WWW 服务器技术
——Apache 使用指南与实现原理

牛锦中 牛锦宇 李锦涛 著

2002 年 1 月 28 日

目　　次

前　言

网络的兴起使计算机世界焕然一新。在网络技术的发展中，除了 TCP/IP 协议的诞生之外，我们认为，就应属 WWW（World Wide Web）影响巨大。WWW 面向终端用户，提供了在 Internet 上共享信息的友好且充分的手段。很多网络技术在出现后，由于灵活性或适应性的不足，很快便销声匿迹；与之相反，WWW 技术则由于其极强的扩展能力和符合人们获取信息的习惯，成为 Internet 应用层的主宰，是呈几何级数增长的网络知识空间的基础。

WWW 服务器是网上信息的“集散地”，是 WWW 世界的“神经中枢”，从而也是 WWW 知识共享模式中的要害，因此 WWW 服务器的性能至关重要，尤其是那些异常繁忙的门户网站；另一方面，Internet 是公认的自由世界，有众多的“大虾”、“虫虫”希望拥有自己的一片天地，选择并用好 WWW 服务器是日后在网上“扬名”的重要一环。这一切都需要有对 WWW 技术尤其是 WWW 服务器技术的深刻理解为前提。

但我们看到，关于 WWW 服务器的书籍并不很多，能够深入到技术内部的更是少之又少。本书希望能为各位读者在这方面提供一定的帮助。本书的最大特点是：它是中国科学院计算技术研究所家庭网络课题组实际工作成果的总结，对 WWW 服务器进行了清晰透彻的分析，特别适合于广大希望了解网络软件核心技术的爱好者及广大学生和科研人员。

说到 WWW 服务器技术，则不能不说 Apache。在诸多 WWW 服务器产品中，Apache 无疑是最著名的一个，有相当多的门户网站如新浪网等均采用 Apache 作为其运行平台，足见其魅力，其市场分额超过 60%，极具代表性。Apache 的模块化设计提供了规范扩展接口，其进程与线程管理、内存空间的分配等方面均匠心独具，开放源代码的特点更使其性价比极高。

本书以 Apache 为对象，对 WWW 服务器的使用与实现原理进行了详细阐述和深入分析。本书共 9 章。第 1、2 章分别描述了 WWW 服务器和 Apache 的概貌；第 3、4 章则对 Apache 服务器的安装、使用、运行、访问等进行了介绍；Apache 服务器的配置管理、配置命令的使用集中在第 5 章；第 6 章概括性地阐述了 Apache 的整体设计思想——模块化结构，希望在 Apache 服务器上增加额外功能的读者通过阅读本章，可以了解扩展模块的开发及运作的基本步骤；第 7 章在本书中所占篇幅最大，详细讨论了 Apache 服务器的具体实现，读者按照该章脉络，对 Apache 源代码进行分析，将深刻体会前面各章对 Apache 的概念性描述，并对什么是好的软件体系结构有一个身临其境的感受；鉴于 Perl 和 Java Servlet 是近年来较为流行的服务器端编程技术，第 8 章介绍了 Apache 的扩展模块 mod_perl 和 mod_jserv，并给出了一个基于 Java Servlet 的分布式、可扩展、能容错和动态负载平衡的大型站点设计的例子，具有实用参考价值；WWW 服务器从根本上说是 HTTP 协议的服务器端实现，第 9 章则是一个完整的 HTTP 协议规范，任何涉及该协议的开发均可参考该部分。

很久以来，一般认为缺乏管理上的规范化和进而规模化是中国软件业的主要问题。我们在实际工作中深刻体会到，作为软件开发者的个体对软件技术的把握，特别是已实现正确功能的前提下对软件可用性或说能用性的把握，也是尤为重要的问题之一。实际上，在软件组织结构上的高水平，通常自然地产生对软件开发过程规范化的深刻理解。我们现在还很少能看到由中国人完成的软件核心技术的创新，希望本书通过对 Apache 服务器实现原理的介绍，对我国计算机技术人员集中精力钻研核心技术并进而有一定创新起到推动作用。

作者

2001 年 11 月 15 日

第1章 概述

WWW 是近几年 Internet 上最热门的话题，一时间它以异乎寻常的速度迅猛发展，目前已经跃升为 Internet 上最主要的应用。其引发的信息流量是如此之大，经常造成某些区域网络甚至整个 Internet 的性能下降。如在北约轰炸南斯拉夫时，众多网民为获取快捷迅速的信息，将新浪网都“点”瘫痪了。

WWW 技术和应用，包括 HTML，HTTP，各种 Web 服务器软件，还有 Mosaic、Netscape 等客户端软件，以及飘香整个网络的的 Java 等，都是二十世纪九十年代信息技术领域非常了不起的发明。WWW 的崛起为信息发现和获取带来了新的机遇和思路，它和其它计算机网络技术一道，必将进一步促进社会的网络化和网络的社会化，对人类社会的发展产生深远的影响。

在网络社会，WWW 站点正在并将至少在很长的时间内充当信息集散地的角色，而且不仅再是简单的信息，逐渐地，各种商务活动、软件服务等形形色色的东西也将被囊括其中。而 WWW 服务器是 WWW 站点的基础软件平台，居于重要地位。本书介绍的 Apache 服务器就是一种最受欢迎的 WWW 服务器。

在了解具体的 WWW 技术和 Apache 服务器之前，我们首先来回顾以下历史，以便使我们在学习 WWW 技术特别是 WWW 服务器技术时，能有一种顶天立地的感觉，能知过去将来事。

第1节 WWW 的发展历史

早在 1965 年，Ted Nelson 便首先提出了“超文本”的概念，并在 1967 年把相应的实现计划命名为 Xanadu。但该项目于 1987 年才算完成，只设计出“一个运行于 SUN 工作站上的粗糙工具”(Conklin)。

欧洲粒子物理实验室（即 CERN）的 Tim Berners-Lee 受到 Nelson 的影响，提出一项计划，目的是使科学家们能很容易地查阅同行的文章。该项目从 1990 年 10 月开始到 1990 年 12 月完成，结果出版了命令行方式浏览器和 NeXTStep 浏览器。该浏览器可用于浏览服务器超文本文件及 CERN 的 USENET。1992 年 7 月，WWW 在 CERN 内部广泛使用。到 1993 年 1 月为止，全世界共有 30 台 WWW 服务器，并有多种浏览器版本发行。

1993 年伊利诺斯大学 Urbana-Champaign 分校的国家超级计算应用中心 NCSA(National Center for Supercomputing Applications）发行了一个新的浏览器软件——Mosaic 浏览器。Mosaic 浏览器在继承前人成果的基础上，不仅能处理基本的文档共享和链接，而且还支持多种网络协议及 MIME 协议，用户在浏览器中能直接浏览图形甚至听到声音。这时 WEB 基本成熟。

尽管 WWW 的发展历史在浏览器的演进中更明显，但它们在“前台的表演”,离不开“幕后的英雄”——WWW 服务器。

WWW 服务器中两个“元老”—— CERN 和 NCSA 服务器是几乎其它一切服务器软件的祖先。本书的主角 Apache 服务器便是来自于 NCSA 服务器的。

后来直至现在，各种商业和免费 WWW 服务器纷争，Netscape 和 IE 两种主流浏览器龙虎斗，更使 WWW 在 Internet 中空前火爆，从此 WWW 进入了快速发展的阶段。

第2节　什么是 WWW

WWW（World Wide Web）在我国称为全球网或万维网，简称 Web。WWW 实质上是用超链接（hyperlink）将互联网（Internet）上的所有信息连接在一起的机制。比如，电子邮件、远程登录、新闻阅读、gopher 资源、超文本文档等都可通过 WWW 的机制访问。通过 WWW 能访问的，除文本形式的外，还有声音、图像、动画、数据库等等。

WWW 采用的是客户机/服务器体系。用户所用的浏览程序为 WWW 的客户机端，运行 WWW 服务的计算机为 WWW 服务器。用户用浏览器浏览 WWW 服务器管理和提供的信息，它们之间是通过连接机制（connection）相互通信的。

用户浏览到的 WWW 信息资源是以页（Page）为单位，其中用户所在网络的 WWW 服务器的第一页称为主页（Homepage）。页实质上是一包含超链接的超文本文件（Hypertext document），用户从一页到另一页的浏览实质是通过超链接进行的。

第3节　WWW 服务器及其技术

1.3.1. 概貌

WWW 服务器系统构成

WWW 服务器的构成可以简单地用下述公式表达：

服务器 = 平台 + 软件 + 信息

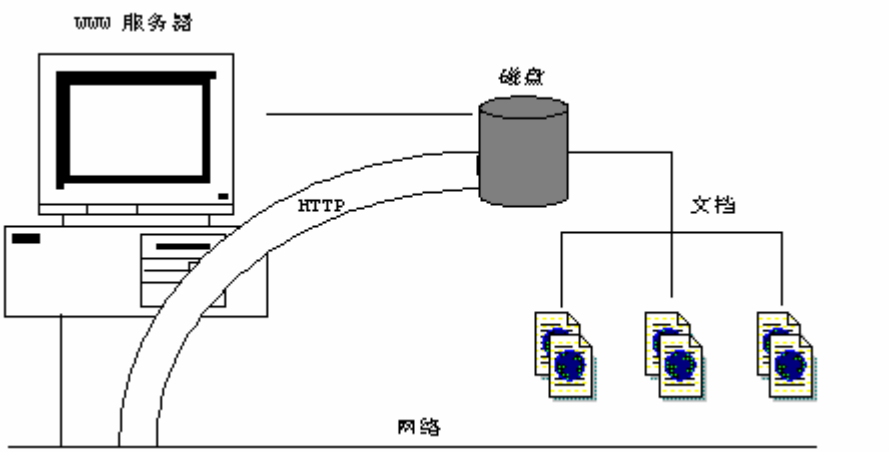


图 1.1

WWW 服务器必须是一台连接到 Internet 上的计算机。计算机硬件、操作系统和网络通信软件构成了计算“平台”。 WWW 服务器的第二部分组成是 WWW 服务器软件本身，即公式中的“软件”。而最重要的，则是 WWW 服务器提供的信息服务！没有了信息，WWW 服务器就没有了存在的道理。

如图 1.1 所示，WWW 服务器软件将整个系统组合在一起，是连接包含信息的文档集合与计算平台的桥梁。

WWW 服务器软件的工作原理非常简单：接收浏览器对文档的请求，并进行解析以确定被请求的文档，进而取到该文档，通过建立的网络连接传回浏览器。让我们来看一下 WWW 服务器软件是如何一步步工作的。

WWW 服务器软件持续运行，等待来自 Internet 上客户的请求。有请求时，创建网络连接，接收该请求。那么 WWW 服务器软件是如何知道有请求的呢？实际上，Web 服务同底层网络的交互与 Internet 上的其他服务类似。计算机上的操作系统负责面对复杂多样的底层网络，对其进行简单的抽象，以便人与程序能更容易地理解和工作。例如：程序通过在两个端口（port）之间建立连接来实现网络通信。“端口”是一种抽象，为创建和使用网络连接提供简便、一致的方式。当一个程序连接另一个程序时，它会给后者的端口发送请求。该端口就确定了前者要连接的程序。在发送者的机器上，数据通过写端口被输出，到达接收程序的输入端口时被读取。

每个 WWW 服务器都被赋予一个用数字表示的端口。一个端口就表示与计算机系统的一个网络连接。浏览器，作为 WWW 客户端程序，就把 WWW 服务器的端口看作其在网络上的可访问点。当浏览器要请求 WWW 服务器时，便向 WWW 服务器被赋予的端口发请求。而对 WWW 服务器来说，数据就从该端口处进来。WWW 服务器软件只需要知道如何读写端口，而不需知道信息在网络上传输的具体细节。对客户与服务器间连接的复杂的管理是存在于两端口机器的操作系统和网络通信软件中的。这也正是 Internet 协议分层机制的突出优点。

当 WWW 服务器程序接收到请求时，它必须对请求进行解析，并响应之。WWW 服务器程序必须定位目标文件，并把它从磁盘上拷贝到网络上传给客户。被请求的文档是按照名字即在 web 文档树中的位置来指定的。要根据名字确定对应文档，服务器只需在文档树中搜索名字的各个部分，即逐步地确定文档位置。WWW 服务器软件实际上是把确定文件位置的工作交给操作系统完成的。然后该文件便会从磁盘上被读入并通过网络发送给客户。WWW 服务器程序实际上并不知道文档中有什么以及请求的文档是否正确。它知道的只是文件的名字，且也只是将请求中指定的文件返回。

WWW 服务器模式

我们知道，在计算机网络中，服务器起着核心作用。在 WWW 中，有客户向服务器提出请求，而事务处理一般由服务器完成。

如果一个服务器一次只能收到一个请求，并且在下一次客户发出请求、提交任务前有足够的时间和空间去响应这次请求的话，那么服务器端的工作就简单了。但实际上，服务器在同一时刻可能收到多个客户的请求，从而产生“拥挤”，此时如何响应客户的请求就成了一个问题。根据不同的解决方案，我们可以把服务器的工作模式分为以下几种：

1. 阻塞模式（Blocking Model）

阻塞模式下，如果服务器在响应一个客户的请求，则其它的客户的请求均不被响应，必须等到当前的客户的请求被处理完毕之后，如果这样，用户可能会花很长时间去等待服务器的响应。所以负载较重的服务器一般都不采用这种模式。

2. 多路复用模式（Multiplexing Model）

该模式下，服务器可以同时为多个客户的请求服务，不过，服务器一般采取“平均制”的工作方式：为第一个客户的请求服务一会儿，又去为第二个客户的请求服务一会儿……。如果客户很多，一个客户要得到完全的服务将要花很多时间，这也是不合理的。

3. 派生模式（Forking Model）

在派生模式下，服务器按客户的请求派生出一个与自己一样的进程去响应用户的请求，这样，许多用户同时访问服务器，也不会等待太长时间。我们知道服务器复制自己是需要时间的，同样也会占用内存。内存是一个有限的资源，尽管虚拟内存解决了一部分问题，但是进程是不能在硬盘上运行的。所以进程需要在硬盘和内存之间调入调出，这样也就降低了系统的效率。不过，派生模式是 WWW 服务器常用的模式，CERN 和 NCSA 均采用这种模式。

4. 进程池模式

进程池模式是对派生模式的改进。我们知道在派生模式下，服务器产生了许多与自己一样的进程副本去响应用户的请求，完成任务后就退出，而在进程池模式下，进程池服务器会创建一组子进程，而非进程副本去快速响应客户的请求，这样可大大减小系统对内存的要求。同样，这样的子进程存在于整个服务器生存期间，而不象进程副本那样频繁地创建、销毁进程。目前采用进程池模式的 WWW 服务器有 NCSA 的 HTTP1.4、Netscape 的 NetSite 和本书的 Apache 服务器等。

HTTP 协议

WWW 的实质是信息的传输：从各个服务器到 Internet 的用户。但是既然其中涉及许多不同的计算机网络和许多不同的连接方式，所以任何两台计算机要通信必须有共同认定的标准和规定。在计算机网络领域用于通信的标准和规定，称为协议。

WWW 所使用的协议是 HTTP 协议（HyperText Transfer Protocol，超文本传输协议）。WWW 服务器和客户端程序必须遵守 HTTP 协议的规定来通信。HTTP 协议基于 TCP/IP 连接实现。没有了 HTTP，WWW 也将不复存在。因此通常还将 WWW 服务器程序称为 httpd（HTTP Daemon，HTTP 服务奴），尤其在 UNIX 领域更是如此。

本书的第 9 章包含了 HTTP 协议规范的详细介绍，对想了解 HTTP 协议细节或要实现 WWW 服务器或浏览器的读者将大有作用。在读本书前对 HTTP 一无所知的读者，请先读一下该规范的概貌部分，以作为阅读本书其它部分的基础。另外，还可以结合下一节的 WWW 服务工作过程部分来理解。

1.3.2. 信息的组织

WWW 服务器上的文档是以层状或树型结构组织的。与传统的层次化文件系统类似，文档树也有一个根及一些包含若干子目录和文件（文档）的目录。WWW 服务器上的每个文档都有一个唯一的名字。该名取从根“/”开始的路径，所以一个名为 tree.html 的文件将可能有一个类似于/some/where/in/the/tree.html 的完整名字。

但是需要特别注意的是：在 WWW 服务器上包含 web 文档的文件实际可能有许多不同的组织方式。web 文档树只是反映了一个具有复杂结构的文档集的简单视图。这个简单视图作用重大：它让人们在复杂的服务器系统中能查找所需要的东西！

WWW 服务器上的每个文档都是文档树的一部分。它们有各自的名字，从而使浏览器

可以对它们包含的信息提出请求。HTML 文档中的超链接，使用文档名来指向对应的信息。链接可以是文档的完整名字，如：

```
http://www.server.org/there/silly.html
```

或是相对于当前文档的名字，如：

```
./silly.gif
```

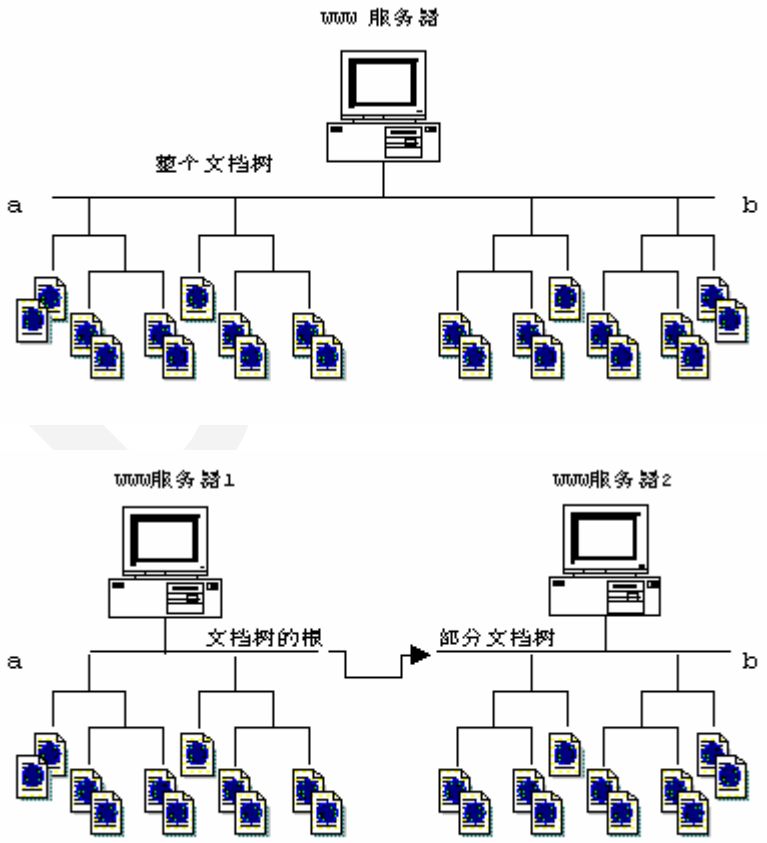
第二种方式表明 silly.gif 与包含该链接的文档处于同一路径（除了文件名外）。也就是说，如果链接存在于/there/silly.html 中的话，silly.gif 的全名将是：

```
http://www.server.org/there/silly.gif
```

相对名字通常用于内嵌的图象。一般地，一个文档的各个所含部分都在 WWW 服务器的同一个地方。

需要特别强调的是，web 文档树并非反映了 web 文档的真正组织结构。那么，web 文档为什么不组织成简单的树型结构呢？这有很多原因。比如，web 文档是由若干个小组分别制作的，他们每个都有自己的工作区和存储区；再如，web 文档太多，使得只放在一个文件系统下，甚至一台计算机上都过于庞大；WWW 服务器的流行与飞速发展，也使得它有必要把文档分布到若干台计算机上，以提高整个 web 系统的最大可承受负载。

图 1.2 给出了 WWW 服务器上文档组织的三种重要形式。所有文档可以是位于一台机器上，构成一棵树，也可分散到几台机器上，还可以在不同机器上对同一文档树进行镜像。需要注意的是，尽管内部组织方式不同，但对用户请求来说，则无半点区别。



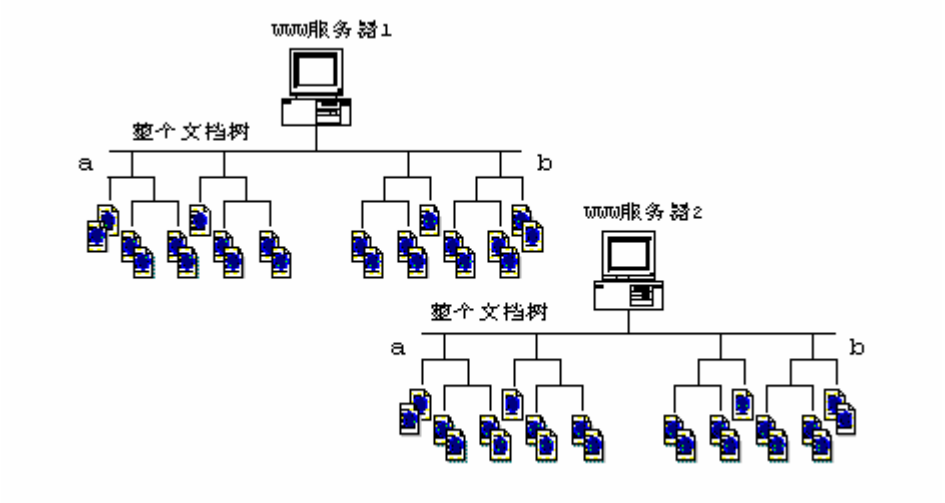


图 1.2

1.3.3. 工作方式

现在我们已经知道，WWW 服务器拥有一系列组织成文档树的信息，并遵循 HTTP 协议为客户服务。那么它是如何工作的呢？下面分步骤展示 WWW 服务器的工作过程：

第一步：等待客户请求。

WWW 服务器等待来自 Internet 的客户的请求，如图 1.3 所示。WWW 服务器监听一个端口，直到有请求进来。此时，服务器处于“睡眠状态”。

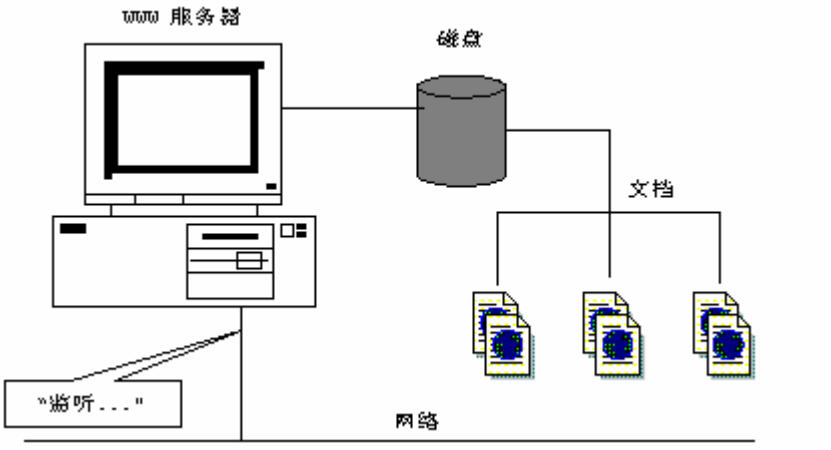


图 1.3

第二步：客户请求到来。

客户通过浏览器对文档提出请求。用户可能是敲入了一个 URL 地址，也可能是点击了一个 HTML 文档中的锚点。无论何种方式，均会发出请求。本例中使用的 URL 地址是：

```
http://www.server.org/sample.html
```

它也可能是位于一个 HTML 锚点中，如：

```
<A HREF="http://www.server.org/sample.html"> Click here </A>
```

客户机上的网络软件负责定位服务器主机，并建立一条客户与服务器 www.server.org 间

的双向网络连接。实现这一点依赖于 Internet 上的诸多协议及域名服务系统。

一旦连接建立，客户便按照 HTTP 协议发出请求。在本例中，客户请求文档/sample.html。发出如下 ASCII 字符串：

```
GET /sample.html HTTP/1.0
```

这些字符串经过网上传输，被服务器接收并保存在内存中。图 1.4 给出了基于与 WWW 服务器的连接的请求的发送。

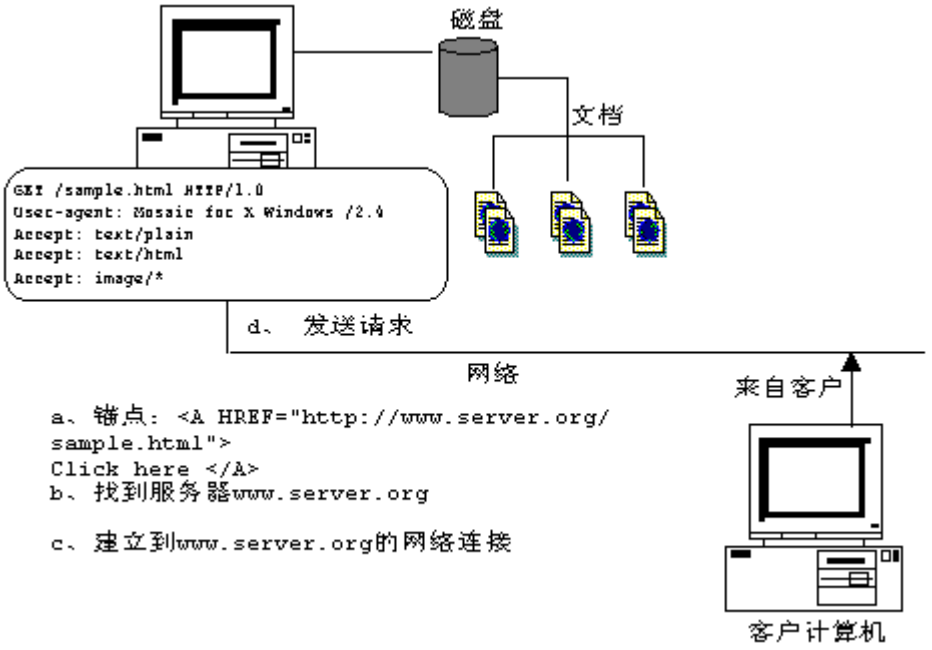


图 1.4

第三步：服务器解析请求

WWW 服务器对请求按照 HTTP 协议进行解码来确定进一步的动作。如图 1.5 所示。涉及的内容有三个要点：方法（GET）、文档（/sample.html）和浏览器使用的协议（HTTP/1.0）。其中方法告诉服务器应完成的动作，GET 方法的含义很显然是：服务器应定位、读取文件并将它返回给客户。

WWW 服务器软件现在就知道了：它应找到文件/sample.html，并使用 HTTP/1.0 协议将内容返回给客户。信息是经过与请求到来相同的连接发出的，所以服务器不需要定位客户或创建新的连接。

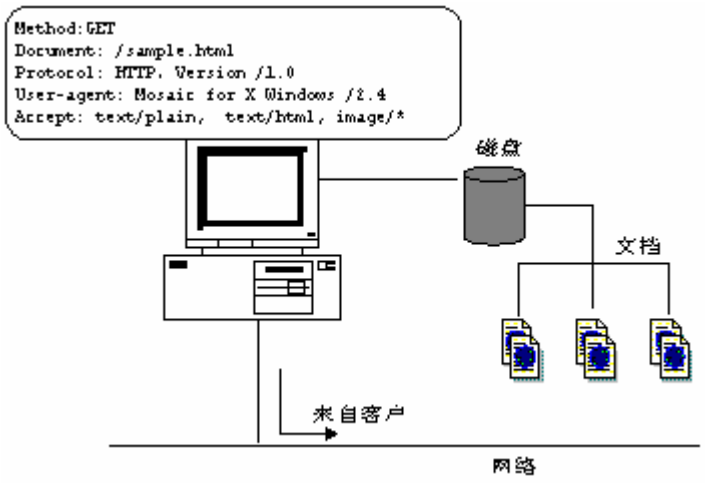


图 1.5

第四步：读取其它信息（如果必要的话）

WWW 服务器根据需要去读取请求的其它部分。在 HTTP/1.0 下，客户还应给服务器提供关于它的一些信息。元信息（metainformation）可用来描述浏览器及其能力，以使服务器能据此确定如何返回应答。在本例中，还有如下请求信息：

```
User-ageng: Mosaic for X Windows /2.4
Accept: text/plain
Accept: text/html
Accept: image/*
```

这些内容表明浏览器是 NCSA Mosaic。它可以显示文本及任意图象。这些信息在本例中不重要，所以下面不再讨论。

第五步：完成请求的动作

若现在没有错误出现，WWW 服务器将执行请求所要求的动作。要获取（GET）一个文档，WWW 服务器在其文档树中搜索请求的文件（/sample.html）。这是由服务器机器上作为操作系统一部分的文件系统完成的。若文件能找到并可正常读取，则服务器将把它返回给客户，如图 1.6 所示。

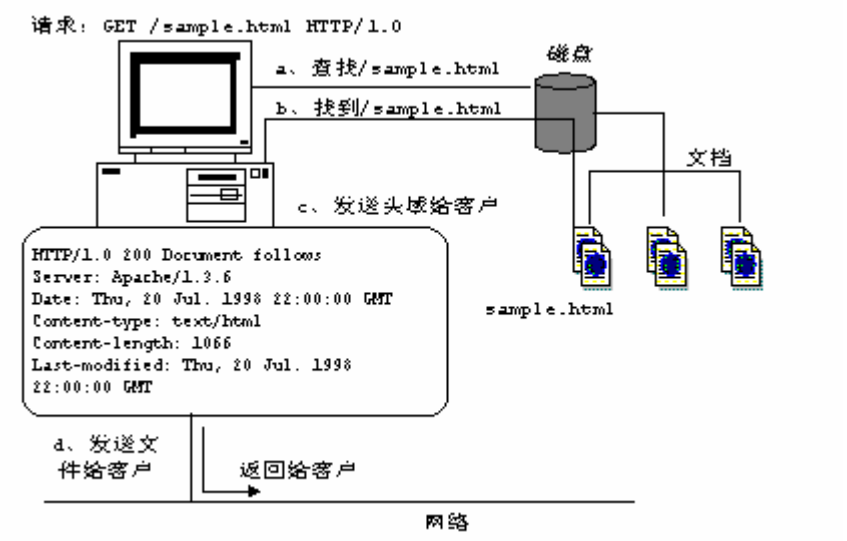


图 1.6

若成功：文件被发送出去。

首先，WWW 服务器发送一个状态码及一些描述信息。既然文件已找到，则发送状态码 200，表示一切都好（OK），文档随后发出。因为发送的信息是 HTML 文档，所以 Content-type 取值为 text/html。文档长为 1066 个字节，所以 Content-Length 取 1066。服务器软件的标识及文件的时间属性信息也被包含在头域中。如下所示：

```
HTTP/1.0 200 Document follows
Server: Apache/1.3.6
Date: Thu, 20 Jul. 1998 22:00:00 GMT
Content-type: text/html
Content-length: 1066
```

```
Last-modified: Thu, 20 Jul. 1998 22:00:00 GMT
```

在给出这些描述信息之后，WWW **服务器便从磁盘上读取该文件，并输出到网络端口上。**

若失败：返回错误指示。

若被请求的文件没有找到或找到但无法读取，则请求无法满足。这时将返回不同于 200 **的状态码。最常见的问题是请求中的文件名拼写有误，所以服务器无法找到该文件。例如误敲的**/smple.html **无法找到，因为本来就没有这个文件。这种情况下，服务器将发送一个状态码——**403**（没有找到）给客户，并没有其它的具体内容。这样的响应如下所示：**

```
HTTP/1.0 403 Not Found
Server: Apache/1.3.6
Date: Thu, 20 Jul. 1998 22:00:00 GMT
Content-type: text/html
Content-length: 0
```

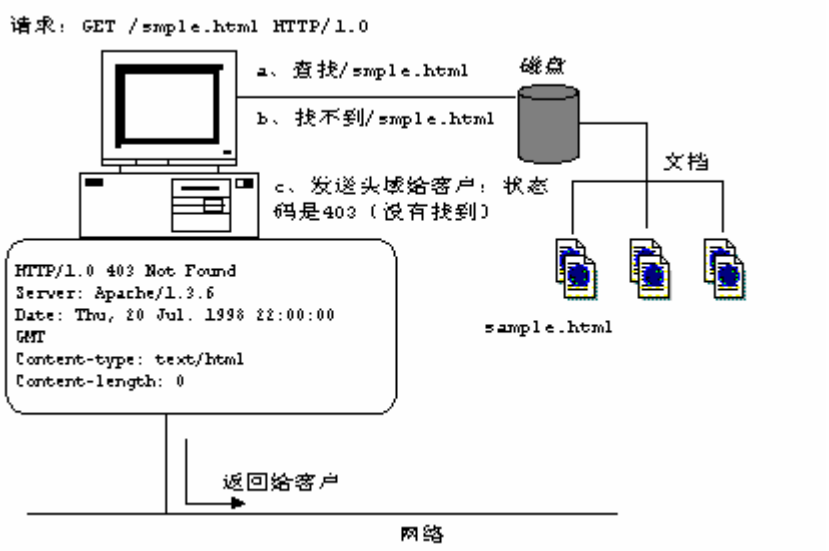


图 1.7

第六步：关闭文件和网络连接，结束会话。

当文件已被发出或错误已发出后，WWW **服务器结束整个会话。它关闭打开的被请求的文件，关闭网络端口从而结束网络连接。有关的其它工作则是由客户端来完成的，包括接收数据，并以用户可读的方式呈现出来。这些与服务器无关。**

第七步：转回到第一步。

WWW **服务器准备接收下一个请求，并不断重复上述的各个步骤。**

第4节　Proxy 服务器技术

因为 Apache **服务器不仅是** WWW **服务器，还是代理服务器，且代理功能为众多用户所关注，所以设置此节介绍代理服务器的基本原理。**

1.4.1. 概貌

在介绍代理服务器之前，先介绍一下什么是防火墙（Firewall）。在一个封闭的网络中，人们为了安全的考虑，不让这个网络内的机器直接与外界沟通，只允许它们先连到某一台主机，然后再由那台主机与外界交流。因为整个网络和外界是封闭的，只有防火墙和外界交流，所以若要防止它人破坏或盗取这个网络中的信息，只要把防火墙的安全性加强，整个区域网络的安全性也就变得较可靠了。即使这道防火墙被突破，那也只限于此，不能再深入到整个区域网络。

再说 Proxy，它是代理、委任的意思，在 WWW 世界的术语中，它指的是一个 HTTP 服务器，通常运行在设有防火墙的网络机器上，可以说是防火墙内网络中所有 WWW 客户端的代理人。当客户端想要某一篇文章或是网络上的资源时，它得先向代理服务器提出要求，再由后者将要求送出去。而外界的机器传回来的信息同样要通过代理这台机器才能转送给客户，所以代理服务器在有防火墙的网络世界中是很重要的。

那么对于没有防火墙的网络，代理服务器还有什么用处吗？答案是肯定的，且是非常重要的作用，表现在对资源的缓冲功能上。若有一台代理服务器上缓存了很多远程的资料，那么作为此代理服务器的客户，就不必大老远地再重新连到地球的另一端，辛辛苦苦地再把资料传回来，只要使代理服务器将它所缓冲的资料传给客户端即可。因为刚刚提到客户端发出请求时，会先连到代理服务器，如果我们将代理服务器设成具有缓冲功能，则它就会在每一次接到客户端的请求时去连接远程的机器，把那边的资料抓回来存在自己的硬盘上，再把这份资料传给客户。乍看之下好像不大对劲，这样不是反而多了一道手续吗？即要在代理服务器上存一份，又要再把它送给客户？其实若是这份资料在很远处，需要极长的时间才能传回来的话，那么这多存一份的代价就可以忽略。况且，好处在后头！其他的客户或许也想要这份资料，那么这个客户第一步当然是向代理服务器请求！代理服务器接到请求，首先检查一下看看自己的缓冲中有没有这份资料，而不必急着将这个请求往外发。若是所要求的资料刚好曾被请求并被缓冲存在硬盘上，那就不必再大老远地向远端服务器请求，直接将缓冲的副本送给客户即可；只有在请求的资料不在缓冲中时才需到远方去取。

另外还需提及的是既然代理服务器是客户的代理人，则它自然要懂得所有客户使用的协议以及所要求的资源类型，

总之，代理服务器的主要作用在于用户可以从内部穿过防火墙（如图 1.8 所示）进行网上冲浪。代理服务器是一种运行在防火墙（机器）上的特殊的 HTTP 服务器。它接收来自防火墙内部的请求，转发给外部的远程服务器，然后读取响应，并返回给客户。

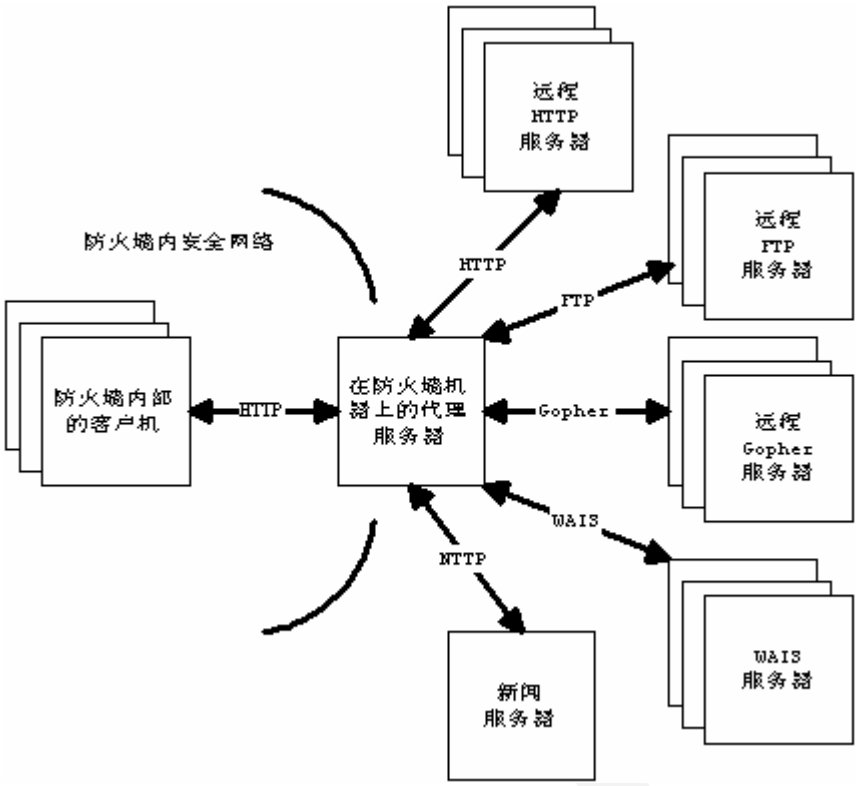


图 1.8

一般地，一个局域网中的用户共享一个代理服务器。用户相同的兴趣使他们会去访问相同的站点，这样就为代理服务器提供了进行缓冲的可能。这样代理服务器高效的缓冲机制，将可以明显地提高效率，降低带宽需求。因为重复的请求响应是从代理服务器的缓冲区中提取的，避免了进一步的远程访问，节省了带宽，又由于是本地网通信，节省了时间。

1.4.2. 应用级 Proxy

实际上，上面介绍的代理服务器应属于应用级 Proxy，即从 TCP/IP 协议栈的应用层实现代理的功能，如 HTTP、FTP、POP3 等。Apache 支持 HTTP 和 FTP 的代理。与之对应的还有 Socket 级 Proxy，即在 TCP 层上运行完成数据的转发，而不关心涉及的是何种应用协议和转发的数据是什么，此种代理不在本书讨论的范围内。

应用级代理服务器有着诸多优点。

应用级代理服务器能使内部网在防火墙的保护下安全工作，访问网上资源；同时，又不给黑客留下任何安全漏洞。

对 Web 客户来说，要支持代理服务器来访问 Internet，仅是几分钟的事，甚至更短。

应用级代理服务器可以同时支持多种应用层协议和在应用层上进行一些处理,如协议转换等。它可以使客户端收到的 FTP 和 Gopher 的信息能以一致的方式显示出来，而无须同时使用多个分离的 FTP、Gopher 和 WAIS 等客户程序，避免了用户去熟悉客户程序各自不同的风格。它使客户程序员不必再犯愁为支持所有的网络协议而去编写上万行代码。他们完全能集中精力做其它的事情。有了代理服务器，客户程序可能只需支持 HTTP 协议，其它的协议则由代理服务器来处理。因为客户程序的 FTP、Gopher 和其它的 Web 协议的请求均可以映射成 HTTP 协议形式，而由代理服务器再以各种协议与远程网络通信（请参阅图 1.11 中的 HTTP 与 FTP 之间的协议转换）。

有了代理服务器，无法使用 DNS 服务的客户也可以上网冲浪了。你需要知道的唯一的东西是代理服务器的 IP 地址。一些单位的内部网中使用如 10.*.*.*形式的私有地址空间，但仍然可以访问外部的世界，只要代理服务器对内对外均是可见的即可。

代理服务器支持网络应用层上客户访问的日志记录，包括客户的 IP 地址、访问日期和时间、URL 地址、传输的数据量和响应代码。在 HTTP 事务中涉及的任何头域和元信息均可以被选择记录。这一点要在 IP 或 TCP 层上是不可能实现的。

代理服务器还支持应用层上的对客户请求的过滤。它可以基于访问方法、客户主机或远程服务器的地址或所在域进行访问控制。

应用层代理服务器机制便于进行信息的缓冲。把缓冲区设在公共的代理服务器上比分散在各个客户端要高效得多。这不仅体现在节省磁盘空间和对外的数据流量上，更重要地，代理服务器可以进行数据挖掘，基于大量客户的兴趣对被访问文档进行统计分析，总结出哪些文档应该保存时间长，哪些则应保存得短一些。使用该代理服务器的客户越多，结果越准确，越贴近实际。

缓冲还使用户在外部的原始服务器已关闭时仍能进行网上冲浪，只要该用户可以连到代理服务器上即可。这一点对于一些非常繁忙的 FTP 或 Gopher 站点来说意义非凡。你甚至可以构建一个缓冲区来实现一个非常慢的网络站点或虚构站点的映象，或者你还可以一下子下载大量的文档到缓冲中，然后把机器带到一个非常舒适但没有网络的地方，细细品味。

1.4.3. 若干技术问题

在应用级代理服务器中，要着重解决以下几个技术问题。读者可以在在 Apache 服务器中知道其解决方案。

连接和角色转换

客户发送一个正常的 HTTP 请求时，HTTP 服务器得到的 URL 地址中仅包括路径和可能的 CGI 参数部分；其他的部分，如协议头“ http: ”和主机名，则被省去（如图 1.9 所示）。显然客户一定知道该服务器所在主机的地址，也知道它是一个 HTTP 服务器。不然该客户就不会将请求送到该服务器上了。请求的路径表示了要访问的文档和 CGI 脚本，或者服务器上的其它资源。

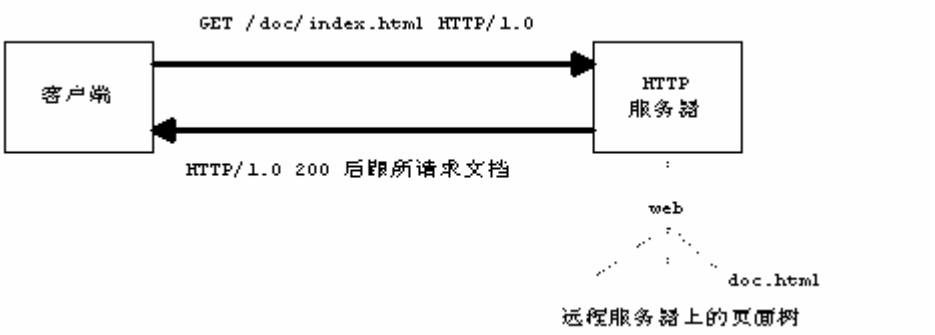


图 1.9

客户给代理服务器发请求则不同。客户不能仅给出路径和可能的参数，而是应提供一个完整的 URL 地址（如图 1.10 和图 1.11 所示）。这样代理服务器才有足够的信息给实际的服

务器发送请求，该服务器的地址和要使用的协议就从客户提供的 URL 地址中获取。

代理服务器既是服务器也是客户。当它接收客户的 HTTP 请求时，以服务器的身份出现；而当它为客户再到远程服务器上获取信息时，又变成了客户。代理服务器从远程服务器获得的信息原封不动地发送给客户，所以使用代理服务器不会带来任何的信息丢失。

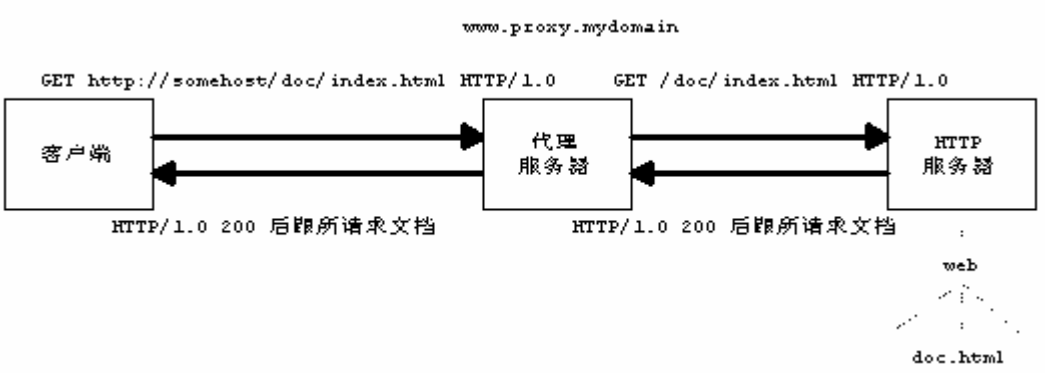


图 1.10

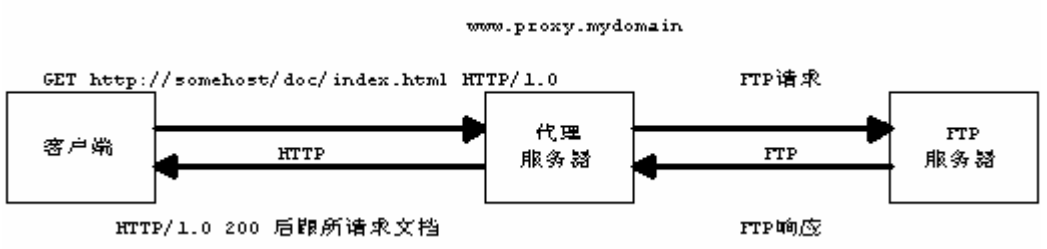


图 1.11

缓冲

缓冲的基本思想非常简单：将获得的文档变成本地文件以便将来再用，这样当该文档被重复请求时，就不必在到远程服务器上去取了（如图 1.12 和图 1.13 所示）。

但是缓冲并非这么简单，还有许多问题需要处理：缓冲的文档应该保存多久？如何确保文档不是过时的呢？如何确定对那些文档进行缓冲呢？

HTTP 协议中定义了一个头域来指定文档的过期时间（请参阅 HTTP 协议一章中的相关内容）。但是，现在很有一些服务器在 HTTP 应答中不支持它，代理服务器必须依据其它的东西来确定过期时间，或甚至仅给出一个大致估计。

即使再退一步，服务器提供了文档的过期时间。但问题解决了吗？需要注意的是，网上的许多文档都是经常变化的。可能某个文档长期没有改变，而突然哪一天又需要更改。这一点连文档作者本人都无法预计。因此也不能指望服务器提供完全准确的过期信息。

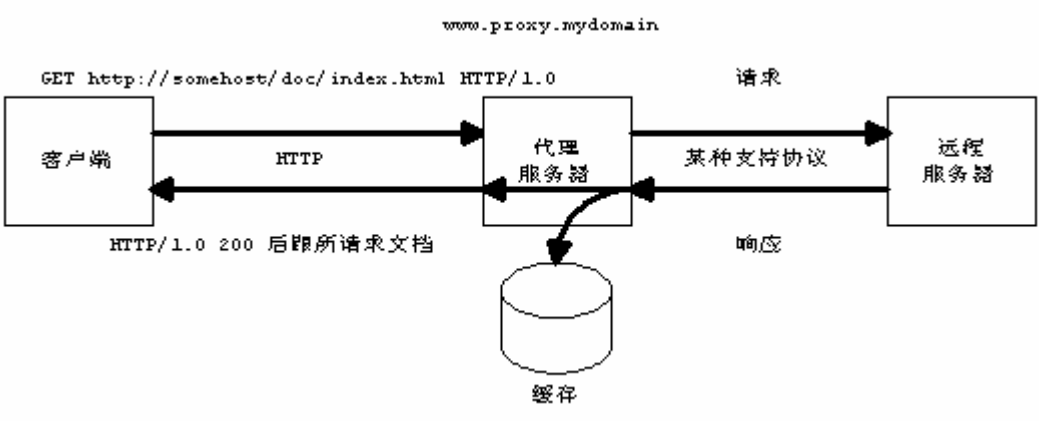


图 1.12

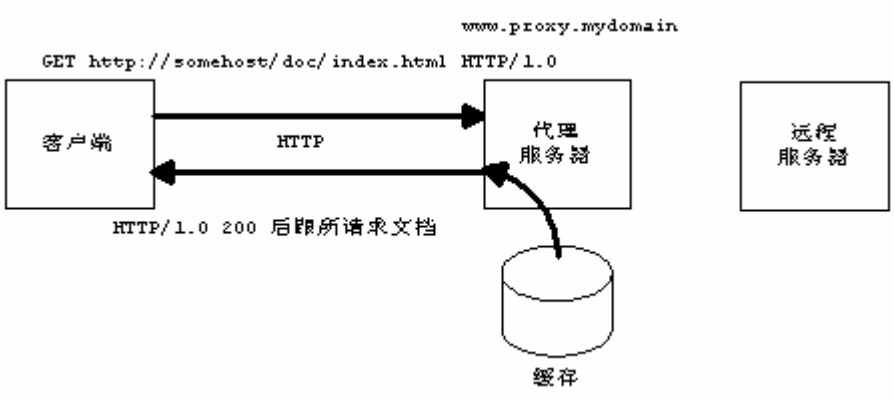


图 1.13

协议支持

要确保文档不过时，必须用 GET **方法来访问远程服务器。**HTTP **协议已经包含了** HEAD **方法来获取文档的头信息，而不必取得整个文档。这对于检查某个文档是否在上次访问后更改过是非常有用的。**

但当文档已更改过时，再进行一次 GET **请求来获取最新版本将使效率降低。因为建立一次连接的代价还没到可以忽略的程度。**

因此 HTTP **协议又增加一个** If-Modified-Since**（在** XXX **之后是否更改过）请求头，来使客户可以进行有条件的** GET **请求。新的请求与旧的相比增加一个** If-Modified-Since **头，其中给出请求的文档在本地的备份的最后一次更改的时间。**

若该文档在指定的时间之后，没有进行修改，则不必再返回之，而仅返回一些头信息和状态码。若文档已变化，则该请求将被认为与普通的 GET **请求没什么两样。**

有条件 GET **请求在许多情况是有用的。如一个大型的软件镜像站点，要保证内容有效，必须经常地更新其中的大量文件。有了** If-Modified-Since **条件将显著提高效率。**

有条件 GET **请求是向后兼容的。在** HTTP **协议中未知的头域是被忽略掉的。如果某个** HTTP **服务器不支持带条件** GET **请求，则也不会有什么损害，服务器仅是发送一个完整的文档而已。庆幸的是，现在绝大多数的服务器已支持了有条件** GET **请求。**

从下一章开始将具体介绍 Apache 服务器的使用方法及工作原理。

第2章 Apache 服务器简介

Apache 是由一些网络管理员自发组织起来志愿开发的一个 www-proxy 合一的服务器。它最初是基于 NCSA 的 httpd 1.3 服务器的源代码开发的，但现在版本的 Apache 代码已经完全重写过了。它现在在 www 服务器领域占的份额超过其他所有服务器的总和。

Apache 服务器比任何一个免费的服务器性能都要好；而与商业服务器相比，它又具有免费的巨大优势。运行 Apache 的某些站点在经受每天数百万次的点击时仍是岿然不动。免费可取的源代码又使用户可以自由自在地定制。正是这样两方面使得 Apache 性价比极高，而从众多竞争者中脱颖而出。

第1节 Apache 服务器的特点

Apache 服务器的优点可以总结如下：

- Apache 是 proxy 和 www 合一的服务器。可以在作为应用系统运行平台的同时，提供巨大的数据缓冲，有效地降低局域网用户访问 Internet 的带宽。
- Apache 是一个免费的软件，性价比极高。开发者的高水平使其有一个很高的起点，免费带来的广泛的用户反馈又使之不断改进和提高。
  Apache 的最新版本既支持 Windows 平台，也支持 UNIX 平台，安装方便，易于使用。
- Apache 是代码开放的且是模块化设计，为将来扩充成为各种应用系统提供了坚实的基础。

另外，对 Apache 的配置是通过文本格式的配置文件实现的。这样虽然方便高水平的用户灵活使用，但给普通用户却带来操作上的困难。这是 Apache 服务器的一点不足。希望本书的操作指南部分能给各位读者提供帮助。

第2节 对 WWW 服务器的调查

来自权威机构 Netcraft 对 Web 服务器的最新调查显示，尽管服务器领域新军频现，WebLogic、iPlanet 、AOLserver 等均在不断地攻城掠地，但 Apache 仍是处惊不乱，市场份额稳做龙头，见表 2.1。

表 2.1

服务器	2000.8	份额	2000.9	份额	变化
Apache	12222228	61.66	12705194	60.02	-1.64
Microsoft-IIS	3888590	19.62	4138551	19.55	-0.07
Netscape-Enterprise	1389991	7.01	1472689	6.96	-0.05
Zeus	396637	2.00	606987	2.87	0.87
WebLogic	484980	2.45	566509	2.68	0.23
Rapidsite	317198	1.60	332246	1.57	-0.03
AOLserver	540	0.00	201471	0.95	0.95
thttpd	222282	1.12	190371	0.90	-0.22
WebSitePro	102185	0.52	103617	0.49	-0.03
WebSTAR	90450	0.46	93185	0.44	-0.02

图 2.1 为近年来服务器份额的变化趋势图：

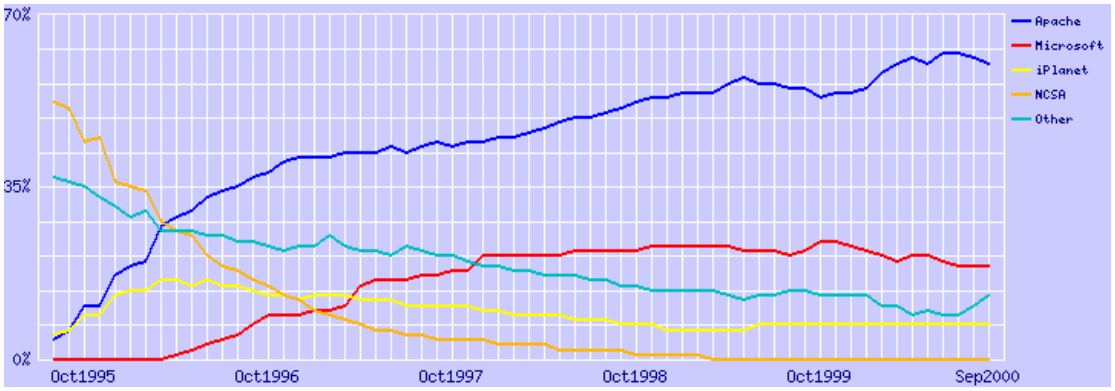


图 2.1

其中：

Planet 是 iPlanet-Enterprise、Netscape-Enterprise、Netscape-FastTrack、Netscape-Commerce、Netscape-Communications、Netsite-Commerce、Netsite-Communications 的总和。

Microsoft 是 Microsoft-Internet-Information-Server、Microsoft-IIS、Microsoft-IIS-W、Microsoft-PWS-95、Microsoft-PWS 的总和。

上述统计是 Netcraft 向各个服务器发送 HTTP 请求，从其响应中获取服务器种类标识完成的。

来自 ZDNet 的关于服务器性能的评测显示（http://www.zdnet.com/pcmag/features/webserver/bench.html）：几乎所有的服务器都可以很好处理对静态页面的请求，而在基于 CGI 的动态页面生成上则相差较大，其中 Apache 服务器与 Microsoft 的 Internet Information Server 表现最好。如下为各种服务器性能的对比示意图（图 2.2 为静态页面的响应性能，图 2.3 则为动态页面的响应性能）：

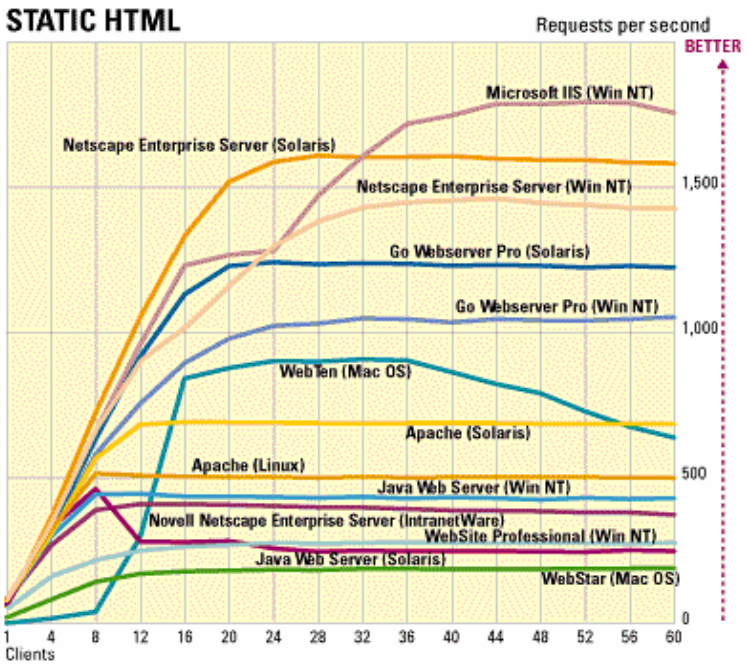


图 2.2

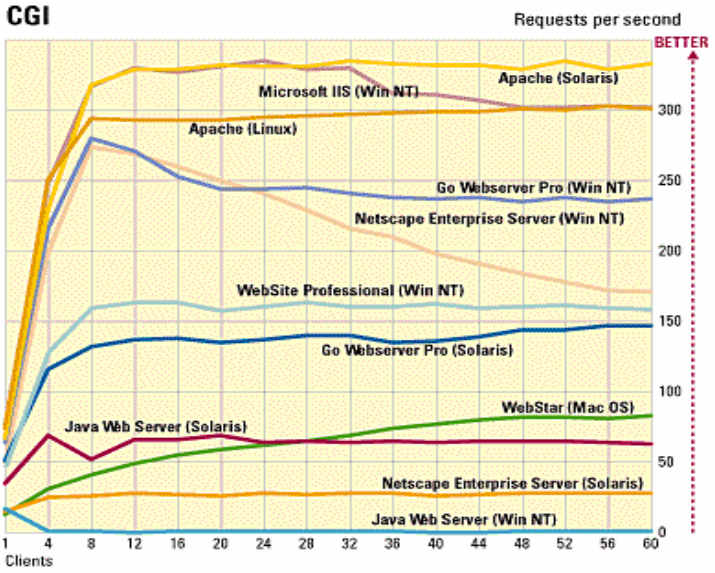


图 2.3

第3节　各流行 WWW 服务器的比较

下面对四种流行的 WWW **服务器做进一步详细的比较，从中基本可见** Apache **服务器的全貌：**

1. Apache

- **支持的操作系统：**
  NetBSD、Digital UNIX、BSDI、AIX、OS/2、SCO、HPUX、Windows NT、Linux、FreeBSD、IRIX、Solaris
- **价格：**
  免费
- **管理与安装：**
  可使用多个日志文件；
  支持虚拟主机；
  日志文件可自动更新清除或归档；
  可生成 referer **日志项；**
  可生成与点击无关的日志项（如注释）；
  CGI **脚本可生成自己的日志项；**
  可根据 IP **地址使用不同的根页面；**
  CERN/NCSA **兼容的日志格式；**
  可以 Windows NT **服务或普通应用的形式运行；**
  可基于 inetd**（对** Unix **和** OS/2 **系统而言）；**
  可监听多个地址和端口；
  基于点击的日志项格式可定制；
  日志可与 syslog (Unix)**或** Event Log (Windows NT)**相关联；**
  可生成浏览器日志项。

- **协议与 Include 功能支持：**
  支持 HTTP/1.1 持续连接；
  支持 HTTP/1.1 字节区间功能（byte ranges）；
  CGI 或其它脚本可访问服务器 状态变量；
  可基于 Accept 头域选择响应文档；
  支持 HTTP/1.1 的 PUT 功能；
  基于 HTML 注释支持 Include 功能；
  服务器可以强制加入页面片段；
  支持基于请求头域的 Include 功能；
  基于 User-Agent 头域选择响应文档；
  有内嵌 image-map 支持；
  支持 HTTP/1.1 请求的完全 URI 地址；
  对 If-Modified-Since 请求的响应；
  有内嵌的脚本语言；
  自动在响应中包含 HTTP 头域。
- **安全性：**
  基于域名的访问控制；
  基于 UID 的 CGI 脚本执行；
  基于 IP 地址的访问控制；
  基于用户及用户组的访问控制；
  在不重新启动服务器的情况下可以改变用户访问控制表；
  对目录结构的文档可进行结构化的访问控制；
  基于目录和文件的访问控制；
  可配置的用户组（不只单一用户组）；
  基于规则可隐藏文档的一部分；
  支持 SSL v. 2；
  支持 SSL v. 3；
  可以要求用户进行口令认证；
  支持基于 URL 地址的安全规则。
- **其它特征：**
  基于 GUI 的安装；
  基于输出媒体类型的不同脚本或动作支持；
  含有全部的源代码；
  基于 GUI 的维护；
  可作为 HTTP 代理服务器；
  有兴趣邮件组的支持；
  用户交互工具；
  支持多线程；
  支持非 CGI 形式的与 DBMS 的连接；
  自动进行目录列表；
  支持用户目录；
  搜索引擎；
  支持代理缓冲；
  远程维护。

2. AOL

- **支持的操作系统：**
  Digital UNIX、HPUX、Linux、IRIX、Solaris
- **价格：**
  免费
- **管理与安装：**
  可使用多个日志文件；
  支持虚拟主机；
  日志文件可自动更新清除或归档；
  可生成 referer **日志项；**
  CGI **脚本可生成自己的日志项；**
  CERN/NCSA **兼容的日志格式；**
  可基于 inetd**（对** Unix **和** OS/2 **系统而言）；**
  可监听多个地址和端口；
  可在日志中记录某个用户的行踪；
  日志可与 syslog (Unix)**或** Event Log (Windows NT)**相关联；**
  可生成浏览器日志项。
- **协议与** Include **功能支持：**
  支持 Windows CGI **接口；**
  支持 HTTP/1.1 **持续连接；**
  支持 HTTP/1.1 **字节区间功能（**byte ranges**）；**
  CGI **或其它脚本可访问服务器 状态变量；**
  有专门脚本处理不支持的方法；
  可基于 Accept **头域选择响应文档；**
  支持 HTTP/1.1 **的** PUT **功能；**
  基于 HTML **注释支持** Include **功能；**
  服务器可以强制加入页面片段；
  支持基于请求头域的 Include **功能；**
  基于 User-Agent **头域选择响应文档；**
  有内嵌 image-map **支持；**
  支持 HTTP/1.1 **请求的完全** URI **地址；**
  对 If-Modified-Since **请求的响应；**
  有内嵌的脚本语言；
  自动在响应中包含 HTTP **头域。**
  支持 Microsoft ISAPI**。**
- **安全性：**
  集成的授权服务器；
  基于域名的访问控制；
  基于 UID **的** CGI **脚本执行；**
  基于 IP **地址的访问控制；**
  基于用户及用户组的访问控制；
  支持 S-HTTP**；**
  在不重新启动服务器的情况下可以改变用户访问控制表；

对目录结构的文档可进行结构化的访问控制；
基于目录和文件的访问控制；
可配置的用户组（不只单一用户组）；
基于规则可隐藏文档的一部分；
支持 SSL v. 2；
支持 SSL v. 3；
可以要求用户进行口令认证；
支持基于 URL 地址的安全规则。

- 其它特征：
  基于 GUI 的安装；
  基于输出媒体类型的不同脚本或动作支持；
  基于 GUI 的维护；
  可作为 HTTP 代理服务器；
  有兴趣邮件组的支持；
  支持多线程；
  支持非 CGI 形式的与 DBMS 的连接；
  自动进行目录列表；
  搜索引擎；
  支持代理缓冲；
  远程维护。

3. Oracle

- 支持的操作系统：
  HPUX、Windows NT、Windows 95、Solaris
- 价格：
  商议
- 管理与安装：
  可使用多个日志文件；
  日志文件可自动更新清除或归档；
  可生成与点击无关的日志项（如注释）；
  性能日志；
  CGI 脚本可生成自己的日志项；
  CERN/NCSA 兼容的日志格式；
  可以 Windows NT 服务或普通应用的形式运行；
  可基于 inetd（对 Unix 和 OS/2 系统而言）；
  可监听多个地址和端口；
  基于点击的日志项格式可定制；
  可在日志中记录某个用户的行踪；
  日志可与 syslog (Unix)或 Event Log (Windows NT)相关联；
  可生成浏览器日志项。
- 协议与 Include 功能支持：
  支持 SNMP 代理；
  支持 Windows CGI 接口；
  支持 HTTP/1.1 持续连接；

支持 HTTP/1.1 字节区间功能（byte ranges）；
CGI 或其它脚本可访问服务器 状态变量；
支持 HTTP/1.1 的 PUT 功能；
基于 HTML 注释支持 Include 功能；
服务器可以强制加入页面片段；
支持基于请求头域的 Include 功能；
基于 User-Agent 头域选择响应文档；
有内嵌 image-map 支持；
支持 HTTP/1.1 请求的完全 URI 地址；
支持 Netscape Server API；
对 If-Modified-Since 请求的响应；
有内嵌的脚本语言；
自动在响应中包含 HTTP 头域；
支持 Microsoft ISAPI。

