InfoQ 软件开发丛书 *Software Development Series*

架构风格与基于网络应用软件的架构设计（中文修订版）

◎作者：Roy Thomas Fielding博士
◎译者：李锟　◎审校：马国耀

InfoQ

架构风格与基于网络应用软件的架构设计

（博士论文）
（中文修订版）

作者：Roy Thomas Fielding 博士

译者：李锟
审校：马国耀

作者简介：

Roy Thomas Fielding 博士是 HTTP 和 URI 两个 Web 基础技术架构规范的主要设计者，Apache HTTP 服务器的主要开发者，Apache 软件基金会的合作创始人和前主席。他所做的具有开创性的杰出工作为 Web 的技术架构奠定了坚实基础，没有这些基础，Web 的几何级数式发展是完全不可想象的。

缘起：

本论文是互联网发展史上一篇非常重要的技术文献。出于社会责任感，译者认为非常有必要将它介绍给国人，使国人得以一窥 HTTP 和 URI 等 Web 基础技术架构规范背后的设计原则。基于相同的设计原则，Web 开发者们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。

中文版最初版本的译者为李锟、廖志刚、刘丹、杨光四人，由李锟负责全部内容的审校和润色。于 2007 年下旬完成，发布在满江红开源网站。在此向廖志刚、刘丹、杨光曾经作出的辛勤工作表示诚挚的感谢。

在 2013 年年底，李锟对中文版最初版本进行了修订，纠正了因为译者水平有限而导致的大量翻译错误，并且使得语句阅读起来更加流畅。中文修订版于 2014 年 2 月制作完成，发布在 InfoQ 中文站。马国耀负责全部内容的审校，他的工作使得中文版的质量有了极大提升，在此向他表示衷心感谢。此外，在“REST 实战讨论组”（QQ 群号：81207617）中的一些热心网友也参与了相关的讨论。他们的姓名无法一一列举，在此也向他们表示感谢。

原文链接：

Architectural Styles and the Design of Network-based Software Architectures

中文版版式说明：

论文中文版的版式参考了英文版 A4 尺寸 PDF 的排版样式，与英文版尽量保持一致。正文中的中文统一使用宋体，英文版中为斜体的专业术语、引用文字统一使用楷体，英文版中为粗体的专业术语在中文版中同样为粗体。同一英文专业术语在全文中有相同的中文翻译，以确保读者获得一致的阅读体验。在容易引起歧义的语句和术语之后都附加了原文，以便细心的读者加以指正。

中文版的读者想要参与深入的讨论，可加入“REST 实战讨论组”的 QQ 群，或者向译者发电子邮件。译者的邮箱为：dlee.cn@gmail.com。

目录

致谢

当我在加州大学欧文分校（UCI）念博士生期间，我与该校的教师、员工和学生们一同工作，感到非常的愉快。假如我未能获得自由从事于自己所感兴趣的研究，这些工作是根本不可能完成的。大部分的感谢应该归于我长期以来的导师、论文答辩委员会主席 Dick Taylor 教授，感谢他的和善和给予我的大量指导。Mark Ackerman 也应该得到很多感谢，正是他在 1993 年讲授的分布式信息服务课程，引导我走进了 Web 开发者社区，并且开展了本论文中描述的所有设计工作。同样的，David Rosenblum 在互联网规模软件架构方面的工作，使得我确信应该从架构的层面上思考我自己的研究，而并非仅仅思考超媒体或者应用层协议的设计。

Web 的架构风格是在 6 年时间里采用迭代的方式开发出来的，但是主要是在 1995 年最初的 6 个月时间里。它受到了与很多 UCI 的研究者、W3C 的工作人员、IETF 的 HTTP 和 URI 工作组的工程师们无数次讨论的影响。我特别想要感谢 Tim Berners-Lee、Henrik Frystyk Nielsen、Dan Connolly、Dave Raggett、Rohit Khare、Jim Whitehead、Larry Masinter 和 Dan LaLiberte。关于 Web 架构的特性和目标，我与他们开展了很多深思熟虑的对话（many thoughtful conversations）。我也想要感谢 Ken Anderson，为他关于开放的超文本社区的真知灼见，和在 UCI 所做的关于超媒体的开创性研究。同样要感谢在我之前就在 UCI 从事软件架构研究的同行们，包括 Peyman Oreizy、Neno Medvidovic、Jason Robbins 和 David Hilbert。

Web 的架构是基于大量软件开发志愿者的协同工作而形成的，他们几乎没有人为自己在 Web 成为一种商业现象之前的开创性工作得到过赞誉。除了 W3C 的成员外，服务器端开发者的贡献应该得到认可，这些服务器促使 Web 在 1993-1994 年飞速发展（我相信这比浏览器更为重要）。这些开发者包括 Rob McCool (开发了 NCSA httpd)、Ari Luotonen (开发了 CERN httpd/proxy)和 Tony Sanders(开发了 Plexus)。同样需要感谢“Mr. Content”（满意先生）Kevin Hughes，他超越了超文本，首次在 Web 上实现了展示信息的大多数方式。早期的客户端开发者同样值得感谢：Nicola Pellow (开发了 line-mode)、Pei Wei (开发了 Viola)、Tony Johnson (开发了 Midas)、Lou Montulli (开发了 Lynx)、Bill Perry (开发了 W3)、Marc Andreessen 和 Eric Bina (开发了基于 X Window 的 Mosaic)。 最后，我个人感谢 libwww-perl 项目的合作者们：Oscar Nierstrasz、Martijn Koster 和 Gisle Aas，感谢大家的付出！

现代 Web 架构更多地是由私人志愿者，而不是由任何单独的公司来定义的。主要的一些志愿者都是 Apache 软件基金会的成员。特别感谢 Robert S.，他为 Apache 1.0 设计的 Shambhala 架构具有难以致信的健壮性。关于哪些 Web 扩展是所期待的，哪些是想要避免的，我与他开展了很多讨论。感谢 Dean Gaudet 教会了我更多关于系统性能演化方面的知识，比我认为我需要知道的还要多得多。感谢 Alexei Kosut 第一个在 Apache 中实现了大多数 HTTP/1.1 协议的功能。额外的感谢归于 Apache 开发组（the Apache Group）的创建者，包括 Brian Behlendorf、Rob Hartill、David Robinson、Cliff Skolnick、Randy Terbush 和 Andrew Wilson，他们创建了一个我们能够引以为傲和再次改变世界的社区。

我还想要感谢 eBuilt 的所有人们，他们使得这里成为一个如此伟大的工作场所。特别感谢 4 位技术创始人——Joe Lindsay、Phil Lindsay、Jim Hayes 和 Joe Manna ——他们创建了一种使得工程变得很有趣的文化。感谢 Mike Dewey、Jeff Lenardson、Charlie Bunten 和 Ted Lavoie，

他们使得我有可能在赚钱的同时又享受到乐趣。特别感谢 Linda Dailing，她是将我们所有人融合在一起的融合剂。

感谢在 Endeavors Technology 公司的团队：Greg Bolcer、Clay Cover、Art Hitomi 和 Peter Kammer，并祝愿他们好运。最后，我想要感谢我的三位缪斯——Laura、Nikki 和 Ling——感谢她们为我在撰写论文期间为我带来的灵感。

我的博士论文研究得到了国防高等研究计划署（Defense Advanced Research Projects Agency）和空军研究实验室（Airforce Research Laboratory）、空军器材司令部（Air Force Materiel Command）、美国空军（USAF）的赞助，合同号为 F30602-97-2-0021。因此我授权美国政府，可以为了与政府有关的目的重新制作和分发本论文，而无须附加任何版权标记。本论文中的观点和结论全部是作者本人的，不应该被视作代表了国防高等研究计划署、空军研究实验室或美国政府的官方政策或背书，无论是明确的或隐含的。

论文摘要

架构风格与基于网络应用软件的架构设计

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

Web（万维网，英文全称 World Wide Web，简称 Web）的成功，很大程度上是因为其软件架构的设计满足了拥有互联网规模（Internet-scale）的分布式超媒体系统的需求。在过去 10 年间，通过对定义 Web 架构的规范所做的一系列修改，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必要的修改，需要一种现代的 Web 架构模型，用来指导 Web 的设计、定义和部署。

软件架构方面的研究探索的是如何以最佳的方式划分一个系统、如何标识组件、组件之间如何通信、信息如何表达、系统中的元素如何独立地进化，以及上述所有内容如何使用形式化的和非形式化的符号加以描述。我所做工作的动机是希望理解和评估基于网络应用软件的架构设计（the architectural design of network-based application software），通过有原则地使用架构约束，从而从架构中获得所期待的功能、性能和社交三方面的属性。一种架构风格是一组已命名的、相互协作的架构约束。

这篇论文定义了一个框架，它致力于通过架构风格来理解软件架构，它还展示了如何使用架构风格来指导基于网络应用的架构设计。本文对基于网络应用的架构风格做了一番调查，根据不同的架构风格在为分布式超媒体（distributed hypermedia）设计的架构中产生的架构属性，来对这些架构风格进行分类。然后我介绍了 REST（表述性状态移交，英文全称 Representational State Transfer，简称 REST）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 的软件工程原则和为支持这些原则而选择的交互约束，并将它们与其他架构风格的约束进行了对比。最后，我描述了从把 REST 应用于 HTTP（超文本移交协议）和 URI（统一资源标识符）两个规范，并进一步将这两个规范部署到 Web 客户端和服务器软件的过程中学习到的经验教训。

绪论

抱歉……您说的难道是“屠宰刀”？
——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。软件架构方面的研究探索的是如何以最佳的方式划分一个系统、如何标识组件、组件之间如何通信、信息如何表达、组成系统的元素如何独立地进化，以及上述所有内容如何使用形式化的和非形式化的符号加以描述。

一个优秀的架构并非是靠凭空想象得到的。每一个架构级的设计决策，都应该是根据被设计系统的功能、行为和社交三方面的需求而作出的。这是一个基本的设计原则，既适用于软件架构领域，同样也适用于传统的建筑架构领域。“形式追随功能”的指导方针来自于从数百年失败的建筑项目中获得的经验，但是它却常常被软件从业者们所忽视。上面引用的那句滑稽搞笑的话来自于 Monty Python 系列讽刺剧，这句话表达的是当一位建筑师在面对设计一个城市住宅区的目标时，头脑里所抱有的荒诞想法。他想要使用所有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，但是对于预期的居民来说却谈不上舒适，因为他们将不得不战战兢兢地行穿行在安装着旋转式屠宰刀的走廊中。

《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最时髦的架构设计，到了后来却发现满足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，出现很多此类行为是由于设计者并不理解为何（why）一组特定的架构约束是有用的。换句话说，当选择那些优秀的软件架构来使用时，它们背后的推理过程（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）边界上的交汇点。软件方面的研究长期以来关注于对软件设计进行分类和开发设计方法学，但是却很少能够客观地评估不同的设计选择对于系统行为的影响。网络方面的研究则恰恰相反，聚焦于系统之间普通通信行为的细节和提高特殊通信技术的性能，却常常忽略了一个事实，即改变一个应用的交互风格（the interaction style）对于性能产生的影响要比改变交互所使用的通信协议更大。我的工作的动机是希望理解和评估基于网络应用的架构设计，通过有原则地使用架构约束，从而从架构中获得所期待的功能、性能和社交三方面的属性。当为一组相互协作的架构约束取了一个名字之后，这组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，展示了如何使用架构风格来指导基于网络应用软件的架构设计。当将常见的架构风格应用于基于网络的超媒体系统（network-based hypermedia）的架构设计时，将会产生一系列架构属性，根据这些架构属性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 Web 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 旨在成为一个互联网规模（Internet-scale）的分布式超媒体系统，这意味着它的内涵远远超越了单纯的地理上的分布。互联网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满足无法控制的可伸缩性（anarchic scalability）和软件组件的独立部署（the independent deployment of software components）两方面的要求。通过将动作控件（action controls）内嵌在

从远程站点获取到的信息表述之中，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）的数据对象进行通信。

第 5 章介绍并详细描述了为分布式超媒体系统设计的 REST（表述性状态移交）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 的软件工程原则和为支持这些原则而选择的交互约束，并将它们与其他架构风格的约束进行了对比。

如同第 6 章中所展示的那样，在过去的 6 年间，我们使用 REST 架构风格来指导现代 Web 架构（译者注：与“早期 Web 架构”相对应）的设计和开发。这个工作是与我所创作的 HTTP（超文本移交协议）和 URI（统一资源标识符）两个互联网规范共同完成的，这两个规范定义了在 Web 上进行交互的所有组件所使用的通用接口。

就像大多数真实世界中的系统一样，并非所有已部署的 Web 架构组件都服从于其架构设计中所给出的每一个约束。REST 既可以被用作定义架构改进的方法，也可以被用作识别架构不匹配（architectural mismatches）的方法。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，就会发生架构不匹配。尽管往往无法避免架构不匹配，但是有可能在它们成为正式规范之前识别出它们。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背离 REST 进行了分析。

概括来说，这篇论文对于互联网和计算机科学领域的软件研究作出了如下贡献：

- 定义了一个通过架构风格来理解软件架构的框架，包括一组自恰的术语，用来描述软件架构；
- 通过当某种架构风格被应用于一个为分布式超媒体系统（distributed hypermedia systems）设计的架构时，将会产生的架构属性，来对基于网络应用软件的架构风格进行分类。
- 描述了 REST——一种为分布式超媒体系统设计的新型架构风格；以及
- 在设计和部署现代 Web 架构的过程中，应用和评估 REST 架构风格。

第 1 章 软件架构

尽管软件架构的研究领域吸引了很多人的兴趣，但是对于在架构的定义中应该包含什么，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方面。在本章中，以检查文献中现有的定义和我自己在基于网络应用的架构方面的领悟为基础，定义了一套自恰的软件架构术语。每一个定义都使用方框突出显示，随后讨论该定义是如何得来的，或者与相关研究进行比较。

1.1 运行时抽象（Run-time Abstraction）

> **软件架构**是一个软件系统在其运行过程中某个阶段的运行时元素（run-time elements）的抽象。一个系统可能由很多层抽象和很多个运行阶段组成，每一个抽象和运行阶段都有自己的软件架构。

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架构属性[117]。一个复杂的系统会包含有多层抽象，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，架构元素（architectural elements）可以通过自身提供给同一层其他元素的抽象接口来描述[9]。在每一个架构元素之中，可能还存在着另一个架构，由其定义由其众多子元素构成的系统，该系统实现了由父元素的抽象接口所展示的行为。这样的架构可以递归下去直到最基本的系统元素：不能再被分解为抽象层次更低的元素。

除了架构的层次，软件系统通常会有多个运行阶段，例如启动、初始化、正常处理、重新初始化和停止。每个运行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不会被当作一个架构元素，因为在这个阶段配置文件中的信息已经分布到了整个系统之中。事实上，也可以用配置文件来定义正常处理阶段的架构。系统架构的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”（transformers of data），而 Shaw 等人[118]将组件描述为“计算和状态的所在地”（the locus of computation and state）。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。将源代码的模块结构与正在运行的系统中的行为部件对应起来，以及使用相同的代码部分（例如共享库）来实现独立的软件组件，尽管这些做法确实有很多好处，然而我们将软件的架构和源代码结构分离开是为了更好地关注软件在运行时的特性，这些特性并不依赖于某个特定的组件实现。因此，尽管架构的设计和源代码结构的设计关系密切，它们其实是相互分离的设计活动。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 架构元素（Elements）

> **软件架构是**由一些架构元素（组件、连接器和数据）的配置来定义的，这些元素之间的关系受到约束，以获得所期待的一组架构属性。

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特定形式（form）的架构元素（elements），并通过一组基本原理（rationale）来描述。架构元素包括处理元素、数据元素、连接元素，其形式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。通过捕获选择架构风格、以及选择架构元素和形式的动机，这些基本原理为架构提供了底层的基础。

我的软件架构定义建立在 Perry 和 Wolf [105] 的模型基础之上，是一个更加详尽的版本，但是没有包括基本原理的部分。尽管基本原理是软件架构研究中很重要的一个方面，尤其是在对于架构的描述方面，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，确实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基本原理而独立存在。反射型的系统（reflective systems）[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低层次的架构替换另一个低层次的架构，而不是将基本原理包括在那些架构之中。

用一个类比来做说明，想象一下，当一个大楼的蓝图和设计计划被烧毁了将会发生什么事情？大楼会瞬间倒塌么？不会的，因为支撑着屋顶重量的墙体仍然完好无损。按照设计，一个架构会拥有一组架构属性，允许该架构满足甚至超出系统的需求。忽视这些架构属性，在将来的改造中可能会违反架构的约束，这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，而是包括了架构属性。基本原理详细说明了这些架构属性，缺少基本原理可能会导致架构随时间的推移而逐渐退化，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，按照一些组件和这些组件之间的交互来定义一个软件系统的架构。除了指出了系统的结构和拓扑以外，架构还展示了期望实现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。

在 Shaw 等人的模型中，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中能够看到的东西：方框（组件）和直线（连接器），而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。这样的一个模型不足以描述基于网络应用软件的架构，因为对于这类软件而言，数据元素在系统中的特性、位置和移动，常常是系统行为唯一的和最重要的决定因素（the single most significant determinant）。

