隔（length-delimited）。其次，中间组件将不得不提取出每一个消息，以确定它能够在本地满足哪一个请求的需要。最后，这样做大大增加了请求方法的数量，并且使得有选择地拒绝特定方法的机制变得复杂化。

相反，我们采用了一种形式的持久连接，它使用了长度分隔的消息，以便在单个连接[100]中发送多个 HTTP 消息。对于 HTTP/1.0，通过使用在 Connection 头信息字段中的“keep-alive”指令来达到这个目标。不幸的是，这样做通常无法工作，因为这个头信息字段可以被中间组件转发到其他的不理解 keep-alive 的中间组件，从而导致一种死锁的状况。HTTP/1.1 最终决定将持久连接作为默认的选项，这样通过 HTTP-version 的值作为持久连接存在的信号就足够了，并且仅仅需要使用连接指令“close”来改变这个默认值。

仅仅在 HTTP 消息被重新定义为自描述的和独立于底层的传输协议之后，才有可能实现持久连接，注意到这一点是很重要的。

6.3.3.2 直写式（write-through）缓存

HTTP 不支持回写式（write-back）缓存。一个 HTTP 缓存不能假设通过它写入的内容与来自相同资源的一个后续请求的可获取的内容是相同的，因此它不能缓存一个 PUT 请求的消息体，并且将其内容重用于稍后的 GET 响应。定义这个规则有两个理由：1) 元数据可能会以后台的方式（behind-the-scene）来生成，并且 2) 对于以后的 GET 请求的访问控制无法根据 PUT 请求来确定。然而，因为使用 Web 的写入动作是极其罕见的，缺乏回写式缓存并不会对性能产生严重影响。

6.3.4 REST 在 HTTP 中的不匹配

在 HTTP 中有一些架构上的不匹配，一些是由于在标准过程之外部署的第三方扩展所导

致的，其他的则是由于与已部署的 HTTP/1.0 组件保持兼容的必要性所导致的。

6.3.4.1 区分非权威的响应

HTTP 中仍然存在的一个弱点是，没有一致的机制来区分权威的响应（由来源服务器为响应当前请求而生成）和非权威的响应（从一个中间组件或缓存中获得，而没有访问来源服务器）。这种区分对于请求权威响应的应用来说是很重要的，例如在卫生保健行业中使用的安全性至关重要的信息用具。当返回一个错误的响应时，客户端会纳闷错误是由于来源服务器造成的，还是由于某个中间组件造成的。使用额外的状态码来尝试解决这个问题并不成功，因为是否权威这个特性通常与响应的状态是正交的。

HTTP/1.1 确实添加了一种机制来控制缓存的行为，这样可以表明想要得到一个权威的响应。一个请求消息上的“no-cache”指令要求任何缓存将请求转发到来源服务器，甚至是在它拥有一个被请求内容的已缓存版本的时候。这允许一个客户端刷新一个已知被破坏了或者过期了的缓存副本。然而，频繁使用这个字段会影响从缓存中获得的性能好处。一个更加通用的解决方案是，要求无论何时当一个动作没有导致访问来源服务器时，都要将响应标记为非权威的。出于这个目的（以及其他的目的），在 HTTP/1.1 中定义了一个 Warning 响应头信息字段，但是在实践中并未被广泛实现。

6.3.4.2 Cookie

一个对于协议作出不适当的扩展，以支持与想要得到的通用接口相矛盾的功能的例子，就是以 HTTP Cookie[73]的形式引入了站点范围的状态信息。Cookie 交互与 REST 关于应用状态的模型不匹配，因此常常会导致典型浏览器应用的混淆。

一个 HTTP Cookie 是不透明的数据，来源服务器通过将它包括在一个 Set-Cookie 响应头信息字段中，将它设置给一个用户代理，用户代理应该在所有将来的请求中包括这个相同的 Cookie，直到它被替换或者过期。这样的 Cookie 通常包含一个特定于用户的配置选项，或者包含一个在将来的请求中与服务器的数据库进行匹配的标记（token）。问题在于一个 Cookie 被定义为被附加在任何将来的请求上，对于特定的一组资源标识符，Cookie 通常是与一个完整的站点相关联，而不是与浏览器中特定的应用状态（当前呈现的表述的那一组状态）相关联。当随后使用浏览器的历史功能（“Back”按钮）回退到 Cookie 所反映的视图之前的一个视图时，浏览器的应用状态不再匹配 Cookie 所代表的已保存状态。因此，发送到相同服务器的下一个请求包含的是一个并没有代表当前应用上下文的 Cookie，这会使得通信的两端都产生混淆。

Cookie 也违反了 REST，因为它们允许数据在没有充分表明其语义的情况下进行传递，这样就成为了一个安全和隐私方面的关注点。结合使用 Cookie 和 Referer[sic]头信息字段，有可能当用户在多个站点之间浏览时，对他进行跟踪。

其结果是，Web 上基于 Cookie 的应用永远都不值得信任。相同的功能应该通过匿名的认证和真正的客户端状态来完成。一种涉及到首选项的状态机制，能够通过明智地使用上下文设置（context-setting）URI，而不是使用 Cookie，来更加有效地实现。在这里“明智”一词意味着每种状态一个 URI，而不是由于内嵌了一个 user-id 而导致无限数量的 URI。同样地，对于使用 Cookie 在一个服务器端的数据库中标识一个特定于用户的“购物篮”而言，更加有效的方法是通过在超媒体数据格式中定义购物栏目的语义来实现，这允许用户代理在它们自己的客户端购物篮中选择和保存那些栏目，当客户端准备好购买时，使用一个 URI 来完成结账。

6.3.4.3 必需扩展（Mandatory Extensions）

HTTP 头信息字段名称仅当它们所包含的信息对于正确理解消息并非是必需的时候，才能够被任意扩展。对必需的头信息字段进行扩展，需要对协议作出一次大的修订，或者对于方法语义作出一个实质性的改变，如同在[94]中所提议的那样。这是在现代 Web 架构中一个尚未匹配 REST 架构风格的自描述消息约束的方面，主要是因为在现有 HTTP 语法中实现一个支持必需扩展的框架的成本，超过了我们可能从必需扩展中获得的任何明显的好处。然而，当现有的语法向后兼容的约束不再适用后，期待在下一次对于 HTTP 所作的大的修订中支持对于必需字段名称的扩展是合情合理的。

6.3.4.4 混合元数据（Mixing Metadata）

HTTP 被设计用来跨越一个网络连接扩展通用的连接器接口。因此，它有意匹配这个接口的特性，包括将参数描述为控制数据、元数据、以及表述。然而，HTTP/1.x 协议家族的两个最严重的局限是：它没有从语义上区分表述的元数据和消息的控制信息（都是作为头信息字段来传输）；而且不允许为了对消息进行完整性检查，而对元数据进行有效地分层。

REST 将这些问题识别为在早期标准化过程中制定的协议中的局限，预见到它们将会在其他功能的部署中导致出现问题，例如持久连接和摘要认证。为此开发出了一些变通方案（workarounds），包括添加 Connection 头信息以标识出不能够安全地被中间组件转发的每一个连接的控制数据，同时也包括了对于头信息字段摘要的一个规范的处理算法。

6.3.4.5 MIME 语法

HTTP 从 MIME[47]中继承了消息的语法，以便保持与其他 Internet 协议的共同点，并且重用很多已经标准化了的字段来描述消息中的媒体类型。不幸的是，MIME 和 HTTP 具有非常不同的目标，这些语法仅仅是为 MIME 的目标而设计的。