- 安全性：
  集成的授权服务器；
  基于域名的访问控制；
  基于 UID 的 CGI 脚本执行；
  基于 IP 地址的访问控制；
  基于用户及用户组的访问控制；
  支持 S-HTTP；
  在不重新启动服务器的情况下可以改变用户访问控制表；
  对目录结构的文档可进行结构化的访问控制；
  基于目录和文件的访问控制；
  可配置的用户组（不只单一用户组）；
  支持 PCT；
  支持 SSL v. 2；
  支持 SSL v. 3；
  支持 Set；
  可以要求用户进行口令认证；
  支持基于 URL 地址的安全规则。
- 其它特征：
  基于 GUI 的安装；
  基于输出媒体类型的不同脚本或动作支持；
  基于 GUI 的维护；
  可作为 HTTP 代理服务器；
  有兴趣邮件组的支持；
  用户交互工具；
  支持非阻塞的 DNS；
  支持多线程；
  支持非 CGI 形式的与 DBMS 的连接；
  自动进行目录列表；
  支持用户目录；
  搜索引擎。

4. Java

- **支持的操作系统：**
  OS/2、HPUX、Windows NT、Linux、Windows 95、IRIX、Solaris
- **价格：**
  若干百美金。
- **管理与安装：**
  日志文件可自动更新清除或归档；
  可生成 referer **日志项；**
  可生成与点击无关的日志项（如注释）；
  性能日志；CERN/NCSA **兼容的日志格式；**
  可以 Windows NT **服务或普通应用的形式运行；**
  可基于 inetd**（对** Unix **和** OS/2 **系统而言）；**
  可监听多个地址和端口；
  基于点击的日志项格式可定制；
  可在日志中记录某个用户的行踪；
  日志可与 syslog (Unix)**或** Event Log (Windows NT)**相关联；**
  可生成浏览器日志项。
- **协议与** Include **功能支持：**
  支持 SNMP **代理；**
  支持 HTTP/1.1 **持续连接；**
  支持 HTTP/1.1 **字节区间功能（**byte ranges**）；**
  CGI **或其它脚本可访问服务器 状态变量；**
  有专门脚本处理不支持的方法；
  可基于 Accept **头域选择响应文档；**
  支持 HTTP/1.1 **的** PUT **功能；**
  基于 HTML **注释支持** Include **功能；**
  服务器可以强制加入页面片段；
  支持基于请求头域的 Include **功能；**
  基于 User-Agent **头域选择响应文档；**
  有内嵌 image-map **支持；**
  支持 HTTP/1.1 **请求的完全** URI **地址；**
  对 If-Modified-Since **请求的响应；**
  有内嵌的脚本语言；
  自动在响应中包含 HTTP **头域。**
- **安全性：**
  集成的授权服务器；
  基于域名的访问控制；
  基于 UID **的** CGI **脚本执行；**
  基于 IP **地址的访问控制；**
  基于用户及用户组的访问控制；
  在不重新启动服务器的情况下可以改变用户访问控制表；
  对目录结构的文档可进行结构化的访问控制；
  基于目录和文件的访问控制；

基于规则可隐藏文档的一部分；
支持 SSL v. 3；
可以要求用户进行口令认证；
支持基于 URL 地址的安全规则。

- 其它特征：
  基于 GUI 的安装；
  基于输出媒体类型的不同脚本或动作支持；
  基于 GUI 的维护；
  可以支持其它的 TCP 协议；
  有兴趣邮件组的支持；
  用户交互工具；
  支持非阻塞的 DNS；
  支持多线程；
  实时性能测试工具；
  支持非 CGI 形式的与 DBMS 的连接；
  自动进行目录列表；
  支持用户目录；
  搜索引擎；
  远程维护。

从中可见，Apache 服务器在与同类服务器的比较中并不居下风，免费使用且附源代码，有此免费午餐何乐而不为？

下面的第 3、4 章将介绍 Apache 服务器的初步使用，包括如何安装、运行和访问等。

第3章 Apache 服务器的安装、运行和控制

第1节 Apache 服务器软件的安装

从 Apache 服务器的开发小组设立的站点 www.apache.org 上，可以获得有关 Apache 的详尽信息、软件和资料，既有软件源码供研究和编译使用，也有打好的安装程序包。用户可各取所需。

下面对 Windows 平台下的 Apache 服务器的安装和使用作详细讲解：

3.1.1. 基于 Apache 源码的编译和安装

编译是研究和使用 Apache 服务器源码的基础。编译 Apache 需要使用 Microsoft Visual C++ 5.0 开发环境（或更高版本）。有两种方法：

命令行

使用命令行工具（如 nmake 等）最为方便。具体做法如下：

首先将 Apache 软件包解到一个适当的目录中，然后打开一个命令行窗口，并进入 Apache 软件包的 src 子目录中。

Apache 项目的编译指令放在 Makefile.nt 文件中。要在 Windows NT 上编译 Apache，可在如下的命令中择一：

```
nmake /f Makefile.nt _apacher (release 版本)
nmake /f Makefile.nt _apached (debug 版本)
```

而要在 Windows 95 上编译 Apache（1.3.4 或更高版本），则使用如下方法：

```
nmake /f Makefile_win32.txt (release 版本)
nmake /f Makefile_win32_debug.txt (debug 版本)
```

上面的方法均可达到目的。但是后者（debug 版本）会在生成的文件中包含调试信息，这样可实现方便的查错和排错。

集成开发环境

Apache 也可以在 VC++的 Visual Studio 集成开发环境中编译。尽管操作不是太方便，但却可以容易地修改 Apache 的源码；且在命令行工具没有安装的情况下，这种方式更是不可缺少。

Apache 中各个部分的 VC 项目文件（.DSP）都含在了软件包中。在使用这些项目文件来编译 Apache 时，由于各部分相互之间有依赖关系，你需要遵照如下次序来进行：

1. os\win32\ApacheOS.dsp
2. regex\regex.dsp

3. ap\ap.dsp
4. main\gen_uri_delims.dsp
5. main\gen_test_char.dsp
6. ApacheCore.dsp
7. Apache.dsp

另外，在 src\os\win32 子目录中，包含有可选模块的项目文件。

安装

Apache 编译完成后，还要安装到服务器的根目录中。缺省的根目录是当前驱动器的\Apache 目录。

可以使用如下的方法之一来完成安装：

```
nmake /f Makefile.nt installr INSTDIR=dir (for release build)
nmake /f Makefile.nt installd INSTDIR=dir (for debug build)
```

对于 Windows 95 平台（Apache 应是 1.3.4 或更高版本），则使用：

```
nmake /f Makefile_win32.txt install INSTDIR=dir (for release build)
nmake /f Makefile_win32_debug.txt install INSTDIR=dir (for debug build)
```

其中的 INSTDIR 参数给出安装目录。没有时，默认是\Apache。

如上过程将安装如下内容：

dir\Apache.exe - Apache 执行程序
dir\ApacheCore.dll - 主要的动态链接库
dir\modules\ApacheModule*.dll - 可选的 Apache 模块（7 个文件）
dir\conf - 空的配置目录，用来存放 Apache 文本格式的配置文件
dir\logs - 空的日志目录，存放 Apache 日志文件

若你不使用 nmake，或你要将 Apache 装到另外的目录中，也一定使用上面的命名和分布约定。

在运行该服务器之前，你还必须填充 conf 目录：将*.conf-dist-win 文件从软件包目录树中拷贝到 conf 目录中，并改名为*.conf。将文件中的 ServerRoot 项数值改为当前的实际服务器根目录（如"C:\apache"）。同样地完成 conf/magic 和 conf/mime.types 文件的拷贝。

至此，Apache 服务器已完成安装，遵照下文的执行方法，可启动 Apache。

3.1.2. 基于 Apache 可执行代码的安装

首先从 www.apache.org 站点上获取安装程序包 APACHE_1_3_6_WIN32.EXE 或名称类似的其它版本的程序包。

然后，运行该程序包去安装 Apache。其间，会问及：

- 安装 Apache 的目录，缺省是\Program Files\Apache Group\Apache，当然你可以用另外的目录。
- Windows 启动菜单中对应菜单项的名字，缺省是“ Apache Web Server ”。
- 安装类型。"Typical"安装类型将安装除源码以外的一切东西；而 "Minimum"选择则不安装手册和源码。如果你要安装源码请选择"Custom"类型。

在安装时，Apache 将在你选择的安装目录的 conf 子目录中设置配置文件。但如果其中某一文件已经存在，它也并不会被覆盖掉，而是将要拷贝的文件的名字加上后缀.default。例如，若 conf\httpd.conf 已经存在，则不会有警告信息出现，而是仅是将本来要拷入的文件命名为 conf\httpd.conf.default。在安装完毕后，你可以手工地查看在.default 文件中有什么新内容，有必要时可以更新你已有的配置文件。

需要注意的是，若 htdocs\index.html 已经存在，则它不会被覆盖掉，也不会有 index.html.default 文件创建。

如上的处理使已安装的 Apache 能顺利地过渡到新版本。但是在安装前，你必须停止已有 Apache 服务器的运行，而在安装完成后，再重新启动之。

安装完成后，可以根据需要编辑 conf 目录中的配置文件。但不进行任何修改启动服务器运行也是毫无问题的。

具体的安装过程如下所示：

首先执行安装程序——APACHE_1_3_6_WIN32.EXE，会出现下列画面：

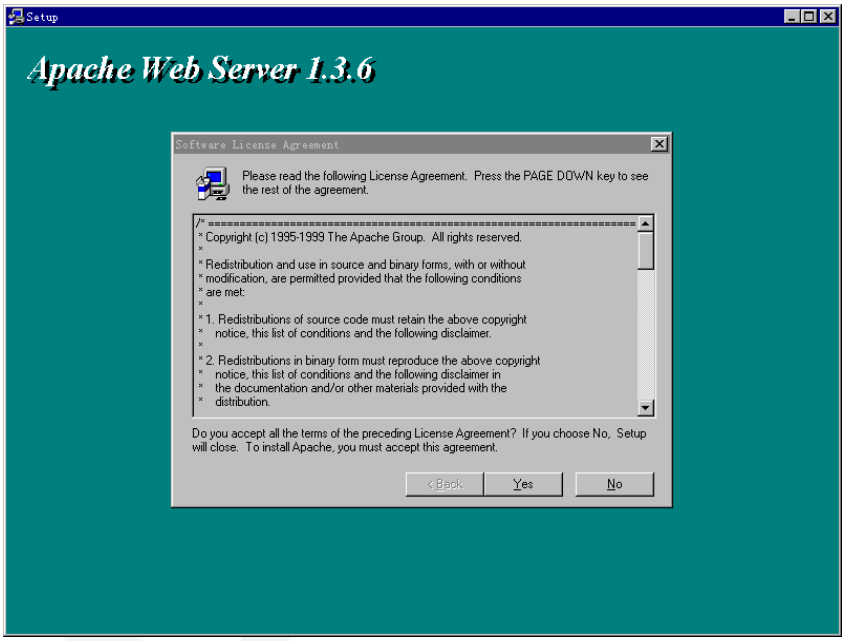


图 3.1

点击“ Yes ”，继续出现画面：

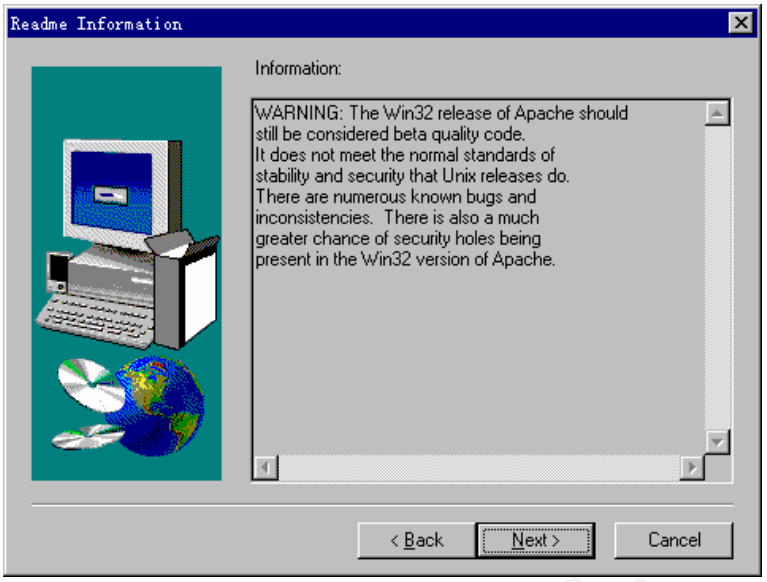


图 3.2

其中告诉用户 Apache 服务器在 Windows 平台下的版本尚不足够稳定，应认为是 Beta 版本。点击“ Next ”继续：

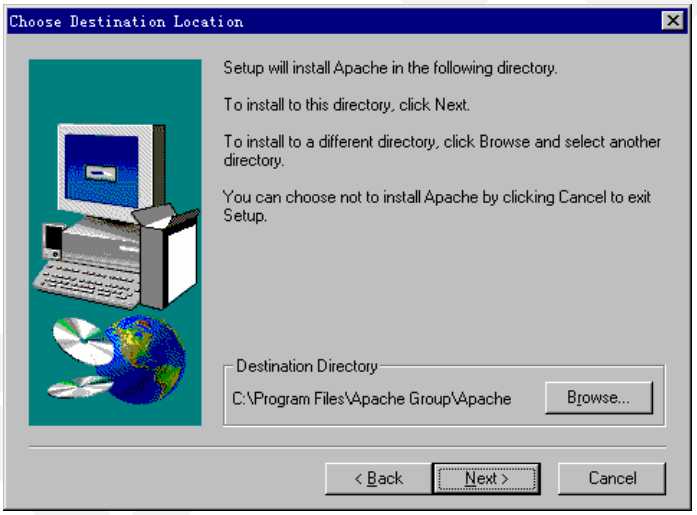


图 3.3

其中用户可以选择 Apache 服务器的安装目录，缺省情况下安装到“ C:\Program Files\Apache Group\Apache ”中。若用户要安装到其它目录，可以点击“ Browse ”，在出现的如下窗口中选择新位置：

图 3.4

在选好目标目录后点击“ OK ”，将返回先前的画面。继续点击“ Next ”出现如下画面：

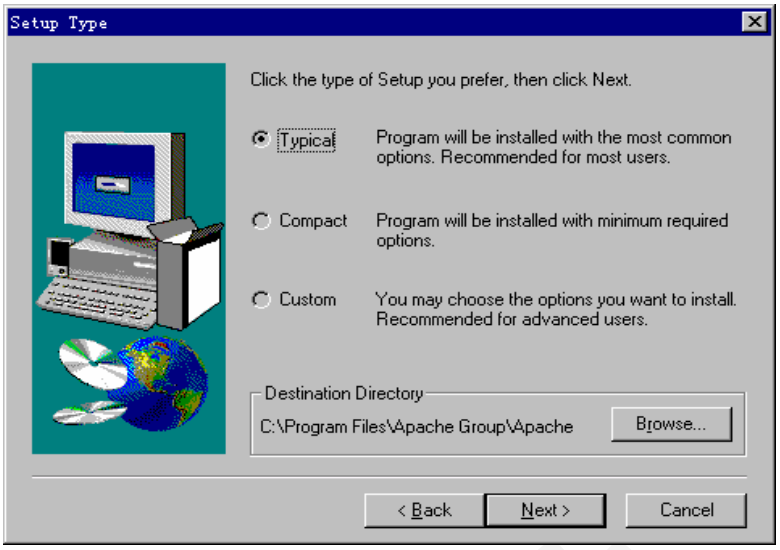


图 3.5

在此窗口中用户可以选择安装 Apache 服务器的方式：典型（Typical）、紧缩（Compact）和用户定制（Custom）。

Apache 服务器共有如下几部分：

- 应用程序——778K
- 手册——1313K
- 附加模块——130K
- 所需目录——2K
- 图标——44K
- 源代码——3392K

其中典型方式将安装常用的部分；而紧缩方式则不安装源代码；用户定制则使用户可以对构成 Apache 服务器的各部分进行选择安装。安装所有部分需大约 5661K 的空间。

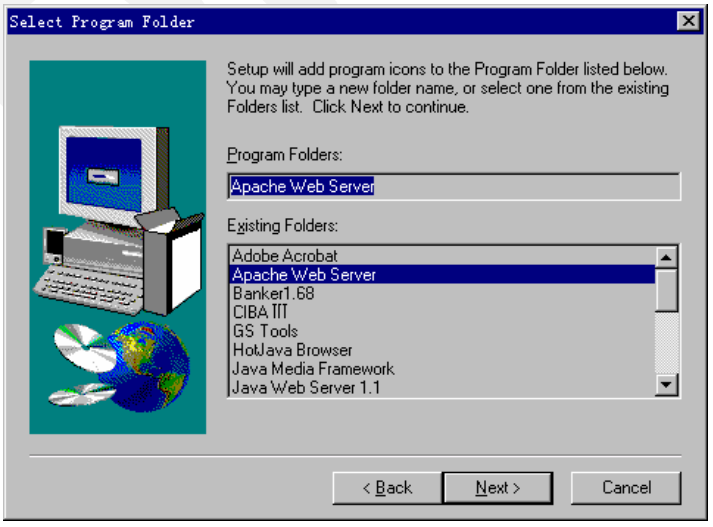


图 3.6

在选择好安装方式后，会提示如上画面：询问在 Windows 的启动菜单中增加的菜单项的名字，缺省值已给出。点击“ Next ”出现如下窗口提示正在准备安装：

图 3.7

稍等片刻后，即会开始安装。安装完毕，若出现问题，会给出提示。比如，你先前曾安装过 Apache，原来的配置文件仍然存在，则按照提示：原配置文件将保留。

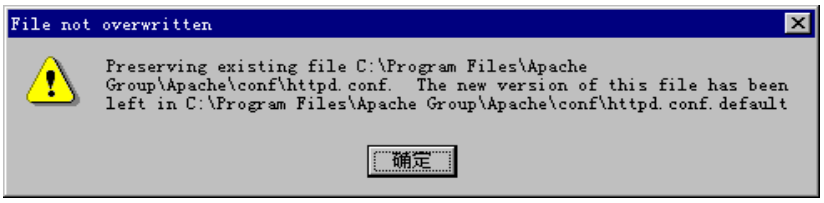


图 3.8

最后会提示，在使用 Apache 前应重新启动机器。重启机器后，即可使用 Apache 服务器了。

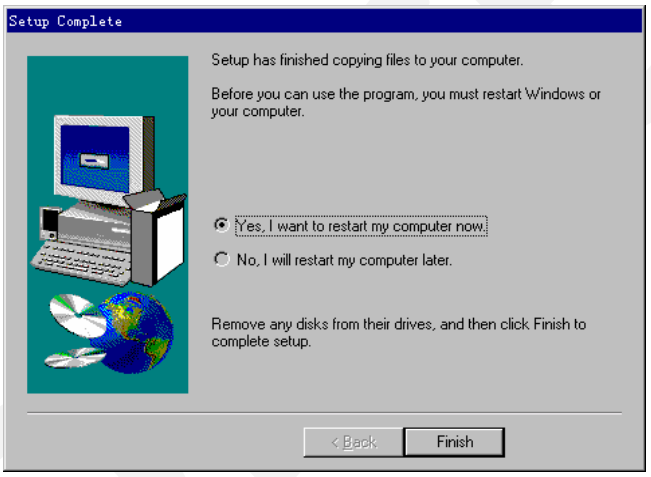


图 3.9

若选择立即重启，则系统将关闭并重新启动；若不想立即启动机器，则会出现最后一个画面提示：是否阅读 README 文件。在最后点击“ Finish ”按钮，系统整个安装完毕。

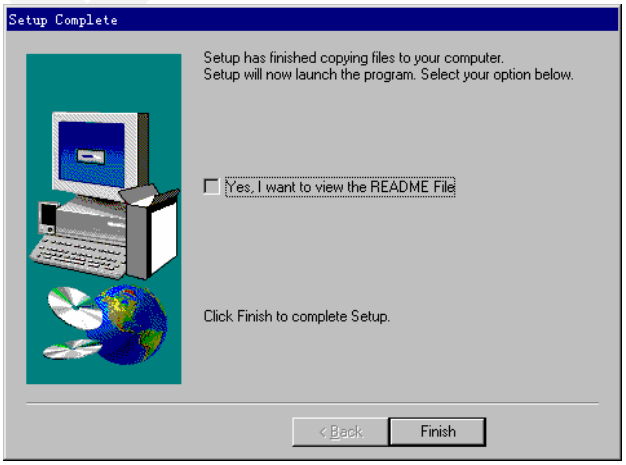


图 3.10

正常情况下，以控制台方式运行 Apache 服务器应出现如下的画面，其中的提示说明

Apache 服务器已正常工作。

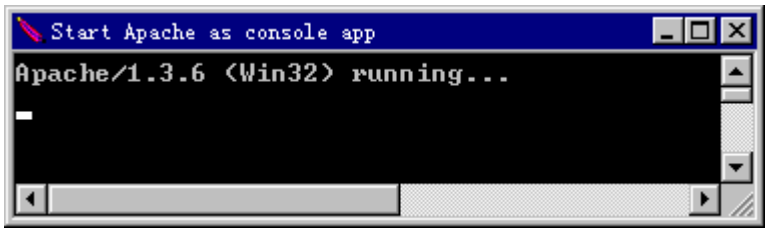


图 3.11

第2节　Apache 服务器的运行

在 Windows 平台下 Apache 有两种运行模式：

- 作为一个“服务”(只在 NT 或 2000 平台上)。如果你想在启动机器时自动启动 Apache 而在注销时还能保持运行，这是最好的选择。
- 控制台窗口模式。这是 Windows 9x 用户的唯一选择。

3.2.1. 服务模式

要以服务模式启动 Apache，你首先需要将它安装为“服务”。方法是在 Windows 启动菜单中使用“Install Apache as Service (NT only)”项。完成后，打开控制面板中 Services（服务）窗口，选择“Apache”，然后点击“Start”即可启动 Apache 服务。Apache 服务将在后台运行，你可以点击“Stop”来停止 Apache 的执行。

另外，你还可以使用如下的命令来分别启动和停止 Apache。

```
NET START APACHE
NET STOP APACHE
```

3.2.2. 控制台窗口模式

在控制台窗口模式下，又有两种方式：菜单方式与命令行方式。

菜单方式

要以控制台窗口模式运行 Apache，可以在启动菜单中选择“Start Apache as console app”项（在 Apache 1.3.4 或更早版本中，该项称为“Apache Server”)。这将会打开一个控制台窗口并在其中运行 Apache。该窗口将一直持续到 Apache 被停止掉。要停止 Apache，选择启动菜单中的“Shutdown Apache console app”项（在 Apache 1.3.4 或更早版本无此项），或者参阅第 3 章第 4 节部分来了解如何在命令行方式下控制 Apache。

在启动 Apache 之后（无论是控制台窗口还是服务方式），它将监听 80 端口（除非你改变了配置文件中的 Port、Listen 或 BindAddress 等配置命令参数)。要连接到该服务器上，并访问缺省的页面，需要在打开的浏览器中输入 URL 地址：

```
http://localhost/
```

正常时将会出现一个欢迎画面，其中含有指向 Apache 在线手册的链接。如果什么也没有发生，或有错误信息显示，请查看 logs 目录中的错误日志文件 error_log 确定问题的所在。

在 Apache 已正常运转的情况下，你可以进一步地编辑 conf 目录中的文件来更改某些配置以满足自己的特定需求。

命令行方式

尽管 Windows 下的“开始”菜单按钮和 NT 或 2000 下的服务管理器可以给用户提供一种简单的管理 Apache 服务器的界面，但是在某些情况下命令行方式却是更好的选择。

要在命令行方式下以控制台窗口模式启动 Apache，应使用如下命令：

```
apache -s
```

这种情况下，终止 Apache 要使用 control-C。（在 Windows 9x 平台下，-s 参数是不需要的，但是在 Windows NT 或 2000 下，它会使 Apache 不再去检查是否有 Apache 服务在运行。

你可以用如下命令来将 Apache 安装为 Windows NT 或 2000 下的服务：

```
apache -i
```

删除服务则用：

```
apache -u
```

第3节 Apache 配置文件的查找

要使用 Apache，我们必须了解它是如何查找配置文件的。Apache 查找的次序如下所示：

-C 参数开关设置的 ServerRoot 值；

命令行下的-f 开关指定的参数；

命令行下的-d 开关指定的参数；

注册项（在直接用安装程序安装 Apache 时存在）；

编译进 Apache 代码中的服务器根目录的值。

最后者的值通常是"/apache"。使用 -v 开关来启动 Apache 将显示标记为 HTTPD_ROOT 的该缺省值。

实际上，Apache 启动时的当前工作目录对 Apache 的行为没有任何的影响。

当使用“开始”菜单按钮或服务管理器时，Apache 通常不接收任何参数传递，这样使用注册项则是应该的选择了。

在直接使用安装程序安装时，会生成一个注册项，例如：

```
HKEY_LOCAL_MACHINE\Software\Apache Group\Apache\1.3.6\ServerRoot
```

这个键值已被编译进服务器中，使得您可以在不影响当前工作版本的情况下测试新的版本。当然，你必须保证不会使新版本覆盖掉旧版本在文件系统中的存在。你也不能在 Windows 下同时运行 Apache 的两个不同版本。

若你不是用安装程序安装的，Apache 有时可能会报告：找不到注册项。这一警告完全可忽略掉，如果 Apache 可以找到对应的配置文件的话。

上面的注册项的值指定作为 "ServerRoot" 的目录，该目录包含了存放配置文件的 conf 子目录。当 Apache 启动时，它会在该子目录中读取 httpd.conf 文件。若该文件中的 ServerRoot 配置参数指定了一个与如上注册项不同的值，Apache 取消原来的值而使用配置文件中的值。当你手工地将 Apache 目录或其配置文件移动到一个新位置时，你必须更新在

httpd.conf 文件中的 ServerRoot 配置参数的值。

第4节 以信号方式控制运行中的 Apache

Apache 支持用 Windows 中的信号机制给运行中的 Apache 进程施加控制。信号机制是内部的实现，对外部用户来说是看不到的，能看到的只是如何使用和效果。有如下两种使用方法：

3.4.1. 终止

在 Windows 下以控制台窗口方式运行的 Apache 可以用如上机制被终止。方法是：重新打开一个控制窗口，运行命令：

```
apache -k shutdown
```

注意：此种使用方法只适用于 Apache 1.3.3 或更高版本。对于早期的版本，你需要用 Control-C 来终止 Apache 服务器的运行。对于 Apache 1.3.3 或更高版本，鼓励使用前一种方式，因为它可以使 Apache 在“不慌不忙”中结束当前的事务处理，“体面”地结束，避免各种“垃圾”的遗留。

3.4.2. 重启

你还可以用发信号的方式告诉 Apache 重启。这可以使 Apache 重新读取配置文件。任何进行中的事务在结束前均不会被打断。具体的方法是运行如下的命令：

```
apache -k restart
```

注意：此种使用方法只适用于 Apache 1.3.3 或更高版本。对于早期的版本，你需要用 Control-C 来终止 Apache 服务器的运行。

对于熟悉 Unix 版本 Apache 的读者可能会想象：这些命令实际上提供了一种 Windows 下类似 kill -TERM pid 和 kill -USR1 pid 的方法。命令行参数-k，可以认为取 Unix 下的"kill"之意。

第5节 Apache 服务器的目录结构

在 Windows 下安装完 Apache 服务器后，在以"Apache Group\ Apache"为后缀的根目录中，有如下的目录布局：

<root>：有 Apache 服务器执行程序、核心模块动态链接库、readme 等文件。
|---bin：存放一些可执行的命令行工具，如生成口令文件的 htpasswd。
|---cgi-bin：存放用户设计的一系列 CGI 程序。
|---conf：存放系统配置文件的目录。
|---htdocs：对 Apache 服务器页面树的根，缺省情况下有 Apache 服务器的使用说明。
|---icons：存放一系列的指示文档类型的图标，当以列表形式显示目录内容时用到。
|---logs：存放生成的日志文件及服务器进程 IP 临时文件。
|---modules：存放动态链接库形式的各个 Apache 功能扩充模块，可经过配置被链入。
|---src：源代码目录，若安装时未选择安装则此目录不存在。

第4章 访问 Apache 服务器

对 Apache 服务器的使用，有两种使用方式：WWW 和 Proxy。可以只用前者，也可同时使用两种功能。

第1节　使用 WWW 服务

若是把 Apache 当作 WWW 服务器来使用，则客户端一般使用浏览器。用户只需在浏览器的地址栏中输入 Apache 服务器所在的域名或 IP 地址，即可进行访问。若是读者自己安装了 Apache 服务器来做测试，只需输入地址“ localhost ”即可出现 Apache 服务器的缺省根界面。如图 4.1 和图 4.2 所示分别为微软的 IE 浏览器及 Netscape 的 Communicator 浏览器，其中显示 URL 地址的位置即为地址栏。

若服务器不是使用缺省的 80 作为 HTTP 的监听端口，则输入的地址还需附带对应的端口号。若服务器要求进行认证，用户还需提供用户名及口令信息。

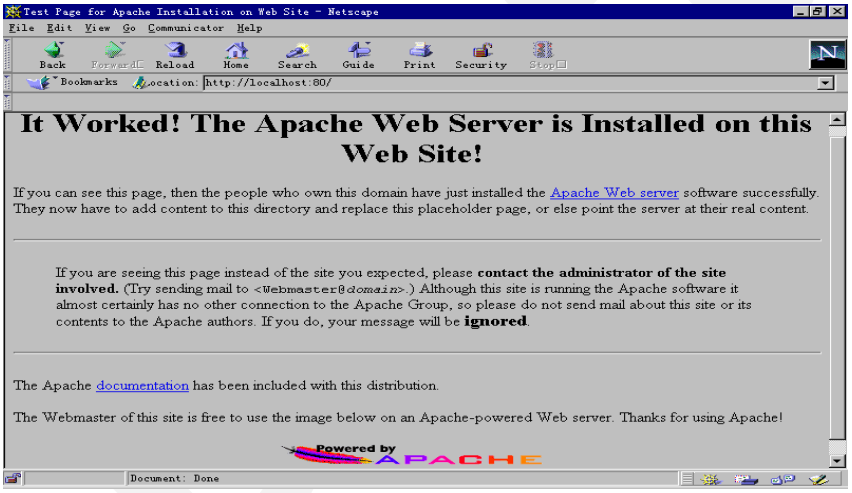


图 4.1

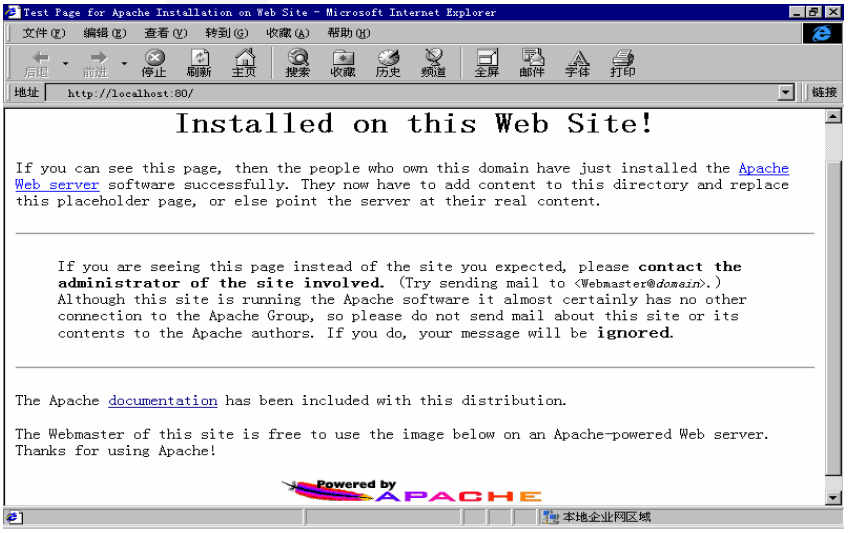


图 4.2

第2节　使用 Proxy 服务

虽然 Apache 是将两种功能合一的服务器，但 Proxy 功能并不是缺省就配置好的。若是您要使用 Apache 的代理功能，则还需要改动服务器的配置，具体请参阅第 7 章第 6 节。

对于客户端来说，您需要将 Apache 代理服务器的地址和端口号设置在浏览器中。对于，

- IE4.0

在主窗口中，点击“查看”菜单的“Internet 选项”项，将出现如下窗口：

图 4.3

再选择“连接”面板，则出现代理服务器的配置界面。

在“代理服务器”框中，设“通过代理服务器访问 Internet”为有效，这时下面的“地址”和“端口”项变为有效，在其中输入 Apache 代理服务器对应的 IP 地址和端口号即可。还可以点击“高级”按钮，进行较复杂的设置。

- Netscape Communicator4.0x

在主窗口中，选择“Edit”菜单的“Preferences”项，可出现如下的配置窗口：

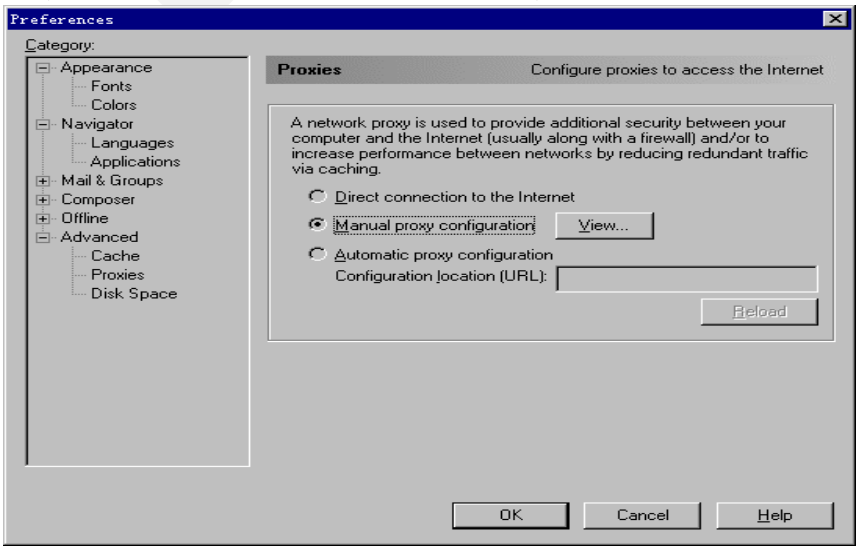


图 4.4

在左面的树型结构中，点击“ Advanced-->Proxies ”，右面可出现设置代理服务器的面板，其中有三种选择：直接连出去（不使用代理服务器）、手工配置和自动配置。

在已知道代理服务器的信息后，直接使用第二种选择。使之有效后，“ View ”按钮变为有效，点击该按钮可出现如下的具体配置窗口：

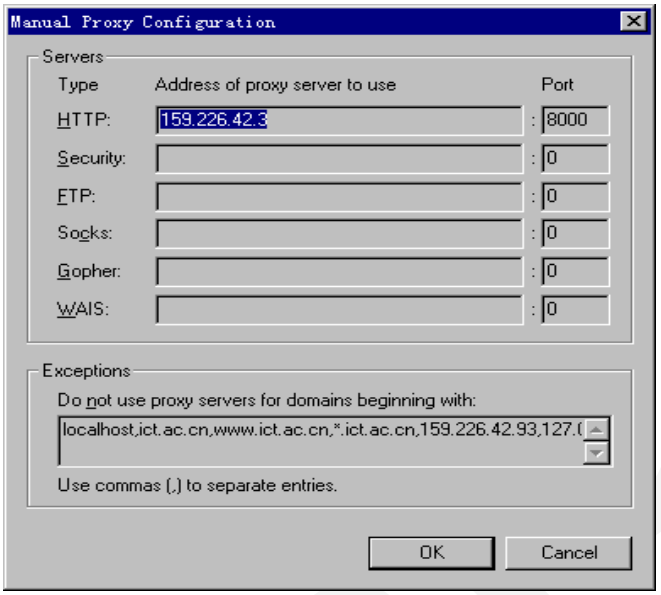


图 4.5

在该窗口中，你可以针对不同的协议使用不同的代理服务器。图 4.5 中设置以“ 159.226.42.3 ”作为面向 HTTP 的代理服务器，8000 为端口号。

设置好后，点击“ OK ”即可。

一般代理服务器都设置了认证机制，从而防止外部用户使用该服务。在基于这样的代理服务器访问某站点时，会出现如下要求提供认证信息的窗口：

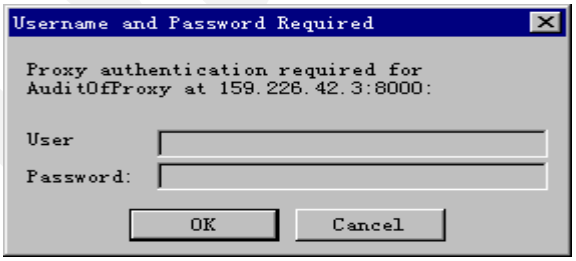


图 4.6

你只需给出正确的用户名及口令，即可通过认证，从而轻松冲浪。

下面的第 5 章和第 6、7 章为本书的核心部分，分别介绍 Apache 服务器的配置管理及实现原理。

第5章 Apache 服务器的配置

上面介绍了 Apache 服务器的安装和运行，如果正常的话，您应该是可以享受 Apache 的服务了。但是，若您有特殊的要求或想使 Apache 服务器发挥更大的作用，往往得自己更改配置信息。

Apache 服务器的配置是通过文本格式的配置文件来完成的，文件中包含逐条的配置命令对服务器的方方面面进行控制。

涉及的配置文件有：

conf/httpd.conf（主配置文件）：主要用来配置 Apache 服务器的基本参数，如服务器名、超时设置等，通过使用-f 命令行参数可以指定不同的主配置文件；

conf/srm.conf（资源映射配置文件）：包含服务器发送给客户（浏览器）的各种文档的格式说明（如使用什么字体等），使用 ResourceConfig 配置命令可以指定不同的资源映射配置文件；

conf/access.conf（访问控制文件）：包含对文档的访问控制配置，使用 AccessConfig 配置命令可以指定不同的访问控制文件。

上面所列的配置文件名是相对于 Apache 服务器的根目录的。

尽管任何一个配置命令可以出现在任何一个配置文件中，但为了清晰、条理，这里还是建议用户能分门别类地按上面提到的原则组织配置信息。实际上，Apache 中缺省地将全部配置都放在主配置文件中，其余两个文件为空。

第1节 配置命令说明

5.1.1. 注意事项

在对配置文件进行更改时，要注意如下规则：

- 使用 Unix 的路径法则：在所有路径设置中使用“ / ”代替 DOS 下的“ \ ”作为路径分隔符；
- 注释行以#开头，注释必须在一行结束，如果一行注释不下，下一行必须以#开头；
- 这些配置文件对大小写不敏感。但建议对非关键词均小写，对关键词使用匈牙利方法，如 ServerRoot、TypesConfig 等；
- 每行只准配置一个参数，配置的基本格式为：
  参数　　参数值
- 系统将忽略配置文件中多余的空白字符。

5.1.2. 格式说明

在详细介绍各个配置命令的使用之前，还需要将其使用的格式做一解释。

每个 Apache 配置命令均采用如下的格式进行定义：

Syntax（语法）：	directive-name（配置命令名）args（参数）
Default（缺省值）：	directive-name default-value（缺省值）
Context（允许的上下文环境）：	context-list（环境列表）
Override（覆盖控制）：	override（其作用是否可被覆盖的描述）
Status（状态）：	status（当前的状态描述）
Module（模块）：	module-name （所属模块名）
Compatibility（兼容性）：	compatibility notes（兼容性注意事项）

下面逐项进行说明：

Syntax

说明配置命令使用的格式。

Default

若某配置命令有缺省值（即若配置文件中未定义该命令，则 Apache 服务器将以此缺省值指定的行为进行工作），则该值应在此给出。若无缺省值，则给出“ None ”。

Context

本项指示当前配置命令可以出现在 Apache 服务器配置文件的何处。其值是一个以“ , ”为分隔符的包含一个或多个参数的列表。使用的参数有：

- server config：
  表示当前配置命令可以被用在服务器的配置文件中（即：httpd.conf、srm.conf 和 access.conf），但却不能出现在<VirtualHost>或<Directory>配置命令片段中。该命令也不能出现在.htaccess 文件中。
- virtual host：
  表示配置命令可以出现在服务器的配置文件的<VirtualHost>配置命令片段中。
- directory：
  表示配置命令可以出现在服务器的配置文件的<Directory>配置命令片段中。
- .htaccess：
  表示配置命令可以出现在目录的访问控制文件.htaccess 中。但该配置命令是否有效依赖于当前 overrides 配置是否有效。

配置命令只能在指定的环境中使用。若在其他地方使用，则会被告之配置出错。该错误会使服务器在该环境中工作异常，甚至根本无法工作。

一个配置命令可以使用的环境实际上是上面所列的环境的布尔或。比如：一个标记了"server config, .htaccess"的配置命令将可以在 httpd.conf 和.htaccess 配置文件中使用，但却不能在任何<Directory>和<VirtualHost>配置片段中使用。

Override

本配置命令属性用来指示当需要使.htaccess 文件中的配置命令生效时何种配置覆盖应是允许的。如果一个配置命令不能在.htaccess 文件中使用，则本属性应被说明成"Not applicable"（不适用）。

通过在某一范围（如一个目录）内使用 AllowOverride 配置命令，可以使配置覆盖生效，除非在更小范围内对 AllowOverride 配置又进行了修改。在本章第 2 节中，列出了 AllowOverride 配置命令中可以使用的覆盖参数。

Status

本属性指示当前配置命令与服务器的密切程度。比如，若某配置命令被标记为 Extension，则你需要先将对应扩展（Externsibal）模块编译进 Apache 才能使用该配置命令及其功能。本属性使用的参数值有：

- Core：
  表示当前配置命令属 Apache 服务器的核心部分，总是可以使用的。
- Base：
  表示当前配置命令属 Apache 服务器的标准模块，只要你不将对应模块从服务器中移去，该命令总是可用的。
- Extension：
  表示当前配置命令属 Apache 服务器的扩展模块。该模块被包含在 Apache 服务器软件包中，但却没有被编译进服务器执行程序中。要使用当前配置命令及其功能，你需要先将编译好的对应模块链入到服务器程序中，具体如何进行参阅本章第 3 节。
- Experimental：
  表示当前配置命令虽被包含在 Apache 软件包中，但你有权决定是否使用它。仅仅为了完整性才包含了该配置命令，而却未必支持之。支持该配置命令的模块或许被包含或不被包含在服务器中,你通过检查相关文档来了解某一配置命令究竟是否可用。

Module

本项仅用于列出定义了当前配置命令的源代码模块名。

Compatibility

若某配置命令不是在 Apache 的版本 1 中就有的话，需要在此处注明它被引入的版本。若某配置命令与 NCSA HTTPd 服务器的命令重名，则它们之间的任何差别也应该在这里列出来，否则应标上"No compatibility issues" （没有兼容性问题）。

第2节　配置命令介绍

下面分模块介绍各自处理的配置命令的格式及用法：

5.2.1. mod_core 配置（基本功能）

由 mod_core **模块处理配置命令，构成了** Apache **服务器的基本配置。它们包含：**

AccessFileName
AddModule
AllowOverride
AuthName
AuthType
BindAddress
ContentDigest
DefaultType
<Directory>
<DirectoryMatch>
DocumentRoot
ErrorDocument
ErrorLog
<Files>
<FilesMatch>
KeepAlive
KeepAliveTimeout
<Limit>
<LimitExcept>
LimitRequestLine
Listen
<Location>
<LocationMatch>
LogLevel
MaxClients
MaxKeepAliveRequests
MaxRequestsPerChild
NameVirtualHost
Options
PidFile
Port
require
ResourceConfig
ServerAdmin
ServerAlias
ServerName

```
ServerPath
ServerRoot
ServerSignature
ServerType
ThreadsPerChild
TimeOut
<VirtualHost>
```

AddModule 配置命令

```
Syntax: AddModule module module ...
Context: server config
Status: core
Compatibility: 只在 Apache 1.2 或更高版本中可用。
```

Apache 服务器可以编译进某个模块但却不激活它。本命令则正是用来激活这类模块的。Apache 中专门有一个要激活的模块列表，即 modules.c 中的 ap_preloaded_modules 数组，该表可以用 ClearModuleList 清除掉。

AccessFileName 配置命令

```
Syntax: AccessFileName filename filename ...
Default: AccessFileName .htaccess
Context: server config, virtual host
Status: core
Compatibility: 在且仅在 Apache 1.3 或更高版本中可以定义多个文件名。
```

本命令给出对用户访问进行控制的访问控制文件名，该文件仅对所在目录及其子目录所含文件的访问有效。当要传给客户一个文档时，若目录中允许使用访问控制文件，则服务器要去查看该文档所在路径的每一个目录中具有 AccessFileName 命令定义的名字的每一个文件，以确定是否允许客户访问。例如有如下定义：

```
AccessFileName .acl
```

则在返回文档/usr/local/web/index.html 之前，服务器会去读/.acl、/usr/.acl、/usr/local/.acl 和/usr/local/web/.acl 中的配置命令，除非它们已被使用如下方法禁止掉：

```
<Directory />
AllowOverride None
</Directory>
```

AllowOverride 配置命令

```
Syntax: AllowOverride override override ...
Default: AllowOverride All
Context: directory
Status: core
```

当服务器发现一个.htaccess 文件（由 AccessFileName 配置指定）时，它需要知道在该文件中定义的配置命令是否可以覆盖掉先前处理过的访问控制信息。

当将 Override 参数项指定为 None 时，服务器将不再去读此类文件；而指定为 All 则使服务器认为所有配置命令均是可覆盖的。Override 还可以是如下的取值：

- AuthConfig：
  允许身份认证方面的配置命令使用，包括 AuthDBMGroupFile, AuthDBMUserFile, AuthGroupFile, AuthName, AuthType, AuthUserFile, require 等。
- FileInfo：
  允许控制文档类型的配置命令使用，包括：AddEncoding, AddLanguage, AddType, DefaultType, ErrorDocument, LanguagePriority 等。
- Indexes：
  允许与目录内容列表相关的配置命令使用，包括：AddDescription, AddIcon, AddIconByEncoding, AddIconByType, DefaultIcon, DirectoryIndex, FancyIndexing, HeaderName, IndexIgnore, IndexOptions, ReadmeName 等。
- Limit：
  允许对客户主机地址进行访问控制的配置命令的使用，包括：allow, deny 和 order。
- Options：
  允许对特定目录特征进行控制的配置命令的使用，包括 Options 和 XBitHack。

AuthName 配置命令

```
Syntax: AuthName auth-domain
Context: directory, .htaccess
Override: AuthConfig
Status: core
```

本配置命令定义了某目录的身份认证域的名称。该名称将提示给用户使后者可以知道应该提供针对多个目标的多个用户名和密码对中的哪一个。AuthName 中使用一个参数，但若该域名中包含空格的话，它必须用引号括起来。本配置命令必须与 AuthType 和 require 命令，与 AuthUserFile 和 AuthGroupFile 等命令一起使用。

AuthType 配置命令

```
Syntax: AuthType type
Context: directory, .htaccess
Override: AuthConfig
Status: core
```

本配置命令为某目录选择用户身份认证的类型。当前只实现了 Basic 和 Digest 两种。它必须与 AuthName 和 require 命令，与类似 AuthUserFile 和 AuthGroupFile 的命令一起使用。

BindAddress 配置命令

```
Syntax: BindAddress saddr
```

```
Default: BindAddress *
Context: server config
Status: core
```

在操作系统的支持下，一个 http **服务器既可以监听与服务器主机的每一** IP **地址的连接，也可以只监听一个** IP **地址。参数** Saddr **可以取如下的值：**

- *
- **一个** IP **地址**
- **一个完整的** Internet **域名**

若取值为*，**则服务器将监听与服务器主机的每一** IP **地址的连接，否则它只能监听指定的** IP **地址。监听的端口由** Port **命令给出。**

BindAddress **配置命令只能使用一次。要想对** Apache **监听的地址和端口进行更多的控制，可以使用** Listen **配置命令。**

BindAddress **可被用来代替**<VirtualHost>**实现多个独立服务器的虚拟主机。**

ClearModuleList 配置命令

```
Syntax: ClearModuleList
Context: server config
Status: core
Compatibility: 只在 Apache 1.2 或更高版本中可用。
```

请参阅 AddModule **配置命令。**

ContentDigest 配置命令

```
Syntax: ContentDigest on|off
Default: ContentDigest off
Context: server config, virtual host, directory, .htaccess
Override: Options
Status: experimental
Compatibility: 只在 Apache 1.1 和更高版本中使用。
```

本配置命令可以使在 RFC1864（The Content-MD5 Header Field）**和** RFC2068（Hypertext Transfer Protocol -- HTTP/1.1）**中定义的** Content-MD5 HTTP **头域有效。**

MD5 **是一个计算任意长数据的“消息摘要（有时也称为指纹）”的算法。它的最大优点是源数据中的任何变化都将引起消息摘要的明显变化。请参阅第** 7 **章第** 9 **节。**

Content-MD5 **头域可以提供一种端到端的整个消息体的完整性检查（**Message Integrated Check**）。代理服务器或客户程序可以通过检查该头域发现消息体在传输过程中的变化。比如：有头域**

```
Content-MD5: AuLb7Dp1rqtRtxz2m9kRpA==
```

注意：由于每次请求都要进行计算，所以使用该头域会影响服务器的性能（该计算结果是不被缓冲的）；而且只有由 core **模块处理的文档才会计算** Content-MD5 **头域。比如** SSI **文档、**CGI **脚本的输出和以字节为单位的响应均不会有此头。**

DefaultType 配置命令

```
Syntax: DefaultType MIME-type
Default: DefaultType text/html
Context: server config, virtual host, directory, .htaccess
Override: FileInfo
Status: core
```

服务器经常会遇到客户请求一个由其 MIME 类型映射文件（缺省为 conf/mime.types，通过 TypesConfig 命令定义）无法确定其类型的文档。而服务器又必须告诉该文档的类型，这时它便使用 DefaultType 配置中给出的缺省类型。例如定义：

```
DefaultType image/gif
```

将可以很方便地给存放没有.gif 扩展名的 gif 图象文件的目录以文件类型的信息。

<Directory>配置命令

```
Syntax: <Directory directory> ... </Directory>
Context: server config, virtual host
Status: Core.
```

<Directory> 和 </Directory>用于封装一组配置命令，这些配置命令只应用于指定的目录及其子目录。任何可以在目录环境中使用的配置命令都可以在此出现。其中的 Directory 参数既可以是一个目录的完整绝对路径，也可以是一个包含通配符的字符串，这样的字符串中，`?' 匹配任何单个字符，而`*' 则匹配字符的任意序列。在 Apache 1.3 中，你还可以使用`[]' ，用法与 Unix shell 中的相同。而且在 Apache 1.3，没有任何一个通配符可以和`/' 匹配，这也完全是模仿 Unix shells 的用法。例如：

```
<Directory /usr/local/httpd/htdocs>
Options Indexes FollowSymLinks
</Directory>
```

在 Apache 1.2 及更高版本中，包含'~'字符的扩展正则表达式也可以使用。例如：

```
<Directory ~ "^/www/.*/[0-9]{3}">
```

将可以匹配在目录/www/下名字包含三个连续数字的任何子目录。

如果包含某文档的某目录与多个（非正则表达式类型）directory 配置片段匹配，那么它将遵循最短优先的原则进行配置的处理（即针对小范围——子目录的配置可覆盖掉先应用的针对大范围——父目录的配置），其间可能又夹杂一些来自.htaccess 的配置命令。例如与访问/home/web/dir/doc.html 文档相关的配置如下：

```
<Directory />
AllowOverride None
</Directory>

<Directory /home/*>
AllowOverride FileInfo
</Directory>
```

则处理的步骤是：

应用配置命令 AllowOverride None （将禁止.htaccess 文件）

应用配置命令 AllowOverride FileInfo（对于/home/web 目录）。

应用定义在/home/web/.htaccess 文件中的任何 FileInfo 类的配置命令。

在 Apache 1.2和 1.3版本中，对正则表达式类的 directory 配置的处理稍有不同。在 Apache 1.2 中，它们是和非正则表达式类的 directory 配置间隔开的，严格按照它们在配置文件中出现的次序。它们只在按照最短优先原则可以匹配时应用一次。而在 Apache 1.3 版中，正则表达式类的 directory 配置的优先级要低于所有的非正则表达式类的 directory 配置。它们内部之间则按照在配置文件中出现的次序进行匹配。例如有如下定义：

```
<Directory ~ abc$>
... 配置命令...
</Directory>
```

假设访问的文档是/home/abc/public_html/abc/index.html，服务器将依次考虑/, /home, /home/abc, /home/abc/public_html 和/home/abc/public_html/abc 等各级目录。在 Apache 1.2 中，当考虑/home/abc 时，该表达式将被匹配并应用。而在 Apache 1.3 中，处理/home/abc 式，此表达式则根本不被考虑。只有在所有的非正则表达式类的<Directory>配置和相应.htaccess 文件都已处理过后，才会获得考虑。该表达式将在处理/home/abc/public_html/abc 时获得匹配，并被应用。

注意：在<Directory />配置片段中缺省的访问配置是 Allow from All。这意味着 Apache 将会满足客户对任何文档的请求。建议将该配置片段改为：

```
<Directory />
    Order Deny,Allow
    Deny from All
 </Directory>
```

而在你希望被访问的目录上再覆盖掉此配置。

directory 配置片段照理应出现在 access.conf 配置文件中，但它们是可以出现在任何配置文件中的。<Directory> 配置不能嵌套，也不能出现在<Limit>或<LimitExcept> 片段中。

在 Apache 中除了<Directory>，<DirectoryMatch>、<Location>、<LocationMatch>、<Files>和<FilesMatch>片段均也可以包含应用于目录、URL 地址和文件访问控制的配置信息。各目录中的.htaccess 文件也可以包含对本目录的访问控制信息。那么同时定义的这些命令是如何作用的呢？

实际上，这些控制配置在作用之前，需进行合并，合并的次序是：

1. 不含正则表达式的<Directory>与被允许覆盖的.htaccess 的合并；
2. 合并<DirectoryMatch>和含正则表达式的<Directory>；
3. 合并<Files>和<FilesMatch>；
4. 合并<Location>和<LocationMatch>。

除了<Directory>配置是按其对应目录名长度从小到大处理的以外，其余均是按照它在配置文件中次序处理的。而当有多个<Directory>与某目录匹配时，也按照它们在配置中出现的次序处理。

配置文件之间是按照 httpd.conf、srm.conf、access.conf 的顺序处理的。对于 Include 引入的配置文件，则认为其处在 Include 所在的位置上。

在<VirtualHost>片段中定义部分则在主服务器配置之后处理，从而可以覆盖后者。

<DirectoryMatch>

```
Syntax: <DirectoryMatch regex> ... </DirectoryMatch>
```

Context: server config, virtual host

Status: Core.

Compatibility: 仅在 Apache 1.3 或更高版本中可用。

<DirectoryMatch>和</DirectoryMatch>用来封装一组配置命令，这些命令将应用于指定的目录及其各级子目录，在这一点上与<Directory>同。不同之处在于，它取一个正则表达式作为参数。例如：

```
<DirectoryMatch "^/www/.*/[0-9]{3}">
```

将与/www/目录中的任何名称包含三个连续数字的子目录相匹配。

DocumentRoot 配置命令

Syntax: DocumentRoot directory-filename

Default: DocumentRoot /usr/local/apache/htdocs

Context: server config, virtual host

Status: core

本配置命令指定服务器页面所在的目录。除非与 Alias 配置命令相匹配，服务器将总是把请求 URL 中的路径部分附加到此目录上来获得指向被访问文档的路径。比如定义了：

```
DocumentRoot /usr/web
```

则对 http://www.my.host.com/index.html 的访问将实际上访问/usr/web/index.html 文件。

在 mod_dir 模块中存在一个错误：当 DocumentRoot 配置中的目录结尾有'/'（比如"DocumentRoot /usr/web/"）时，会出现问题。请多加注意，避免之。

ErrorDocument 配置命令

Syntax: ErrorDocument error-code document

Context: server config, virtual host, directory, .htaccess

Status: core

Override: FileInfo

Compatibility: 在 directory 和 .htaccess 环境中的使用只在 Apache 1.1 或更高版本中有效。

在 Apache 服务器处理过程出现问题时，它可以根据配置采取如下的四种方法：

1. 输出一条有错误编号的消息。
2. 输出一条管理员指定的消息。
3. 重定向到一个本地 URL 来处理或指示该问题。
4. 重定向到一个外部 URL 来处理或指示该问题。

第一种选择是缺省的，而第 2-4 种则是需用 ErrorDocument 配置命令指定的。该命令中要给出一个 HTTP 状态码和一条消息或一个 URL 地址。

其中的消息要以"开头，"不作为消息的内容。Apache 对某些问题有时会给出一些额外的信息，指示发生了什么错误。

URL 地址可以以“/”开头，表示是本地的 URL 地址，或是一个完整的客户可解析的 URL 地址。例如：

```
ErrorDocument 500 http://foo.example.com/cgi-bin/tester
ErrorDocument 404 /cgi-bin/bad_urls.pl
```

```
ErrorDocument 401 /subscription_info.html
ErrorDocument 403 "Sorry can't allow you access today （抱歉您不能访问）
```

注意，当你指定一个远程的 URL 地址（即以"http'等访问方法开头的地址描述）作为 ErrorDocument 时，Apache 将发送给客户一个重定向应答，告之到何处可以访问到他所要的文档。需注意的是，对在同一服务器上的后者的访问也是有可能出现同一问题的。这有几种可能，其中最经常出现的的是“你使用了"ErrorDocument 401" 配置命令，而它又给出一个本地的文档进行重定向”。这是由 HTTP 的基本身份认证机制决定的。

ErrorLog 配置命令

```
Syntax: ErrorLog filename|syslog[:facility]
Default: ErrorLog logs/error_log (Unix)
Default: ErrorLog logs/error.log (Windows and OS/2)
Context: server config, virtual host
Status: core
```

ErrorLog 配置命令给出服务器用来记录错误信息的文件。若文件名不以“/”开头，则认为它是相对于 ServerRoot（服务器根目录）的。若文件名以管道标志“|”开头，则认为是将错误日志交给了一个单独的程序来处理。

在安全性方面，应确保无关人员对日志文件没有“读写”的权利，以免系统安全性受到损害。

请参阅： LogLevel。

<Files>配置命令

```
Syntax: <Files filename> ... </Files>
Context: server config, virtual host, .htaccess
Status: core
Compatibility: 只在 Apache 1.2 或更高版本中可用。
```

<Files> 配置命令提供了基于文件名的访问控制方式。它和<Directory> 配置命令和<Location> 配置命令相似，需要和</Files> 配置命令协同使用。在<Files>片段中出现的配置命令将对任何扩展名与指定名相配的文件有效。<Files>片段是按照它们在配置文件中出现的次序，在<Directory>片段和.htaccess 文件后被处理的，但在<Location>片段之前。注意<Files>片段可以嵌套在<Directory>片段内部来进行更细的控制。

filename 参数应该是一个文件名，或是一个通配符字符串。带有“~”字符的扩展正则表达式也可使用。例如：

```
<Files ~ "\.(gif|jpe?g|png)$">
```

将匹配绝大多数常见的 Internet 图象格式文件。在 Apache 1.3 或更高版本中，建议使用<FilesMatch>。

注意与<Directory>和<Location>片段不同，<Files>片段可以使用在.htaccess 文件中。这使管理员或文件所有者可以在文件一级控制客户对文件的访问。

<FilesMatch>

Syntax: <FilesMatch regex> ... </FilesMatch>
Context: server config, virtual host, .htaccess
Status: core
Compatibility: 只在 Apache 1.3 或更高版本中可用。

<FilesMatch> 配置命令提供了与<Files> 配置命令相同的基于文件的访问控制。但它只能使用正则表达式。例如：

```
<FilesMatch "\.(gif|jpe?g|png)$">
```

将匹配绝大多数常见的 Internet 图象格式文件。

<IfDefine>配置命令

Syntax: <IfDefine [!]parameter-name> ... </IfDefine>
Default: None
Context: all
Status: Core
Compatibility: 只在 Apache 1.3.1 或更高版本中可用。

The <IfDefine 条件表达式>...</IfDefine>片段用来定义有条件配置命令。只有当条件成立时，IfDefine 片段中的配置命令才会被处理；否则全部内容都被忽略。

< IfDefine >中的条件有如下两种形式：

```
parameter-name
!parameter-name
```

前一种情况下，只有当 Apache 启动时通过“ -D ”参数定义了名为 parameter-name 的参数时，片段中的命令才有效；后一种则完全相反。

<IfDefine>片段可以嵌套，从而可以实现简单的多参数测试。例如：

```
> httpd -DReverseProxy ...

# httpd.conf
<IfDefine ReverseProxy>
LoadModule rewrite_module modules/mod_rewrite.dll
LoadModule proxy_module   modules/libproxy.dll
</IfDefine>
```

<IfModule>配置命令

Syntax: <IfModule [!]module-name> ... </IfModule>
Default: None
Context: all
Status: Core
Compatibility: IfModule 只在 Apache 1.2 或更高版本中可用。

<IfModule 条件表达式>...</IfModule>片段用来定义有条件配置命令。只有当条件成立

时，IfModule 片段中的配置命令才会被处理；否则全部内容都被忽略。

<IfModule>中的条件有如下两种形式：

```
module-name
! module-name
```

前一种情况下，只有当 Apache 存在具有指定名字的模块时，片段中的命令才有效；后一种则完全相反。

module-name 参数应使用模块对应的源代码文件的名字，如“ mod_rewrite.c ”。

<IfModule>片段可以嵌套。

Include 配置命令

```
Syntax: Include filename
Context: server config
Status: Core
Compatibility: 只在 Apache 1.3 或更高版本中可用。
```

本命令可指定其它额外的配置文件，如 第 8 章第 2 节Java Servlet 部分 mod_jserv 模块配置文件的加入。

KeepAlive 配置命令

```
Syntax: (Apache 1.1) KeepAlive max-requests
Default: (Apache 1.1) KeepAlive 5
Syntax: (Apache 1.2) KeepAlive on/off
Default: (Apache 1.2) KeepAlive On
Context: server config
Status: Core
Compatibility: 只在 Apache 1.1 或更高版本中可用。
```

此配置命令提供对 Keep-Alive 配置的支持。

在 Apache 1.1 中，将 max-requests 参数设成你希望 Apache 在每次连接中处理请求的最大次数。该限制是为了使某个客户不会始终站用服务器的资源。将其设成 0 将关闭该功能。

在 Apache 1.2 及更高版本中，设置 "On" 来打开“ 持续连接 ”的支持，而用"Off" 关掉它。

请参阅 MaxKeepAliveRequests 配置命令。

KeepAliveTimeout 配置命令

```
Syntax: KeepAliveTimeout seconds
Default: KeepAliveTimeout 15
Context: server config
Status: Core
Compatibility: 只在 Apache 1.1 或更高版本中可用。
```

此配置命令给出 Apache 在关闭一个连接前对后续请求的等待时间。一旦请求处理完，

Timeout 配置命令指定的定时器即开始计数。

<Limit>配置命令

```
Syntax: <Limit method method ... > ... </Limit>
Context: any
Status: core
```

<Limit>和</Limit>用来封装一组配置命令来对指定的访问方法进行控制，该方法可以是任何合法的 HTTP 方法。其中可以使用除<Limit>或<Directory>外的任何配置命令。例如：

```
<Limit GET POST>
require valid-user
</Limit>
```

若一个访问控制配置命令出现在一个<Limit> 配置命令的外部，则它将应用于所有的访问方法。在<Limit>中列出的访问方法可以是 GET, POST, PUT, DELETE, CONNECT 或 OPTIONS 中的一个或多个。方法名是大小写敏感的。如果使用了 GET 方法，则 HEAD 方法也会受到限制。如果你想对所有的访问方法均进行控制，则你应该根本不使用<Limit>配置命令。

<LimitExcept>配置命令

```
Syntax: <LimitExcept method method ... > ... </LimitExcept>
Context: any
Status: core
Compatibility: 只在 Apache 1.3.5 或更高版本中可用。
```

<LimitExcept>和</LimitExcept>用来封装一组访问控制配置命令来应用于指定的 HTTP 访问方法之外的方法，即它完成与<Limit>片段相反的控制，可以被用来控制非标准的或不能识别的访问方法，对标准的自是不在话下。

LimitRequestLine 配置命令

```
Syntax: LimitRequestLine number
Default: LimitRequestLine 8190
Context: server config
Status: core
Compatibility: 只在 Apache 1.3.2 或更高版本中可用。
```

Number 参数是一个以字节为单位的在 0 到常量 DEFAULT_LIMIT_REQUEST_LINE（编译在服务器代码内部，现在取值 8190）之间的整数值。

LimitRequestLine 配置命令允许服务器管理员对客户程序的 HTTP 请求行长度进行控制。既然请求行由 HTTP 访问方法、访问的 URI 地址和协议版本组成，那么 LimitRequestLine 配置命令实际上是对请求的 URI 地址的长度进行了控制，因此该值应该足够大，以充分满足服务器上所有资源名称的要求，比如在 GET 请求中可能会传来的比较多的 query 参数。

此配置命令实际上提供给服务器管理员对异常的客户请求的更多控制，从而可以有效地

避免一些“拒绝服务”类型的攻击（denial-of-service attack，2000 年国际多家著名网站如 Yahoo、eBay、Amazon 等被攻击致瘫痪，其中黑客使用的就是此种攻击，即利用多台机器同时对某网站发送大量数据包，致使该网站资源被占而无法进行正常的服务）。一般地，完全可以使用缺省值，而不必更改。

Listen 配置命令

```
Syntax: Listen [IP address:]port number
Context: server config
Status: core
Compatibility: 只在 Apache 1.1 或更高版本中可用。
```

Listen 配置命令指示 Apache 监听多个 IP 地址或端口，缺省时它会对所有 IP 地址的请求进行响应，但只限于 Port 配置命令指定的端口。

Listen 可以用来代替 BindAddress 和 Port。它告诉服务器接收从指定端口或特定地址的指定端口来的请求。如果使用第一种格式，即只指定一个端口号，则服务器将监听所有网络接口的该端口，而忽略 Port 配置命令指定的端口号。若在给出端口号的同时还给出了一个 IP 地址，则服务器将只监听该地址的该端口。

Listen 配置命令可以出现多次，对每次定义的地址和端口，服务器都将进行响应。

如要使服务器接收端口 80 和 8000 上的连接，可使用：

```
Listen 80
Listen 8000
```

要使服务器可以接收两个特定地址和端口组合，应使用：

```
Listen 192.170.2.1:80
Listen 192.170.2.5:8000
```

<Location>配置命令

```
Syntax: <Location URL> ... </Location>
Context: server config, virtual host
Status: core
Compatibility: 只在 Apache 1.1 或更高版本中可用。
```

<Location> 配置命令提供对 URL 的访问控制。它与<Directory> 配置命令类似，标志一个配置片段的开始，结束则使用</Location> 配置命令。<Location>片段是按照它们在配置文件中出现的顺序来依次获得处理的，但排在所有 <Directory>片段和.htaccess 文件及 <Files>片段之后。

应当强调的是<Location>中定义的 URL 地址完全与本地文件系统无关。

对于所有原始的 HTTP 请求（非 proxy 请求），其中的 URL 地址是/path/形式的，不应带 http://servername 前缀。而对 proxy 请求，URL 地址则应是 scheme://servername/path 的形式，必须包含前缀。

URL 地址可以使用通配符。在通配符字符串中，用`?' 匹配任何单个字符，用`*'匹配字符的任意序列。

对 Apache 1.2 及更高版本：包含“~”字符的扩展的正则表达式也可使用，例如：

```
<Location ~ "/(extra|special)/data">
```

将与包含"/extra/data"或"/special/data"子字符串的 URL 地址匹配。在 Apache 1.3 及更高版本中，应使用新增的配置命令 <LocationMatch>来代替正则表达式类的<Location>配置片段的使用。

与 SetHandler 配置命令结合，将使 Location 配置发挥很大的作用。例如：为了允许查看服务器状态（status）的请求，而又想只对来自 foo.com 的用户开放，你可以用下面的方法：

```
<Location /status>
SetHandler server-status
order deny,allow
deny from all
allow from .foo.com
</Location>
```

斜杠“/”在 Apache 1.3 及更高版本中的用法是：斜杠“/”在 URL 地址串中的位置决定了它的含义。在文件系统中，多个连续的“/”将被缩成一个（即 /home///foo 与/home/foo 是相同的）。而在谈及 URL 地址时却未必如此。<LocationMatch> 配置命令和正则表达式版本的<Location>命令在处理你设置的多个“/”时会认为你就是想进行多个“/”的匹配。例如：<LocationMatch ^/abc>将匹配对 URL 地址 /abc 的请求而不会匹配对//abc 的请求。而非正则表达式类的<Location> 配置命令在处理 proxy 请求时规则相同，但当处理非 proxy 请求时它将隐含地将多个“/”与一个“/”相匹配。例如：<Location /abc/def> 将和对 /abc//def 的请求相匹配。

<LocationMatch>

```
Syntax: <LocationMatch regex> ... </LocationMatch>
Context: server config, virtual host
Status: core
Compatibility: 只在 Apache 1.3 或更高版本中使用。
```

<LocationMatch> 配置命令提供对 URL 地址的访问控制，使用方法和<Location>命令相同。不同的是，它以正则表达式为参数，而不是一个简单的字符串。例如

```
<LocationMatch "/(extra|special)/data">
```

将匹配任何包含子串"/extra/data" 或 "/special/data"的 URL 地址。

LogLevel 配置命令

```
Syntax: LogLevel level
Default: LogLevel error
Context: server config, virtual host
Status: core
Compatibility: 只在 Apache 1.3 或更高版本中使用。
```

LogLevel 控制在错误日志中记录的信息的详细程度（参阅 ErrorLog 配置命令)。可以设置的日志详细程度按从详到略的次序如表 5.1：

表 5.1

程度	描述	示例
emerg	Emergencies - system is unusable. 紧急 - 系统无法运行。	"Child cannot open lock file. Exiting" 子进程不能打开被锁定的文件。系统退出。
alert	Action must be taken immediately. 必须采取有效措施。	getpwuid: couldn't determine user name from uid getpwuid 过程无法从 uid 中确定用户的名字。
crit	Critical Conditions. 出现严重问题。	"socket: Failed to get a socket, exiting child" 无法创建 socket，子进程退出。
error	Error conditions. 出现错误。	"Premature end of script headers" 脚本的输出内容头不完整。
warn	Warning conditions. 出现警告错误。	child process 1234 did not exit, sending another SIGHUP 又给子进程 1234 发送一个 SIGHUP，但其仍没有退出。
notice	Normal but significant condition. 系统正常，但出现需注意的问题。	httpd: caught SIGBUS, attempting to dump core in ... 服务器捕获 SIGBUS 信号，正在将内存映象保存。
info	Informational. 提供建议信息。	Server seems busy, (you may need to increase StartServers, or Min/MaxSpareServers)... 系统好象正忙，你可能需要增加 StartServers 或 Min/MaxSpareServers 的参数值。
debug	Debug-level messages. 调试类信息。	Opening config file ... 正在打开配置文件...