1.2.1 组件（Components）

组件是软件指令和内部状态的抽象单元，通过其接口提供数据的转换能力。

在软件架构中，组件是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位于连接器中的软件（software within connectors）区分开来。

组件是软件指令和内部状态的抽象单元，通过其接口提供数据的转换能力。数据转换的例子包括从二级存储将数据加载到内存、执行一些计算、转换为另外一种格式、封装在其他数据之中等等。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）或辨别出来（discerned）[9]。换句话说，应该由某个组件为其他组件提供的接口和服务来定义该组件，而不是由隐藏在该组件提供的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（Connectors）

连接器是对于组件之间的通讯、配合或者协作进行中间斡旋的一种抽象机制。

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：连接器是对于组件之间的通讯、配合或者协作进行中间斡旋的一种抽象机制。连接器的示例有：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。连接器将数据元素从它的一个接口移交（transferring）到另一个接口而不改变数据，以此来支持组件之间的通信。在其内部，连接器可以包含一个由组件组成的子系统，为了移交的目的对数据进行某种转换、执行移交、然后做相反的转换并交付与原始数据相同的结果。然而，在架构所能捕获到的外部行为抽象中，可以忽略这些细节。与之相反，从外部的角度观察，组件可以（尽管并非总会这样）对数据进行转换。

1.2.3 数据（Data）

数据是组件通过连接器接收或发送的信息元素。

上面已经提到，是否包含数据元素是 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面。Jackson [67]也有相似的观点。

数据是组件通过连接器接收或发送的信息元素。数据的示例有：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，即文件系统组件从其接口接收到“文件名”数据，并将其转换为记录在（隐藏的）内部存储系统中的字节序列。组件也能够生成数据，例如与一个时钟或传感器相对应的软件封装。

在基于网络应用的架构中，数据元素的特性往往决定了一种特定的架构风格是否适用。在对移动代码设计范例（mobile code design paradigms）的比较中 [50] 尤其明显，在这里你必须要在两种架构风格中做出选择：是直接与组件进行交互；还是将组件转换为一个数据元素，通过网络移交该数据元素，然后再对该数据元素做相反的转换，得到一个能够在本地与之交互的组件。如果不在架构层面上考虑数据元素，是不可能对架构做出评估的。

1.3 配置（Configurations）

配置是在系统运行期间的组件、连接器和数据之间的架构关系的结构。

Abowd 等人[1]将架构的描述定义为：支持根据三个基本的语义分类来对系统进行描述。这三个语义分类是：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架构风格相关的形象化符号来可视化地展示这些概念，以便于描述合法的计算和交互，以及所期待的系统约束。

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，使得架构师更容易区分主动的配置与所有合法配置可能会影响的领域。Medvidovic 和 Taylor [86]提出了额外基本原理（additional rationale），用来在架构描述语言中区分出配置相关的内容。

1.4 架构属性（Properties）

软件架构的**架构属性**集合包括了对组件、连接器和数据的选择和排列所产生的所有属性。架构属性的例子包括了由系统获得的功能属性（functional properties achieved by the system）以及非功能属性（例如，进化的容易程度、组件的可重用性、效率、动态扩展能力，这些常常被称作品质属性）（quality attributes）[9]。

架构属性是由架构中的一组架构约束所产生的。架构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）架构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”，目的是获得上述两个所期待的品质，当在架构中实现了这种架构风格时，这两个品质就成为了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，取决于所期待的系统本身的特性。在 2.3 节检查了那些对于基于网络应用的架构特别重要的架构属性。

1.5 架构风格（Styles）

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

因为一种架构既包含功能属性又包含非功能属性，直接比较不同类型的系统的架构，或者甚至比较在不同环境中的相同类型系统的架构，都会比较困难。架构风格是一种机制，用来对架构进行分类并且定义它们的公共特征 [38]。每一种架构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：对于各种架构元素类型（element types）的抽象和多个特定架构的形式化方面（formal aspects，也许仅仅集中在架构的一些特定方面，而不是架构的所有方面）。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。这个定义允许架构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架构风格。明确地说，一种架构风格决定了在此架构风格的架构实例中能够使用哪些组件和连接器，以及一组能够将它们组合在一起的约束[53]。对于架构风格的这种狭隘观点，是他们的软件架构定义的直接结果——即，将架构看作是形式化的描述，而不是正在运行的系统，这就导致了他们仅仅基于从包含方框和直线的图中总结出

来的共享模式（shared patterns）来进行抽象。Abowd 等人[1]更进一步，将架构风格明确定义为：用来解释一类架构描述（architectural descriptions）的一个约定的集合（collection of conventions）。

新的架构能够被定义为特定架构风格的实例（instances）[38]。因为架构风格可以强调软件架构的不同方面，一个特定的架构可能是由多种架构风格组成的。同样地，可以通过将多种基本架构风格组合为单个协作式架构风格（coordinated style），来形成一种混合的架构风格。

一些架构风格常常被描述为适合于所有形式软件的“银弹”式解决方案。然而，一个好的设计者应该选择一种与正在解决的特定问题最为匹配的架构风格 [119]。为一个基于网络应用选择正确的架构风格，必须要理解该应用所属的问题领域 [67]。因此需要了解应用的通信需求，理解不同的架构风格和它们所导致的特殊问题，并且有能力预测每种交互风格对基于网络通信的特性（the characteristics of network-based communication）的敏感度（sensitivity）[133]。

不幸的是，使用术语“风格”来表达一组相互协作的架构约束，常常会令人感到困惑。这种用法与“风格”一词在词典中的用法有很大的不同，后者强调的是设计过程的个性化。Loerke [76]专门用一章来贬低专业建筑师在工作中为个性化风格保留位置的想法。与之相反，他将风格描述为挑剔者对于过去架构（past architecture）的观点，即，应该由可选择的原料、社区文化、统治者的自我意识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于设计之上的约束，达到或复制一种特定的风格应该是设计者的最低目标。由于将一组已命名的架构约束称作一种风格，使得对公共约束（common constraints）的特征的表达变得更加容易，我们使用架构风格作为一种进行抽象的方法，而不是暗示存在一种个性化的设计。

1.6 模式和模式语言（Patterns and Pattern Languages）

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，都是基于 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一组预期交互的方法。换句话说，通过在设计和实现选择（implementation choices）方面遵循一条固定的路径，一种模式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 提出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（人类的活动和情感）事件的模式。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，识别出在目标文化（target culture）中有哪些公共的生活模式（pattern of life，译者注：可以与上文说的“事件的模式”关联起来），确定有哪些架构约束可以被用来划分出一种特定的空间，使得期望的生活模式能够在这个空间中自然地产生。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，每一种模式都是在特定环境中的一种关系，是在那个环境中重复出现的一种特定的力学系统。存在一种特定的空间配置，允许系统中的这些力量（forces）

为自己求解，以相互支持，最终达到一种稳定的状态。

作为模式语言中的一个元素，模式代表的是一种指导，展示出在某种有意义的环境中，如何能够一次又一次地重复使用这种空间配置，来为特定的力学系统求解。

模式，简而言之，既是在世界上出现的一个事物，又是一种规则，这种规则告诉我们如何创建这个事物，以及在何时必须要创建它。模式既是一个过程也是一个事物；既是对一个活着的事物的描述，也是对生成这个事物的过程的描述[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的共同点更多。一种架构风格，作为一组相互协作的架构约束，应用于一个设计空间之上，以求促使系统出现所期待的架构属性。通过应用一种架构风格，架构师通过区分不同的软件设计空间，希望结果能够更好地匹配应用中所固有的一些必须满足的先决条件，这会导致系统的行为增强了自然的模式（natural pattern，译者注：可以与上文说的“自然地产生”关联起来），而不是与之相冲突（译者注：这段话非常符合老子“道法自然”的思想）。

1.7 视图（Views）

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（state and control approaches）、数据的表述（data representation）、事务生命周期（transaction life cycle）、安全保护（security safeguards）、峰值要求（peak demand）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]

观察一种架构，除了可从一个系统中的很多架构及组成架构的很多架构风格的角度来之外，还有可能从很多其他的角度来观察。Perry 和 Wolf [105] 描述了三种重要的软件架构视图：处理、数据、连接。处理视图侧重于流经组件的数据流，以及组件之间连接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。

使用多种架构视图，在特定架构的案例研究中是司空见惯的[9]。在“4+1 视图模型” [74] 这种架构设计方法学中，使用了 5 种相互协作的视图来组织对于软件架构的描述，每一种视图致力于解决一组特定的关注点。

1.8 相关工作

我在这里只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（architectural analysis techniques）、架构的恢复与再造（architecture recovery and re-engineering）、架构设计的工具和环境（tools and environments for architectural design）、从规范到实现的细化（architecture refinement from specification to implementation）、以及已部署软件架构的案例研究（case studies of deployed software architectures）[55]。有关架构风格的分类、分布式过程范例（distributed process paradigms）、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方法学（design methodologies）上面。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（pipe-and-filter，又称数据流 data flow）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出一种架构风格。

研究者们对架构分析和开发的方法学进行了一些初步的研究。Kazman 等人通过使用 SAAM[68] 的基于场景的分析（scenario-based analysis）和 ATAM[69] 的架构权衡分析（architectural trade-off analysis），对用于识别出架构方面（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设计（box-and-arrow designs）图，在每一种设计中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（architectural handbooks）的开发。面向对象编程社区率先开发了设计模式的目录，例子有“四人帮”的书籍（译者注：即著名的《设计模式》黑皮书）[51]、Coplien 和 Schmidt [33] 的文章。

软件设计模式与架构风格相比，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] 对基于对象开发（object-based development）中公共的架构模式进行了全面的检查。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（domain taxonomies）、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，用来作为软件架构的建造块（building block）；而架构风格则是一组运行特征（sets of operational characteristics），用来标识独立于应用领域的一个架构家族（architectural family）。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

研究者们开发出了各种参考模型，为描述架构和展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，OMA 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)一个包含了可重用领域专家知识的组件库。c) 一种应用的配置方法，用来在架构中选择和配置组件，以满足特殊的应用需求。Tracz [130]为 DSSA 作了综述（general overview）。

通过将软件开发空间（software development space）限制于一个满足某个领域需求的特定架构风格，DSSA 项目成功地将架构决策的关注点转移到了正在运行的系统（running systems）上面（译者注：而不是仅仅关注方框直线图） [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（a common architectural domain）组件的重用，而不是如何选择特定于每一个系统的架构风格。

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，ADL 是一种语言，用来明确说明软件系统的概念架构（conceptual architecture）和为这些概念架构建模。ADL 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）语言，它旨在成为一种通用的表述方法，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]是一种语言及相关的工具集的合称，用来将一组受限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）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（formal basis）。

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 是专门为单个架构风格而设计的，这样的 ADL 是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化的 ADL。ACME 所做的权衡是，它不支持特定于架构风格的分析和建造实际的应用，而是支持在不同的分析工具之间交换分析结果（interchange among analysis tools）。

1.8.5 形式化的架构模型

Abowd 等人[1]声称，根据很小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，使用明确说明的规则来控制在这些元素之上发生的反应。CHAM 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很多单一转换传播到整个系统中，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：假如某种架构形式的目的超出了仅仅转换一个数据流，如何使用 CHAM 来描述这种架构形式？

Rapide [78]是一种并行的、基于事件的模拟语言，专门为定义和模拟系统架构而设计。模拟器生成一组部分排序的（partially-ordered）事件，这些事件能够用于分析在内部连接（interconnection）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方式（formalism）。

1.9 小结

本章探讨了本论文的背景，引入并形式化了一组自恰的软件架构的概念术语。为了避免出现文献中常见的架构和架构描述之间的混淆，这些术语是必需的。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一些其他的研究。

后面两章将通过聚焦于基于网络应用的架构，来继续我们对于背景材料的讨论，并且描述如何使用架构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属性，这些架构属性是在将架构风格应用于基于网络的超媒体架构时所产生的。

第 2 章 基于网络应用的架构

本章通过聚焦于基于网络应用的架构，继续讨论上一章中提出的背景材料，并且描述如何使用架构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，在这里能够通过网络通信来实现组件之间的交互。我们将讨论限制在基于网络应用的架构风格之上，这样可以减少需要研究的架构风格之间的差异维度。

2.1.1 基于网络 vs. 分布式

通常来说，基于网络的架构（network-based architectures）与软件架构（software architectures）的主要区别是：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有效的机制）[128]。

Tanenbaum 和 van Renesse [127] 是这样来区分分布式系统（distributed systems）和基于网络系统（network-based systems）的：分布式系统在用户看来好像是普通的集中式系统（centralized system），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行（capable of operation across a network），但是这一点无需表达为对用户透明的方式。在某些情况下，甚至还希望用户知道：一个需要使用网络请求的动作，和一个在他们的本地系统就能满足的动作，两者之间有很大的差别，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，并没有局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限制在对应用软件的架构的讨论上，不包括操作系统、网络软件（networking software）和一些仅仅为得到系统支持而使用网络（only use a network for system support）的架构风格 (例如，进程控制风格 process control styles [53])。应用软件代表的是系统中能够“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（the goals of a user action）可以被表示为功能性的架构属性（functional architectural properties）。例如，一个超媒体应用必须关注信息页面的位置、执行请求和对数据流做呈现。这与网络抽象（a networking abstraction）形成了对比，网络抽象之目的是将比特从一个地点移动到另一个地点，而不关心为何要移动这些比特。只有在应用的层面上，我们才能够基于一些因素来评估设计上的权衡，这些因素包括每个用户动作的交互数量、应用状态的位置（the location of application state）、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通信的程度（the extent of communication）等等。

2.2 评估应用软件架构的设计

为选择或创建最适合于某一特定应用领域（application domain）的架构提供设计指导，这是本论文的一个目的。请记住架构是架构设计的实现，而并非架构设计本身。我们能够根据一个架构在运行时的特征来对其加以评估，但是很明显我们更希望能够在不得不实现所有的候选架构设计之前，有一种对这些架构设计进行评估的机制。不幸的是，众所周知，很难

通过一种客观的方式来对架构设计加以评估和比较。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经完成的工作，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置于系统之上的架构约束背后的设计基础理论，并且将因那些架构约束而产生的架构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运转，比较就无实际意义。虽然这样会将一些候选的架构设计排除在外，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性需求的架构设计。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系统必需的各种非功能性架构属性。因为架构属性是由架构约束产生的，所以有可能通过以下方式来评估和比较不同的架构设计：识别出每个架构的架构约束，评估每个架构约束所产生的一组架构属性，并将设计累积的架构属性与那些应用所要求的架构属性加以比较。

正如上一章中所描述的，一种架构风格是一组相互协作的架构约束，给它取一个名称是为了便于引用。每一个架构设计决策可以被看作是对一种架构风格的应用。因为添加的一个架构约束可能是从一种新的架构风格获得的，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（a derivation tree），这棵树的根节点是“空”风格（没有任何架构约束）。当多种架构的架构约束不存在冲突时，它们就可以进行组合，形成一种混合架构风格（a hybrid style），最终可以形成一种代表了架构设计完整抽象的混合架构风格。因此通过将一个架构设计的架构约束集合分解到一棵继承树上，就能够对这个架构设计进行分析，并且可以评估由这棵继承树所代表的架构约束的累积效果。如果我们理解了每种基本架构风格所产生的架构属性，那么遍历这棵继承树可以使我们理解架构设计的全部的架构属性，然后就能够将应用的特定需求与架构设计的架构属性进行匹配。这样比较不同的架构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期待的架构属性。

必须意识到，一种架构约束的效果可能会抵消一些其他架构约束带来的好处。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，然后使用这棵继承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。

在一棵架构风格的继承树中对架构属性所进行的评估，是特定于一个特殊应用领域之需求的，因为特定架构约束的影响常常取决于应用的特性。例如，当我们将管道和过滤器架构风格用于要求在组件之间进行数据转换的系统中时，会产生一些积极的架构属性；但是当系统仅仅由控制消息（control messages）组成时，它只会增加系统的开销（overhead），而不会带来任何好处。因为跨不同的应用领域对架构设计进行比较用处不大，因此确保一致性的最简单的方法是使这棵继承树特定于某个领域。

对架构设计做评估往往是一个在不同的权衡点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点的方法，给每个架构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标准度量手段。然而，为了让权重成为一个有意义的度量手段，必须使用一种对于所有架构属性一致的客观尺度来小心地选取每个权重。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，直到结果与他们的直觉匹配，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。这将会在下一章中进行演示。

2.3 关键关注点的架构属性

本节描述了多种架构属性，它们将用于对本论文中的架构风格进行区别和分类。我并没有想要列出一份全面的清单，而是根据我所调查的一组架构风格，只列出了那些明显受到这

些架构风格影响的架构属性。其他的架构属性，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）

聚焦于基于网络应用的架构风格的主要原因之一，是因为组件交互（component interactions）对于用户感知的性能（user-perceived performance）和网络效率（network efficiency）来说是一个决定性因素。由于架构风格影响到这些交互的特性，选择合适的架构风格能够决定基于网络应用部署的成败。