在 MIME 中，一个用户代理将一捆信息发送给一个永远也不直接进行交互的未知的接收者，希望这捆信息被看作是一个连贯的整体。MIME 假设用户代理想要在一个消息中发送所有的信息，因为跨 Internet 邮件发送多个消息是低效的。这样，MIME 的语法被构造为将消息打包在一个部分或者多个部分（multipart）之中，与邮局使用额外的纸张来包装多个包裹的方式差不多。

除非是为了安全封装或打包归档的目的，在 HTTP 中，将不同的对象打包在单个消息中是没有意义的，因为为那些尚未缓存的文档发送单独的请求会更加有效率。HTTP 应用使用像 HTML 这样的媒体类型作为“包”的引用的容器（containers for references to the “package”）——一个用户代理随后能够选择作为单独的请求来获取包的哪些部分。尽管以下情况是有可能的，HTTP 能够使用一个包含多个部分（multipart）的包，在其中在第一部分之后仅包括有非 URI 的资源（non-URI resources，译者注：即没有相对应的 URI 的资源），但是对于这种情况的需求并不是很多。

MIME 语法的问题在于它假设传输机制是有损耗的，会故意将换行和内容长度等信息破坏掉。因此其语法有很多的冗余，并且对于任何并非基于有损耗传输机制的系统来说都是低效的，这使得它对于 HTTP 是不适合的。既然 HTTP/1.1 有能力支持不兼容协议的部署，保留 MIME 的语法对于 HTTP 的下一个主要的版本而言并不是必需的，尽管如此，还是有可能为表述的元数据继续使用很多标准化的协议元素。

6.3.5 将响应匹配到请求

当需要描述哪一个响应属于哪一个请求的时候，HTTP 消息并不是自描述的。早期的

HTTP 基于每个连接单个请求和响应，因此没有觉察到需要有将响应与相关的请求绑定在一起的消息控制数据。因此，请求的顺序决定了响应的顺序，这意味着 HTTP 依赖于传输机制的连接（transport connection）来决定这个匹配。

尽管 HTTP/1.1 被定义为独立于传输协议，但是仍然假设通信是发生在一个同步的传输机制之上。很容易通过添加一个请求标识符来扩展 HTTP，使得它能够在一个异步的传输机制（例如 e-mail）之上工作。这样的扩展在广播（broadcast）或者多播（multicast）的情况下是很有用的，在那里响应可能是在与请求不同的一个频道中接收到的。同样地，在有很多请求悬而未决的情况下，这样的扩展允许服务器选择响应的转移顺序，这样更小的或者更重要的响应就能够被首先发送。

6.4 技术迁移

尽管 REST 对于创作 Web 标准有着最直接的影响，以商业级（commercial-grade）实现的形式来部署这些标准的过程，也对作为一种架构设计模型的 REST 的使用进行了验证。

6.4.1 libwww-perl 的部署经验

我参与 Web 标准的定义开始于开发维护机器人 MOMspider[39]及其相关的协议库 libwww-perl。模仿最初由 Tim Berners-Lee 开发的 libwww 和 CERN 的 WWW 项目，libwww-perl 提供了一个统一的接口，来为使用 Perl 语言[134]编写的客户端应用发送 Web 请求和解释 Web 响应。它是第一个独立于最初的 CERN 项目开发的 Web 协议库，反映出对于 Web 接口的一个比旧的代码基础（译者注：即更老的 libwww 库）更加现代的解释。libwww-perl 所提供的接口成为了设计 REST 的基础。

libwww-perl 由单个请求接口组成，它使用了 Perl 的自求值（self-evaluating）代码功能，基于请求 URI 的模式（scheme）来动态加载合适的传输协议包。例如，当请求发送一个对于 URL<http://www.ebuilt.com/>的“GET”请求时，libwww-perl 将从 URL（“http”）中提取出模式，并且使用它来构造一个对 wwwhttp 的 resuest()方法的调用。这里使用的是一个对于所有的资源类型（HTTP、FTP、WAIS、本地文件等等）都是通用的接口。为了获得这种通用的接口，这个库看待所有调用的方式与一个 HTTP 代理大致相同。它提供了一个使用 Perl 数据结构的接口，该接口与一个 HTTP 请求具有相同的语义，而不管资源的类型是什么。

libwww-perl 展示了通用接口的好处。在它的最初版本发布后的一年之中，超过 600 个独立软件开发者在他们自己的客户端工具中使用了这个库，从命令行下载脚本到全功能的浏览器。这个库是当前大多数 Web 系统管理工具的基础。

6.4.2 Apache 的部署经验

随着 HTTP 规范的开发工作开始接近尾声，我们需要有服务器软件，既能够有效地展示提议的标准协议，也能够作为有价值的扩展的一个测试台。那个时候，最流行的 HTTP 服务器（httpd）是公共域（public domain）软件，由 Rob McCool 在伊利诺斯大学香槟分校的国家超级计算应用中心（NCSA）开发的。不过，在 Rob 离开了 NCSA 后，开发仍然持续到了 1994 年，很多站长开发了他们自己的扩展以及公共分发（common distribution）的需要除去了 bug 的版本。出于协调我们所做的改变（作为对最初源代码的“补丁”）的目的，我们中的一组人创建了一个邮件列表。在这个过程中，我们创建了 Apache HTTP 服务器项目[89]。

Apache 项目是一个协作软件开发的尝试，目标是创建一个健壮的、商业级的、功能完善的、开源的 HTTP 服务器的软件实现。这个项目由一批分布在世界各地的志愿者来共同管理，使用 Internet 和 Web 来进行沟通、制订计划、开发服务器及其相关的文档。这些志愿者

被称作 Apache 开发组。后来，这个开发组形成了非盈利性质的 Apache 软件基金会，作为一个法定的组织提供财务上的支持，以支持 Apache 开源项目的继续开发。

Apache 变得知名是因为它能够以很健壮的方式支持对于一个 Internet 服务（an Internet service）的各种需求，以及它对于 HTTP 协议标准的严格实现。我在 Apache 开发组中作为“协议警官”，为核心的 HTTP 解析功能编写代码，通过解释标准支持其他的任务，并且在标准论坛中扮演一个倡导者，帮助 Apache 开发者理解什么是“实现 HTTP 的正确方式”。本章中所描述的很多经验和教训，是从在 Apache 项目中创建和测试不同的 HTTP 实现的过程中获得的，而且协议背后的理论受到了 Apache 开发组的苛刻评论的影响。

Apache httpd 被广泛地看作最为成功的软件项目之一，并且是最早的占据市场统治地位的开源软件产品之一，与此同时市场中还存在有重要的商业竞争者。2000 年 7 月 Netcraft 调查了公共的 Internet 网站，发现超过两千万的站点基于 Apache 软件，代表了被调查的全部站点 65%以上的比例[http://www.netcraft.com/survey/]。Apache 是第一个支持 HTTP/1.1 协议的主流服务器，并且通常被看作是该协议的参考实现，所有的客户端软件均基于它来进行测试。Apache 开发组荣获了 1999 年的 ACM 软件系统大奖，作为对于我们在 Web 架构的标准方面影响力的认可。

6.4.3 开发顺从于 URI 和 HTTP/1.1 的软件

除了 Apache，还有很多其他的项目，包括商业软件和开源软件，都采用和部署了基于现代 Web 架构的协议的软件产品。尽管可能是个巧合，自从微软的 Internet Explorer 成为第一个实现了 HTTP/1.1 客户端标准的主要的浏览器之后，Internet Explorer 在 Web 浏览器市场的份额超过了 Netscape 的 Navigator。除此之外，在标准化的过程中还定义了很多个别的 HTTP 扩展，例如 Host 头信息字段，现在已经被广泛地部署。