当指定某一种程度时，在该程度之上的各种信息均会被报告。比如：当定义 LogLevel info 时，对 notice 和 warn 级的信息也将被报告。

建议至少使用 crit 级的日志记录配置。

MaxClients 配置命令

```
Syntax: MaxClients number
Default: MaxClients 256
Context: server config
Status: core
```

MaxClients 配置命令对同时可响应和处理的请求的最大数目进行控制，即创建的子服务器进程数目不会超过 MaxClients 配置命令指定的数目。要想同时可为多于 256 个客户服务，必须编辑 httpd.h 中的 HARD_SERVER_LIMIT 项的值，并重新编译。

当有连接需要服务，而子服务器进程已达到 MaxClients 指定的限制时，该连接请求将会进入一等待队列，该队列的最大长度是由 ListenBacklog 配置命令指定的。一旦有子进程处理完先前的请求，则等待的连接将会得到处理。

MaxKeepAliveRequests 配置命令

```
Syntax: MaxKeepAliveRequests number
```

```
Default: MaxKeepAliveRequests 100
Context: server config
Status: core
Compatibility: 只在 Apache 1.2 或更高版本中使用。
```

MaxKeepAliveRequests 配置命令用于对一次连接中处理的请求数进行限制，但只有当 KeepAlive 配置为 on 时，该限制才有效。如果该值被设为 "0"，则表示对请求数没有限制。建议设置的值高一些以便服务器的性能能充分地发挥出来。

MaxRequestsPerChild 配置命令

```
Syntax: MaxRequestsPerChild number
Default: MaxRequestsPerChild 0
Context: server config
Status: core
```

MaxRequestsPerChild 配置命令限制每个服务器子进程处理的请求的最大数目。

若某进程已处理完 MaxRequestsPerChild 指定数目的请求，则它要退出。但如果 MaxRequestsPerChild 的值为 0，则该子进程将永远不会由此退出。

将 MaxRequestsPerChild 设成一个非零值有两点作用：

- 它可以控制子进程由于偶然的空间丢失而消耗的内存（参阅第 6 章第 2 节和第 7 章第 3 节）；
- 赋给子进程一个有限的生命周期,将有助于在服务器负载下降时减少存在的子进程数目。

注意此配置命令在 Win32 平台下无效。

NameVirtualHost 配置命令

```
Syntax: NameVirtualHost addr[:port]
Context: server config
Status: core
Compatibility: 只在 Apache 1.3 或更高版本中可用。
```

NameVirtualHost 配置命令在构建基于域名的虚拟主机时使用。

尽管 addr 参数可以是主机名，我们还是建议你使用 IP 地址，例如：

```
NameVirtualHost 111.22.33.44
```

你在该 NameVirtualHost 命令中指定的 IP 地址就是解析对应的虚拟主机的域名而得到的 IP 地址。若你要在多个地址上建立多个基于名称的主机，可针对每个地址重复使用本配置命令。

注意：主服务器和任何的 _default_ 类服务器均不会对 NameVirtualHost 配置中指定的 IP 地址进行响应。（除非由于某种原因，你虽然定义了 NameVirtualHost 命令，但却没有为之给出对应的 VirtualHost 配置时，才非如此）。

你还可以对使用的基于域名的虚拟主机指定一个端口号，如：

```
NameVirtualHost 111.22.33.44:8080
```

请参阅本章第 3 节虚拟主机部分。

Options 配置命令

```
Syntax: Options [+|-]option [+|-]option ...
Context: server config, virtual host, directory, .htaccess
Override: Options
Status: core
```

Options 配置命令用于控制服务器的特征在某一目录上的继承。

option 参数可以设为 None。表示任何特征都被禁止。它还可有如下取值：

- All：
  除了 MultiViews 以外的所有特征都被继承。此是缺省的配置。
- ExecCGI：
  允许执行 CGI 脚本。
- FollowSymLinks：
  允许符号连接（symbolic links）的使用。
  注意 :尽管使用了符号连接,但与<Directory>片段进行匹配的路径仍用原来的名字。且如果此配置放在了<Location>片段中，则将被忽略。
- Includes：
  允许使用 SSI（Server-side includes）脚本。
- IncludesNOEXEC：
  允许使用 SSI（Server-side includes）脚本，但其中的 #exec 和 #include 命令禁止使用。
- Indexes：
  若请求的 URL 地址映射到一个目录上，且该目录中没有 DirectoryIndex 配置（象 index.html），则服务器将返回该目录内容的列表。
- MultiViews：
  允许内容协商（Content Negotiation）的 MultiViews。
- SymLinksIfOwnerMatch：
  服务器将仅允许这样的符号连接：连接和指向的目的文件和目录的所有者相同。
  注意：若出现在<Location>片段中，本配置将被忽略。

一般地，若对某个目录同时定义了多个 Options 配置命令，则实际采用的是限制最多的一个，即交集原则，而不是取其合集。但如果 Options 配置命令指定的参数有+ 或 - 开头，则将取合集。带 + 的范围将加到现在计算出的范围上，而带- 的范围将从当时的范围中删除。

一个没有 + 和- 符号的例子如下：

```
<Directory /web/docs>
Options Indexes FollowSymLinks
</Directory>
<Directory /web/docs/spec>
Options Includes
</Directory>
```

在该例子中，/web/docs/spec 目录将只认为设置了 Includes。但如果第二个 Options 配置命令中使用了 + 和-符号，即：

```
<Directory /web/docs>
```

```
    Options Indexes FollowSymLinks
    </Directory>
    <Directory /web/docs/spec>
    Options +Includes -Indexes
    </Directory>
```

则/web/docs/spec 目录将认为允许 FollowSymLinks 和 Includes。

注意：使用-IncludesNOEXEC 或 -Includes 将完全禁止 SSI 的使用，而不管先前是什么配置。

PidFile 配置命令

```
Syntax: PidFile filename
Default: PidFile logs/httpd.pid
Context: server config
Status: core
```

PidFile 配置命令指定记录服务器进程 ID 的文件。若文件名不以“/”开头，则认为是相对于 ServerRoot 路径的。该 PidFile 配置只在服务器的独立运行模式（standalone mode）下有效。

用户有时可能需要通知服务器，先关闭而后重新打开 ErrorLog 日志和 TransferLog 日志及重新读入经过修改的配置文件，以使之生效。这可通过给服务器进程发 SIGHUP 信号来完成，而需要的进程 ID 就可由 PidFile 获得，参阅第 7 章第 14 节信号控制部分。

PidFile 配置决定了进程 ID 的存放位置，从而与安全性是相关的。

Port 配置命令

```
Syntax: Port number
Default: Port 80
Context: server config
Status: core
```

number 参数是一个从 0 到 65535 的数字；一些端口号（尤其是低于 1024 的值）已被保留用于特定的协议。http 协议的端口号是 80。

Port 配置命令有两种用法（其中第一种使 Apache 与 NCSA 服务器保持兼容）：

- 在没有使用 Listen 或 BindAddress 配置命令指定监听的端口号时，在配置文件的主配置部分定义的 Port 命令（即在<VirtualHost> 片段之外）将可代行其功能。而当 Listen 或 BindAddress 配置命令指定了端口号时，Port 命令将在对应的地址上无效。
- Port 配置命令还用来设置 SERVER_PORT 环境变量（用于 CGI 和 SSI），且在服务器创建指向自身的 URL 地址时也用到（比如创建一个重定向到本地的 URL 地址）。此功能还受到 UseCanonicalName 配置命令的控制。

Port 配置永远不会对虚拟主机监听的端口有任何的影响，实际上 VirtualHost 配置命令就可用于端口的配置。

Port 配置的作用可以认为和 ServerName 配置命令相似。ServerName 和 Port 一同指定

服务器的规范地址，参阅 UseCanonicalName 配置命令。

Port 80 在 Unix 平台上是一个特殊的端口。所有小于 1024 的端口号均被保留给系统使用，即普通（非超级）用户不能使用它们，而只能使用更大的端口号。要使用 80 端口，你必须以超级用户的身份启动服务器。而在绑定到该端口后且在接收请求之前，Apache 将改成一个拥有较小权利的用户，该用户由 User 配置命令给出。

如果你不能使用 80 端口，可以选择任意的其他的未用端口。这样非管理员的用户将必须使用大于 1023 的端口，如 8000。

在安全性上请注意：当你以管理员身份启动服务器时，一定不要将 User 配置设为管理员。以管理员身份启动的服务器在处理连接时，将可能会遭到攻击。

require 配置命令

```
Syntax: require entity-name entity entity...
Context: directory, .htaccess
Override: AuthConfig
Status: core
```

此配置命令指定哪些经过身份认证的用户可以访问某目录。允许的语法为：

- require user 用户名 1 用户名 2 ...
  只有列出名字的用户才可以访问对应的目录。
- require group 组名 1 组名 2 ...
  只有属于指定组的用户才可以访问对应目录。
- require valid-user
  所有合法的用户均可以访问此目录。

如果 require 命令出现在<Limit> 片段内，那么它将只限制对指定的访问方法的使用，否则将对所有访问方法均有限制。例如：

```
AuthType Basic
AuthName somedomain
AuthUserFile /web/users
AuthGroupFile /web/groups
<Limit GET POST>
require group admin
</Limit>
```

Require 配置命令与 AuthName 和 AuthType 命令，与 AuthUserFile 和 AuthGroupFile（用来定义用户和用户组）等配置命令配合使用。

ScriptInterpreterSource 配置命令

```
Syntax: ScriptInterpreterSource 'registry' or 'script'
Default: ScriptInterpreterSource script
Context: directory, .htaccess
Status: core （只在 Windows 下有效）
```

本命令用来控制 Apache 1.3.5 及更高版本如何找到运行 CGI 脚本所需的解释器。本来缺省情况下是在脚本中用“ #! ”行指定脚本解释器的路径，而使用 ScriptInterpreterSource 设置

将使 Apache 去搜索 Windows 注册表来查询与脚本扩展名（例如.pl）相匹配的解释器。

ServerAdmin 配置命令

Syntax: ServerAdmin email-address
Context: server config, virtual host
Status: core

ServerAdmin 命令用于设置一个 Email 地址，该地址将在服务器遇到问题时提供给客户（包含在提供给客户的出错提示信息中）。

专门为此目的设置一个邮件地址可能是有益的，例如

```
ServerAdmin www-admin@foo.bar.com
```

因为不这样的话，用户给管理员发信将与其它信件混在一起，而某些粗心的用户可能根本没有说清楚：他在谈服务器的事！

ServerAlias 配置命令

Syntax: ServerAlias host1 host2 ...
Context: virtual host
Status: core
Compatibility: 只在 Apache 1.1 或更高版本中可用。

ServerAlias 配置命令用于为基于虚拟主机的服务器指定别名。

请参阅本章第 3 节的虚拟主机部分。

ServerName 配置命令

Syntax: ServerName fully-qualified domain name
Context: server config, virtual host
Status: core

ServerName 配置命令用于设置服务器的主机名，其只在构造重定向 URL 地址时用到。若不指定此设置，服务器将试图从它自己的 IP 地址解析出其域名；然而这可能无法可靠地工作，或者可能无法得到正确的主机名。比如实际的机器地址是 monster.wibble.com，则应定义：

```
ServerName www.wibble.com
```

ServerPath 配置命令

Syntax: ServerPath pathname
Context: virtual host
Status: core
Compatibility: 只在 Apache 1.1 或更高版本中可用。

ServerPath 配置命令用于在基于域名的虚拟主机下设置主机的根目录。

请参阅本章第 3 节虚拟主机部分。

ServerRoot 配置命令

```
Syntax: ServerRoot directory-filename
Default: ServerRoot /usr/local/apache
Context: server config
Status: core
```

ServerRoot 配置命令指定服务器所在的目录。该目录包含 conf/和 logs/子目录。在其它配置文件中使用的相对路径均是基于此目录的。

请参阅服务器的启动部分中 -d 参数的使用。

在 ServerRoot 路径上的访问权限的设置事关系统安全，需特别小心。

ServerType 配置命令

```
Syntax: ServerType type
Default: ServerType standalone
Context: server config
Status: core
```

在 Windows 平台下此命令没有意义。

ServerType 配置命令设置了服务器在操作系统中运行的方式,。有如下两种：

- inetd：
  服务器将附在 Unix 的 inetd 系统进程运行；启动服务器的命令需要加入到/etc/inetd.conf 文件中。
- standalone：
  服务器将以独立方式运行；启动的命令应加入到系统启动脚本文件（/etc/rc.local 或 /etc/rc3.d/....）中。

Inetd 是较少使用的一种情况。在该方式下，每接收到一个连接，便会启动一个完整的服务器。在处理连接完成后，该程序就退出。这样每个连接花费的代价非常高，但由于安全性上的原因，一些网络管理员总喜欢这种方式。这种方式将来不建议使用，因为服务器工作起来有时会不正常。应尽可能避免使用它。

Standalone 是最常用的一种 ServerType 设置，因为它非常地高效。服务器启动后，将为后来的所有连接提供服务。若你要用 Apache 构建一个繁忙的站点的话，standalone 方式将是你的最佳选择。

Satisfy 命令

```
Syntax: Satisfy 'any' or 'all'
Default: Satisfy all
Context: directory, .htaccess
Status: core
Compatibility: 只在 Apache 1.2 及以后版本中可用。
```

本配置命令用来表达 allow 和 require 配置命令同时被定义时访问控制的策略，可以使用'all'或'any'两种参数。

注意，本命令只在既用了用户名/口令对也用了客户端主机地址来控制对特定资源的访问时才用到。这种情况下使用缺省参数"all"表示用户要访问特定的资源必须不仅来自允许的主机地址，还要提供合法的用户名及口令。"any"参数则表示用户只需通过上述两项检查之一便可访问指定资源。

此命令可用来在基于口令方式控制对特定资源的访问的同时，允许来自特定网络地址范围的客户无需口令即可访问资源。

ThreadsPerChild

```
Syntax: ThreadsPerChild number
Default: ThreadsPerChild 50
Context: server config
Status: core (Windows)
Compatibility: 只在 Windows 平台的 Apache 1.3 或更高版本中可用。
```

此配置命令告诉服务器它可以创建多少个线程。这也是服务器同一时刻可以处理的最大数目的连接。在服务器被访问频繁时，务必将此值设得高一些。

此配置命令在 Unix 平台上无效。Unix 用户请参考 StartServers 和 MaxRequestsPerChild 配置命令。

TimeOut 配置命令

```
Syntax: TimeOut number
Default: TimeOut 300
Context: server config
Status: core
```

TimeOut 配置命令定义了 Apache 可以忍受的最长等待时间。对如下的三种时间计数：

1. 等待 GET 请求的时间。
2. 在 POST 或 PUT 请求时等待 TCP 包的时间。
3. 在发送 ACK 响应的 TCP 包之间的时间。

将来某个时候此配置将分离成三个独立的配置。在 1.2 版本之前缺省值取 1200，现在则是 300，即使如此仍是远远超过所需。之所以没有设那么低，是因为在代码中某些地方在发送一个数据包时可能忘记了将计数器重置。

UseCanonicalName 配置命令

```
Syntax: UseCanonicalName on|off
Default: UseCanonicalName on
Context: server config, virtual host, directory, .htaccess
Override: Options
Compatibility: 只在 Apache 1.3 或更高版本中可用。
```

在很多情况下，Apache 必须构造自引用的 URL，即仍指向本地原服务器的 URL 地址。在 UseCanonicalName 设为 on 时或在 1.3 前的所有版本中，Apache 会使用 ServerName 和 Port 的设置值来构造服务器的名字。该名字被用在所有自引用 URL 地址和用于 CGI 的 SERVER_NAME 与 SERVER_PORT 等环境变量中。

当 UseCanonicalName 设为 off 时，Apache 将用客户提供的主机名和端口号来构造自引用 URL（否则会使用标准 URL 地址）。当设置的是基于域名的虚拟主机时，两种情况下的值是相同的。CGI 变量 SERVER_NAME 和 SERVER_PORT 的构造也将使用客户提供的值。

下面的例子将有助于理解这一点：比如一个局域网服务器，用户要访问可以使用如 www 形式的简略的名字。你会注意到，若用户输入了一个简略的名字和一个表示目录的 URL 地址时，如 http://www/splat，其中没有以“/”结尾的话，Apache 将让用户重定向到 http://www.domain.com/splat/。如果你已经过了身份的认证，它会导致你必须再进行一次认证（一次针对 www ，一次针对 www.domain.com）。但如果 UseCanonicalName 被设为 off，则 Apache 会重定向到 http://www/splat/，从而不会出现上面的问题。

这里需要注意一点，即 CGI 脚本是否假设 SERVER_NAME 变量会受到如上问题的影响。若是，即主机名受到上述问题的影响，则客户要避免其出现，就必须自己在访问时提供一个完整的主机名。但如果 CGI 脚本仅把 SERVER_NAME 用于构造自引用的 URL 的话，则又不会有什么问题。

User 配置命令

```
Syntax: User unix-userid
Default: User #-1
Context: server config, virtual host
Status: core
```

User 配置命令给出服务器运行所基于的用户 ID。本命令只在 UNIX 平台下有效。

要使用此命令，独立运行（standalone）的服务器必须以管理员身份执行。unix-userid 参数可以取如下值之一：

- 用户名：用名字标识用户；
- #加数字：用数字标识用户。

该用户不能具有访问不应向外界公开的文件的权力，同时也不能具有执行与 HTTP 请求处理无关的代码的权利。这里建议您专门为执行 Apache 服务器新建一个用户组和用户。某些管理员喜欢在这里使用并不存在的用户，这也是不可取的。比如若在 mod_proxy 模块中使用了缓冲区，则此处的用户必须可以访问缓冲区对应的目录（参阅 CacheRoot 配置命令）。

注意，若你不是以管理员身份启动的服务器，则随后服务器将无法再变成以权力更小的用户身份执行，而是继续以原用户身份执行；而若你的确是以管理员身份启动的服务器，则服务器中的父进程仍然以该身份继续运行。尤其需要注意的是，不要在此命令中设置管理员身份，除非你确切知道后果如何。

<VirtualHost>配置命令

```
Syntax: <VirtualHost addr[:port] ...> ... </VirtualHost>
Context: server config
Status: Core.
```

Compatibility: 仅在使用非 IP 地址虚拟主机的 Apache 1.1 及更高版本中可用。

Compatibility: 多地址仅在 Apache 1.2 及更高版本中使用。

<VirtualHost> 和 </VirtualHost> 用来封装一组只对特定虚拟主机有效的配置命令。

任何可在虚拟主机环境下使用的配置命令均可在其中出现。当服务器接收对特定虚拟主机的请求时，便会使用在对应<VirtualHost>片段中使用的配置命令。

Addr 参数可以是：

- 虚拟主机的 IP 地址
- 虚拟主机的 IP 地址所对应的一个域名

例如：

```
<VirtualHost 10.1.2.3>
ServerAdmin webmaster@host.foo.com
DocumentRoot /www/docs/host.foo.com
ServerName host.foo.com
ErrorLog logs/host.foo.com-error_log
TransferLog logs/host.foo.com-access_log
</VirtualHost>
```

每个 VirtualHost 必须使用不同的 IP 地址、或不同的端口号、或不同的服务器主机名，在后一种情况下，操作系统配置必须是允许该机器接收送给多个地址的 IP 包的。（如果机器没有多个实际的网络接口，那么可通过 ifconfig 命令来定义别名（如果你的操作系统支持这一点的话），或者在系统中增加类似 VIF （面向 SunOS（TM）4.1.x）的核心补丁。

“_default_”作为一个专用名，可以用来指示对应的虚拟主机将处理任何和其它虚拟主机指定的 IP 地址无法匹配的请求。当没有“ _default_ ”虚拟主机定义时，在任何虚拟主机配置之外的主服务器配置将应用于与任何虚拟主机不匹配的请求。

你可以为虚拟主机指定一个端口号。当没有时，它将取主服务器配置中最近一个 Port 语句指定的值。你还可以通过指定“:*”来匹配该机器地址上的所有端口。在使用_default_虚拟主机时我们建议这样做。

注意：使用<VirtualHost>对 Apache 要监听的地址没有什么影响。相反你应该在 BindAddress 和 Listen 配置中使用了正确的地址。

请参阅本章第 3 节虚拟主机部分。

5.2.2. mod_proxy 配置（代理服务器）

本模块负责处理的配置命令有：

```
ProxyRequests
ProxyRemote
ProxyPass
ProxyPassReverse
ProxyBlock
AllowCONNECT
ProxyReceiveBufferSize
NoProxy
ProxyDomain
ProxyVia
CacheRoot
```

CacheSize
CacheMaxExpire
CacheDefaultExpire
CacheLastModifiedFactor
CacheGcInterval
CacheDirLevels
CacheDirLength
CacheForceCompletion
NoCache

下面逐一介绍：

ProxyRequests 配置命令

Syntax： ProxyRequests on/off
Default： ProxyRequests Off
Context： server config， virtual host
Override： **不适用**
Status： Base
Module： mod_proxy
Compatibility： ProxyRequests **只在** Apache **服务器** 1.1 **及更高版本中支持。**

本配置命令控制 Apache **服务器的代理功能有效与否。**

注意，将 ProxyRequests **设为** 'off' **并不禁止** ProxyPass **配置命令的使用，实际上后者与代理功能基本无关。**

ProxyRemote 配置命令

Syntax： ProxyRemote <match> <remote-server>
Default： None
Context： server config， virtual host
Override： **不适用**
Status： Base
Module： mod_proxy
Compatibility： ProxyRemote **只在** Apache **服务器** 1.1 **及更高版本中支持。**

本命令可为当前的 proxy **服务器再定义一个它使用的远程** proxy **服务器。** <match> **可以是远程服务器支持的** URL **形式的地址，也可以是远程服务器上的相对** URL**，还可以是**'*' **表示对所有的请求都使用该远程代理。**<remote-server> **是远程服务器的部分** URL **地址，具体的语法是：**

<remote-server> = <protocol>：//<hostname>[：port]

其中的<protocol>**是连接远程服务器所使用的协议，现在只支持**"http"**。例如：**

ProxyRemote http：//goodguys .com/ http://mirrorguys.com:8000
ProxyRemote * http://cleversite.com
ProxyRemote ftp http://ftpproxy.mydomain.com:8080

在最后一例中，proxy **将** FTP **请求封装成** HTTP proxy **请求，前传给负责处理的远程** proxy

服务器。

ProxyPass 配置命令

Syntax： ProxyPass <path> <url>
Default： None
Context： server config， virtual host
Override： 不适用
Status： Base
Module： mod_proxy
Compatibility： ProxyPass 只在 Apache 服务器 1.1 及更高版本中支持。

本配置命令允许将远程服务器映射到本地服务器空间中，本地服务器实际不是起代理的作用，而是成为远程服务器的镜像。<path>是本地虚拟路径名；<url>是远程服务器的 URL。

假定本地服务器地址是：http://wibble.org/，则如下配置

```
ProxyPass /mirror/foo/ http://foo.com/
```

将使对本地地址 <http://wibble.org/mirror/foo/bar> 的请求在内部实际地转换成对 <http://foo.com/bar>的 proxy 请求。

ProxyPassReverse 配置命令

Syntax： ProxyPassReverse <path> <url>
Default： None
Context： server config， virtual host
Override： 不适用
Status： Base
Module： mod_proxy
Compatibility： ProxyPassReverse 只在 Apache 服务器 1.3.6 及更高版本中支持。

本配置命令使 Apache 服务器可以调整 HTTP 重定向响应的 Location 头域中的 URL 地址信息。比如在 Apache 服务器作为逆向 proxy 时就需要此功能来实现透明的重定向，因为 HTTP 请求通过此逆向 proxy 被重定向到位于后面的实际服务器上，而后者是不应让用户知道其存在的。

<path>是本地虚拟路径。<url>是远程服务器的 URL，与 ProxyPass 配置命令中的用法相同。

举例说明之，假设本地服务器地址是 http://wibble.org/，则配置

```
ProxyPass        /mirror/foo/ http://foo.com/
ProxyPassReverse  /mirror/foo/ http://foo.com/
```

将不仅使对 <http://wibble.org/mirror/foo/bar> 的本地请求在内部转变成对 <http://foo.com/bar>的 proxy 请求（由 ProxyPass 实现）。该服务器还负责重定向服务器 foo.com 返回的响应：当 http://foo.com/bar 被 foo.com 重定向到 http://foo.com/quux 时，Apache 服务器会在把 HTTP 重定向响应返回客户前将其调整成 http://wibble.org/mirror/foo/quux。

注意 ProxyPassReverse 配置命令也可与 mod_rewrite 的 proxy 传递（pass-through）功能共同使用，因为它并不依赖必须有对应的 ProxyPass 配置命令。

AllowCONNECT 配置命令

Syntax： AllowCONNECT <port list>
Default： AllowCONNECT 443 563
Context： server config， virtual host
Override： 不适用
Status： Base
Module： mod_proxy
Compatibility： AllowCONNECT 只在 Apache 服务器 1.3.2 及更高版本中支持。

AllowCONNECT 配置命令定义一系列 proxy 的 CONNECT 方法可以监听的端口号。现在的浏览器当基于隧道代理进行 HTTPS （安全 HTTP）连接时将使用 CONNECT 方法。缺省情况下，只有默认的 https 端口（443）和 snews 端口（563）被开放。使用 AllowCONNECT 配置命令可以改变默认值，控制允许连接的端口。

ProxyBlock 配置命令

Syntax： ProxyBlock <word/host/domain list>
Default： None
Context： server config， virtual host
Override： 不适用
Status： Base
Module： mod_proxy
Compatibility： ProxyBlock 只在 Apache 服务器 1.2 及更高版本中支持。

ProxyBlock 配置命令中可以指定一系列用空格分隔开的主机或域地址。与指定地址匹配的 HTTP、HTTPS 和 FTP 请求将被阻塞掉。proxy 模块在启动时还会去解析列出的各域名对应的 IP 地址，并存下来在进行匹配处理时使用。比如有如下配置：

```
ProxyBlock joes-garage.com some-host.co.uk rocky.wotsamattau.edu
```

若使用'rocky.wotsamattau.edu'对应的 IP 地址进行请求也仍将被阻塞。
注意仅使用'wotsamattau' 也能与'wotsamattau.edu'匹配。
而如下配置则阻塞对所有站点的连接请求。

```
ProxyBlock *
```

ProxyReceiveBufferSize 配置命令

Syntax： ProxyReceiveBufferSize <bytes>
Default： None
Context： server config， virtual host
Override： 不适用
Status： Base
Module： mod_proxy
Compatibility： ProxyReceiveBufferSize 只在 Apache 服务器 1.3 及更高版本中支持。

ProxyReceiveBufferSize 配置命令定义一个网络缓冲区来处理 HTTP 和 FTP 连接。该值

必须取大于 512 的值，而若要使用系统给的缺省值则可以设成 0。例如：

```
ProxyReceiveBufferSize 2048
```

NoProxy 配置命令

Syntax： NoProxy { <Domain> | <SubNet> | <IpAddr> | <Hostname> }
Default： None
Context： server config， virtual host
Override： 不适用
Status： Base
Module： mod_proxy
Compatibility： NoProxy 只在 Apache 服务器 1.3 及更高版本中支持。

本配置命令只对位于各 Intranet 中的 Apache 服务器有用。NoProxy 配置命令中可以指定一系列空格分隔的子网地址、IP 地址、主机或域地址。对于与指定地址匹配的请求，本 proxy 服务器将直接连接目的服务器，而不是将请求前传给设置的 ProxyRemote 命令中指定的远程 proxy 服务器。例如：

```
ProxyRemote  *  http://firewall.mycompany.com:81
NoProxy        .mycompany.com 192.168.112.0/21
```

NoProxy 配置命令中使用的参数类型可以是如下之一：

- 域：

域地址是一个以“ . ”开头的部分 DNS 域名。它代表逻辑上属于同一 DNS 域的主机（即主机地址是以域地址结尾的）。例如：

```
.com .apache.org
```

为了将域地址和主机地址分辨开（不仅从句法上而且从语义上，DNS 域需要定义对应的 DNS A 记录），域地址前总加上一个“ . ”。

注意：域名的比较是忽略大小写的；且域总是认为是位于 DNS 树的根部的，因此域.MyDomain.com 和 .mydomain.com. （请留意后者结尾处的“ . ”）被认为是相同的。既然域地址的比较并不涉及 DNS 查寻，所以这种类型比下面的子网类型要高效得多。

- 子网：

一个子网地址是数字 IP 地址的一部分，还可能后跟一个斜线及网络地址掩码，来指示确定子网的二进制位。它可用来表示通过公共网络接口可到达的系列主机构成的子网。若没有显式给出网络掩码，则认为是用省略的数位或取 0 值的位来指定掩码（在这种情况下，掩码必须占 8 的整数倍位）。例如：

192.168 或 192.168.0.0

子网 192.168.0.0 隐含表示掩码占 16 位（有时用掩码形式 255.255.0.0）。

192.168.112.0/21

子网 192.168.112.0/21 指示有 21 位掩码（也可用 255.255.248.0 的形式）。

极端情况下，有 32 位均是有效地址的子网只匹配对应的单个 IP 地址，而没有一位有效地址的子网（例如 0.0.0.0/0）则与下面提到的_Default_常量相同，匹配任何 IP 地址。

- IP 地址：

IP 地址表现为一个数字形式的完整的 Internet 地址。通常，这样的地址表示一个主机，

但却不一定有 DNS 域名与之对应。例如：

```
192.168.123.7
```

注意：IP 地址不需要经过 DNS 系统的解析，所以这种类型的定义会提高 Apache 服务器的性能。

- 主机地址：

一个主机地址是一个完整的 DNS 域名，它可以经过 DNS 域名服务被解析成一个或多个 IP 地址。它表示一个逻辑主机（与域的概念相对），必须可解析成至少一个 IP 地址（或是具有不同 IP 地址的一组主机）。例如：

```
prep.ai.mit.edu www.apache.org.
```

注意：在许多情况下，使用 IP 地址要比用主机地址高效，因为可以避免 DNS 解析。尤其当 Apache 服务器是通过很慢的 PPP 连接域名服务器时会耗时甚长。

注意：主机名的比较与域名类似也是忽略大小写的；主机名也总是默认位于 DNS 树的根部，因此两个主机 WWW.MyDomain.com 和 www.mydomain.com.（请留意后者结尾处的“.”）被认为是相同的。

ProxyDomain 配置命令

```
Syntax： ProxyDomain <Domain>
Default： None
Context： server config， virtual host
Override： 不适用
Status： Base
Module： mod_proxy
Compatibility： ProxyDomain 只在 Apache 服务器 1.3 及更高版本中支持。
```

此配置命令只对位于各 Intranet 中的 Apache 服务器有用。ProxyDomain 配置命令指定 Apache 服务器所属的缺省域。若被请求的主机地址中没有给出域名，则会生成一个重定向到主机名加上指定域名得到的地址的响应。例如有如下配置：

```
ProxyRemote  *  http://firewall.mycompany.com:81
NoProxy        .mycompany.com 192.168.112.0/21
ProxyDomain     .mycompany.com
```

ProxyVia 配置命令

```
Syntax： ProxyVia { off | on | full | block }
Default： ProxyVia off
Context： server config， virtual host
Override： 不适用
Status： Base
Module： mod_proxy
Compatibility： ProxyVia 只在 Apache 服务器 1.3.2 及更高版本中支持。
```

此配置命令可定义 HTTP 协议中 Via 头域的使用方式。它被用来控制级联的多个 proxy 服务器间的 proxy 请求流。请参阅 RFC2068 （HTTP/1.1）中对 Via 头域的解释。在此：

- 若设成缺省值 off，则不会有特别的处理，即若请求或应答中含有 Via 头域，则会直传而不做改变。
- 若设为 on，则每个请求和响应在经过每个主机时均会附加一个 Via 头域。
- 若设为 full，附加上的 Via 头域将还包括 Apache 服务器的版本信息。
- 若设为 block，则 proxy 服务器将把到来请求中所有的 Via 头域删除，并增加上新的关于本地主机的 Via 头域。

CacheForceCompletion 配置命令

Syntax：CacheForceCompletion <percentage>
Default： 90
Context： server config， virtual host
Override： 不适用
Status： Base
Module： mod_proxy
Compatibility： CacheForceCompletion 只在 Apache 服务器 1.3.1 及更高版本中支持。

若正在进行并被缓冲的 HTTP 传输被用户要求取消，但传输的部分已超过了 CacheForceCompletion 配置中指定的百分比，则传输将继续直至完成缓冲。

其中的参数，是一个百分数，必须是 1 到 100 之间的数。取 0 则表示用缺省值；取 100 则表示只有当整个传输能完成时，才进行缓冲。一般建议取 60 到 90 之间的数。

CacheRoot 配置命令

Syntax： CacheRoot <directory>
Default： None
Context： server config， virtual host
Override： 不适用
Status： Base
Module： mod_proxy
Compatibility： CacheRoot 只在 Apache 服务器 1.1 及更高版本中支持。

本配置命令用来设置存放缓冲文件的目录；该目录必须是服务器可写的（参阅 User 配置命令）。若设置了 CacheRoot，则表示允许进行 proxy 应答缓冲；若没有设置 CacheRoot 但 ProxyRequests 被设为 On，则表示仅支持 proxy 功能，但不进行缓冲处理。

CacheSize 配置命令

Syntax： CacheSize <size>
Default： CacheSize 5
Context： server config， virtual host
Override： 不适用
Status： Base
Module： mod_proxy

Compatibility： CacheSize 只在 Apache 服务器 1.1 及更高版本中支持。

本配置用来设置缓冲区的大小，以 KB（1024 个字节）为单位。当所占空间超过设置值时，垃圾收集过程将删除某些文件以使空间占用低于此设置值。建议使用少于可用空间的 20 到 40 %，具体要视服务器负载和 CacheGcInterval 的设置来定。

CacheGcInterval 配置命令

Syntax： CacheGcInterval <time>
Default： None
Context： server config， virtual host
Override： 不适用
Status： Base
Module： mod_proxy
Compatibility： CacheGcinterval 只在 Apache 服务器 1.1 及更高版本中支持。

每经过<time>个小时，则检查一次缓冲区，若占用空间超过了 CacheSize 设置的值，则删除某些文件。注意，<time>可以是一个浮点数，你可以用 CacheGcInterval 1.5 来每 90 分钟检查一次缓冲区（若没有设置，则不进行垃圾收集，缓冲区将无限增大）。注意，CacheGcInterval 值越大，为在垃圾收集间隔里超出缓冲区设计大小而预留的空间应越大。

CacheMaxExpire 配置命令

Syntax： CacheMaxExpire <time>
Default： CacheMaxExpire 24
Context： server config， virtual host
Override： 不适用
Status： Base
Module： mod_proxy
Compatibility： CacheMaxExpire 只在 Apache 服务器 1.1 及更高版本中支持。

被缓冲的 HTTP 文档将最多被认为在<time>个小时内不是过期的，即不去访问原始服务器来更新。这样文档顶多经过<time>个小时便会过期。这一限制是强制性的，即使文档被指定了过期时间。

CacheLastModifiedFactor 配置命令

Syntax： CacheLastModifiedFactor <factor>
Default： CacheLastModifiedFactor 0。1
Context： server config， virtual host
Override： 不适用
Status： Base
Module： mod_proxy
Compatibility： CacheLastModifiedFactor 只在 Apache 服务器 1.1 及更高版本中支持。

若原始 HTTP 服务器没有为文档指定过期时间，则使用如下公式会计算一个大致的过期

时间：

```
expiry-period = time-since-last-modification * <factor>
```

例如：某文档是 10 **个小时前修改的，**<factor> **取** 0.1**，则过期时间将是** 10*0.1= 1 **小时。**

若计算的过期时间大于 CacheMaxExpire **设置值，则取后者判定文档是否过期。**

CacheDirLevels 配置命令

Syntax： CacheDirLevels <levels>

Default： CacheDirLevels 3

Context： server config， virtual host

Override： **不适用**

Status： Base

Module： mod_proxy

Compatibility： CacheDirLevels **只在** Apache **服务器** 1.1 **及更高版本中支持。**

CacheDirLevels **规定在缓冲区中设置的子目录的层数。被缓冲的数据将存放到** CacheRoot **下第**<levels>**级子目录中。**

与本命令相关的缓冲区文件系统路径的设置参阅第 7 **章第** 6 **节和第** 7 **章第** 9 **节。**

CacheDirLength 配置命令

Syntax： CacheDirLength <length>

Default： CacheDirLength 1

Context： server config， virtual host

Override： **不适用**

Status： Base

Module： mod_proxy

Compatibility： CacheDirLength **只在** Apache **服务器** 1.1 **及更高版本中支持。**

CacheDirLength **设置缓冲区子目录名字的长度，即包含的字符个数。**

CacheDefaultExpire 配置命令

Syntax： CacheDefaultExpire <time>

Default： Cache

DefaultExpire 1

Context： server config， virtual host

Override： **不适用**

Status： Base

Module： mod_proxy

Compatibility： CacheDefaultExpire **只在** Apache **服务器** 1.1 **及更高版本中支持。**

若文档是基于不支持过期时间的协议获取的，则用这里的<time>**个小时作为过期时间。**

CacheMaxExpire **不对此设置进行覆盖。**

NoCache 配置命令

Syntax： NoCache <word/host/domain list>
Default： None
Context： server config， virtual host
Override： 不适用
Status： Base
Module： mod_proxy
Compatibility： NoCache 只在 Apache 服务器 1.1 及更高版本中支持。

NoCache 配置命令定义一系列由空格隔开的主机或域地址。与指定地址匹配基于 HTTP 和无密码要求的 FTP 协议获取的文档将不被缓冲。proxy 模块在启动时还将对给出的主机地址进行解析取得对应的 IP 地址，并记录下来以备后用。例如：

```
NoCache joes-garage.com some-host.co.uk bullwinkle.wotsamattau.edu
```

'bullwinkle.wotsamattau.edu' 也将与用 IP 地址提出的请求相匹配。

注意：'wotsamattau' 也与'wotsamattau.edu'匹配。而

```
NoCache *
```

则将完全禁止缓冲功能。

5.2.3. mod_cgi 配置（CGI 脚本）

与 mod_cgi 相关的配置命令，有如下三个：

ScriptLog 配置命令

Syntax: ScriptLog filename
Default: none
Context: server config
Status: mod_cgi

ScriptLog 配置命令设定 CGI 错误日志文件。若没有设置 ScriptLog，则将不进行 CGI 错误日志，否则任何 CGI 执行错误都将记录到 filename 参数指定的文件中。若指定的是一个相对文件路径，则认为是相对于服务器的根目录的。

该日志将以服务器子进程的用户身份（即由 User 配置命令给出的用户身份）被打开，因此无论是日志文件所在的目录，还是该文件本身都必须是该用户身份可写的。如果你将该文件放在你专门存放日志的目录中的话，千万不要改变上述用户身份对此目录的访问权限，因为那会在安全性上留下可乘之机。

注意，CGI 错误日志文件是为了调试 CGI 程序而提供的，而不意味着在服务器已正常工作时仍使用该功能。该日志功能未经过速度或效率上的优化，也可能会造成安全问题。

ScriptLogLength 配置命令

Syntax: ScriptLogLength size

```
Default: 10385760
Context: server config
Status: mod_cgi
```

ScriptLogLength 用来限制 CGI 错误日志文件的大小。由于该文件每次都记录大量的信息（全部的请求头域、全部的 CGI 输出），故它会迅速变得非常庞大。为了避免无限制的增长而出现问题，可以使用本配置命令来设置一个最大限制。若文件超过了指定的大小，服务器将不会再记录任何信息。

ScriptLogBuffer 配置命令

```
Syntax: ScriptLogBuffer size
Default: 1024
Context: server config
Status: mod_cgi
```

任何记录到日志文件中的 PUT 或 POST 实体的大小都是有限制的，以避免日志文件变大的过快。缺省时，最多记录 1024 个字节的信息，通过本配置命令可以改变该值。

5.2.4. mod_access 配置（访问控制）

与本模块直接相关的配置命令有：

```
allow
allow from env=
deny
deny from env=
order
```

具体格式和含义如下：

allow 配置命令

```
Syntax: allow from host host ...
Context: directory, .htaccess
Override: Limit
Status: Base
Module: mod_access
```

allow 配置命令控制哪些主机可以访问特定目录。Host 参数可以是如下形式之一：

- all：
  所有主机均可访问；
- 域名（片段）：
  与指定域名匹配或以其结尾的主机可访问；
- 完整的 IP 地址：
  具备指定 IP 地址的主机可访问；
- IP 地址片段：

IP 地址开始的 1 到 3 个字节，可以进行针对子网的访问控制；

- 网络地址/掩码对（适用于 Apache 1.3 及以后版本）：
  a.b.c.d 形式的 IP 网络地址和 w.x.y.z 形式的掩码，可以进行细粒度的子网访问控制（例如：10.1.0.0/255.255.0.0）；
- 网络地址与 nnn 形式的 CIDR 结合（适用于 Apache 1.3 及以后版本）：
  与前一种情况相同，nnn 表示对应的掩码的 nnn 个高位上均为 1（例如：10.1.0.0/16 与 10.1.0.0/255.255.0.0 含义相同）。

例如：

```
allow from .ncsa.uiuc.edu
```

在指定域中的所有主机均有访问权。

注意：上述匹配要求完整匹配，比如 bar.edu 就不与 foobar.edu 匹配。

参阅 deny，order 和 BrowserMatch 配置命令。

```
Syntax: allow from env=variablename
Context: directory, .htaccess
Override: Limit
Status: Base
Module: mod_access
Compatibility: Apache 1.2 及以后版本适用
```

allow from env 配置命令可基于某个环境变量的定义与否来控制对目录的访问。

例如：

```
BrowserMatch ^KnockKnock/2.0 let_me_in
<Directory /docroot>
    order deny,allow
    deny from all
    allow from env=let_me_in
</Directory>
```

表示包含 KnockKnock/2.0 字样的客户端浏览器被允许访问，其它的则拒绝。

参阅 deny from env 和 order 配置命令。

deny 配置命令

```
Syntax: deny from host host ...
Context: directory, .htaccess
Override: Limit
Status: Base
Module: mod_access
```

deny 配置命令控制哪些主机可以访问特定目录。Host 参数的取值与 allow 配置命令相同。

例如：

```
deny from 16
```

表示在指定网络中的所有主机都被拒绝访问。

参阅 allow 和 order。

```
Syntax: deny from env=variablename
Context: directory, .htaccess
```

```
Override: Limit
Status: Base
Module: mod_access
Compatibility: Apache 1.2
```
及更高版本可用。

deny from env 配置命令可基于某个环境变量的定义与否来控制对目录的访问。

例如：

```
BrowserMatch ^BadRobot/0.9 go_away
<Directory /docroot>
    order allow,deny
    allow from all
    deny from env=go_away
</Directory>
```

表示包含 BadRobot/0.9 字样的客户端浏览器被拒绝访问，其它的则允许。

参阅 allow from env 和 order 配置命令。

order 配置命令

```
Syntax: order ordering
Default: order deny,allow
Context: directory, .htaccess
Override: Limit
Status: Base
Module: mod_access
```

order 配置命令控制 allow 和 deny 配置命令被应用的先后次序。Ordering 参数可取如下形式之一：

- deny,allow
  deny 先于 allow 配置命令被应用（初始状态是所有的主机访问均认为 OK）。
- allow,deny
  allow 先于 deny 配置命令被应用（初始状态是所有的主机访问均认为 FORBIDDEN）。
- mutual-failure
  只有那些出现在 allow 列表中但未在 deny 列表中的主机才被允许访问（与初始状态无关）。

注意：在任何情况下所有的 allow 和 deny 语句均被处理，而不存在“短路”的情况。

例如：

```
order deny,allow
deny from all
allow from .ncsa.uiuc.edu
```

表示在 ncsa.uiuc.edu 域中的主机被允许，而其它主机则被拒绝。

5.2.5. 认证模块配置（身份认证控制）

- 与 mod_auth 模块相关的配置命令有：

AuthGroupFile
AuthUserFile
AuthAuthoritative

- 与 mod_auth_anon 模块相关的配置命令有：

Anonymous
Anonymous_Authoritative
Anonymous_LogEmail
Anonymous_MustGiveEmail
Anonymous_NoUserID
Anonymous_VerifyEmail

- 与 mod_auth_db 模块相关的配置命令有：

AuthDBGroupFile
AuthDBUserFile
AuthDBAuthoritative

- 与 mod_auth_dbm 模块相关的配置命令有：

AuthDBMGroupFile
AuthDBMUserFile
AuthDBMAuthoritative

- 与 mod_digest 模块相关的配置命令有：

AuthDigestFile

AuthGroupFile 配置命令

Syntax： AuthGroupFile filename
Context： directory，.htaccess
Override： AuthConfig
Status： Base
Module： mod_auth

AuthGroupFile 配置命令指定一个正文文件名，该文件中含有用于用户身份认证的用户组列表。Filename 参数给出该文件的路径。若路径不是绝对路径（即不是以斜线开头），则认为是相对于 ServerRoot 的。

该组列表文件中每一行对应一个组。首先是组名，后跟“ :”然后是空格隔开的组内各成员的名字列表。比如：

mygroup: bob joe anne

注意,对较大的正文文件的搜索效率很抵,这时应使用基于数据库的 AuthDBMGroupFile 方式。在安全方面，务必把 AuthGroupFile 文件存放在服务器文档树外，否则客户将可能下载该文件。

请参阅 AuthName，AuthType 和 AuthUserFile 等命令。

AuthUserFile 配置命令

Syntax： AuthUserFile filename
Context： directory，.htaccess

```
Override： AuthConfig
Status： Base
Module： mod_auth
```

AuthUserFile 配置命令指定一个正文文件名，该文件中含有用于用户身份认证的用户名及口令。Filename 参数给出该用户文件的路径。若该路径不是绝对路径（即不是以斜线开头），则认为是相对于 ServerRoot 的。

该文件中每一行对应一个用户。在 Windows 平台下，用户密码的生成是通过使用 htpasswd 工具完成的（具体实现在 ap 项目中的 ap_md5.c 文件的 ap_MD5Encode 方法中）。

通过其处理可知，在密码文件中每一行均为由用户名及相应密码信息组成的记录，形式为：

[用户名]：[加密处理后的密码部分]

其中后一部分的结构是：

[Apache 加密算法特征串][加密用的盐-salt]"$"[MD5 处理过的密码]

其中的[Apache 加密算法特征串]为"$apr1$"，而[加密用的盐-salt]则是一个在生成该记录时取得的一个随机数，该随机数则又是使用了 srand、rand 系统函数和 ap_md5.c 文件中的 to64 函数得到的一个长为 8 的字符串。最后的[MD5 处理过的密码]是基于用户密码明文、[Apache 加密算法特征串]和[加密用的盐-salt]经一系列变换得到的。

在需要对用户进行验证时，首先检查相应的如上的密码部分开头是否是[Apache 加密算法特征串]，若是才认为其是合法的基于 Apache 的算法生成的串。然后把盐取出，基于该盐和[Apache 加密算法特征串]对用户传来的密码明文进行加密，再与原始[MD5 处理过的密码]做比较，相同则通过身份认证，否则失败。

在 Unix 平台下，口令是用 crypt()方法生成的。

注意，对较大的正文文件的搜索效率很抵，这时应使用基于数据库的 AuthDBMUserFile 方式。在安全方面，务必把 AuthUserFile 文件存放在服务器文档树外，否则客户将可能下载该文件。

参阅 AuthName，AuthType 和 AuthGroupFile 等命令。

AuthAuthoritative 配置命令

```
Syntax： AuthAuthoritative < on(default) | off >
Context： directory，.htaccess
Override： AuthConfig
Status： Base
Module： mod_auth
```

如果对同一目录用了不止一个认证模式，可以将此配置设为 off。当第一种认证模式对用户名/口令的验证失败后，可以将它传送到下一层（更低层）。

例如，如果用 mod_auth_db 和标准的 mod_auth 模块来提供认证服务，其中一个模块对用户提供的用户名/口令的认证失败后，则另一个模块用来认证这一用户。到底哪个模块被认为是较低或较高层的认证模块呢？这由这些模块在配置文件中出现的顺序来决定。

在任何情况下，当所有模块对用户名/口令的认证均失败时，服务器将返回状态码 401 和 WWW 认证应答头域要求重新进行认证。然而，如果用户名/口令成功地通过了某一模块的认证，那么较低模块就收不到用户名/口令对。

取缺省值时，认证处理不前传。一个未知的用户名将使服务器返回“需要验证

（Authorization Required）”的响应。不对此配置命令进行设置将能保证系统的安全，从而表现为与 NCSA 服务器兼容的模式。

参阅 AuthName，AuthType 和 AuthGroupFile 等命令。

Anonymous 配置命令

```
Syntax： Anonymous user user ...
Default： none
Context： directory， .htaccess
Override： AuthConfig
Status： Extension
Module： mod_auth_anon
```

本配置命令可以定义一系列特殊的用户 ID，使用这些用户名的访问将不进行口令验证。各 ID 用空格间隔开。要在用户名中使用空格，则需要使用“ ' ”和“ " ”，也可以用\开头的转义字符。

请注意这里的比较是大小写不敏感的。强烈建议你将'anonymous'用户名一定作为其中的一个匿名用户名。

例如：

```
Anonymous anonymous "Not Registered" 'I don\'t know'
```

此例中的设置将使用户输入用户名 'anonymous' 或 'AnonyMous' 或 'NotRegistered' 或 'I Don't Know'即可访问站点而不进行口令验证。

Anonymous_Authoritative 配置命令

```
Syntax： Anonymous_Authoritative on | off
Default： Anonymous_Authoritative off
Context： directory， .htaccess
Override： AuthConfig
Status： Extension
Module： mod_auth_anon
```

设为 'on' 表示：在身份验证失败时也不前传给其它认证方法。所以如果客户给出的用户 ID 不能与 Anonymous 配置命令中给出的名字列表匹配的话，访问将被拒绝。

在改变此配置时一定要考虑清楚。切记各个相关模块（在配置文件中）的相对位置决定了它们依次被调用的顺序。

Anonymous_LogEmail 配置命令

```
Syntax： Anonymous_LogEmail on | off
Default： Anonymous_LogEmail on
Context： directory， .htaccess
Override： AuthConfig
Status： Extension
```

Module： mod_auth_anon

当设为缺省值 'on'**时，输入的口令将被记录在错误日志中。**

Anonymous_MustGiveEmail 配置命令

Syntax： Anonymous_MustGiveEmail on | off
Default： Anonymous_MustGiveEmail on
Context： directory， .htaccess
Override： AuthConfig
Status： Extension
Module： mod_auth_anon

指定匿名用户是否必须给出一个 email **地址作为口令。这可以禁止用户提供空的口令。**

Anonymous_NoUserID 配置命令

Syntax： Anonymous_NoUserID on | off
Default： Anonymous_NoUserID off
Context： directory， .htaccess
Override： AuthConfig
Status： Extension
Module： mod_auth_anon

当设为'on'**时，用户在匿名访问时可以不给出用户** ID**（甚至口令）。这对于浏览器的用户来说是非常自然的，也是非常方便的，因为他们需要做的只是敲一下回车或点一下** OK **按钮。**

Anonymous_VerifyEmail 配置命令

Syntax： Anonymous_VerifyEmail on | off
Default： Anonymous_VerifyEmail off
Context： directory， .htaccess
Override： AuthConfig
Status： Extension
Module： mod_auth_anon

若设为 'on' **，则输入的口令将被检查，看其中是否含有**'@' **和** '.' **以让用户输入合法的** email **地址（参阅上面的** Auth_LogEmail **配置命令）。**

例如 access.conf **文件片段如下所示：**

```
Anonymous_NoUserId off
Anonymous_MustGiveEmail on
Anonymous_VerifyEmail on
Anonymous_LogEmail on
Anonymous anonymous guest www test welcome
AuthName Use 'anonymous' & Email address for guest entry
```

```
AuthType basic
# An AuthUserFile/AuthDBUserFile/AuthDBMUserFile
# directive must be specified, or use
# Anonymous_Authoritative for public access.
# In the .htaccess for the public directory, add:
<Files *>
order deny,allow
allow from all
require valid-user
</Files>
```

其中规定如下要求：

- **用户必须输入一个用户名（**Anonymous_NoUserId**）；**
- **用户必须输入一个口令（**Anonymous_MustGiveEmail**）；**
- **输入的口令必须是一个合法的** email **地址，即包含至少一个**'@' **和一个**'.' **（**Anonymous_VerifyEmail**）；**
- **用户名必须是**'anonymous'、'guest'、'www'、'test'、'welcome' **其中之一，且比较不是大小写敏感的；**
- **在口令域中输入的** Email **地址将记录到错误日志文件中（**Anonymous_LogEmail**）。**

当上例和基于正文文件（或 GDM、mSQL **等）结合使用时，将使系统支持外来的匿名访问者。**

AuthDBGroupFile 配置命令

```
Syntax: AuthDBGroupFile filename
Context: directory, .htaccess
Override: AuthConfig
Status: Extension
Module: mod_auth_db
```

AuthDBGroupFile **配置命令设置包含用于身份认证的用户组信息的** DB **文件的名字。**Filename **参数是该组文件的绝对路径。**

该组文件以用户名为键，对应的值则是用户所属的各个组的列表，其中各个组用“ ,”隔开。列表中不能出现空格和“ :”。

在安全方面，务必把 AuthDBGroupFile **文件存放在服务器文档树外，否则客户将可能下载该文件。**

有时在一个数据库中同时管理用户口令和用户组可能更方便。这将简化某些支持工具的编写，因为它们只需要更改和锁定一个 DB **文件。这可以通过设置组文件和口令指向同一个** DB **文件来实现：**

```
AuthDBGroupFile /www/userbase
AuthDBUserFile /www/userbase
```

在 Unix **平台下，该** DB **文件以用户名为键。对应的值包括：**

```
Unix 方式加密的口令：组列表[ : （忽略部分）]
```

各部分之间用“ :” 分隔开。其中口令部分包含以 Unix **中的** crypt()**方法加密的口令。**Windows **平台下的情形参阅** AuthUserFile **配置命令。**

第二部分则是用“ , ”分隔的组列表。其它内容则是可选的，放在 DB 文件中，但被认证模块忽略掉。

参阅 AuthName、AuthType 和 AuthDBUserFile 等配置命令。

AuthDBUserFile 配置命令

```
Syntax： AuthDBUserFile filename
Context： directory， .htaccess
Override： AuthConfig
Status： Extension
Module： mod_auth_db
```

AuthDBUserFile 配置命令设置存放用于身份认证的用户名及口令列表的 DB 文件的名字。Filename 参数指定该用户文件的绝对路径。

在 Unix 平台下，用户文件以用户名为键，对应的值是经过 crypt()方法加密的口令，另外还可以后跟以“ :”开头的其它信息。最后的信息将被服务器所忽略。Windows 平台下的情形参阅 AuthUserFile 配置命令。

在安全方面，务必把 AuthDBUserFile 文件存放在服务器文档树外，否则客户将可能下载该文件。

注意，Apache 中实现的"dbopen" 是从 DB 数据结构中读取经过哈希处理的特定长度的值，而不是根据传结尾的 NULL 来判定字符串结束。某些应用，如 Netscape 服务器则依赖于串结尾的 NULL。所以若你在使用 DB 文件时出现问题，则可能是上述原因造成的。

参阅 AuthName、AuthType 和 AuthDBGroupFile 等配置文件。

AuthDBAuthoritative 配置命令

```
Syntax： AuthDBAuthoritative < on(default) | off >
Context： directory， .htaccess
Override： AuthConfig
Status： Base
Module： mod_auth
```

将 AuthDBAuthoritative 配置命令显式地设为 'off' 将使身份认证可以传递给低级的模块来处理。缺省值为 'on'，表示只有第一个模块负责处理之，若客户提供的用户名和口令无法通过验证，则返回“ 需要重新验证 ”的响应。

请参阅 AuthName、AuthType 和 AuthDBGroupFile 等配置命令。

AuthDBMGroupFile 配置命令

```
Syntax： AuthDBMGroupFile filename
Context： directory， .htaccess
Override： AuthConfig
Status： Extension
Module： mod_auth_dbm
```

AuthDBMGroupFile 配置命令设置包含用于身份认证的用户组信息的 DBM 文件的名字。Filename 参数是该组文件的绝对路径。

该组文件以用户名为键，对应的值则是用户所属的各个组的列表，其中各个组用“，”隔开。列表中不能出现空格和“：”。

在安全方面，务必把 AuthDBMGroupFile 文件存放在服务器文档树外，否则客户将可能下载该文件。

有时在一个数据库中同时管理用户口令和用户组可能更方便。这将简化某些支持工具的编写，因为它们只需要更改和锁定一个 DBM 文件。这可以通过设置组文件和口令指向同一个 DBM 文件来实现：

```
AuthDBMGroupFile  /www/userbase
AuthDBMUserFile  /www/userbase
```

在 Unix 平台下，该 DBM 文件以用户名为键。对应的值包括：

Unix 方式加密的口令：组列表[： （忽略部分）]

各部分之间用“：”分隔开。其中口令部分包含以 Unix 中的 crypt()方法加密的口令。Windows 平台下的情形参阅 AuthUserFile 配置命令。

第二部分则是用“，”分隔的组列表。其它内容则是可选的，放在 DBM 文件中，但被认证模块忽略掉。

参阅 AuthName、AuthType 和 AuthDBMUserFile 等配置命令。

AuthDBMUserFile 配置命令

```
Syntax： AuthDBMUserFile filename
Context： directory， .htaccess
Override： AuthConfig
Status： Extension
Module： mod_auth_dbm
```

uthDBMUserFile 配置命令设置存放用于身份认证的用户名及口令列表的 DBM 文件的名字。Filename 参数指定该用户文件的绝对路径。

在 Unix 平台下，用户文件以用户名为键，对应的值是经过 crypt()方法加密的口令，另外还可以后跟以“：”开头的其它信息。最后的信息将被服务器所忽略。Windows 平台下的情形参阅 AuthUserFile 配置命令。

安全性：务必把 AuthDBMUserFile 文件存放在服务器文档树外，否则客户将可能下载该文件。

注意，Apache 中实现的"dbm_open" 是从 DBM 数据结构中读取经过哈希处理的特定长度的值，而不是根据传结尾的 NULL 来判定字符串结束。某些应用，如 Netscape 服务器则依赖于串结尾的 NULL。所以若你在使用 DBM 文件时出现问题，则可能是上述原因造成的。

参阅 AuthName、AuthType 和 AuthDBMGroupFile 等配置文件。

AuthDBMAuthoritative 配置命令

```
Syntax： AuthDBMAuthoritative < on(default) | off >
Context： directory， .htaccess
Override： AuthConfig
```

Status： Base

Module： mod_auth

将 AuthDBMAuthoritative 配置命令显式地设为 'off' 将使身份认证可以传递给低级的模块来处理。缺省值为 'on'，表示只有第一个模块负责处理之，若客户提供的用户名和口令无法通过验证，则返回“需要重新验证”的响应。

请参阅 AuthName、AuthType 和 AuthDBMGroupFile 等配置命令。

AuthDigestFile 配置命令

Syntax: AuthDigestFile filename

Context: directory, .htaccess

Override: AuthConfig

Status: Base

Module: mod_digest

AuthDigestFile 配置命令用来设置一个文本文件名，该文件中含有用于摘要认证的一系列用户名和经过编码的口令。Filename 参数应用绝对路径给出。

摘要文件使用一种特殊的格式。你可以使用 Apache 源代码目录下 support 子目录中的 htdigest 工具来创建该摘要文件。htdigest 工具并非是现成的，你还需要建立一个 VC 项目对 mod_digest.c 进行编译才能得到它。

5.2.6. mod_log_config 配置（日志）

与本模块相关的配置命令有：

CookieLog

CustomLog

有条件 CustomLog

LogFormat

TransferLog

下面分别解释之：

CookieLog 配置命令

Syntax： CookieLog filename

Context： server config， virtual host

Module： mod_cookies

Compatibility： 只在 Apache 1.2 或更高版本中可用。

CookieLog 配置命令指定记录 cookies 的文件名称。文件名是相对于 ServerRoot 路径的。此配置仅在 mod_cookies 模块存在时才起作用，不建议使用它。

CustomLog 配置命令

Syntax： CustomLog file-pipe format-or-nickname
Context： server config， virtual host
Status： Base
Compatibility：只在 Apache 1.3 或更高版本中可用。
Module： mod_log_config

第一个参数是记录日志项目的文件名称，其用法与 TransferLog 配置命令中的对应参数类似，即或者是一个绝对路径，或者是一个相对于服务器根目录的相对路径。

format 参数为日志文件的每一行指定格式，其中可使用的选项与 LogFormat 配置命令中对应参数完全相同。若 format 参数值中包含空格，则应用双引号将整个串括起来使用。

你还可以不用一个实际的格式化串，而用已由 LogFormat 配置命令定义的格式别名代替。

有条件 CustomLog 配置命令

Syntax： CustomLog file-pipe format-or-nickname env=[!]environment-variable
Context： server config， virtual host
Status： Base
Compatibility：只在 Apache 1.3.5 或更高版本中可用。
Module： mod_log_config

本配置命令与标准的 CustomLog 配置用法基本相同。唯一的区别在于'env=' 条件子句可以控制在日志文件中是否记录特定请求。若在请求中指定的环境变量被定义（或者未被定义，对应 'env=!name' 子句），则请求将被记录。

环境变量可以使用 mod_setenvif 和/或 mod_rewrite 模块在请求级别上定义。例如，你想把对服务器上所有 GIF 图象文件的请求记录到一个单独的日志文件而不是主日志中，你可以用如下配置：

```
SetEnvIf Request_URI \.gif$ gif-image
CustomLog gif-requests.log common env=gif-image
CustomLog nongif-requests.log common env=!gif-image
```