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

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的开销）在组件之间移交的速率。*开销*（overhead）可分为初始化开销（initial setup overhead）和每次交互（都会产生的）开销（per-interaction overhead），这种区分有助于识别出能够跨多次交互共享（*分摊* amortization）初始化开销的连接器。*带宽*（bandwidth）是在特定网络连接上可用的最大吞吐量。*可用带宽*（usable bandwidth）是指应用实际可用的那部分带宽。

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

2.3.1.2 用户感知的性能（User-perceived Performance）

用户感知的性能（user-perceived performance）与网络性能（network performance）的区别是，根据一个动作对于使用应用的用户的影响来度量这个动作的性能，而不是根据网络移动信息的速率来度量。用户感知的性能的主要度量手段是延迟（latency）和完成时间（completion time）。

延迟（latency）是指从最初的触发请求（initial stimulus）到得到最早的响应指示（the first indication of a response）之间持续的时间。延迟会发生在基于网络应用的处理过程中如下几个点上：1) 应用对于触发动作的事件的反应时间；2) 在组件之间建立交互所需的时间；3) 将交互请求数据传输到每个组件所需的时间；4) 在那些组件上处理每个交互请求所需的时间；以及 5) 应用能够对可用结果做呈现之前，完成移交和处理交互结果数据所需的时间。重要的是要注意到：虽然只有在第(3)点和第(5)点中存在着真正的网络通讯，但是架构风格对以上所有五点都会产生影响。此外，每个用户动作的多次组件交互也会增加延迟，除非它们能够并行发生（take place in parallel）。

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能够增量地处理正在接收数据的程度。例如，一个能够在接收数据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能显示图片的 Web 浏览器相比，前者的用户感知的性能会比后者好得多，即使两者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，如果所用算法在对数据进行编码之前，对其重要部分进行采样，那么对这段数据流的压缩就能够得到更高的编码效率。对于跨越网络移交已编码的数据来说，这样就会得到更短的完成时间。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，在移交之前缓存大型的采样数据会产生不可接受的延迟。在这些权衡点之间取得平衡是很困难的，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）

我们观察到，基于网络应用的一个有趣现象是：最佳的应用性能是通过不使用网络而获得的。这意味着对于基于网络应用来说，只有在可能的情况下有效地将对于网络的使用减到最少，才是最高效的架构风格。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）来减小某些交互的必要性。

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

2.3.2 可伸缩性（Scalability）

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

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

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

2.3.3 简单性（Simplicity）

通过架构风格产生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，那么理解和实现这些组件就会更加容易。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（the task of reasoning about the overall architecture）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架构属性中，

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）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作出改变的容易程度。可修改性能够被进一步分解为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作出改变时，无需停止和重新启动整个系统。

即使有可能建造一个与用户的需求完美匹配的软件系统，那些需求也会随时间而发生变化，就像社会的变化一样。因为基于网络应用中的组件可能会跨多个组织的边界来分布，系统必须准备好应对逐渐的和片段的改变（gradual and fragmented change），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自己的扩展功能。

2.3.4.1 可进化性（Evolvability）

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

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能够将功能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。产生可扩展性的方法是在一个架构中减少组件之间的耦合，就像基于事件的集成（event-based intergration）架构风格展示的那样。

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，随后该元素能够提供一种非常规的服务。如果一个组件的客户端能够扩展该组件的服务，而不会对该组件的其他客户端产生影响，则该组件就是可定制的 [50]。支持可定制性的架构风格也可能会提高简单性和可伸缩性，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，将会降低服务组件（service components）的尺寸和复杂性。可定制性是通过使用远程求值（remote evaluation）架构风格和按需代码（code-on-demand）架构风格而产生的一种架构属性。

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

2.3.4.5 可重用性（Reusability）

可重用性是应用架构的一个属性，如果一个应用架构中的组件、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。在架构风格中产生可重用性的主要机制是降低组件之间的耦合（一个组件知道其他组件的标识）和强制使用通用的组件接口。统一管道和过滤器架构风格中包括了这两种架构约束。

2.3.5 可见性（Visibility）

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

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]。能够产生可移植性架构属性的架构风格包括那些将代码和代码处理的数据放在一起移动的架构风格，例如虚拟机（virtual machine）架构风格和移动代理（mobile agent）架构风格；以及那些限制只能使用标准格式的数据元素的架构风格。

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来改善可靠性：避免单点故障、增加冗余、允许监视、以及将故障的范围缩小到一个可恢复的动作（reducing the scope of failure to a recoverable action）。

2.4 小结

通过聚焦于基于网络应用的架构，并且描述如何使用架构风格来指导这些架构的设计，本章对本论文所涉及到的范围进行了检查。本章还定义了架构属性的集合，在本论文的剩余部分中，将使用这些架构属性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应用的架构风格。当将架构风格应用于基于网络的超媒体系统的架构时，将会产生一系列架构属性。这个分类框架根据这些架构属性，来对每一种架构风格进行评估。

第 3 章 基于网络应用的架构风格

本章在一个分类框架中，调查了一些常见的基于网络应用的架构风格。当将架构风格应用于基于网络的超媒体系统的架构时，将会产生一系列架构属性。在这个分类框架中，我根据这些架构属性，对每种架构风格进行了评估。

3.1 分类方法学

建造软件的目的，是为了创造出满足或超出应用需求的系统，而不是为了创造出一种特殊的交互拓扑或者使用一种特殊的组件类型。为系统设计所选择的架构风格，必须与这些需求相一致，而不是相抵触。因此，应当基于这些架构风格所产生的架构属性来对架构风格进行分类，这样才能提供有用的设计指导。

3.1.1 选择哪些架构风格来进行分类

在这个分类框架中，我并没有打算全面包括所有可能的基于网络应用的架构风格。事实上，只要将一个架构约束添加到任意一种被调查的架构风格中，就能够形成一种新的架构风格。我的目标是描述一组有代表性的架构风格样本，特别是那些在软件架构文献中已经得到确认的架构风格（already identified within the software architecture literature），并且提供一个分类框架，使得其他架构风格一旦被开发出来，就能够添加到这个分类框架中。

我有意排除了一些架构风格，这些架构风格与被调查的架构风格结合后，并不能为（所形成的基于网络应用的）通信或交互方面的架构属性带来任何增强。例如，黑板风格（blackboard architectural style）[95]由一个中央仓库（central repository）和一组在这个仓库上机会性地执行操作（operate opportunistically upon the repository）的组件（知识源 knowledge sources）组成。通过将组件部署为分布式的组件（distributing the components），可以将黑板风格的架构扩展为基于网络的系统，但是这样一种扩展的架构属性完全取决于所选取的支持组件分布式部署的交互风格——通过基于事件的集成做通知（notifications via event-based integration）、按照客户-服务器方式做轮询（polling a la client-server）、或者仓库的复制（replication of the repository）。因此，尽管这种混合架构风格具备网络能力，但是将它包括到这个分类框架中却没有增加任何价值。

3.1.2 架构风格所产生的架构属性

在我的分类框架中，关注每种架构风格对架构属性带来的相对变化，并将其作为展示该架构风格应用于分布式超媒体系统时所产生的效果的方法。请注意，正如在 2.2 节中所描述的，针对某个特定属性对一种架构风格进行评估，依赖于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架构属性，或者在增强架构属性的某个方面的同时削弱架构属性的另一个方面。同样地，增强一个架构属性也可能会导致削弱另一个架构属性。

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，但是我们对于每种架构风格的评估是以它对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，我们才可以使用与系统设计者评估那些优点相同的方法，识别出一种架构风格超越其他架构风格的优点。由于我们不打算宣称某种架构风格对于所有的软件类型都是普遍适用的，因此通过限制我们评估的焦点，有效地减少

了需要评估的架构风格的数量。为其他应用类型评估架构风格，对于未来的研究而言，是一个开放的领域。

3.1.3 展示方式

这个分类框架的主要展示方式，是一张架构风格和架构属性的比较表。表中的数值表明了特定行中架构风格对于列中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（±）表示影响的性质取决于问题领域的某个方面。尽管对于下面各节中的细节而言，这只是一个粗略的简化，但它确实能够表明一种架构风格对架构属性所产生的正面（或负面）的影响。

另外一种展示方式，是使用一张基于架构属性的继承关系图，来对架构风格进行分类。根据各种架构风格之间的继承关系来分类，架构风格之间的弧线显示出得到或失去的架构属性。这张图的起点是“空”风格（没有约束）。这样一张图是有可能从对于架构风格的描述直接导出的。

3.2 数据流风格（Data-flow Styles）

表 3-1 评估数据流风格对基于网络超媒体系统的影响

风格	继承	网络性能	用户感知的性能	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，通常会对输入数据流应用一种转换并增量地处理它们，以便在完全处理完输入之前就能够开始输出 [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过滤器（零耦合）：它不能在其上行和下行数据流接口与其他过滤器共享状态、控制线程（control thread）或标识信息（identity）[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。图像处理方面的 *Khoros* 软件开发环境[112]是一个很好的例子，它可以使用管道和过滤器风格建造的一系列应用。

Garlan 和 Shaw [53] 描述了管道和过滤器风格的几个有利的架构属性。首先，PF 允许设计者将系统全部的输入/输出看作是个别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，只要允许数据在它们之间传输（可重用性）。第三，对 PF 系统做维护和增强很容易：能够将新的过滤器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 允许对系统做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（用户感知的性能）。

PF 风格的缺点有：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

输入，那么只能进行批量的顺序处理（batch sequential processing occurs），此时不允许进行任何交互。一个过滤器无法与其环境进行交互，因为它无法知道是哪个特定输出流与哪个特定输入流共享同一个控制器（shares a controller）。如果正在解决的问题不适合使用数据流的模式，这些架构属性会降低用户感知的性能。

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。通常在激活每个过滤器之前，一个由多个过滤器组成的网络就已经被安排好了。这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）。这部分控制器功能（controller function）被看作系统的一个独立的运行阶段（a separate operational phase），因此是一个独立的架构，即使这部分功能脱离了其他功能便无法独立存在。

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。这种架构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。理解一个特定的过滤器如何运转也更加容易了。

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一种特定的格式，这个约束可能会降低网络性能。

3.3 复制风格（Replication Styles）

表 3-2 评估复制风格对基于网络超媒体系统的影响

风格	继承	网络性能	用户感知的性能	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”。主要的例子有诸如 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 风格的主要优点是改善了用户感知的性能。实现途径是减少处理正常请求的延迟，并在主服务器（primary server）故障或有意脱离网络（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，$）

复制仓库风格是缓存风格的一种变体：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可能存在的数据集（potential data set）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（complete access to the repository）的地方。可以执行延迟复制（lazy replication）：当复制某个请求的尚未缓存的响应数据时，根据被引用数据所处的位置（locality of reference）和共同感兴趣的内容（commonality of interest），将有用的数据项复制到缓存中以备稍后重用。还可以执行主动复制（active replication）：根据预测的请求来预先获取可缓存的数据项。

与复制仓库风格相比，缓存风格对于用户感知的性能方面的改善不大，因为没有命中缓存的请求会更多，离线操作只能使用近期被访问过的数据。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。将缓存风格与客户-无状态-服务器风格（client- stateless-server style）结合后，就形成了一种基于网络的架构风格。

3.4 分层风格（Hierarchical Styles）

表 3-3 评估分层风格对基于网络超媒体系统的影响

风格	继承	网络性能	用户感知的性能	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）

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

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

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

客户-服务器的基本形式并不限制应用状态在客户组件和服务器组件之间如何划分。这常常由连接器实现所使用的机制来负责，例如，远程过程调用（remote procedure call）[23]或者面向消息的中间件（message-oriented middleware）[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”的架构风格，但是在基于网络的系统中，对分层系统的使用仅限于与客户-服务器风格相结合，形成分层-客户-服务器风格。

内部层对相邻外部层之外的所有层而言，是被隐藏起来的。通过这样做，分层系统减少

了跨越多层的耦合，从而改善了可进化性和可重用性。分层系统的例子包括分层通信协议的处理，例如，TCP/IP 和 OSI 协议栈 [138]，以及硬件接口库。分层系统的主要缺点是它们增加了处理数据的开销和延迟，降低了用户感知的性能[32]。

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

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

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

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

这些约束改善了可见性、可靠性和可伸缩性等三个架构属性。改善了可见性是因为监视系统再也不必为了确定请求的全部性质而查看多个请求的数据。改善了可靠性是因为这些约束使得从部分故障中恢复更加简单 [133]。改善了可伸缩性是因为不必保存多个请求之间的状态，允许服务器组件迅速释放资源并进一步简化其实现。

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

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

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

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。使用此架构风格的范例系统是互联网的域名系统（DNS）。

LC$SS 风格的优点和缺点是 LCS 风格和 C$SS 风格的优点和缺点的集合。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，它试图使客户组件（而非服务器组件）的复杂性最小化或者使它们的可重用性最大化。每个客户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：想要使用一个通用客户（generic client）（例如 TELNET [106]）或者通过一个模仿通用客户的接口（例如 FTP [107]）来访问远程服务。

远程会话风格的优点是：集中维护在服务器上的接口更加容易；当对功能进行扩展时，可以减少已部署客户中的不一致性问题；如果交互利用了服务器上扩展的会话上下文，还能够提高效率。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，也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和服务器上。客户以一种标准格式发送一个数据库查询（例如 SQL）请求给服务器，服务器分配一个工作空间并执行这个查询，这可能会产生一个巨大的结果集。客户能够在结果集上进行进一步操作（例如表连接）或者每次仅获取结果集的一部分。客户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：在服务器端能够通过多次迭代的方式，逐渐减小一个巨大的数据集，而不需要通过网络传输完整的数据集，从而改善了效率；通过使用标准的查询语言善了可见性。这种架构风格的缺点是：客户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（例如两阶段提交）来修正可靠性的问题，但其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）

移动代码风格使用了移动性（mobility），以便动态地改变处理过程与数据源或结果目的地之间的距离。Fuggetta 等人[50]全面地检查过这些架构风格。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），将其作为主动的配置（the active configuration）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一般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度（proximity）和它的交互的质量，减少交互开销从而提高效率和用户感知的性能。

在所有的移动代码风格中，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：将编码的尺寸（code's size）作为一个数据成员，与在正常的数据移交中所节省的部分作比较。如果在软件架构的定义中排除了数据元素，那么就不可能从架构的观点来建模了。

表 3-4 评估移动代码风格对基于网络超媒体系统的影响

风格	继承	网络性能	用户感知的性能	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架构风格，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，在一个特定平台上将指令（instruction）和实现（implementation）（可移植性）分离开，并且改善了可扩展性。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，因此会降低可见性。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，客户组件必须知道如何执行一个服务，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给客户。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，它对相同服务器的其他客户不会产生影响。

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；如果代码能够修改它的动作（adapt is actions），以适应服务器内部的环境（而不是客户端通过一系列交互来做同样的事情），这时能够得到更好的效率；由于需要管理求值的环境，简单性降低了，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，虽然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。最大的限制是，由于客户发送的是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，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，客户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请求，以获取如何处理资源的代码。接收到这些代码之后，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添加功能，改善了可扩展性和可配置性；如果代码能够修改它的动作，以适应客户端的环境，并在本地与用户交互而不必通过远程交互，这时能够得到更好的用户感知的性能和效率。由于需要管理求值环境，简单性降低了，但在一些情况下可以通过简化客户端的静态功能得到补偿。由于服务器将工作交给了客户（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，因此这样做并不会妨碍 LC$SS 风格的优点。此架构风格的一个范例是 HotJava Web 浏览器 [java.sun.com]，它允许将 applet 和协议扩展（protocol extensions）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。此架构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（the mobility works both ways）。

超越了那些已经在 REV 风格和 COD 风格中描述过的优点，移动代理风格的主要优点是：对于选择在何时移动代码而言，它具有更大的活力（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：即不必等待信息处理完）。此外，因为应用状态任何时刻都是在一个地点，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 评估点对点风格对基于网络超媒体系统的影响

风格	继承	网络性能	用户感知的性能	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。通过消除了解连接器接口的标识信息（identity on the connector interface）的必要性，它降低了组件之间的耦合。此架构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。在某个事件发布后，系统中的其他组件能够注册对于该事件类型的兴趣，由系统本身来调用所有已注册的组件[53]。此架构风格的范例包括：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]。

通过使得添加侦听事件的新组件更加容易（可扩展性）、鼓励使用通用的事件接口和集成机制（可重用性）、允许替换现有组件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]。对于以数据监视（data monitoring）为主，而不是以数据获取（data retrieval）为主的应用，EBI 通过使得轮询式交互（polling interactions）变得不再必要，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，这会立即导致与以下问题相关的可伸缩性问题：通知的数量、由通知引发其他组件广播而导致的事件风暴（event storms）、在通知传送系统中的单点故障。这些问题能够通过使用分层系统和事件过滤来改善，但却是以损害了简单性为代价的。