REST 架构风格成功地指导了现代 Web 架构的设计和部署。到目前为止，新引入的标准并不存在严重的问题，甚至是在它们不得不在遗留的 Web 应用旁边逐渐地和片段地部署的情况下，也没有出现什么问题。此外，新的标准对于 Web 的健壮性产生了积极的影响，并且支持通过缓存的层次结构（caching hierarchies）和内容分布网络（content distribution networks）改善用户可觉察的性能的新方法。

6.5 架构上的教训

从现代 Web 架构和由 REST 识别出的问题中可以总结出很多通用的架构上的教训。

6.5.1 基于网络的 API 的优势

将现代 Web 与其他中间件[22]区分开的是它使用 HTTP 作为一个基于网络的应用编程接口（API）。其实并非是一向如此，早期的 Web 设计利用了一个程序库（CERN 的 libwww）作为所有的客户端和服务器所使用的单个协议实现库。CERN libwww 提供了一个基于库的（library-based）API 来建造可互操作的 Web 组件。

一个基于库的 API 提供了一组代码入口点和相关联的符号/参数集，假如一个程序员遵循来自那些代码的架构和语言上的约束，这个程序员就能够使用其他人的代码来完成维护相似系统之间的真实接口的繁重工作。其假设就是通信的所有参与方都要使用相同的 API，因此接口的内部实现仅仅对于 API 的开发者来说是重要的，对于应用的开发者来说是不重要的。

这种使用单个库的方法终止于 1993 年，因为它无法与参与 Web 开发的组织的社会动力学相匹配。NCSA 的开发团队比在 CERN 曾经出现过的团队还要大得多，NCSA 通过该团队加快了 Web 开发的步伐，这时 libwww 的源代码出现了“分叉”（分裂为分别维护的代码基

础），因为在 NCSA 的人无法等待 CERN 跟上他们的改进要求。与此同时，独立开发者，例如我自己，开始为 CERN 代码尚未支持的语言和平台开发协议库。Web 的设计必须从开发一个参考的协议库转移到开发一个基于网络的 API，以跨平台和实现的方式扩展期待的 Web 语义。

一个基于网络的 API 对于应用的交互而言，是一种包含有已定义语义的在线（on-the-wire）的语法。一个基于网络的 API 没有在应用的代码上强加任何除了读/写网络的需求之外的限制，但是确实在能够有效地跨接口进行通信的一组语义上添加了限制。其有利的方面就是，性能仅仅受限于协议的设计，而不是受限于该设计的特殊实现。

一个基于库的 API 为程序员做的工作要多得多，但是通过做这些工作，也带来了大量更多的复杂性，并且其负担超出了任何单个系统必须承受的限度，这种方法在一个不同种类的网络中的可移植性较差，而且总是会导致首先选择通用性，而不是性能。作为一个副作用，它也导致了在开发过程中产生惰性（为任何事情都去责备 API 代码），而不去努力解决其他通信参与方不合作的行为。

然而，很重要的是要记住在任何架构中都包括有不同的层，包括现代 Web 的架构。像 Web 这样的系统使用一个基于库的 API（socket）来访问多个基于网络的 API（即 HTTP 和 FTP），但是 socket API 本身是低于应用层的。同样地，libwww 是一个有趣的混合物，它为了访问一个基于网络的 API 而发展成为了一个基于库的 API，从而提供了可重用的代码，但是却没有假设其他的通信应用也正在使用 libwww。

这与类似 CORBA[97]那样的中间件形成了对比。因为 CORBA 仅仅允许通过一个 ORB 来进行通信，它的转移协议 IIOP 对于通信的参与方使用什么内容来进行通信做了太多的假设。HTTP 请求消息包括了标准化的应用语义（application semantics），而 IIOP 消息却没有包括。“Request”符号在 IIOP 中仅仅提供了方向性，这样 ORB 能够根据自身是否应该作出回应（即“LocateRequest”）或者是否应该通过一个对象来解释该请求，来对请求进行路由。具体的语义通过一个对象的 key 和操作的组合来表达，它是特定于对象的，而不是跨所有对象标准化的。

一个独立开发者能够生成与一个 IIOP 请求相同的比特，而不使用相同的 ORB，但是比特本身是由 CORBA 的 API 和它的接口定义语言（IDL）定义的。生成这些比特需要一个由 IDL 编译器生成的 UUID、一段对 IDL 操作签名做镜象的结构化二进制内容、遵循 IDL 规范的回应数据类型（definition of the reply data type(s)）的定义。其语义因此由网络接口（IIOP）来定义，而不是由对象的 IDL 规范（IDL spec）来定义。这是否是一件好事，取决于应用的性质——对于分布式对象这是必需的，对于 Web 这并不是什么好事。

为何这些差别是很重要的？因为它将一个网络中间组件能够成为有效的代理（effective agent）的系统，和一个网络中间组件最多只能成为路由器的系统区分了开来。

这种形式的区分也可以在将消息解释为一个单元（a unit）或者解释为一个流（a stream）中看到。HTTP 允许接收者或发送者来自行决定。CORBA 的 IDL 甚至（仍然）不允许使用流，即使当它确实得到了扩展以支持流之后，通信的双方仍然被绑定在相同的 API 上（译者注：即 CORBA 的基于库的 API），而不是能够自由地使用最适合于它们的应用类型的东西。

6.5.2 HTTP 并不是 RPC

人们常常错误地将 HTTP 称作一种远程过程调用（RPC）[23]机制，仅仅是因为它包括了请求和响应。调用远程机器上的一个过程（procedure）的观念，是 RPC 与其他形式的基于网络的应用通信的区别所在。RPC 的协议识别出过程并且传递给它固定的一组参数，然后等待在使用相同接口返回的一个消息中提供的回答。远程方法调用（RMI）也是类似的，除

了过程被标识为一个｛对象，方法｝的组合，而不是一个简单的服务过程（service procedure）。被代理的 RMI 添加了名称服务的间接层和少量其他的技巧（trick），但是接口基本上是相同的。

将 HTTP 和 RPC 区分开的并不是语法，甚至也不是使用一个流作为参数所获得的不同的特性，尽管它帮助解释了为何现有的 RPC 机制对于 Web 来说是不可用的。使得 HTTP 与 RPC 存在重大不同的是：请求是使用具有标准语义的通用的接口定向到资源的，这些语义能够被中间组件和提供服务的来源机器进行解释。结果是使得一个应用支持分层的转换（layers of transformation）和间接层（indirection），并且独立于消息的来源，这对于一个 Internet 规模、多个组织、无法控制的可伸缩性的信息系统来说，是非常有用的。与之相比较，RPC 的机制是根据语言的 API（language API）来定义的，而不是根据基于网络的应用来定义的。

6.5.3 HTTP 并不是一种传输协议

HTTP 并不是被设计为一种传输协议（transport protocol），它是一种转移协议（transfer protocol）（译者注：非常不幸，HTTP 刚刚传入我国时，即被翻译为“超文本传输协议”，因为“transport”和“transfer”在中文中都具有“传输”的含意，之后以讹传讹贻害无穷。为了以示区别，译文中一律将“transfer”翻译为“转移”）。在 HTTP 协议中，消息通过在那些资源的表述上的转移和操作，来对资源执一些动作，从而反映出 Web 架构的语义。使用这个非常简单的接口来获得广泛的功能是完全有可能的，但是必须要遵循这个接口，以便 HTTP 的语义被保持为对于中间组件是可见的。