LogFormat 配置命令

Syntax： LogFormat format [nickname]
Default： LogFormat "%h %l %u %t \"%r\" %s %b"
Context： server config， virtual host
Status： Base
Compatibility：只在 Apache 1.3 或更高版本中可用。
Module： mod_log_config

此命令设置由 TransferLog 配置命令指定的缺省日志的记录格式。关于具体的格式定义参阅第 7 章第 10 节日志格式部分。

若你在命令行中使用了 nickname（别名）参数，则你可以在其他的 LogFormat 和

CustomLog 配置中使用该别名，而无须重复定义整个的格式化串。包含 nickname 参数的 LogFormat 配置命令仅仅是定义了一个别名，而不再定义缺省的日志格式。

TransferLog 配置命令

```
Syntax： TransferLog file-pipe
Default： none
Context： server config， virtual host
Status： Base
Module： mod_log_config
```

TransferLog 配置命令指示增加一个日志文件，格式由最近定义的 LogFormat 配置命令给出，当没有显式的缺省日志格式定义时使用普通日志格式。File-pipe 参数在如下各值中取一：

- 文件名：
  相对于 ServerRoot 路径；
- 后跟命令名的`|'：
  命令名指定的程序接收 agent 日志信息作为其标准输入。注意当是虚拟主机继承了来自主服务器的 TransferLog 配置时并不会启动新的程序进程。另外，指定的程序是在运行服务器的用户权限下运行的,比如以管理员身份启动服务器则该程序也将以该身份运行，这时应特别注意该程序是否设计安全，有无漏洞。

5.2.7. mod_status 配置（服务器状态信息反馈）

mod_status 的相关配置命令有：

```
ExtendedStatus
```

ExtendedStatus 配置命令

```
Syntax: ExtendedStatus On|Off
Default: ExtendedStatus Off
Context: server config
Status: Base
Module: mod_status
Compatibility: ExtendedStatus 仅在 Apache 1.3.2 及以后版本中可用。
```

本命令控制服务器是否跟踪每次请求的扩展状态信息，当然此命令只有在 mod_status 模块被装入时才有效。

本配置适用于整个服务器，而无法针对单个的虚拟服务器来控制。

5.2.8. mod_headers 配置（头域定制）

相关的配置命令有：

```
Header
```

Header 配置命令

```
Syntax: Header [ set | append | add ] header value
Syntax: Header unset header
Context: server config, virtual host, access.conf, .htaccess
Status: optional
Module: mod_header
```

本配置命令可以替换、合并或删除 HTTP **响应头域。采取的动作由第一个参数指定。该参数有如下取值：**

- set
  用于设置响应头域，并将原有相同名字的头域清除掉。
- append
  将指定头域附加到原有的相同名字的头域上。合并时，用“，”将新头域和原有头域分隔开。这是在 HTTP **头域中设置多值的标准方式。**
- add
  指定头域将以独立头域的形式增加到响应中，而不管是否已有相同名字的头域存在。这可能会造成两个（或更多）头域具有相同的名字，从而造成无法预见的结果，所以一般尽量使用 "append" **。**
- unset
  清除可能存在的指定名字的头域。如果有多个这样的头域存在，则只清除第一个。

随后的参数用于指定一个头域名，该名最后可以有一个“:”，但也可没有，且名字是忽略大小写的。在 add、append **和** set **方式下，还有第三个参数。若该值中含有空格，则需要使用双引号。而对于** unset **方式，则无该参数。**

这里需要特别强调的是多个 Header **配置命令的先后作用次序。**

在服务器配置文件中，Header **配置命令几乎可以出现在任何地方。它可以用于主服务器和虚拟主机配置，包含在**<Directory>、<Location> **和** <Files>**片段中；也可以用在**.htaccess **文件中。**

Header **配置命令的处理次序如下：**

1. **主服务器**
2. **虚拟主机**
3. <Directory>**片段和**.htaccess
4. <Location>**片段**
5. <Files>**片段**

次序是非常重要的。如下的两个头域如果颠倒了会有不同的效果：

```
Header append Author "John P. Doe"
Header unset Author
```

若是如上次序，则将没有 Author **头域设置。而若颠倒过来，将有值为**"John P. Doe"**的** Author **头域设置。**

Header **配置是在响应被适当的处理方法返回前被处理的，这就意味着某些头域无法被清除或覆盖。因为它们可能在处理方法中设置，例如**"Date" **和** "Server"**头域。**

5.2.9. mod_info 配置（服务器模块信息反馈）

mod_info 模块处理的配置命令有：

AddModuleInfo

AddModuleInfo 配置命令

Syntax: AddModuleInfo module-name string

Context: server config, virtual host

Status: base

Module: mod_browser

Compatibility: Apache 1.3 及以上版本。

本命令支持将描述模块的附加字符串信息以 HTML 形式显示给客户。

例如：

```
AddModuleInfo mod_auth.c 'See <A HREF="/docs/mod/mod_auth.html" >
/docs/mod/mod_auth.html</A>'
```

第3节　配置命令使用

此处对常用的某些方面的配置进行更为实用的综合性的介绍。涉及的配置命令请参阅前面各章节。

5.3.1. proxy（代理服务器）

代理服务器功能由 mod_proxy 模块实现。该模块是动态装入的扩展模块。应在缺省的配置文件 httpd.conf 中将如下的行前的注释符“ # ”去掉：

LoadModule proxy_module modules/ApacheModuleProxy.dll

Apache 处理该 LoadModule 指令，完成将 mod_proxy 模块的装入，同时将其连进链表，并完成配置信息块初始化和自身初始化。同样地，将与代理及其缓冲功能相关的配置项亦消去注释：如 ProxyRequest、CacheRoot 等。

在 mod_proxy 模块被装入之后，即可对相关的配置命令进行处理。换句话说，就是与 proxy 相关的其它配置命令一定要在配置文件中装入 mod_proxy 模块的 LoadModule 命令后出现，否则无法处理。

其它动态装入模块的处理大致类似。

在使用代理功能时，您特别需要注意的是对该功能要进行访问控制，即禁止无关人员使用该功能；否则你可能会遭受严重损失。

在中国，尤其是一些读书的学生，由于囊中羞涩，而又抵不过网上冲浪的诱惑，经常会利用一些如 ProxyHunter 等工具搜索网上的可自由使用的代理服务器,以便能访问国外站点。如果您只是想自己使用设置的代理服务器，而并不希望其成为大家共享的“自由乐园”的话，就必须进行代理的访问控制。对于 Apache 代理服务器的访问控制，已包含在了下一节中基于 IP 地址的访问控制中。

5.3.2. auth（访问控制）

Apache 服务器可以对用户的访问施加控制。

根据 Apache 中访问控制配置信息所在的位置不同，可以分为全局性访问控制和目录级的访问控制。前者是在全局配置文件中定义相关配置命令实现；而后者则是在相应的目录下专门设置一访问控制配置文件，在文件中给出具体的控制方法。

而控制策略或方法则又分为基于 IP 地址的访问控制和基于身份认证的访问控制。如下首先对如何用此两种策略进行全局性的访问控制进行介绍，最后描述如何实现目录级的访问控制。

基于 IP 地址的访问控制

1. 首先修改配置文件，增加相应的控制片段：

如下是一个对 proxy 服务进行访问控制的配置片段，基于 IP 地址实现：

```
<Directory proxy:*>
    Order deny,allow
    deny from all
    allow from 159.226.42.66 159.226.42.67 159.226.42.68 159.226.42.69
    allow from 159.226.42.93 159.226.42.92 159.226.42.91 159.226.42.71
</Directory>
```

其中<Directory proxy:*>表示后续配置命令的作用范围。其中的 proxy:*就表示控制的应用范围是所有的 proxy 请求，因为在 Apache 中，一旦一个请求被判定是 proxy 请求则在其 URL 上用 proxy 代替掉原来的协议模式串，如 http，以标识其是 proxy 请求，请参阅 mod_proxy 模块的实现代码。

Order deny,allow 表示 deny 的控制要服从于 allow 控制，即 deny 定义允许的范围还必须在 allow 定义允许的范围内，即后者有最终的决定权。

```
deny from all
allow from 159.226.42.66 159.226.42.67 159.226.42.68 159.226.42.69
allow from 159.226.42.93 159.226.42.92 159.226.42.91 159.226.42.71
```

如上三行指示 159.226.42.66 159.226.42.67 159.226.42.68 159.226.42.69、159.226.42.93 159.226.42.92 159.226.42.91 159.226.42.71 等八个 IP 地址的机器有访问本 proxy 服务的权限。

这里还需要说明的是，当要对文件系统部分进行控制时，需要用<Directory>和/或<Files>来完成；而在对 URL 进行控制时，则要用<Location>。

但有一个例外是：当要对 proxy 服务进行访问控制时，要用<Directory>，而不能用<Location>。造成其的原因是：proxy 服务是在<Location>之前引入的。在将来的版本中，将移到<Location>中。

2. 重新启动服务器即可。

基于身份认证的访问控制

1. 文件创建。执行 ServerRoot 下的 bin 目录中 htpasswd 命令：htpasswd -c <passwd file path>，完成所对应的密码文件的创建。

2. **用户增加。执行：**htpasswd -m <passwd file path> <new user name>**，其中要指定密码文件及新用户名。随后会要求输入两次密码，成功后，即创建了该用户项。**

3. **修改配置文件。增加如下片段：**

```
<Directory *>
Order deny,allow
allow from all
AuthUserFile <passwd file path>
AuthName AnyName
AuthType basic
require valid-user
</Directory>
```

4. **重新启动** Apache **服务器即可。**

基于 IP **地址的访问控制和基于身份认证的访问控制可以合并到一起，同时作用。如：**

```
<Directory proxy:*>
    Order deny,allow
    deny from all
    allow from 159.226.42.66 159.226.42.67 159.226.42.68 159.226.42.69
    allow from 159.226.42.93 159.226.42.92 159.226.42.91 159.226.42.71
    AuthUserFile control/passwd
    AuthName ProxyOnHomenetwork
    AuthType basic
    require valid-user
</Directory>
```

即对来自合法 IP **地址的用户还要通过身份认证才有访问权限。**

基于目录的访问控制

此种控制是在相应的目录下专门设置一访问控制配置文件,在文件中给出具体的控制策略。

1. **定义访问控制配置文件。**

在全局配置文件中，使用如下的方式指出目录下缺省的配置文件：

```
AccessFileName .htaccess
```

即指示每个目录下的.htaccess **文件，为进行访问控制的配置文件。为了避免外部访问此文件，需对其再专门设置访问控制，如：**

```
<Files .htaccess>
   Order allow,deny
   Deny from all
</Files>
```

即指示拒绝所有的通过 www **来访问此文件。**

2. **在要进行访问控制的目录下创建一**.htaccess **文件，其内容如下例所示：**

```
AuthUserFile \control\passwd
AuthGroupFile Null
AuthName AnyNameForUserToIdentifyTheDomain
AuthType basic
```

```
<Limit GET>
require user home
</Limit>
```

其中，AuthUserFile 定义存放用户名及密码的文件；AuthGroupFile 定义存放用户分组信息的文件；<Limit>片段则定义对指定的方法如 GET 有指定的访问控制。require user home 指示只有用户 home 才有权限，且其要提供正确的口令。

在该文件中可以使用基于 IP 的或基于用户身份认证的任意策略。

3. 若用户密码文件已创建好，则不需要进一步的操作；否则还要创建该文件。

要使用 Apache 中附带的 htpasswd 命令，来创建该文件，及添加一用户及其口令。

若还想对用户进行分组并基于之进行访问控制，则还需创建分组文件。其是一个纯文本的无加密文件，用任何一个文本编辑器即可进行设置。其内容如下所示：

```
group1 user11 user12 user13 ...
group2 user21 user22 user23 ...
```

对于具体的 htpasswd 使用方法及分组文件格式，请参阅 Apache 服务器附带的帮助文档。

5.3.3. log（日志）

设置日志可以方便对服务器的日常监控和维护。

在系统配置文件中可以使用 LogLevel、LogFormat、CustumLog、ErrorLog 等命令对日志的各方面进行配置。例如如下片段：

```
1.  ErrorLog logs/error.log

2.  LogLevel warn

3.  LogFormat "%h%l%u%t \"%r\" %>s %b \"%{Referer}i\" \"%{User-Agent}i\""
combined
4.  LogFormat "%h %l %u %t \"%r\" %>s %b" common
5.  LogFormat "%{Referer}i -> %U" referer
6.  LogFormat "%{User-agent}i" agent

7.  CustomLog logs/access.log common
8.  CustomLog logs/referer.log referer
9.  CustomLog logs/agent.log agent
10. #CustomLog logs/access.log combined
```

其中：

第一句指定相对于服务器根目录的 logs/error.log 作为错误日志文件；

第二句指定错误日志记录的级别是警告级，即在警告级或以上的错误均要进行日志记录；

第三到六句定义了四种日志记录格式，并分别给出别名——combined、common、referer 和 agent。common 格式定义日志记录包括远程主机、远程客户的日志名、远程用户名、请求时间、请求的首行、响应状态码和返回的字节数等信息；referer 格式定义日志记录包括原始请求 URI 和实际目的文档路径的转换信息；agent 格式定义日志记录只包括客户端程序名信息；combined 格式则是前三者的综合。

第七到十句分别指示用指定的格式将日志信息写入指定的文件中。其中第七、八、九句

分别用 logs/access.log、logs/referer.log 、logs/agent.log 文件记录普通（common）、地址转换（referer）、用户端代理程序（agent）等信息；第十句则是相当于将前三种信息经过组合（combined）后都写到 logs/access.log 文件中。实际上，第七、八、九句的组合和第十句是两种不同的选择，本例中第十句为注释行，不起作用。

5.3.4. virtual hosts（虚拟主机或服务器）

何为虚拟主机？

虚拟主机模式是指在一台机器上运行多个服务器。Apache 自诞生之日起即支持基于 IP 地址的虚拟主机模式，自 1.1 开始又支持基于名称或主机的虚拟主机模式。

在基于 IP 地址的虚拟主机模式下，机器必须为每一个虚拟主机设置一个单独的 IP 地址。这既可以通过具有多个物理的网络连接来实现，也可以借助于某些现代操作系统的先进功能来虚拟地实现。可以为每个虚拟主机单独设置一个服务器程序，也可以只用一个程序来支持所有的虚拟主机。

基于名称的虚拟主机模式则不需要任何的硬件或软件上的特别支持。HTTP1.1 对此提供了充分的支持（Host 头域），即可以特别地指明要访问的主机名。这种模式下，可以无限地增加虚拟主机的数量。但其缺点是要求用户程序即浏览器必须支持 HTTP1.1 中的相应功能。

在 RFC2068（HTTP1.1）中，与虚拟主机有关的内容有 5.2 和 19.5.1 节。

Apache 中的虚拟主机

Apache 中可以有一个主服务器（main server），它使用在<VirtualHost>片段以外的所有配置；另外还可有多个虚拟服务器（即虚拟主机——vhosts），它们则使用各自的<VirtualHost>片段中定义的配置。

在任何一个服务器的配置范围中，都可以定义 Port, ServerName, ServerPath 和 ServerAlias 等配置命令。它们可以出现在任何位置，但是后面的出现会覆盖掉前面的定义。主服务器的 Port 的缺省值是 80，ServerPath 和 ServerAlias 命令则没有缺省值，ServerName 的缺省值则可以由服务器的 IP 地址逆向解析得到。

主服务器的 Port 命令实际有两个作用：一是给出 Apache 监听的端口值，此作用现已被 Listen 命令所替代；二是在重定向时用来构造完整的指向服务器本身的完整 URL 地址。与主服务器不同，虚拟主机上设置的端口则对 Apache 监听连接没有任何影响。

在<VirtualHost>片段中可以指定任意的端口，缺省时会取主服务器中的最近的一次配置值。

对于任何 IP 地址，除非有明确的 NameVirtualHost 配置命令定义，否则第一个出现的对应此 IP 地址的虚拟服务器均被看作是基于 IP 模式的；而若想基于名称来实现虚拟服务器，则必须使用 NameVirtualHost 配置命令来定义相应的 IP 地址。

每个 NameVirtualHost 配置命令可以分别对应一系列的<VirtualHost>片段，但对于任何一个确定的 IP:Port，只能有一个 NameVirtualHost 配置命令对应。

NameVirtualHost 配置和<VirtualHost>片段之间的次序是无关紧要的，只有各个<VirtualHost>片段之间的次序会有影响。

在初始化阶段，针对每个 IP 地址均会创建一个名字列表。如果 IP 地址是用在 NameVirtualHost 中的，则对应列表中存放所有相应的基于名称的虚拟服务器，若这样的的虚拟服务器为空，则会在解析配置文件时报错。而如果是基于 IP 地址模式的，则对应的列表为空。所有的列表均放到一个哈希表中。针对 IP 地址的哈希是极快的，时间甚至可以忽略不计，而且其还针对通常多个 IP 地址是最后一个数字不同的情况进行了优化。

主服务器的配置被用来作为各虚拟服务器的缺省值，即使是主配置在<VirtualHost>片段后面定义也是如此。

任何一个用“_default_:*”定义的虚拟服务器，均认为具有与主服务器相同的名字。

在设置了多个虚拟服务器的情况下，当 Apache 接收到请求时，要确定请求针对的是哪个虚拟服务器，即要进行虚拟服务器的匹配。具体过程如下：

1. 搜索哈希表（Hash table lookup）：

当与客户建立连接时，即在上述的哈希表中查找对应的 IP 地址。若找不到，则用_default_虚拟服务器代替；若_default_虚拟服务器没有定义，则用主服务器代替之。

基于此，继续确定是基于 IP 还是基于名称。

2. 匹配基于 IP 的虚拟服务器：

如果 IP 对应的名称列表为空，则认为是一个基于 IP 的，则直接用当前找到的虚拟服务器服务之。

3. 匹配基于名称的虚拟服务器：

如果 IP 对应的名称列表中包含多个虚拟服务器项，则它们的次序是与配置文件完全相同的。

表中的第一个拥有最高的优先权，当请求的是一个未知的服务器或请求中没有 Host 头域时，则由它来进行响应。

若请求中有 Host 头域，则在列表中查找一服务器，它的 ServerName 或 ServerAlias 是与 Host 中提供的名称是相匹配的。若找到，则由其来服务。Host 头域中也可以使用端口，但 Apache 总是使用客户端提出请求时连接的端口来进行匹配。

如果客户使用 HTTP1.0 协议，请求中没有 Host 头域，则用请求中的 URI 与各服务器中的 ServerPath 相匹配。若能匹配，则用相应的虚拟服务器来服务。

若没能找到匹配的虚拟服务器，则用列表中的与客户连接的端口相符的服务器来响应。

4. 持续连接：

在一次 TCP/IP 会话中，IP 的查寻进行且仅进行一次。而在一次持续连接中，名称的查寻则是每请求一次进行一次。这样，在一次连接中，客户可以请求不同虚拟服务器上的多个页面。

5. 绝对路径：

如果请求中包含绝对 URL，且其中的地址和端口与某服务器匹配，还与接收请求的地址和端口匹配，则 URL 中的协议模式、主机地址和端口号就被删去，留下的相对 URL 则会传给相应的服务器来处理；而若没有匹配的服务器，则原 URL 保持不变，而直接传给 proxy 部分来当作 proxy 请求来处理，因为含有绝对 URL 地址的请求本来就认为是 proxy 请求。

还需要注意的问题有：

- 对 ServerAlias 和 ServerPath 的检查不适用于基于 IP 模式的虚拟服务器。

- 出于安全上的考虑，Host 头域中出现的端口号并不被使用，而是被忽略掉的。
- 若一个虚拟主机的 ServerPath 值是另一个虚拟主机对应配置值的前缀，则前者将总是被匹配。当然前提是请求中没有 Host 头域通过匹配将两者分开。
- 若两个基于 IP 的服务器，偶然用了同样的 IP 地址，则后一个将始终被匹配，在日志中也将有一警告错误。
- _default_虚拟服务器将匹配没有服务器匹配的任何请求，但必须端口号匹配，除非前者用*来指示可匹配任何端口。
- 如果请求中不包含 Host 头域，且客户连接的地址被用于基于名称的虚拟主机设置，则_default_虚拟服务器和主服务器永远也得不到匹配。因为由上文可以知道，这时地址哈希表中对应此 IP 地址的名称列表必不为空，所以至少其中的第一项（即优先级最高项）会得到匹配。

虚拟主机的设置

- 在基于 IP 地址的虚拟主机模式下：

例 1：服务器有两个 IP 地址——111.22.33.44 和 111.22.33.55，分别对应域名 server.domain.tld 和 www.otherdomain.tld，www.domain.tld 又是前者的别名（CNAME）。具体配置如下：

```
Port 80
DocumentRoot /www/domain
ServerName www.domain.tld

<VirtualHost 111.22.33.55>
DocumentRoot /www/otherdomain
ServerName www.otherdomain.tld
...
</VirtualHost>
```

这样访问 www.domain.tld 必须通过 111.22.33.44，而访问 www.otherdomain.tld 则是对应 111.22.33.55。前者作为主服务器出现。

例 2：若不希望有主服务器，则可将例 1 配置改为：

```
Port 80
ServerName server.domain.tld

<VirtualHost 111.22.33.44>
DocumentRoot /www/domain
ServerName www.domain.tld
...
</VirtualHost>

<VirtualHost 111.22.33.55>
DocumentRoot /www/otherdomain
ServerName www.otherdomain.tld
...
</VirtualHost>
```

这样主服务器将无法接收除 localhost **以外的任何请求，因为服务器的所有** IP **地址均被基于** IP **的虚拟主机使用。**

例 3**：服务器有两个** IP **地址——**111.22.33.44 **和** 111.22.33.55**，分别对应域名** server.domain.tld **和** www.otherdomain.tld**。**www.domain.tld **为前者的别名（**CNAME**），代表主服务器，端口为** 80**；**www-cache.domain.tld **则为具缓冲功能的代理服务器，监听端口** 8080**。具体配置如下：**

```
Port 80
Listen 111.22.33.44:80
Listen 111.22.33.55:8080
ServerName server.domain.tld

<VirtualHost 111.22.33.44:80>
DocumentRoot /www/domain
ServerName www.domain.tld
...
</VirtualHost>

<VirtualHost 111.22.33.55:8080>
ServerName www-cache.domain.tld
...
   <Directory proxy:>
   order deny,allow
   deny from all
   allow from 111.22.33
   </Directory>
</VirtualHost>
```

从另一个角度看，若为每个虚拟主机单独设置一个服务器，则应使用 Listen **配置命令来指明该服务器所服务的主机，如：**

Listen www.smallco.com:80**。**

若只用一个服务器来支持所有的虚拟主机，则在配置文件中使用 VirtualHost **来实现。在** VirtualHost **中对不同的虚拟主机分别设置不同的** ServerAdmin**、** ServerName**、**DocumentRoot**、**ErrorLog **和** TransferLog **或** CustomLog **值。如：**

```
<VirtualHost www.smallco.com>
ServerAdmin webmaster@mail.smallco.com
DocumentRoot /groups/smallco/www
ServerName www.smallco.com
ErrorLog /groups/smallco/logs/error_log
TransferLog /groups/smallco/logs/access_log
</VirtualHost>

<VirtualHost www.baygroup.org>
ServerAdmin webmaster@mail.baygroup.org
DocumentRoot /groups/baygroup/www
ServerName www.baygroup.org
```

```
ErrorLog /groups/baygroup/logs/error_log
TransferLog /groups/baygroup/logs/access_log
</VirtualHost>
```

其中 www.smallco.com **和** www.baygroup.org **对应同一宿主机器上的不同** IP **地址，此对应关系是定义在** DNS **服务器中的。**

- **在基于名称的虚拟主机模式下：**

例 1**：服务器具有一个** IP **地址——**111.22.33.44**，对应域名** server.domain.tld**，又有两个别名（**CNAME**）——**www.domain.tld **和** www.sub.domain.tld**。具体配置如下：**

```
Port 80
ServerName server.domain.tld

NameVirtualHost 111.22.33.44

<VirtualHost 111.22.33.44>
DocumentRoot /www/domain
ServerName www.domain.tld
...
</VirtualHost>

<VirtualHost 111.22.33.44>
DocumentRoot /www/subdomain
ServerName www.sub.domain.tld
...
</VirtualHost>
```

由于除了 localhost **之外，再无未明确定义的地址/端口对，所以主服务器将仅会接收到** localhost **请求，而** www.domain.tld **在基于** 111.22.33.44 **的虚拟主机中具有最高优先级，所以它实际可被认为是主服务器。**

例 2**：服务器具有两个** IP **地址——**111.22.33.44 **和** 111.22.33.55**，它们分别对应** server1.domain.tld **和** server2.domain.tld**。别名** www.domain.tld **要用来作为主服务器，匹配任何对未在配置中明确定义的地址的请求。我们用** www.otherdomain.tld **作为一个虚拟主机；还要有一个虚拟主机匹配任何对形式为***.sub.domain.tld **的地址的请求，其服务器名（**server name**）为** www.sub.domain.tld**。**111.22.33.55 **用来支持虚拟主机的设置。**

```
Port 80
ServerName www.domain.tld
DocumentRoot /www/domain

NameVirtualHost 111.22.33.55

<VirtualHost 111.22.33.55>
DocumentRoot /www/otherdomain
ServerName www.otherdomain.tld
...
</VirtualHost>
```

```
<VirtualHost 111.22.33.55>
DocumentRoot /www/subdomain
ServerName www.sub.domain.tld
ServerAlias *.sub.domain.tld
...
</VirtualHost>
```

任何对不同于 111.22.33.55 **的地址的访问均会送主服务器来处理；而对** 111.22.33.55 **的不含或含有未知** Host **头域的请求则由** www.otherdomain.tld **来处理。**

总之，除了需要用 NameVirtualHost **为虚拟主机指定对应的** IP **地址外，此种配置与上面的基于** IP **的方法基本是相同的；且在** NameVirtualHost **中使用的** IP **地址的请求只有与** <VirtualHost>**配置中的虚拟主机名相匹配才能获得服务。**

若某个虚拟主机希望能处理对多个主机名的访问，则可以用 ServerAlias **配置命令来定义一些别名，其中可以使用*或？。**

- **基于** IP **地址和基于名称的混合模式：**

有如下配置：

```
Port 80
ServerName www.dtl.ict.ac.cn
NameVirtualHost 159.226.42.93

<VirtualHost 159.226.42.93>
ServerAdmin username@domainname1
DocumentRoot "d:/Program Files/Apache Group/Apache/htdocs"
ServerName homeserver.ict.ac.cn
ErrorLog logs/error.log
TransferLog logs/access.log
</VirtualHost>

<VirtualHost 159.226.42.93>
ServerAdmin username@domainname2
DocumentRoot "d:/Program Files/Apache Group/Apache/htdocs/home93"
ServerName home93.ict.ac.cn
ErrorLog logs/home93/error.log
TransferLog logs/home93/access.log
</VirtualHost>

<VirtualHost 159.226.42.9>
ServerAdmin username@domainname3
DocumentRoot "d:/Program Files/Apache Group/Apache/htdocs/home9"
ServerName home9.ict.ac.cn
ErrorLog logs/home9/error.log
TransferLog logs/home9/access.log
</VirtualHost>
```

其中定义了三个名字分别为 homeserver.ict.ac.cn、home93.ict.ac.cn、home9.ict.ac .cn 的虚拟服务器。它们对应的 IP 地址分别是：159.226.42.93、159.226.42.93、159.226.42.9。其中既有不同的 IP 地址，也有同一 IP 上的不同的域名，实现了两种模式的混合。但事先要把域名与 IP 地址的对应关系加入到 DNS 服务器中，以能正确解析域名，而找到相应的机器。

上面的三个服务器分别有各自的主页、访问日志记录、错误日志记录、管理员的邮件地址等。使用如上的方法建立的虚拟服务器，可以实现在一台机器上，在用户来看，是独立的多个服务器，从而可以方便地实现为多个单位或多种用途提供 WWW 发布服务。

- 基于端口的虚拟主机设置：

服务器有一个 IP 地址 111.22.33.44，对应域名 www.domain.tld，若我们无法为服务器新增一个地址或别名（这需要有 DNS 服务器的配合），则可以使用基于端口的方式来实现不同配置的多个虚拟主机。具体配置如下：

```
Listen 80
Listen 8080
ServerName www.domain.tld
DocumentRoot /www/domain

<VirtualHost 111.22.33.44:8080>
DocumentRoot /www/domain2
...
</VirtualHost>
```

其中对 www.domain.tld 的端口 80 的请求将由主服务器来处理，而对 8080 端口的请求则交给虚拟主机完成。

- ServerPath 配置命令的使用：

对于基于名称的虚拟主机，要进行正确的匹配，客户端必须发送正确的 Host 头域。但过时的 HTTP/1.0 客户程序并不支持此头域，这样 Apache 将不知道它要访问哪个虚拟主机，而用优先级高者（主虚拟服务器）服务之。为了实现尽可能的向后兼容性，我们可以在主虚拟服务器上创建一个页面（如下的/www/subdomain/index.html），其中包含了带有标识各个基于名称的虚拟主机的 URL 地址前缀的链接。

```
NameVirtualHost 111.22.33.44

<VirtualHost 111.22.33.44>
# primary vhost
DocumentRoot /www/subdomain
RewriteEngine On
RewriteRule ^/.* /www/subdomain/index.html
...
</VirtualHost>

<VirtualHost 111.22.33.44>
DocumentRoot /www/subdomain/sub1
ServerName www.sub1.domain.tld
ServerPath /sub1/
```

```
RewriteEngine On
RewriteRule ^(/sub1/.*) /www/subdomain$1
...
</VirtualHost>

<VirtualHost 111.22.33.44>
DocumentRoot /www/subdomain/sub2
ServerName www.sub2.domain.tld
ServerPath /sub2/
RewriteEngine On
RewriteRule ^(/sub2/.*) /www/subdomain$1
...
</VirtualHost>
```

由于 ServerPath **命令的设置，对** http://www.sub1.domain.tld/sub1/**地址的请求将总由** sub1 **虚拟主机来处理。而对** http://www.sub1.domain.tld/**的请求，只有包含了正确的** Host **头域，才会送给** sub1 **虚拟主机；若没有该头域，则客户端将收到来自主虚拟服务器的上面提到的页面。**

需要注意的是，若没有给出正确的 Host **头域，则对** http://www.sub2.domain.tld/sub1/**的请求将总是被** sub1 **虚拟主机来处理。**

其中的 RewriteRule **命令用来使提供了正确的** Host **头域的客户端可以使用两种** URL **形式的任何一种，即带有** URL **前缀的形式及没有前缀的形式（关于** RewriteRule **命令的相关用法未被本书收入，请参阅** Apache **服务器软件附带的相关文档）。**

第6章 Apache 服务器的模块结构

在了解了如何安装使用 Apache 服务器后，从本章开始对 Apache 服务器的设计和实现进行介绍，先介绍 Apache 的整体设计思想——模块化结构。但思想是通过具体实现表达出来的，所以本章也引用了较多的 Apache 源代码，读者可以结合本书对具体实现的介绍进行理解。

第1节　模块接口结构概述

Apache 服务器，为了功能扩充的方便采用了一种模块结构，来组织构成 Apache 的各个功能部分。该模块结构充分体现了封装和抽象的思想。而实现则是登记和派送的机制。

在该结构体系中，每个模块均必须遵从如下的标准接口（API）：

1. 每个模块均有一个 module 结构类型的接口控制块，其中按规定次序定义了一些数据结构和句柄，由主程序在适当的时候访问和调用。

2. 每个模块均有若干配置信息块存在主程序的相应配置信息链表里，其中存放与本模块相关的配置信息。配置信息块是与特定环境相关联的，即或是针对特定虚拟服务器（或虚拟主机）、或是针对特定目录。

3. 每个模块均有一个单元类型为 command_rec 结构的配置命令列表，其中存放本模块可处理的配置命令及相应的处理方法句柄。

4. 每个模块均有一个单元类型为 handler_rec 结构的返回响应处理句柄列表，其中给出本模块可在指定情况下使用相应的处理方法句柄返回请求的响应。

5. 以上四者之间的关系是：module 结构为总控，其中包含了配置命令列表结构的指针、返回响应处理句柄列表结构的指针和对配置信息块进行初始化的方法的句柄（调用该方法即可完成配置块的创建，返回该块的指针）；配置命令列表中给出的处理方法则完成对实际定义的相应配置命令的处理，实际就是把有关信息存到配置信息块的特定位置。实际例子请参阅附录 Apache 各模块配置接口定义；返回响应处理实际上只是 Apache 处理请求的一个阶段，其它阶段的处理句柄都是直接定义在 module 结构中，但在本阶段中，某个模块可能要针对不同情况（由 handler_rec 结构的 content_type 域指示）定义多个处理句柄，故单独将其拿出来，并设置一个列表存储。

关于上述各种数据结构的详细介绍，参阅附录 Apache 各模块配置接口定义和第 7 章第 3 节。

在 Apache 主程序读配置文件之前首先要进行初始化。表现为由所有基本（缺省）模块中接口控制块的 create_server_config 和 create_default_per_dir_config 方法分别完成本模块的针对服务器的和针对目录的配置信息块的初始化。而在配置文件中指定动态装入的模块则用 Apache 主程序中的 ap_single_module_config 方法来完成单个模块的配置信息块初始化。

上述配置信息块的初始化要与模块自身的初始化相区分。module 结构中的 init 句柄完成模块自身的初始化。

各个模块的接口控制块连成链，由主程序在处理请求时进行遍历。遍历的目的是查找各个模块中登记的“声称对请求的某一处理步骤负责”的方法，并在找到时，实际地去调用它。这样 Apache 对客户请求的处理即是在不同处理阶段去驱使不同的“自告奋勇登记”的处理方法，“登记”所遵循的规则及涉及的数据结构的定义的汇总就是这里要讲的“API”。

各模块的配置信息块实际是可能有多个的，每个对应某种特定的环境对象，如目录或虚拟主机等。借用面向对象理论的说法：这里的模块配置信息块的结构定义相当于类，是静态的；而在服务器启动后处理配置命令时生成的各个模块的多个填充了实际数据的配置信息块则相当于对象，是动态的。而应该生成多少个对象，则是与配置文件中不同的处理环境相对应的，比如每定义一个虚拟服务器，各模块便会创建一个配置信息块存放对应的配置信息。

第2节 Apache 服务器 API

本部分将介绍 Apache 模块 API 及其中涉及到的数据结构。尽管 API 现在仍在不断地变化，但您仍有必要读一读本部分内容，因为有了一定的基础，您极有可能会享受到自己的代码集成到 Apache 中运行的乐趣。特别是代码不是以 CGI 的方式，而是作为 Apache 的一部分来运行！

为了简洁，下文中的某些引自 Apache 源代码的数据结构定义是不完整的。具体的取舍与选作示例的 Apache 特定模块相关。

6.2.1. 基本概念

我们首先来看 API 中涉及的各种基本概念，以及它们在 Apache 代码中的表现形式。

请求处理方法

Apache 中将对请求的处理过程分成若干阶段(这一点与 Netscape 的 www 服务器 NetSite 的处理相似)。它们是：

- 地址转换阶段：URI 地址到文件名的转换；
- 身份确认阶段：访问者 ID 的确认（即是否是一个合法用户）；
- 身份认证阶段：访问者身份认证（即该用户是否被授权）；
- （非身份认证的）权限确认阶段：除身份认证以外的其它访问控制；
- MIME 类型识别阶段：确定被访问对象的 MIME 类型；
- 补充（Fixup）处理阶段：到现在为止，基本模块在此阶段还没有任何的操作，但为将来扩展留下了余地（比如：SetEnv 配置命令的处理，把它放在其它任何阶段均不合适）；
- 响应阶段：给客户返回响应；
- 日志记录阶段：对此次请求处理进行日志记录。

这些阶段的具体处理是：依次去查找一个模块链中的每个模块，看它是否定义了对应当前阶段的处理方法，如果有，则去调用该方法。某阶段的处理方法下面我们也称为某处理方法，如“响应处理方法”等。所有处理方法根据处理结果可以分为三种：

- 处理了请求，并返回 OK（与下文提及的 DECLINED 同，均是 Apache 中定义的常量）；
- 拒绝处理请求，返回 DECLINED，这种情况下从外部看，就好象根本没有定义该处理方法一样；
- 报告错误，返回一个 HTTP 错误码，这会终止正在进行的正常处理，并发给客户一个指示有错误发生的文档，该文档由 ErrorDocument 设置的。该错误最后还要写入

错误日志。

在大多数阶段，找到的第一个定义有处理方法的模块的处理即会使该阶段结束（即其它定义的本阶段处理方法将得不到调用）。但是对于日志记录、补充处理和非身份认证的权限确认阶段，所有定义的处理方法均会得到调用（除非有错误发生），而且某些模块可能会基于被请求对象的 MIME 类型定义一个派送表（即本章第 1 节提到的单元类型为 handler_rec 结构的返回响应处理句柄列表），其中每个 MIME 类型对应一个响应处理方法。模块中还可以通过定义一个对应“ */* ”（可以匹配任何 MIME 类型）的响应处理方法处理所有的请求。但是这样的包括通配符的 MIME 类型的匹配优先级是最低的。服务器总是试图去与一个更明确、特指的 MIME 类型匹配，除非这样的类型根本没定义或者均无法匹配。

上述的处理方法有且仅有一个传入参数（为 request_rec 结构类型的变量），返回值是一个整数（如 OK、DECLINE 等）。

模块概述

下面，我们以 CGI 模块为例来介绍模块的结构。CGI 模块包含对 CGI 脚本请求的响应和 ScriptAlias 命令的处理。它实际上比大多数模块都要复杂。但如果我们只选一例的话。CGI 模块无疑是最佳选择，因为它涉及的东西相对其它模块都更全面。

CGI 模块定义了一个响应处理方法来处理对 CGI 脚本的请求并返回响应。由于还涉及 ScriptAlias 命令，它还定义了用于地址转换（识别与替换 ScriptAlias 涉及的 URI 地址）和用于类型检查的处理方法。实际上，任何经过 ScriptAlias 处理的请求均被认为是脚本类型的。

模块中还需要维护一些涉及虚拟主机的信息，因此模块结构中还包含构造配置信息块的方法的指针、合并两个配置信息块的方法的指针（将均涉及 ScriptAlias 定义的主服务器配置信息块和某虚拟主机配置信息块合并）。

最后，CGI 模块中还包含处理 ScriptAlias 配置命令的代码。该模块中只处理这样一条命令，而其它模块则完全可能处理非常多的命令，因此模块中还有一个命令处理表，在配置命令与对应的处理方法之间建立映射，其中具体声明了该模块负责处理的配置命令、它们的适用范围及它们的处理方法被调用的方式。

某些命令处理方法的参数类型如下：

pool：是一个指向资源池（source pool，请参阅第 7 章第 3 节）结构的指针类型。它们被服务器用来在处理请求或自身的配置时跟踪已分配的内存、使用的文件句柄等资源。这样在请求处理完成后，占用的内存可以被完全释放掉，打开的文件也不会被忘记关闭，从而避免很多错误的出现。

cmd_parms：该结构中存放读取的配置文件的信息及状态信息，提供给配置命令处理方法使用。

CGI 模块的大致内容如下：

```
/* Declarations of handlers. */

int translate_scriptalias (request_rec *);
int type_scriptalias (request_rec *);
int cgi_handler (request_rec *);

/* Subsidiary dispatch table for response-phase handlers, by MIME type */

handler_rec cgi_handlers[] = {
```

```
    { "application/x-httpd-cgi", cgi_handler },
    { NULL }
    };

    /* Declarations of routines to manipulate the module's configuration
     * info.  Note that these are returned, and passed in, as void *'s;
     * the server core keeps track of them, but it doesn't, and can't,
     * know their internal structure.
     */

    void *make_cgi_server_config (pool *);
    void *merge_cgi_server_config (pool *, void *, void *);

    /* Declarations of routines to handle config-file commands */

    extern char *script_alias(cmd_parms *, void *per_dir_config, char *fake, char
*real);

    command_rec cgi_cmds[] = {
    { "ScriptAlias", script_alias, NULL, RSRC_CONF, TAKE2,
        "a fakename and a realname"},
    { NULL }
    };

    module cgi_module = {
       STANDARD_MODULE_STUFF,
       NULL,                /* initializer */
       NULL,                /* dir config creator */
       NULL,                /* dir merger --- default is to override */
       make_cgi_server_config,      /* server config */
       merge_cgi_server_config,     /* merge server config */
       cgi_cmds,                /* command table */
       cgi_handlers,             /* handlers */
       translate_scriptalias,         /* filename translation */
       NULL,                /* check_user_id */
       NULL,                /* check auth */
       NULL,                /* check access */
       type_scriptalias,            /* type_checker */
       NULL,                /* fixups */
       NULL,                /* logger */
       NULL                 /* header parser */
    };
```

实际上，Apache **中已包含了一个名为** mod_example.c **的文件，其中定义了** example **模块，相当于给出了一个模块的模板，想自己编写模块的读者可以参考之。**

6.2.2. 请求处理方法的工作方式

传给各个请求处理方法的唯一参数是一个 request_rec 结构。该结构描述了客户对服务器的请求。大多数情况下，与客户的每个连接仅创建一个 request_rec 结构。HTTP1.1 中允许客户建立一次连接，而可以发出多次请求，从而可以在一次连接中取回完整页面包含的多个文件，每个文件对应的一次请求，这样效率大大提高，此也称为持续连接机制。Apache 中是用 conn_rec 结构描述一次连接的。

request_rec 概貌

request_rec 中首先包含指向如下结构的指针：资源池 pool，其中记录各种占用的资源，在服务器处理完该请求后释放之；存放关于服务器（server）、当前连接（connection），尤其是当前请求的信息的结构。

在这些信息中，最重要的是描述被请求对象属性的若干字符串，包括它的 URI 地址（uri）、文件名（filename）、内容类型（Content Type ，content_type）和内容编码（Content Encoding，content_encoding）（这些值会在地址转换与类型检查的处理方法中被赋上）。此外，HEAD 请求标志（header_only）、使用协议（protocol）及请求方法（method）等则是对当前请求的基本描述。

其它一些常用的表（table，请参阅第 7 章第 3 节）形式的数据项还有：描述客户原始请求中 MIME 头域的表（headers_in），响应中要包含的 MIME 头域表（headers_out，任何模块均可以根据需要设置它）及一些在处理请求中创建的子进程所需的环境变量构成的表（subprocess_env）。这些表由 ap_table_get 和 ap_table_set 例程来处理。

注意：各模块中内容处理方法（content-handlers）不能使用 ap_table_*方法来设置 Content-type 头域，而应该使用 request_rec 结构中的 content_type 项进行字符串的赋值完成，如：

```
r->content_type = "text/html";
```

实际上，HTTP 请求的响应中可以设置许多头域，Content-type 即为其一。这些头域本是设置在 request_rec 结构的 headers_out 表中，在发送响应时发送出去。而 Content-type 头域由于经常被用到，则专门设置了一个变量存放之。

最后，request_rec 结构中还有指向两种特定数据结构（向量）的指针。向量中又包含了指向每个模块配置结构的指针，从而实现对后者的间接访问。更明确地说，上述两种数据结构分别包含两种指针，它们分别指向：各模块为特定环境（如.htaccess 文件和<Directory>配置片段）创建的数据块和存放模块内部处理信息的数据块（这种设置实际上使得模块中的某些处理方法可以把信息传递给后面被调用的处理方法）。request_rec 还包含指向 server_rec 结构的指针。每个 server_rec 结构中包含一个（虚拟）服务器的配置数据，其中也有上述配置向量结构。

下面是一个 request_rec 结构的简略定义，只给出了最常用的数据项：

```
struct request_rec {

  pool *pool;
  conn_rec *connection;
  server_rec *server;
```

```
    /* What object is being requested */

    char *uri;
    char *filename;
    char *path_info;
    char *args;          /* QUERY_ARGS, if any */
    struct stat finfo;   /* Set by server core; st_mode set to zero if no such
file */

    char *content_type;
    char *content_encoding;

    /* MIME header environments, in and out.  Also, an array containing
     * environment variables to be passed to subprocesses, so people can
     * write modules to add to that environment.
     *
     * The difference between headers_out and err_headers_out is that
     * the latter are printed even on error, and persist across internal
     * redirects (so the headers printed for ErrorDocument handlers will
     * have them).
     */

    table *headers_in;
    table *headers_out;
    table *err_headers_out;
    table *subprocess_env;

    /* Info about the request itself... */

    int header_only;     /* HEAD request, as opposed to GET */
    char *protocol;      /* Protocol, as given to us, or HTTP/0.9 */
    char *method;      /* GET, HEAD, POST, etc. */
    int method_number;   /* M_GET, M_POST, etc. */

    /* Info for logging */

    char *the_request;
    int bytes_sent;

    /* A flag which modules can set, to indicate that the data being
     * returned is volatile, and clients should be told not to cache it.
     */

    int no_cache;
```

```
    /* Various other config info which may change with .htaccess files
     * These are config vectors, with one void* pointer for each module
     * (the thing pointed to being the module's business).
     */

    void *per_dir_config;   /* Options set in config files, etc. */
    void *request_config;   /* Notes on *this* request */

};
```

request_rec 结构的构造

大多数 request_rec **结构是在读取客户的** HTTP **请求时创建的，其中的某些字段也在该时刻填充。但也有几个例外：**

- **如果请求的是一个** ImageMap**（可能是一个点击式的图象，即** *.var **文件，或是一个返回本地** URI **地址的** CGI **脚本），则用户得到的响应内容的** URI **地址将不是其最初提供的** URI **地址。这种情况，我们说服务器进行了一次内部重定向（**internal redirect**），其间便为新的** URI **地址又构造了一个** request_rec **结构。就好象客户本来请求的就是这个** URI **地址一样。**
- **若某些处理方法报告错误，并且又设置了** ErrorDocument **配置时，也会发生同上的内部重定向。**
- **还有一种情况，某个处理方法有时会需要知道在其他请求执行后的情况。比如，目录列表模块（**mod_index**）需要知道用户对列表页面中每个目录项链接的请求会对应什么** MIME **类型，以便能确定它应该为该目录项使用什么图标。**
  具体是使用 ap_sub_req_lookup_file **和** ap_sub_req_lookup_uri **等方法来构造一个子请求（**sub_request**）。该请求也会有对应的** request_rec **结构被创建并处理，直至该请求的响应之前（若子请求的** URI **地址与原请求的** URI **地址对应相同目录，则该子请求不会再进行访问控制检查）。**
  特别地，SSI **的处理模块，也要创建子请求，但它还进一步通过** ap_run_sub_request **方法调用了返回响应处理方法。**

请求的处理、拒绝与错误码应答

如前所述，每个被调用来处理 request_rec **的处理方法，必须返回一个指示处理结果的整数，可能的取值情况是：**

OK**：客户请求已被正确处理，这可能结束当前阶段的处理；**

DECLINED**：尽管没有错误发生，但是当前模块拒绝（不处理）该请求。服务器会再找其它处理方法。**

HTTP **错误码：这种情况会使服务器退出对请求的处理。**

注意，如果返回的错误码是 REDIRECT**，则处理模块还应在当前请求的** headers_out **表（在** request_rec **结构中，记录要返回的响应中包含的头域）中设置一个** Location **头域，来指示客户程序应重定向到何处。**

响应处理方法

大多数阶段的处理方法仅是对 request_rec 结构的几个域赋值（在访问控制检查阶段则仅是返回一个合理的错误码），只有响应处理方法实际地给客户发回一个响应。

响应以 HTTP 响应头开始，该响应头的发送是由 ap_send_header 方法完成的（但对于基于 HTTP 0.9 的请求，该方法则不做任何事）。若请求被标记为 header_only（对应 HTTP HEAD 请求，参阅 HTTP 协议），则该响应头就是整个响应了，否则还要发送一个请求体给客户。所基于的方法有：ap_rputc 与 ap_rprintf（用于在内部构造的内容输出）及 ap_send_fd（用来将某些文件内容作为响应输出）。

如下为 Apache 中唯一一个响应 GET 请求的处理方法，在 http_core.c 中定义。它同时也负责处理带附加条件的 GET 请求，其中调用 ap_set_last_modified 方法对客户请求中可能设置的 If-modified_since 头域的值进行检查，并返回一个整数代码（非 0 的 USE_LOCAL_COPY 表示客户可以使用其本地已有的备份），其中的 ap_set_content_length 本无需有返回值，但为了格式的“规整”和“美观”，它也返回一个错误码。

```
int default_handler (request_rec *r)
{
    int errstatus;
    FILE *f;

    if (r->method_number != M_GET) return DECLINED;
    if (r->finfo.st_mode == 0) return NOT_FOUND;

    if ((errstatus = ap_set_content_length (r, r->finfo.st_size))
        || (errstatus = ap_set_last_modified (r, r->finfo.st_mtime)))
        return errstatus;

    f = fopen (r->filename, "r");

    if (f == NULL) {
        log_reason("file permissions deny server access",
                 r->filename, r);
        return FORBIDDEN;
    }

    register_timeout ("send", r);
    ap_send_http_header (r);

    if (!r->header_only) send_fd (f, r);
    ap_pfclose (r->pool, f);
    return OK;
}
```

与上述已较复杂的处理相比，某些处理方法的如下处理则很简单：其一，仅返回一个错误代码，而不生成任何输出，这时服务器会自动给出一个错误响应。其二，调用

internal_redirect 来通过内部的重定向将“皮球”踢给其他处理方法，并返回“OK”。

注意：在非响应阶段的处理方法中调用 internal_redirect 会使系统发生混乱。

身份认证处理方法

若被请求对象所在的目录不要求身份认证，则将不会调用认证阶段的处理方法。

一般的身份认证配置信息存在 core 模块的基于目录的配置信息数据结构中。对它们的访问是通过 ap_auth_type、ap_auth_name 和 ap_requires 等来实现的。

关于身份认证的具体处理请参阅第 7 章第 9 节auth 模块的实现情况的介绍。

重定向时日志处理方法的特殊考虑

当一个请求被重定向时，就出现了如何进行日志记录的问题。Apache 的处理是：将连续的重定向组织成一个 request_rec 结构的链。它们是通过 r->prev 和 r->next 指针连成的。这种情况下传递给日志阶段处理方法的 request_rec 结构是对应最原始客户请求的那一个。需要注意的是：只在该链的最后一个 request_rec 中记录的 bytes_sent 域的值才是正确的，因为也只有它是真正发回响应的。

6.2.3. 资源分配和资源池

设计和实现一个服务器池模式的服务器的最大问题就是资源遗漏问题，即分配的资源（内存、打开的文件等）忘记了回收，而资源池方法为服务器提供了一种可以在使用完资源后，使之自动被回收，而避免遗漏问题出现的途径。

其工作方式是将处理请求时分配的内存、文件句柄等与专为该请求分配的资源池建立关联。该池是一种用来跟踪资源的数据结构。

当请求处理完毕时，该资源池将被清除，即与之关联的内存被释放以待重用。相关的文件被关闭，其它与该池关联的清除方法也将被调用。当清除完时，我们可以确信所有与该池关联的资源均已被释放，而无一遗漏。

服务器重启和为每个服务器分配内存等资源的处理方法与上相同，如用配置资源池（configuration pool）来跟踪读配置文件和处理其中的配置命令时分配的资源（如为各服务器中每个模块的配置信息块分配的内存、打开的日志文件与其它文件所用文件句柄等）。当服务器重启时，它会重新读取并处理配置文件。但在此之前，配置资源池会被清除掉，从而使得先前用过的内存和文件描述符等资源可以重复使用。

需要说明的是，并非一定要使用上述的“缓冲池”机制。但对日志处理方法来说，该机制却是不可或缺的。因为你需要记录一些清除工作，以使在服务器重新启动时，日志文件被关闭（最好的方法是用 ap_pfopen，因为它在某些子进程执行前，如 CGI 脚本，会先将涉及的文件关闭）。缓冲池机制有两大优点：分配的内存从不会遗漏（即使是匆忙分配的一个小小字符串空间也是如此）；另外，内存分配中，ap_palloc 比 malloc 更快。

本部分，我们首先来看内存的分配，然后再讨论其它资源。

内存的分配

在缓冲池中内存的分配是通过调用方法 ap_palloc 来实现的。它带两个参数，一个是指向资源池结构的指针，另一个记录分配内存的数量（以 char 为单位）。在请求处理方法中，获取资源池的方法是取相关的 request_rec 结构中的 pool 数据项。如下所示的一段代码，您会在各个模块的代码中多次见到：

```
int my_handler(request_rec *r)
{
    struct my_structure *foo;
    ...

    foo = (foo *)ap_palloc (r->pool, sizeof(my_structure));
}
```

注意这里没有所谓的“ap_pfree”（如与 malloc 对应有 free）。用 palloc 方式分配的内存只有在清除资源池时才会释放掉。这意味着用 ap_palloc 比用 malloc 花费少。对前者来说，需要做的仅是：对申请的空间大小取整，创建一个指针，并进行范围检查。

需要注意的一种可能是：大量的使用 ap_palloc 会使服务器进程变得过于庞大。有两种方法可以解决这一问题。简单地说，你可以使用 malloc 代替 ap_palloc，但需保证所有分配的内存均已被释放，或你可以创建一个主资源池的子池，使用子池来分配资源，并定期地清除掉它。后一种技术将在随后的关于子资源池的创建和处理部分中提到，并在目录列表模块中得到应用，来避免当目录中含有成千上万的文件时耗费过多的资源。

特殊处理的内存分配

ap_palloc 仅完成空间的分配，还有几个方法除完成空间分配外，还要进行一些特定操作，它们是：

ap_pcalloc 方法与 ap_palloc 接口相同，但在返回前会将分配的内存进行初始化（清 0）。

ap_pstrdup 方法接收一个 pool 类型值和一个 char *类型值为参数，完成内存分配和该字符串的复制，并将复制的字符串的指针返回。

ap_pstrcat 是一个可变参数的方法，它要有一个 pool 类型和至少两个 char *类型的参数，最后一个必须是 NULL。它完成所有字符串的复制和连接，并为得到的新串分配足够的内存空间，例如：

```
ap_pstrcat (r->pool, "foo", "/", "bar", NULL);
```

将返回一个指向 8 字节大小的空间的指针，该空间的内容是“foo/bar”。

常见的一些资源池变量

资源池也是有其生命周期的。但在 http_main 代码文件中定义了一些静态的资源池，它们在适当的时候会作为参数传给 http_main 外部的方法。它们是：

- permanent_pool
  从来不会作为参数传给其它方法，它仅以其它所有资源池的祖先身份出现。
- pconf

permanent_pool 的子池。
在配置处理周期开始时创建，直到服务器关闭或重启；作为参数传给所有在配置处理过程中调用的方法，或者通过 cmd->pool，或者以"pool *p" 的形式作为参数传递给各个模块的 init()方法。

- ptemp
  表示父进程中的 ptrans，以与子进程中的后来定义的 ptrans 相对应。
  permanent_pool 的子池。
  在配置处理周期开始时创建，直到配置命令解析结束；作为参数以 cmd->temp_pool 的形式传给配置处理过程中调用的各个例程。它为各个配置处理方法中调用的各个方法提供临时空间，并在配置处理完毕时被清除。
- pchild
  permanent_pool 的子池。
  在子进程或线程创建时被创建，直到该进程或线程销毁。
  传递给各个模块的 child_init 方法，就在 child_exit 方法被调用时被销毁。
- ptrans
  应该是 pchild 的子池，但现在是 permanent_pool 的子池。
  在子进程进入 accept 方法的监听连接循环时被清除，以 connection->pool 的形式被使用。
- r->pool
  对主请求来说，它是 connection->pool 的子池；但对于子请求，则是主请求的资源池的子池。
  它在请求处理结束前一直存在（即在 process_request 方法执行完后在 ap_destroy_sub_req 或 child_main 方法中被销毁）。
  注意 r 本身即是从 r->pool 中分配的；即 r->pool 先创建，r 是从 r->pool 中第一个申请资源的。

在大多数场合下，使用的资源池都是 r->pool。但你会看到其它的资源池，如 pchild，对某些模块也是非常有用的，例如某些在子进程执行中需要与数据库建立连接的模块，在子进程退出时需要将已用的大量资源释放掉。

你还会看到与资源池使用相关的一些问题。例如，把从 r->pool 中申请空间来存放的值赋给 connection->user，而 r->pool 要比 connection 更短命。因为一次连接中可能会涉及多次请求处理。请求已成历史，资源申请也成历史，现实的指针还指向历史的遗迹，自然是不正常的。所以正确的做法应该是从 connection->pool 中申请空间。

在 mod_include/mod_cgi 中还有一个很有意思的情况。这两个模块在池中登记要清除的是子进程！如果登记在 r->pool 中，则当子请求结束时将调用 wait()来等待子进程结束。在 mod_include 中，等待的延迟甚至可达 3 秒或更长，并且经常发生。相反，若登记在 r->main->pool 中，则只有当整个请求结束处理时（即输出已传给客户并完成了日志记录）才会进行资源的释放，从而避免问题的发生。

跟踪打开的文件等

如上所示，资源池也被用来跟踪除内存以外的其他类型的资源。最常见的便是打开文件。常见的方法为 ap_pfopen，它以 pool 类型值和两个字符串为参数，其中的字符串与 C 的标准函数 fopen 的对应参数意义用法相同。如：

```
FILE *f = ap_pfopen (r->pool, r->filename, "r");
```

```
if (f == NULL) { ... } else { ... }
```

还有一个 ap_popenf 例程，内部的多个 open 操作是并行的。这些例程都可以使资源池清除时，文件被及时关闭。

与内存分配不同的是：与 pfopen 和 popenf 对应的还有用来关闭文件的 ap_pfclose 和 ap_pclosef 方法。使用 close 方法并不是必须的。因为使用的文件处理方法最终会被释放，但若你的模块需要打开大量文件，则你就需要及时地将无用文件关闭。这是因为，在许多系统中，一个进程可以打开的文件数目是有限的。使用上面的方法打开的文件必须用这两个方法来关闭。因为不然的话，在诸如 Linux 的系统中会发生致命错误，表现在当同一个文件处理方法被关闭两次时，系统会出现异常。

更细的控制——对子请求中子资源池的创建和处理

尽管可能性很小，但仍会出现这样的情况：太自由地使用 palloc 或类似原语会造成资源的“挥霍无度”。解决的办法是创建一个子资源池，并使用它来进行资源的分配，并在最终清除或销毁该子资源池，完成资源的释放。一般子资源池并非必须的，迄今出现的唯一需求是在目录列表模块中。即使如此，也只在目录中文件数目极多时才会有；相反多余的使用反倒会使代码繁乱。

创建子资源池的原语是 ap_make_sub_pool，它以另一资源池（父池）为参数，当父池被清除时，子池也会被清除。此外，还可以随时调用 ap_clear_pool 和 ap_destroy_pool 分别清除或销毁子池。它们之间的区别在于 ap_clear_pool 仅将池中的资源释放掉，而 ap_destroy_pool 还要将池自身所占空间释放，从而销毁池本身。在前一种情况下，你还可以从池中分配新的资源，然后再清除，进而周而复始；后一种情况则是“一切都已随风逝去”。

子请求有其自己的资源池，该池又是主请求的资源池的子池。收回子请求中分配的资源要用 ap_destroy_sub_request。它同时也会将子池本身所占资源释放掉。在调用这个方法之前，一定要把子池管理的空间中存放的信息复制到更“安全”的地方（如 request_rec 结构中存放被请求文件名称的 filename 数据项）。

但大多数情况下，你不一定非要去调用这个方法。因为一个普通的子请求只会占用 2k 左右的内存，而当主请求资源池清除时，它也会被清除。只有在一个主请求中存在非常多的子请求时，才需要考虑 destroy 方法的使用。

6.2.4. 配置命令

Apache 服务器的一个设计目标是与 NCSA 1.3 服务器能保证兼容，即可以处理相同的配置文件，成为 NCSA 服务器的替代品。设计的另一目标是把尽可能多的功能移到模块中，各个模块与服务器的核心联系尽可能地少，那么满足上面两个目标的办法就是将大多数配置命令的处理从单一的服务器核心中移到模块里。

但是，仅仅为模块定义一命令映射表并不足以将其与核心分开。服务器还必须记录各条命令定义的信息以使它能在处理后发挥作用。这就涉及到了对模块私有信息的维护。这些信息或是针对某个虚拟服务器的，或是针对某个目录的。大多数情况是属于后者，包括特定的访问控制和身份认证信息，以及根据文件扩展名确定文件类型涉及的 AddType 和 DefaultType 等配置信息。一般地，任何可以以目录的观点来管理和配置的内容均以目录的方式来控制；而针对虚拟服务器的配置一般地由处理象 Alias 和 Redirect 的模块来负责，因为这两种配置在将请求定位到一个具体的文件之前发挥作用。

要“模仿”NCSA 服务器，Apache 还要能处理 NCSA 服务器使用的针对目录的配置文件（通常命名为.htaccess）。因此在完成 URI 到文件名的转换后而在其它处理之前，服务器会遍历文件系统中的指定路径的各级目录，读取可能存在的.htaccess 文件。读到的信息将和服务器本身的配置文件内容进行合并。

最后，在处理完一次请求后，需要释放处理过程中占用的存储空间。采用的方法是与解决其它地方的类似问题相同，即使用资源池结构。

针对目录的配置结构

本节以 mod_mime.c 为例来看针对目录的配置信息是如何处理的。mod_mime.c 中定义了文件类型判定处理方法，模仿 NCSA 服务器根据文件扩展名来确定文件的类型。下面的代码实现了对 AddType 和 AddEncoding 配置命令的处理。由于这些命令可以出现在.htaccess 文件中，所以必须在模块针对目录的私有数据结构中保存。如下代码所示，该结构实际由两个分别用于 MIME 类型和编码信息的表构成：

```
typedef struct {
    table *forced_types;      /* Additional AddTyped stuff */
    table *encoding_types;    /* Added with AddEncoding... */
} mime_dir_config;
```

当服务器处理包含有 MIME 相关配置命令的配置文件或<Directory>片段时，它需要创建一个 mime_dir_config 结构来保存相应信息，以便其能发挥作用。完成的实际过程是：服务器到 mod_mime 的接口控制块 module 中取得创建目录配置结构的方法句柄（create per_dir config），调用该方法需要传递两个参数：配置信息针对的目录名（针对 srm.conf 时为 NULL）和资源池指针。

如果我们读的是.htaccess 文件，则该资源池就是针对每个请求建立的资源池；否则它是专用于处理配置信息的资源池，到重启时将会被清除。无论是哪一种方式，只要是从资源池中申请的空间，在该池清除时均会被清除。

对于 MIME 模块，针对目录的配置结构创建方法仅完成如下操作：“palloc”上述结构，并在其中创建两个表。如下所示：

```
void *create_mime_dir_config (pool *p, char *dummy)
{
    mime_dir_config *new =
      (mime_dir_config *) ap_palloc (p, sizeof(mime_dir_config));

    new->forced_types = ap_make_table (p, 4);
    new->encoding_types = ap_make_table (p, 4);

    return new;
}
```

现在，我们假设已经读入了一个.htaccess 文件，则在读下一级目录前我们已经建立了一个目录配置结构。如果新目录中没有.htaccess 文件，则我们可以直接使用原来的目录配置结构，否则我们需要将两个结构合并。

要完成合并，服务器要调用模块的目录配置合并方法（per_directory config merge）。该方法接收三个参数：要合并的两个结构和为合并后的结构提供资源的资源池。对于 MIME

模块，需要做的仅是用父目录配置结构与新的配置结构合并（overlay）：

```
void *merge_mime_dir_configs (pool *p, void *parent_dirv, void *subdirv)
{
    mime_dir_config *parent_dir = (mime_dir_config *)parent_dirv;
    mime_dir_config *subdir = (mime_dir_config *)subdirv;
    mime_dir_config *new = (mime_dir_config *)ap_palloc (p,
                                  sizeof(mime_dir_config));

    new->forced_types = ap_overlay_tables (p, subdir->forced_types,
                                 parent_dir->forced_types);
    new->encoding_types = ap_overlay_tables (p, subdir->encoding_types,
                                  parent_dir->encoding_types);

    return new;
}
```

如果没有定义目录配置合并方法，则服务器将使用子目录的配置信息而忽略掉父目录。对某些模块来说，这样就足够了（如 includes 模块，它的目录配置信息仅包括 XBITHACK 开关状态），所以对于这样的模块。我们完全可以不定义任何的处理方法，而仅是赋一个 NULL 即可。

配置命令处理

现在我们已经清楚了配置数据结构的创建，那么其中的内容是如何填充的呢？对 MIME 模块来说，就是 AddType 和 AddEncoding 配置命令的具体处理了。首先服务器要查找各个模块的配置命令列表。该表定义了配置命令的参数数目和格式、允许使用的范围等。这些信息对大多数配置命令处理方法已足够用。我们来看 AddType 的处理方法，如下所示（AddEncoding 的处理方法类似，不再赘述）：

```
char *add_type(cmd_parms *cmd, mime_dir_config *m, char *ct, char *ext)
{
    if (*ext == '.') ++ext;
    ap_table_set (m->forced_types, ext, ct);
    return NULL;
}
```

该处理方法出奇地简单。其中接收四个参数：两个传送经过解析的配置参数，一个是模块的目录配置结构，一个是指向 cmd_parms 结构的指针。cmd_parms 结构包含一系列处理某些配置命令时需要的内容，包括资源池和对应的虚拟服务器。

该方法出奇地简单部分地在于它没有错误处理部分。实际上，当有错误发生时，应该返回一个错误消息串，而不是返回 NULL。若错误发生在处理主配置文件时，服务器会将该消息串输出到标准错误流中，并立即退出；而若是处理.htaccess 文件时发生，错误信息将被记录到错误日志中（并会指明它的来源），当前请求会收到一个服务器错误应答（HTTP 错误状态码 500）。

MIME 模块的命令处理映射表如下：

```
command_rec mime_cmds[] = {
{ "AddType", add_type, NULL, OR_FILEINFO, TAKE2,
```

```
    "a mime type followed by a file extension" },
{ "AddEncoding", add_encoding, NULL, OR_FILEINFO, TAKE2,
    "an encoding (e.g., gzip), followed by a file extension" },
{ NULL }
};
```