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 风格[128]直接支持大粒度的重用，并且通过加强基层独立性（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。这个约束加强了对高层依赖的松耦合（服务请求可以被忽略），并且与底层实现了零耦合（无需知道系统使用了通知），从而既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 风格并不对在通知中应该包括什么内容加以限制：标志、状态的 delta 改变量、或者完整的状态表述都是有可能的。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。在 C2 风格中，能够使用包括了监视能力的重量级连接器（heavyweight connectors）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。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操作或过程、以及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，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。

操作可以调用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，这有利于使状态尽可能保持最新，但是不利之处是难以获得系统活动的总体视图（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信息。当一个对象的标识信息改变时，必须修改所有显式调用它的其他对象[53]。因此必须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，因此该架构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会使用一种或更多种中间架构风格（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]。在被代理的分布式对象风格中，引入了名称解析组件——其目的是将该组件接收到的客户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的对象来处理客户请求。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感知的性能。

代理的分布式对象系统目前以两个规范为主导：OMG [97] 所开发的 CORBA 行业规范和 ISO/IEC [66] 所开发的开放分布式处理（ODP）国际规范。

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，它们最适合于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值得关注的问题。

3.7 局限性

每一种架构风格都在组件之间推崇一种特定的交互类型。当组件跨广域网（wide-area network）分布时，应用的可用性取决于对网络的使用或者误用。通过以架构风格对于架构属性的影响（尤其是对于分布式超媒体系统这类基于网络应用在性能上的影响）来刻画架构，我们才有能力选择出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

存在着一些局限性。

第一个局限性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；而且如果用户的交互是通信所必需的，管道和过滤器风格根本就不适用。同样地，如果对客户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。诸如此类的区别并没有出现在这个分类中，而只能在对于每种架构风格的讨论中进行非形式化的探讨。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、远程信息监视与搜索、远程控制系统、以及分布式处理。

第二个局限性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产生的一些特殊的方面，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架构风格而言。然而，将一个有很多抽象概念的架构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架构风格。一种解决方案就是在分类中既包括特殊的架构属性，也包括概括的架构属性。

尽管如此，这些最初的调查和分类，对于任何（可能解决这些局限性的）更进一步的分类而言，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对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]对这些架构风格进行了扩展。Shaw 和 Clements[122]提出了这些架构风格的初步分类，Bass 等人[9]重复了他们的工作，其中使用了以控件和数据（control and data issues）为坐标轴的二维的、表格化的分类策略，按照以下功能类别加以组织：在架构风格中使用哪种组件和连接器；在组件之间如何共享、分配和移交控件；数据如何通过系统来进行通信；数据和控件如何交互；何种类型的推理机制是与该架构风格相兼容的。这种分类方法的主要目的是标识架构风格的特征，而不是帮助对架构风格进行比较。它总结出了一组“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者提供一种有用的方法，以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，并不能够帮助设计者找到符合他们需要的架构风格。这样做也混淆了架构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，并且模糊了架构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，例如基于网络应用。最后，它无法描述出架构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、结构原则来对模式加以组织的分类方案（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，例如分离关注点和产生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架构模式，以及它们与软件架构的关系。

Zimmer[137]使用了一个以设计模式之间的关系为基础的图表，来对设计模式加以组织，

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

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

Fuggetta 等人 [50] 对移动代码范例进行了彻底的调查和分类。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架构风格进行了比较，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（a distributed system service）。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，以及相反的，特定架构的如何限制对于中间件的选择。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，对基于网络应用的常见架构风格进行了调查。当将架构风格应用于一种基于网络的超媒体系统的架构时，将会产生一系列架构属性。在这个分类框架中，我根据这些架构属性，对每种架构风格进行了评估。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领悟，推导出开发和评估一种架构风格的方法，用来对改进现代 Web 架构设计的任务提供指导。

表 3-6 全部评估的总结

风格	继承	网络性能	用户感知的性能	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	-	++	++	++	+±	++	+		+	+	±	+	+
MA	REV+COD		+	++		±		++	+	+		-	+	
EBI				+	--	±	+	+		+	+	-		-
C2	EBI+LCS		-	+		+	++	+		+	++	±	+	±
DO	CS+CS	-		+			+	+		+	+	-		-
BDO	DO+LCS	-	-				++	+		+	++	-	+	

第 4 章 设计 Web 架构：问题与领悟

本章介绍了 Web 架构的需求，以及在对 Web 的关键通信协议的改进提议（proposed improvements）做设计和评估的过程中遇到的问题。我使用在对基于网络的超媒体系统的架构风格进行调查和分类的过程中所获得的领悟，推导出了开发某种架构风格的方法，用来为改进现代 Web 架构的设计工作提供指导。

4.1 Web 应用领域的需求

Berners-Lee[20]写到：“Web 的主要目的，是旨在形成一种共享的信息空间（a shared information space），人类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而言都是可用的，并且能够引用和构造（reference and structure）由其他人保存的信息，而不必要求每个人都保持和维护一份本地的副本。

这个系统最初所期望的终端用户，是分布在世界各地、通过互联网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簿（organizational phone listings）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连接到这个项目（译者注：即 Web）时，可以进行增量的部署（incrementally deployable）。

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。这种策略被适用于 Web 架构的所有使用者：阅读者、创作者和应用开发者。

选择超媒体作为用户界面（user interface），是因为它的简单性和通用性：无论信息来源于何处，都能够使用相同的界面；超媒体关系（链接）的灵活性允许对其进行无限的构造（allows for unlimited structuring）；对于链接的直接操作允许在信息内部建立复杂的关系（allows the complex relationships within the information），来引导阅读者浏览整个应用。因为通过一个搜索接口（a search interface）访问大型数据库中的信息，常常比通过浏览方式（browsing）来访问更加容易，所以 Web 也包括了以下这种能力：通过将用户输入的数据提供给服务，然后在客户端呈现超媒体形式的结果，来执行简单的查询。

对于创作者来说，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互联网，都期望创作者能够以这种格式来保存个人研究笔记（personal research notes）之类的创作内容，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。我们鼓励创作者在开发信息源的过程中进行合作，因此无论引用是写在电子邮件中的说明（e-mail directions）还是在会议中临时写在餐巾纸的背面（written on the back of a napkin at a conference），这些引用都必须是容易表达的（references needed to be easy to communicate）。

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格式，

所以能够对通信进行观察，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（interactively tested using existing network tools）。这使得协议能够尽早地得到采用（early adoption of the protocols），尽管当时还缺少规范。

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达（the presentation of information）之中，或者作为信息的表达之上的一层。分布式超媒体允许在远程地点存储表达和控制信息（the presentation and control information）。由于这一特性，分布式超媒体系统中的用户动作（user actions within a distributed hypermedia system）需要将大量数据从其存储地移交到其使用地。这样，Web 架构必须被设计为支持大粒度的数据移交（large-grain data transfer）。

超媒体交互的可用性（the usability of hypermedia interaction）很容易影响用户感知的延迟（user-perceived latency，在选择一个链接和呈现可用结果之间的时间）。因为 Web 的信息源是跨越整个互联网分布的，这种架构必须尽量减少网络交互（在数据移交协议中的往返）的次数（minimize network interactions）。

4.1.4 互联网规模

Web 是旨在成为一个互联网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。互联网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有能力满足无法控制的可伸缩性和软件组件的独立部署两方面的要求。

4.1.4.1 无法控制的可伸缩性

大多数软件系统在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目标行动，而不是有着各自不同的目标。当系统在互联网上开放地运行时（runs openly on the Internet），无法安全地满足这样的假设。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，它们需要在遇到以下情况时仍然能够继续正常运行：遭遇未曾预测的负载（unanticipated load）、收到错误格式或恶意构造的数据（malformed or maliciously constructed data）。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适用于所有的架构元素。不能期望客户保持所有服务器的信息，也不能期望服务器跨多个请求保持状态信息。超媒体数据元素不能保持 “回退指针”（back-pointers，即引用该数据元素的某个数据元素的标识），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：如果有一千万人对此信息感兴趣，就需要保存一千万个引用）。特别是在有报道价值的信息能够导致“瞬间拥塞”（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峰值）的情况下。

架构元素的安全性和它们的运行平台也成为了重要的关注点。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。应用中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做一些额外的认证。这要求该架构能够表达认证数据和授权控制两种信息（be capable of communicating authentication data and authorization controls）。然而，因为认证过程降低了可伸缩性，架构的默认操作应该被限制在不需要可信数据的动作（actions that do not need trusted data）上，即一组具有良好定义的语义的安全操作（a safe set of operations with well-defined semantics）（译者注：这里所说的安全操作，含义是不会对服务器端造成危害的操作，因此不需要做认证）。

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改变（gradual and fragmented change），旧的组件实现将会与新的组件实现共存，而不会妨碍新的组件实现使用它们的扩展功能。现有的架构元素的设计需要考虑到以后将会添加新的架构功能。同样地，旧的实现必须易于识别（need to be easily identified），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影响。架构作为一个整体，必须被设计为其架构元素易于以一种部分的、迭代的方式（in a partial, iterative fashion）来部署，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。当一些个人开始对那些令他们着迷的话题发布信息时，这种社会化的网络效应使得网站数量以指数的速度增长，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（publish on an international scale）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，但互联网开发者社区（the Internet developer community）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，将会很快超越互联网基础设施的容量，并且导致全面的崩溃。Web 应用的交互性质也发生了变化，情况因此而更加恶化。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，导致出现了不同的浏览交互模式（interaction profile for browsing）。在对可扩展性、共享缓存、中间组件的支持等方面，已部署的架构存在着严重的局限，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与 Web 协议相矛盾的新的功能提议大量出现。

在互联网工程工作组（IETF）中形成了三个工作组，为 Web 的三个主要的规范而工作：URI、HTTP 和 HTML。这些工作组的主要任务是定义（在早期 Web 架构中被普遍地、一致地实现的）现有架构性通信的子集（define the subset of existing architectural communication），并且识别出在这个架构中存在的问题，然后指定一组规范来解决那些问题。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 解决之道（Approach）

早期的 Web 架构基于一些可靠的设计原则：分离关注点、简单性、通用性，但是缺乏对于架构的描述和基础理论。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（用于客户端和服务器的 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各种其他实现中。

可以使用一种架构风格来定义 Web 架构背后的设计原则，这样，这些设计原则对于未来的架构而言就是可见的了。如同在第 1 章中所讨论的那样，一种架构风格是一组已命名的架构元素之上的架构约束，由它产生了架构所期待的一组架构属性。因此，我所提出的解决之道的第一步，就是识别出一组存在于早期 Web 架构中的架构约束，这些约束负责产生出所期待的架构属性。

> **假设一**：能够通过一种由应用于 Web 架构中的元素之上的架构约束组成的架构风格，来描述在 Web 架构背后的设计基础理论。

为了扩展在架构实例上产生的架构属性，可以在一种架构风格上应用额外的架构约束。我的解决之道的下一步，是识别出在互联网规模的分布式超媒体系统中所期待的架构属性，然后选择额外的会产生那些架构属性的架构风格，将它们与早期的 Web 架构中的约束相结合，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 Web 架构风格添加约束，从而获得更好地反映一个现代 Web 架构所期待的架构属性的新的混合架构风格。

使用新的架构风格作为指导，我们能够对提出的对于 Web 架构的扩展和修改与架构风格中的架构约束进行比较。如果存在着冲突，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，每当新的功能被使用时，通过要求使用一个特定的指示（the use of a specific indicator），能够除掉存在的冲突。在影响一个响应的默认可缓存能力（cacheability）的 HTTP 扩展中，常常会这样做。对于严重的冲突，例如改变了交互风格，要么使用更加有益于 Web 架构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 并行运行的单独架构（译者注：即，不要基于 URI + HTTP 来实现）。

> **假设三**：修改 Web 架构的提议能够与更新后的 Web 架构风格进行比较，以便在部署之前分析出可能存在的冲突。

修订后的协议规范是根据新的架构风格的指导来编写的。最后，如同修订后的协议规范中定义的那样，通过参与到（建造大多数 Web 应用的）基础设施（infrastructure）和中间件软件（middleware software）的开发过程中，对更新后的 Web 架构进行部署。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、网景的 Navigator、Lynx、微软的 IE 三种浏览器、还有为很多其他实现的开发者提供建议而得到的间接经验，这些经验已经作为了 IETF 演讲（the IETF discourse）的一部分。

尽管我是以一个固定的顺序来描述这个解决之道的，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、通过客户和服务器软件的实验性扩展来测试这些约束对于 Web 协议规范的影响。同样地，其他人也曾建议为架构添加某些功能，这些功能超出了我所研究的当前模型架构风格的范围，但是并没有与该架构风格相冲突，这导致我回过头去修订架构的约束，以便更好地反映出改进后的架构。我的目标就是总是维持一个一致的、正确的模型，使用这个模型来表达我所希望的 Web

架构运转的方式（maintain a consistent and correct model of how I intend the Web architecture to behave），这样就能够使用它作为指导，定义适当行为（define appropriate behavior）的协议规范，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架构约束的人造模型（an artificial model）。

4.4 小结

本章提出了 Web 的架构需求，以及在对 Web 关键通信协议的改进提议进行设计和评估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提议的方法，使得能够在改进提议部署之前对它们进行评估。我所提出的解决之道是使用一种架构风格来定义和改进在 Web 架构背后的设计基础理论，使用架构风格作为严格的测试，在部署提议的扩展之前，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，然后再部署这些扩展。

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移交（REST）架构风格，开发出该架构风格是用来代表现代 Web 的运转方式（how the modern Web should work）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 表述性状态移交

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移交（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而来的一种混合架构风格，并且添加了一些额外的架构约束，用来定义统一的连接器接口。我使用了第 1 章中提出的软件架构框架来定义 REST 的架构元素，并检查其原型架构（prototypical architectures）的过程样本（sample process）、连接器和数据视图。

5.1 推导 REST

Web 架构背后的设计基础理论，可描述为由一组应用于架构中元素之上的架构约束组成的架构风格。当逐步将每个架构约束添加到正在进化中的架构风格时，会对架构风格产生一些影响。通过检查这些影响，我们就能够识别出 Web 的架构约束所产生的架构属性。然后就能够应用额外的架构约束来形成一种新的架构风格，这种架构风格能够更好地反映出现代 Web 架构所期待的架构属性。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 架构风格的各种特定的架构约束。

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者一切从零开始——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者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 是使用后一种观点开发出来的。图 5-1 至 5 -8 以图形化的方式，描绘了以下过程：随着架构约束逐渐被加入，已应用的架构约束如何将架构的过程视图区分开（how the applied constraints would differentiate the process view of an architecture）。

“空”风格（图 5-1）仅仅是一个空的架构约束集合。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

首先添加到我们的混合架构风格中的架构约束，是来自 3.4.1 小节描述的客户-服务器风格（图 5-2）。客户-服务器架构约束背后的原则是分离关注点。通过分离用户界面和数据存储

这两个关注点，我们改善了用户界面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使得组件可独立地进化，从而支持多个组织领域的互联网规规的需求。

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架构约束：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，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端的上下文，会话状态因此要全部保存在客户端。

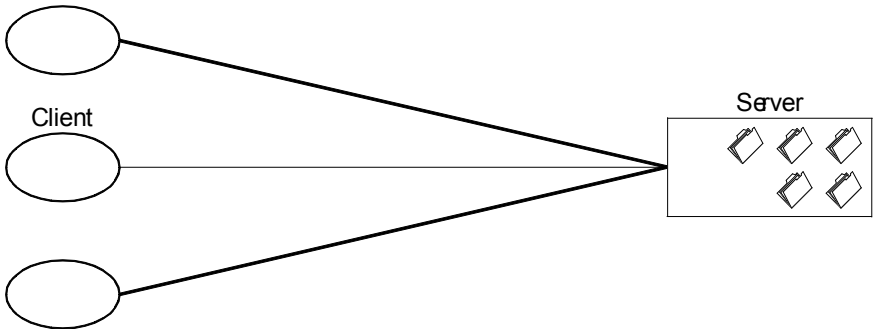

图 5-3 客户-无状态-服务器风格

这一约束产生了可见性、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情况。

与大多数架构抉择一样，无状态这一架构约束反映出设计上的权衡点。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能力，因为这样一来，应用就得依赖于跨多个客户端版本（semantics across multiple client versions）的语义的正确实现。

5.1.4 缓存

为了改善网络的效率，我们添加了缓存这个架构约束，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）。缓存架构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。

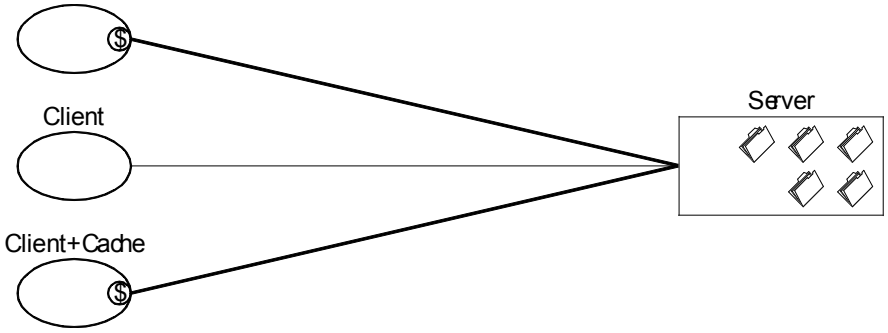

图 5-4 客户-缓存-无状态-服务器风格

添加缓存架构约束的好处在于，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、可伸缩性和用户感知的性能。然而，为此付出的代价是，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-服务器的架构约束集合来定义的。也就是说，1994 年之前的 Web 架构的设计基础理论聚焦于在互联网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（non-shared caches）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。相反地，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。