这就是为何 HTTP 可以穿越防火墙的原因。大多数当前提议的对于 HTTP 的扩展，除了 WebDAV[60]以外，仅仅使用 HTTP 作为一种使其他的应用协议穿越防火墙的方法，这从根本上来说是一种有误导性的想法。不仅仅是因为这种扩展方式挫败了拥有一个防火墙的目的，而且从长远来看它将无法工作，因为防火墙的厂商将会不得不执行额外的协议过滤。因此这种扩展方式对于那些在 HTTP 之上的扩展而言是没有意义的，因为在这种情况下 HTTP 所完成的唯一的事情就是添加了来自一个遗留语法的负载（译者注：即添加了额外的 HTTP 协议负载）。一个真正的 HTTP 应用应该将协议用户的动作映射到能够使用 HTTP 语义来表达的某个事物，以这种方式创建一个基于网络的 API 来提供服务，能够被用户代理和中间组件所理解，而不需要知道关于应用的任何知识。

6.5.4 媒体类型的设计

REST 的一个对于一种架构风格来说不同寻常的方面，就是它对于 Web 架构中的数据元素的定义的影响程度。

6.5.4.1 一个基于网络的系统中的应用状态

REST 定义了一个期待的应用行为的模型，它支持简单的和健壮的应用，能够在很大程度上免疫于困扰大多数基于网络的应用的局部故障（partial failure）状况。然而，这并没有阻止应用的开发者引入违反这个模型的功能。最频繁出现的是违反有关应用状态和无状态交互方面的约束。

由于将应用状态放错了地方而造成的架构不匹配并不仅限于 HTTP Cookie。超文本标记语言（HTML）中引入的“frame”（帧）引起了类似的混淆。frame 允许一个浏览器窗口被分割为子窗口，每个子窗口都有自己的导航状态。在一个子窗口中的链接选项对于正常的迁移来说是无法辨别的，但是响应结果的表述是在子窗口中呈现的，而不是在完整的浏览器应用

工作区（full browser application workspace）中呈现。假设不存在到想要作为子窗口的信息领域的链接，这样做没有什么问题。但是当确实存在这种情况时，用户会发现他们自己正在查看的应用嵌入到了另一个应用的子上下文（subcontext）中（译者注：普通的链接无法表达其响应结果的表述是应该显示在完整的浏览器应用工作区中，还是应该显示在一个子窗口中，因此呈现的结果往往并非是用户的原意）。

对于 frame 和 cookie，其失败之处就在于，提供的间接的应用状态无法被用户代理来管理或解释。另一种设计是将这些信息放在一个主要的表述中，以通知用户代理如何为一个特定的资源领域管理超媒体工作区。这种设计能够完成相同的任务而不会违反 REST 的约束，同时还带来了更好的用户接口和对于缓存更少的干扰。

6.5.4.2 增量处理

通过将减少延迟作为一个架构目标，REST 能够按照用户可觉察的性能来对媒体类型（表述的数据格式）加以区分。尺寸、结构、增量呈现（incremental rendering）的能力都会影响在转移、呈现、操作表述的媒体类型时的延迟，并且因此能够严重影响系统的性能。

HTML[18]是一种媒体类型的例子，大部分情况下，它具有很好的延迟特性。在早期 HTML 中的信息能够在正在被接收的同时进行呈现，因为在早期所有的呈现信息都是可用的——在构成 HTML 的少量的标记标签的标准化的定义中。然而，HTML 在有些方面对于延迟而言设计得并不好。这些例子包括：放在一个文档的 HEAD 标签中的内嵌的元数据，导致了在呈现引擎能够阅读那些显示一些有用的东西给用户[93]的部分之前，必需转移并处理可选的信息；内嵌的图片如果没有呈现尺寸的提示，就会要求在包围它的其余的 HTML 能够被显示之前，要能够接收到图片最初的一些字节（包含了布局尺寸的信息）；动态设置尺寸的表格栏目，要求呈现引擎在它能够开始显示表格顶部之前阅读整个表格并确定尺寸；关于错误格式化的标记语法的懒惰规则（lazy rules），常常要求呈现引擎在它能够确定缺少了一个关键的标记字符（译者注：即“<”和“>”字符）之前，解析整个文件。

6.5.4.3 Java vs. JavaScript

REST 也能够被用来获得以下的洞察力：为何一些媒体类型与其他类型相比，在 Web 架构中得到了更加广泛的采用，甚至是在某种媒体类型并非是开发者的最爱的情况下。Java applet 对抗 JavaScript 就是一个例子。

Java[45]是一种流行的编程语言，源自一个电视机顶盒应用的开发，但是使它初次名声大噪的是当它被引入到 Web，用来作为实现按需代码（code-on-demand）功能的一种方法。尽管 Java 语言从其拥有者 Sun 微系统公司那里得到了巨大的支持，而且当软件开发者寻求一种 C++语言的替代者时，受到了广泛的关注。它却并没有被 Web 中按需代码的应用开发者所广泛采用。

稍后于 Java 的引入，Netscape 通信公司的开发者创建了一种独立的语言，用来支持内嵌的按需代码，最初叫做 LiveScript，但是后来因为市场原因（这两种语言除了这一点外，几乎没有什么共同之处）改名为 JavaScript[44]。尽管最初因嵌入在 HTML 中和与正常的 HTML 语法不兼容而遭到了嘲笑，JavaScript 自从它引入之后却出现了稳定的增长。

问题就是：为何 JavaScript 在 Web 上比 Java 更加成功？这当然不是因为它作为一种语言的技术品质，因为它的语法和执行环境与 Java 相比都被认为是很糟糕的。也不是因为市场营销方面的原因：Sun 公司在营销方面花的钱要多得多，并且还在继续这样做。同样也不是因为任何语言本质特性方面的原因，因为 Java 在所有其他的编程领域（独立运行的应用，servlet 等等）都要比 JavaScript 成功得多。为了更好地理解这种差异的原因，我们需要按照 Java 在 REST 中作为一种表述的媒体类型的特征，来对它进行评估。

JavaScript 更好地适合于 Web 技术的开发模型。它具有低得多的门槛，既是因为它作为一种语言的总体复杂性比较小，也是因为一个新手程序员将他最初的工作代码整合起来需要花费的努力比较小。JavaScript 对于交互的可见性所产生的影响也比较少。独立的组织能够按照与复制 HTML 相同的方式来阅读、验证和复制 JavaScript 的源代码。与之相反，Java 是作为二进制包下载的——用户因此必需信任 Java 执行环境中的安全限制。同样，Java 拥有很多更多的功能，允许这些功能存在于一个安全环境中被认为是很可疑的，包括将 RMI 请求发送到来源服务器的能力。RMI 并不支持中间组件的可见性。

也许两者最重要的区别是，JavaScript 仅仅导致了很少的用户可觉察的延迟。JavaScript 通常作为主要表述的一部分来下载，然而 Java applet 要求一个独立的请求。Java 代码一旦被转换为字节代码的格式，要比通常的 JavaScript 代码大得多。最后一点是，当 HTML 页面的其余部分正在被下载时 JavaScript 就能够执行，Java 则要求包含类文件的完整的包被下载并且安装之后，应用才能够开始执行，因此 Java 并不支持增量的呈现。