表中的各项的含义依次是：

- 配置命令的名字，如“AddType”；
- 处理该命令的方法，如“add_type”；
- 一个（void *）类型的指针，放在 cmd_parms 结构中传递给命令处理方法——它使服务器可以用同一方法处理多个类似的命令。
- 指示该命令适用范围的掩码。AllowOverride 命令的各个参数都有一个掩码与之对应，另外还有一个，RSRC_CONF，表示该命令可以出现在服务器的配置文件中，但不能在.htaccess 文件中使用。
- 指示命令处理方法接收参数个数和方式的标志。TAKE2 表示要接收两个参数。其它的选择是：TAKE1，表示需要一个参数；FLAG，表示参数应该是一个开关值，是 On 或 Off； RAW_ARGS，表示应该把原始的配置命令参数串（配置命令名除外）直接传递给处理方法；ITERATE，表示处理方法接收参数的方式与 TAKE1 相同，但若定义了多个参数的话，则应去多次调用处理方法；ITERATE2，表示处理方法接收参数的方式同 TAKE2，但当定义有更多参数时，应分多次调用该方法。
- 最后一项给出一个描述配置参数的字符串。这样当实际的配置定义不符合规定时，该字符串将被显示出来给出一个较明确的说明。完全可以只给一个 NULL 值。

定义好这样的表之后，我们就可以来使用它了。具体的使用体现在模块中的处理方法中，例如判定文件类型的处理方法定义如下文所示，其中的目录配置结构实际上是通过 ap_get_module_config 方法从 request_rec 结构的目录配置向量中抽取出来的。

```
int find_ct(request_rec *r)
{
    int i;
    char *fn = ap_pstrdup (r->pool, r->filename);
    mime_dir_config *conf = (mime_dir_config *)
           ap_get_module_config(r->per_dir_config, &mime_module);
    char *type;

    if (S_ISDIR(r->finfo.st_mode)) {
        r->content_type = DIR_MAGIC_TYPE;
        return OK;
    }

    if((i=ap_rind(fn,'.')) < 0) return DECLINED;
    ++i;

    if ((type = ap_table_get (conf->encoding_types, &fn[i])))
    {
        r->content_encoding = type;
```

```
        /* go back to previous extension to try to use it as a type */

        fn[i-1] = '\0';
        if((i=ap_rind(fn,'.')) < 0) return OK;
        ++i;
    }

    if ((type = ap_table_get (conf->forced_types, &fn[i])))
    {
        r->content_type = type;
    }

    return OK;
}
```

针对（虚拟）服务器的配置处理

各个模块中对针对服务器的配置的处理与针对目录的配置处理方法相似，也有一个创建结构方法（create per_server config**）和一个合并方法（**merge per_server config**），在需要将虚拟服务器的配置结构同主服务器的配置结构合并时会调用合并方法。**

针对服务器和针对目录的配置的处理唯一不同之处在于：当一个命令是针对虚拟服务器定义的时，对应的处理方法需要到 cmd_parms **结构中获取对应虚拟服务器的信息。如下为来自** alias **模块的例子，它同时还显示了语法错误的处理过程（注意该处理方法的目录配置结构参数定义成了“** dummy **”，意思是“虚构的、假的”，因为该模块根本没有针对目录的配置信息）：**

```
char *add_redirect(cmd_parms *cmd, void *dummy, char *f, char *url)
{
    server_rec *s = cmd->server;
    alias_server_conf *conf = (alias_server_conf *)
          ap_get_module_config(s->module_config,&alias_module);
    alias_entry *new = ap_push_array (conf->redirects);

    if (!ap_is_url (url)) return "Redirect to non-URL";

    new->fake = f; new->real = url;
    return NULL;
}
```

第7章 Apache 服务器实现原理

第1节 概述

说到 Apache 服务器的实现原理，可以分为两个层次：较为概括性的原理（或者可以说成软件结构设计）和较为具体的实现方法。

对于第一个层次，我们已经在第 6 章中进行了初步的介绍，即模块化的组织。对于具体的实现方法，则在本章进行分析。

对于代码的分析，我们遵循从骨架到细节、从表象到内涵的思路来进行。首先确定 Apache 服务器所有源代码的功能分布；然后再对各个功能模块再进行分析。

7.1.1. 源代码的组织

以 Windows 平台下的 Apache1.3.6 为例，下面给出了源代码的目录结构：

<root>
|---ap：若干.c 文件定义了若干字符串处理方法及 MD5 具体实现。
|---include：存放一系列的.h 头文件。
|---main：存放主程序及核心模块的目录。
|---modules：存放各个功能模块。
|　　|---example：给出了一个模块的源代码例子。
|　　|---experimental：存放实验性模块的目录。
|　　|---proxy：存放 proxy 模块的实现代码。
|　　|---standard：存放一系列的 Apache 模块。
|---os：存放与操作系统机制相关的实现。
|　　|---win32：与 Windows 操作系统相关的实现。
|---regex：存放正则表达式的处理方法。
|---support：存放一些支持工具程序的源代码。

其中 main 目录中的内容是最核心的实现部分，其中最主要的有 7 个文件：

- http_main.c:

负责系统的启动、重启和连接的建立及超时处理。

- http_protocol.c:

完成与客户端直接相关的操作。

- http_request.c:

完成客户端请求的派送。

- http_config.c:

完成配置文件的读取及向命令处理方法（句柄）的派送。

- http_log.c:

完成错误信息的日志记录。出错记录中的另外部分由 http_protocol.c 和 http_request.c 分

别完成。

- http_vhost.c:

与虚拟主机地址相关的功能实现。

- http_core.c:

定义有 Apache 核心部分的模块结构、命令表、命令句柄及文件名转换例程。

其中最后一个文件，严格地说应该放到 modules 目录中，且应该命名为 mod_core.c，但由于该文件实现了 WWW 服务器的最基本功能，即在没有其它任何模块的情况下，整个服务器也可基本正常工作，所以它的重要程度使之被“高看一眼”放到了 main 目录中。

在 modules 目录下，同样由于重要性的原因，proxy 模块也被单独放到一个目录中，其它模块则都放到 standard 目录中，之所以命名为 standard，估计是 Apache 的各位开发者要说这些模块都是符合 Apache 模块规范的标准模块（请参阅第 6 章）。各个模块如下：

- mod_access.c:

完成基于域名与 IP 地址的访问控制。定义并实现了 order, allow 和 deny 等配置命令。

- mod_actions.c:

支持基于 MIME 类型和 HTTP 的脚本的执行。

- mod_alias.c:

处理 Alias 命令和完成文件转换。

- mod_asis.c:

ASIS 文件处理。

- mod_auth.c/ mod_auth_anon.c/ mod_auth_db.c/ mod_auth_dbm.c:

实现基于 HTTP 的身份验证。

- mod_autoindex.c:

实现 HTML 格式的索引快速生成。

- mod_cern_meta.c:

模仿 CERN HTTPD 服务器的 Meta 文件功能，可在响应中放入额外的头域。

- mod_cgi.c:

实现通用网关协议（CGI）。

- mod_digest.c:

完成摘要认证。

- mod_dir.c:

支持对包容在<Directory>和</Directory>中的一系列命令的支持。

- mod_env.c:

支持向 CGI/SSI 脚本传送环境变量。

- mod_expires.c:

支持在目录级的配置文件中设置过期（expire）控制。

- mod_headers.c:

支持 HTTP 头的增加、附加和删除。

- mod_imap.c:

支持 image map（server 根据 client 在 browser 中点击一幅图象的特定位置而执行特定操作的功能，需要在坐标区域和执行动作之间建立一种映射）功能。

- mod_includes.c:

定义了 server 端相关的头文件。

- mod_info.c:

支持与服务器及扩展模块相关的配置信息的显示。

- mod_isapi.c:

实现了 Microsoft 的 ISAPI。

- mod_log_config.c:

实现日志功能。

- mod_mime.c:

实现基于文件扩展名的编码内容与 MIME 类型的对应。

- mod_negotiation.c:

实现关于内容的协议。定义并实现了 CacheNegotiatedDocs 配置命令。

- proxy*.c:

Proxy 模块部分，实现了支持缓存的 HTTP/FTP 代理功能。

- mod_rewrite.c:

使用一基于规则的改写引擎来改写请求的 URL 地址。

- mod_setenvif.c:

用来设置环境变量，其基于从请求包头或属性到正则表达式字符串的映射。

- mod_so.c:

用来完成在运行时将 Apache 模块装入。

- mod_speling.c:

支持对敲错的 URL 地址的智能更正。

- mod_status.c:

完成与 Apache 服务器的性能及运作相关的信息的显示。

- mod_userdir.c:

完成与 UserDir directive 相关的处理。

- mod_usertrack.c:

基于 Cookie 协议，实现对用户访问某一站点的路径的跟踪。

第2节　主控程序——Apache 控制处理流程

Apache 有两种工作模式：单进程模式和多进程模式。前者是为了调试方便而专门设置的，需要修改 http_main.c 中的 one_process 变量使之为 1，才能使单进程模式有效。缺省时是多进程模式。

Apache 的主控程序是 http_main.c 中的 apache_main 方法，在其中，对有关标志进行判断，从而决定是以单进程模式还是以多进程模式运行。

若是单进程模式，则调用 worker_main，在其中创建若干线程，完成具体的请求处理，线程体是 child_sub_main 方法。单进程模式下服务器的处理过程如图 7.2 所示。

若是多进程模式，则调用 service_main，进一步调用多进程模式的核心——master_main。在 master_main 中，又会启动一子进程执行 worker_main，完成具体的操作。父进程则负责监控，并在子进程死亡时，启动一新的子进程。多进程模式下服务器的处理过程如图 7.4 所示。

如下是有关过程和方法的详细描述。

7.2.1. 主程序

整个主程序的处理过程如图 7.1 **所示：**

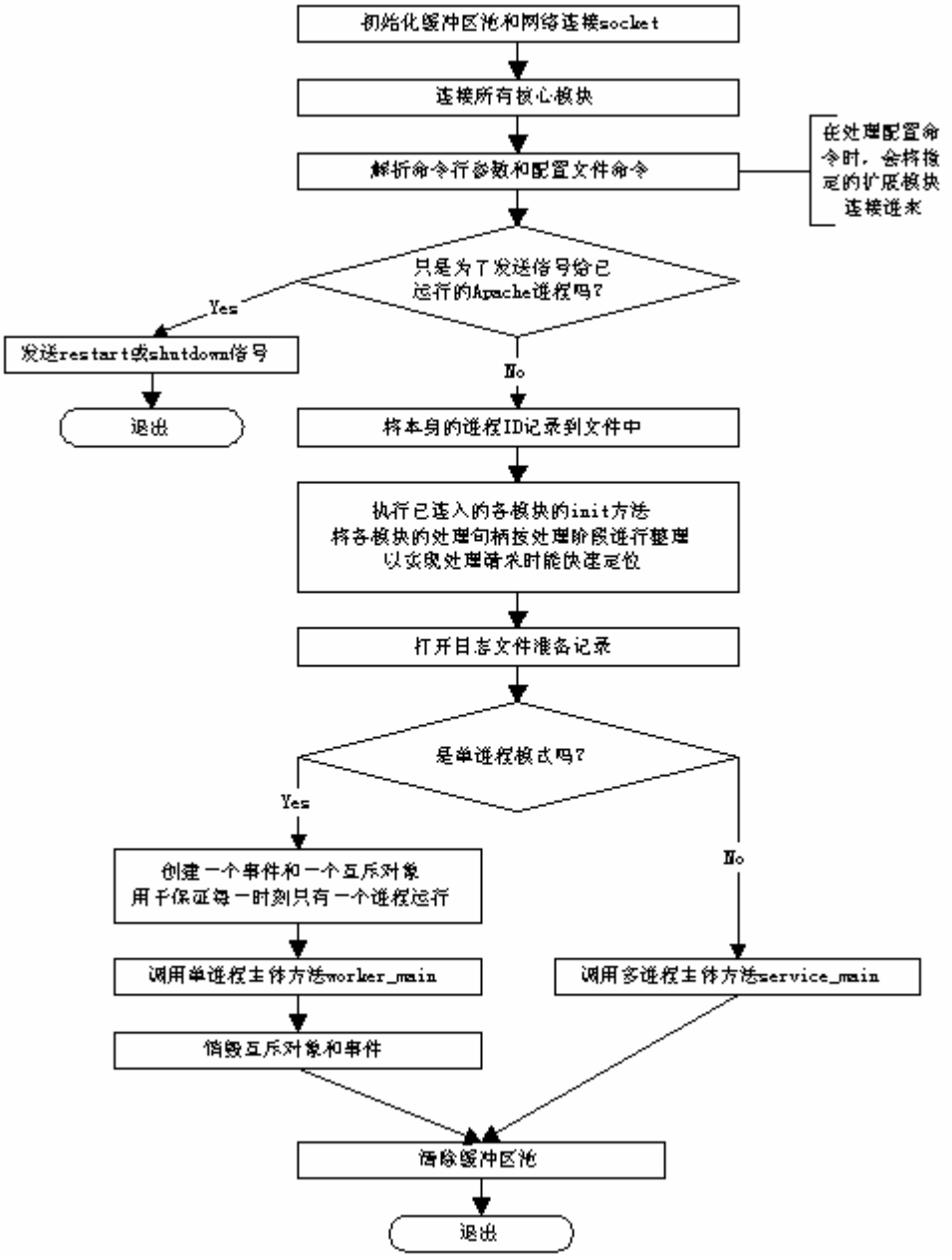


图 7.1

下面进一步结合程序语句更详细的给出每一步的处理细节：

```
common_init();
```

初始化 pool **及** socket**。**

```
ap_setup_prelinked_modules();
```

连接所有核心模块。

```
getopt(argc, argv, "D:C:c:Xd:f:vVlLZ:iusSthk:")
```

对传入的参数进行解析，以确定并执行本次任务。

```
server_conf = ap_read_config(pconf, ptrans, ap_server_confname);
```

```
    init_config_globals->ap_init_vhost_config(p);
    process_command_config(s, ap_server_pre_read_config, p, ptemp);
        ap_srm_command_loop(&parms, s->lookup_defaults);
```

处理命令行传入的读配置文件前的配置。

```
    ap_process_resource_config(s, confname, p, ptemp);
        ap_srm_command_loop(&parms, s->lookup_defaults);
            ap_handle_command(parms, config, l);
    ap_process_resource_config(s, s->srm_confname, p, ptemp);
    ap_process_resource_config(s, s->access_confname, p, ptemp);
```

读三个配置文件，逐条处理命令。

```
    process_command_config(s, ap_server_post_read_config, p, ptemp);
```

处理命令行传入的读配置文件后的配置。

```
    fixup_virtual_hosts(p, s);
```

根据主服务器的信息对虚拟主机的配置进行补充和整理

```
    default_listeners(p, s);
```

设置缺省的对地址/端口对进行监听的 listen_rec **结构，每个** listen_rec **结构对应一个要监听的地址/端口对。**

```
    ap_fini_vhost_config(p, s);
```

对关于虚拟服务器的信息进行整理和组织。

读配置文件，处理配置命令。

```
if (signal_to_send) {   send_signal(pconf, signal_to_send); exit(0);    }
```

若本次运行只为发信号，则给已运行的进程发 shutdown **或** restart **消息，然后退出。**

```
ap_log_pid(pconf, ap_pid_fname);
```

将进程 ID **记录在指定文件中，将来可接收其它进程传来的消息。**

```
ap_init_modules(pconf, server_conf);
```

执行已连入的模块的 init **方法,通过** build_method_shortcuts **将各个模块中的非空句柄按照对应阶段进行组织，将句柄放在** method_ptrs **数组中，各阶段句柄的开始地址放在** offsets_into_method_ptrs **中。调用** init_handlers **将各个模块中的非空** handler **进行组织，将** wildcard handlers **和普通的** handlers **分别放在以** wildhandlers **和** handlers **为头的链表中。**

```
ap_open_logs(server_conf, pconf);
```

打开错误日志文件作为 strerr**，包括了对主服务器和所有虚拟服务器的相应处理。**

```
if (one_process && !exit_event)  exit_event = create_event(0, 0, NULL);
if (one_process && !start_mutex)  start_mutex = ap_create_mutex(NULL);
```

若是单进程模式则创建一个事件和一个互斥对象。

```
if (child || one_process) {
    worker_main();
    ap_destroy_mutex(start_mutex);
    destroy_event(exit_event);
}else {
    service_main(master_main, argc, argv, "Apache", install,
                  run_as_service);
}
```

若是单进程模式，则调用 worker_main**，完后将有关的互斥对象和事件销毁；否则调用** service_main**。**

```
clean_parent_exit(0);
    ap_destroy_pool(pconf);
```

清除缓冲区池，并返回。

```
return 0;
```

只为了编译时不报错而设置，实际不会执行到。

7.2.2. 单进程模式——worker_main

在单进程模式下，通过互斥变量实现每一时刻只有一个进程运行。进程中又创建多个线程，其中主控制流程用来接收外部请求，并将请求加入到工作任务队列中，而其它线程（工作线程），则互斥地从工作任务队列中逐个取出请求，处理之，返回响应，并如此周而复始。本方法在 http_main.c **中定义。如图** 7.2 **所示：**

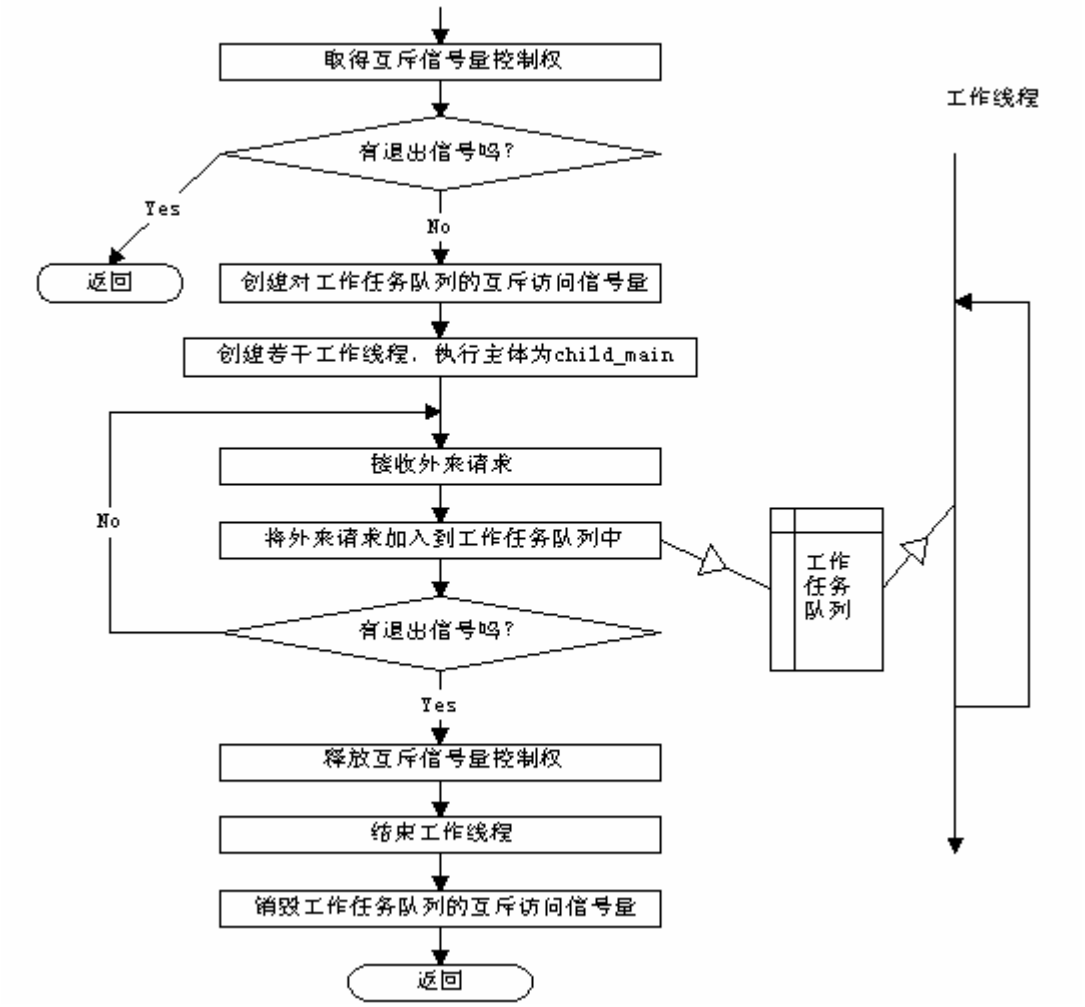


图 7.2

详细的各个步骤的语句如下所列：

```
copy_listeners(pconf);
```

对 ap_listeners **为头的链表环进行复制，存在以** old_listeners **为头的链中。**

```
reinit_scoreboard(pconf);
```

重新初始化 scoreboard**。**

```
hObjects[0] = (HANDLE)start_mutex;
hObjects[1] = (HANDLE)exit_event;
rv = WaitForMultipleObjects(2, hObjects, FALSE, INFINITE);
```

等待获取 start_mutex **与** exit_event**。**

```
if (rv == WAIT_OBJECT_0 + 1) {
/* exit event signalled - exit now */
ap_destroy_pool(pchild);
cleanup_scoreboard();
exit(0);
}
```

接收到了 exit_event**，退出。**

否则获得了 start_mutex **控制权，开始工作。**

```
setup_listeners(pconf);
    make_sock(p, &lr->local_addr);
        socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)
        bind(s, (struct sockaddr *) server, sizeof(struct sockaddr_in))
        listen(s, ap_listenbacklog)
        setsockopt();
```

创建以 ap_listeners **为头的链表中的** socket **进行监听，并把该链表转换成单向循环链表。**

将 old_listeners **为头的链表中未用的** socket **关闭，并释放该链表。**

```
ap_child_init_modules(pconf, server_conf);
```

对各个模块的 child_init **句柄进行调用，完成初始化。**

```
allowed_globals.jobsemaphore = create_semaphore(0);
allowed_globals.jobmutex = ap_create_mutex(NULL);
```

创建一个信号量和一个互斥对象，用于对处理请求（job**：工作）的队列进行互斥访问。**

```
/* spawn off the threads */
child_handles = (thread *) alloca(nthreads * sizeof(int));
for (i = 0; i < nthreads; i++) {
child_handles[i] = create_thread((void (*)(void *)) child_main, (void *) i);
}
if (nthreads > max_daemons_limit) {
max_daemons_limit = nthreads;
}
```

创建指定数目的线程。每个线程均是去运行 child_main **，实际上是去执行** child_sub_main**。**

```
while (1) {.........
```

开始循环，接收请求，并处理之。

```
rv = wait_for_many_objects(nthreads, child_handles, 0);
```

等待有线程退出。

```
    if (rv != WAIT_TIMEOUT)  {.. cleanup_thread(child_handles, &nthreads,
rv);..break;.}
```

有线程退出，则将该线程清除。

```
else{..........
srv = ap_select(listenmaxfd + 1, &main_fds, NULL, NULL, &tv);
```

等待有 socket **端口可读。**

```
lr = find_ready_listener(&main_fds);
if (lr != NULL) {
sd = lr->fd;
```

从 socket **组中找到** select **的** socket**。**

```
do {
csd = accept(sd, (struct sockaddr *) &sa_client, &clen);
} while (csd < 0 && errno == EINTR);
```

接收远程客户的连接。

```
add_job(csd);
total_jobs++;
}
```

将上面接收的客户的连接请求放到 allowed_globals.jobhead **为头的链表中。**

结束 while **循环。**

```
if (!start_mutex_released) {
ap_release_mutex(start_mutex);
}
```

释放 start_mutex **互斥对象。**

```
for (i = 0; i < nthreads; i++) {
kill_thread(child_handles[i]);
free_thread(child_handles[i]);
}
```

结束各线程。

```
destroy_semaphore(allowed_globals.jobsemaphore);
ap_destroy_mutex(allowed_globals.jobmutex);
```

释放有关的互斥对象和信号量。

```
ap_child_exit_modules(pconf, server_conf);
```

调用各模块的 child_exit **句柄。**

```
cleanup_scoreboard();
```

清除 scoreboard**。**

```
clean_parent_exit(0);
```

清除服务器的占用的资源，并退出。

7.2.3. 单进程模式的核心（线程体）——child_sub_main

本方法在 http_main.c **中定义。**

流程如图 7.3 **所示：**

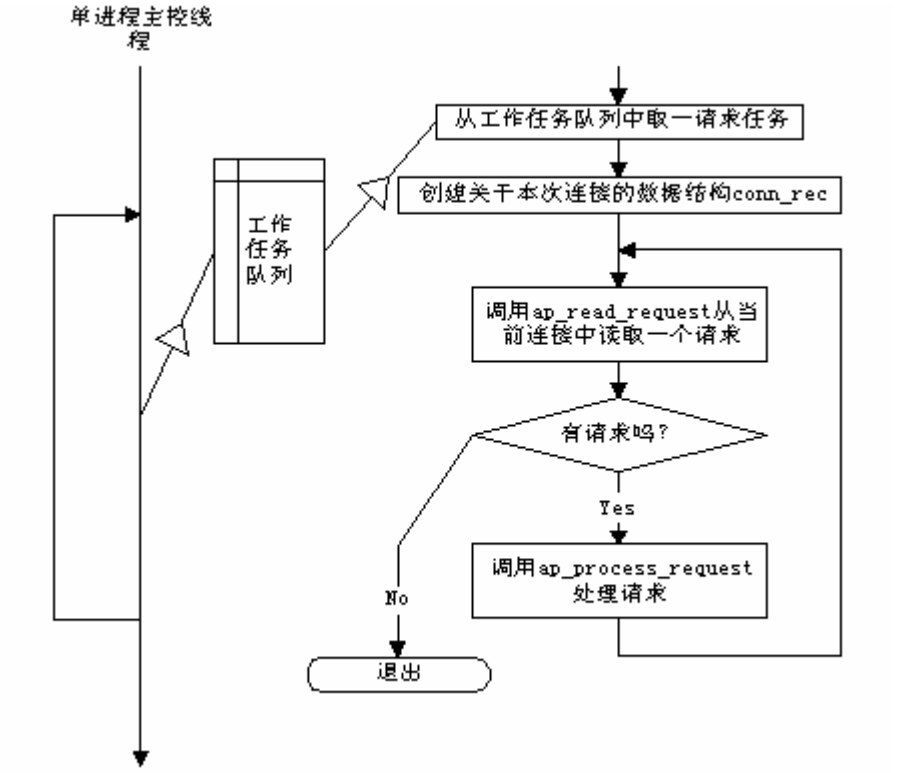


图 7.3

```
ptrans = ap_make_sub_pool(pconf);
```

申请缓冲区。

```
(void) ap_update_child_status(child_num, SERVER_READY, (request_rec *)
NULL);
```

对本 child **线程的状态进行更新，记录在相应的** scoreboard **中。**

```
setjmp(jmpbuffer);
```

保存当前状态在 jumpbuffer **中。**

```
while (1) {....
```

进入循环，每次取一 job**，完成对应请求的处理。**

```
(void) ap_update_child_status(child_num, SERVER_READY,
                              (request_rec *) NULL);
```

更新状态为 SERVER_READY**。**

```
csd = remove_job();
```

取得一个 job**，若没有则一直阻塞，直到有请求** job **为止。**

```
ap_note_cleanups_for_socket(ptrans, csd);
```

对接收客户请求从而又新分配的端口号进行记录，待以后清除。

```
if (getsockname(csd, &sa_server, &clen) < 0) {...}
```

取得 socket **端口** csd **对应的本地地址。**

```
if ((getpeername(csd, &sa_client, &clen)) < 0) {...}
```

取得 socket **端口** csd **对应的客户端的地址。**

```
(void) ap_update_child_status(child_num, SERVER_BUSY_READ,
          (request_rec *) NULL);
```

更新状态为 SERVER_ BUSY_READ**。**

```
conn_io = ap_bcreate(ptrans, B_RDWR | B_SOCKET);
```

创建一个缓冲区，完成网络连接的 I/O**。**

```
ap_bpushfd(conn_io, csd, dupped_csd);
```

将缓冲区 conn_io **与** socket **端口** csd **及其副本** dupped_csd **建立联系。**

```
current_conn = new_connection(ptrans, server_conf, conn_io,
                    (struct sockaddr_in *) &sa_client,
                    (struct sockaddr_in *) &sa_server,
                   child_num);
```

创建一个 conn_rec **类型的结构，记录本次连接的信息。包含了一些非常基础和重要的域的初始化。**

```
while ((r = ap_read_request(current_conn)) != NULL) {
    ap_process_request(r);
}
```

循环用 ap_read_request **读取请求，放到** r **中，用** ap_process_request **处理请求。直到连接关闭。**

```
ap_kill_cleanups_for_socket(ptrans, csd);
```

清除本次连接占用的 socket **信息。**

```
}
```

while(1)**无限循环结束处，循环取任务。**

```
ap_destroy_pool(ptrans);
```

清除本线程占用的资源。

7.2.4. 多进程模式前奏

在多进程模式下，service_main **方法是处理的主体，完成与** NT **下服务相关的操作。**

service_main

service_main **方法的处理框架如图** 7.4 **所示：**

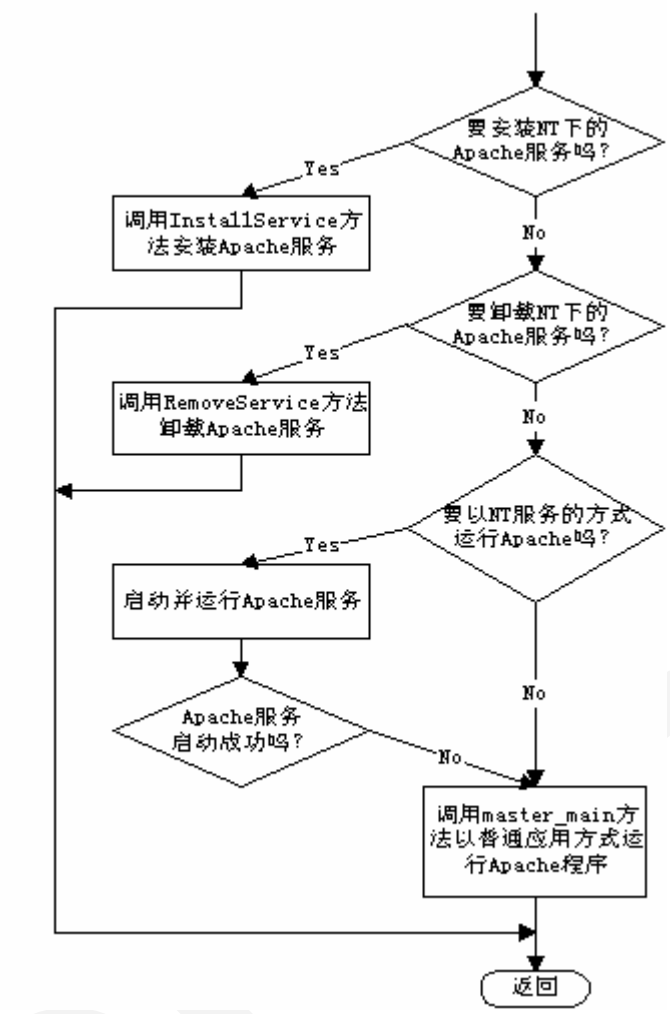


图 7.4

详细的处理过程及相关代码如下：

```
if(install_flag > 0)
{
    InstallService();
    return(0);
}
```

完成要求的服务的安装。

```
else if(install_flag < 0)
{
    RemoveService();
    return(0);
}
```

完成要求的服务的删除。

```
if(run_as_service){
     if(!StartServiceCtrlDispatcher(dispatchTable))
     {
         return((*main_fn)(argc, argv));
```

```
    }
    else
    {
        return(globdat.exit_status);
    }
```

若是以服务方法运行，则调用 StartServiceCtrlDispatcher**，实际是执行** service_main_fn **方法，再进一步地执行** master_main **方法，在退出时，返回记录在** globdat.exit_status **中状态值。若服务启动出错（**StartServiceCtrlDispatcher **方法返回** 0**），则直接执行** master_main **方法。**main_fn **为调用** service_main **方法时传入的参数，为指向** master_main **方法的指针。**

```
else
{
    globdat.connected = 0;
    return((*main_fn)(argc, argv));
}
```

不以服务方式运行，直接调用 master_main **方法。**

结束。

service_main_fn

在 service_main **方法中调用** StartServiceCtrlDispatcher**，实际上是去间接调用已在派送表中定义的** service_main_fn **方法。**service_main_fn **方法按照** NT **下服务的执行方式启动** Apache **服务。而且服务仅是一个外壳，处理核心则是外部传入的指定方法——**master_main**。**

```
if(!(globdat.hServiceStatus = RegisterServiceCtrlHandler( globdat.name,
service_ctrl)))
{
    globdat.exit_status = -1;
    return;
}
```

调用 RegisterServiceCtrlHandler **来设置接收对本服务的控制信息的句柄为** service_ctrl**。失败时记录并返回。**

```
ReportStatusToSCMgr(
    SERVICE_START_PENDING, // service state
    NO_ERROR,            // exit code
    3000);                 // wait hint
```

报告当前状态。

```
globdat.exit_status = (*globdat.main_fn)( argc, argv );
```

调用 master_main **进入核心部分。**

```
ReportStatusToSCMgr(SERVICE_STOPPED, NO_ERROR, 0);
```

报告当前状态。

```
return;
```

返回。

7.2.5. 多进程模式的核心——master_main

master_main 方法是多进程模式下的处理核心。由于此部分处理复杂，所以下面分为两个流程图对处理过程进行描述，包括主要框架和外部大循环体。

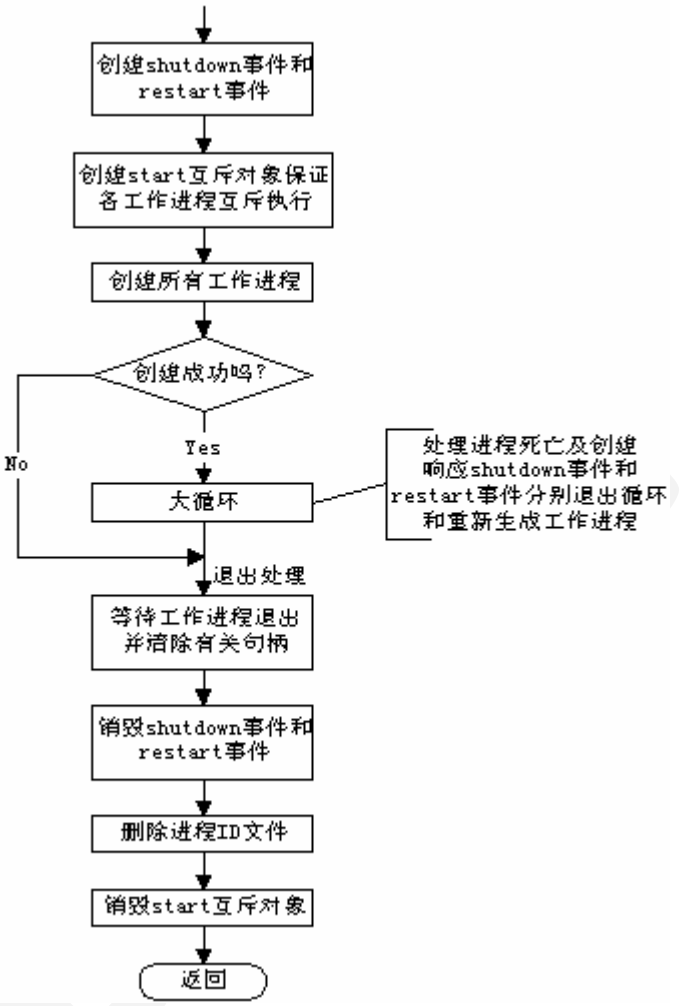


图 7.5

```
nchild = 1;     /* only allowed one child process for current generation */
processes_to_create = nchild;
```

指示只创建一个子进程。

```
signal_shutdown_event = CreateEvent(NULL, TRUE, FALSE,
                        signal_shutdown_name);
```

创建具有公共所知名字的 shutdown 事件。

```
signal_restart_event = CreateEvent(NULL, TRUE, FALSE, signal_restart_name);
```

创建具有公共所知名字的 restart 事件。

```
start_mutex = ap_create_mutex(signal_prefix_string);
```

创建具有公共所知名字的 start 互斥对象，用来保证各个子进程互斥地执行。

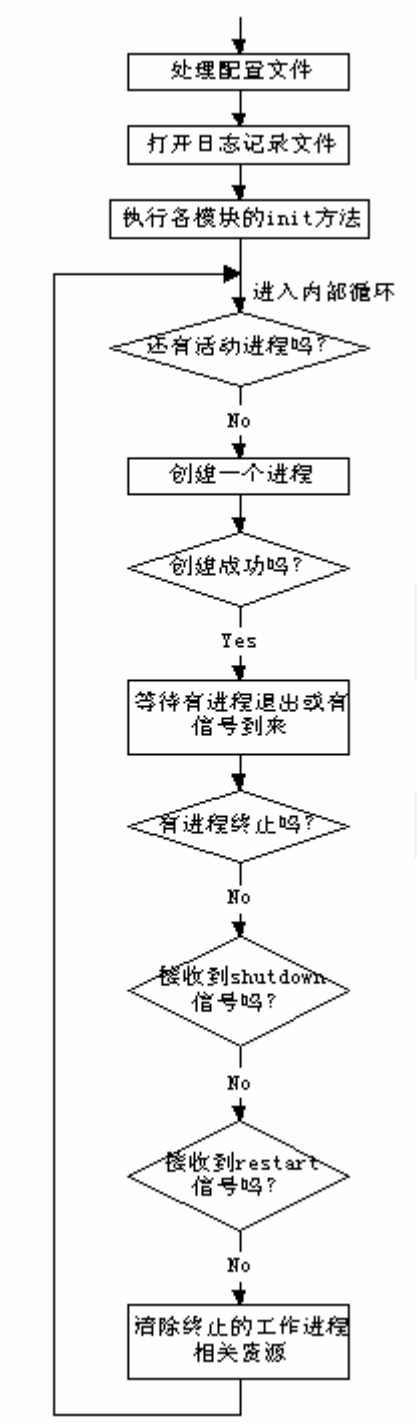


图 7.6

```
while (processes_to_create--) {
    service_set_status(SERVICE_START_PENDING);
    if (create_process(process_handles, process_kill_events,
        &current_live_processes, &child_num, signal_prefix_string, argc,
                                                        argv) < 0) {
            goto die_now;
    }
```

```
}
```

循环调用 create_process**，并实际间接地进一步调用** create_event_and_spawn **来完成子进程的创建，同时按一定规则为每个子进程构造一名字来创建一退出事件。子进程的句柄记录在** process_handles **中，而各退出事件记录在** process_kill_events **中。一旦创建进程操作失败，则退出，去执行退出操作。**

每个子进程通过传入的-Z **参数也会获得相应的退出事件名，从而可对该事件进行操作。通过该事件，父进程可以通知子进程退出。**

通过-Z **参数，子进程将其** child **变量设为** 1**，从而去执行** worker_main **方法，即与单进程模式相同，在其中又会创建一系列的线程来完成具体的操作。**

```
do { /* restart-pending */
```

开始一外部大循环，直至 restart_pending **为** 0**，即要退出，而非重启。**

```
server_conf = ap_read_config(pconf, pparent, ap_server_confname);
```

读配置文件，处理配置命令。

```
ap_open_logs(server_conf, pconf);
```

打开错误日志文件作为 strerr**，包括了对主服务器和所有虚拟服务器的相应处理，是主程序中的重复。**

```
ap_set_version();
```

设置版本信息。

```
ap_init_modules(pconf, server_conf);
```

是主程序中的重复，执行已连入的模块的 init **方法，通过** build_method_shortcuts **将各个模块中的非空句柄按照对应阶段进行组织，将句柄放在** method_ptrs **数组中，各阶段句柄的开始地址放在** offsets_into_method_ptrs **中。调用** init_handlers **将各个模块中的非空** handler **进行组织，将** wildcard handlers **和普通的** handlers **分别放在以** wildhandlers **和** handlers **为头的链表中。**

```
while (1) {
```

进入内部循环，直到有子进程死去，或被服务管理器或 RPC **服务器通知停止。**

```
if (current_live_processes == 0) {
/* Shouldn't happen, but better safe than sorry */
ap_log_error(APLOG_MARK,APLOG_ERR|APLOG_NOERRNO, server_conf,
    "master_main: no child processes alive! creating one");
if (create_process(process_handles, process_kill_events,
    &current_live_processes, &child_num, signal_prefix_string,
    argc, argv) < 0) {
    goto die_now;
}
if (processes_to_create) {
    processes_to_create--;
}
}
```

若当前已没有活动的子进程，则创建一个，以便能处理请求。

```
process_handles[current_live_processes] = signal_shutdown_event;
process_handles[current_live_processes+1] = signal_restart_event;
```

将 shutdown **事件和** restart **事件记录在** process_handles **数组中，为后续使用。**

```
rv = WaitForMultipleObjects(current_live_processes+2,
```

```
                                    (HANDLE *)process_handles, FALSE, INFINITE);
```

对各个子进程和上述的两个事件进行无限等待,直到有任何一个进程终止或任何一个事件为信号态。

```
    if (rv == WAIT_FAILED) {
        /* Something serious is wrong */
        ap_log_error(APLOG_MARK,APLOG_CRIT|APLOG_WIN32ERROR, server_conf,
        "WaitForMultipeObjects on process handles and apache-signal -- doing
shutdown");
        shutdown_pending = 1;
        break;
    }
```

若等待中出错，则退出内部循环，并通过设置 shutdown_pending **为** 1**，指示要退出执行。**

```
    if (rv == WAIT_TIMEOUT) {
    /* Hey, this cannot happen */
    ap_log_error(APLOG_MARK, APLOG_ERR, server_conf,
       "WaitForMultipeObjects with INFINITE wait exited with WAIT_TIMEOUT");
    shutdown_pending = 1;
    }
```

实际上此部分不应该得到执行，因为永远不会超时。

```
    cld = rv - WAIT_OBJECT_0;
    if (cld == current_live_processes) {
        /* shutdown event signalled, we should exit now */
        if (ResetEvent(signal_shutdown_event) == 0) {
          ap_log_error(APLOG_MARK, APLOG_ERR|APLOG_WIN32ERROR, server_conf,
            "ResetEvent(signal_shutdown_event)");
        }
        shutdown_pending = 1;
        break;
    }
```

若接收到 shutdown **信号，则将其设置成非信号态，并退出内部循环，且通过设置** shutdown_pending **为** 1**，指示要退出执行。**

```
    if (cld == current_live_processes+1) {
        /* restart event signalled, we should exit now */
        if (ResetEvent(signal_restart_event) == 0) {
          ap_log_error(APLOG_MARK, APLOG_ERR|APLOG_WIN32ERROR, server_conf,
            "ResetEvent(signal_restart_event)");
        }
        restart_pending = 1;
        break;
    }
```

若接收到 restart **信号，则将其设置成非信号态，并退出内部循环，且通过设置** restart_pending **为** 1**，指示要重新启动执行。**

```
    cleanup_process(process_handles, process_kill_events, cld,
                            &current_live_processes);
```

清除掉终止的子进程的无效句柄及相应事件，并整理句柄数组。

```
}
```

内部循环的结束处。

```
if (!shutdown_pending && !restart_pending) {
    ap_log_error(APLOG_MARK,APLOG_CRIT|APLOG_NOERRNO, server_conf,
     "master_main: no shutdown or restart variables set -- doing shutdown");
    shutdown_pending = 1;
}
```

若既无 shutdown **标志也无** restart **标志，则记录此不正常状态，并显式地设置** shutdown **标志。**

```
if (shutdown_pending) {
    /* tell all child processes to die */
    for (i = 0; i < current_live_processes; i++) {
    if (SetEvent(process_kill_events[i]) == 0)
        ap_log_error(APLOG_MARK,APLOG_WIN32ERROR, server_conf,
         "SetEvent for child process in slot #%d", i);
    }
    break;
}
```

若要 shutdown**，则通过设各子进程的退出事件为信号态通知它们退出，并退出外部** do..while **循环。**

```
if (restart_pending) {
    int children_to_kill = current_live_processes;

    processes_to_create = nchild;
    for (i = 0; i < nchild; ++i) {
    if (current_live_processes >= MAX_PROCESSES)
        break;
    create_process(process_handles, process_kill_events,
                   &current_live_processes,
        &child_num, signal_prefix_string, argc, argv);
    processes_to_create--;
    }
    for (i = 0; i < children_to_kill; i++) {
    if (SetEvent(process_kill_events[i]) == 0)
        ap_log_error(APLOG_MARK,APLOG_WIN32ERROR, server_conf,
         "SetEvent for child process in slot #%d", i);
    }
}
```

若要 restart**，则重新创建原始数目的子进程，并通知原来的各子进程退出。**

```
++ap_my_generation;
```

记录当前是第几代，即重启过几次。

```
} while (restart_pending);
```

外部循环结束处。

```
die_now:
```
开始退出的有关处理。
```
tmstart = time(NULL);
while (current_live_processes && ((tmstart+60) > time(NULL))) {
    rv = WaitForMultipleObjects(current_live_processes,
            (HANDLE *)process_handles, FALSE, 2000);
    cld = rv - WAIT_OBJECT_0;
    ap_assert(rv < current_live_processes);
    cleanup_process(process_handles, process_kill_events, cld,
            &current_live_processes);
}
```
在 60 秒内等待所有子进程退出，并清除有关句柄。
```
for (i = 0; i < current_live_processes; i++) {
ap_log_error(APLOG_MARK,APLOG_ERR|APLOG_NOERRNO, server_conf,
   "forcing termination of child #%d (handle %d)", i, process_handles[i]);
TerminateProcess((HANDLE) process_handles[i], 1);
}
```
若仍有子进程未退出，则强制其及其线程退出。
```
CloseHandle(signal_restart_event);
CloseHandle(signal_shutdown_event);
```
关闭两事件句柄。
```
{
const char *pidfile = NULL;
pidfile = ap_server_root_relative (pparent, ap_pid_fname);
if ( pidfile != NULL && unlink(pidfile) == 0)
   ap_log_error(APLOG_MARK, APLOG_NOERRNO|APLOG_INFO,
         server_conf,
         "removed PID file %s (pid=%ld)",
         pidfile, (long)getpid());
}
```
删除用于记录进程 ID 的临时文件。
```
ap_destroy_mutex(start_mutex);
```
销毁互斥控制对象 start_mutex。
```
return(0);
```
退出执行。

第3节　重要的数据结构

本部分集中介绍 Apache 中比较重要的数据结构及相关的处理方法。

7.3.1. 资源池结构

Apache 中使用如下定义的资源池结构来管理资源的分配和去配。

```
struct pool {
    union block_hdr *first;
    union block_hdr *last;
    struct cleanup *cleanups;
    struct process_chain *subprocesses;
    struct pool *sub_pools;
    struct pool *sub_next;
    struct pool *sub_prev;
    struct pool *parent;
    char *free_first_avail;
};
```

pool 结构有父和子之分，parent 指向父池，sub_pools 指向子池的链表的头，sub_next 指向后一个兄弟子池，sub_prev 指向前一个子池，即该链表组织成循环结构。

管理的内存资源是以块的形式存在的。block_hdr 联合结构作为内存块的头，存放该块的信息。

```
union block_hdr {
    union align a;
    struct {
        char *endp;
        union block_hdr *next;
        char *first_avail;
    } h;
};
```

其中 h.next 可以指向下一个块的头，h.first_avail 指向当前空闲块的起始位置，h.endp 指向终止位置。

pool 结构管理的内存块组织成链表形式,其中的 first 和 last 分别指向链表的第一块和最后一块，free_first_avail 则指向第一块的空闲空间起始位置。

Apache 中定义了如下的 permanent_pool 作为所有资源池的父池。

```
static pool *permanent_pool;
```

相关的方法如下。

ap_init_alloc 方法

```
pool *ap_init_alloc(void)
```

本方法完成系统的最初的资源池初始化工作。

调用 ap_create_mutex 方法分别创建 alloc_mutex 和 spawn_mutex 互斥信号量。

调用 ap_make_sub_pool 方法创建系统资源池 permanent_pool，返回其指针。

malloc_block 方法

```
static union block_hdr *malloc_block(int size)
```

从系统中申请一个指定大小的内存块，并创建一个 block_hdr 联合结构作为块头指向该空闲块，返回 block_hdr 的指针。

实际上是调用 malloc 方法申请了一个大小为 block_hdr 联合结构和指定大小之和的一个内存块，其中开始部分作为块头——block_hdr 联合结构。

若申请失败，则直接调用 exit，退出整个程序。

new_block 方法

```
static union block_hdr *new_block(int min_size)
```

申请一个指定大小的内存块，返回其块头的指针。

首先查看空闲块链表 block_freelist 中是否有大小满足要求的空闲块。若有，则直接取出该块返回之；否则调用 malloc_block 方法重新申请一个内存块，返回之。

ap_make_sub_pool 方法

```
API_EXPORT(struct pool *) ap_make_sub_pool(struct pool *p)
```

本方法创建一个新的资源池作为父池 p 的子池，返回其指针。

方法体通过申请和释放互斥信号量 alloc_mutex，实现互斥执行。

首先调用 new_block 方法申请一个空闲内存块，该块的开始大小为 sizeof(struct pool)的部分被用做 pool 结构，而该块则成为该资源池管理的第一个块，现在也是唯一一个，即资源池本身占用的空间是从自己管理的内存块中申请的。

最后若父池 p 非空，则将新池链到 p 的子池链表中。

ap_palloc 方法

```
API_EXPORT(void *) ap_palloc(struct pool *a, int reqsize)
```

从资源池 a 中申请大小为 reqsize 的空间，返回其地址指针。

首先查看 a 中最后一个空闲块（最新分配的一个块）大小是否满足需要，是则从中分配指定大小空间，返回其地址；否则调用 new_block 方法创建一个新的内存块加入到资源池中，再从中分配要求的空间。

ap_pcalloc 方法

```
API_EXPORT(void *) ap_pcalloc(struct pool *a, int size)
```

从资源池 a 中申请一块大小为 size 的空间，并将其单元初始化为 0，返回该空间块的指针。具体调用 ap_palloc 方法实现。

ap_pstrdup 方法

```
API_EXPORT(char *) ap_pstrdup(struct pool *a, const char *s)
```

从资源池 a 中分配空间复制字符串 s，返回复制串的指针。具体调用 ap_palloc 方法实现空间申请。

ap_pstrndup 方法

```
API_EXPORT(char *) ap_pstrndup(struct pool *a, const char *s, int n)
```

从资源池 a 中分配空间复制字符串 s 的头 n 个字符，返回复制串的指针。具体调用 ap_palloc 方法实现空间申请。

ap_pstrcat 方法

```
API_EXPORT_NONSTD(char *) ap_pstrcat(pool *a,...)
```

从资源池 a 中分配空间存放可变数目的多个字符串的连接，返回复制串的指针。具体调用 ap_palloc 方法实现空间申请。

ap_pvsprintf 方法

```
API_EXPORT(char *) ap_pvsprintf(pool *p, const char *fmt, va_list ap)
```

本方法用来从资源池 p 中申请空间，将原始可变参数列表 ap 的内容以指定格式 fmt 输出到字符串中，并返回结果。

ap_psprintf 方法

```
API_EXPORT_NONSTD(char *) ap_psprintf(pool *p, const char *fmt, ...)
```

本方法与 ap_pvsprintf 方法类似，只是调用的参数形式不同，实际上内部实现上正是调用后者实现的。

7.3.2. 数组结构

Apache 中很多地方都使用数组结构来存放信息，如 HTTP 请求的一系列头域。在 alloc.c 文件中，定义如下的数组头结构来存放数组的信息。

```
typedef struct {
    ap_pool *pool;
    int elt_size;
    int nelts;
    int nalloc;
    char *elts;
} array_header;
```

其中 pool 为数组空间来自的资源池，nalloc 为分配的数组单元个数，elt_size 为单元大小，elts 指向各个单元，nelts 为已占用的单元个数，初始值为 0。

这样实际上一个数组头结构就代表一个数组。在下文中我们有时也会用一个头的名字指代一个数组。

make_array_core 方法

```
make_array_core(array_header *res, pool *p, int nelts, int elt_size)
```

分配 nelts **个大小为** elt_size **的空闲单元构成一个数组,地址等信息存在** res **的各个域中，同时还记录该数组所在的资源池** p **的引用。**

ap_push_array 方法

```
API_EXPORT(void *) ap_push_array(array_header *arr)
```

本方法实现从数组 arr **取得一个空闲单元，返回该单元的指针。**

如果当前 array **已满**(arr->nelts == arr->nalloc)**，则申请一块更大的空间，将原来的数据拷贝到新数组中，然后再划出一个单元。**

table_push 方法

```
static table_entry *table_push(table *t)
```

同 ap_push_array **方法，从指定的数组中申请一个单元，即从预申请的空闲单元中拿出一个，并标为占用，并将该单元的引用返回。**

ap_array_cat 方法

```
API_EXPORT(void) ap_array_cat(array_header *dst, const array_header *src)
```

将数组 src **附加到数组** dst **中。**

若数组 dst **中空间不够，则重新从** dst **对应的资源池中申请足够空间来扩充数组** dst**。**

ap_make_array 方法

```
ap_make_array(pool *p, int nelts, int elt_size)
```

在资源池 p **中申请一个** nelts **个大小为** elt_size **的空闲单元的数组，并生成一个数组头，返回其引用。调用** make_array_core **方法申请指定数组。**

ap_copy_array_hdr 方法

```
API_EXPORT(array_header *) ap_copy_array_hdr(pool *p, const array_header
*arr)
```

创建一个新的数组头，并使之指向与 arr **所指同一数组。调用了** copy_array_hdr_core **方法。**

copy_array_hdr_core 方法

```
static ap_inline void copy_array_hdr_core(array_header *res, const array_header *arr)
```

将数组头 arr **中记录的信息复制到** res **中，从而使两者指向同一数组。**

ap_append_arrays 方法

```
API_EXPORT(array_header *) ap_append_arrays(pool *p, const array_header *first, const array_header *second)
```

将数组 second **和数组** first **合并，返回结果数组。**

与 ap_array_cat **方法不同，此处得到的结果数组不是原来数组中的任何一个。**

ap_array_pstrcat 方法

```
API_EXPORT(char *) ap_array_pstrcat(pool *p, const array_header *arr, const char sep)
```

将数组 arr **的各个单元（字符串）连接起来。若** sep **非空，则作为分隔符。结果串所用空间在资源池** p **中申请，返回结果串。**

7.3.3. 表结构

Apache **在数组结构基础上又定义了** table **结构：**

```
struct table {
    array_header a;
};
```

从中可以看出，table **与数组结构实际是完全相同的。**

进一步地，又定义了 table_entry **结构：**

```
typedef struct {
    char *key;
    char *val;
} table_entry;
```

在以 table **方式访问数组时，每个数组单元均以** table_entry **结构类型对待。这样** table **实际上存放一系列的键值对（指针）。**

有时还需要更复杂的单元结构，因此也定义了 overlap_key **结构：**

```
typedef struct {
    char *key;
    char *val;
    int order;
} overlap_key;
```

每一项在 table_entry **的基础上又增加了** order **域。**

相关的方法如下。

ap_make_table 方法

```
API_EXPORT(table *) ap_make_table(pool *p, int nelts)
```

创建一个 nelts 个单元的表，返回之。

ap_copy_table 方法

```
API_EXPORT(table *) ap_copy_table(pool *p, const table *t)
```

复制表 t，返回其指针。

需要注意的是，由于表中直接存放的是键和值的指针，所以复制的仅是指针，而非实际内容。

ap_clear_table 方法

```
API_EXPORT(void) ap_clear_table(table *t)
```

清空表 t，只需将 t->a.nelts 设为 0。

ap_table_get 方法

```
API_EXPORT(const char *) ap_table_get(const table *t, const char *key)
```

检查表 t 中是否有键 key 对应的值。有则返回该值的指针，否则返回 NULL。

ap_table_set 方法

```
API_EXPORT(void) ap_table_set(table *t, const char *key, const char *val)
```

将表 t 中键为 key 的值设为 val。若有多个项匹配，则删除重复项。若没有匹配的项，则增加一个对应键值对。

其中设置的键和值均为传入参数值的拷贝，此为本方法与 ap_table_setn 方法之间的唯一区别。

ap_table_setn 方法

```
API_EXPORT(void) ap_table_setn(table *t, const char *key, const char *val)
```

将表 t 中键为 key 的值设为 val。若有多个项匹配，则删除重复项。若没有匹配的项，则增加一个对应键值对。

其中设置的键和值均直接为传入参数值，此为本方法与 ap_table_set 方法之间的唯一区别。

ap_table_unset 方法

```
API_EXPORT(void) ap_table_unset(table *t, const char *key)
```

将表 t **中键为** key **的项删除。**

ap_table_merge 方法

```
API_EXPORT(void) ap_table_merge(table *t, const char *key, const char *val)
```

将传入的键值对合并到表 t **中。**

若表 t **中已有键** key**，则将值** val **附加到原有的值（串）后，并用“，”分隔；否则，在表中增加该键值对。**

ap_table_mergen 方法

```
API_EXPORT(void) ap_table_mergen(table *t, const char *key, const char *val)
```

将传入的键值对合并到表 t **中。**

若表 t **中已有键** key**，则将值** val **附加到原有的值（串）后，并用“，”分隔；否则，在表中增加该键值对。与** ap_table_merge **方法不同的是不进行复制，而是直接将键和值加到表** t **中。**

ap_table_add 方法

```
API_EXPORT(void) ap_table_add(table *t, const char *key, const char *val)
```

将 key **及** val **复制并加到** t **中。**

ap_table_addn 方法

```
API_EXPORT(void) ap_table_addn(table *t, const char *key, const char *val)
```

与 ap_table_add **唯一不同的是不对键和值进行复制，而是直接加到表** t **中。**

ap_overlay_tables 方法

```
API_EXPORT(table *) ap_overlay_tables(pool *p, const table *overlay, const
table *base)
```

将表 a **和表** b **进行合并，而不管各自的键值对是否重复，返回结果表。**

ap_table_do 方法

```
API_EXPORT(void) ap_table_do(int (*comp) (void *, const char *, const char *), void *rec, const table *t,...)
```

功能抽象。对表 t **中的每一个与传入的可变参数相同的键值对均用指定的方法进行计算。直到计算结果为** 1 **或表遍历完。**

sort_overlap 方法

```
static int sort_overlap(const void *va, const void *vb)
```

本方法实现对 va **和** vb **表示的两个** overlap_key **结构进行比较，返回比较结果。**

若两者的 key **相同，则返回** 1**；否则返回两者的** order **值的差。**

ap_overlap_tables 方法

```
API_EXPORT(void) ap_overlap_tables(table *a, const table *b, unsigned flags)
```

将表 a **和表** b **进行合并，进行快速排序，键相同项的对应值合并，结果表由** a **指向。**

7.3.4. cleanups

pool **结构中定义了** cleanups **域，为指向如下** cleanup **结构的指针类型。**

```
struct cleanup {
    void *data;
    void (*plain_cleanup) (void *);
    void (*child_cleanup) (void *);
    struct cleanup *next;
};
```

Apache **中使用** cleanup **结构登记除空间外的其他资源的占用，如句柄、套接字等；其中的** plain_cleanup **和** child_cleanup **是两个函数指针，可用它们释放对应资源。**

各个 cleanup **结构组成单链表，表头指针则存于** pool **结构的** cleanups **域。**

这样用 cleanup **结构专门对资源的占用进行跟踪，可有效地消除分散在程序各处的被占用资源忘记释放的问题。**

如下为几组相关方法：

- **通用** cleanup **方法：**

```
ap_register_cleanup
ap_kill_cleanup
ap_run_cleanup
run_cleanups
run_child_cleanups
```

- **基于通用方法针对文件句柄资源的处理方法：**

```
fd_cleanup
```

```
ap_note_cleanups_for_fd
ap_kill_cleanups_for_fd
ap_popenf
ap_pclosef
```

- **针对普通句柄（HANDLE）资源的处理方法：**

```
h_cleanup
ap_note_cleanups_for_h
ap_pcloseh
```

- **针对套接字资源的处理方法：**

```
socket_cleanup
ap_note_cleanups_for_socket
ap_kill_cleanups_for_socket
ap_psocket
ap_pclosesocket
```

ap_register_cleanup 方法

```
API_EXPORT(void) ap_register_cleanup(pool *p, void *data, void
(*plain_cleanup) (void *), void (*child_cleanup) (void *))
```

创建 cleanup **结构记录相关信息，并链入到指定资源池的** cleanups **链表中。**

ap_kill_cleanup 方法

```
API_EXPORT(void) ap_kill_cleanup(pool *p, void *data, void (*cleanup) (void
*))
```

本方法将在资源池 p **的** cleanups **链表中与指定** data **和** cleanup **值匹配的** cleanup **项去掉。**

ap_run_cleanup 方法

```
API_EXPORT(void) ap_run_cleanup(pool *p, void *data, void (*cleanup) (void
*))
```

针对 data **执行指定的清除方法** cleanup**，并调用** ap_kill_cleanup **方法将** p-> cleanups **中记录的对应项删除。**

run_cleanups 方法

```
static void run_cleanups(struct cleanup *c)
```

完成 cleanup **链表** c **中记录的资源清除工作,对每一项针对** data **执行** plain_cleanup **方法。**

run_child_cleanups 方法

```
static void run_child_cleanups(struct cleanup *c)
```

与 run_cleanups 方法类似，但执行的是 cleanup 结构中的 child_cleanup 结构。

fd_cleanup 方法

```
static void fd_cleanup(void *fdv)
```

完成指定文件句柄资源的清除工作，具体是关闭该句柄。

ap_note_cleanups_for_fd 方法

```
API_EXPORT(void) ap_note_cleanups_for_fd(pool *p, int fd)
```

调用 ap_register_cleanup 方法，登记对文件句柄 fd 的清除，清除方法为 fd_cleanup。

ap_kill_cleanups_for_fd 方法

```
API_EXPORT(void) ap_kill_cleanups_for_fd(pool *p, int fd)
```

将在资源池 p 的 cleanup 链表中与指定文件句柄匹配的 cleanup 结构去除。

ap_popenf 方法

```
API_EXPORT(int) ap_popenf(pool *a, const char *name, int flg, int mode)
```

本方法将文件打开工作与记录对应文件句柄的清除信息工作封装在一起。name 为要打开的文件名。调用 ap_note_cleanups_for_fd 方法记录对文件句柄的占用。

ap_pclosef 方法

```
API_EXPORT(int) ap_pclosef(pool *a, int fd)
```

将文件句柄 fd 对应文件关闭，并去除资源池 a 中记录的 cleanup 信息。

h_cleanup 方法

```
static void h_cleanup(void *fdv)
```

调用 CloseHandle 方法关闭指定句柄。

ap_note_cleanups_for_h 方法

```
API_EXPORT(void) ap_note_cleanups_for_h(pool *p, HANDLE hDevice)
```

调用 ap_register_cleanup **方法登记针对** hDevice **的** cleanup **信息，**h_cleanup **为清除方法。**

ap_pcloseh 方法

```
API_EXPORT(int) ap_pcloseh(pool *a, HANDLE hDevice)
```

将关闭句柄 hDevice**，并去除资源池** a **中记录的** cleanup **信息。**

socket_cleanup 方法

```
static void socket_cleanup(void *fdv)
```

调用 closesocket **方法关闭指定套接字** fdv**。**

ap_note_cleanups_for_socket 方法

```
API_EXPORT(void) ap_note_cleanups_for_socket(pool *p, int fd)
```

调用 ap_register_cleanup **方法登记针对** fd **的** cleanup **信息，**socket_cleanup **为清除方法。**

ap_kill_cleanups_for_socket 方法

```
API_EXPORT(void) ap_kill_cleanups_for_socket(pool *p, int sock)
```

将在资源池 p **的** cleanup **链表中与指定套接字** sock **匹配的** cleanup **结构去除。**

ap_psocket 方法

```
API_EXPORT(int) ap_psocket(pool *p, int domain, int type, int protocol)
```

按指定的域、类型及协议创建套接字，并调用 ap_note_cleanups_for_socket **方法记录对应的** cleanup **信息。**

ap_pclosesocket 方法

```
API_EXPORT(int) ap_pclosesocket(pool *a, int sock)
```

将套接字 sock **关闭，并调用** ap_kill_cleanup **方法将对应的** cleanup **结构从资源池** a **的** cleanup **链表中删除。**

7.3.5. 子进程信息

Apache 中对子进程的管理专门设计了一种机制，之所以未使用上述管理各种资源的通用机制，是出于效率的考虑。

定义了如下的 process_chain 结构来记录子进程的信息。

```
struct process_chain {
    pid_t pid;
    enum kill_conditions kill_how;
    struct process_chain *next;
};
```

其中 next 指针可使上述结构连成链，pool 结构的 subprocesses 域则为资源池对应子进程构成链表的头指针；pid 为子进程的 ID；kill_how 则表示销毁该子进程的方式，取如下枚举类型值：

```
enum kill_conditions {
    kill_never,  /* process is never sent any signals */
    kill_always, /* process is sent SIGKILL on pool cleanup */
    kill_after_timeout,  /* SIGTERM, wait 3 seconds, SIGKILL */
    just_wait,   /* wait forever for the process to complete */
    kill_only_once   /* send SIGTERM and then wait */
};
```

相关的处理方法如下。

ap_note_subprocess 方法

```
API_EXPORT(void) ap_note_subprocess(pool *a, pid_t pid, enum kill_conditions how)
```

本方法创建 process_chain 结构，记录给定子进程的信息，并链入到资源池 a 的 subprocesses 链表中。在 spawn_child_core 方法中被调用。

spawn_child_core 方法

```
static pid_t spawn_child_core(pool *p, int (*func) (void *, child_info *), void *data,enum kill_conditions kill_how, int *pipe_in, int *pipe_out, int *pipe_err)
```

被下面的 ap_spawn_child 方法调用执行(*func) (data, &info)方法，其中创建三个管道，它们的一端分别连到 func 方法对应子进程的标准输入、标准输出及标准错误输出，另一端的句柄则放到 pipe_in、pipe_out、pipe_err 所指向的空间中，传到外面。

ap_spawn_child 方法

```
API_EXPORT(int) ap_spawn_child(pool *p, int (*func) (void *, child_info *),
```

```
void *data, enum kill_conditions kill_how, FILE **pipe_in, FILE **pipe_out, FILE
**pipe_err)
```