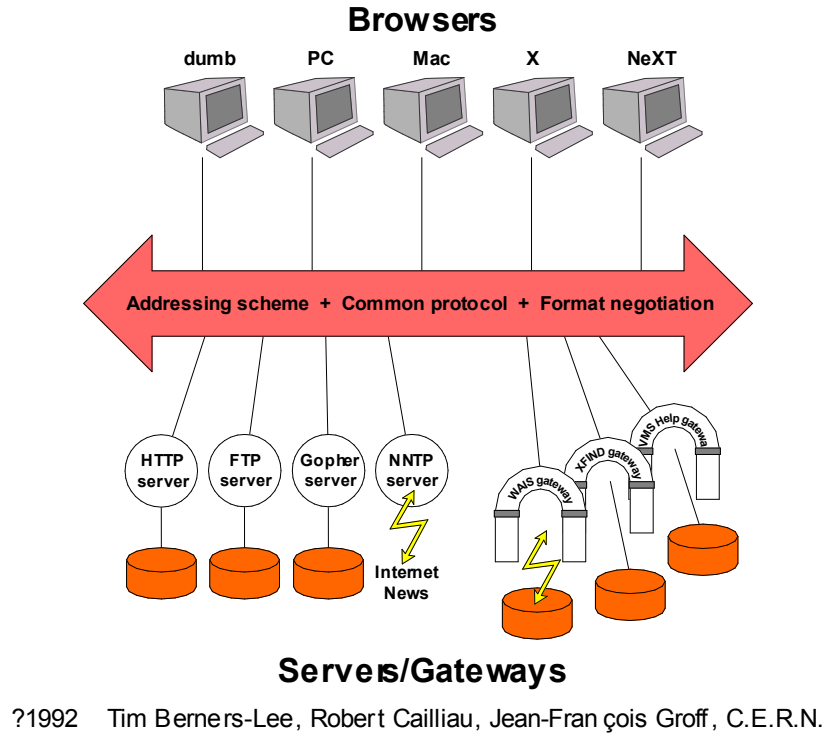

图 5-5 早期 Web 的架构图

Web 实现的开发者们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的响应，例如图像映射图（image-maps）[Kevin Hughes]和服务器端脚本（server-side scripts）[Rob McCool]。人们也以代理（proxies）[79]和共享缓存（shared caches）

[59]的形式开展了对中间组件的开发工作，但是必须对现有的协议进行扩展，这样中间组件才能够可靠地通信。以下几小节描述了添加到早期 Web 架构风格中的架构约束，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，简化了整体的系统架构，也改善了交互的可见性。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，需要付出的代价是，统一接口降低了效率，因为信息都使用标准化的形式来移交，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移交大粒度的超媒体数据，并针对 Web 的常见情况做了优化，但是这也导致该接口对于其他形式的架构交互而言并不是最优的。

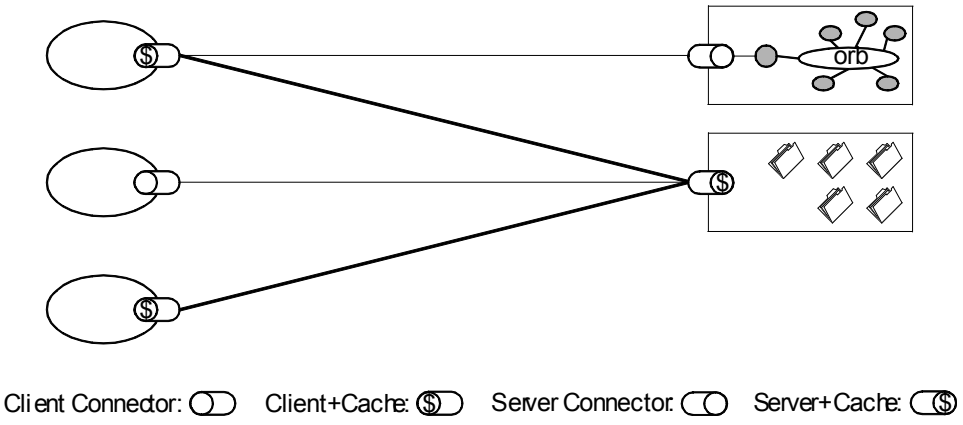

图 5-6 统一-客户-缓存-无状态-服务器风格

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 由四个接口架构约束来定义：资源的识别（identification of resources）、通过表述来操作资源（manipulation of resources through representations）、自描述的消息（self−descriptive messages）、以及作为应用状态引擎的超媒体（hypermedia as the engine of application state，译者注：简称 HATEOAS）。这些架构约束将在 5.2 节中讨论。

5.1.6 分层系统

为了进一步改善与互联网规模这个需求相关的行为，我们添加了分层系统架构约束（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”与其交互的相邻层），将架构分解为若干层级。通过将组件对系统的知识限制在单一层级内，为整个系统的复杂性设置了边界，并且提高了底层独立性。我们能够使用层级来封装遗留服务，使新的服务免受遗留客户端的影响，做法是将不常用功能转移到一个共享中间组件中，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。

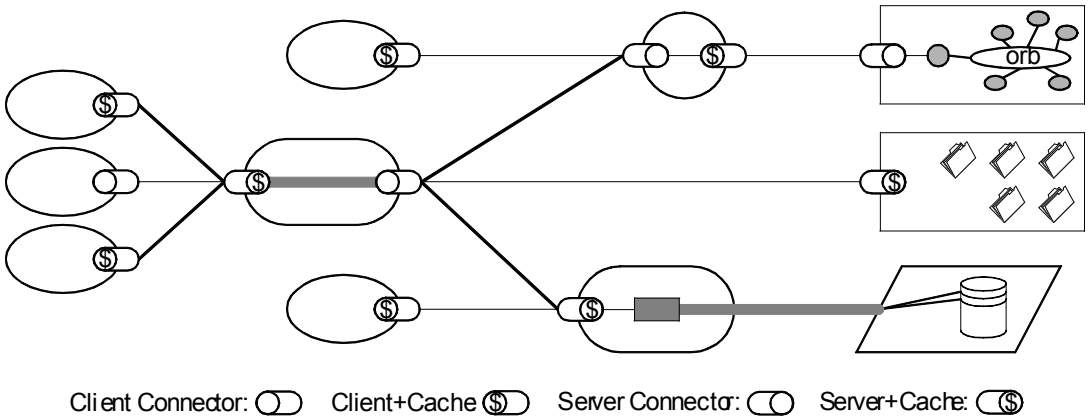

图 5-7 统一-分层-客户-缓存-无状态-服务器风格

分层系统的主要缺点是：增加了数据处理的开销和延迟，因此降低了用户感知的性能[32]。对于一个支持缓存架构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。通过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]。我们还能够通过这些中间层，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]。

分层系统架构约束和统一接口架构约束相结合，产生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数据流中的每一个，都可以被当作一个数据流网络（a data-flow network）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，它们在数据传递的过程中对其内容进行转换（transform the content as it passes）[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 添加的最后的架构约束来自于 3.5.3 小节中描述的按需代码风格（图 5-8）。REST 允许通过下载并执行 applet 形式或脚本形式的代码，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，这样做降低了可见性，因此它只是 REST 的一个可选的架构约束。

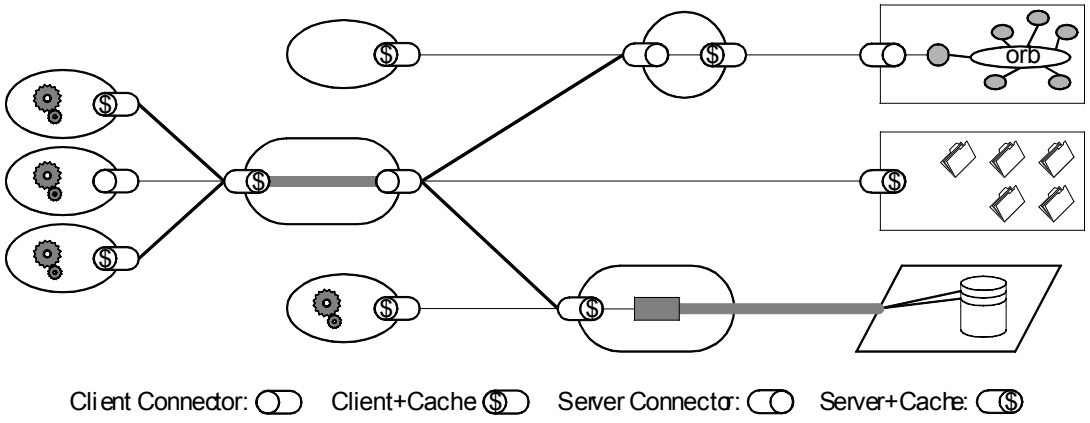

图 5-8 REST 风格

可选的架构约束这个想法，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，它确实是很有用的。这意味着只有当明确了可选的架构约束对于整个系统中某些领域（some realm of the overall system）有效的情况下，架构才会从可选的架构约束中得到好处（或受到损害）。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可构造为可以通过下载 Java 类来增强客户端的功能，以便从这一可选的架构约束中得到好处。然而，与此同时，该组织的防火墙可能会阻止移交来自外部资源的 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。一个可选的架构约束允许我们设计在一般场合下（in the general case）支持期待行为的架构，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 架构风格由一组经过选择的架构约束组成，通过这些架构约束在候选架构上产生所期待的架构属性。尽管能够独立考虑其中的每一个架构约束，但是根据它们在公共架构风格（common architectural styles）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础理论更加容易。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 架构风格的架构约束的来源。

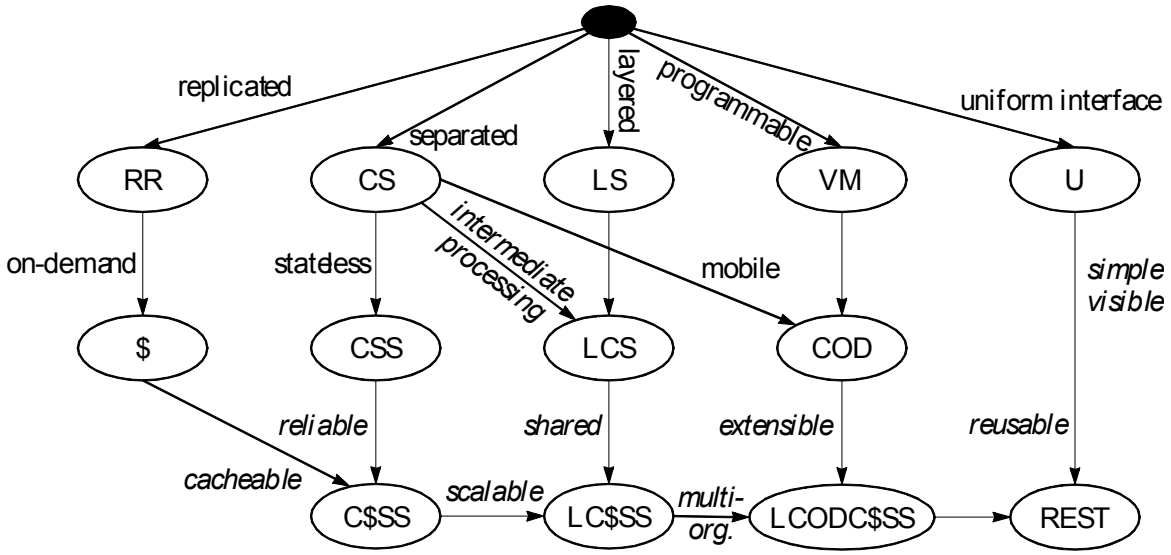

图 5-9 通过架构风格的架构约束推导出 REST

5.2 REST 架构的元素

表述性状态移交（REST）架构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 包括一组应用在组件、连接器和数据之上的基本架构约束，这些架构约束定义了 Web 架构的基础（define the basis of the Web architecture），因此它代表了基于网络应用的行为的本质（the essence of its behavior as a network-based application）。

5.2.1 数据元素（Data Elements）

在分布式对象风格 [31] 中，所有的数据都被封装和隐藏在数据的处理组件（processing components）之中。与分布式对象风格不同的是，REST 的关键方面之一是架构的数据元素的性质和状态。在分布式超媒体的特性中，能够看到这一设计的基础理论。当用户选择了一个链接后，该链接所指向的信息需要从它的存储地移动到其使用地。在大多数情况下，信息的用户是一个人类阅读者。这与很多其他的分布式处理模型 [6,50] 是不同的，在那些模型中，可能的做法，而且通常也是更高效的做法，是将“处理代理”（processing agent）（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程序的所在地。

对于一个分布式超媒体系统的架构师来说，他只能在三种基本选项中作出选择：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镜像（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]，它使得与数据的自然特性（the true nature of the data）有关的所有信息都被隐藏在数据发送者之中，防止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，这导致了可伸缩性方面的问题。第二种选项对应于可移动对象（mobile object）风格 [50]，它支持对于信息的隐藏，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移交的数据量。第三种选项允许发送者保持简

单性和可伸缩性，同时还使得需要移交的数据最小化。但是，它丧失了信息隐藏的优点，并且要求发送者和接收者都必须理解相同的数据类型。

REST 聚焦于分享对于（包含元数据的）数据类型的理解，但是对于作为标准化接口暴露的操作的范围进行了限制。通过这种做法，REST 提供的是所有三种选项的一个混合体。REST 组件之间通过以一种数据格式移交资源的表述来进行通信，可以基于接收者的能力和所期待的内容、以及资源的性质，在一组进化中的标准数据类型（an evolving set of standard data types）之中动态选择所使用的数据格式。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。通过发送一个表述，可以获得近似于可移动对象风格的好处，这个表述由一个封装过的呈现引擎（an encapsulated rendering engine）（例如，Java [45]）的标准数据格式中的指令组成（译者注：现在更常见的是 JavaScript 脚本）。REST 因此获得了客户-服务器风格的分离关注点的好处，而且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 的数据元素总结见表 5-1。

表 5-1 REST 的数据元素

数据元素	现代 Web 实例
资源	一个超文本引用所指向的概念性目标（conceptual target）
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	source link、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，可能作为创作者的超文本引用的目标（the target of an author’s hypertext reference）的任何概念都必须符合资源的定义。资源是到一组实体的概念性映射（a conceptual mapping），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 $M_R(t)$，该函数根据时间 t 将资源映射到一个实体或值的集合，集合中的值可能是*资源表述*（resource representations）和/或*资源标识符*（resource identifiers）（两者是等价的）。一个资源可以映射到空的集合，这使得可以在一个概念的任何实现出现之前引用这个概念——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]。在创建一组资源之后，在任何时刻来检查这些资源，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。而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的首选版本”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“包含在 Orange 发布版本中的修订版”。

正是资源的这个抽象定义，使得 Web 架构的核心功能得以实现。首先，它包容了很多信息的来源，并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（based on characteristics of the request）来进行内容协商。最后，它允许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*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，而无须关心其成员函数（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，映射的语义有效性也由同样的命名权威来负责维护（例如，确保成员函数不会改变）。

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完全无法满足 Web 的超大规模和跨越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资金成正比，因此随着暂时性的（或支持不良的）信息的移动或消失，会导致出现破损的链接。

5.2.1.2 表述（Representations）

REST 组件使用表述来捕获某个资源的当前状态或预期状态，随后在组件之间移交该表述，通过这种方式在资源上执行各种动作（perform actions on a resource）。*表述*（representation）由一个字节序列和描述这些字节的*表述元数据*（representation metadata）构成。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，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（control data）定义了在组件之间移交的消息的用途，例如被请求的动作或响应的含义。它也可用于提供请求的参数，或覆盖某些连接元素（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。

一个特定表述的具体含义取决于消息中的控制数据，该表述表示的可能是被请求资源当前状态或预期状态，或者某个其他资源的值，例如在一个客户端查询表单中的输入数据，或在一个响应中的某种出错状况。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端，这样就为该资源赋予了一个值，以后的请求可以获取这个值。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*媒体类型*（media type）[48]。发送者能够将一个表述包含在一个消息中，移交给接收者。接收者在接收到消息之后，根据消息中的控制数据和媒体类型的性质（the nature of the media type），来对该消息进行处理。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查看的，还有少数是可以同时用于两种用途的。还可以使用组合的媒体类型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感知的性能。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。假如一种数据格式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，这样的数据格式将改善用户感知的性能，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感知的性能，与那些在呈现之前要等待整个文档完全接收的浏览器相比会好得多。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。如果呈现动态尺寸的表格及其内嵌对象（dynamically-sized tables and embedded objects）的所需的屏幕尺寸必须要在呈现它们之前确定，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（译者注：需要将内嵌对象的数据全部加载完，才能够确定其大小）。