一旦使用与 REST 约束背后的基本原理相同的规则来评估语言的特性，按照这些技术在现代 Web 架构中的行为来评估它们就变得容易多了。

6.6 小结

本章描述了在创作超文本转移协议（HTTP）和统一资源标识符（URI）的 Internet 标准的过程中学到的经验和教训。这两个规范定义了所有在 Web 上进行交互的组件所使用的通用的接口。此外，我以 libwww-perl 客户端库、Apache HTTP 服务器项目、以及协议标准的其他实现的形式，描述了从部署这些技术的过程中学到的经验和教训。

结论

我们每个人在内心深处都怀有一个梦想：创造一个鲜活的世界，一个宇宙。那些处在我们生活的中心、被训练为架构师的人们，拥有着这样的渴望：某一天，在某个地方，因为某种原因，创造出一个不可思议的、美丽的、夺人心魄的场所，在那里人们可以行走，可以梦想，历经数百年经久不衰。

——Christopher Alexander[3]

我们在 Internet 工程工作组（IETF）定义了现有的超文本转移协议（HTTP/1.0）[19]，并且为 HTTP/1.1[42]和统一资源标识符（URI）[21]的新标准设计扩展。在我们最初的工作中，我认识到需要建立一个关于万维网应该如何运转的模型。这个关于整个 Web 应用中的交互的理想化的模型被称作表述性状态转移（REST）架构风格，成为了现代 Web 架构的基础。它提供了指导原则，通过这些原则的指导，能够识别出先前存在于架构中的缺陷，并且使得各种扩展在部署之前就能够得到验证。

REST 是一组协作的架构约束，它试图使延迟和网络通信最小化，同时使组件实现的独立性和可伸缩性最大化。REST 通过将约束放置在连接器的语义上来达到这些目标，而其他的架构风格则聚焦于组件的语义。REST 支持交互的缓存和重用、动态替换组件、以及中间组件对于动作的处理，因此满足了一个 Internet 规模的分布式超媒体系统的需求。

这篇论文对于信息和计算机科学领域作出了如下贡献：

- 一个通过架构风格来理解软件架构的框架，包括了一组相容的术语，用来描述软件架构；
- 通过当某种架构风格被应用于一个分布式超媒体系统的架构时，它将导致的架构属性，来对基于网络的应用的架构风格进行分类。
- REST，一种新颖的分布式超媒体系统的架构风格；以及
- 在设计和部署现代万维网的架构的过程中，应用和评估 REST 架构风格。

现代 Web 是 REST 风格架构的一个实例。尽管基于 Web 的应用能够包括对于其他风格的交互的支持，但它的协议和性能所关注的焦点是分布式超媒体。REST 仅仅详细描述了架构中那些被认为对于 Internet 规模的分布式超媒体交互来说最为本质的部分。在现有协议无法表达组件交互的所有可能语义的地方，以及在语法细节能够被替换为更加有效的形式而不需要改变架构的能力的地方，都存在着 Web 架构需要加以改进的领域。同样地，协议扩展能够被拿来与 REST 进行比较，检查它们是否符合这个架构；如果不符合，更加有效的做法是将那个功能重定向到一个使用更加适用的架构风格并行运行的系统。

在一个理想的世界里，软件系统的实现将精确地匹配它的设计。现代 Web 架构的一些功能确实完全符合它们在 REST 中的设计标准，例如将 URI[21]作为资源标识符来使用，以及使用 Internet 媒体类型[48]来标识表述的数据格式。然而，由于未做遗留系统试验（但是必需要保持向后兼容性）和不了解架构风格的开发者所部署的扩展，在现代 Web 协议中也存在着一些不考虑架构设计的方面。REST 提供了一个模型，不仅可以用来开发和评估新的功能，也可以用来识别和理解破损的功能。

可以论证万维网是世界上最大型的分布式应用。理解在 Web 底层的关键架构原则能够帮助解释它在技术上的成功，并且为其他的分布式应用带来改进，特别是那些服从于相同的或相似的交互方法的应用。REST 既贡献了在现代 Web 软件架构背后的基本原理，也为我们上了重要的一课，展示了软件工程原则如何能够被系统地应用在对于一个真实软件系统的设计和评估的过程中。

对于基于网络的应用来说，系统的性能受到了网络通信的支配。对于一个分布式超媒体系统来说，组件交互由大粒度的数据转移组成，而不是由计算密集型的任务组成。REST 风格被开发出来以满足这些需求，它聚焦于资源和表述的通用的连接器接口，支持中间组件进行处理和缓存以及组件的替换，这使得基于 Web 的应用的规模从在 1994 年的每天 10 万个请求发展到了在 1999 年的每天 6 亿个请求。

通过对 HTTP/1.0[19]和 HTTP/1.1[42]标准 6 年的开发、精心设计的 URI[21]和相关的 URL[40]标准、以及在现代 Web 架构中成功地部署很多独立开发的、商业级的应用系统，REST 架构风格得到了验证。它既可以作为一个指导设计的模型，也可以作为一个对于 Web 协议的架构扩展的严格的测试。

将来的工作聚焦于对 HTTP/1.x 协议家族的替代者的开发，以及使用一个更加有效的符号化（tokenized）的语法来扩展这个架构指导，但不会丢失由 REST 识别出的想要得到的属性。无线设备的需求（它有着很多与 REST 背后的原则相同的特性）将会激发应用级协议设计和包括了主动的中间组件的架构这两方面的进一步发展。在扩展 REST 以便考虑可变的请求优先级、有区别的服务质量（quality-of-service）、以及由持续的数据流组成的表述（例如那些由广播的音频和视频来源生成的数据）等方面，也存在着一些兴趣。

参考文献

1. G. D. Abowd, R. Allen, and D. Garlan. Formalizing style to understand descriptions of software architecture. ACM Transactions on Software Engineering and Methodology, 4(4), Oct. 1995, pp. 319-364. A shorter version also appeared as: Using style to understand descriptions of software architecture. In Proceedings of the First ACM SIGSOFT Symposium on the Foundations of Software Engineering (SIGSOFT`93), Los Angeles, CA, Dec. 1993, pp. 9-20.

2. Adobe Systems Inc. PostScript Language Reference Manual. Addison-Wesley Publishing Company, Reading, Massachusetts, 1985.

3. C. Alexander. The Timeless Way of Building. Oxford University Press, New York, 1979.

4. C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, and S. Angel. A Pattern Language. Oxford University Press, New York, 1977.

5. R. Allen and D. Garlan. A formal basis for architectural connection. ACM Transactions on Software Engineering and Methodology, 6(3), July 1997. A shorter version also appeared as: Formalizing architectural connection. In Proceedings of the 16th International Conference on Software Engineering, Sorrento, Italy, May 1994, pp. 71-80. Also as: Beyond Definition/Use: Architectural Interconnection. In Proceedings of the ACM Interface Definition Language Workshop, Portland, Oregon, SIGPLAN Notices, 29(8), Aug. 1994.

6. G. Andrews. Paradigms for process interaction in distributed programs. ACM Computing Surveys, 23(1), Mar. 1991, pp. 49-90.

7. F. Anklesaria, et al. The Internet Gopher protocol (a distributed document search and retrieval protocol). Internet RFC 1436, Mar. 1993.

8. D. J. Barrett, L. A. Clarke, P. L. Tarr, A. E. Wise. A framework for event-based software integration. ACM Transactions on Software Engineering and Methodology, 5(4), Oct. 1996, pp. 378-421.

9. L. Bass, P. Clements, and R. Kazman. Software Architecture in Practice. Addison Wesley, Reading, Mass., 1998.

10. D. Batory, L. Coglianese, S. Shafer, and W. Tracz. The ADAGE avionics reference architecture. In Proceedings of AIAA Computing in Aerospace 10, San Antonio, 1995.