与下面的 ap_bspawn_child **方法功能类似，不同之处在于本方法建立的管道两端以** FILE **形式出现，而** ap_bspawn_child **方法则使用** BUFF **结构。**

ap_bspawn_child 方法

```
API_EXPORT(int) ap_bspawn_child(pool *p, int (*func) (void *, child_info *),
void *data, enum kill_conditions kill_how, BUFF **pipe_in, BUFF **pipe_out, BUFF
**pipe_err)
```

本方法通过调用 func(data, &info)**执行指定的子进程。传入的** pipe_in、pipe_out、pipe_err **参数在外部环境和子进程间提供了通信的途径（管道）。详细的介绍请参阅本章第** 7 **节中关于** mod_cgi **的部分。**

若读者仔细读对应的源代码的话，可能会发现上述方法中，针对 Windows **平台的部分，并未创建新进程，而其他平台则使用了** fork **调用。在** Windows **平台下，上述方法虽然从名字上看是创建子进程的，实际并非如此，而仅是创建了一些管道，来进行交互而已。真正创建子进程的动作实际是在上述各方法的参数** func **指示的函数中完成的。例如在** mod_cgi **模块中，**cgi_child **方法被作为** func **传入，该方法中调用** ap_call_exec **方法实际创建了子进程来执行请求的脚本。再如在** mod_rewrite **模块中，**rewritemap_program_child **方法也被作为** func **传入，其中直接调用了** CreateProcess **方法来创建子进程。**

7.3.6. server_rec 结构

server_rec **结构用来记录服务器的信息。这里的“服务器”，不是指“**Apache **服务器”中的“服务器”，而是指** Apache **中采用虚拟主机机制而引出的主服务器或虚拟服务器（或主机）。具体定义如下：**

```
struct server_rec {
    server_rec *next;

    const char *defn_name;
    unsigned defn_line_number;

    char *srm_confname;
    char *access_confname;

    char *server_admin;
    char *server_hostname;
    unsigned short port;

    char *error_fname;
    FILE *error_log;
    int loglevel;
```

```
    int is_virtual;
    void *module_config;
    void *lookup_defaults;

    server_addr_rec *addrs;
    int timeout;
    int keep_alive_timeout;
    int keep_alive_max;
    int keep_alive;
    int send_buffer_size;

    char *path;
    int pathlen;

    array_header *names;
    array_header *wild_names;

    uid_t server_uid;
    gid_t server_gid;

    int limit_req_line;
    int limit_req_fieldsize;
    int limit_req_fields;
};
```

Apache 在 ap_read_config 方法中，在处理配置文件之前，首先调用 init_server_config 方法创建一个 server_rec 结构存放主服务器的配置信息，并被 ap_read_config 方法返回后存放在全局变量 server_conf 中。

若配置文件中定义了 VirtualHost 片段，则在 core 模块的 virtualhost_section 方法（负责处理<VirtualHost 配置命令）中，会再创建一个 server_rec 结构，存放相应虚拟主机的配置信息，并被链接到对应外部环境（对 VirtualHost 片段而言）的 server_rec 结构后，放到以 server_conf 为头指针的链表中。

在 server_rec 结构中，next 使 server_rec 结构可以相互连接构成链表；defn_name 和 defn_line_number 指示当前虚拟主机对应的 VirtualHost 片段所在文件及具体位置（行号）；srm_confname 和 access_confname 分别存放 srm.conf 文件和 access.conf 配置文件的完整路径；server_admin 存放当前服务器管理员的 Email 地址（由 ServerAdmin 配置命令给出）；server_hostname 和 port 给出服务器的地址名及端口号，这在进行重定向时会用到（参阅 ServerName 和 Port 配置命令）；error_fname 、error_log 和 loglevel 分别为出错日志文件名、句柄和日志级别（参阅 TransferLog 和 LogLevel 配置命令）；is_virtual 指示当前描述的是主服务器还是虚拟主机；module_config 为一向量结构，存放各模块针对本服务器的对应配置信息结构；lookup_defaults 与 module_config 类似，但它是针对目录级别上的配置信息结构；addrs 给出服务器绑定的地址集合；timeout、keep_alive_timeout、keep_alive_max、keep_alive、send_buffer_size、limit_req_line、limit_req_fieldsize、limit_req_fields 等给出关于 HTTP 连接的一些参数；path 存放 ServerPath 配置值，pathlen 为其长度；names 和 wild_names 两个数组分别存放服务器的普通别名和包含通配符的别名（参阅 ServerAlias 配置命令）。

7.3.7. command_rec 结构

command_rec 结构用来定义配置命令的处理信息。各个模块必须用 command_rec 结构的数组来定义它们能处理哪些配置命令及如何处理。

```
typedef struct command_struct {
    const char *name;
    const char *(*func) ();
    void *cmd_data;
    int req_override;
    enum cmd_how args_how;
    const char *errmsg;
} command_rec;
```

其中 name 中存放配置命令名；func 则为当前模块中负责处理对应配置命令的方法；cmd_data 存放配置命令的额外信息，从而使同时处理多条命令的方法能将各配置区分开，如在 core 模块中 dirsection 方法同时负责处理 Directory 和 DirectoryMatch，此域分别置为 0（NULL）和 1，或者存放一个地址偏移使同一个方法（如 ap_set_string_slot 等）可以完成对多个配置命令的简单处理（数值记录）；req_override 指示该命令适用范围的掩码；args_how 则给出配置参数的格式。对此更详细的介绍参阅第 6 章第 2 节。

7.3.8. cmd_parms 结构

cmd_parms 结构用来存放处理配置命令时所需的辅助内容。在处理任何配置信息文件时，均要创建一个此结构，来传递相关信息给命令处理方法。

```
typedef struct {
    void *info;
    int override;
    int limited;
    configfile_t *config_file;
    ap_pool *pool;
    struct pool *temp_pool;
    server_rec *server;
    char *path;
    const command_rec *cmd;
    const char *end_token;   /* end token required to end a nested section */
} cmd_parms;
```

其中 cmd 为各模块中声明可处理当前配置命令的 command_rec 数组的对应单元；info 域与 cmd 中的 cmd_data 取值相同；path 中存放当前正在处理的目录路径，不适用时则为 NULL；end_token 为当前处理的嵌套的配置命令片段的结束标识；server 为配置信息被处理后被记录存放的位置，即处理的信息是针对由 server 代表的对象的，如某个虚拟主机；config_file 存放要处理的配置文件句柄；pool 和 temp_pool 则是处理过程中使用的资源池，其中后者仅用来满足在处理请求前临时性的空间等资源需求。

实际上，关于 cmd_parms 结构的使用是通过如下 cmd_parms 类型的全局变量 default_parms 来完成的。

```
static cmd_parms default_parms =
{NULL, 0, -1, NULL, NULL, NULL, NULL, NULL, NULL, NULL};
```

default_parms 中各域均取缺省值。

在 Apache 对某一系列配置信息进行处理之前，总是在 default_parms 的基础上先更新某些信息，然后再将其作为参数传递到随后的具体配置命令处理方法中。如在处理主配置文件时，是在 ap_process_resource_config 方法中进行上述过程的；而在处理目录下的配置文件时，则在 ap_parse_htaccess 方法中；由启动参数“-C”或“-c”指定的配置文件的处理中，上述过程则出现在 process_command_config 方法中。

7.3.9. 模块接口控制块

在第 6 章中，我们已经知道 Apache 采用模块机制来分布功能实现。每个模块都对应一个接口控制块——module 结构，其具体定义如下：

```
typedef struct module_struct {
    int version;
    int minor_version;
    int module_index;
    const char *name;
    void *dynamic_load_handle;

    struct module_struct *next;

    unsigned long magic;

    void (*init) ();
    void *(*create_dir_config) ();
    void *(*merge_dir_config) ();
    void *(*create_server_config) ();
    void *(*merge_server_config) ();

    const command_rec *cmds;
    const handler_rec *handlers;

    int (*translate_handler) (request_rec *);
    int (*ap_check_user_id) (request_rec *);
    int (*auth_checker) (request_rec *);
    int (*access_checker) (request_rec *);
    int (*type_checker) (request_rec *);
    int (*fixer_upper) (request_rec *);
    int (*logger) (request_rec *);
    int (*header_parser) (request_rec *);

    void (*child_init) ();
    void (*child_exit) ();
```

```
    int (*post_read_request) (request_rec *);
} module;
```

其中 version 和 minor_version 共同给出当前遵循的 Apache 模块 API 的版本号（注意并非是模块自身的版本号）；module_index 存放本结构在模块链中的偏移位置；name 给出模块名。

next 提供了 module 结构相互链接的机制，通过该域多个接口控制块可以连成链表。

init 存放模块初始化方法的句柄，在 Apache 处理完配置信息后，但在创建子进程之前，会调用各个模块的 init 方法，来对各个模块进行初始化；create_dir_config 和 create_server_config 分别存放创建针对特定目录的和针对服务器的配置信息块的方法句柄，而 merge_dir_config 和 merge_server_config 则分别完成将上述同种信息块进行合并。

cmds 给出当前模块所能处理的配置命令列表；而 handlers 则给出响应处理方法列表。

translate_handler、ap_check_user_id、auth_checker、access_checker、type_checker、fixer_upper、logger 等均为除响应阶段外的对应请求处理各阶段的方法句柄（参阅第 6 章第 2 节）。

child_init 和 child_exit 域的设置是针对重量级进程（heavy weight process）而设置的，即这些进程在运行时要占用较多的内存、文件句柄等资源。严格地讲，它们并非天生就是“重量级”的，而是由于它们要处理某些特定的请求才使然；请求不同，处理请求涉及的模块也就不同，而正是某些“重量级”的模块使这些进程有此称谓。正由于某些模块资源需求大，所以为了避免资源的占用时间过长，专门设置了 child_init 和 child_exit 方法句柄，分别在子进程创建和销毁时被调用，同时完成资源的申请和释放；相反若没有此两个方法，则要在 init 方法中申请资源，并在 Apache 服务器退出时才释放，占用时间过长会使 Apache 服务器负载过重。

源代码根目录的 os\win32\modules.c 文件中，定义了如下两个数组，声明了一系列的 Apache 缺省模块。

ap_preloaded_modules 数组定义了要装入的模块：

```
module *ap_preloaded_modules[] = {
  &core_module,
  &so_module,
  &mime_module,
  &access_module,
  &auth_module,
  &negotiation_module,
  &includes_module,
  &autoindex_module,
  &dir_module,
  &cgi_module,
  &userdir_module,
  &alias_module,
  &env_module,
  &config_log_module,
  &asis_module,
  &imap_module,
  &action_module,
```

```
  &setenvif_module,
  &isapi_module,
  NULL
};
```

ap_prelinked_modules **数组定义了在装入后要激活的模块：**

```
module *ap_prelinked_modules[] = {
  &core_module,
  &so_module,
  &mime_module,
  &access_module,
  &auth_module,
  &negotiation_module,
  &includes_module,
  &autoindex_module,
  &dir_module,
  &cgi_module,
  &userdir_module,
  &alias_module,
  &env_module,
  &config_log_module,
  &asis_module,
  &imap_module,
  &action_module,
  &setenvif_module,
  &isapi_module,
  NULL
};
```

Apache **中又定义了两个变量** ap_loaded_modules **和** top_module**，分别存放装入与激活的模块控制块指针。但是，前者为一个单元类型为** module **结构指针的数组（以** NULL **表示结束），而后者则表现为单链表的头指针，每激活一个模块，就将对应的控制块添加到链表头，其中每个** module **结构的** next **域指向下一个链节。**

```
API_VAR_EXPORT module **ap_loaded_modules;
API_VAR_EXPORT module *top_module = NULL;
```

又定义了 total_modules **和** dynamic_modules **两个变量分别存放装入与激活的模块数目。**

```
static int total_modules = 0;
static int dynamic_modules = 0;
```

7.3.10. 处理方法句柄的简捷组织结构

上文我们已经提到各模块可以定义针对访问请求某处理阶段的处理方法，方法的句柄要存放在对应模块的 module **结构中。**

这里的问题是，一般模块只是定义一两个处理句柄，若处理请求时遍历所有模块及所有句柄，非空者再被调用的话，效率将很低。可能处理一两次请求消耗的时间并不多，但当请求量单位时间很大时，则时间延迟将非常可观。为了消除这一问题，Apache **中在将各模块**

装入并完成了初始化后，实际上对各模块的处理方法句柄又进行了组织，即将所有非空的句柄单独拿出来，并按照处理阶段进行排列，处理请求时直接访问此结构，从而大大提高了效率。

Apache 中定义了如下两个变量：

```
static handler_func *method_ptrs;
static struct {
    int translate_handler;
    int ap_check_user_id;
    int auth_checker;
    int access_checker;
    int type_checker;
    int fixer_upper;
    int logger;
    int header_parser;
    int post_read_request;
} offsets_into_method_ptrs;
```

其中，method_ptrs 为一个大数组，其中空间被按照处理阶段分为若干片段，分别连续存放对应阶段的非空处理句柄。如图 7.7 所示：

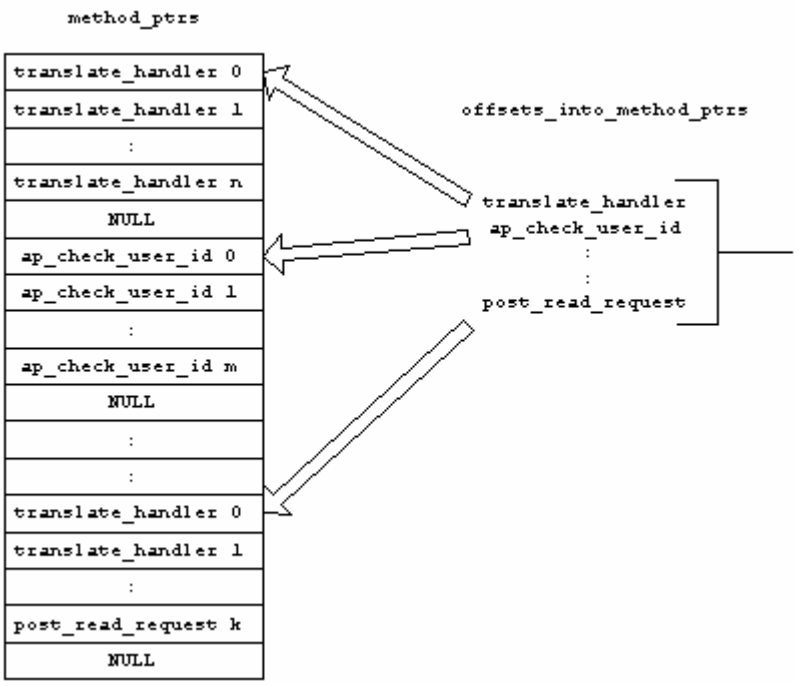


图 7.7

offsets_into_method_ptrs 中各个域则存放 method_ptrs 数组中各处理阶段片段的偏移地址，即提供快速定位信息，如 method_ptrs [offsets_into_method_ptrs.logger]就指向第一个日志阶段处理方法句柄。

细心的读者可能会发现请求处理的响应阶段在上面并未被考虑。实际上，Apache 中专门定义了如下变量：

```
static fast_handler_rec *handlers;
static fast_handler_rec *wildhandlers;
```

两者均为数组结构，最后一个单元为 NULL，表示结束。

wildhandlers 为包含统配符的响应处理句柄的链表头指针，如在 core 模块中定义的：

```
{ "*/*", default_handler }
```

handlers 则为不含统配符的响应处理句柄的链表头指针，如在 proxy 模块中定义的：

```
{"proxy-server", proxy_handler},
```

具体的源代码实现请参见 http_config.c。

build_method_shortcuts 方法

```
static void build_method_shortcuts(void)
```

本方法用来构造各模块中对访问请求各阶段处理方法句柄的简捷组织方式。

具体过程如下：

首先统计全部的处理方法的数目，据此数目分配足够的空间赋给全局变量 method_ptrs，该变量即存放各句柄组织结构的指针。

然后,将各非空的处理方法句柄按处理阶段的不同分别连续地放入 method_ptrs 空间中，并用 offsets_into_method_ptrs 变量存放辅助的位置信息。

run_method 方法

```
static int run_method(request_rec *r, int offset, int run_all)
```

本方法完成针对当前请求 r 的各个模块定义的同一类处理方法的调用。其中 offset 为需要执行的该类方法句柄在 method_ptrs 数组中的偏移下标地址，run_all 指示是否执行所有的该类方法。

本方法循环对每个处理方法进行调用。当为如下两种情况之一时，跳出循环，返回当前处理方法的返回值：

1. 返回值既非 DECLINED（表示不进行处理）也非 OK（表示进行了正常处理），表示处理时出现异常；

2. 返回值为 OK 即已进行了正常处理，且 run_all 为 0 即未要求执行所有处理方法。

这样直至循环结束，如果 run_all 为 1，返回 OK，否则返回 DECLINED。

在 http_config.c 中，读者会看到有如下方法实际是调用了 run_method 方法分别各类请求处理方法句柄的执行：

表 7.1

方法名	请求处理方法类型	run_all
ap_translate_name	translate_handler	0
ap_check_access	access_checker	1
ap_find_types	type_checker	0
ap_run_fixups	fixer_upper	1
ap_log_transaction	logger	1
ap_header_parse	header_parser	1
ap_run_post_read_request	post_read_request	1
ap_check_user_id	ap_check_user_id	0
ap_check_auth	auth_checker	0

ap_invoke_handler 方法

```
int ap_invoke_handler(request_rec *r)
```

本方法调用合适的返回响应句柄对请求 r 进行处理。

init_handlers 方法

```
static void init_handlers(pool *p)
```

本方法对统配符（*）请求处理句柄及各特定处理句柄进行整理和组织，分别放入全局变量 wildhandlers 和 handlers 中，以提高请求处理速度。

7.3.11. listen_rec 结构

Apache 可以在多个地址（包含端口号）上进行监听，为了表达这种对特定地址的监听，定义了如下的 listen_rec 结构：

```
struct listen_rec {
    listen_rec *next;
    struct sockaddr_in local_addr;  /* local IP address and port */
    int fd;
    int used;
};
```

其中 next 可使 listen_rec 结构连接成链表；local_addr 为监听请求时使用的本地地址；fd 为监听到请求后客户端的连接句柄。

同时，Apache 采用了多进程及多线程模式对外部请求进行处理，故为便于请求处理任务的分配，又设置了如下两个变量管理可能的多个监听地址构成的链表：

```
listen_rec *ap_listeners;
static listen_rec *head_listener;
```

在初始化期间，ap_listeners 链表是一个以 NULL 结束的线性链表。为了使链表中后面的 socket 得到及时处理，将 ap_listeners 连成循环链表，每处理完一个 socket，head_listener 指向下一个 socket。

7.3.12. 工作任务队列

上文我们提到，Apache 在 worker_main 方法的主控制流程中通过 listen_rec 结构进行监听，而以 child_sub_main 为主体的工作线程则对接收到的连接请求进行处理。这样，在监听方与处理方之间需要有某种机制来表达要处理的连接请求，即工作任务。

Apache 中定义了如下的 joblist_s 结构来表达一个工作任务：

```
typedef struct joblist_s {
    struct joblist_s *next;
    int sock;
} joblist;
```

其中 next 使 joblist_s 结构可以构成链结构，而 sock 域则为接收到请求的 socket 端口号。

负责监听的 worker_main 方法的主控制流程每接收到一个连接请求，则调用 add_job 方法创建一个 joblist 结构，并将该结构附加到工作任务队列中。该队列由如下 globals 结构和 allowed_globals 变量定义：

```
typedef struct globals_s {
    int exit_now;
    semaphore *jobsemaphore;
    joblist *jobhead;
    joblist *jobtail;
    mutex *jobmutex;
    int jobcount;
} globals;
```

其中，jobmutex 为互斥访问工作任务队列的互斥量，使添加和删除可协调工作；jobhead 和 jobtail 分别为队列的首和尾指针；jobcount 为队列中工作任务的数目。

每增加一个新任务，总添加到队列的末尾；每取出一个任务，则从队首取。

add_job 方法

```
void add_job(int sock)
```

本方法根据给定的 socket 端口号创建一个新的工作任务，即 joblist 结构，从添加到工作任务队列中。

remove_job 方法

```
int remove_job(void)
```

本方法从工作任务队列中，取出一个任务，返回对应的接收到连接请求的端口号。

7.3.13. BUFF 结构

Apache 中并未使用 C 库中现成的 I/O 例程，而是又重新定义了一整套具有缓冲功能的 I/O 体系。其定义的 I/O 体系有如下优点：

- 对文件描述符的处理有确定的语义；
- 对同一文件描述符同时进行读写不会出现问题；
- 对可打开的文件句柄数目没有限制；
- 只使用内存资源，从而省去了每次都要记着调用 close 例程的麻烦；
- 在读写操作前可以对超时进行控制；
- 具有错误处理机制。

在整个 Apache 的 I/O 体系中，BUFF 结构为核心。有如下定义：

```
typedef struct buff_struct BUFF;

struct buff_struct {
    int flags;
```

```
    unsigned char *inptr;
    int incnt;
    int outchunk;
    int outcnt;
    unsigned char *inbase;
    unsigned char *outbase;
    int bufsiz;
    void (*error) (BUFF *fb, int op, void *data);
    void *error_data;
    long int bytes_sent;
    ap_pool *pool;
    int fd;
    int fd_in;
    HANDLE hFH;
    void *t_handle;
};
```

其中 inbase、outbase **分别为进行读、写操作时使用的缓冲区；**inptr **为读指针，指示** inbase **中要读的下一个字节的地址；**incnt、outcnt、bytes_sent **分别记录** inbase **中未读的字节数、**outbase **中已写入的字节数及实际写到目标流中的字节数；**flags **为** BUFF **结构的属性标志，其中每个** bit **表示一种含义，如** B_SOCKET、B_RD、B_WR **等；**error、error_data **分别为出现错误时使用的处理方法句柄及辅助信息。**

与 BUFF **结构相关的方法按功能的不同，可分为如下几类：**

- **流管理函数（完成流的创建、关闭及参数的设置等）：**

```
ap_bcreate
ap_bpushfd
ap_bpushh
ap_bsetopt
ap_bgetopt
ap_bsetflag
ap_bclose
```

- **底层输入输出函数：**

```
sendwithtimeout
recvwithtimeout
ap_read
buff_read
ap_write
buff_write
doerror
```

- **与数据块（**chunk**）相关的方法：**

```
start_chunk
end_chunk
```

- **高层读写方法：**

```
ap_bnonblock
ap_bfileno
```

```
ap_bhalfduplex
saferead_guts
read_with_errors
ap_bread
ap_bgets
ap_blookc
ap_bskiplf
ap_bflsbuf
ap_bfilbuf
write_it_all
write_with_errors
bcwrite
ap_bwrite
bflush_core
ap_bflush
ap_bputs
ap_bvputs
ap_bonerror
```

- printf **类方法：**

```
bprintf_flush
ap_bprintf
ap_vbprintf
```

ap_bclose 方法

```
API_EXPORT(int) ap_bclose(BUFF *fb);
```

本方法同步（synchronize**）并关闭与** fb **关联的文件，并设置** EOF **或** EOUT **标志，指示不能再读或写。**

ap_bcreate 方法

```
API_EXPORT(BUFF *) ap_bcreate(pool *p, int flags);
```

创建一个 BUFF **结构，**flags **为设置的属性。**

若该 BUFF **结构将用于读，即** flags **中的** B_RD **位为** 1**，则创建读缓冲区，其地址存于** BUFF **结构的** inbase **域中，否则** inbase **域为** NULL**；若该** BUFF **结构将用于写，即** flags **中的** B_WR **位为** 1**，则创建写缓冲区，其地址存于** BUFF **结构的** outbase **域中，否则** outbase **域为** NULL**。本方法还对缓冲区读写指针、读写字节计数及各种句柄进行了初始化。**

ap_bfilbuf 方法

```
API_EXPORT(int) ap_bfilbuf(BUFF *fb)
```

本方法调用 ap_bread **方法装满** fb **的输入缓冲区，并从中取第一个字节返回。若遇到流**

结束或出错，返回 0。

ap_bfileno 方法

```
API_EXPORT(int) ap_bfileno(BUFF *fb, int direction)
```

本方法按 direction 的指示返回 fb 中的输入或输出文件描述符。

ap_bflsbuf 方法

```
API_EXPORT(int) ap_bflsbuf(int c, BUFF *fb)
```

本方法调用 ap_bwrite 向 fb 输出缓冲区中写一个字节。

ap_bflush 方法

```
API_EXPORT(int) ap_bflush(BUFF *fb)
```

本方法发送输出缓冲区的所有数据。成功时返回 0，否则返回-1。
具体处理为：
如果 fb 当前不是处于正常的可写状态，则返回-1。
如果是分块编码模式，则调用 end_chunk 方法结束当前块。
调用 bflush_core 实际发送缓冲区中所有数据。
如果发送成功并且是分块方式，开始下一个块（start_chunk）。
返回状态信息。

ap_bgetopt 方法

```
API_EXPORT(int) ap_bgetopt(BUFF *fb, int optname, void *optval)
```

本方法获取通过 fb 发送的字节数，为 fb->bytes_sent 与 fb->outcnt 之和，存于*optval 中传出。

ap_bgets 方法

```
API_EXPORT(int) ap_bgets(char *buff, int n, BUFF *fb);
```

本方法从 fb 指示的缓冲流中读取指定数目的字节或一行放到 buff 中，返回放在 buff 中的字节数。若直接读到文件结尾，则返回 0；否则即出错，返回-1。读时对可能出现的回车/换行符进行了处理，用一个换行符代替。

ap_bhalfduplex 方法

```
API_EXPORT(void) ap_bhalfduplex(BUFF *fb)
```

本方法用来保证在进行下次读写操作时不会造成阻塞。实际只是在 saferead_guts 方法中被使用，来确保读操作的成功。

ap_blookc 方法

```
API_EXPORT(int) ap_blookc(char *buff, BUFF *fb);
```

本方法将 fb 输入缓冲区中的第一个字节拷贝到 buff，但不从输入缓冲区中移走该字节。成功时返回 1，遇到流结束返回 0，否则返回-1。

ap_bnonblock 方法

```
API_EXPORT(int) ap_bnonblock(BUFF *fb, int direction);
```

在 Win32 平台下本方法未做任何操作。

ap_bonerror 方法

```
API_EXPORT(void) ap_bonerror(BUFF *fb, void (*error) (BUFF *, int, void *),
void *data);
```

本方法设置 BUFF 结构 fb 中的错误处理方法句柄为 error 及辅助信息为 error_data。

ap_bprintf 方法

```
API_EXPORT_NONSTD(int) ap_bprintf(BUFF *fb, const char *fmt, ...)
```

本方法以指定格式将字符串写入 fb 的输出缓冲区，如果必要发送区中数据。其中使用了 AP\ap_snprintf.c 中的 ap_vformatter 方法。

ap_bpushfd 方法

```
API_EXPORT(void) ap_bpushfd(BUFF *fb, int fd_in, int fd_out);
```

本方法将 fb 的 fd 和 fd_in 域的值分别置为传入的文件句柄 fd_in 和 fd_out，即实际是将 fb 与对应的文件相联系，当对 fd_in 读或对 fd_out 写时，使用 fb 来缓冲。

ap_bpushh 方法

```
API_EXPORT(void) ap_bpushh(BUFF *fb, HANDLE hFH);
```

本方法将 fb->hFH 置为 hFH，通过 fb 来缓冲对句柄 hFH 所代表的流的读或写。

ap_bputs 方法

```
API_EXPORT(int) ap_bputs(const char *x, BUFF *fb);
```

本方法调用 ap_bwrite 方法写一个字符串。成功时返回字符串长度，否则返回-1。

ap_bread 方法

```
API_EXPORT(int) ap_bread(BUFF *fb, void *buf, int nbyte);
```

从 fb 指示的缓冲流中读取指定数目的字节放到 buff 中，返回放在 buff 中的字节数。若直接读到文件结尾，则返回 0；否则即出错，返回-1。本方法被 ap_bfilbuf 方法调用。

ap_bsetopt 方法

```
API_EXPORT(int) ap_bsetopt(BUFF *fb, int optname, const void *optval)
```

本方法用来设置 fb->bytes_sent 的值，*optval 中的值为 fb->outcnt 与 fb->bytes_sent 之和。

ap_bskiplf 方法

```
API_EXPORT(int) ap_bskiplf(BUFF *fb);
```

本方法跳过数据，直到遇到 LF 字符。若成功则返回 1，没有找到返回 0，出现错误返回-1。

ap_bvputs 方法

```
API_EXPORT(int) ap_bvputs(BUFF *fb,...);
```

本方法向缓冲区中写入传入的可变数目的字符串，并返回写入的字节数。

ap_bvputs 方法

```
API_EXPORT_NONSTD(int) ap_bvputs(BUFF *fb,...)
```

本方法写多个字符串。如果出错，返回-1；否则返回写入的数据总长度。

ap_bwrite 方法

```
API_EXPORT(int) ap_bwrite(BUFF *fb, const void *buf, int nbyte);
```

本方法向 fb 的输出缓冲区写 nbyte 个字节，返回实际写入的字节数。如果在错误发生之前没有写入任何数据，返回-1。

ap_read 方法

```
static int ap_read(BUFF *fb, void *buf, int nbyte)
```

本方法读 fb->hFH 代表的非 socket 流，并返回操作结果状态。

ap_vbprintf 方法

```
API_EXPORT(int) ap_vbprintf(BUFF *fb, const char *fmt, va_list vlist);
```

本方法将由 fmt，ap 定义的格式化字符串写入输出缓冲区，如果必要发送区中数据。与 ap_bprintf 方法的不同在于参数个数可变。

ap_write 方法

```
static int ap_write(BUFF *fb, const void *buf, int nbyte)
```

本方法与 ap_read 方法类似，但方向相反，是向 fb->hFH 代表的非 socket 流中写。

bcwrite 方法

```
static int bcwrite(BUFF *fb, const void *buf, int nbyte)
```

本方法实现非缓冲方式写流功能，支持分块模式，并负责错误处理及刷新 bytes_sent。在分块模式下，调用 write_it_all 方法具体实现，否则调用 write_with_errors 方法。在 ap_bwrite 方法中被使用。

bflush_core 方法

```
static int bflush_core(BUFF *fb)
```

本方法调用 write_with_errors 方法发送输出缓冲区中的数据。

具体是循环发送数据，并检查实际发送的数据字节数。如果小于 0，则返回-1；否则若输出缓冲区中还有尚未发送的数据，则将其移到缓冲区开始位置。重复执行上述操作，直至发送完所有数据或出错。

bprintf_flush 方法

```
static int bprintf_flush(ap_vformatter_buff *vbuff)
```

本方法调用 ap_bflush 方法发送 vbuff->fb 的输出缓冲区数据，刷新其 outcnt 值，并重新设置 vbuff 的缓冲区起始及结束位置。在 ap_bprintf、ap_vbprintf 中被使用。

buff_read 方法

```
static ap_inline int buff_read(BUFF *fb, void *buf, int nbyte)
```

本方法从 fb 指示的流中读取 nbyte 个字节的数据放入 buf 中。如果目标流是 socket 流，则调用 recvwithtimeout 方法具体处理，否则调用 ap_read。本方法被 saferead_guts 方法调用。

buff_write 方法

```
static ap_inline int buff_write(BUFF *fb, const void *buf, int nbyte)
```

本方法完成将指定数目的数据写入 fb 指示的流中。如果目标流是 socket 流，则调用 sendwithtimeout 方法具体处理，否则调用 ap_write。本方法被 write_it_all、write_with_errors 方法调用。

doerror 方法

```
static void doerror(BUFF *fb, int direction)
```

按 direction 指示的读写方向，在 fb->flags 中记录出错标志。并去调用 fb 中已记录其句柄的出错处理方法。

end_chunk 方法

```
static void end_chunk(BUFF *fb)
```

本方法结束以分块编码模式发送数据。

具体处理为：

如果当前不是进行分块发送，则直接返回。

如果块的长度为 0，由于长度为 0 的块表示 EOF，所以删除该块，更新指针及计数器并返回。否则，生成最后一个块结构，并写入 CRLF，表示 EOF。

最后设 fb->outchunk 为-1，取消分块编码模式。

read_with_errors 方法

```
static int read_with_errors(BUFF *fb, void *buf, int nbyte)
```

本方法完成读流操作。

具体则是在 saferead（即 saferead_guts）方法的基础上又进行了出错及 EOF 流结束标志检查。若遇到流结束，则在 fb->flags 中设置 B_EOF 标志。出错时调用 doerror 方法进行错误处理。最终返回 saferead 的返回值。

recvwithtimeout 方法

```
int recvwithtimeout(int sock, char *buf, int len, int flags)
```

本方法实现 Win32 平台下读 socket 流功能。具体读操作是通过 recv 方法完成的。

saferead_guts 方法

```
static ap_inline int saferead_guts(BUFF *fb, void *buf, int nbyte)
```

本方法为安全读操作的核心程序，对 buff_read 方法进一步包装。在 read_with_errors 方法中被调用，表现为 saferead 方法。

sendwithtimeout 方法

```
int sendwithtimeout(int sock, const char *buf, int len, int flags)
```

本方法实现 Win32 平台下写 socket 流功能。具体读操作是通过 send 方法完成的。

start_chunk 方法

```
static void start_chunk(BUFF *fb)
```

本方法完成以分块编码模式发送数据的初始化。

具体处理为：

若已经进行了分块（即 fb->outchunk != -1）或 fb 当前不是处于正常的可写状态，则直接返回。

如果当前空闲输出缓冲区小于 CHUNK_HEADER_SIZE + 1，则调用 bflush_core 方法发送缓冲区中的数据，以便腾出足够的空闲空间。

最后在 outchunk 中设置分块开始位置，并更新输出缓冲区计数器 outcnt。

write_it_all 方法

```
static int write_it_all(BUFF *fb, const void *buf, int nbyte)
```

本方法向 fb 指示的目标流写入 buf 中的 nbyte 个字节数据，直至发送完毕或出错，并刷新 bytes_sent 值。若发生错误，在 fb->flag 中置 B_WRERR 标志。本方法成功时返回 0，否则返回-1。具体处理调用 buff_write 完成。

write_with_errors 方法

```
static int write_with_errors(BUFF *fb, const void *buf, int nbyte)
```

本方法进一步包装基本的 buff_write 方法，完成包括错误处理及 bytes_sent 域更新在内的写流操作。成功时返回 0，否则返回错误值。本方法又被 bcwrite、ap_bwrite、bflush_core

方法调用。

7.3.14. conn_rec 结构

conn_rec 结构记录每一次客户到服务器的连接信息。具体定义入下：

```
struct conn_rec {
    ap_pool *pool;
    server_rec *server;
    server_rec *base_server;
    void *vhost_lookup_data;
    int child_num;
    BUFF *client;
    struct sockaddr_in local_addr;
    struct sockaddr_in remote_addr;
    char *remote_ip;
    char *remote_host;
    char *remote_logname;
    char *user;
    char *ap_auth_type;
    unsigned aborted:1;
    signed int keepalive:2;
    unsigned keptalive:1;
    signed int double_reverse:2;
    int keepalives;
};
```

在工作线程的主体 child_sub_main 中，从工作任务队列中取得一个 socket 端口号后，调用 new_connection 方法创建一个 conn_rec 结构，表示本次连接。

在 conn_rec 结构中，pool 为处理本次连接所使用的资源池；server 为当前服务器对应的配置结构；child_num 为当前子进程标识；client 为进行 I/O 所需的缓冲区；local_addr、remote_addr 分别为本地和远程的 sockaddr 结构类型地址；remote_ip、remote_host 分别为远程主机的 IP 地址串和域名；user 和 ap_auth_type 中则存放与身份认证相关的用户名和认证类型；aborted 表示当前连接是否还有效；keepalive 和 keptalive 分别表示当前是否正在和曾经使用持续连接机制。

conn_rec 结构创建后，可以被传递给 ap_read_request 方法，调用该方法一次读取一次请求，并进一步调用 ap_process_request 方法对请求进行处理。

7.3.15. request_rec 结构

```
struct request_rec {
    ap_pool *pool;
    conn_rec *connection;
    server_rec *server;
    request_rec *next;
```

```
request_rec *prev;
request_rec *main;
char *the_request;
int assbackwards;
int proxyreq;
int header_only;
char *protocol;
int proto_num;
const char *hostname;
time_t request_time;
const char *status_line;
int status;
const char *method;
int method_number;
int allowed;
int sent_bodyct;
long bytes_sent;
time_t mtime;
int chunked;
int byterange;
char *boundary;
const char *range;
long clength;
long remaining;
long read_length;
int read_body;
int read_chunked;
unsigned expecting_100;
table *headers_in;
table *headers_out;
table *err_headers_out;
table *subprocess_env;
table *notes;
const char *content_type;
const char *handler;
const char *content_encoding;
const char *content_language;
array_header *content_languages;
char *vlist_validator;
int no_cache;
int no_local_copy;
char *unparsed_uri;
char *uri;
char *filename;
```

```
    char *path_info;
    char *args;
    struct stat finfo;
    uri_components parsed_uri;
    void *per_dir_config;
    void *request_config;
    const struct htaccess_result *htaccess;
};
```

在 HTTP1.1 的持续连接机制下，建立一次连接可以发送多个请求。上文的 conn_rec 结构记录连接的信息，而 request_rec 结构则记录一次请求的信息。

在 request_rec 结构中，connection、server 等域是对外部大环境的引用；next、prev、main 等则使与请求重定向、产生子请求等相关的若干 request_rec 结构相互关联；其它域则基本是存放当前请求的各方面信息。

实际上在第 6 章第 2 节中已经对 request_rec 结构进行了介绍，在此不再赘述。

7.3.16. handler_rec 结构

handler_rec 结构用来定义某模块在指定情况下可使用的相应处理方法句柄，来返回请求的响应。具体定义如下：

```
typedef struct {
    const char *content_type;
    int (*handler) (request_rec *);
} handler_rec;
```

其中，content_type 区分不同的请求；handler 则是处理方法的句柄。

每个模块均可以处理一种或多种不同的请求，从而构成 handler_rec 结构的列表，该列表的指针存放在 module 结构的 handles 域中。

请参阅 http_config.c 中的 init_handlers、ap_invoke_handler 等方法。

第4节　主控程序的实现

前面已经介绍了 Apache 的控制流程，以期给读者一个比较整体性的连贯的思路。本节则详细分析其中重要的处理方法，从而使读者对 Apache 的实现机制既有全面的视图，也有细节上的了解。

本部分首先介绍 Apache 执行代码的部署形态，然后分别给出主程序各文件中方法的详细实现。

7.4.1. 主执行程序的形态

Windows 版本的 Apache 的入口点是在源代码根目录的 os\win32 子目录的 main_win32.c 文件中定义的：

```
__declspec(dllexport) int apache_main(int argc, char *argv[]);
int main(int argc, char *argv[])
```

```
{
    return apache_main(argc, argv);
}
```

其中可看出此入口点仅为形式上的，实际的位置则是某动态链接库中定义的 apache_main 方法，具体位置在 http_main.c 中。

上面我们已经提到 Apache 的主程序逻辑上由 6 个文件构成：http_main.c、http_protocol.c、http_request.c、http_config.c、http_log.c、http_vhost.c。但从物理存在的角度上看，这 6 个文件则是与 core 模块对应的 http_core.c 及其它各个标准模块的对应文件编译在一起，以 ApacheCore.dll 动态链接库的形式出现的。

这也是为什么我们看到在 Apache 执行目录下，仅包含 main_win32.c 的 Apache 执行程序 Apache.exe 大小为 4k，另外却还有一个 263k 的 ApacheCore.dll 的原因（以 Windows 下 Apache1.3.6 为例）。

如此设计的最大好处在于：将 Apache 分成了若干个相对独立的块，它们之间按照动态链接库的方式进行相互调用，而非通常的源代码间的直接调用，这样就使软件的协同开发变得非常方便，也易于软件的个别部分单独升级。

下面按方法所在的源代码文件分别介绍实现的具体细节。

7.4.2. http_main.c

ap_add_version_component 方法

```
API_EXPORT(void) ap_add_version_component(const char *component)
```

向版本标志串中增加指定的部分。

若 version_locked 非 0 表示标志串已被锁定，不进行更改；否则将 component 直接附加到标志串后。

ap_get_server_version 方法

```
API_EXPORT(const char *) ap_get_server_version(void)
```

获取版本标志串，返回之。

ap_set_version 方法

```
static void ap_set_version(void)
```

调用 ap_add_version_component 方法向版本标志串中增加版本信息。

ap_util_init 方法

为空方法。

ap_util_uri_init 方法

为空方法。

common_init 方法

```
static void common_init(void)
```

本方法完成基本的系统初始化。具体过程为：

调用 INIT_SIGLIST()宏（实际为 siglist_init 方法）初始化系统信号。

调用 AMCSocketInitialize 初始化 WinSock。

调用 ap_init_alloc 创建 permanent_pool 缓冲区池，然后赋给 pconf。

调用 ap_make_sub_pool 创建 pconf 的子池 ptrans。

调用 ap_util_init 方法，方法体实际为空。

调用 ap_util_uri_init 方法，方法体实际为空。

调用 ap_make_sub_pool 创建 pcommands 缓冲区池，并从其中分别申请空间创建 ap_server_pre_read_config、ap_server_post_read_config、ap_server_config_defines 三个数组，分别用于保存读配置文件前、后及文件中的配置信息。

default_listeners 方法

```
static void default_listeners(pool *p, server_rec *s)
```

本方法对 ap_listeners 链表进行初始化，创建一对应 s 的 listen_rec 结构（表示当前服务器与指定的地址和端口绑定），放入链表中，在 http_config.c 中的 ap_read_config 方法中被调用。

make_sock 方法

```
static int make_sock(pool *p, const struct sockaddr_in *server)
```

本方法根据提供的地址/端口号创建一个服务器端监听请求的 socket，并返回之。具体则是通过 socket、setsockopt、bind、listen 等系统调用来完成的。本方法在 setup_listeners 方法中被调用，设置所有的监听端口。

reset_version 方法

```
static void reset_version(void *dummy)
```

将版本标志字符串清空，并允许后续的更改。

setup_listeners 方法

```
static void setup_listeners(pool *p)
```

本方法创建所有监听端口，并将本是单向链表的 ap_listeners 结构改为环型，并初始化全局变量 head_listener 指向一起始位置。其中调用了 make_sock 方法来创建端口并使之进行监听。

setup_signal_names 方法

```
void setup_signal_names(char *prefix)
```

本方法用来根据传入的前缀串来构造 signal_name_prefix、signal_restart_name 和 signal_shutdown_name 等三个全局变量，后两者被用来创建信号量，实现进程间的交互。

siglist_init 方法

```
static void siglist_init(void)
```

初始化系统信号表 ap_sys_siglist，在 common_init 方法中被调用。

7.4.3. http_protocol.c

ap_basic_http_header 方法

```
API_EXPORT(void) ap_basic_http_header(request_rec *r)
```

将响应行及 Date 和 Server 头域发送给提出请求 r 的客户端。并将 Date 和 Server 头域从 r->headers_out 中清除，以免以后重发。

ap_discard_request_body 方法

```
API_EXPORT(int) ap_discard_request_body(request_rec *r)
```

在 HTTP1.1 中，任何请求都可以带请求体。但对于大多数 GET 请求处理方法来说，它们并不能识别请求体。本方法可以读取并简单地抛弃请求体，从而消除异常的发生。之所以这样做，是因为在持续连接的情况下，若对请求体处理不当，会对后续请求的接收造成影响。

ap_finalize_request_protocol 方法

```
API_EXPORT(void) ap_finalize_request_protocol(request_rec *r)
```

在要结束发送响应时调用本方法。

若是 chunked 结构的响应，则发送一个空的 chunked 块，表示已经发送完毕。否则，什

么也不做。

ap_get_client_block 方法

```
API_EXPORT(long) ap_get_client_block(request_rec *r, char *buffer, int bufsiz)
```

从请求 r 指示的与客户的连接中读取数据体，放到 buffer 中。

若不是要读 chunked 块，则读尽可能多的数据到 buffer 中，返回读入的数目。若读时，客户终止了连接，则设置 r->connection->keepalive 为-1。

按 chunked 方式取数据，若缓冲区 buffer 太小，则返回-1。

检查已读的数据量是否已超过了允许的大小,是则记录错误信息,并终止与客户的连接，返回-1。

否则判断是否应重新读一个 chunked 块。是则去调用 getline 从客户端读取一行。若读出错或读入的内容不是要表达 chunked 块长度的十六进制数，则终止与客户端的连接，并返回-1。否则将块长取出。

若块长为非 0，则将其记录在 r 的 remaining 域中；反之为 0，表示 chunked 块序列结束，这时若需要对 chunked 结构进行消去，则调用 get_mime_headers 从客户连接中读取作为 ChunkedBody 的 footer 的实体头域序列，并清除请求中的 Transfer-Encoding 头域，增加 Content-Length 头域，最后返回 0。若不必进行消去，则将 r->remaining 置为-1，表示已读到 footer 部分。

若需要将 chunked 结构原封不动地向前传递，则重新在读到的 chunked-size 行尾加上滤去的 CRLF，同时不清除缓冲区。否则将缓冲区中读到的 chunked-size 行清除掉。

若已读到 footer 部分（r->remaining==-1），则将循环把 footer 中的内容读入缓冲区。若读出错，则退出循环，终止与客户的连接，并返回-1。若读到一空行，同样退出循环，并在缓冲区中增加一个空行，且将 r->remaining 置为-2。若是由于缓冲区空间不足也退出循环。最后更新 r 中已读的字节数，并返回本次读的字节数。

若 footer 部分已读完（r->remaining==-2），则将 r->remaining 置为 0，指示不需要在读数据了，然后返回 0。

否则，实际上仍位于读 chunked 块的过程中。读尽可能多的数据到缓冲区中。若读出错，则终止与客户的连接，返回-1。

若已没有可读的数据，则认为到达了当前的 chunked 块的尾。继续读入块尾的 CR 与 LF。若出错则终止连接并退出，返回-1；否则若需要将 chunked 结构原封不动地向前传递，则在行尾加上滤去的 CRLF。

最后更新 r 中已读的字节数，并返回本次读的字节数。

ap_index_of_response 方法

```
API_EXPORT(int) ap_index_of_response(int status)
```

取得状态码 status 对应的全局变量 status_lines 数组中的对应项的下标，返回之。失败时返回 LEVEL_500。

ap_method_number_of 方法

```
API_EXPORT(int) ap_method_number_of(const char *method)
```

本方法对传入的 HTTP 请求方法类型字符串 method 进行判定，返回对应的整数常量。若类型无法识别，则返回 M_INVALID。正常情况下的对应关系为：

```
"HEAD"              M_GET
"GET"               M_GET
"POST"              M_POST
"PUT"               M_PUT
"PATCH"             M_PATCH
"PROPFIND"          M_PROPFIND
"PROPPATCH"         M_PROPPATCH
"DELETE"            M_DELETE
"CONNECT"           M_CONNECT
"COPY"              M_COPY
"MKCOL"             M_MKCOL
"MOVE"              M_MOVE
"OPTIONS"           M_OPTIONS
"TRACE"             M_TRACE
"LOCK"              M_LOCK
"UNLOCK"            M_UNLOCK
```

ap_parse_uri 方法

```
CORE_EXPORT(void) ap_parse_uri(request_rec *r, const char *uri)
```

对由 uri 表示的对应请求 r 的 URL 地址进行解析。

r->unparsed_uri 中存放 uri 的副本。

调用 ap_parse_uri_components 对 uri 进行解析，并将解析结果进行适当处理后，存在 r 的相应域中。

ap_read_request 方法

```
request_rec *ap_read_request(conn_rec *conn)
```

本方法读取由 conn 指示的与客户的连接上传来的请求，构造 request_rec 结构，记录本次请求及其处理的信息，并返回该 request_rec 结构。若返回 NULL，表示读请求出错。

首先根据 conn 中的信息，构造一 request_rec 结构。

调用 read_request_line 读取请求行，并进行必要的解析和记录。

若处理失败：如果是因为请求行过长，则进行日志记录，调用 ap_send_error_response 方法发送给客户端出错响应，并返回 r；如果非是请求行过长，则直接返回 NULL。

若处理正常，则进一步判断当前请求是否为 HTTP 0.9 请求。

若不是，则调用 get_mime_headers 读取各头域。若出错，则发送指示错误的响应，并

记录，返回 r。

若是 HTTP 0.9 请求且只是请求 HEAD，则认为客户发送了非法的 HEAD 请求，调用 ap_send_error_response 返回指示错误的响应，最后用 ap_log_transaction 将此次连接写入日志，返回 r。

然后调用 ap_update_vhost_from_headers 确定当前请求针对的基于名称的虚拟服务器。

若是 HTTP1.1 请求，但没有包含 Host 头域或未包含完整的 URI 地址，则进行出错处理，返回 r。

若有非空的 Expect 头域，且头域值为“100-continue”，则将 r->expecting_100 置为 1。若头域值无法识别，则进行出错处理，发送响应，调用 ap_discard_request_body 方法忽略掉还未处理的请求体，并返回 r。

调用 ap_run_post_read_request 来执行各模块中定义的针对读完请求后阶段的句柄。若返回非 0 的错误代码，则调用 http_request.c 中的 ap_die 方法进行出错响应处理并记入日志，返回 NULL。

最后返回 r。

ap_setup_client_block 方法

```
API_EXPORT(int) ap_setup_client_block(request_rec *r, int read_policy)
```

将请求 r 中与 Transfer-Encoding 和 Content-Length 相关的一些属性值与传入的服务器的期望值进行比较，匹配时返回 OK，否则返回对应的错误码，并记入日志。返回值即是状态码。

ap_send_header_field 方法

```
API_EXPORT_NONSTD(int) ap_send_header_field(request_rec *r, const char *fieldname, const char *fieldval)
```

向 r 指示的与客户端的连接上发送一个由 fieldname 和 fieldval 确定的头域。

ap_send_http_header 方法

```
API_EXPORT(void) ap_send_http_header(request_rec *r)
```

向 r 指示的客户端发送 HTTP 响应行及头域。

首先检查，对应请求是否是 HTTP0.9，若是则进行适当处理后，即返回。

否则，先将表 r->err_headers_out 和表 r->headers_out 进行合并，然后调用 ap_basic_http_header 发送响应行及 Date 和 Server 头域，进而根据 r 中的信息，分别发送 Transfer-Encoding、Content-Type、Content-Encoding、Content-Language、Expires，最后调用 terminate_header 来发送空行结束头域部分，最后返回。

ap_send_http_trace 方法

```
API_EXPORT(int) ap_send_http_trace(request_rec *r)
```

处理对 TRACE 请求的正常响应，而出错时仅将指示错误的状态码返回即退出。其中调用了 ap_send_http_header、ap_rvputs、ap_rputs 等方法完成各种响应信息的应答。

ap_should_client_block 方法

```
API_EXPORT(int) ap_should_client_block(request_rec *r)
```

判断是否还应去读请求 r 的体。

判断进行读操作或还没有读到 chunked 体但剩余的字节数为 0，则返回 0。否则，若客户支持 HTTP1.1 且等待 100（Continue）的应答，则发送 100 响应给客户，并返回 1。

ap_rputs 方法

```
API_EXPORT(int) ap_rputs(const char *str, request_rec *r)
```

将指定字符串写入请求 r 指示的与客户的连接。若客户端断开连接或本处写出错，则返回 EOF；否则成功时，返回写入的字节数。

具体调用 buff.c 中的 ap_bputs 方法完成写入。

ap_rvputs 方法

```
API_EXPORT_NONSTD(int) ap_rvputs(request_rec *r,...)
```

将可变数目的字串写入请求 r 指示的与客户的连接，实际是只写入缓冲区。若客户端断开连接或本处写出错，则返回 EOF；否则成功时，返回写入的字节数。

具体调用 ap_bwrite 方法完成写入。

ap_send_error_response 方法

```
void ap_send_error_response(request_rec *r, int recursive_error)
```

本方法根据请求 r 向客户发送指示错误的响应。

首先使用 ap_send_http_header 来先发送响应行及诸头域。

然后将以响应行为主题以错误代码描述为 H1 级标题的一个 HTML 文件发送给客户端。该文件中还包含了与状态码对应的一些详细的描述。

最后调用 ap_finalize_request_protocol 结束响应发送。

get_mime_headers 方法

```
static void get_mime_headers(request_rec *r)
```

从请求 r 指示的与客户的连接中读取头域，放到 r->headers_in 中。

getline 方法

```
static int getline(char *s, int n, BUFF *in, int fold)
```

从 in 指示的缓冲区中读取一行。若 fold 为 1，则继续读取，不受一行的限制。或者读够 n 个字符为止。读到的字符放到 s 中。返回-1 表示失败，否则返回存到 s 中的字符数。

read_request_line 方法

```
static int read_request_line(request_rec *r)
```

从 r 指示的请求连接中读取请求行。对请求行进行必要的解析和记录。成功返回 1，否则返回 0。

循环调用 getline，直到读到一个空行为止。

将请求行记录在 r->the_request 中，记录当前时间在 r->request_time 中，记录方法在 r->method 中，并将对应的方法的序号放在 r->method_number 中。

若请求方法是 HEAD，则标记 r->header_only 为 1。

调用 ap_parse_uri 对 uri 进行解析，结果记录在 r 中。

检查请求行是否过长，是则将状态码 r->status 赋为 HTTP_REQUEST_URI_TOO_LARGE，指示请求地址过长，最后返回 0。

否则，将检查是否是 HTTP0.9 请求，结果记录在 r->assbackwards 中，并在 r->proto_num 中设置对应的版本号。

最后返回 1。

terminate_header 方法

```
static void terminate_header(BUFF *client)
```

发送代表头域部分结束的空行给客户端。

7.4.4. http_request.c

ap_die 方法

```
API_EXPORT(void) ap_die(int type, request_rec *r)
```

本方法为非正常请求处理程序（如出错、需要内部重定向、请求资源不存在处理等）。其中 type 为错误类型指示。

具体处理为：

如果错误类型 type 为 DONE，则表示已经完成非正常请求处理，则直接调用 ap_finalize_request_protocol 方法结束当前请求处理并返回。

然后设置当前请求的状态码 r->status 为 type，并对其进行判定。

若是 AUTH_REQUIRED 且当前请求为代理服务器请求，则设置 r->status 为

HTTP_PROXY_AUTHENTICATION_REQUIRED。

如果连接仍然有效并且 r->status **不是** HTTP_NOT_MODIFIED**（因为若状态码为** HTTP_NOT_MODIFIED**，则请求必然为有条件的** GET **请求，是没有请求体的）或** HTTP_NO_CONTENT**，并且也不是需要断开连接的状态，则调用** ap_discard_request_body **方法忽略请求体。**

如果自定义的状态码字符串信息不为空，则：

- **如果是一个** URI**，则设置状态码为** REDIRECT**（重定向），在** r->headers_out **中设置响应头域** Location**，其值即为上述** URI**；**
- **如果自定义的响应信息是一个本地路径，则设置** r->no_local_copy = 1**，并通过在** r->subprocess_env **中设置**"REQUEST_METHOD"、"ERROR_NOTES"**请求环境变量的值，记录当前请求信息，然后设置请求方法为** GET**，调用** ap_internal_redirect **进行内部重定向，最后返回；**
- **如果不是前两种情况，则认为设置的重定向地址信息有误，设置** recursive_error = SERVER_ERROR**（重定向递归），记录错误信息。**

最后调用 ap_send_error_response **方法发送出错响应，返回。**

ap_process_request方法

```
void ap_process_request(request_rec *r)
```

本方法完成对请求 r **的处理。具体是先调用** process_request_internal **进行处理，然后调用** ap_log_transaction **进行日志，且在开始和结束时均用** ap_update_child_status **方法更新当前工作线程的状态信息。**

decl_die 方法

```
static void decl_die(int status, char *phase, request_rec *r)
```

本方法类似于 ap_die **方法，也是非正常请求处理程序。不同之处是，本方法的参数中可给出一个字符串，描述出错的处理阶段。具体的处理实际是调用** ap_die **完成的。**

directory_walk 方法

```
static int directory_walk(request_rec *r)
```

功能：检查 Directory **一级的目录配置向量，将匹配的目录配置向量合并到当前请求的目录配置向量** r->per_dir_config **中。**

具体过程如下：

如果 r->filename **不是绝对路径，则在服务器的所有目录配置向量中查找匹配的目录配置向量，并且将其合并到** r->per_dir_config **中，然后返回** OK**（匹配算法与** location_walk **相似）。**

调用 ap_os_case_canonical_filename**、**ap_os_canonical_filename **方法规范化** r->filename**。**

调用 ap_os_is_filename_valid **方法检查** r->filename **文件名是否含有非法字符或系统保留** base_path**。是则记录错误，并返回** HTTP_FORBIDDEN**。**

调用 check_safe_file **方法进行文件安全性检查。若未通过，即要访问的文件禁止访问，**

则记录错误并返回。

然后，循环对请求地址的每一级目录进行如下处理：调用 util.c 中的 ap_make_dirstr_prefix 方法取地址中至当前目录级别的路径前缀；将当前服务器目录配置向量中与当前路径匹配的目录配置向量合并到当前请求的目录配置向量中；同时若对应目录允许.htaccess 文件覆盖先前配置，则调用 ap_parse_htaccess 方法对.htaccess 文件进行分析，并将配置信息合并到当前请求的目录配置向量中。

检查服务器的特殊目录配置向量（参阅 core 模块的 reorder_sorter 方法），也将匹配的目录配置向量合并到当前请求的目录配置向量中。

最后返回 OK。

file_walk 方法

```
static int file_walk(request_rec *r)
```

本方法检查 Files 一级的目录配置向量，将匹配的目录配置向量合并到当前请求的目录配置向量 r->per_dir_config 中。

location_walk 方法

```
static int location_walk(request_rec *r)
```

本方法检查配置文件中定义的 Location 配置信息，每个 Location 命令均对应一个配置环境，找出所有匹配的配置环境将其合并，并将结果存到当前请求的目录配置向量（r->per_dir_config）中。

具体处理如下：

若当前服务器没有定义 URL 一级的目录配置向量，则返回 OK；否则依次检查当前服务器的每一个 URL 一级的目录配置向量，若与当前请求地址匹配，则将该 URL 目录配置向量合并到当前请求的目录配置向量（r->per_dir_config）中。

匹配的大致过程如下：

如果当前 URL 一级目录配置向量的 URI（entry_url）是一个扩展规则表达式，则调用 regexec 进行匹配；否则

如果 entry_url 是含有通配符的 URI（entry_core->d_is_fnmatch），则调用 ap_fnmatch 进行匹配；否则

检查 entry_url 是否为当前请求地址的上级目录。

process_request_internal 方法

```
static void process_request_internal(request_rec *r)
```

本方法具体负责完成对请求 r 的各步处理。

处理过程如下：

首先若请求非是 proxy 请求且请求 URI 非空，则用 ap_unescape_url 消去转义字符。如果处理出错，则调用 ap_die 方法给出出错响应，并直接返回。

然后用 ap_getparents 对 r->uri 进行化简，消去冗余。

随后依次调用 location_walk、ap_translate_name 等方法分别进行匹配的 Location 配置环

境的合并及请求地址的必要转换。

若当前请求非是 proxy 但为 TRACE 请求，则调用 ap_send_http_trace 方法和 ap_finalize_request_protocol 方法返回响应，并返回；遇到错误时，调用 ap_die 方法。

继续依次调用 directory_walk、file_walk、location_walk、ap_header_parse。若遇到问题则，调用 ap_die 或 decl_die 完成错误响应，返回。需要说明的是，此处已经是第二次调用 location_walk 了，由于前面的 directory_walk 调用会清除掉已设置的 per_dir_config 域，所以需要重新处理。ap_header_parse 方法是进行请求真正处理前的头域分析，但本版本 Apache 中没有定义任何操作。

调用 ap_satisfies，得到对用户进行访问控制和身份认证的策略，区别情况，调用 ap_check_access、ap_auth_type、ap_some_auth_required、ap_check_user_id、ap_check_auth 来完成处理。

继续依次进行：非基于 HTTP 的 proxy 请求时的 ap_find_types、ap_run_fixups、ap_invoke_handler。若遇到问题则调用 ap_die 或 decl_die 完成错误响应，返回。

最后调用ap_finalize_request_protocol方法结束对请求的处理。

7.4.5. http_config.c

ap_add_loaded_module 方法

```
API_EXPORT(void) ap_add_loaded_module(module *mod)
```

本方法实现指定模块 mod 的动态加载及激活，其中调用了 ap_add_module 方法实现激活。

本方法在 so 模块中处理 LoadModule 配置命令时被调用。

ap_add_module 方法

```
API_EXPORT(void) ap_add_module(module *m)
```

将指定的模块激活。

实际上，本方法仅是将模块配置块 m 加入 top_module 指向的链表中。

本方法要检查模块 m 的版本号是否与主程序一致，否则报错并退出执行。

若传入的模块是动态加载的，则判断该类模块的总数目是否超出限制，是则报错并退出执行。

ap_check_access 方法

```
int ap_check_access(request_rec *r)
```

执行权限确认阶段的所有处理方法，有 mod_access 模块的 check_dir_access 方法。

ap_add_named_module 方法

```
API_EXPORT(int) ap_add_named_module(const char *name)
```

本方法将已装入即已存在于 ap_loaded_modules 数组中的具有指定名称 name 的模块激活。成功时，返回 1；否则返回 0。

ap_check_auth 方法

```
int ap_check_auth(request_rec *r)
```

执行身份认证阶段的所有处理方法，有 mod_auth 模块的 check_user_access 方法、mod_digest 模块的 digest_check_auth 方法、mod_anon_auth 模块的 check_anon_access 方法、mod_auth_db 模块的 db_check_auth 方法、mod_auth_dbm 模块的 dbm_check_auth 方法。

ap_check_user_id 方法

```
int ap_check_user_id(request_rec *r)
```

执行身份确认阶段的所有处理方法，有 mod_auth 模块的 authenticate_basic_user 方法、mod_digest 模块的 authenticate_digest_user 方法、mod_anon_auth 模块的 anon_authenticate_basic_user 方法、mod_auth_db 模块的 db_authenticate_basic_user 方法、mod_auth_dbm 模块的 db_authenticate_basic_user 方法。

ap_child_init_modules 方法

```
void ap_child_init_modules(pool *p, server_rec *s)
```

本方法对各个模块进行初始化，与 ap_init_modules 方法不同的是，此处是去执行每个模块的 child_init 方法。本方法在子进程体 worker_main 方法中被调用。

请参阅本章第 3 节。

ap_clear_module_list 方法

```
API_EXPORT(void) ap_clear_module_list()
```

本方法实现将 top_module 链表清除，在 core 模块的处理 ClearModuleList 配置命令时被调用。

ap_create_per_dir_config 方法

```
CORE_EXPORT(void *) ap_create_per_dir_config(pool *p)
```

本方法调用 create_empty_config 方法创建一空的针对目录级别的配置信息结构指针向量，并返回之。

需要说明的是，这里并没有进一步创建各模块的配置信息块，而是仅仅为它们的指针分配空间。实际上，在 ap_handle_command 方法中处理具体配置命令时会对此一点进行检查，并在发现配置信息块不存在时创建之。

ap_find_command 方法

```
CORE_EXPORT(const command_rec *) ap_find_command(const char *name, const command_rec *cmds)
```

本方法判断某模块是否可处理配置命令 name。

具体是在配置命令列表 command_rec 结构类型参数 cmds 中查找是否有配置命令 name。若有说明此 command_rec 结构对应模块可处理该配置命令，否则说明不能处理。前者返回 cmds，后者返回空。

ap_find_command_in_modules 方法

```
CORE_EXPORT(const command_rec *) ap_find_command_in_modules(const char *cmd_name, module **mod)
```

在以 mod 为头指针的模块链表中查找可处理配置命令 cmd_name 的模块。

此方法对 mod 链表进行遍历，调用 ap_find_command 方法判断某个模块是否可处理配置命令 cmd_name。若找到一个这样的模块，则用 mod 保存其指针，并返回其对应的 command_rec 结构；否则返回空。

ap_find_linked_module 方法

```
API_EXPORT(module *) ap_find_linked_module(const char *name)
```

本方法判定是否有名为 name 的已激活模块存在于 Apache 中。有则返回该模块的接口控制块指针，否则返回 NULL。本方法在 http_core.c 中的 start_ifmod 方法被调用。

ap_find_types 方法

```
int ap_find_types(request_rec *r)
```

执行 MIME 类型识别阶段的所有处理方法，有 mod_mime 模块的 find_ct 方法、mod_mime_magic 模块的 magic_find_ct 方法、mod_negotiation 模块的 handle_multi 方法。

ap_handle_command 方法

```
CORE_EXPORT(const char *) ap_handle_command(cmd_parms *parms, void *config, const char *l)
```

本方法对参数 l 中存放的配置命令行进行处理。

具体过程如下：

如果命令行的第一个字符是'#"，则认为是注释行，返回 NULL；否则调用 ap_getword_conf 方法，取配置命令名 cmd_name。

下面进入命令处理循环：

调用 ap_find_command_in_modules 方法查找处理 cmd_name 命令的模块（mod）及其 command_rec 结构的指针(cmd)。若失败返回错误信息字符串，否则继续；

调用 ap_get_module_config 方法获取 mod 的当前目录配置结构（mconfig）及服务器配置结构（sconfig）；

如果 mconfig 没有定义并且 mod 定义了 per dir 配置创建函数，则创建该模块的当前目录配置记录，并将其地址保存在当前目录的配置向量中；

如果 sconfig 没有定义并且 mod 定义了 per server 配置创建函数，则创建该模块的服务器配置记录，并将其地址保存在服务器配置向量（module_config）中（实际上 sconfig 并未在后面处理配置信息中用到）；

执行命令（invoke_cmd），如果命令返回值为 NULL 或不是 DECLINE_CMD,则退出循环；否则继续在其他模块中查找 cmd_name 的处理函数。

ap_header_parse 方法

```
int ap_header_parse(request_rec *r)
```

执行读请求后各阶段定义用来分析请求头域的所有处理方法,但目前没有模块定义该类方法。

ap_init_modules 方法

```
void ap_init_modules(pool *p, server_rec *s)
```

本方法对各个模块进行初始化（即实际的模块激活过程）。

具体过程包括三个步骤：

执行top_module模块链表中每个模块的init初始化方法；调用build_method_shortcuts方法初始化所有模块的快捷请求处理方法结构；调用init_handlers方法初始化所有模块的响应处理方法紧凑结构。

ap_init_vhost_config 方法

```
void ap_init_vhost_config(pool *p)
```

初始化虚拟主机的配置结构。

ap_init_virtual_host方法

```
CORE_EXPORT(const char *) ap_init_virtual_host(pool *p, const char *hostname,
server_rec *main_server, server_rec **ps)
```