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移交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，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变得可能：如果用户仅能通过一个抽象的接口来访问系统，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（efficiency and responsiveness）。

表 5-2 REST 的连接器

连接器	现代互联网实例
客户	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 CONNECT 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。这一限制得到四个功能：1）它使得连接器无须在请求之间保持应用的状态，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，处理机制无须理解交互的语义；3）它允许中间组件孤立地查看和理解一个请求，当对服务进行动态安排时（when services are dynamically rearranged），这是必需的；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（might factor into the reusability of a cached response）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。其输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。其输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。从抽象的观点来看，调用是同步的，但是输入参数和输出参数都可以作为数据流来传递。换句话说，对于请求的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，从而避免了对于移交大型数据进行批量处理而产生的延迟。

主要的连接器类型是客户和服务器。两者之间的本质区别是：客户（client）通过发送请求来发起通信；服务器（server）侦听连接并对请求作出响应，以便为其所暴露的服务提供访

问途径。一个组件可能包括客户和服务器两种连接器。

第三种连接器类型是*缓存*（cache）连接器，它可以位于客户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重用。客户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。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以外的其他客户所使用。共享缓存可以有效地减少热门服务器（a popular server）的负载出现“瞬间拥塞”的几率，尤其是当缓存以分层的形式来安排，以便覆盖大量用户的时候，例如，那些使用由公司内联网（intranet）、互联网服务提供商的顾客、或共享国家网络主干（sharing a national network backbone）的大学所提供的互联网接入服务的用户。然而，假如被缓存的响应与新请求原本应该获得的响应并不匹配，共享缓存就会导致应用出错。REST 试图在透明的缓存行为和高效地使用网络这两个所期待的架构属性之间取得平衡，而不是假设无论何时都需要绝对的透明性。

缓存连接器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。如果在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。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。

解析器（resolver）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，将其作为标识该资源的命名权威的机制。为了发起一个请求，Web 浏览器会从 URI 中提取出主机名，并利用 DNS 解析器来获得该命名权威的互联网协议（IP）地址。另一个例子是某些识别模式（例如 URN[124]）要求中间组件将永久的标识符翻译为一个更为短暂的地址，以便访问被标识的资源。通过增加中介（indirection）的方式使用一个或多个中间解析器（intermediate resolvers），能够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*（tunnel），它简单地跨越连接的边界（例如一个防火墙或更低层的网关）对通信进行中继。将隧道作为 REST 的一部分来建模，而不是做为网络基础设施的一部分，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隧道行为（switch from active component behavior to that of a tunnel）。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 组件可根据它们在整个的应用动作（an overall 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

用户代理（user agent）使用客户连接器发起请求，并作为响应的最终接收者。最常见的例子是 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。

来源服务器（origin server）使用服务器连接器管理被请求资源的命名空间。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对资源的值进行修改的请求的最终接收者。每个来源服务器都以资源层次结构的形式，为其所暴露的服务提供了一个通用的接口。资源的实现细节被隐藏在这一接口的背后。

中间组件同时扮演了客户和服务器两种角色，以便支持请求和响应的转发（forward），可能还会对请求和响应进行翻译。代理（proxy）组件是由客户选择的中间组件，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。网关（gateway）（也叫作反向代理 reverse proxy）组件是由网络或来源服务器强加的中间组件，用来为其他的服务提供接口封装，以执行数据转换、性能增强，或安全增强（security enforcement）。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（a REST-based architecture）的过程视图，其中包括了对三个并行请求的处理。

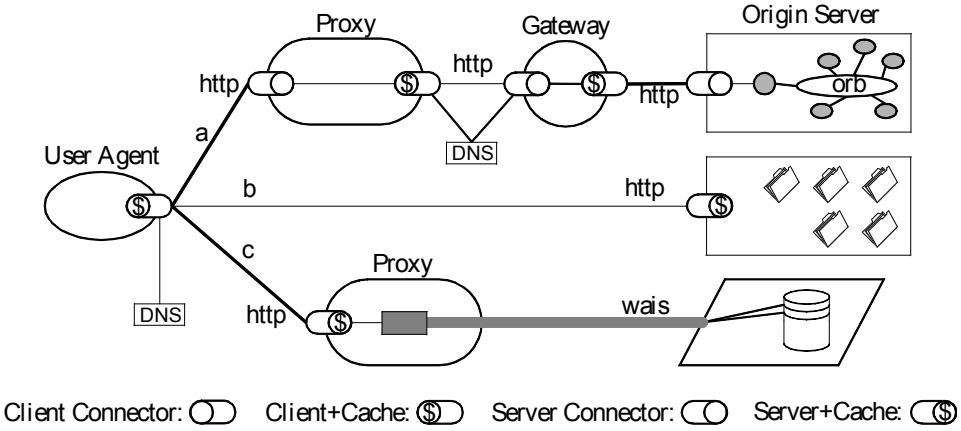

图 5-10 一个基于 REST 的架构的过程视图

一个用户代理正在处理三个并行的交互（a、b 和 c）。用户代理的客户连接器的缓存无法满足请求，因此它根据每个资源标识符的属性和客户连接器的配置，将每个请求路由到资源的来源服务器。请求(a)被发送到一个本地代理，该代理随后访问一个通过 DNS 查找发现的缓存网关（a caching gateway），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，服务器内部的资源是由一个封装过的对象请求代理（object request broker）架构来定义的。请求(b)被直接发送到一个来源服务器，该服务器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 架构相分离的信息服务），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。每个组件只知道与它们自己的客户或服务器连接器的交互；而整个过程的拓扑结构正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。分层系统的架构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，从而允许使用它们来辅助通信的翻译（communication translation），或通过大规模的、共享的缓存来改善性能。REST 通过强制要求消息具有自描述性来支持中间组件的处理，其具体体现为：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、以及响应可以明确地表明其可缓存性。

由于组件之间是动态连接的，因此对于一个特定应用动作的安排和功能（arrangement and function）而言，其性质与管道和过滤器风格是类似的。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却是独立的，因此容易受到数据流转换器（stream transducers）（即过滤器）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（因为这对于一个互联网规模的架构来说是不可能完成的任务），并且允许组件根据每个请求的目标来动态决定自己的角色，要么作为目的地、要么作为中间组件。连接器只需要在它们的通信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（cache the existence and capabilities of other components）。

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。对于一个基于 REST 的架构而言，我们对定义通用资源接口的架构约束尤其感兴趣。

客户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，我们可以对一个客户连接器这样配置，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，也可以配置客户连接器使其拒绝对于标

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，但是它会对组件之间的接口作出约束，因此也会对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）作出约束。例如，Web 的主要移交协议（the Web’s primary transfer protocol）是 HTTP，但是 REST 架构也包括了对来源于其他协议的网络服务器的资源（包括 FTP [107]、Gopher [7] 和 WAIS [36]）的无缝访问，这些网络服务器在 Web 出现之前就已存在了。尽管如此，与其他协议的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所具有的单一的、通用的接口的好处，这一架构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（the stateful interaction）的好处。作为回报，使用通用的接口使得通过单个代理访问多个服务成为了可能。如果一个应用需要获得另一架构的额外能力，它可以将该架构作为一个与 Web 单独并行运行的系统来实现，并在 Web 中调用那些能力，这与 Web 架构使用“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它们想要完成的任务。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。

组件之间的交互是以移交动态尺寸的消息（dynamically sized messages）的形式来进行的。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。最常见的请求语义的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）都集中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。因此，应用的状态由以下几方面来定义：悬而未决的请求（pending requests）、相连接组件（有些组件可能是用来过滤被缓冲的数据）的拓扑结构、连接器上活跃的请求（the active requests on those connectors）、在请求响应中的表述数据流、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，这种稳定状态与一个“网页”（包括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）相对应。应用的稳定状态之所以很重要，是因为它会同时影响用户感知的性能和网络请求流量的峰值。

浏览器应用的用户感知的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 的架构主要通过移交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的可用性（the availability of layout information）（内嵌对象的视觉尺寸）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

在只有较小请求命中了缓存的情况下也是如此。

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”（“respond first and think later”）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，假如第一个响应无法满足客户端的需要，再提供一组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的状态，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，这个做法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，链接映射图和表述的预先获取，link maps and prefetching of representations）、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，这与超媒体浏览器的用户界面完全匹配。然而，REST 风格并未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，还是查找与特定查询标准相匹配的数据的个人代理，或者是忙于巡视破损的引用或被修改的内容的、执行维护任务的爬虫应用。

5.4 相关工作

Bass 等人 [9] 在其文献中用了一章篇幅来介绍 Web 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 架构的设计和基础理论的人们开发出来的，尽管它们反映出了很多 REST 的设计约束，但是真正的架构是独立于任何单一实现的。现代 Web 是由它的规范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 架构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 中的组件交互被结构化为一种分层的客户-服务器风格，但是额外添加的通用的资源接口的架构约束，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性的资源，而不是某个实现的标识符。

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”，而实际上排除这些内容会使得这些模型之下的假设变得无效。REST 运转得很好，是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为匹配的实现，并且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）在状态改变时会产生一个事件，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 架构风格中，消费组件通常需要自己去拉表述。尽管当单个客户希望监视单个资源时，这种方式的效率会低一些（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 架构风格，这种方法与 C2 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 风格缺乏在 REST 中对中间组件友好的那些架构约束，例如通用的资源接口、确保无状态的交互，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移交（REST）架构风格。REST 提供了一组架构约束，当将其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 的软件工程原则和为支持这些原则而选择的交互约束，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习到的经验和教训，提供了对于 REST 架构的评估。这些工作包括创作超文本移交协议（HTTP/1.1）和统一资源标识符（URI）的这两个当前的互联网标准跟踪规范（the current Internet standards-track specifications），以及通过 libwww-perl 客户端协议库（client protocol library）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移交协议（HTTP）和统一资源标识符（URI）两个互联网规范（这两个规范定义了在 Web 上进行交互的所有组件所使用的通用接口）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、协议标准的其他实现的过程中，应用 REST 所学习到的经验和教训。

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 的运转方式创建一种架构模型，使之成为 Web 协议标准的指导框架。REST 被用来描述期待的 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 成功的那些核心架构约束（the core constraints）。这项工作是作为互联网工程工作组（IETF）和万维网协会（W3C）定义 Web 架构的规范（define the architectural standards）（即 HTTP、URI 和 HTML）工作的一部分来完成的。

我参与 Web 规范的创作过程开始于 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 的架构由以下三部分内容来描述：一组非形式化的超文本笔记（a set of informal hypertext notes）[14]、两篇早期的介绍性论文 [12,13]、展现出一些提议的 Web 功能的超文本说明草稿（draft hypertext specifications representing proposed features for the Web）（一些功能已经被实现了），分布于世界各地的 Web 项目的参与者们使用公开的 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] 诞生之后 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与来源 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，并不知道存在有缓存、代理或网络爬虫，甚至是在它们的实现已经存在并且非常流行（running amok）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] 创建了 W3C，将其作为 Web 架构的智库，并且为编写 Web 规范并开发其参考实现提供所需的创作资源，但是标准化本身是由互联网工程工作组 [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 规范的主要的架构师，并且最终创作了成为 URI 通用语法标准（the standard on URI generic syntax）[21] 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，在我编写 HTTP/1.0 规范和最初的 HTTP/1.1 建议（the initial HTTP/1.1 proposal）时，将它用来作为表达各种 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“表述性状态移交”是有意唤起人们对于一个设计良好的 Web 应用如何运转（how a well-designed Web application behaves）的印象：Web 应用是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（状态迁移）

在应用中前进，导致系统将下一个页面（代表应用的下一个状态）的数据移交给用户，并且为他们呈现出来，以便他们来使用。

REST 并未想要捕获到 Web 协议规范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 确实完全捕获了一个分布式超媒体系统中最重要的方面，即那些被认为是 Web 的行为需求和性能需求的核心部分（that are considered central to the behavioral and performance requirements of the Web）的方面。因此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 架构上的那些架构约束同样也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 架构中最简单的元素，也是最重要的元素。URI 还有很多的名称：Web 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。Web 地址的规范也定义了我们所说的“资源”这个概念的范围和语义，然而资源这个概念从早期的 Web 架构以来发生了变化。REST 既被用来为 URI 规范 [21] 定义“资源”这个术语，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根据网络上一个文档的位置来定义标识符。然后就能够使用 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所移交的内容，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的是一个服务，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，而不是访问该服务而获取到的某个特定结果。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命名，而不是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（the Web uses embedded identifiers rather than link servers），创作者需要一个标识符，这个标识符能够与他们想要通过一个超媒体引用来表达的语义紧密地匹配，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 通过以下做法达到了这个目标：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所对应的值。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”定义为使用一个 URI 来标识的一个概念，而不是标识的一个文档，这给我们带来了另一个问题：一个用户如何访问、操作或移交一个概念，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 通过以下方式回答了这个问题：定义在被标识资源的“表述”之上执行的操作，而不是定义在资源本身之上执行的操作。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移交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。强制要求接口的定义与接口的需求相匹配（forcing the interface definitions to match the interface requirements），会使得协议看起来含糊不清，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资源的底层实现到表述的映射。

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操作资源的表述，而不是直接访问资源的实现，因此资源的实现可以以任何其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分离开的。一个单独的服务器可以将它的命名空间的一部分映射到一个文件系统，接下来将文件系统映射到一个 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的是一种将一个资源与一组表述相关联的方法，而不是识别资源本身的方法。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。

为了对一个现有资源进行创作，创作者必须首先获得特定的来源资源（source resource）URI：绑定到目标资源（target resource）处理器的底层表述（the handler's underlying representation）的那一组 URI。一个资源并不总是会映射到单个静态文件，所有非静态的资源会来自于某些其他资源，通过跟踪继承树，创作者能够最终找到所有必须编辑的资源，以便修改资源的表述。这些相同的原则适用于以任何形式继承而来的表述，无论它是来自何种机制，例如，内容协商、脚本程序（scripts）、servlet、托管的配置（managed configurations）、翻译（versioning）等等。

资源不是存储对象（storage object）。资源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规则实现（current mapping implementation，通常是与特定集合相关的树的深度遍历 和/或 哈希表查找）上，以发现当前负责处理该资源的处理器实现（handler implementation），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被隐藏在 Web 接口之后，仅能够通过 Web 接口访问资源的客户端无法对它们（译者注：即，特定于实现的问题）的真实性质作出假设。

例如，考虑在以下场景中将要发生的事情。随着一个网站的用户量的增长，开发者决定将旧的基于 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，新服务器上的命名空间可以与原先老服务器的命名空间保持完全的镜象，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。也可以有两个 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——它们仅仅扮演了一个管道，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，在服务器上其实没有资源，而是有一种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（译者注：资源是抽象的概念，而不是具体的实现）。

按照（用来组成已完成产品的）组件的特性来定义事物（define things in terms of the characteristics of the components），是每一个工程师的天性。Web 却并非是以这种方式来运转的（the Web doesn't work that way）。基于每个组件在一个应用动作期间的角色，Web 架构由在组件之间的通信模型（the communication model between components）之上的架构约束组成。这防止了组件对于每件事物作出超越资源抽象的假设，因此隐藏了位于抽象接口任何一端的真实实现机制。

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 架构组件都服从 Web 架构设计中给出的每一个架构约束。REST 既可以被用作定义架构改进的方法，也可以被用作识别架构不匹配的方法。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，就会发生架构不匹配。尽管往往无法避免架构不匹配，但是有可能在它们成为正式规范之前识别出它们。

尽管 URI 的设计与 REST 的标识符这个架构概念相匹配，单单依靠语法却不足以迫使命名权威们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超媒体响应形式的表述（a hypermedia response representation）（译者注：通常就是 HTML）所引用的所有的 URI 中包括标识当前用户的信息。这样内嵌的用户标识可用于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 的架构约束，这些系统会降低共享缓存的效率，也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得到不希望的结果。

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其信息的实现，有工具能够跨越多个站点对这些信息做镜象，从而可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与一个文件系统的语义相匹配，而且资源的表述中同时包括有数据和元数据（对于一个表述的语义而言，这是非常重要的）。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已知其表述确实为静态的资源（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互联网的边缘，以降低延迟和减轻来源服务器的负担）。

6.3 将 REST 应用于 HTTP

超文本移交协议（HTTP）在 Web 架构中有一个特殊的角色，它既作为在 Web 组件之间通信的主要的应用级协议，也作为特别为移交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增强提议很保守，因此在对 HTTP 的扩展进行部署之前，需要对扩展加以证实，并且会受到对于规范的评论的影响。我们使用 REST 来识别出现有的 HTTP 实现中的问题，指定了一个能够与 HTTP/1.0 协议 [19] 互操作的协议子集，分析 HTTP/1.1 [42] 的扩展提议，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，并且预测持久连接和内容协商这些扩展对 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 的应用也能够同样地对规范产生影响。

6.3.1 可扩展性

REST 的主要目标之一是，支持以通过逐渐和片段的方式对一个已经部署架构（译者注：即老版本的 HTTP 协议实现）中的改变进行部署（support the gradual and fragmented deployment of changes within an already deployed architecture）。我们对 HTTP 协议做了修改，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 命名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在每一个消息[90]中的 HTTP-version 协议元素之上的架构约束。