11. T. Berners-Lee, R. Cailliau, and J.-F. Groff. World Wide Web. Flyer distributed at the 3rd Joint European Networking Conference, Innsbruck, Austria, May 1992.

12. T. Berners-Lee, R. Cailliau, J.-F. Groff, and B. Pollermann. World-Wide Web: The information universe. Electronic Networking: Research, Applications and Policy, 2(1), Meckler Publishing, Westport, CT, Spring 1992, pp. 52-58.

13. T. Berners-Lee and R. Cailliau. World-Wide Web. In Proceedings of Computing in High Energy Physics 92, Annecy, France, 23-27 Sep. 1992.

14. T. Berners-Lee, R. Cailliau, C. Barker, and J.-F. Groff. W3 Project: Assorted design notes.

Published on the Web, Nov. 1992. Archived at <http://www.w3.org/History/19921103-hypertext/hypertext/WWW/WorkingNotes/Overview.html>, Sep. 2000.

15. T. Berners-Lee. Universal Resource Identifiers in WWW. Internet RFC 1630, June 1994.

16. T. Berners-Lee, R. Cailliau, A. Luotonen, H. Frystyk Nielsen, and A. Secret. The World-Wide Web. Communications of the ACM, 37(8), Aug. 1994, pp. 76-82.

17. T. Berners-Lee, L. Masinter, and M. McCahill. Uniform Resource Locators (URL). Internet RFC 1738, Dec. 1994.

18. T. Berners-Lee and D. Connolly. Hypertext Markup Language -- 2.0. Internet RFC 1866, Nov. 1995.

19. T. Berners-Lee, R. T. Fielding, and H. F. Nielsen. Hypertext Transfer Protocol -- HTTP/1.0. Internet RFC 1945, May 1996.

20. T. Berners-Lee. WWW: Past, present, and future. IEEE Computer, 29(10), Oct. 1996, pp. 69-77.

21. T. Berners-Lee, R. T. Fielding, and L. Masinter. Uniform Resource Identifiers (URI): Generic syntax. Internet RFC 2396, Aug. 1998.

22. P. Bernstein. Middleware: A model for distributed systems services. Communications of the ACM, Feb. 1996, pp. 86-98.

23. A. D. Birrell and B. J. Nelson. Implementing remote procedure call. ACM Transactions on Computer Systems, 2, Jan. 1984, pp. 39-59.

24. M. Boasson. The artistry of software architecture. IEEE Software, 12(6), Nov. 1995, pp. 13-16.

25. G. Booch. Object-oriented development. IEEE Transactions on Software Engineering, 12(2), Feb. 1986, pp. 211-221.

26. C. Brooks, M. S. Mazer, S. Meeks, and J. Miller. Application-specific proxy servers as HTTP stream transducers. In Proceedings of the Fourth International World Wide Web Conference, Boston, Massachusetts, Dec. 1995, pp. 539-548.

27. F. Buschmann and R. Meunier. A system of patterns. Coplien and Schmidt (eds.), Pattern Languages of Program Design, Addison-Wesley, 1995, pp. 325-343.

28. F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. Pattern-oriented Software Architecture: A system of patterns. John Wiley & Sons Ltd., England, 1996.

29. M. R. Cagan. The HP SoftBench Environment: An architecture for a new qeneration of software tools. Hewlett-Packard Journal, 41(3), June 1990, pp. 36-47.

30. J. R. Cameron. An overview of JSD. IEEE Transactions on Software Engineering, 12(2), Feb. 1986, pp. 222-240.

31. R. S. Chin and S. T. Chanson. Distributed object-based programming systems. ACM Computing Surveys, 23(1), Mar. 1991, pp. 91-124.

32. D. D. Clark and D. L. Tennenhouse. Architectural considerations for a new generation of protocols. In Proceedings of ACM SIGCOMM`90 Symposium, Philadelphia, PA, Sep. 1990, pp. 200-208.

33. J. O. Coplien and D. C. Schmidt, ed. Pattern Languages of Program Design. Addison-Wesley, Reading, Mass., 1995.

34. J. O. Coplien. Idioms and Patterns as Architectural Literature. IEEE Software, 14(1), Jan. 1997, pp. 36-42.

35. E. M. Dashofy, N. Medvidovic, R. N. Taylor. Using off-the-shelf middleware to implement connectors in distributed software architectures. In Proceedings of the 1999 International Conference on Software Engineering, Los Angeles, May 16-22, 1999, pp. 3-12.

36. F. Davis, et. al. WAIS Interface Protocol Prototype Functional Specification (v.1.5). Thinking Machines Corporation, April 1990.

37. F. DeRemer and H. H. Kron. Programming-in-the-large versus programming-in-the-small. IEEE Transactions on Software Engineering, SE-2(2), June 1976, pp. 80-86.

38. E. Di Nitto and D. Rosenblum. Exploiting ADLs to specify architectural styles induced by middleware infrastructures. In Proceedings of the 1999 International Conference on Software Engineering, Los Angeles, May 16-22, 1999, pp. 13-22.

39. R. T. Fielding. Maintaining distributed hypertext infostructures: Welcome to MOMspider's web. Computer Networks and ISDN Systems, 27(2), Nov. 1994, pp. 193-204.

40. R. T. Fielding. Relative Uniform Resource Locators. Internet RFC 1808, June 1995.

41. R. T. Fielding, E. J. Whitehead, Jr., K. M. Anderson, G. Bolcer, P. Oreizy, and R. N. Taylor. Web-based development of complex information products. Communications of the ACM, 41(8), Aug. 1998, pp. 84-92.

42. R. T. Fielding, J. Gettys, J. C. Mogul, H. F. Nielsen, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol -- HTTP/1.1. Internet RFC 2616, June 1999. [Obsoletes RFC 2068, Jan. 1997.]

43. R. T. Fielding and R. N. Taylor. Principled design of the modern Web architecture. In Proceedings of the 2000 International Conference on Software Engineering (ICSE 2000), Limerick, Ireland, June 2000, pp. 407-416.

44. D. Flanagan. JavaScript: The Definitive Guide, 3rd edition. O'Reilly & Associates, Sebastopol, CA, 1998.

45. D. Flanagan. JavaTM in a Nutshell, 3rd edition. O'Reilly & Associates, Sebastopol, CA, 1999.

46. J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, E. Sink, and L. Stewart. HTTP Authentication: Basic and Digest Access Authentication. Internet RFC 2617, June 1999.

47. N. Freed and N. Borenstein. Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies. Internet RFC 2045, Nov. 1996.

48. N. Freed, J. Klensin, and J. Postel. Multipurpose Internet Mail Extensions (MIME) Part Four: Registration Procedures. Internet RFC 2048, Nov. 1996.

49. M. Fridrich and W. Older. Helix: The architecture of the XMS distributed file system. IEEE Software, 2, May 1985, pp. 21-29.

50. A. Fuggetta, G. P. Picco, and G. Vigna. Understanding code mobility. IEEE Transactions on Software Engineering, 24(5), May 1998, pp. 342-361.

51. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-oriented Software. Addison-Wesley, Reading, Mass., 1995.

52. D. Garlan and E. Ilias. Low-cost, adaptable tool integration policies for integrated environments. In Proceedings of the ACM SIGSOFT `90: Fourth Symposium on Software Development Environments, Dec. 1990, pp. 1-10.

53. D. Garlan and M. Shaw. An introduction to software architecture. Ambriola & Tortola (eds.), Advances in Software Engineering & Knowledge Engineering, vol. II, World Scientific Pub Co., Singapore, 1993, pp. 1-39.