创建针对虚拟主机的 server_rec 结构并进行初始化，该结构的指针放入 ps 中传出去。hostname 为虚拟主机名，main_server 则对应主（基本）服务器。

其中 server_rec 结构的 is_virtual 域设置为 1，表示服务器为虚拟主机。

最后调用 ap_parse_vhost_addrs 方法对虚拟主机地址进行解析，并返回解析结果。成功时为 NULL，否则为错误信息串。

ap_log_transaction 方法

```
int ap_log_transaction(request_rec *r)
```

执行日志记录阶段的所有处理方法，有 mod_log_referer 模块的 referer_log_transaction 方法、mod_log_config 模块的 multi_log_transaction 方法、mod_log_agent 模块的 agent_log_transaction 方法。

ap_process_resource_config 方法

```
void ap_process_resource_config(server_rec *s, char *fname, pool *p, pool
*ptemp)
```

对 fname 指示的配置文件进行处理。

具体处理过程为：

若 fname 为相对路径，则调用 ap_server_root_relative 方法得到绝对路径。

若 fname 是 RESOURCE_CONFIG_FILE 或 ACCESS_CONFIG_FILE，且 fname 所指定的文件不存在，则直接返回；否则继续。

若 fname 是 SERVER_CONFIG_FILE，且定义了命令行配置（存放该配置的 ap_server_pre_read_config 和 ap_server_post_read_config 非空），则若指定文件不存在，则直接返回。

初始化 parms 结构，其将作为参数传递到 ap_srm_command_loop 方法中，其中包含要处理的配置文件名、使用的资源池及配置信息存放的位置（server 域）等，参阅本章第 3 节。

打开配置文件时若失败，则提示打开文件出错信息，退出系统。

调用 ap_srm_command_loop 方法，具体处理配置文件。如果配置文件处理错误，则提示错误信息（包含出错位置等）并退出系统。

最后关闭配置文件，并返回。

ap_merge_per_dir_configs 方法

```
void * ap_merge_per_dir_configs(pool *p, void *base, void *new)
```

与上述 merge_server_configs 方法类似，但本方法是合并两个针对目录的配置信息块向量，并返回之。

ap_read_config 方法

```
server_rec *ap_read_config(pool *p, pool *ptemp, char *confname)
```

本方法对配置文件进行处理。

首先调用 init_server_config 方法，初始化 server_rec 结构。然后初始化配置方面的全局

变量。

然后调用 ap_process_resource_config 方法和 process_command_config 方法继续对配置文件、读文件前的配置及读文件后的配置进行处理。

处理的结果是将各种配置信息记录到对应的数据结构中，在处理网络请求时使用。

最后调用 fixup_virtual_hosts、default_listeners、ap_fini_vhost_config 完成对虚拟主机信息的补充和调整及初始化 ap_listeners 链表准备进行监听请求。

ap_remove_module 方法

```
API_EXPORT(void) ap_remove_module(module *m)
```

本方法的处理与 ap_add_module 方法完全相反。

ap_remove_loaded_module 方法

```
API_EXPORT(void) ap_remove_loaded_module(module *mod)
```

本方法的处理与 ap_add_loaded_module 方法完全相反。

ap_run_fixups 方法

```
int ap_run_fixups(request_rec *r)
```

执行补充处理阶段的所有处理方法，有 mod_negotiation 模块的 fix_encoding 方法。

ap_run_post_read_request 方法

```
int ap_run_post_read_request(request_rec *r)
```

执行各模块定义的在读取请求后时刻进行的所有处理方法，有 mod_setenvif 模块的 match_headers 方法、mod_unique_id 模块的 gen_unique_id 方法。

ap_setup_prelinked_modules 方法

```
void ap_setup_prelinked_modules()
```

本方法将 Apache 的缺省模块装入并激活。

本方法首先根据 ap_preloaded_modules 数组中定义的各个 module 结构指针，对 ap_loaded_modules 数组进行初始化，完成装入。

然后调用 ap_add_module 方法激活 ap_prelinked_modules 数组中定义的各个模块。

详细介绍请参阅第 6 章及本章第 3 节。

ap_srm_command_loop 方法

```
API_EXPORT(const char *) ap_srm_command_loop(cmd_parms *parms, void *config)
```

对 parms-> config_file 指示的配置文件中定义的命令进行处理。

循环读取文件中的每一行，用 ap_handle_command 方法对其进行处理。若出错，则返回出错信息；否则返回 NULL。

ap_translate_name 方法

```
int ap_translate_name(request_rec *r)
```

执行地址转换阶段的所有处理方法，有 mod_core 模块的 core_translate 方法、mod_proxy 模块的 proxy_trans 方法。

create_default_per_dir_config 方法

```
static void *create_default_per_dir_config(pool *p)
```

与上述 create_server_config 方法类似，但本方法是创建一个记录各模块针对目录的配置信息块的向量，并返回之。

create_empty_config 方法

```
static void *create_empty_config(pool *p)
```

本方法从缓冲池 p 中申请空间，创建一个存放各模块配置信息结构指针的向量，并返回之。

create_server_config 方法

```
static void *create_server_config(pool *p, server_rec *s)
```

本方法创建一个记录各模块针对服务器的配置信息块的向量，并返回之。

对于 Apache 中的每一个模块，若其接口控制块的 create_server_config 域不为空，表示该模块定义了创建服务器级配置信息块的方法，继续调用该方法，返回的信息块指针记录到向量中。

fixup_virtual_hosts 方法

```
static void fixup_virtual_hosts(pool *p, server_rec *main_server)
```

本方法使用主服务器的配置值对各个虚拟主机的配置块进行补充，在 ap_read_config 方法中被调用。

init_config_globals 方法

```
static void init_config_globals(pool *p)
```

对配置方面的全局变量及虚拟主机方面进行初始化。

init_server_config 方法

```
static server_rec *init_server_config(pool *p)
```

创建 server_rec **结构，并进行初始化。**

特别需要提及的是，server_rec **结构的** module_config **域为一个向量，存放各模块针对服务器（即对整个服务器而言）的配置信息块，调用** create_server_config **方法进行初始化；** lookup_defaults **域也是一个向量，存放各模块针对目录（即对目录级别有意义）的配置信息块，调用** create_default_per_dir_config **方法进行初始化。请参阅本章第** 3 **节。**

invoke_cmd 方法

```
static const char *invoke_cmd(const command_rec *cmd, cmd_parms *parms, void
*mconfig, const char *args)
```

本方法对一条配置命令进行处理，正常时返回 NULL **或** DECLINE_CMD**，表示已正确处理该命令或不能处理该命令；否则返回错误信息字符串，指示配置命令有误。其中** cmd **为先前调用** ap_find_command_in_modules **方法查找到的处理该配置命令的模块中定义的相应** command_rec **结构；**parms **为** cmd_parms **结构指针，存放当前处理配置文件的辅助信息；** mconfig **为要记录配置信息的各模块相应结构的向量；**args **则存放除配置命令名外的参数串部分。**

具体处理过程为：

若当前处理环境不允许定义此配置命令，即(parms->override & cmd->req_override)== 0 **为** TRUE**，则返回错误信息串退出；**

然后根据配置参数的格式（cmd->args_how**），区别情况分别进行处理，大致的过程均是将参数先一个个解析出，然后调用** cmd **中给出的处理方法进行处理。**

merge_server_configs 方法

```
static void merge_server_configs(pool *p, void *base, void *virt)
```

对两个针对服务器的配置信息块向量 base **和** virt **进行合并，结果存于后者中。**

顾名思义，前者为基准参考，后者为针对某虚拟主机的定制。若某个配置信息块 virt **中没有定义，则取** base **中的值；否则，调用对应模块的** merge_server_config **方法对两个信息块进行合并，结果存于** virt **中。**

process_command_config 方法

```
static void process_command_config(server_rec *s, array_header *arr, pool *p, pool *ptemp)
```

本方法对运行时以命令行参数形式加入的配置信息文件进行处理。文件名在数组 arr **中存放。**

全局变量 ap_server_pre_read_config **中存放要在处理正常配置文件前处理的配置信息文件名；**ap_server_post_read_config **中则存放处理正常配置文件后在处理的文件名。**

其中调用 ap_srm_command_loop **方法对文件中的命令进行处理。**

7.4.6. http_log.c

ap_log_error 方法

```
API_EXPORT(void) ap_log_error(const char *file, int line, int level, const server_rec *s, const char *fmt, ...)
```

本方法与 ap_log_rerror **方法一道，为被外界使用进行错误日志的主方法。具体实现调用** log_error_core **方法完成。**

ap_log_pid 方法

```
void ap_log_pid(pool *p, char *fname)
```

本方法将当前 Apache **主服务器的进程** id **写入以** fname **为名的日志文件中，此信息可用于基于进程间通信的** Apache **的停止或重启控制。请参阅第** 14 **节事件控制。**

ap_log_rerror 方法

```
API_EXPORT(void) ap_log_rerror(const char *file, int line, int level, const request_rec *r, const char *fmt, ...)
```

本方法除了具有 ap_log_error **方法的全部功能外，还增加如下功能：**

当错误级别是或高于"warning"**时，并且没有与此错误相关的解释性文字，则将此处传入的文本内容也设置为** r->notes **的**"error-notes"**值，从而在** Apache **的后续处理中可以被** ErrorDocument **或其它一些错误处理器利用。如果要使此功能无效，可以在调用此函数之前，往**"error-notes"**中填入一些字符串（甚至空串）。**

ap_open_logs 方法

```
void ap_open_logs(server_rec *s_main, pool *p)
```

本方法打开错误日志。具体是调用 open_error_log 方法打开错误日志文件，然后用该文件流替代标准错误输出流 stderr。

log_error_core 方法

```
static void log_error_core(const char *file, int line, int level, const server_rec *s, const request_rec *r, const char *fmt, va_list args)
```

此方法为进行错误日志的核心方法，被 ap_log_error 和 ap_log_rerror 方法调用。

若当前在进行标准错误日志 stderr logging （启动阶段），则不记录级别高于默认服务器日志级别（DEFAULT_LOGLEVEL）的消息，除非是关于启动／关闭的 notice 级别信息；若当前在进行正常错误日志，则不记录级别高于当前服务器级别（s->loglevel）日志的消息，除非是关于启动／关闭的 notice 级别信息；若当前在进行系统日志，则不记录级别高于当前服务器日志级别的消息，包括关于启动／关闭的 notice 级别信息。

在本方法中，使用了 ap_snprintf 方法来生成日志文本信息。

open_error_log 方法

```
static void open_error_log(server_rec *s, pool *p)
```

本方法打开错误日志文件。

具体处理如下：

若错误日志是送给另外一个日志处理程序处理的，则调用 ap_spawn_child 方法创建相应子进程，设定将日志信息作为该进程的标准输入；否则表示将日志信息写入普通文件，则调用 openlog 方法打开对应文件。

7.4.7. http_vhost.c

ap_fini_vhost_config 方法

```
void ap_fini_vhost_config(pool *p, server_rec *main_s)
```

本方法对关于虚拟服务器的信息进行整理和组织，如虚拟主机域名及地址等，仅在 ap_read_config 中读完配置文件之后被调用。

ap_init_vhost_config 方法

```
void ap_init_vhost_config(pool *p)
```

本方法在读取配置前的http_config.c的init_config_globals方法中被调用，完成与虚拟服务器相关的若干全局变量的初始化。

ap_matches_request_vhost方法

```
ap_matches_request_vhost(request_rec *r, const char *host, unsigned port)
```

本方法判定指定的主机名和端口号是否与接收请求 r 的服务器的名字及端口匹配。是则，返回 1；否则，返回 0。其中对名字的检查，包括了对服务器的所有名字（或虚拟主机名）的检查。

ap_parse_vhost_addrs 方法

```
const char *ap_parse_vhost_addrs(pool *p, const char *hostname, server_rec
*s)
```

本方法对在<VirtualHost>中定义的虚拟主机名 hostname 进行解析，得到的一系列 server_addr_rec 结构地址放到 s->addrs 为头指针的链表中。

若成功，返回 NULL；否则，返回其中调用的核心方法 get_addresses 的返回值——错误描述串。

ap_set_name_virtual_host方法

```
ap_set_name_virtual_host (cmd_parms *cmd, void *dummy, char *arg)
```

本方法实现根据 arg 表示的主机名和缺省的端口号 cmd->server->port 调用 get_addresses 获得与定义的名称虚拟服务器所基于的 IP 地址对应的所有底层地址。

是对应 NameVirtualHost 配置命令的处理句柄。

ap_update_vhost_from_headers 方法

```
void ap_update_vhost_from_headers(request_rec *r)
```

本方法根据可能存在的请求 r 中的 Host 头域信息，确定对应的基于名称的虚拟服务器。

具体处理如下：

首先调用 fix_hostname 将 r->hostname 中可能包含的端口号去掉，只留下主机名。

然后，若 r->hostname 非空，则调用 check_hostalias，在 r->connection 中记录找到的与主机名匹配的虚拟服务器。否则，调用 check_serverpath，在 r->connection 中记录找到的其 ServerPath 与请求 r 的 URL 中的路径匹配的虚拟服务器。

最后返回。

本方法只被 http_protocol.c 中的 ap_read_request 方法调用。由于 new_connection 与 ap_read_request 的先后次序，使得 ap_update_vhost_given_ip 和 ap_update_vhost_from_headers 先后被调用，从而实现在确定针对特定 IP 的服务器后，进一步确定对应的基于名称的虚拟服务器。

ap_update_vhost_given_ip 方法

```
ap_update_vhost_given_ip(conn_rec *conn)
```

本方法根据 conn 中地址信息，确定“处理”请求的虚拟服务器。

实际上，处理仍是由服务器的当前进程完成的，只不过所用的配置信息是这里找到的虚拟服务器对应的配置信息。

首先调用 find_ipaddr，根据连接 conn 中本地的 IP 地址信息，确定匹配的 IP 哈希表项。若找到，则将对应的 server_rec 和基于名字的虚拟服务器链表分别放入 conn 中的 server 和 vhost_lookup_data 域，覆盖掉原来的值。然后返回。

若没找到，则将 vhost_lookup_data 域置为 NULL，并调用 find_default_server 来查找一缺省的服务器，若能查到，则放到 conn->server 中，否则退出。

本方法只被 http_main.c 中的 new_connection 方法调用。

check_hostalias 方法

```
check_hostalias(request_rec *r)
```

本方法查找其名称或别名及端口号与 r 中的主机名和本地端口号相匹配的虚拟服务器。

其中使用 matches_aliases 完成针对别名的匹配检查。

若找到，则将其放到 r->server 及 r->connection->server 中。

find_ipaddr 方法

```
find_ipaddr(struct in_addr *server_ip, unsigned port)
```

本方法根据给出的地址和端口号确定 IP 哈希表中的对应虚拟服务器。找到时返回哈系表项，否则返回 NULL。

哈希表中的项是 ipaddr_chain 结构类型。该结构的 sar 域记录对应的服务器的地址信息。

其中使用的哈希算法是 hash_inaddr。对应同一哈希结果的源在表中连续存放。

find_default_server 方法

```
find_default_server(unsigned port)
```

本方法找到一个与指定端口匹配的缺省的服务器。

在 default_list 表中查找能与端口 port 匹配的名为“_default_”的配置中定义的缺省的服务器。

找到则返回对应的 ipaddr_chain 项，否则返回 NULL。

fix_hostname 方法

```
fix_hostname(request_rec *r)
```

将 r->hostname 中可能包含的端口号去掉，只留下主机名。

get_addresses 方法

```
static const char *get_addresses(pool *p, char *w, server_addr_rec ***paddr,
unsigned port)
```

本方法取得地址由 w **指定的虚拟服务器（主机）的所有** hostent **结构类型的地址，存在** paddr **间接指向的链表中，**port **为服务器缺省的端口号。**

w **为“** hostname:port number **”形式，可以是“** *:* **”、“** _default_ **”、“** 255.255.255.255 **”等** IP **数字形式，也可以是域名形式。若是前者，则调用** ap_inet_addr **方法（在** Windows **下实际为** inet_addr **调用）处理，否则调用** gethostbyname **方法（为** Windows **的系统调用）处理，最后都得到** in_addr **结构的地址表示，放到** paddr **指向的位置，若有多个地址，可以形成一个链表。**

处理出错时，返回非空描述串，否则返回 NULL**。**

hash_inaddr 方法

```
hash_inaddr(unsigned key)
```

本方法是一个冲突可能极小的哈希函数。返回一个在哈希表范围 IPHASH_TABLE_SIZE **内的值。**

matches_aliases 方法

```
matches_aliases(server_rec *s, const char *host)
```

本方法检查 host **指定的主机名是否与当前服务器的** ServerName **及** ServerAlias **值匹配。**

首先检查主机名 host **是否是服务器** s **对应的名字，是则返回** 1**；否则看是否是** s **的不带通配符的别名，是则返回** 1**；否则，看是否与** s **的带通配符的别名匹配，是则返回** 1**；否则返回** 0**。**

第5节　core 模块

core **模块实现了** WWW **服务器的最基本功能，在没有其它任何模块的情况下，整个服务器也可基本正常工作。本模块在** http_core.c **中实现。**

7.5.1. mod_core 的数据结构

core_dir_config 结构

core **模块的针对目录级别的配置结构为：**

```
typedef struct {
```

```
    char *d;
    unsigned d_components;
    allow_options_t opts;
    allow_options_t opts_add;
    allow_options_t opts_remove;
    overrides_t override;
    char *ap_default_type;
    int satisfy;
    char *ap_auth_type;
    char *ap_auth_name;
    array_header *ap_requires;
    char **response_code_strings;
    unsigned int hostname_lookups : 4;
    signed int do_rfc1413 : 2;
    signed int content_md5 : 2;
    unsigned use_canonical_name : 2;
    unsigned d_is_fnmatch : 1;
    unsigned long limit_req_body;
    enum { srv_sig_off, srv_sig_on, srv_sig_withmail } server_signature;
    int loglevel;
    array_header *sec;
    regex_t *r;
    interpreter_source_e script_interpreter_source;
} core_dir_config;
```

其中 d **中存放当前** core_dir_config **结构针对的目录路径；**d_components **为** d **中的目录级数；**d_is_fnmatch **为一个** bit**，存放判定** d **是否含有通配符的结果，由于该判定经常要用到，所以专门定义此位来存放结果 若** d **为正则表达式，则在** r **中存放相应的表达式信息；**opts **与** override **分别存放针对当前的** Options **和** AllowOverride **配置信息，控制服务器的特征在某一目录上的继承，**opts_add **与** opts_remove **则用来记录** Options **配置命令中使用**"+|-"**增加和删除的功能特征；**ap_default_type **用来存放服务器返回的响应的默认的** MIME **类型；**response_code_strings **用来存放自定义的状态码字符串；**satisfy**、**ap_auth_type **、**ap_auth_name **、**ap_requires **中分别存放对应当前配置环境的与身份认证相关的** Satisfy**、**AuthType**、**AuthName**、**require **配置信息；**hostname_lookups **、**do_rfc1413**、**content_md5 **、**use_canonical_name **等均为长短有异的** bit **位，分别记录** HostNameLookups**、**IdentityCheck**、**ContentDigest**、**UseCanonicalName **等配置信息，分别表示进行** DNS **查询的方式、是否按**RFC1413 **协议确定客户名、是否对响应进行** MD5 **编码及构造自引用** URL **地址的方式；**limit_req_body **记录** LimitRequestBody **配置信息；**loglevel **记录日志级别；**sec **为一数组，每个单元均对应一个** Files **配置定义，存放相应的配置向量；**script_interpreter_source **中存放**ScriptInterpreterSource **配置信息。**

core_server_config 结构

如下为针对服务器级别的配置结构：

```
typedef struct {
```

```
    char *ap_document_root;

    char *access_name;
    array_header *sec;
    array_header *sec_url;
} core_server_config;
```

其中 ap_document_root **存放** DocumentRoot **配置信息，即本服务器对应的页面树根目录路径；**access_name、sec、sec_url **与访问控制有关，分别表示** acess **文件路径、各** Directory **配置片段针对的地址、各** Location **配置片段针对的地址。**

7.5.2. mod_core 中的处理方法

处理方法可分为如下几组：

- **与初始化相关的方法有：**

```
create_core_dir_config
merge_core_dir_configs
create_core_server_config
merge_core_server_configs
```

- **与配置命令处理相关的方法有：**

```
ap_add_per_dir_conf
ap_add_per_url_conf
add_file_conf
reorder_sorter
ap_core_reorder_directories
ap_limit_section
endlimit_section
dirsection
filesection
urlsection
start_ifmod
end_ifmod
ap_exists_config_define
start_ifdefine
end_ifdefine
virtualhost_section
end_nested_section
require
satisfy
set_access_name
set_document_root
set_error_document
set_override
set_options
set_server_alias
```

```
add_module_command
clear_module_list_command
set_server_string_slot
server_type
server_port
set_user
set_server_root
set_timeout
set_keep_alive_timeout
set_keep_alive
set_keep_alive_max
set_pidfile
set_serverpath
set_content_md5
set_use_canonical_name
set_server_limit
set_max_requests
set_threads
set_bind_address
set_listener
include_config
set_loglevel
set_authname
set_limit_req_line
set_interpreter_source
```

- **与请求处理有关的方法有：**

```
ap_document_root
ap_requires
ap_satisfies
ap_response_code_string
ap_get_remote_host
ap_get_server_name
ap_get_server_port
ap_construct_url
core_translate
do_nothing
default_handler
```

- **与身份认证相关的处理方法有：**

```
ap_auth_type
ap_auth_name
ap_default_type
ap_note_auth_failure
ap_note_basic_auth_failure
ap_note_digest_auth_failure
```

```
ap_get_basic_auth_pw
```

各方法具体实现如下：

ap_add_per_dir_conf方法

```
CORE_EXPORT(void) ap_add_per_dir_conf(server_rec *s, void *dir_config)
```

本方法被 dirsection **方法调用，将** Directory **或** DirectoryMatch **配置片段针对的本地地址** dir_config **加入到针对当前服务器环境** s **的** core **模块配置结构** core_server_config **的** sec **数组中。**

ap_add_per_url_conf方法

```
CORE_EXPORT(void) ap_add_per_url_conf(server_rec *s, void *url_config)
```

本方法被 urlsection **方法调用，将** Location **或** LocationMatch **配置片段针对的** url **地址** url_config **加入到针对当前服务器环境** s **的** core **模块配置结构** core_server_config **的** sec_url **数组中。**

ap_allow_options 方法

```
API_EXPORT(int) ap_allow_options(request_rec *r)
```

本方法返回当前 core_dir_config **结构的** opts **域中存放的** Options **配置信息。**

ap_allow_overrides 方法

```
API_EXPORT(int) ap_allow_overrides(request_rec *r)
```

本方法返回 request_rec->per_dir_config **中** core **模块配置中的** override **成员,该信息来自当前请求对应的目录的相应配置中的** AllowOverride **命令。**

ap_auth_type 方法

```
API_EXPORT(const char *) ap_auth_type(request_rec *r)
```

本方法返回 request_rec->per_dir_config **中** core **模块配置中的** ap_auth_type **成员，该信息来自当前请求对应的目录的相应配置中的** AuthType **命令。**

ap_auth_name 方法

```
API_EXPORT(const char *) ap_auth_name(request_rec *r)
```

本方法返回 request_rec->per_dir_config **中** core **模块配置中的** ap_auth_name **成员，该信息来自当前请求对应的目录的相应配置中的** AuthName **命令。**

ap_check_cmd_context 方法

```
API_EXPORT(const char *) ap_check_cmd_context(cmd_parms *cmd, unsigned
forbidden)
```

本方法检查 cmd **指示的上下文是否符合由** forbidden **指示的限定条件。符合则返回** NULL**，否则返回非空错误信息串。**

其中 forbidden **取如下在** http_config.h **中定义的** bit **值的组合：**

```
#define  NOT_IN_VIRTUALHOST     0x01 /* <Virtualhost> */
#define  NOT_IN_LIMIT          0x02 /* <Limit> */
#define  NOT_IN_DIRECTORY       0x04 /* <Directory> */
#define  NOT_IN_LOCATION        0x08 /* <Location> */
#define  NOT_IN_FILES          0x10 /* <Files> */
#define  NOT_IN_DIR_LOC_FILE  (  NOT_IN_DIRECTORY  |  NOT_IN_LOCATION  |
NOT_IN_FILES )
#define    GLOBAL_ONLY    (    NOT_IN_VIRTUALHOST    |    NOT_IN_LIMIT    |
NOT_IN_DIR_LOC_FILE )
```

ap_construct_url 方法

```
API_EXPORT(char *) ap_construct_url(pool *p, const char *uri, const
request_rec *r)
```

本方法根据传入的路径 uri **及当前请求** r **构造包含协议名、域名及端口在内的完全** URI **地址，并返回之。**

ap_core_reorder_directories 方法

```
void ap_core_reorder_directories(pool *p, server_rec *s)
```

本方法对 s **中** core **模块配置结构** module_config **中的** sec **数组进行升序排列，排序算法使用** qsort **调用，具体的比较逻辑则由** reorder_sorter **方法给出。**

排序的思路为 按照目录的深度对 s->module_config **向量中** core **模块服务器配置中** sec **数据项保存的目录配置记录进行排序，**1-component **在前，然后是** 2-component**，...，最后是**"special" sections**。**"special" sections **是指正则表达式，或不以/开始的入口。**

ap_custom_response 方法

```
API_EXPORT(void) ap_custom_response(request_rec *r, int status, char *string)
```

本方法在当前 core_dir_config **结构的** response_code_strings **数组中设置状态码** status **的定制状态码。**

实际此方法并未在 Apache **中用到。**

ap_default_type 方法

```
API_EXPORT(const char *) ap_default_type(request_rec *r)
```

本方法返回 request_rec->per_dir_config **中** core **模块配置中的** ap_default_type **成员，如果该成员没有定义，则返回** DEFAULT_CONTENT_TYPE **（实际为**"text/plain"**）。当无法确定被请求资源的类型时，使用** DEFAULT_CONTENT_TYPE **类型作为缺省的** Content-Type **响应头域取值。**

ap_document_root 方法

```
API_EXPORT(const char *) ap_document_root(request_rec *r)
```

本方法返回当前 core_server_config **结构的** ap_document_root **域中存放的服务器页面根目录地址。**

ap_exists_config_define 方法

```
API_EXPORT(int) ap_exists_config_define(char *name)
```

本方法判定 Apache **启动时是否通过“** -D **”参数定义了名为** name **的 参数。是则返回** 1**，否则返回** 0**。**

add_file_conf 方法

```
static void add_file_conf(core_dir_config *conf, void *url_config)
```

本方法被 filesection **方法调用，将** File **或** FilesMatch **配置片段针对的地址** url_config **加入到当前目录环境的** core **模块配置结构** core_dir_config **的** sec **数组中。**

ap_get_basic_auth_pw

```
API_EXPORT(int) ap_get_basic_auth_pw(request_rec *r, const char **pw)
```

本方法检查认证类型是否为"Basic"**，从请求头域中抽取用户名、密码信息。**

处理过程如下：

如果服务器认证类型（AuthType**）不是基本认证，则返回** DECLINED**；**

如果没有设置认证名称（AuthName**），则登记错误信息，返回** SERVER_ERROR**；**

如果当前请求中没有 "Proxy-Authorization" **或** "Authorization" **头域，则调用** ap_note_basic_auth_failure **方法设置要求用户认证的头域信息，返回** AUTH_REQUIRED**；**

如果客户端使用的认证机制不是"Basic"**，则记录错误信息，返回** AUTH_REQUIRED**；**

将请求头域中的用户名、密码部分解码（调用 ap_uudecode **方法）；将用户名、认证类型信息记入** r->connection **中对应数据域；将密码送入** pw**，从而传递出去。**

最终返回 OK**。**

ap_get_remote_host方法

```
API_EXPORT(const char *) ap_get_remote_host(conn_rec *conn, void *dir_config,
int type)
```

本方法返回当前连接 conn **对应的远程主机地址。其中** type **给出远程主机名查找方式，在** http_core.h **中定义。**

具体过程如下：

首先从当前 core_dir_config **结构的** hostname_lookups **域中取** HostnameLookups **配置设置信息。**

然后根据 type **及** HostnameLookups **配置设置进行判定。若要求查找，则根据远程端口地址查找远程主机名（**gethostbyaddr**）。如果查找成功，则将主机名存到** conn->remote_host **中，而且如果** HostnameLookups **设置为双向查找，则调用** do_double_reverse **方法进行反向查找，失败时置** conn->remote_host **为** NULL**。**

如果 type **为** REMOTE_DOUBLE_REV**，则进行反向查找，失败时返回** NULL**。**

最后如果远程主机名不为空，则返回该名；否则根据 type**，返回远程端口地址或** NULL**。**

ap_get_server_name 方法

```
API_EXPORT(const char *) ap_get_server_name(const request_rec *r)
```

本方法返回当前服务器名称。

在确定服务器名称时，有两种选择：一种是规范名称，它是由 ServerName**、**Port **配置信息定义的；另一种是从请求中的** Host **头域或完全** URI **地址获得。**Apache **永远假设请求头域中传送的端口值信息不可靠，而是使用真正接收请求的** socket **端口值。**

如果当前配置定义为使用规范名称，则返回 r->server->server_hostname**；否则返回如果请求中定义了** hostname**，则返回它，否则仍然返回** r->server->server_hostname**。**

ap_get_server_port方法

```
API_EXPORT(unsigned) ap_get_server_port(const request_rec *r)
```

本方法返回当前服务器的端口号。请参阅 ap_get_server_name **方法。**

ap_limit_section 方法

```
CORE_EXPORT_NONSTD(const char *) ap_limit_section(cmd_parms *cmd, void *dummy,
const char *arg)
```

本方法处理 Limit **和** LimitExcept **配置片段首行，将指定的受限制请求方法信息存放在** cmd->limited **中，这样在随后处理配置信息时可以很方便地获取当前此类信息。**

add_module_command 方法

```
static const char *add_module_command(cmd_parms *cmd, void *dummy, char *arg)
```

本方法处理 AddModule **配置命令，实现将已装入的模块激活。**

ap_note_auth_failure 方法

```
API_EXPORT(void) ap_note_auth_failure(request_rec *r)
```

本方法设置提示用户提供认证信息的响应头域。

若服务器的认证方法是"Basic"**，则调用** ap_note_basic_auth_failure **通知进行基本认证；而若认证方法是**"Digest"**，则调用** ap_note_digest_auth_failure **通知进行摘要认证。**

ap_note_basic_auth_failure 方法

```
API_EXPORT(void) ap_note_basic_auth_failure(request_rec *r)
```

在本方法中，如果服务器认证方法是"Basic"**，则设置**"Proxy-Authenticate"**或**"WWW-Authenticate"**响应头域，通知客户需要进行代理服务器或原始服务器基本认证；否则调用** ap_note_auth_failure **通知进行其他认证（如摘要认证）。**

ap_note_digest_auth_failure 方法

```
API_EXPORT(void) ap_note_digest_auth_failure(request_rec *r)
```

本方法设置 Proxy-Authenticate **或** WWW-Authenticate **响应头域，通知客户需要进行代理服务器或** WWW **摘要认证。**

ap_requires 方法

```
API_EXPORT(const array_header *) ap_requires(request_rec *r)
```

本方法返回当前 core_dir_config **结构的** ap_requires **域中存放的** require **认证信息。**

ap_response_code_string 方法

```
char *ap_response_code_string(request_rec *r, int error_index)
```

本方法返回当前 core_dir_config **结构的** response_code_strings **数组中存放的对应出错状态码** error_index **的提示字符串。**

ap_satisfies 方法

```
API_EXPORT(int) ap_satisfies(request_rec *r)
```

本方法返回当前 core_dir_config 结构的 satisfy 域中存放的 satisfy 认证信息。参阅 satisfy 方法。

clear_module_list_command 方法

```
static const char *clear_module_list_command(cmd_parms *cmd, void *dummy)
```

本方法处理 ClearModuleList 配置命令，实现将已装入并激活的模块清除，具体调用 http_config.c 中的 ap_clear_module_list 方法实现。

core_translate 方法

```
static int core_translate(request_rec *r)
```

本方法完成最基本的请求地址的转换。根据原始请求地址 r->uri、服务器所在路径 r->server_path 及 core 模块配置中的页面目录树根路径 ap_document_root 等信息计算得到 r->filename。

具体处理为：

如果使用代理服务器，则返回 HTTP_FORBIDDEN。

如果 r->uri 不以'/'打头且不等于"*"，则返回 BAD_REQUEST。

如果定义了 r->server->path 并且 r->uri 所指定的资源肯定在 r->server->path 目录下，则将 r->filename 赋值为 core 模块定义的 ap_document_root + r->uri 中除去 r->server->path 的剩余部分；否则将 r->filename 直接赋为 ap_document_root + r->uri。

最后返回 OK。

create_core_dir_config 方法

```
static void *create_core_dir_config(pool *a, char *dir)
```

本方法初始化 core 模块的目录配置信息块，返回其指针。

create_core_server_config 方法

```
static void *create_core_server_config(pool *a, server_rec *s)
```

本方法初始化 core 模块的服务器配置信息块，返回其指针。

default_handler 方法

```
static int default_handler(request_rec *r)
```

本方法为在没有定义其他请求处理句柄的情况下可匹配任何 MIME 类型请求的缺省处理方法。目前只处理 GET、OPTION 命令。

处理过程如下：

首先调用 ap_discard_request_body 方法略去可能存在的请求体，并设置此处理方法允许

处理的请求方法类型为 GET 和 OPTIONS。

然后，对请求 r 中记录的信息进行检查并执行响应处理，包括：

1. 若请求方法类型为 M_INVALID 即非法的请求方法，则进行错误日志记录，并返回 NOT_IMPLEMENTED；

2. 若请求方法为 OPTIONS，则调用 ap_send_http_options 方法处理请求，并返回其返回值；

3. 如果客户请求为 PUT，则返回 METHOD_NOT_ALLOWED；

4. 如果客户请求的文件不存在，则返回 HTTP_NOT_FOUND；

5. 如果客户请求不是 GET，则返回 METHOD_NOT_ALLOWED；

6. 打开客户请求的文件，如果失败，则返回 FORBIDDEN；

7. 设置响应头域 Last-Modified（最后修改时间）的值；

8. 设置响应头域 Accept-Ranges 的值为"bytes";

9. 设置"Content-Length"响应头域的值（内容长度）

10. 如果 content_md5 末位非 0，即使用 MD5 方式返回请求，则设置响应头域 Content-MD5，其值由 ap_md5digest 方法计算得到；

11. 调用 ap_set_byterange 方法处理 Range 或 Request-Range 请求头域；

12. 调用 ap_send_http_header 方法发送响应头信息；

13. 若请求非是 HEAD 类型，且指定只发送文件的某个片段，则调用 ap_send_fd_length 方法发送指定区间，否则调用 ap_send_fd 方法发送整个被请求文件内容做为响应体。

14. 调用 ap_pfclose 方法关闭打开的被请求文件。

最后返回 OK。

dirsection 方法

```
static const char *dirsection(cmd_parms *cmd, void *dummy, const char *arg)
```

本方法处理 Directory 或 DirectoryMatch 配置片段。

具体处理过程如下：

调用 ap_create_per_dir_config 方法创建一个新的针对当前片段指定目录的配置信息向量。

检查当前配置是否在合法的上下文中出现，否则返回出错信息串。

检查是否缺少闭括号“ > ”，是则返回出错信息。

将当前片段指定目录路径存于 cmd->path 中。

继续调用 ap_srm_command_loop 方法对片段内定义的配置命令进行处理。正常情况下，遇到片段结束标识时，由于 core 模块声明可以处理该命令的方法 end_nested_section 总是返回一个非空串，所以 ap_srm_command_loop 方法会认为出错从而返回，返回值即为无法处理的结束标识。

对返回的标识进行判定，看是否是期望的字符串。否，则直接退出，并返回上述标识。

调用 ap_add_per_dir_conf 方法将上面创建的配置向量加入到当前服务器环境 cmd->server 中的 core 模块配置结构中。

将配置片段中给出的路径表达式存于配置向量的 core 模块配置结构的 r 域中。

最后返回 NULL。

do_nothing 方法

```
static int do_nothing(request_rec *r) { return OK; }
```

目前是一个空方法，不进行任何操作。

end_ifdefine 方法

```
static const char *end_ifdefine(cmd_parms *cmd, void *dummy)
```

本方法处理可有可无的</IfDefine>配置命令，与 end_ifmod 方法类似。

end_ifmod 方法

```
static const char *end_ifmod(cmd_parms *cmd, void *dummy)
```

本方法处理可有可无的</IfModule>配置命令，实际直接返回 NULL。

由于 Apache 中处理 IfModule 配置片段与上述 Directory、Location 及 Files 片段不同，前者并没有调用 ap_srm_command_loop 方法进行结束标识无法处理（从而可以退出 ap_srm_command_loop）的嵌套片段的处理，而是直接用 ap_cfg_getline 方法读取配置文件中的一行，通过比较是否为“ </IfModule> ”来确定结束。请参阅 start_ifmod 方法。

end_nested_section 方法

```
static const char *end_nested_section(cmd_parms *cmd, void *dummy)
```

本方法处理 Directory、DirectoryMatch、Location、LocationMatch、Files、FilesMatch、VirtualHost 等配置片段结束标识。正常时返回所期望的正确结束标识，否则返回出错信息串。

本方法与众多配置命令处理方法不同，即使在正常时也不返回 NULL，实际上，正是这一点使调用的处理配置片段的 ap_srm_command_loop 方法错认为已出错而恰好退出。

endlimit_section 方法

```
static const char *endlimit_section(cmd_parms *cmd, void *dummy, void *dummy2)
```

本方法处理 Limit 和 LimitExcept 配置片段结尾，将 cmd->limited 中存放的先前的受限制请求方法信息清空（置为-1）。

filesection 方法

```
static const char *filesection(cmd_parms *cmd, c ore_dir_config *c, const char
*arg)
```

本方法处理 Files 或 FilesMatch 配置片段。

具体处理过程与上述 dirsection 方法基本相同。

include_config 方法

```
static const char *include_config (cmd_parms *cmd, void *dummy, char *name)
```

本方法处理 Include **配置命令，进一步调用** ap_process_resource_config **方法对指明要包括的配置文件进行处理。**

merge_core_dir_configs 方法

```
static void *merge_core_dir_configs(pool *a, void *basev, void *newv)
```

本方法将两个 core **模块的目录配置信息块合并，返回合并结果指针。**

merge_core_server_configs 方法

```
static void *merge_core_server_configs(pool *p, void *basev, void *virtv)
```

本方法将两个 core **模块的服务器配置信息块合并，返回合并结果指针。**

missing_endsection 方法

```
static const char *missing_endsection(cmd_parms *cmd, int nest)
```

当处理配置片段还未结束但已读到文件尾时，调用本方法返回一个出错信息串。

reorder_sorter 方法

static int reorder_sorter(const void *va, const void *vb)

本方法实现 va **和** vb **之间的比较，被** qsort **方法调用。当** va> vb **时，返回** 1**；当** va= vb **时，返回** 0**；当** va< vb **时，返回**-1**。**

参数 va **和** vb **均为** reorder_sort_rec **结构类型：**

```
struct reorder_sort_rec {
    void *elt;
    int orig_index;
};
```

其中 elt **为对应的** core **模块配置信息块，为** core_dir_config **结构类型，而** orig_index **则为排序前的原始下标。**

若用 core_a **和** core_b **分别表示** va **和** vb **中的** core_dir_config **结构，则具体比较过程为：**

如果 core_a **是特殊的，**core_b **不是特殊的，**

返回 1**；**

如果 core_a **不是特殊的，**core_b **是特殊的，**

返回-1**；**

否则两者都不是特殊的或都是特殊的：

如果 core_a->d_components > core_b->d_components，
即 core_a->d 路径中的目录级数大于 core_b->d 的级数，
返回 1；
如果 core_a->d_components < core_b->d_components，
返回-1；
否则，
返回 a->orig_index - b->orig_index，即保持两者在表中原来的先后次序。

其中，如果一个 core_dir_config 结构中的 r 不为空或者 d 不是以“ /: ”开头，则认为其是 “ 特殊的 ”。

require 方法

```
static const char *require(cmd_parms *cmd, core_dir_config *c, char *arg)
```

本方法处理 Satisfy 配置命令，将配置值存放到新创建的 require_line 结构中，添加到当前 core_dir_config 结构的 ap_requires 数组中。

require_line 结构定义如下：

```
typedef struct {
    int method_mask;
    char *requirement;
} require_line;
```

其中 requirement 存放 arg 表示的需要进行身份认证的用户或用户组列表；method_mask 则以 bit 形式存放 cmd->limited 记录的当前配置信息处理上下文中受限制的请求方法。

satisfy 方法

```
static const char *satisfy(cmd_parms *cmd, core_dir_config *c, char *arg)
```

本方法处理 Satisfy 配置命令，并将配置值存放到当前 core_dir_config 结构的 satisfy 域中。

server_type 方法

```
static const char *server_type(cmd_parms *cmd, void *dummy, char *arg)
```

本方法处理 ServerType 配置命令，并将信息记录在全局变量 ap_standalone 中。实际在 Windows 平台下，此配置无效。

server_port 方法

```
static const char *server_port(cmd_parms *cmd, void *dummy, char *arg)
```

本方法处理 Port 配置命令，将合法的端口号记录在 cmd->server->port 中。

set_access_name 方法

```
static const char *set_access_name(cmd_parms *cmd, void *dummy, char *arg)
```

本方法处理 AccessFileName **配置命令，并将指定路径存放到当前** core_server_config **结构的** access_name **域中。**

set_authname 方法

```
static const char *set_authname(cmd_parms *cmd, void *mconfig, char *word1)
```

本方法处理 AuthName **配置命令，并将配置值存放到当前** core_dir_config **结构的** ap_auth_name **域中。**

set_bind_address 方法

```
static const char *set_bind_address(cmd_parms *cmd, void *dummy, char *arg)
```

本方法处理 BindAddress **配置命令，并将配置值存放到全局变量** ap_bind_address.s_addr **中。**

set_content_md5 方法

```
static const char *set_content_md5(cmd_parms *cmd, core_dir_config *d, int
arg)
```

本方法处理 ContentDigest **配置命令，并将配置值存放到当前** core_dir_config **结构的** content_md5 **域中。**

set_document_root 方法

```
static const char *set_document_root(cmd_parms *cmd, void *dummy, char *arg)
```

本方法处理 DocumentRoot **配置命令，并将指定路径存放到当前** core_server_config **结构的** ap_document_root **域中。**

set_error_document 方法

```
static const char *set_error_document(cmd_parms *cmd, core_dir_config *conf,
char *line)
```

本方法处理 ErrorDocument **配置命令，并将定制的出错提示信息存放到当前** core_dir_config **结构的** response_code_strings **数组的相应单元中。**

set_interpreter_source 方法

```
static const char *set_interpreter_source(cmd_parms *cmd, core_dir_config
*d, char *arg)
```

本方法处理 ScriptInterpreterSource **配置命令，并将配置的确定脚本解释器的方法信息存放到当前** core_dir_config **结构的** script_interpreter_source **域中。**

set_keep_alive_timeout方法

```
static const char *set_keep_alive_timeout(cmd_parms *cmd, void *dummy, char
*arg)
```

本方法处理 KeepAliveTimeout **配置命令，并将配置值存放到** cmd->server->keep_alive_timeout **中。**

set_keep_alive 方法

```
static const char *set_keep_alive(cmd_parms *cmd, void *dummy, char *arg)
```

本方法处理 KeepAlive **配置命令，并将配置值存放到** cmd->server->keep_alive **中。**

set_keep_alive_max 方法

```
static const char *set_keep_alive_max(cmd_parms *cmd, void *dummy, char *arg)
```

本方法处理 MaxKeepAliveRequests **配置命令，并将配置值存放到** cmd->server->keep_alive_max **中。**

set_limit_req_line 方法

```
static const char *set_limit_req_line(cmd_parms *cmd, void *dummy, char *arg)
```

本方法处理 LimitRequestLine **配置命令，并将配置值存放到** cmd->server->limit_req_line **中。**

set_listener 方法

```
static const char *set_listener(cmd_parms *cmd, void *dummy, char *ips)
```

本方法处理 Listen **配置命令，将配置值存放到新创建的** listen_rec **结构中，并放到全局变量** ap_listeners **链表的头部。**

set_loglevel 方法

```
static const char *set_loglevel(cmd_parms *cmd, void *dummy, const char *arg)
```

本方法处理 LogLevel **配置命令，并将配置值存放到** cmd->server->loglevel **中。**

set_max_requests 方法

```
static const char *set_max_requests(cmd_parms *cmd, void *dummy, char *arg)
```

本方法处理 MaxRequestsPerChild **配置命令，并将配置值存放到全局变量** ap_max_requests_per_child **中。**

set_override 方法

```
static const char *set_override(cmd_parms *cmd, core_dir_config *d, const
char *l)
```

本方法处理 AllowOverride **配置命令，并将配置值存放到当前** core_dir_config **结构的** override **域中。**

set_options 方法

```
static const char *set_options(cmd_parms *cmd, core_dir_config *d, const char
*l)
```

本方法处理 Options **配置命令，并将配置值存放到当前** core_dir_config **结构的** opts、opts_add、opts_remove **等域中。**

set_pidfile 方法

```
static const char *set_pidfile(cmd_parms *cmd, void *dummy, char *arg)
```

本方法处理 PidFile **配置命令，并将配置值存放到全局变量** ap_pid_fname **中。**

set_server_alias 方法

```
static const char *set_server_alias(cmd_parms *cmd, void *dummy, const char
*arg)
```

本方法处理 ServerAlias **命令，将参数** arg **中的虚拟服务器的别名存入对应** server_rec **结构的** wild_names**（当** arg **含有通配符时）或** names**（不含通配符）域中。若出现的上下文非法，则返回错误信息串；否则返回** NULL**。**

set_server_string_slot 方法

```
static const char *set_server_string_slot(cmd_parms *cmd, void *dummy, char
*arg)
```

本方法实现字符串形式的简单配置信息在 cmd->server **结构中的记录。负责处理的配置命令有：**ServerAdmin、ServerName、ErrorLog、AccessConfig、ResourceConfig。

set_server_limit 方法

```
static const char *set_server_limit (cmd_parms *cmd, void *dummy, char *arg)
```

本方法处理 MaxClients **配置命令，并将配置值存放到全局变量** ap_daemons_limit **中。**

set_server_root 方法

```
static const char *set_server_root(cmd_parms *cmd, void *dummy, char *arg)
```

本方法处理 ServerRoot **配置命令，并将指定路径存放到全局变量** ap_server_root **中。**

set_serverpath 方法

```
static const char *set_serverpath(cmd_parms *cmd, void *dummy, char *arg)
```

本方法处理 ServerPath **配置命令，并将配置值存放到** cmd->server->path **中，**cmd->server->pathlen **则存放其长度。**

set_threads 方法

```
static const char *set_threads(cmd_parms *cmd, void *dummy, char *arg)
```

本方法处理 ThreadsPerChild **配置命令,并将配置值存放到全局变量** ap_threads_per_child **中。**

set_timeout 方法

```
static const char *set_timeout(cmd_parms *cmd, void *dummy, char *arg)
```

本方法处理 Timeout **配置命令，并将配置值存放到** cmd->server->timeout **中。**

set_use_canonical_name 方法

```
static const char *set_use_canonical_name(cmd_parms *cmd, core_dir_config
*d,  int arg)
```

本方法处理 UseCanonicalName 配置命令，并将配置值存放到当前 core_dir_config 结构的 use_canonical_name 域中。

set_user 方法

```
static const char *set_user(cmd_parms *cmd, void *dummy, char *arg)
```

本方法处理 User 配置命令，在 Windows 平台下无效，参阅第 5 章第 2 节User 配置命令。

start_ifmod 方法

```
static const char *start_ifmod(cmd_parms *cmd, void *dummy, char *arg)
```

本方法处理 IfModule 配置片段。

首先通过调用 ap_find_linked_module 方法判定指定模块是否在 Apache 中，从而确定 IfModule 条件是否成立。是则直接返回 NULL。否则持续读配置文件，且并不处理读取的内容，直至遇到“ </IfModule> ”。

若在遇到“ </IfModule> ”前文件读完，则返回错误信息串；否则返回 NULL，即正常跳过不应处理的配置片段。

start_ifdefine 方法

```
static const char *start_ifdefine(cmd_parms *cmd, void *dummy, char *arg)
```

本方法处理 IfDefine 配置片段，除调用 ap_exists_config_define 方法来判定条件是否成立外，与 start_ifmod 方法处理相同。

unclosed_directive 方法

```
static char *unclosed_directive(cmd_parms *cmd)
```

本方法返回指示在指定配置命令中“ 缺少闭括号 ”的错误信息串，被所有处理配置命令片段的方法调用。

urlsection 方法

```
static const char *urlsection(cmd_parms *cmd, void *dummy, const char *arg)
```

本方法处理 Location 和 LocationMatch 配置片段。

具体处理过程与上述 dirsection 方法基本相同。

virtualhost_section 方法

```
static const char *virtualhost_section(cmd_parms *cmd, void *dummy, char *arg)
```

本方法处理 VirtualHost 配置命令片段。

具体过程如下：

检查当前配置是否在合法的上下文中出现，否则返回出错信息串。

调用 ap_init_virtual_host 创建并初始化一 server_rec 结构存放当前虚拟服务器的配置信息。若该服务器给定地址解析失败，则直接返回错误信息；

将虚拟服务器配置结构加入到以 main_serser->next 为头指针的服务器链表的头部；

调用 ap_srm_command_loop 方法处理 VirtualHost 配置片段；

若定义了此虚拟服务器特有的 srm 和 access 配置文件，则调用 ap_process_resource_config 方法分别处理之；

最后，若片段正常结束，返回 NULL，否则返回错误信息串。

第6节 proxy 模块

本模块为 Apache 服务器实现了 proxy/cache（代理/缓冲）功能。它支持 FTP、SSL CONNECT、HTTP/0.9 和 HTTP/1.0。本模块还可以被配置成连接到其它的 proxy 模块，即支持代理服务器的级联。

7.6.1. mod_proxy 的实现

mod_proxy 的实现过程可以简单描述如下（实际上是各个模块的共性）：

- 模块的装入——连进链表——配置信息块初始化；
- 自身初始化；
- 配置命令处理——配置信息记录——配置信息块填充；
- 处理请求。

在 mod_proxy 模块处理请求过程中，几个重要的分处各个阶段的处理方法，包括：

- proxy_detect：
  检查是否是 proxy 请求，是则设置 request_rec.proxyreq 为 1。
- proxy_trans：
  将请求 r 中可能包含的虚拟本地路径替换成实际远程服务器的地址。
- proxy_fixup：
  对 proxy 请求 r 中的 URL 进行标准化或修正。
- proxy_handler：
  对 proxy 请求进行处理。

进一步地，在 proxy_handler 方法中，区别不同的情况分别用 ap_proxy_connect_handler 方法、ap_proxy_http_handler 方法和 ap_proxy_ftp_handler 方法来完成基于管道、HTTP、FTP 的 proxy 请求处理。具体的处理过程为：

1. 若 r-> method_number 指示请求为 TRACE 方法，则附加上本服务器的信息，若转发次数已超过最大限值，则直接发响应给客户，并返回。
2. 看在缓冲中是否有用户要求的资源，有则将其发给客户，并返回。
3. 检查请求的 URL 是否对应配置中定义的某级联代理服务器，是则记录相应信息。
4. 下面分三种情况：

- 若是 CONNECT 请求，则是 SSL（Secure Socket Layer）请求，则建立一条从客户到远程服务器或级联代理服务器的隧道(参阅第 9 章HTTP1.1 协议中的基本概念部分)，对传输的内容不进行任何处理，直接转发。
- 若是 HTTP 请求，则完成如下过程：
  检查目的主机是否是被禁止访问的，是则返回 HTTP_FORBIDDEN；
  否则与远程主机建立连接 ;前传客户请求,及可能涉及由配置控制的 Via 头域；
  发送请求体给远程主机；
  接收、前传响应并更新整理缓冲。首先，通过 ap_proxy_cache_update 对缓冲进行更新，同时发送响应。然后，用 ap_proxy_cache_tidy 清除缓冲中的临时文件。最后，用 ap_proxy_garbage_coll 对缓冲进行垃圾收集。
- 若是 FTP 请求,则调用 proxy_ftp.c 文件中的 ap_proxy_ftp_handler 方法处理之。由于篇幅所限，面向 FTP 的代理机制未被纳入本书，读者可参考本书相关部分对其进行分析。

在上述过程中涉及的各个方法按功能不同，可分为以下若干类：

- 配置信息初始化方法。包含如下的函数，完成初始化及把从配置文件中读取的与 proxy 有关的数据记入当前调用服务器的 server_rec 结构中的 proxy 模块配置信息块（proxy_server_conf）中：

```
create_proxy_config
set_proxy_req
add_proxy
add_pass
add_pass_reverse
set_proxy_exclude
set_recv_buffer_size
set_proxy_dirconn
set_proxy_domain
set_allowed_ports
set_cache_root
set_cache_size
set_cache_maxex
set_cache_defex
set_cache_factor
set_cache_gcint
set_cache_dirlevels
set_cache_dirlength
set_cache_exclude
set_recv_buffer_size
set_cache_completion
set_via_opt
proxy_init
```

- 网络请求解析、响应处理、字符串处理、时间格式转换、网络地址验证等支持方法：

```
ap_proxy_cache_error
ap_proxyerror
ap_proxy_doconnect
```

```
ap_proxy_bputs2
ap_proxy_send_hdr_line
ap_proxy_send_fb
ap_proxy_send_headers
proxy_getline
proxy_get_host_of_request
ap_proxy_read_headers
ap_proxy_c2hex
ap_proxy_hex2c
ap_proxy_liststr
ap_proxy_hash
ap_proxy_sec2hex
ap_proxy_hex2sec
ap_proxy_date_canon
ap_proxy_canonenc
ap_proxy_canon_netloc
proxy_match_ipaddr
proxy_match_domainname
proxy_match_hostname
proxy_match_word
ap_proxy_is_ipaddr
ap_proxy_is_domainname
ap_proxy_is_hostname
ap_proxy_is_word
ap_proxy_host2addr
```

- **与缓冲区相关的处理方法（对应** proxy_cache.c **文件）：**

```
ap_proxy_garbage_init
ap_proxy_garbage_coll
add_long61
sub_long61
cmp_long61
gcdiff
sub_garbage_coll
help_proxy_garbage_coll
rdcache
ap_proxy_cache_update
ap_proxy_cache_tidy
ap_proxy_cache_check
```

- **与网络连接相关的处理方法（对应** proxy_connect.c **文件）：**

```
allowed_port
ap_proxy_connect_handler
```

- **与** HTTP **代理请求相关的处理方法（对应** proxy_http.c **文件）：**

```
ap_proxy_http_canon
clear_connection
```

```
proxy_location_reverse_map
ap_proxy_http_handler
```

- **客户请求处理方法：**

```
alias_match
proxy_detect
proxy_trans
proxy_fixup
proxy_needsdomain
proxy_handler
```

下面对各个方法进行详细的讲解：

create_proxy_config 方法

```
create_proxy_config(pool *p, server_rec *s)
```

初始化本 proxy **模块的配置信息块，返回其指针。**

set_proxy_req 方法

```
set_proxy_req(cmd_parms *parms, void *dummy, int flag)
```

记录由 ProxyRequests **定义的是否接受** proxy **请求的开关变量的值。将** flag **中的标志赋到** proxy_server_conf->req **中。**

add_proxy 方法

```
add_proxy (cmd_parms *cmd, void *dummy, char *f, char *r)
```

增加一个在指定请求模式下用来传递请求的远程级联代理服务器，由 ProxyRemote **定义。参数** f **表示** ProxyRemote **设置中的** URL **模式，**r **表示远程服务器的定义。本方法将** r **中的协议、主机地址、端口号以及** f **表示的** URL **模式记录在** proxy_server_conf.proxy **的一个单元中。若端口没有显式的指定，则根据** r **中的协议到预先定义好的映射表** defports **中去查找缺省值代替。**

add_pass 方法

```
add_pass (cmd_parms *cmd, void *dummy, char *f, char *r)
```

增加一条虚拟本地路径来映像指向远程的服务器的一个 URL**，用** ProxyPass **来定义。将参数** f **表示的虚拟本地地址和用** r **表示的远程地址填入到** proxy_server_conf.aliases **中的一个单元中。**

add_pass_reverse 方法

```
add_pass_reverse(cmd_parms *cmd, void *dummy, char *f, char *r)
```

与 add_pass 相反，指示访问指定的远程服务器的请求将被重定向到指定的本地路径。需注意 add_pass 与 add_pass_reverse 关联的配置项并不完全相逆，因为可能还要受到可由 RewriteRule 命令定义的 rewrite（地址重写）操作的影响。由于篇幅有限，本书未深入涉及 Rewrite 部分，读者若感兴趣可参考 Apache 软件所附文档。将参数 f 表示的虚拟远程 URL 地址和用 r 表示的本地地址填入到 proxy_server_conf.raliases 中的一个单元中。

set_proxy_exclude 方法

```
set_proxy_exclude(cmd_parms *parms, void *dummy, char *arg)
```

记录定义的一系列禁止使用本代理服务来访问的目的主机或域。Apache 会将各个主机的对应 IP 地址查找到，以便在客户用 IP 地址访问而定义的项是 DNS 地址时避免出现漏洞，用 ProxyBlock 定义。参数 arg 中存放要记录的主机名（IP 地址或域名）。其中，对已记录的地址将不再处理；对得到的地址信息和传入的名字将放入 proxy_server_conf.noproxies 的一个单元中。

set_recv_buffer_size 方法

```
set_recv_buffer_size(cmd_parms *parms, void *dummy, char *arg)
```

记录定义的向外部进行 http 和 ftp 连接时用来存放数据的缓冲区的大小，用 ProxyReceiveBufferSize 定义。将参数 arg 指向的字串中的数字解析出，放到 proxy_server_conf.recv_buffer_size 中。

set_proxy_dirconn 方法

```
set_proxy_dirconn (cmd_parms *parms, void *dummy, char *arg)
```

登记那些被访问时代理服务器直接连接，而不再通过可能存在的远程级联代理服务器的目的主机或域，用 NoProxy 定义。参数 arg 中存放定义的地址，其中对可能的重复进行了检查，然后仅是简单地将 arg 中的内容放入了 proxy_server_conf.dirconn 数组中新单元的 name 域中。

set_proxy_domain 方法

```
set_proxy_domain(cmd_parms *parms, void *dummy, char *arg)
```

记录指定的代理服务器所在域的地址，当请求的目的地址中没有指明完整的地址而只有主机名时，将把此域地址附加到请求的主机地址上，并产生一个重定向响应（redirection response），用 ProxyDomain 定义。参数 arg 为传入的域名，其中要求域名字符串必须以“.”开头。

set_allowed_ports 方法

```
set_allowed_ports (cmd_parms *parms, void *dummy, char *arg)
```

记录了由 allowCONNECT 指定的一系列 CONNECT 连接可以使用的端口。参数 arg 为一表示数字的字符串，解析出的端口号将放入 proxy_server_conf.allowed_connect_ports 数组中。

set_cache_root 方法

```
set_cache_root(cmd_parms *parms, void *dummy, char *arg)
```

记录由 CacheRoot 指定的用于缓冲区的目录的路径。若不定义 CacheRoot，则代理服务器将在无缓冲的情况下进行服务。将参数 arg 中的路径存到 proxy_server_conf. cache_conf.root 中。

set_cache_size 方法

```
set_cache_size(cmd_parms *parms, char *struct_ptr, char *arg)
```

记录由 CacheSize 指定的缓冲区可以占用的最大空间，以 kb 为单位。将参数 arg 指向的字串中的数字解析出，放到 proxy_server_conf. cache_conf.space 中。

set_cache_maxex 方法

```
set_cache_maxex (cmd_parms *parms, void *dummy, char *arg)
```

记录在不访问原始服务器的情况下文档在缓冲区中可以存放的最长时间，由 CacheMaxExpire 指定。该设置将不受由文档来源指定的过期时间的影响。将解析参数 arg 得到的小时数换算成秒数存在 proxy_server_conf. cache_conf.maxexpire 中。

set_cache_defex 方法

```
set_cache_defex (cmd_parms *parms, void *dummy, char *arg)
```

记录由 CacheDefaultExpire 指定的文档过期的时间。这一设置只在取文档时的协议不支持过期时才有效。它不被 CacheMaxExpire 设置所覆盖。将解析参数 arg 得到的小时数换算成秒数存在 proxy_server_conf. cache_conf.defaultexpire 中。

set_cache_factor 方法

```
set_cache_factor (cmd_parms *parms, void *dummy, char *arg)
```

记录由 CacheLastModifiedFactor 指定的过期因子，用来在原始服务器没有指定某文档的过期时间时，计算一个过期时间，公式是：

```
expiry-period = time-since-last-modification * <factor>
```

将参数 arg 字串进行解析存到 proxy_server_conf. cache_conf. lmfactor。

set_cache_gcint方法

```
set_cache_gcint(cmd_parms *parms, void *dummy, char *arg)
```

记录由 CacheGcInterval 指定的垃圾收集间隔。当不指定此设置时，cache 将无限增大。将解析参数 arg 得到的小时数换算成秒数存在 proxy_server_conf. cache_conf. gcinterval 中。

set_cache_dirlevels 方法

```
set_cache_dirlevels(cmd_parms *parms, char *struct_ptr, char *arg)
```

记录由 CacheDirLevels 指定的缓冲区中子目录层数。将参数 arg 字串中的数字解析出，检查它的合法性，并放到 proxy_server_conf. cache_conf.dirlevels 中。数字的大小限制与所用的 Hash 函数的结果值的位数有关，参阅本章第 9 节MD5 算法的相关部分。

set_cache_dirlength 方法

```
set_cache_dirlength(cmd_parms *parms, char *struct_ptr, char *arg)
```

记录由 CacheDirLength 指定的缓冲区中子目录名的长度。将 arg 字串中的数字解析出，检查它的合法性，并放到 proxy_server_conf. cache_conf.dirlength 中。数字的大小限制与所用的 Hash 函数的结果值的位数有关，因为路径名是从 Hash 函数的结果中取得的。

set_cache_exclude 方法

```
set_cache_exclude (cmd_parms *parms, void *dummy, char *arg)
```

记录由 NoCache 指定的一系列域或主机地址，对来自指定地址的文档将不进行缓冲。经过重复性检查之后，将由 arg 字串表示的机器名及转化得到的 hostent 结构信息存入 proxy_server_conf.nocaches 数组中。

set_recv_buffer_size 方法

```
set_recv_buffer_size(cmd_parms *parms, void *dummy, char *arg)
```

记录由 ProxyReceiveBufferSize 指定的缓冲区大小。将 arg 字串中的数字解析出，检查它的合法性，并放到 proxy_server_conf. cache_conf. recv_buffer_size 中。

set_cache_completion 方法

```
set_cache_completion(cmd_parms *parms, void *dummy, char *arg)
```

记录由 CacheForceCompletion 指定的一个百分数。当读取一个文档超过指定的百分程度时，即使客户取消此次请求，代理服务器也将完成该传输并将文档送入缓冲区。将 arg 字串中的数字解析出，检查它的合法性，并放到 proxy_server_conf. cache_conf. cache_completion 中

set_via_opt 方法

```
set_via_opt(cmd_parms *parms, void *dummy, char *arg)
```

记录由 ProxyVia 指定的 proxy 对 HTTP 协议的 Via 头域的处理方式，有四种选择：on | off | block | full。判断 arg 字串的值，并进行匹配，并往 proxy_server_conf.cache_conf.viaopt 中赋对应的符号常量。

proxy_init 方法

```
proxy_init(server_rec *r, pool *p)
```

调用 ap_proxy_garbage_init，在后者中创建一个 mutex（互斥访问量），用来控制对缓冲区的访问，其引用存放在 proxy_cache.c 中的 garbage_mutex 变量中。

ap_proxy_cache_error 方法

```
ap_proxy_cache_error(cache_req *c)
```

对向临时文件中写出错进行记录。完成临时文件的删除，返回 NULL。

ap_proxyerror 方法

```
ap_proxyerror(request_rec *r, int statuscode, const char *message)
```

根据传入的状态码 statuscode 及错误信息 message，;在请求 r 中设置状态行、error-notes 域、verbose-error-to 域。

ap_proxy_doconnect 方法

```
ap_proxy_doconnect(int sock, struct sockaddr_in *addr, request_rec *r)
```

完成与指定的目的网络地址的连接，出错时记录，并返回-1，否则返回 0。

ap_proxy_bputs2 方法

```
ap_proxy_bputs2(const char *data, BUFF *client, cache_req *cache)
```

将数据 data，写入 client 与 cache->fd 两个流。返回写入的字节数。

ap_proxy_send_hdr_line 方法

```
ap_proxy_send_hdr_line(void *p, const char *key, const char *value)
```

用来往 p 指示的网络连接及 cache 中临时文件里发送传入的键值。只被 table_do_args 方法调用。在发送前，先要确认是 HTTP1.x 的协议。总是返回 1。

ap_proxy_send_fb 方法

```
ap_proxy_send_fb(BUFF *f, request_rec *r, cache_req *c)
```

将缓冲区中读入数据，写到 r 指示的与客户端的连接上，返回写入的字节数。

ap_proxy_send_headers 方法

```
ap_proxy_send_headers(request_rec *r, const char *respline, table *t)
```

发送响应行及各头域。

首先由 r 中取到与客户端的连接，然后将响应行 respline 及 t 中存放的头域发送出去，并将头域存入请求 r 的 headers_out 中。

proxy_getline 方法

```
proxy_getline(char *s, int n, BUFF *in, int fold)
```

与 getline 完全相同。

proxy_get_host_of_request 方法

```
proxy_get_host_of_request(request_rec *r)
```

从请求 r 中解析出主机名返回之。出错返回 NULL。

若已解析过，则直接返回。

否则，若 URL 不是以“ scheme:// ”开头，则记录错误返回 NULL。否则继续调用 ap_proxy_canon_netloc 对 URL 进行全面的解析，若正常解析，则只取主机名，放入 r 的 hostname 域中。

ap_proxy_read_headers 方法

```
ap_proxy_read_headers(request_rec *r, char *buffer, int size, BUFF *f)
```