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精简过的（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。在一个请求/响应链条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，而不必考虑作为这个链条一部分的某些客户端或服务器的限制。

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的次版本，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备功能（required features）（译者注：协议的功能分为必备功能和可选功能两类，必备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）的修改，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变更的 HTTP 功能，仅限于那些在通信中的被其直接邻居解释的功能，因为 HTTP 并不要求整个请求/响应链条中的所有中间组件都使用相同的版本。

这些规则的存在为同时部署多个协议修订版提供了支持，并且防止了 HTTP 协议的架构师们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修改和不兼容的修改很容易区别来实现的。兼容的修改将很容易部署，对于协议接受能力差异的交互能够在协议流（protocol stream）中完成。而不兼容的修改将难以部署，因为它们在协议流开始工作之前，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 包括了很多单独的命名空间，每一个都有不同的架构约束，但是不能对可扩展性设置限制（being extensible without bound）是所有命名空间的共同需求。一些命名空间由单独的互联网标准来管理，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]、字符集的值（charset

values）、语言标签（language tags）。而其他的命名空间则由 HTTP 来管理，包括：方法名称、响应状态码、非 MIME 头信息字段名（non-MIME header field names）、以及标准的 HTTP 头信息字段中的值。既然早期的 HTTP 没有为如何部署在这些命名空间中的改变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：如果一个接收者需要提前知道一个方法的语义，然后才能通过一个中间组件安全地转发该方法（if a recipient needs to know the semantics of a method before it can be safely forwarded by an intermediary），那么必须更新所有的中间组件，才能部署新的方法（译者注：从上下文理解，绝大多数中间组件，对于所转发的请求是没有记忆的，因此无法支持上述的 HEAD 方法的部署）。

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，在其中条件性请求头信息字段（conditional request header fields）具有刷新缓存的语义，而对于所有其他的方法，条件性请求头信息字段的含义是需要满足的一个前提条件（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，这样就可以部署新的响应而不至于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暂时性的信息响应（a provisional information response），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试（译者注：即重复发送相同的请求）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通过重试（或许是通过某个其他服务器）得到一个更好的响应。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修改。修改因此能够被部署在一个现有架构之上，而无须担心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，通过允许客户端在旧的协议流中表达它希望使用另一个更好的协议，从而减少了部署不兼容修改的难度。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过的协议，并且希望进行切换，它简单地以一个 101 状态码作为响应并且继续通信，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

REST 要求组件之间消息是自描述的，以便支持中间组件对于交互的处理。然而，早期 HTTP 协议的一些方面并不是自描述的，其中包括在请求中缺乏主机标识，无法依照语法来区分消息控制数据和表述元数据（message control data and representation metadata），无法区分仅仅由直接连接的双方（the immediate connection peer）使用的控制数据和所有接收者都使用的元数据（译者注：在一个请求/响应链条中，可能会有多个连接，接收者包括位于直接连接

另一端的接收者和相隔多个连接的接收者），而对于强制性扩展（mandatory extensions）以及通过元数据来描述带有分层编码的表述（representations with layered encodings）也缺乏支持。

6.3.2.1 Host（主机）

早期 HTTP 设计中一个最糟糕的错误决定是，不发送表示请求消息目标的完整的 URI，而仅发送那些不用于建立连接的部分。其所做的假设是：一个服务器将会基于连接的 IP 地址和 TCP 端口得到它自己的命名权威。然而，这个假设没有预测到多个命名权威可能会存在于单个服务器上，随着 Web 和域名（http 的 URL 命名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。部署这一功能非常重要，以至于 HTTP/1.1 规范要求服务器拒绝任何没有包括 Host 字段的 HTTP/1.1 请求。结果，现在有很多大型的 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 分层编码

HTTP 为了描述表述的元数据，继承了多用途互联网邮件扩展（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。早期的一个 HTTP 扩展解决了这个问题，它在 Content-Encoding 字段中分别列出外层媒体类型（可能会有多个）的编码，并且将最内层媒体类型的标签放在 Content-Type 中。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造成混淆。

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，上述第一个扩展的缺点是在其被部署了之后才发现的。

REST 确实识别出了另一层编码的需求（the need for another layer of encodings）：由一个连接器将那些编码添加到一个消息上，从而改善了消息在网络上的可移交性（transferability）。这个新的层叫做移交编码（transfer−encoding，引用在 MIME 中一个类似的概念），允许为移交而对消息进行编码（allows messages to be encoded for transfer），而不是意味着表述从生成时就是已编码的。移交代理（transfer agents）可根据某种原因增加或删除移交编码，而不会改变表述的语义。

6.3.2.3 语义独立性

如上所述，HTTP 消息的解析与其语义是相分离的。对消息的解析（包括发现和将头信息字段组合在一起）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，并且能够在不破坏现有解析器的情况下对扩展进行部署。

6.3.2.4 传输独立性

早期的 HTTP 协议，包括大多数的 HTTP/1.0 的实现，使用了底层的传输协议来表示响应消息结束。服务器通过关闭 TCP 连接来表明响应消息的结束。不幸的是，这导致协议中出现了一个严重的故障状况：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，在 HTTP/1.0 中重新定义了 Content-Length 头信息字段，以表示消息体的字节长度（只要能够预先知道它的长度），并且在 HTTP/1.1 中引入了“chunked”（分块）这个移交编码。

chunked 编码允许表述在其生成阶段的开始时（at the beginning of its generation）（当头信息字段被发送时）尺寸是未知的，而通过一系列分块来描述它的界限，每个分块的尺寸可在被发送之前单独设置。它也允许在消息的末尾发送元数据，支持在生成消息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附加在消息的末尾）。

6.3.2.5 尺寸限制

对于应用层协议的灵活性而言，常见的障碍是在协议的参数上过度指定（over-specify）尺寸限制的倾向。尽管在协议的实现中总是存在着一些实际的限制（例如，可用的资金），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而进行很大的扩展。

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、或者任何由一列条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，在 HTTP 规范中知道存在这个问题就已经足够了，而无需要求所有的服务器都受到这样的限制（译者注：即，要求所有服务器支持的 URI 不得超过 255 个字符）。没有指定一个协议的最大值的原因是运行在一个可控环境（例如，在一个内联网中）中的应用能够通过替换旧的组件来避免那些限制。

尽管我们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，表明一个特定的协议元素对于一个服务器的处理来说是太长了。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，客户端无法向服务器表明它可能会存在的资源限制。当拥有有限资源的设备（例如 PDA）使用 HTTP 时不通过一个（与此特定设备相关的）中间组件对通信进行调节（adjusting the communication），而是试图直接与服务器交互时，可能会面临一些问题。

6.3.2.6 缓存控制

REST 努力在高效率的、低延迟的行为（efficient, low-latency behavior）和其所期待的语义透明的缓存行为（semantically transparent cache behavior）之间取得平衡，因此它允许由应用确定缓存需求，而不是将该需求硬编码在协议本身之中，这对于 HTTP 协议来说是至关重要的。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移交的数据，使得每个应用都不会受到愚弄，得到了一个东西，却以为是另外一个东西。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（由以下内容组成：方法、标识符、请求头信息、有时带有一个表述）映射到一个响应（由以下内容组成：一个状态码、响应头信息字段、有时带有一个表述）。当一个 HTTP 请求映射到了在服务器端的多个表述时，服务器可以与客户端进行内容协商，以便确定哪一个表述最适合于客户端的需求。这实际上更像是一个“内容选择”（content selection）的过程，而不是协商（negotiation）的过程。

尽管有一些已部署的内容协商的实现，但是它们却未被包括在 HTTP/1.0 规范之中，因为在该规范发布之时，并不存在可互操作的实现子集（interoperable subset of implementations）。部分原因要归咎于在 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 中内容可协商的 URI 远远少于 0.01%，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。

由服务器驱动的抢先式协商（preemptive negotiation）发生在以下情况：服务器根据请求头信息字段的值或正常请求参数之外的某个事物，为某个特殊请求方法/标识符/状态码（method*identifier*status-code）的组合而改变响应的表述。当这种情况发生时，客户端需要得到通知。这样缓存就能够知道，在何时可以在语义上透明地为一个未来的请求使用一个特殊的已缓存的响应，而用户代理一旦知道协商的信息对于所接收到的响应的影响，就能够提供比正常情况下发送的更为详细的首选项。HTTP/1.1 为这个目的引入了 *Vary* 头信息字段。Vary 字段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变化。

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，还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自己个人偏爱的内容正确地配置了他的浏览器。简而言之，服务器几乎无法有效地利用抢先式协商，但是这是早期 HTTP 协议所定义的唯一的自动化内容选择机制。

HTTP/1.1 添加了由用户代理驱动的反作用式协商（reactive negotiation）的概念。在这种情况下，当用户代理请求一个可协商的资源时，服务器使用一组可用表述的列表作为响应。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，条件性 HTML)（conditional HTML）中、或者作为一个“最有可能”的响应的补充。最后一种方式对于 Web 来说是最佳的方式，因为仅在用户代理确定了其他选项中的响应会更好时，才需要额外的交互。反作用式协商只是正常的浏览器模型的一个自动化的反映（automated reflection）（译者注：从上下文理解，应该是说对现有浏览器模型的改动最小），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表达表述维度的真实性质（communicating the actual characteristics of the representation dimensions）（即，如何表明浏览器支持 HTML 表格但不支持 INSERT 元素）的难题（译者注：这里 INSERT 是一个在 HTML 规范中不存在的虚构元素）。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；有更多的上下文信息（context information），当得到替换选项列表时可以使用这些信息作出选择；而不会对缓存产生影响。

第三种形式的协商，透明式协商（transparent negotiation）[64]，是特许一个中间缓存（an intermediary cache）来作为其他的用户代理的代理人（on behalf of other agents），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。这样请求就有可能在该缓存内部通过另一次缓存命中（another cache hit）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 头信息，这样其他的外部缓存（outbound cache）才不会产生混淆（译者注：在请求/响应链条的很多位置都可能会有缓存，如果不加 Vary 头信息，上述中间缓存所得到的表述，可能会来自于其他外部缓存，并不是客户端真正期待的）。

6.3.3 性能

HTTP/1.1 聚焦于在组件之间改善通信的语义，但是对于用户感知的性能也有一些改善，虽然它受到了与 HTTP/1.0 的语法相兼容这个需求的限制。

6.3.3.1 持久连接

在早期 HTTP 协议中，每个连接只能发送单个请求/响应。尽管这种行为实现起来很简单，但是它对于底层 TCP 传输机制的使用非常低效，原因在于每次交互都要付出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。因此，有一些扩展提议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部分（multipart）的 MIME 类型返回。这个建议遭到了拒绝，因为它违反了 REST 的几个架构约束。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，因为必须根据初始请求的头信息字段中的一个 content-length 字段集合来对请求的消息体进行长度分隔（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，我们采用了一种持久连接，它使用按长度分隔的消息（length-delimited messages），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，这样做通常无法正常运转工作，因为这个头信息字段能够被中间组件转发给其他的不理解 keep-alive 指令的中间组件，从而会导致死锁的状况（译者注：不理解 keep-alive 指令的中间组件，会在完成单个请求/响应之后关闭连接）。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 协议不支持回写式（write-back）缓存。HTTP 缓存不能假设通过它写入的内容与来自相同资源的后续请求可能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 请求的响应。定义这个规则有两个理由：1) 元数据可能会在后台（behind-the-scene）生成，并且 2) 对于以后的 GET 请求的访问控制（access control）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（译者注：从上下文理解，这里讲的是通过 PUT 请求实现的写入动作，PUT 请求是幂等的，相当于是对资源执行的赋值操作），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 协议中存在一些架构不匹配，一些是由于在标准过程（the standards process）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（由来源服务器为当前请求生成的）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设备（safety-critical information appliances）。当返回一个错误的响应时，客户端会疑惑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，因为是否为权威响应这个特性通常与响应的状态是正交的。

在 HTTP/1.1 中确实添加了一种控制缓存的行为的机制，这样可以表明所期待的是一个权威的响应。请求消息上的“no-cache”指令要求所有缓存将请求转发到来源服务器，甚至在它持有被请求内容的已缓存版本的时候。这允许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请求没有最终访问来源服务器时，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，但是在实践中并未得到广泛实现。

6.3.4.2 Cookie

对协议作出不适当扩展的例子之一，是以 HTTP Cookie [73]的形式引入站点范围的状态信息（site-wide state information），它所支持的功能与通用接口所期待的架构属性相矛盾。基于 Cookie 的交互与 REST 关于应用状态的模型不匹配，因此常常会在典型的浏览器应用中产生混淆。

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 通常会包含一个特定于用户的配置选项，或者包含着在未来的请求中与服务器端数据库进行匹配的标记（token）。问题是，根据定义，Cookie 应被附加在任何未来的请求上，对于特定的一组资源标识符，Cookie 通常与一个完整的站点相关联，而不是与浏览器中特定的应用的状态（一组当前呈现的表述）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对其进行传递，这会成为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。

结果，Web 上基于 Cookie 实现的应用永远都不值得信任。这一功能应该通过匿名认证和真正的客户端状态来完成。这种与用户偏好有关的状态机制，其实能够通过明智地使用上下文设置 URI（context-setting URI）得到更加有效的实现，而不是使用 Cookie。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 而产生无限数量的 URI。同样地，对于使用 Cookie 在服务器端数据库中标识某个特定于用户的“购物篮”而言，采用以下方式实现会更加有效：在超媒体数据格式中定义购物条目的语义（defining the semantics of shopping items within the hypermedia data formats），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条目。当客户端准备好购买时，使用一个 URI 来完成结账。

6.3.4.3 强制性扩展（Mandatory Extensions）

仅当 HTTP 头信息字段名称（HTTP header field names）所包含的信息对于正确理解消息并非必需的时候，才能够对其进行任意的扩展。对强制性的头信息字段进行扩展，需要对协议作一次大的修订（a major protocol revision），或者对于方法的语义作一次根本的改变（a substantial change），如同在 [94] 中所提议的那样。这是现代 Web 架构中与 REST 架构风格的自描述消息这个架构约束不匹配的方面，主要原因是，在现有 HTTP 语法中实现一个支持强制性扩展框架（a mandatory extension framework）的成本，超过了可能从强制性扩展中获得的任何明显收益。然而，当现有的语法向后兼容的架构约束不再适用后，期待在下一次对于 HTTP 所作的大修订中支持强制性字段名称扩展（mandatory field name extensions）是合情合理的。

6.3.4.4 混合元数据（Mixing Metadata）

HTTP 被设计为跨越网络连接对通用的连接器接口进行扩展。因此，我们期望它与该接口的特性相匹配，包括将参数描述为控制数据、元数据、表述。然而，HTTP/1.x 协议家族有两个最严重的局限：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）；而且不允许对元数据做有效地分层，以便对消息进行完整性检查。

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性，并预见到它们将在部署其他功能时导致出现问题，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 头信息以标识出无法安全地被中间组件转发的每次连接控制数据（per-connection control data that is unsafe to be forwarded by intermediaries），同时也包括了对于头信息字段摘要的一个规范处理的算法（an algorithm for the canonical treatment of header field digests）。

6.3.4.5 MIME 语法

HTTP 从 MIME [47] 中继承了消息的语法，以便保持与其他互联网协议的共同点（retain commonality with other Internet protocols），并且重用了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，而这些语法仅仅是为 MIME 的目标设计的。

在 MIME 中，用户代理将一堆信息（a bunch of information）发送给一个永远也不直接进行交互的未知接收者，希望这堆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，因为跨互联网邮件发送多条消息是低效的。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。

除了获得了安全的封装或包装过的档案（secure encapsulation or packaged archives）之外，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而发送单独的请求会更有效率。HTTP 应用使用像 HTML 这样的媒体类型作为容纳“包”的引用的容器（containers for references to the “package”）——用户代理随后能够选择通过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层是有损耗的（the transport is lossy），会故意将换行和内容长度等信息破坏掉。因此其语法有很多冗余，并且对于任何并非基于有损耗传输层的系统来说都是低效的，这使得它并不适合 HTTP 协议。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须的，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 将响应与请求相匹配

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 协议基于每次连接产生单个请求和响应，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 依赖于传输层的连接（transport connection）来确定这一匹配。

尽管 HTTP/1.1 被定义为是独立于传输协议的，但是仍然假设其通信是发生在一个同步的传输层（a synchronous transport）之上。很容易通过添加一个请求标识符来扩展 HTTP 协议，使得它能够工作在一个异步的传输层（例如 e-mail）之上。这样的扩展对于在广播或者多播

情况下（in a broadcast or multicast situation）的用户代理而言是很有用的，在那里响应可能是在与请求不同的另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移交顺序，这样就能够首先发送更小的或者更重要的响应。

6.4 技术推广

尽管 REST 对于 Web 标准的创作有着最直接的影响，但是将它作为架构设计模型（an architectural design model），是通过各种形式的商业级实现（commercial-grade implementations）来验证的。