54. D. Garlan, R. Allen, and J. Ockerbloom. Exploiting style in architectural design environments. In Proceedings of the Second ACM SIGSOFT Symposium on the Foundations of Software Engineering (SIGSOFT`94), New Orleans, Dec. 1994, pp. 175-188.

55. D. Garlan and D. E. Perry. Introduction to the special issue on software architecture. IEEE Transactions on Software Engineering, 21(4), Apr. 1995, pp. 269-274.

56. D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch, or, Why it's hard to build systems out of existing parts. In Proceedings of the 17th International Conference on Software Engineering, Seattle, WA, 1995. Also appears as: Architectural mismatch: Why reuse is so hard. IEEE Software, 12(6), Nov. 1995, pp. 17-26.

57. D. Garlan, R. Monroe, and D. Wile. ACME: An architecture description language. In Proceedings of CASCON`97, Nov. 1997.

58. C. Ghezzi, M. Jazayeri, and D. Mandrioli. Fundamentals of Software Engineering. Prentice-Hall, 1991.

59. S. Glassman. A caching relay for the World Wide Web. Computer Networks and ISDN Systems, 27(2), Nov. 1994, pp. 165-173.

60. Y. Goland, E. J. Whitehead, Jr., A. Faizi, S. Carter, and D. Jensen. HTTP Extensions for Distributed Authoring -- WEBDAV. Internet RFC 2518, Feb. 1999.

61. K. Gr?nbaek and R. H. Trigg. Design issues for a Dexter-based hypermedia system. Communications of the ACM, 37(2), Feb. 1994, pp. 41-49.

62. B. Hayes-Roth, K. Pfleger, P. Lalanda, P. Morignot, and M. Balabanovic. A domain-specific software architecture for adaptive intelligent systems. IEEE Transactions on Software Engineering, 21(4), Apr. 1995, pp. 288-301.

63. J. Heidemann, K. Obraczka, and J. Touch. Modeling the performance of HTTP over several transport protocols. IEEE/ACM Transactions on Networking, 5(5), Oct. 1997, pp. 616-630.

64. K. Holtman and A. Mutz. Transparent content negotiation in HTTP. Internet RFC 2295, Mar. 1998.

65. P. Inverardi and A. L. Wolf. Formal specification and analysis of software architectures using the chemical abstract machine model. IEEE Transactions on Software Engineering, 21(4), Apr. 1995, pp. 373-386.

66. ISO/IEC JTC1/SC21/WG7. Reference Model of Open Distributed Processing. ITU-T X.901: ISO/IEC 10746-1, 07 June 1995.

67. M. Jackson. Problems, methods, and specialization. IEEE Software, 11(6), [condensed from Software Engineering Journal], Nov. 1994. pp. 57-62.

68. R. Kazman, L. Bass, G. Abowd, and M. Webb. SAAM: A method for analyzing the properties of software architectures. In Proceedings of the 16th International Conference on Software Engineering, Sorrento, Italy, May 1994, pp. 81-90.

69. R. Kazman, M. Barbacci, M. Klein, S. J. Carrière, and S. G. Woods. Experience with performing architecture tradeoff analysis. In Proceedings of the 1999 International Conference on Software Engineering, Los Angeles, May 16-22, 1999, pp. 54-63.

70. N. L. Kerth and W. Cunningham. Using patterns to improve our architectural vision. IEEE Software, 14(1), Jan. 1997, pp. 53-59.

71. R. Khare and S. Lawrence. Upgrading to TLS within HTTP/1.1. Internet RFC 2817, May 2000.

72. G. E. Krasner and S. T. Pope. A cookbook for using the Model-View-Controller user interface paradigm in Smalltalk-80. Journal of Object Oriented Programming, 1(3), Aug.-Sep. 1988, pp. 26-49.

73. D. Kristol and L. Montulli. HTTP State Management Mechanism. Internet RFC 2109, Feb. 1997.

74. P. B. Kruchten. The 4+1 View Model of architecture. IEEE Software, 12(6), Nov. 1995, pp. 42-50.

75. D. Le Métayer. Describing software architectural styles using graph grammars. IEEE Transactions on Software Engineering, 24(7), July 1998, pp. 521-533.

76. W. C. Loerke. On Style in Architecture. F. Wilson, Architecture: Fundamental Issues, Van Nostrand Reinhold, New York, 1990, pp. 203-218.

77. D. C. Luckham, J. J. Kenney, L. M. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and analysis of system architecture using Rapide. IEEE Transactions on Software Engineering, 21(4), Apr. 1995, pp. 336-355.

78. D. C. Luckham and J. Vera. An event-based architecture definition language. IEEE Transactions on Software Engineering, 21(9), Sep. 1995, pp. 717-734.

79. A. Luotonen and K. Altis. World-Wide Web proxies. Computer Networks and ISDN Systems, 27(2), Nov. 1994, pp. 147-154.

80. P. Maes. Concepts and experiments in computational reflection. In Proceedings of OOPSLA `87, Orlando, Florida, Oct. 1987, pp. 147-155.

81. J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In Proceedings of the 5th European Software Engineering Conference (ESEC`95), Sitges, Spain, Sep. 1995, pp. 137-153.

82. J. Magee and J. Kramer. Dynamic structure in software architectures. In Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering (SIGSOFT`96), San Francisco, Oct. 1996, pp. 3-14.

83. F. Manola. Technologies for a Web object model. IEEE Internet Computing, 3(1), Jan.-Feb. 1999, pp. 38-47.

84. H. Maurer. HyperWave: The Next-Generation Web Solution. Addison-Wesley, Harlow, England, 1996.

85. M. J. Maybee, D. H. Heimbigner, and L. J. Osterweil. Multilanguage interoperability in distributed systems: Experience Report. In Proceedings 18th International Conference on Software Engineering, Berlin, Germany, Mar. 1996.

86. N. Medvidovic and R. N. Taylor. A framework for classifying and comparing architecture description languages. In Proceedings of the 6th European Software Engineering Conference held jointly with the 5th ACM SIGSOFT Symposium on the Foundations of Software Engineering, Zurich, Switzerland, Sep. 1997, pp. 60-76.

87. N. Medvidovic. Architecture-based Specification-time Software Evolution. Ph.D. Dissertation, University of California, Irvine, Dec. 1998.

88. N. Medvidovic, D. S. Rosenblum, and R. N. Taylor. A language and environment for architecture-based software development and evolution. In Proceedings of the 1999 International Conference on Software Engineering, Los Angeles, May 16-22, 1999, pp. 44-53.

89. A. Mockus, R. T. Fielding, and J. Herbsleb. A case study of open source software development: The Apache server. In Proceedings of the 2000 International Conference on Software Engineering (ICSE 2000), Limerick, Ireland, June 2000, pp. 263-272.

90. J. Mogul, R. Fielding, J. Gettys, and H. Frystyk. Use and Interpretation of HTTP Version Numbers. Internet RFC 2145, May 1997.

91. R. T. Monroe, A. Kompanek, R. Melton, and D. Garlan. Architectural Styles, Design Patterns, and Objects. IEEE Software, 14(1), Jan. 1997, pp. 43-52.

92. M. Moriconi, X. Qian, and R. A. Riemenscheider. Correct architecture refinement. IEEE Transactions on Software Engineering, 21(4), Apr. 1995, pp. 356-372.

93. H. F. Nielsen, J. Gettys, A. Baird-Smith, E. Prud'hommeaux, H. Lie, and C. Lilley. Network Performance Effects of HTTP/1.1, CSS1, and PNG. Proceedings of ACM SIGCOMM '97, Cannes, France, Sep. 1997.