6.4.1 libwww-perl 的部署经验

我参与到 Web 标准的定义工作，开始于开发维护机器人（maintenance robot）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 Web 项目，libwww-perl 提供了一个统一的接口，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，相比于旧的基础代码库（译者注：即更老的 libwww 库），它反映了对 Web 接口的更加现代的解释。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，当对 URL<http://www.ebuilt.com/>发送“GET”请求时，libwww-perl 将从 URL 中提取出模式“http”，并且使用它来构造一个对 *wwwhttp* 的 *resuest()*方法的调用。这里所使用的接口对于所有的资源类型（HTTP、FTP、WAIS、本地文件等等）都是通用的。为了获得这种通用的接口，这个库采用了与 HTTP 代理大致相同的方式来看待所有的调用。它提供了一个使用 Perl 语言数据结构的接口，该接口与一个 HTTP 请求语义相同，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，从命令行方式的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 规范开始以完整规范（complete specifications）的形式出现（译者注：即，规范的创作工作接近尾声），我们需要有服务器软件，既能够对提议的标准协议进行有效的展示，也能够作为有价值扩展的一个测试台（test-bed）。那个时候，最流行的 HTTP 服务器（httpd）是公共域（public domain）软件，是由 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 bug 的补丁（bug fixes），但是缺乏一个公共的分发版（a common distribution）。为了对我们所做的改变（作为对最初源代码的“补丁”）进行协调，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 项目是采用协作方式做软件开发的一次尝试（a collaborative software development effort），其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，使用互联网和 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（the Apache Group）。后来，这个开发组形成了非盈利性质的 Apache 软件基金会，作为一个法律和财务上的组织（as a legal and financial umbrella organization）来支持 Apache 开源项目的继续开发。

Apache 变得很有名，既是因为它能够以很健壮的方式支持对互联网服务的各种不同的要求，也是因为它对于 HTTP 协议标准的严格实现。我在 Apache 开发组中作为“协议警官”（protocol cop），编写核心的 HTTP 协议解析功能（the core HTTP parsing functions），通过解释标准来支持其他的任务，并且在此标准的相关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，Apache 开发组成员的苛刻评论也影响了协议背后的理论。

Apache httpd 被广泛地看作是最为成功的软件项目之一，并且是最早占据了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 调查了公共的互联网网站，发现超过两千万的站点基于 Apache 软件，占据了被调查的全部站点 65%以上[http://www.netcraft.com/survey/]。Apache 是第一个支持 HTTP/1.1 协议的主流 HTTP 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 架构相关标准方面影响力的认可。

6.4.3 开发顺从于 URI 和 HTTP/1.1 规范的软件

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 架构相关协议的软件产品。尽管可能是个巧合，就在微软的 IE 浏览器成为第一个实现了 HTTP/1.1 客户端规范的主要的浏览器之后不久，IE 浏览器在 Web 浏览器市场的份额超过了网景的 Navigator 浏览器。除此之外，在标准化的过程中还定义了很多个别的 HTTP 协议扩展，例如 Host 头信息字段，现在已经得到了广泛的部署。

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，并没有因为新标准的引入而导致的严重问题（significant problems caused by the introduction of the new standards），甚至是在它们不得不在遗留的 Web 应用旁边以渐进的和片段的方式进行部署的情况下，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）（译者注：又称 content delivery network，缩写为 CDN）改善用户感知的性能的新方法。

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] 相区分的是它使用 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端软件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以及相关的符号/参数集，假如一个程序员遵循来自那些代码的架构上和语言上的约束，这个程序员就能够使用其他人的代码来完成维护类似系统之间真实接口（maintaining the actual interface between like systems）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（the social dynamics）相匹配（译者注：即，无法及时满足因 Web 开发者的大量需求）。

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”（分裂为分别维护的基础代码），因为在 NCSA 的开发者无法等待 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 的设计工作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，跨越多个平台和实现来扩展所期待的 Web 语义（extending the desired semantics of the Web across multiple platforms and implementations）。

对于应用的交互而言，一个基于网络的 API 是一种带有已定义语义的线上语法（an on-the-wire syntax, with defined semantics）。除了需要读/写网络之外，一个基于网络的 API 没有在应用的代码上强加任何限制，但是确实在能够跨接口进行高效通信的语义集合（the set of semantics that can be effectively communicated across the interface）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任何特殊实现。

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，也带来了非常多的额外复杂性，并且其负担超出了任何单个系统所必须承受的限度，这种方法在一个成分混杂的网络中（a heterogeneous network）可移植性很差，而且总是会导致优先选择通用性，而不是优先选择性能。作为一个副作用，它也会导致在开发过程中产生惰性（为任何事情都去责备 API 代码），完全忽略了因其他通信参与方的不合作行为而产生的问题（failure to account for non-cooperative behavior by other parties in the communication）。

然而，很重要的是要记住在任何架构中都包括有不同的层级，包括现代 Web 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，但是它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，它的移交协议 IIOP 对于参与方的通信内容（what the parties are communicating）做了太多假设。HTTP 请求消息中包括了标准化的应用语义（application semantics），而 IIOP 消息中却没有包括这样的语义。“Request”符号在 IIOP 中仅仅提供了方向性的含意，这样 ORB 能够根据是自身应该作出回应（例如“LocateRequest”）还是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作为 key 的对象和操作的组合（the combination of an object key and operation）来表达，它是特定于某个对象的，而并非对所有的对象都是标准化的。

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（a structured binary content that mirrors that IDL operation's signature）、遵循 IDL 规范的回应数据类型的定义（the definition of the reply data type(s)）。其语义因此由网络接口（IIOP）来定义，而不是由对象的 IDL 说明（IDL spec）来定义。这是否是一件好事，取决于应用的性质——对于分布式对象而言这是必需的，对于 Web 而言这并不是什么好事。

为何这些差别很重要呢？因为它可以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 允许由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，即使是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，而不是能够自由地使用最适合于它们的应用类型的东西（whatever is most appropriate for their type of application）。

6.5.2 HTTP 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，仅仅是因为它也包括了请求和响应。RPC 与其他形式的基于网络应用的通信（other forms of network-based application communication）的区别之处在于，从概念上讲它是在调用远程机器上的一个过程（procedure）。在 RPC 协议中，调用方识别出过程并且传递一组固定的参数，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，差异仅仅是将过程标识为一个｛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（Brokered RMI）添加了名称服务的间接层（name service indirection）和少量其他把戏（a few other tricks），但是接口基本上是相同的。

将 HTTP 与 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 而言是不可用的。HTTP 与 RPC 之间的重大区别的是：请求是被定向到使用一个有标准语义的通用接口（a generic interface with standard semantics）的资源，中间组件能够采用与提供服务的机器（the machines that originate services）几乎完全相同的方式来解释这些语义。其结果是使得一个应用能够支持转换的分层（layers of transformation）和独立于信息来源的间接层（indirection that are independent of the information origin），这对于一个需要满足互联网规模、多个组织、无法控制的可伸缩性需求的信息系统来说，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应用的需求来定义的。

6.5.3 HTTP 不是一种传输协议

HTTP 并非被设计为一种传输协议（transport protocol），它是一种移交协议（transfer protocol）。（译者注：非常不幸，HTTP 协议刚刚传入我国时，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“移交”。） 在 HTTP 协议中，通过对资源执行各种动作（实现方式为移交和操作那些资源的表述），消息所反映出的是 Web 架构的语义（the messages reflect the semantics of the Web architecture）。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 协议的语义（HTTP semantics）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，从根本上来这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防火墙的目的，而且从长远来看它将无法运转，因为防火墙的厂商将会不得不执行额外的协议过滤。因此将那些扩展加在 HTTP 之上是没有意义的（makes no sense to do those extensions on top of HTTP），因为在那种场合下 HTTP 所完成的唯一的事情就是为一个遗留语法添加了更多负载（译者注：即，在应用协议本身的负载之外添加了额外的 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（译者注：例如浏览器、缓存、HTTP 代理，其本身并不需要理解特定于应用的业务知识）。

6.5.4 媒体类型的设计

REST 有一个对架构风格来说不同寻常的方面，那就是它对于 Web 架构中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 定义了一个可获得以下其所期待的应用行为（expected application behavior）的架构模型：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应用的局部故障状况（the partial failure conditions）。然而，这并不足以阻止应用的开发者引入违反这个架构模型的功能。最频繁出现的是对应用状态和无状态交互架构约束的违背。

由于将应用状态放错了地方而造成的架构不匹配，并不仅限于前面提到过的 HTTP Cookie。在超文本标记语言（HTML）中引入的“frame”（帧）引起了类似的混淆。frame 允许将一个浏览器窗口分割为子窗口，每个子窗口都有自己的导航状态。在一个子窗口中选择链接而导致的状态迁移，与正常的状态迁移是无法区分的，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。假设不存在离开子窗口信息领域的链接（no link exits the realm of information that is intended for subwindow treatment），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：即，原本希望在完整的浏览器窗口中显示的内容，却出现在了子窗口中）。

frame 和 cookie 的失败之处在于，用户代理无法管理或解释它们所提供的间接应用状态（indirect application state）。替代的设计是将这些信息放在一个主要的表述（a primary representation）中，并且告知用户代理如何去管理这个存放了指定的资源领域的超媒体工作区（the hypermedia workspace for a specified realm of resources）。这种设计能够完成相同的任务而不会违反 REST 的架构约束，同时还带来了更好的用户界面以及对缓存更少的干扰。

6.5.4.2 增量处理

通过将减少延迟作为一个架构目标，REST 能够按照用户感知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）的能力都会影响在移交、呈现、操作表述媒体类型（representation media types）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，它有着很好的延迟特性。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获得的——就在构成 HTML 的少量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：将内嵌的元数据（embedded metadata）放在一个文档的 HEAD 标签中，这会导致浏览器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，必需先移交和处理一些可选的信息；使用没有呈现尺寸提示的内嵌图片，要求在包围它的其余的 HTML 能够被显示之前，要能够接收到该图片的最初的一些字节（带有布局尺寸的信息）；使用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读取整个表格并确定其尺寸；容忍带有错误格式的标记的懒惰规则（lazy rules），常常会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 HTML 文件。

6.5.4.3 Java vs. JavaScript

通过使用 REST，我们还能够获得以下这些领悟：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接受，甚至是在这些类型并未取得开发者偏爱的情况下（even when the balance of developer opinion is not in their favor）。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 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，Java 受到了广泛的关注，但是它却并没有被 Web 中按需代码的应用开发者所广泛接受。

稍后于 Java 的引入，网景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，但是后来因为营销方面的原因改名为 JavaScript [44]（这两种语言除了名称中的四个字母相同外，几乎没有什么共同之处）。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 相比都是很糟糕的。也不是因为营销方面的原因：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 更加适合于 Web 技术的部署模型（the deployment model of Web technology）。它具有低得多的使用门槛（a much lower entry-barrier），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初始工作量（the amount of initial effort）比较小。与 Java 相比，JavaScript 对于交互的可见性所产生的影响也比较小。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（the security restrictions within the Java execution environment）。同样地，Java 还拥有很多在一个安全环境中令人质疑的额外功能，包括将 RMI 请求发送到来源服务器的能力，而 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感知的延迟（user-perceived latency）。JavaScript 通常作为主要表述的一部分来下载的，然而 Java applet 则要求一个独立的请求（译者注：从上下文理解，作者可能是说下载 Java applet 需要通过另外一个连接）。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。

一旦我们将编程语言的特性按照与 REST 架构约束背后的基础理论相同的维度摊开来之后，根据这些技术在现代 Web 架构中的行为来对它们加以评估就变得容易多了。

6.6 小结

本章描述了在创作超文本移交协议（HTTP）和统一资源标识符（URI）两个互联网规范的过程中学到的经验和教训。这两个规范定义了在 Web 上进行交互的所有组件所使用的通用接口。此外，我还以 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所学习到的经验和教训。

结论

我们每个人在内心深处都怀有一个梦想：希望创造出一个鲜活的世界，一个宇宙。那些处在我们生活中心、受到过专业训练的架构师们，都怀有这样的渴望：在某一天，在某个地方，因为某种原因，我要创造出一个不可思议的、美丽的、夺人心魄的场所，在那里人们可以漫步，可以梦想，历经很多世纪生生不息。

——Christopher Alexander[3]

我们在互联网工程工作组（IETF）定义了现有的超文本移交协议（HTTP/1.0）[19]，并且为 HTTP/1.1 [42] 和 URI（统一资源标识符）[21] 的新规范设计扩展。在开展这些工作的最初阶段，我就认识到需要建立一个关于 Web 的运转方式的模型。这个关于整个 Web 应用中的交互的理想化模型，被称作表述性状态移交（REST）架构风格，它成为了现代 Web 架构的基础。REST 提供了指导原则（the guiding principles），通过这些原则的指导，能够识别出先前存在于架构之中的缺陷，并且使得各种扩展在部署之前就能够得到验证。

REST 是一组相互协作的架构约束，它试图使延迟和网络通信降低到最小，同时最大限度提高组件实现的独立性和可伸缩性。REST 通过将架构约束放置在连接器的语义之上来达到这些目标，而其他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、以及通过中间组件来对于动作进行处理（processing of actions by intermediaries），因此满足了一个互联网规模的分布式超媒体系统的要求。

这篇论文对于信息和计算机科学领域作出了如下贡献：

- 定义了一个通过架构风格来理解软件架构的框架，包括了一组自恰的术语，用来描述软件架构；
- 通过当某种架构风格被应用于一个为分布式超媒体系统设计的架构时，将会产生的架构属性，来对基于网络应用软件的架构风格进行分类。
- 描述了 REST——一种为分布式超媒体系统设计的新型架构风格；以及
- 在设计和部署现代 Web 架构的过程中，应用和评估 REST 架构风格。

现代 Web 是 REST 架构风格的一个架构实例。尽管在基于 Web 的应用中也能够包括其他风格的交互（access to other styles of interaction），但是 Web 的协议和性能关注点的核心焦点是分布式超媒体。REST 仅仅详细描述了架构中的一部分内容，对于互联网规模的分布式超媒体交互而言，这些内容是最为本质的部分。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有效的形式而不需要改变架构能力（the architecture capabilities）的地方，都存在着 Web 架构需要加以改进的领域。同样地，我们能够将协议扩展与 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，将那个功能放到一个具有更加适用的架构风格并行运行的系统中来实现。

在一个理想的世界里，软件系统的实现与它的设计有着精确的匹配。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（design criteria），例如将 URI [21] 作为资源标识符来使用，以及使用互联网媒体类型（Internet media types）[48] 来标识表述的数据格式。然而，由于失败的遗留系统试验（legacy experiments that failed） （但是必须保持向后兼容）和对架构风格缺乏了解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

和理解已部署的破损功能（broken features）。

尽管仍然存在争论，Web 事实上已经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，特别是那些遵循着相同的或相似交互方法的应用。REST 既贡献了在现代 Web 软件架构背后的基础理论，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过程中。

对于基于网络应用来说，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移交组成，而不是由计算密集型的任务组成。开发出 REST 架构风格正是为了满足这些需求，它聚焦于资源和表述的通用连接器接口（the generic connector interface），支持使用中间组件进行处理、缓存、组件的可替换性（substitutability of components），这使得基于 Web 的应用的规模从 1994 年的每天 10 万个请求发展到了 1999 年的每天 6 亿个请求。

通过对 HTTP/1.0 [19] 和 HTTP/1.1 [42] 规范长达 6 年的开发、精心设计的 URI [21] 和相关的 URL [40] 规范、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 架构风格得到了充分验证。它既可以被用来作为一个指导设计的模型，也可以被用来对 Web 协议的架构扩展进行严格的测试（an acid test for architectural extensions to the Web protocols）。

我们将来的工作会聚焦于以开发 HTTP/1.x 协议家族的替代者（译者注：例如正在开发的 HTTP/2.0 协议）为目标来扩展这个架构指导（extending the architectural guidance toward the development of a replacement for the HTTP/1.x protocol family），将会使用一个更加有效的符号化语法（tokenized syntax），但是不会丢失由 REST 识别出的所期待的架构属性。无线设备的需求（它有着很多与 REST 背后的原则相同的特性）将会促进应用级协议设计和包括了主动的中间组件的架构（architectures involving active intermediaries）这两方面的发展。我们在以下方面也有一些兴趣：扩展 REST 以便支持可变的请求优先级、有区别的服务质量（quality-of-service）、由持续的数据流（continuous data streams）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）。

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 InternetGopher 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 InternetMail Extensions (MIME) Part One: Format of InternetMessage Bodies. Internet RFC 2045, Nov. 1996.

48. N. Freed, J. Klensin, and J. Postel. Multipurpose InternetMail 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 InternetComputing, 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. InternetSTD 8, RFC 854, May 1983.

107. J. Postel and J. Reynolds. File Transfer Protocol. InternetSTD 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 InternetEnvironments. 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 InternetTechnologies 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.


1kg.org
多背一公斤
爱自然 | 更爱孩子

InfoQueue

促进软件开发领域知识与创新的传播

中文 | 英文 | 日文 | 葡文 | ……

内容投稿 editors@cn.infoq.com | 商务合作 sales@cn.infoq.com | 加入我们 hr@cn.infoq.com