94. H. F. Nielsen, P. Leach, and S. Lawrence. HTTP extension framework, Internet RFC 2774, Feb. 2000.

95. H. Penny Nii. Blackboard systems. AI Magazine, 7(3):38-53 and 7(4):82-107, 1986.

96. Object Management Group. Object Management Architecture Guide, Rev. 3.0. Soley & Stone (eds.), New York: J. Wiley, 3rd ed., 1995.

97. Object Management Group. The Common Object Request Broker: Architecture and Specification (CORBA 2.1). <http://www.omg.org/>, Aug. 1997.

98. P. Oreizy, N. Medvidovic, and R. N. Taylor. Architecture-based runtime software evolution. In Proceedings of the 1998 International Conference on Software Engineering, Kyoto, Japan, Apr. 1998.

99. P. Oreizy. Decentralized software evolution. Unpublished manuscript (Phase II Survey Paper), Dec. 1998.

100. V. N. Padmanabhan and J. C. Mogul. Improving HTTP latency. Computer Networks and ISDN Systems, 28, Dec. 1995, pp. 25-35.

101. D. L. Parnas. Information distribution aspects of design methodology. In Proceedings of IFIP Congress 71, Ljubljana, Aug. 1971, pp. 339-344.

102. D. L. Parnas. On the criteria to be used in decomposing systems into modules. Communications of the ACM, 15(12), Dec. 1972, pp. 1053-1058.

103. D. L. Parnas. Designing software for ease of extension and contraction. IEEE Transactions on Software Engineering, SE-5(3), Mar. 1979.

104. D. L. Parnas, P. C. Clements, and D. M. Weiss. The modular structure of complex systems. IEEE Transactions on Software Engineering, SE-11(3), 1985, pp. 259-266.

105. D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. ACM SIGSOFT Software Engineering Notes, 17(4), Oct. 1992, pp. 40-52.

106. J. Postel and J. Reynolds. TELNET Protocol Specification. Internet STD 8, RFC 854, May 1983.

107. J. Postel and J. Reynolds. File Transfer Protocol. Internet STD 9, RFC 959, Oct. 1985.

108. D. Pountain and C. Szyperski. Extensible software systems. Byte, May 1994, pp. 57-62.

109. R. Prieto-Diaz and J. M. Neighbors. Module interconnection languages. Journal of Systems and Software, 6(4), Nov. 1986, pp. 307-334.

110. J. M. Purtilo. The Polylith software bus. ACM Transactions on Programming Languages and Systems, 16(1), Jan. 1994, pp. 151-174.

111. M. Python. The Architects Sketch. Monty Python's Flying Circus TV Show, Episode 17, Sep. 1970. Transcript at <http://www.stone-dead.asn.au/sketches/architec.htm>.

112. J. Rasure and M. Young. Open environment for image processing and software development. In Proceedings of the 1992 SPIE/IS&T Symposium on Electronic Imaging, Vol. 1659, Feb. 1992.

113. S. P. Reiss. Connecting tools using message passing in the Field environment. IEEE Software, 7(4), July 1990, pp. 57-67.

114. D. S. Rosenblum and A. L. Wolf. A design framework for Internet-scale event observation and notification. In Proceedings of the 6th European Software Engineering Conference held jointly with the 5th ACM SIGSOFT Symposium on the Foundations of Software Engineering, Zurich, Switzerland, Sep. 1997, pp. 344-360.

115. R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun network filesystem. In Proceedings of the Usenix Conference, June 1985, pp. 119-130.

116. M. Shapiro. Structure and encapsulation in distributed systems: The proxy principle. In Proceedings of the 6th International Conference on Distributed Computing Systems, Cambridge, MA, May 1986, pp. 198-204.

117. M. Shaw. Toward higher-level abstractions for software systems. Data & Knowledge Engineering, 5, 1990, pp. 119-128.

118. M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnick. Abstractions for software architecture and tools to support them. IEEE Transactions on Software Engineering, 21(4), Apr. 1995, pp. 314-335.

119. M. Shaw. Comparing architectural design styles. IEEE Software, 12(6), Nov. 1995, pp. 27-41.

120. M. Shaw. Some patterns for software architecture. Vlissides, Coplien & Kerth (eds.), Pattern Languages of Program Design, Vol. 2, Addison-Wesley, 1996, pp. 255-269.

121. M. Shaw and D. Garlan. Software Architecture: Perspectives on an Emerging Discipline. Prentice-Hall, 1996.

122. M. Shaw and P. Clements. A field guide to boxology: Preliminary classification of architectural styles for software systems. In Proceedings of the Twenty-First Annual International Computer Software and Applications Conference (COMPSAC`97), Washington, D.C., Aug. 1997, pp. 6-13.

123. A. Sinha. Client-server computing. Communications of the ACM, 35(7), July 1992, pp. 77-98.

124. K. Sollins and L. Masinter. Functional requirements for Uniform Resource Names. Internet RFC 1737, Dec. 1994.

125. S. E. Spero. Analysis of HTTP performance problems. Published on the Web, <http://metalab.unc.edu/mdma-release/http-prob.html>, 1994.

126. K. J. Sullivan and D. Notkin. Reconciling environment integration and software evolution. ACM Transactions on Software Engineering and Methodology, 1(3), July 1992, pp. 229-268.

127. A. S. Tanenbaum and R. van Renesse. Distributed operating systems. ACM Computing Surveys, 17(4), Dec. 1985, pp. 419-470.

128. R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. Whitehead Jr., J. E. Robbins, K. A. Nies, P. Oreizy, and D. L. Dubrow. A component- and message-based architectural style for GUI software. IEEE Transactions on Software Engineering, 22(6), June 1996, pp. 390-406.

129. W. Tephenhart and J. J. Cusick. A unified object topology. IEEE Software, 14(1), Jan. 1997, pp. 31-35.

130. W. Tracz. DSSA (domain-specific software architecture) pedagogical example. Software Engineering Notes, 20(3), July 1995, pp. 49-62.

131. A. Umar. Object-Oriented Client/Server Internet Environments. Prentice Hall PTR, 1997.

132. S. Vestal. MetaH programmer's manual, version 1.09. Technical Report, Honeywell Technology Center, Apr. 1996.

133. J. Waldo, G. Wyant, A. Wollrath, and S. Kendall. A note on distributed computing. Technical Report SMLI TR-94-29, Sun Microsystems Laboratories, Inc., Nov. 1994.

134. L. Wall, T. Christiansen, and R. L. Schwartz. Programming Perl, 2nd ed. O'Reilly & Associates, 1996.

135. E. J. Whitehead, Jr., R. T. Fielding, and K. M. Anderson. Fusing WWW and link server technology: One approach. In Proceedings of the 2nd Workshop on Open Hypermedia Systems, Hypertext'96, Washington, DC, Mar. 1996, pp. 81-86.

136. A. Wolman, G. Voelker, N. Sharma, N. Cardwell, M. Brown, T. Landray, D. Pinnel, A. Karlin, and H. Levy. Organization-based analysis of Web-object sharing and caching. In Proceedings of the 2nd USENIX Conference on Internet Technologies and Systems (USITS), Oct. 1999.

137. W. Zimmer. Relationships between design patterns. Coplien and Schmidt (eds.), Pattern Languages of Program Design, Addison-Wesley, 1995, pp. 345-364.

138. H. Zimmerman. OSI reference model -- The ISO model of architecture for open systems interconnection. IEEE Transactions on Communications, 28, Apr. 1980, pp. 425-432.