从缓冲区 f 中读取并解析头域，存到 table 结构中返回。

ap_proxy_c2hex 方法

```
ap_proxy_c2hex(int ch, char *x)
```

将字符 ch 转换成“%”加十六进制的 ASCII 码的字符串形式，存于 x 中。例如当 ch 为“a”时调用此方法，x 将指向字符串“%61”(a 的 ASCII 码值为 97=61H)；ch 为“z”时，x 指向“%7A”。

ap_proxy_hex2c 方法

```
ap_proxy_hex2c(const char *x)
```

将存于 x 中的两位十六进制数解析出来，为 ASCII 码值，返回之。例如当 x 指向字符串“61”时调用此方法，将返回字符“a”的 ASCII 码值 97。

ap_proxy_liststr 方法

```
ap_proxy_liststr(const char *list, const char *val)
```

查找列表 list 中是否包含值 val，是则返回 1，否则返回 0。list 是以“，”为分界符的一系列字串。

ap_proxy_hash 方法

```
ap_proxy_hash(const char *it, char *val, int ndepth, int nlength)
```

对串 it（表示 URL）进行 hash 处理，得到 16 个字节的文摘，然后进行 32 基得到 26 个字符。按照提供的目录深度及目录名长度，将 26 个字符转换成绝对路径，放在 val 中。

32 基的含义是将 0-255（占 8 个 bit）范围内的 ASCII 码序列转换成 0-31（占 5 个 bit）范围内的数值序列，故称 32 基。具体说来是将 5 个字节构成的 40 个 bit 每次取 5 个 bit，通过在高 3 位填 0，构成 8 个新字节。对任意数目的字节，每 5 个一组，最后不足 5 个仍按上述方法进行 5bit=〉8bit 的扩充，但得到的输出字节会不足正常情况下的 8 个字节，这时要用‘=’补齐。

此方法实现了文档到缓冲区地址的映射。

ap_proxy_sec2hex 方法

```
ap_proxy_sec2hex(int t, char *y)
```

将秒数转换成 16 进制字符串，并放在 y 中。例如当 t 为 135a9b01H 时调用此方法，y 中将存放字符串“135A9B01”(采用大写字母)。

ap_proxy_hex2sec 方法

```
ap_proxy_hex2sec(const char *x)
```

将 16 进制字符串转换成秒数，并返回之，与 ap_proxy_sec2hex 方法功能相逆。

ap_proxy_date_canon 方法

```
ap_proxy_date_canon(pool *p, const char *x)
```

将 x 中的时间字串进行处理。若 x 是 RFC 850 格式或 asctime()格式，则转换成 RFC 1123 格式，并将其返回；否则直接返回 x。

ap_proxy_canonenc 方法

```
ap_proxy_canonenc(pool *p, const char *x, int len, enum enctype t, int isenc)
```

将指定长度 len 的字符串 x 进行格式转换。t 指示针对的 URL 部分（如是 path 部分，还是 user 部分等，不同的部分有不同的允许使用的保留字符。），isenc 指示是否允许“%”出现，表达转义。

实际上，完成的工作是将“%##”表示的字母或数字消去转义。

成功时返回经过转换的字串，否则返回 NULL。

ap_proxy_canon_netloc 方法

```
ap_proxy_canon_netloc(pool *p, char **const urlp, char **userp, char
**passwordp, char **hostp, int *port)
```

对传入的 URL 地址 urlp 进行检查，并将正确解析出的主机名、路径、端口号、用户名和密码分别放入传入的 hostp、urlp、port、userp、passwordp 中。若出现错误，则返回错误信息串，否则返回 NULL。有如下输入/输出对应关系：

*urlp 不以“//”开头	Malformed URL
密码部分有误	Bad %-escape in URL (password)
用户名部分有误	Bad %-escape in URL (user name)
端口号部分有误	Bad port number in URL
端口号大于了 65535	Port number in URL > 65535
没有主机名部分	Missing host in URL
确认是 IP 地址但 IP 地址非法	Bad IP address in URL
正确	NULL

proxy_match_ipaddr 方法

```
proxy_match_ipaddr(struct dirconn_entry *This, request_rec *r)
```

判断 r 中的主机名是否与给定的 IP 地址或地址片段匹配。

首先从 r 中解析出主机名，失败则返回 0。

成功则按 IP 地址的形式对主机名进行处理，若正常，则将此 IP 地址与 This 指定的 IP 地址及掩码进行比较，匹配返回 1。否则返回 0。

若不能按 IP 地址形式解析，则认为是域名，调用 ap_proxy_host2addr 对其进行查找与转换。若查找不到，则返回 0。否则，再将找到的对应 IP 地址与 This 比较。匹配，返回 1；否则返回 0。

proxy_match_domainname 方法

```
proxy_match_domainname(struct dirconn_entry *This, request_rec *r)
```

判断 r 中的请求的主机是否在 This->name 指示的域中，是则返回 1，否则返回 0。

proxy_match_hostname 方法

```
proxy_match_hostname(struct dirconn_entry *This, request_rec *r)
```

判断 r 中的请求的主机是否与 This->name 指示的主机名相配，是则返回 1，否则返回 0。

proxy_match_word 方法

```
proxy_match_word(struct dirconn_entry *This, request_rec *r)
```

判断 This->name 是否是 r 中的主机名串的一部分，是则返回 1，否则返回 0。

ap_proxy_is_ipaddr 方法

```
ap_proxy_is_ipaddr(struct dirconn_entry *This, pool *p)
```

检查 This 中存放的名字是否是一个合法的 IP 地址或地址的头片段。

ap_proxy_is_domainname 方法

```
ap_proxy_is_domainname(struct dirconn_entry *This, pool *p)
```

检查 This 中存放的名字是否是一个合法的域地址。

ap_proxy_is_hostname 方法

```
ap_proxy_is_hostname(struct dirconn_entry *This, pool *p)
```

检查 This 中存放的名字是否是一个合法的主机地址。

ap_proxy_is_word 方法

```
ap_proxy_is_word(struct dirconn_entry *This, pool *p)
```

字面上是检查 This **中存放的名字是否是一个合法的字，但实际上该方法总是返回** 1**，尽管此前设置了一个似乎是用于判定的方法的句柄在** This->Matcher **中。**

ap_proxy_host2addr 方法

```
ap_proxy_host2addr(const char *host, struct hostent *reqhp)
```

完成从域名或 IP **地址到用** struct hostent **表示的主机地址信息的转换。**host **若只由数字和“**.**”组成，则认为是** IP **地址，通过** inet_addr **方法将其转换成** u_long **型数字，进而用** gethostbyaddr **来获得** hostent **结构信息；否则认为是域名，通过** gethostbyname **来获得** hostent **结构信息。返回的信息的引用放到** reqhp **中。**

若成功，返回 NULL**；否则，返回错误信息串。**

ap_proxy_garbage_init 方法

```
ap_proxy_garbage_init(server_rec *r, pool *p)
```

完成垃圾清理的初始化工作，主要是调用 ap_create_mutex **方法创建垃圾清理互斥控制量（**mutex）garbage_mute**。**

ap_proxy_garbage_coll 方法

```
ap_proxy_garbage_coll(request_rec *r)
```

完成与请求 r **相关的垃圾的清理。相当于在** help_proxy_garbage_coll **外部增加了一保证互斥访问缓冲区的控制。**

add_long61 方法

```
add_long61 (long61_t *accu, long val)
```

完成 long **型及** long61-t **型数的加法，得到** long61-t **类型的结果。返回值保存于** accu **中。**

sub_long61 方法

```
sub_long61 (long61_t *accu, long val)
```

功能与 add_long61 **方法类似，只是这里是减法。**

cmp_long61 方法

```
cmp_long61 (long61_t *left, long61_t *right)
```

比较大小。若 left=right，则返回 0，否则返回差值，小于为负，大于为正。

gcdiff 方法

```
gcdiff(const void *ap, const void *bp)
```

比较两个 gc_ent 的 expire 域（expiration date），其中 ap、bp 是 gc_ent 型的变量指针。若 ap->expire > bp->expire，则返回 1，小于则为-1，等于 0。

实际用来对缓冲的文件的过期时间进行相互比较，以确定谁应该作为垃圾被收集。

sub_garbage_coll 方法

```
sub_garbage_coll(request_rec *r, array_header *files, const char
*cachebasedir, const char *cachesubdir)
```

将 cachebasedir 与 cachesubdir 共同确定的缓冲区子目录中的内容进行整理。

其中采用了 dirent 方式对目录中的文件进行遍历。

对每一个遍历到的文件，进行如下处理：若是“ . ”，不进行处理；若是“ tmp ”开头的文件，则认为是临时文件，将最后一次访问和修改的时间均是一天前的文件删除，其余不管；若是一个目录，则递归用 sub_garbage_coll 进行处理，若该目录中没有内容，则将其删除掉。

否则认为是缓冲文件。从中读取 26 个字符。对时间进行解析。出错时，若该文件的最后一次访问和修改的时间非法则删除之。最后返回。若时间正确解析，则在 files 中记录该文件的信息。

对于目录和文件，均记录其所占的空间大小。

help_proxy_garbage_coll 方法

```
help_proxy_garbage_coll(request_rec *r)
```

对缓冲区进行整理与垃圾收集。

若无缓冲区或垃圾收集间隔无限大或距上次收集时间太短，则直接退出返回。

取缓冲区根目录下的“ .time ”文件的信息。若出错，则去创建该文件，并立即关闭。若创建出错，则记录最后一次收集的时间为当前时间，并返回。

若正确取出，则用该文件的最后一次修改的时间作为上次垃圾收集的时间与当前时间比较，若间隔不够长，则退出。若够长，则重设该文件的最后一次修改的时间为当前时间，实际是把本次垃圾收集的时间记录下来了。

调用 sub_garbage_coll 对整个缓冲区进行整理。并记录整个缓冲区当前所占空间。若未达到上限，则返回。

否则，需要删除缓冲区中的部分内容以使空间占用合乎要求。首先，对缓冲区中的文件按到期时间进行快速排序。从最早到期的文件开始，依次删除之，直到缓冲区的容量达到要求为止。

最后退出。

rdcache 方法

```
rdcache(request_rec *r, BUFF *cachefp, cache_req *c)
```

使用缓冲区 cachefp 及缓冲区的配置信息 c，读取指定的缓冲区文件。成功返回 1，失败返回 0，若是 UNIX 错误，返回-1。

首先读取一行，看是否是 5 个 8 位 16 进制数，将其分别解析，存入 c 中相应域中。

再读一行，看是否是“X-URL：URL”的形式，是则取出其中的 URL，与 c 中原有的 URL 比较。若匹配，则继续取消息的内容。否则，退出，说明该缓冲区文件与当前的请求不匹配。

再读一行，经适当处理后，存入 cache_req. resp_line 中。将其中的状态码取出，放到 cache_req.status 中。

进一步通过调用 ap_proxy_read_headers 来解析出头域。

将可能存在的 Content-Length 头域取出，放到 cache_req.hrdrs 中，退出。

根据此方法及 ap_proxy_cache_update 方法，读者可以明确知道缓冲的文件的格式。

ap_proxy_cache_update 方法

```
ap_proxy_cache_update(cache_req *c, table *resp_hdrs, const int is_HTTP1, int
nocache)
```

基于从远程主机接收的请求，对缓冲区进行更新。

从响应头的 Expires 域中取过期时间，没有则标记 BAD_DATE。

从响应头的 Last-Modified 域中取最后修改时间，没有则标记 BAD_DATE。

如果响应不是 HTTP_OK 或 HTTP_MOVED_PERMANENTLY 或 HTTP_NOT_MODIFIED，或者 Expires 域非空但解析出错，或者响应是 HTTP_NOT_MODIFIED 但对应的缓冲区文件找不到，或者响应是 HTTP_OK 但本 HTTP1.1 响应中的 Last-Modified 域解析出错，或者本次请求只是要头信息（HEAD），或者请求中包含 Authorization 头域，或者 nocache 参数为 1，则不对响应进行缓冲。这时，若对应缓冲文件存在，则首先将其关闭，然后删除之。最后返回 DECLINED。

否则，及需要对响应进行缓冲。首先，取出响应中的 Date 头域。若没有或取不出，则用当前时间增加一个 Date 头域。将可能的 Last-Modified 时间与 Date 时间比较，若前者还晚，则用后者代替响应中的前者。若无 Last-Modified 域且存在缓冲文件，则用 cache 中的 Last-Modified 时间代替前者。

若响应中的 Expires 时间没有，则用缓冲区中的对应值代替之。

若仍没有 Expires 时间，则计算之。若有 Last-Modified 时间，则用（Date 时间-Last-Modified 时间）*因子和最大 Expires 时间的较小者作为相对 Expires 时间；否则用缺省 Expires 时间最为相对 Expires 时间。当前加相对是绝对。

用新的 Date 时间、Last-Modified 时间、绝对 Expires 时间、加过 1 的缓冲文件版本号、响应体长度重新得到一个 45 个字符的头结构。

若响应是 HTTP_NOT_MODIFIED，则进一步比较响应的 Last-Modified 时间与请求中的 If-Modified-Since 时间。若前者早，则将上面创建的头结构覆盖掉 cache 文件中的相应部分。最后返回 HTTP_NOT_MODIFIED。若后者早，则将缓冲文件中的响应头、头域发送给客户，

且若不是只要头，则体也发出去，并也更新 cache 中的 45 字符头结构，最后返回 OK。

若响应不是 HTTP_NOT_MODIFIED，则生成的 45 字符头结构中的 version 部分清为 0。

获得一个在缓冲区根目录下的唯一的临时文件名，并创建该文件。若失败，返回 DECLINED。若成功，则将本次请求的 URL 地址，以 X-URL:的形式写入该文件。若失败，则记录错误，并将该文件删除。

最后返回 DECLINED。

ap_proxy_cache_tidy 方法

```
ap_proxy_cache_tidy(cache_req *c)
```

将与 c 对应的临时文件代替原来的缓冲文件。

首先，比较 c 中记录的原来的数据内容的长度和写入的数据长度。两者若不等，关闭 c 对应的临时文件，并清除之，返回。

若原来本来没有数据，即长度为-1，则用新的长度写入临时文件的适当位置，最后关闭临时文件。期间出错时，进行记录，并返回。

关闭原来的缓冲文件。出错时，记录并返回。

从缓冲区的根开始，重建原来的缓冲文件所在的目录层次。若出错，则进行记录。

将临时文件，移动到原来的缓冲文件处，并改成原名。

ap_proxy_cache_check 方法

```
ap_proxy_cache_check(request_rec *r, char *url, struct cache_conf *conf,
cache_req **cr)
```

在缓冲区中查找指定资源 url，成功则返回一 HTTP 状态码；否则需要检查远程服务器，返回 DECLINED。

首先检索请求中的 If-Modified-Since 头域，将时间值放在 cache_req.ims 中。

按请求中的 URL 地址，得到对应的缓冲文件的完整路径名。

如果缓冲文件名有效且是 GET 请求且 URL 地址长度小于 1024 且 pragma 头域中没有 no_cache 指示且不需要进行身份验证，则去打开对应缓冲文件。打开出错时，显示并记录有关信息并退出。

判断过期时间是否未到，是则进一步判断是否最后一次修改的日期 lm 比请求中指定的日期 ims 早，并且缓冲写入的日期比 ims 晚，则将可能有的 cache 中的 Expires 头域写入应答，而并不写数据，并返回应答码 HTTP_NOT_MODIFIED，。

若 lm 比 ims 晚，则将缓冲中的响应行放入 request_rec.status_line 中，将状态码放到 request_rec.status，并且如果请求不是 HTTP0.9，则将缓冲区中的各个头域也放到响应中。检查请求是否是 HEAD，否则调用 ap_proxy_send_fb 将缓冲区中的实体部分也发送出去。最后关闭打开的缓冲文件，返回 OK。

若缓冲的内容已过期，应重新去原始服务器上取。首先将缓冲中的 lm 与请求中的 ims 进行比较，若 lm 晚，则用 lm 代替 ims，并返回 DECLINED。

allowed_port 方法

```
allowed_port(proxy_server_conf *conf, int port)
```

判定给出的端口是否是被当前的 proxy 服务器所允许的。是则返回 1；否则返回 0。

ap_proxy_connect_handler 方法

```
ap_proxy_connect_handler(request_rec *r, cache_req *c, char *url, const char *proxyhost, int proxyport)
```

对基于 SSL 协议的 proxy 请求进行处理。

首先从目的 url 中取出端口号，无则取缺省的 HTTPS 的端口号。

检查给出的远程的 proxy 是否在由 ProxyBlock 定义的阻塞范围内。若是，则调用 ap_proxyerror 记录错误，然后返回 HTTP_FORBIDDEN。

若无 proxy 所允许的端口的定义,则看上面取的端口号是否是 DEFAULT_HTTPS_PORT 或 DEFAULT_SNEWS_PORT，不是则返回 HTTP_FORBIDDEN。若有允许的端口的定义，则调用 allowed_port 进行检查，若不在允许的范围内，则也返回 HTTP_FORBIDDEN。

若传入的远程 proxy 的地址非空，则显示信息：将连接远程 proxy；否则，显示：将连接目的服务器上的指定端口。

创建一个 socket。若失败，记录并返回 HTTP_INTERNAL_SERVER_ERROR。

取得要连接的远程服务器或目的主机的 struct hostent 形式的地址，并进一步构造将在 connect 方法中使用的远程地址。调用 ap_proxy_doconnect 在本地与远程之间建立连接。

若连接建立失败，则调用 ap_pclosesocket 关闭本地 socket，并记录错误信息，返回 HTTP_INTERNAL_SERVER_ERROR。

否则进一步向远程主机发送请求。

若要连远程 proxy，则将“CONNECT”+目的 URL+“HTTP/1.0”行发给该服务器，然后发送 Proxy-agent 头域将本服务器的版本信息发出；否则是连目的服务器，则将“ HTTP/1.0 200 Connection established”发送给客户端，然后发送 Proxy-agent 头域给客户。

设置对与客户及远程主机的连接同时监视，并进入无限循环。开始时，首先阻塞，直到某连接有数据来为止。

若有来自远程主机的数据，则读出并发送给客户；若有从客户来的数据，则读出并发送给远程主机。若读写出错或其它使阻塞条件不成立的情况出现，则退出循环。

关闭为与远程主机连接而建立的 socket，返回 OK。

ap_proxy_http_canon 方法

```
ap_proxy_http_canon(request_rec *r, char *url, const char *scheme, int def_port)
```

将传入的以“//”开头的 url 及所基于的协议 scheme 进行合并，并进行适当的转换，结果放在 r 的 filename 域中。def_port 指定与 scheme 对应的缺省的端口号。

若处理中有错误，则返回 HTTP_BAD_REQUEST，否则返回 OK。

clear_connection 方法

```
clear_connection(pool *p, table *headers)
```

将 headers 中包含的针对仅此次连接的头域清除掉，参阅 HTTP1.1 文档的 14.10 节。

首先清除掉 headers 中的 Proxy-Connection。

然后取出 headers 中可能存在的 Connection 头域，并把其中指定的头域从 headers 中也清除掉。

最后清除 Connection 头域。

proxy_location_reverse_map 方法

```
proxy_location_reverse_map(request_rec *r, const char *url)
```

判断 url 中是否包含 ProxyPassReverse 配置项中出现的虚拟的远程服务器地址。若出现，则将对应地址部分改成实际要访问的本地的地址。直接在 url 中进行处理。

ap_proxy_http_handler 方法

```
ap_proxy_http_handler(request_rec *r, cache_req *c, char *url, const char
*proxyhost, int proxyport)
```

处理基于 HTTP 通过远程 proxy 进行网络访问。

首先看 url 中是否有“ :// ”，无则返回 HTTP_BAD_REQUEST。

否则，将主机名、端口号、路径分别解析出来，没有显式的端口号则用 DEFAULT_HTTP_PORT 代替。

对目的主机名进行检查，看其是否在 ProxyBlock 配置中出现。是则记录错误，返回 HTTP_FORBIDDEN。

对传入的 proxy 地址 proxyhost 进行检查，非法则返回 DECLINED。

否则对目的主机地址进行检查，非法则进行记录并返回 HTTP_INTERNAL_SERVER_ERROR。

然后，创建一个 socket，出错时进行记录并返回 HTTP_INTERNAL_SERVER_ERROR。

若配置文件中有合法的 ProxyReceiveBufferSize 定义，则对创建的 socket 的对应参数进行设置。若失败，记录错误信息。

调用 ap_proxy_doconnect 在本地与远程主机之间建立连接。若建立失败，则：若目的主机是给定的 proxy，则返回 DECLINED；否则，记录错误信息，返回 HTTP_BAD_GATEWAY。

若连接成功，则调用 clear_connection 清除客户端传来的仅与本 proxy 的连接有关的头域。

然后，向远程主机发送请求行。其中，若远程主机是 proxy，则发送包含协议模式的完整的 URL，否则只发送路径部分。

继续发送 Host 头域，指出目的服务器的地址。并且若显式地给出了不同于缺省的端口号，则也包含在其中。

根据配置文件中的 Via 配置项，对原客户请求中的 Via 头域进行控制。若定义了“ block ”，则将 Via 头域清除掉；若定义了“ off ”，则不进行任何操作；若定义了“ on ”或“ full ”，则将加上关于本 proxy 的 Via 头域。后者的具体操作是：取得接收客户请求的本服务器上的端

口号，若不是缺省值，则转换成字符串表示；然后进一步调用 ap_table_mergen 将本地信息附加到转发的 Via 头域中，且若是 full，则把本 proxy 的版本信息也加上。

将请求中的头域发送给远程主机，除非是键或值为空、或是 Host 头域、或是 Proxy-Authorization 头域。

通过调用 ap_should_client_block 判断是否需要读请求体。需要则调用 ap_get_client_block 读数据，并将数据直接发送给远程的主机。其间可能根据设置将请求体进行解 chunked 格式。

从远程主机接收一行响应，若读出错，则关闭与该主机的连接，记录错误，并返回 HTTP_BAD_GATEWAY。

检查远程主机的响应是否是 HTTPx.x 的格式，否则说明为 HTTP0.9 协议，则记录该信息，并将 200 及对应字串形式放到 r 的 status 及 status_line 域中。

若是，则将协议版本号取出，取不出时用缺省的 1.0；若协议版本号不是 1.x 或响应行长度超出缓冲空间，则关闭与远程主机的连接，返回 HTTP_BAD_GATEWAY。记录当前的响应不是 0.9 版本。将状态码的数字及包括描述在内的对应字符串形式放到 r 的 status 及 status_line 域中。然后读取从远程主机传来的头域。若出错，则记录错误，并标记以后不将该响应存入 cache。按配置要求，将关于当前 proxy 的 Via 头域，增加到响应头域列表中。调用 clear_connection 清除响应头域中针对此次连接的部分。

将响应头域存入 cache_req 的 hdrs 域中。

对响应的 Date、Last-Modified、Expires 头域中的时间格式进行标准化。

调用 proxy_location_reverse_map 对响应的 Location 和 URI 头域的路径进行转换，变成实际的本地地址。

检查是否对目的主机不使用 cache，并设置有关标志。

调用 ap_proxy_cache_update 对 cache 进行更新并进行适当处理。若返回值非 DECLINED，则关闭与远程主机的连接，进一步返回该值。若是 DECLINED，则发送 HTTP1.x 要求的响应行给客户，并将同样的内容写入在 ap_proxy_cache_update 中创建的临时文件中。若写出错则调用 ap_proxy_cache_error 进行记录和处理。然后向连接及临时文件中写入一空行。

若是 HTTP0.9，则将其余数据写入客户连接与临时文件。若是 HTTP1.x 且不是只取头，则将远程主机发来的后续数据写到客户连接与临时文件中。

调用 ap_proxy_cache_tidy 对 cache 中在前面创建的临时文件进行整理，转化成正式的 cache 文件。

关闭打开的临时文件句柄。清除与本次连接有关的资源占用。

最后调用 ap_proxy_garbage_coll 对 cache 进行整理。

返回 OK。

alias_match 方法

```
alias_match(const char *uri, const char *alias_fakename)
```

检查 uri 是否以 alias_fakename 指示的别名为前缀。是则返回 alias_fakename 的结束位置在 uri 中的偏移量，否则返回 0。

proxy_detect 方法

```
proxy_detect(request_rec *r)
```

如果允许 proxy 请求且已经将当前的请求中的 scheme 分离出来，则如果请求中的主机名没有或协议模式不是 HTTP 或请求的端口和主机名与任何虚拟服务器不匹配，则认为是 proxy 请求，设置标志在 request_rec.proxyreq 中，并将‘ proxy: ’及路径存入 request_rec .filename 中，且将进一步处理的句柄设为“ proxy-server ”。否则，如果允许 proxy 请求且当前请求的方法是 M_CONNECT 且其中的主机名和端口名已分离出来，则进行同上的设置。否则，退出。本方法总返回 DECLINED(=0)。

proxy_trans 方法

```
proxy_trans(request_rec *r)
```

将请求 r 中的可能包含的虚拟的本地路径替换成实际的远程服务器的地址（这一部分原本应是在 www 服务器中实现的，但既然 Apache 是一个 www 和 proxy 的混合的服务器，所以将它放到这里也没有关系了。）。

首先，如果当前的请求已经被认为是 proxy 请求（经过了 proxy_detect 的处理），则直接返回，否则依次来判断与在 proxy_server_conf .aliases 中的虚拟本地路径的匹配。若成功，则设置有关标志。总是返回 DECLINED(=0)。

proxy_fixup 方法

```
proxy_fixup(request_rec *r)
```

对 proxy 请求 r 中的 URL 进行标准化或修正。

首先检查 r 是否是 proxy 请求，否则返回 DECLINED。

进一步地，若是基于 HTTP 或 FTP 的请求，分别调用 ap_proxy_http_canon 和 ap_proxy_ftp_canon 对 URL 进行标准化。

若标准化后的 URL 中不含“ :”或以“ :”开头，则返回 HTTP_BAD_REQUEST。否则返回 OK。

proxy_needsdomain 方法

```
proxy_needsdomain(request_rec *r, const char *url, const char *domain)
```

如果不是 proxy 请求或不是 GET 方法或 r 中的主机地址为空，则返回 DECLINED。

否则，看主机地址是否是包含“ . ”或是“ localhost ”，是则返回 DECLINED。

否则，将缺省的域名插入到仅有的机器名后。调用 ap_unparse_uri_components 得到完整的 URI 地址，然后将该地址作为 Location 域的值加到请求 r 的响应部分中。最后记录没有“ 域地址部分 ”错误，然后返回 HTTP_MOVED_PERMANENTLY。

proxy_handler 方法

```
proxy_handler(request_rec *r)
```

如果请求 r 不是 proxy 请求或者 r 中的文件名 filename 不以“ proxy: ”开头，则返回 DECLINED(=0)。

否则，若请求的方法是“ TRACE ”，则检查 Max-Forwards 头域，若对应值小于 1，则重设 proxy 请求标记为 0，并返回对 TRACE 的响应，最后返回 OK。否则将 Max-Forwards 头域的值减 1。并检查是否有 chunked 块，有则返回错误码。

否则，调用 ap_proxy_cache_check 确定 cache 中是否有备份。若返回值非是 DECLINED，则认为已经完成了响应的处理，直接将该值返回退出。

否则，调用 proxy_needsdomain，判断 r 中的主机地址是否包含域地址，若返回的结果是要求重定向的，则返回 HTTP_MOVED_PERMANENTLY。

否则，检查当前的客户是否已被设置在 NoProxy 配置项中。若有，则应该绕过 RemoteProxy 中指定的远程的级联 proxy。否则，应该级联。

若级联，则检查当前的请求是否与定义的级联中的访问模式相符。若相符且是 CONNECT 请求，则调用 ap_proxy_connect_handler 进行处理；若相符且是 http 协议模式，则调用 ap_proxy_http_handler 进行处理。否则仍然认为不级联。当上面的处理出现问题时，将返回的状态码返回。

若不级联或级联的特殊处理已完成，则分为四种情况：

1. 是 CONNECT 请求，则调用 ap_proxy_connect_handler 进行处理；
2. 是 http 协议模式，则调用 ap_proxy_http_handler 进行处理；
3. 是 ftp 协议模式，则调用 ap_proxy_ftp_handler 进行处理；
4. 返回 HTTP_FORBIDDEN。

第7节 CGI模块

作为 Internet 上最主要的信息管理和组织手段，WWW 由一系列相应的技术及应用构成，其组成技术包括 HTTP、HTML、URL 以及 CGI 等。WWW 服务器本身提供一些基本功能，以完成客户端的请求和自身的管理。但不同的用户有不同的功能要求，其中许多功能是 WWW 服务器本身不能提供的（比如在线查询等），它必须提供一种扩展手段，以允许用户编写扩展应用程序来扩展服务器的功能。CGI（Common Gateway Interface，即通用网关接口）就是这样的一种标准扩展技术。

下面首先对 CGI 的基本技术原理及其开发做简单论述，然后讲解 Apache 对 CGI 的支持，包括：配置、使用、实现的细节等。

7.7.1. CGI原理

概述

CGI 是外部扩展应用程序与 WWW 服务器交互的一个标准接口。按照 CGI 标准，编写的外部扩展应用程序可以处理客户端（一般是 WWW 浏览器）输入的协同工作数据，完成客户端与服务器的交互操作。这在实际应用中非常有用，如可以编写 CGI 外部扩展程序来访问外部数据库，客户端用户可以通过它和 WWW 服务器来进行数据查询。

CGI 一般分两种：标准 CGI 和缓冲 CGI。所有的 WWW 服务器均应支持标准 CGI，按标准 CGI 编写的程序与具体的 WWW 服务器无关。而按缓冲 CGI 编写的程序与 WWW 服务器有关。

下面先来看一看标准 CGI 的工作原理。

1. 标准 CGI

客户端、服务器、CGI 接口与 CGI 程序间的关系如图 7.8 所示：

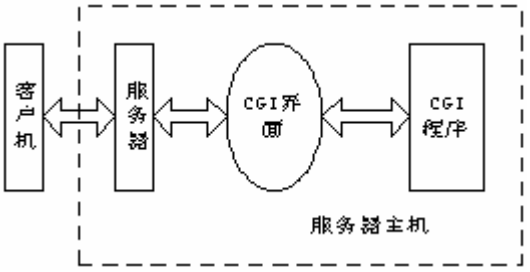


图 7.8

服务器是客户端（如浏览器）与 CGI 程序之间的通道。当客户端的用户完成了一定输入工作（比如填充完 HTML 文档中的 FORM 表）之后向服务器发出 HTTP 请求（称为 CGI 请求），服务器进程接收到该请求后，设置一些处理 CGI 请求所需的环境变量，然后启动 URL 指定的 CGI 程序，并与该进程（称为 CGI 进程）保持同步。CGI 进程将处理结果传递给服务器进程，服务器则进一步将其应答送给客户。

CGI 程序通过环境变量、命令行参数、标准输入输出与 WWW 服务器进行通信，传递有关参数和处理结果。

- 环境变量：当服务器创建 CGI 进程时，设置相应的环境变量和命令行参数，以传递客户端和服务器的有关信息给该进程。
- 命令行参数 :命令行参数仅在请求的来源是 HTML 文档中的 ISINDEX 查询时才使用。
- 标准输入输出：当 HTTP 请求方法为 POST 时，CGI 程序通过标准输入流和有关环境变量来获取客户端传输数据；如是 GET 方法，CGI 程序直接通过环境变量获取客户端传输来的数据。当 CGI 程序要返回处理结果（一般为 HTML 文档）给客户端时，它通过标准输出流将该结果数据传递给服务器进程。

2. 缓冲 CGI

标准 CGI 使用标准输入/输出来进行数据通讯，这是由其最初开发环境（Unix 操作系统）所决定的。许多 Windows 环境下的编程工具（如 VB 和 Delphi 等）并不支持这种 I／O 方式，这时就不能用它们来开发基于标准 CGI 的应用程序。于是有些服务器提出了缓冲 CGI 的概念。缓冲 CGI 亦称为 WinCGI。

此时 CGI 程序与服务器间通过缓冲 CGI 而不是标准 CGI 进行通信，缓冲 CGI 的实现体现在 WWW 服务器中增加的内置缓冲处理程序上，该程序与服务器间的通讯还是遵照标准 CGI 接口。

缓冲 CGI 的工作原理与标准 CGI 相似，不同的是当服务器进程接收到客户端的 CGI 请求时，除了将 CGI 请求的有关数据仍设置成环境变量外，还将它们保存在输入缓冲区中；而缓冲处理程序则在 CGI 程序与服务器间建立两条数据通道（输入／输出缓冲区）。CGI 进程通过输出缓冲区将处理结果传递给服务器进程。

这里的 CGI 程序通过环境变量和输入／输出缓冲区与 WWW 服务器进行通信，传递有关参数和处理结果。这时环境变量的意义同上，不过这些环境变量及其相应值保存在输入缓冲区中。此外，输入缓冲区中还存放客户端的传输数据（如使用 POST 方法的话）。输出缓冲区用来存放 CGI 程序的处理结果。

3. 标准 CGI 与缓冲 CGI 的区别

对 CGI 程序而言，最主要差别在于数据的 I／O 不同：对于缓冲 CGI，服务器与 CGI 程序间的数据交换是通过缓冲区；而标准 CGI 是通过标准 I／O。

使用缓冲 CGI 可选择更多的开发工具，可以开发 Windows95 和 WindowsNT 下的 CGI 程序；而使用标准 CGI 所选用的开发工具必须支持标准 I／O。

只有少数几种 WWW 服务器支持缓冲 CGI，因此基于它的 CGI 程序兼容性不如标准 CGI 好。

CGI 与其他 WWW 技术的关系

CGI 作为 WWW 服务器的功能增强技术，由上面 CGI 的基本原理可知，它和许多其它的 WWW 技术密切相关，如 HTTP、HTML、 MIME 和 URL 等。下面主要就它与前两种技术的关系进行研究。

1. CGI 与 HTTP 协议

CGI 通过 HTTP 协议在客户端和服务端进行通讯：

- 客户端用户代理向服务器发送的请求是 HTTP 请求消息。该消息中含有处理用户输入的 CGI 扩展程序的 URL 值。
- CGI 扩展程序在处理结束后，返回给客户端的应答是 HTTP 应答消息。因此 CGI 程序的输出数据必须符合 HTTP 应答消息的语法格式，这在基于 CGI 标准的开发中非常重要。

2. CGI 与 HTML 语言

CGI 程序的输出数据（HTTP 应答消息）一般有两种：符合 MIME 类型的文档（最普遍的是 HTML 文档，表示为 text／html）；指向其它文档的 URL 链接。这两种方式都与 HTML 语言有关，数据的组织须符合 HTML 语法格式。

CGI 与其它页面动态生成技术的比较

除 CGI 技术外，常见的同时代其它页面动态生成技术还有 API（Application Programming Interface）和 SSI（Server Side Includes）。这两种技术都与具体的 WWW 服务器有关。下面先对这两种技术简要说明，然后再对这三种技术作比较。

SSI 技术：当服务器进程接收到客户端请求时，其子进程扫描客户端所请求的文档，以获取数据插入标志，再插入相关动态数据，然后返回给客户端。这种技术简单，但效率不高。

API 技术：不同的 WWW 服务器提供的 API 可能不同，比如 O＇Reilly Website 提供的是 WSAPI；Purveyor Webserver 和 Microsoft IIS 提供的是 ISAPI。利用这些 API 编写程序的流程和最后生成的文件类型也不一样；比如用 ISAPI，编写程序时必须依照它的特定框架，最后建立的文件可以是 EXE 文件和 DLL 文件。

下表列出了 CGI、SSI 和 API 三种技术在主要性能方面的比较：

表 7.2

	SSI	CGI	WSAPI/ ISAPI

实现的灵活性	差	好	好
可实现的功能	差	强大	强大
对程序员的要求	低	一般	较高
开发与测试时间	短	一般	较长
花费	低	一般	较高
操作的风险性	低	低	较大
移植性	较好	较好	差
CPU 的负载	高	高	较低

CGI 和 API 都需要编程实现，利用它们可访问更多的数据源；而 SSI 不需编程。利用 API 开发的扩展软件，其运行速度比 CGI 快，功能也较 CGI 强；但其操作有一定的风险，出现错误时可能使系统崩溃（CGI 程序不会），并且它非常依赖于具体的服务器而移植性太差，开发周期较长。CGI 在操作的风险性、软件移植性、软件开发难度等方面较 API 有优势；其缺点在于 CPU 负载较高，特别是同时发生的 CGI 请求很多时，将影响服务器的综合性能。

综合比较而言，CGI 比其它两种技术更优秀，在实际应用中也最为普遍。

7.7.2. CGI 程序的编写

基于 WWW 的人机交互一般有两种情况：本地交互和通过网络传输的交互。前者是指客户端用户的输入数据在客户端本地进行处理，然后将处理结果返回给用户，常见的开发工具有 JavaScript（Netscape 开发）和 VBScript（Microsoft 开发）；后者是指客户端用户输入的数据通过网络传输到 WWW 服务器，服务器处理结束后将处理结果返回给客户端用户，常见的开发技术是 WWW 服务器扩展技术（如 CGI，API 等）。此处主要讨论基于标准 CGI 技术的通过网络进行数据传输的交互实现。

对这种人机交互的实现，主要有三个环节需要解决：如何获取客户端传输的数据，如何提取有效数据并处理这些数据，如何向客户端返回应答。下面分别结合有关技术，谈谈这三方面问题的解决。

客户端传输数据的获取

由前面的 CGI 原理可知，当服务器进程接收到客户端用户代理（如浏览器）提交的 CGI 请求时，会设置与 CGI 请求内容有关的环境变量，并建立服务器与外部 CGI 程序之间通讯的通道（即标准 I／O）。CGI 程序可以通过环境变量，标准 I／O 或命令行参数获取客户端用户输入的数据。数据的获取与请求所采用的 HTTP 方法（Method）和用户所使用的请求方式有关。

用户通过 CGI 请求数据一般有三种方式：HTML FORM 表、ISINDEX、可点击图片（ISMAP 或 Imagemaps）。后两种方式是通过命令行参数传递用户的输入数据；在 C 语言中（下面的举例亦然），CGI 程序可以用 argc 和 argv［int］获得这些参数值。而前一种方式则取决于 HTTP 请求方法；但不管采用何种方法，都将用到环境变量来传递有关请求内容。

● 获取环境变量

环境变量的类别很多，包含客户端和服务端的详细信息。在一般 CGI 程序开发中，下述几个环境变量在数据传递中起着重要作用。

- GATEWAY—INTERFACE

CGI 程序所使用的 CGI 标准接口的版本号。如使用的 CGI1．1 版，该变量表示为“ CGI／1．1”*REQUEST—METHOD。其中 REQUEST—METHOD 为 HTTP 请求方法。根据该变量值可判断 CGI 请求所采用的请求方法，以决定是通过 Stdin 还是通过环境变量 QUERY—STRING 获取客户端传输数据。

➢ QUERY—STRING

QUERY—STRING 是 CGI 请求的 URL 中“ ？”之后的数据。当使用 ISINDEX 查询或 FORM 表应用 GET 方法时，客户端传输数据可以通过读取该变量而获得。

➢ CONTENT—LENGTH

CONTENT—LENGTH 表示客户端传输数据的字节数。

➢ CONTENT—TYPE

CONTENT—TYPE 表示客户端传输数据的数据编码类型。

● HTTP 请求方法

客户端用户代理提交的 CGI 请求是 HTTP 请求，其中包括 HTTP 请求方法。HTTP 协议定义的请求方法中常用的主要有 GET 和 POST。客户端 FORM 表的 METHOD 属性用来设置请求方法，其缺省值为 GET。

如果在 FORM 中使用 GET 方法，CGI 程序通过环境变量 QUERY—STRING 获取客户端传输数据。

如果在 FORM 中使用 POST 方法，CGI 程序通过通过 CONTENT—LENGTH 获取客户端传输数据的字节数，通过 Stdin 读取客户端传输数据。

有效数据的提取和处理

通过上述方式获取的客户端传输数据的一般格式为：

name［1］=value［1］&name［2］=value［2］&... name［i］=value［i］... name［n］=value［n］

（1｜=i｜=n）

其中 name［i］表示变量名，它是在 FORM 表中某输入域的名字；value［i］表示变量值，它是用户在 FORM 表中某输入域中输入的值。

客户端传输数据的每对“ Name=Value ”串由‘ & ’字符分隔，其数据编码类型可以从环境变量 CONTENT—TYPE 获取。CGI／1.1 版仅支持‘ application／x－www－form－urlencoded ”编码方式。这种编码方式和 URL 的编码方式一样，遵循两个规则：数据中的空格（ASCII 码值 32）编码成‘ ＋ ’号；保留字符编码成“ %XX ”形式，“ XX ”是该字符 ASCII 值的十六进制表示，比如“ $ ”的编码为“ %24 ”，“ ？”的编码为“ %3F ”。

因此，要获取客户端用户的输入数据，必须对上述获取的数据进行分离和解码等处理。利用函数 strtok（）、strchr（）等可以实现数据分离处理，而数据的解码则需要对整个数据串进行扫描，将数据串中的“ %XX ”复原为对应的 ASCII 码。

在提取到有效数据后，还可能进行许多其他的处理，如数据库查询等。这种处理与普通编程相同。

向客户端返回应答

CGI 程序处理结束后，通过标准输出流将应答信息传递给服务器，再由服务器返回给发出请求的客户端。其输出的应答信息是 HTTP 应答消息，它一般由两部分组成：应答头和应答数据。

常见的应答头包括三种头域：Content—Type（数据编码类型，用 MIME 类型表示），Location（特定文档的 URL，这种情况不直接向客户端输出内容而输出该 URL）和 Status（处理结果的状态码和状态描述）。HTTP 应答头由几行格式相同的文本构成，每行的基本格式为："头域名：该域内容"。应答头和应答体之间用一空行加 LF（或 CR/LF）分隔。应答体为 CGI 扩展程序的输出数据，其数据类型应该与 Content—Type 值相一致。

CGI 程序的输出可以用 printf、puts 等标准 I/O 函数来实现。

CGI 程序的开发及其一般流程

在开发 CGI 程序过程中，可根据实际情况（服务器提供的接口、实际需求和程序员经验等）选择编程语言，如 C/C＋＋、Perl、TCL、Unix shell、VB、Java 等。如果选用 C/C＋＋等语言，必须编译成可执行文件；如果选用 Perl、Java 等解释语言，服务器必须安装相应的解释器。

如下为三个分别用 Perl、Java、Shell 编写的 CGI 程序。程序的功能相同，均是将当前的环境变量输出来：

1. Perl 语言 CGI 程序

```
#!perl.exe

print "Content-type: text/plain\n\n";

for $key(%ENV){
print "$key evaluates to $ENV{$key} ";
}
```

其中第一行是告诉调用者需要使用 perl.exe 解释程序来执行本脚本。

2. Java 语言 CGI 程序

对于 Java 语言，由于其自身设计使访问环境变量很不方便，JVM 使用上的复杂又使启动一个 Java 程序非常困难，因此出现了一些工具来解决这一问题。其中 java_cgi 便是其中之一。你可以从 ftp://ftp.orbits.com/pub/software/java_cgi-0.4.tgz 处获取该工具。

对执行 Java 写的 CGI 程序的困难和问题的解决，java_cgi 采用一个总控程序 java.cgi 及若干 API 来完成。

java.cgi 将环境变量放在了一个临时文件中。而提供的 API 中的 CGI 类可方便读取该文件，以获得某些信息，如环境变量等。对用户来说，编写 Java 程序时，可以使用上述提供的 API，通过它们来访问和使用 CGI 标准（环境）。这样，用户就不必再为 Java 脚本的执行而烦恼了。

页面中对 Java 脚本的引用，均是如下的形式：java.cgi/JavaFileName,在 java.cgi 中通过 $PATH_INFO 可以得到 JavaFileName，从而去调用 java 来解释该 Java 字节码文件。

具体的 java_cgi **工具的安装和使用请参阅其所附带的文档。如下为基于** java_cgi **编写的示例：**

```
import java.io.*;
import java.util.*;

import Orbits.net.*;

//  This class will get all of the CGI information and return it to the
//   browser.
public class CGI_Test {

public static void main( String argv[] ) {

    //  Produce output that the browser will understand.
    System.out.println( "Content-type: text/plain" );
    System.out.println( "" );

    //  Get the CGI information.
    CGI info = new CGI();

    //  Put the list of variable names into a list.
    Enumeration e = info.getNames();

    while ( e.hasMoreElements() ) {

        //  For each name in the list.
        String name = (String) e.nextElement();
        System.out.print( name );
        System.out.print( " evaluates to " );

        //  Get the associated value.
        String value = info.getValue( name );
        System.out.println( value );
    }
}
}
```

其中 Orbits.net **为** CGI **类所在的软件包名。**CGI **类有两个方法：**getNames **和** getValue**。前者是得到当前环境变量名的枚举；后者是获取指定环境变量的值。**

另外 java_cgi **中还为用** Java **语言生成** HTML **文件提供方便。**

3. Shell

```
#!/bin/sh
      echo Content-type: text/plain
      echo
```

```
echo CGI/1.0 test script report:
echo
echo argc is $#. argv is "$*".
echo

echo SERVER_SOFTWARE = $SERVER_SOFTWARE
echo SERVER_NAME = $SERVER_NAME
echo GATEWAY_INTERFACE = $GATEWAY_INTERFACE
echo SERVER_PROTOCOL = $SERVER_PROTOCOL
echo SERVER_PORT = $SERVER_PORT
echo REQUEST_METHOD = $REQUEST_METHOD
echo HTTP_ACCEPT = "$HTTP_ACCEPT"
echo PATH_INFO = $PATH_INFO
echo PATH_TRANSLATED = $PATH_TRANSLATED
echo SCRIPT_NAME = $SCRIPT_NAME
echo QUERY_STRING = $QUERY_STRING
echo REMOTE_HOST = $REMOTE_HOST
echo REMOTE_ADDR = $REMOTE_ADDR
echo REMOTE_USER = $REMOTE_USER
echo CONTENT_TYPE = $CONTENT_TYPE
echo CONTENT_LENGTH = $CONTENT_LENGTH
```

7.7.3. 环境变量配置

下表为 CGI 程序经常用到的一些环境变量及其含义，详细的描述请参阅 CGI 标准规范。

这其中某些是与 CGI 的使用直接相关的，有：GATEWAY_INTERFACE、SERVER_PROTOCOL、REQUEST_METHOD、QUERY_STRING、REQUEST_URI、SCRIPT_NAME、PATH_INFO、PATH_TRANSLATED。其余的则是提供额外的信息。

表 7.3

环境变量名	描述	示例
SERVER_NAME	CGI 所在服务器的主机名或 IP 地址.	SERVER_NAME= 159.226.42.68
SERVER_SOFTWARE	服务器软件的名称和版本，格式为：name/version。	SERVER_SOFTWARE= Apache/1.3.6 或 CERN/3.0 或 NCSA/1.3.
GATEWAY_INTERFACE	使用的 CGI 标准的版本，格式为：CGI/revision。	GATEWAY_INTERFACE= cgi/1.1 dos
SERVER_PROTOCOL	服务器运行基于的 HTTP 协议，格式为：HTTP/revision。	SERVER_PROTOCOL= HTTP/1.1
SERVER_PORT	服务器使用的 TCP 端口，通常 WWW 服务器是 80。	SERVER_PORT =80
REQUEST_METHOD	POST 或 GET, 取决于客户提交任务的方式。	
HTTP_ACCEPT	浏览器能接收和处理的 MIME 类型，格式为：type/subtype。注意在文件中每一种 MIME 类型必须独占一行。	
HTTP_USER_AGENT	递交请求的浏览器的名称、版本和其他平台	HTTP_USER_AGENT=

	性的附加信息。	Mozilla/4.0 (compatible; MSIE 5.0; Windows NT)
HTTP_REFERER	递交表单的文本的 URL，不是所有的浏览器都发出这个信息，不要依赖它。	
PATH_INFO	客户请求中 URL 包含的路径信息。	
PATH_TRANSLATED	服务器将客户请求的 URL 的相对地址转化为绝对地址所需要的那一段路径，即绝对路径之相对路径多的那一段路径。	PATH_TRANSLATED= C:\HTTPD\CGI-BIN\
SCRIPT_NAME	客户在提交请求时在 URL 中指明的 CGI 程序的名字，如客户访问的 URL 为 http://www.vt.com/cgi-bin/prog?p = mrp，则 SCRIPT_NAME 为 prog。	
QUERY_STRING	客户提出的请求 URL 中问号后面的内容，即我们需要查询的字串。注意：只有 REQUEST_METHOD=GET 时，该参数才起作用，这是 CGI 采取环境变量法所独有的。	
REMOTE_HOST	客户主机名。	
REMOTE_ADDR	客户主机的 IP 地址。	
REMOTE_USER	客户的用户名。当服务器使用身份认证时才设置本值。	
REMOTE_IDENT	表示客户是否合法。	
CONTENT_TYPE	该参数只有在 REQUEST_METHOD=POST 时才起作用。该参数表示文件的 MIME 数据类型。	
CONTENT_LENGTH	对于用 POST 递交的表单，标准输入口的字节数.	

7.7.4. Apache 中的 CGI 错误日志文件

调试是 CGI 编程中的一大困难，因为 CGI 程序的标准输出和标准错误输出被 www 服务器接收，我们无法看到。Apache 服务器为 CGI 程序的调试提供了可能。

Apache 服务器对与 CGI 执行相关的错误信息进行日志记录。查看日志文件可以知道执行 CGI 程序发生异常时内部的原因是什么。

在 Apache 中，CGI 程序执行异常的信息可以单独记录在一个 CGI 执行错误日志文件中。当进行了适当的配置后，该文件将记录所有 CGI 程序非正常执行的情况。

每一次事件均在该日志文件中若干行描述。其中头两行的格式如下：

```
%% [time] request-line
%% HTTP-status CGI-script-filename
```

如果出现的问题使 CGI 无法运行，则会再包含如下两行：

```
%%error
error-message
```

而如果是由于 CGI 运行的输出结果所含的头信息有误（通常是由于程序中有 bug 造成的），则会记录如下信息：

```
%request
All HTTP request headers received
POST or PUT entity (if any)
%response
All headers output by the CGI script
%stdout
CGI standard output
```

```
%stderr
CGI standard error
```

其中，若 CGI 程序没有任何的标准输出或标准错误输出，则不会记录上述的%stdout **和** %stderr **信息。**

如下为笔者在实际工作中得到的一个 CGI 执行错误日志文件的片段，供读者参考：

```
%% [Tue Jul 06 09:54:45 1999] GET /cgi-bin/status_report.exe&restart=Apache
HTTP/1.0
%% 404 c:/program files/apache
group/apache/cgi-bin/status_report.exe&restart=apache
%error
script not found or unable to stat

%% [Fri Jul 09 11:58:04 1999] GET /cgi-bin/redirect.pl HTTP/1.0
%% 500 c:/program files/apache group/apache/cgi-bin/redirect.pl
%request
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, image/png, */*
Accept-Charset: iso-8859-1,*,utf-8
Accept-Encoding: gzip
Accept-Language: en
Connection: Keep-Alive
Host: localhost
Pragma: no-cache
User-Agent: Mozilla/4.5 [en] (WinNT; I)
%response
HTTP/1.0 302 Moved
%stdout
Status: 302 Moved
Location: http://localhost/index.html

%% [Fri Jun 25 08:51:26 1999] GET /cgi-bin/CGIexam.pl HTTP/1.0
%% 500 c:/program files/apache group/apache/cgi-bin/cgiexam.pl
%request
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, image/png, */*
Accept-Charset: iso-8859-1,*,utf-8
Accept-Encoding: gzip
Accept-Language: en
Connection: Keep-Alive
Host: niu
User-Agent: Mozilla/4.5 [en] (WinNT; I)
%response
%stderr
syntax error at ConfigIO.pm line 36, near "//local"
syntax error at ConfigIO.pm line 37, near "//print"
```

```
%% [Fri Jun 25 09:02:24 1999] GET /cgi-bin/CGIexam.pl HTTP/1.0
%% 500 c:/program files/apache group/apache/cgi-bin/cgiexam.pl
%request
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, image/png, */*
Accept-Charset: iso-8859-1,*,utf-8
Accept-Encoding: gzip
Accept-Language: en
Connection: Keep-Alive
Host: niu
Pragma: no-cache
User-Agent: Mozilla/4.5 [en] (WinNT; I)
%response
?Error in opening read pipe \\.\pipe\Server2CGI930272542 in ConfPipe.
```

对上面的四条信息做简单解释：

1. **没有找到**/cgi-bin/status_report.exe **程序。**
2. **发生服务器内部错误，实际上是返回的输出结果格式不正确。**
3. CGI **程序执行出错，在涉及的** perl **程序库中有语法错误。**
4. CGI **程序执行遇到错误，其输出的错误提示信息被认为不符合** CGI **的输出内容格式。该输出的错误提示被放在了**%response **后。建议读者在编制** CGI **程序时可以用此方式读取** CGI **程序自己发现异常时输出的提示信息。**

7.7.5. mod_cgi 的实现

cgi_handler 方法

```
cgi_handler(request_rec *r)
```

本方法为 mod_cgi **中的核心请求处理方法，其内部处理过程如图** 7.9 **所示：**

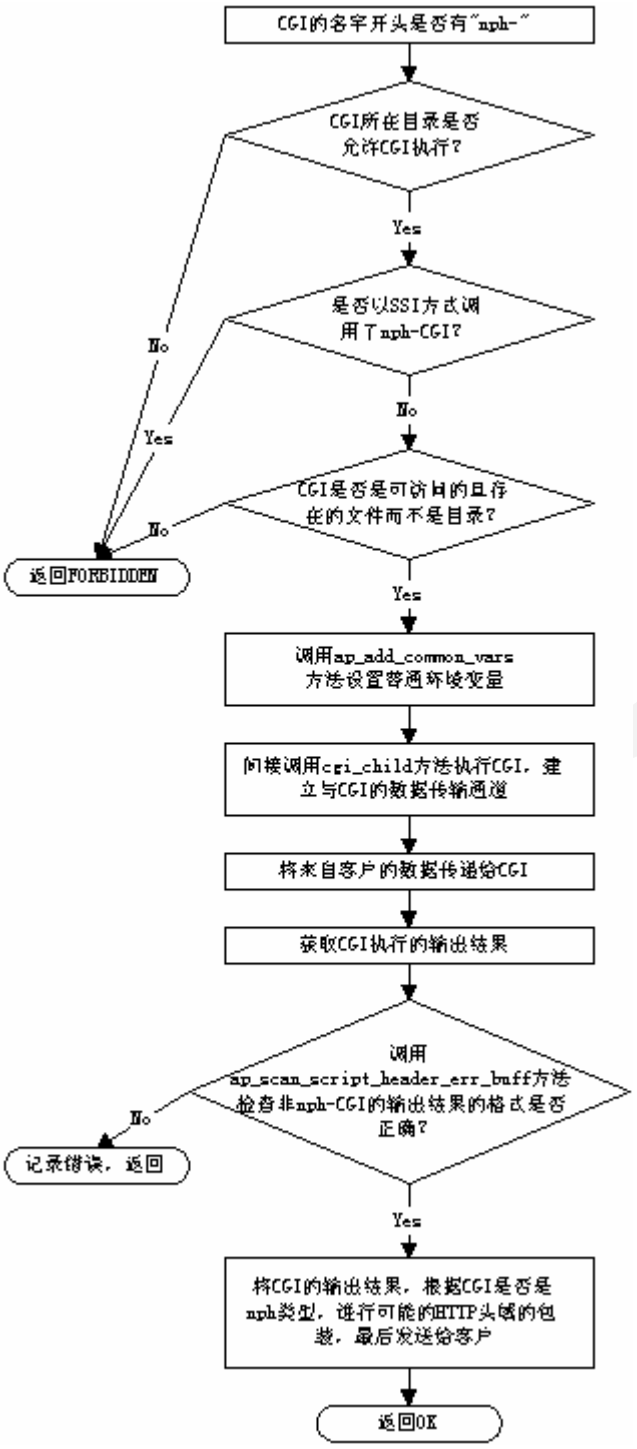


图 7.9

ap_bspawn_child 方法

```
ap_bspawn_child(pool *p, int (*func) (void *, child_info *), void *data, enum
kill_conditions kill_how, BUFF **pipe_in, BUFF **pipe_out, BUFF **pipe_err)
```

ap_bspawn_child **方法被** cgi_handler **方法调用来创建一进程来执行** CGI **程序，其中建立了三个管道** script_in（pipe_out)、script_out（pipe_out)、script_error（pipe_error)，**以完成**

服务器与 CGI 间的数据传输。失败时，记录错误，返回 HTTP_INTERNAL_SERVER_ERROR。管道建立的 I/O 关系如表 7.4 和图 7.10 所示：

表 7.4

cgi_handler 中的称谓	ap_spawn_child 中的称谓
script_in	pipe_out
script_out	pipe_in
script_error	pipe_error

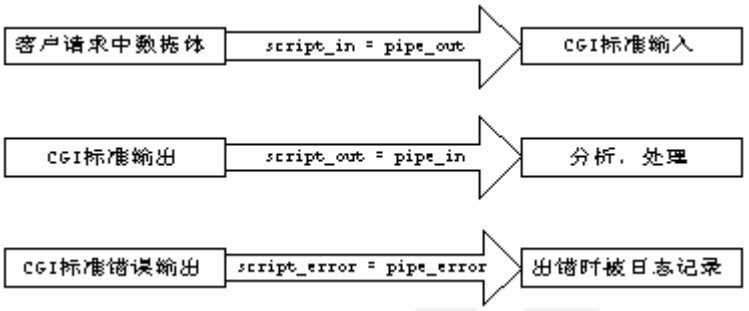


图 7.10

cgi_child 方法

```
cgi_child(void *child_stuff, child_info *pinfo)
```

cgi_child 方法完成 CGI 程序的初始化、执行，并返回对应的进程标识。

cgi_child 方法首先调用 ap_add_cgi_vars 完成环境变量的设置，然后将经过 ap_create_environment 处理的环境变量通过 ap_call_exec 传给启动的 CGI。

执行脚本是通过调用 ap_call_exec 来完成的，最后返回进程标识。

ap_call_exec 方法

```
ap_call_exec (request_rec *r, child_info *pinfo, char *argv0, char **env, int
shellcmd)
```

ap_call_exec 方法执行 r->filename 中含有的 CGI 程序。

该方法首先确定 CGI 的类型，确定其是一个可直接执行的程序还是一个需要某种解释器的脚本。若无法确定则记录错误，返回-1。

其中调用了 http_core.c 中的 ap_get_win32_interpreter 方法。

```
ap_get_win32_interpreter(const  request_rec *r, char*  ext, char**
interpreter )
```

根据传入的文件名后缀，确定执行对应文件所需的解释器。其路径放在 interpreter 中。调用 get_interpreter_from_win32_registry 具体实现。

```
get_interpreter_from_win32_registry(pool *p, const char* ext)
```

根据传入的文件名后缀，确定执行对应文件所需的解释器所在的绝对路径。

方法是：取得\HKEY_LOCAL_MACHINE\ SOFTWARE\Classes\中指定扩展名对应的注

册值，该值是一个对该类文件的描述串，同时它也是\HKEY_LOCAL_MACHINE\SOFTWARE\Classes\中注册目录项，在它下面的 SHELL\OPEN\COMMAND\中记录着打开该类文件所需的可执行程序的绝对路径。

最后将该绝对路径返回（其中剪去了最后附带的%1 串）。

log_scripterror 方法和 log_script 方法

```
log_scripterror(request_rec *r, cgi_server_conf * conf, int ret, int show_errno, char *error)
```

log_scripterror 方法将 CGI 执行错误记入日志。

```
log_script(request_rec *r, cgi_server_conf * conf, int ret, char *dbuf, const char *sbuf, BUFF *script_in, BUFF *script_err)
```

log_script 方法将 CGI 输出格式错误等信息记入日志，此时 CGI 已执行了。

第8节 access 模块

本模块在 mod_access.c 文件中实现，缺省便被包含在服务器程序中。它提供基于客户端域名或 IP 地址的访问控制，通常与身份认证模块协同使用。

7.8.1. mod_access 的实现机制

本模块定义了如下的数据结构来表达配置信息。

```
enum allowdeny_type {//表达访问控制范围的表示方法类型
    T_ENV,   //域名
    T_ALL,   //All（指全部所有主机）
    T_IP,    //全 ip 地址
    T_HOST,  //全主机名
    T_FAIL   //三级 ip
};
typedef struct {
    int limited;
    union {
         char *from;             //记录 from 后所跟的字符串
         struct {
            unsigned long net;       //表示网络地址
             unsigned long mask; //表示掩码
         } ip;
    } x;
    enum allowdeny_type type;        //表达上述联合结构中何域有效
} allowdeny;                         //allow 或 deny 配置信息结构
```

其中 limited 用来记录针对请求方法的访问控制信息，该整数的低 16 位分别对应 16 种

请求方法，为 1 表示后续的控制信息对该方法适用，否则不适用。

```
typedef struct {
    int order[METHODS]; // 对应针对各请求访问方法的 allow 与 deny 配置处理顺序
                        //的编号，对不同的方法（如 GET、PUT）取值未必相同
    array_header *allows;    //allow 参数值
    array_header *denys; //deny 参数值
} access_dir_conf;           //目录访问配置结构
```

其中，allows 和 denys 均是单元类型为 allowdeny 的数组，每个单元可以表示一条 allow 或 deny 配置信息。

- 与初始化相关的方法有：

```
create_access_dir_config
```

- 与配置命令处理相关的方法有：

```
order
is_ip
allow_cmd
```

- 与请求处理有关的方法有：

```
in_domain
find_allowdeny
check_dir_access
```

create_access_dir_config 方法

```
create_access_dir_config(pool *p, char *dummy)
```

本方法创建并初始化 access_dir_conf 结构，返回指向该结构的指针。

order 方法

```
order(cmd_parms *cmd, void *dv, char *arg)
```

本方法处理配置命令 order，根据 arg 表示的 order 配置值，填充 dv 所指的相应内容，dv 为一 access_dir_conf 结构的指针。

若 arg 参数内容为 order 配置命令的合法的三种形式，则记录在 access_dir_conf 结构中，并返回空指针；否则返回“ unknown order ”字串。

is_ip 方法

```
is_ip(const char *host)
```

本方法判断 host 字符串是否为一 ip 地址，是则返回 1，否则返回 0。在 allow_cmd()方法中被调用。

具体处理过程为：

```
while ((*host== '.') || ap_isdigit(*host))//句号或数字，则继续下一个字符
host++;
    return (*host == '\0');//若循环以结束符退出，返回真，否则为假
```

allow_cmd 方法

```
allow_cmd(cmd_parms *cmd, void *dv, char *from, char *where)
```

本方法用来处理 allow 及 deny 配置命令，其中 dv 是一 access_dir_conf 结构指针，from 是配置行中配置命令（如 allow 或 deny）后以空格分隔的第一个字符串，通常应该是“ from ”，where 为配置参数的值（某种形式的 IP 地址）。若配置格式错，则返回出错信息字符串，否则返回 NULL。

该方法首先检查 from 参数的值是否为合法的'from'，否则返回

```
"allow and deny must be followed by 'from'"
```

信息串。然后对 where 参数值进行解析，区分其是属于环境变量类型，还是“ all ”形式，或者是 IP 地址形式。若是 IP 地址形式则还要进行严格的检查，格式有问题时，返回出错信息串。最终参数类型的信息记录到 access_dir_conf 结构的 denys 或 allows 数组的 allowdeny 结构的单元中。

in_domain 方法

```
in_domain(const char *domain, const char *what)
```

本方法用来判断 what 所存放域名是否在 domain 所指的域中。若 what 属于 domain 域，则返回 1，否则返回 0。本方法在 find_allowdeny 方法中被使用。

首先判断 domain 是否与 what 的后缀匹配，否则可知 what 一定不属于 domain；若匹配，为了消除类似 nogood.com 与 good.com 的错误匹配的情况，还需看匹配的 what 的后缀是否是完整的域名，即前一字符是否为点号，是则完全匹配，否则不匹配。

find_allowdeny 方法

```
find_allowdeny(request_rec *r, array_header *a, int method)
```

本方法根据一系列 allow 配置信息或一系列 deny 配置信息，判断请求者是否应被允许以方法 method 访问其访问资源所在的目录。其中，a 指向一系列 allow 或 deny 命令的信息，为数组结构。若用户在允许或拒绝的范围内则返回 1，否则返回 0。或者简单地说，本方法用来判定请求者是否在 allow 或 deny 的 from 范围内。它被 check_dir_access 方法调用。

该方法针对每一条 allow 或 deny 循环进行判定，并跳过请求方法不在针对的方法范围内的情况。一旦判定在指定范围内，则退出循环返回 1，否则一直继续，最终返回 0。

check_dir_access 方法

```
check_dir_access(request_rec *r)
```

本方法判断该请求提出者是否允许访问其申请的目录，并利用了访问配置信息（全局的和相应目录的）。允许则返回 OK，否则返回 FORBIDDEN。

本方法完全按照 order、allow 和 deny 配置命令的应用规则进行判定。

首先是根据 order 的配置，设置一个初始值（具体参阅第 5 章第 2 节order 命令的介绍）。

然后调用 find_allowdeny 方法依次应用 allow 配置或 deny 配置，判定用户是否被允许。

而且，若 Satisfy 配置为 all 或者根本就没有定义 require 命令，则本模块已判定用户应被拒绝的结果就是最终判决，其它模块的判定对此毫无影响，则将拒绝的信息记入日志。最终返回判定结果。

第9节 auth 模块

本部分介绍与用户身份认证相关的 HTTP 基本原理及如下 Apache 模块：

- mod_auth：
  提供基于正文文件的用户身份认证，在 mod_auth.c 中定义。
- mod_auth_anon：
  提供允许匿名（"anonymous"）用户对要求身份认证区域的访问。它的实现在 mod_auth_anon.c 文件中，缺省时并没有编译到服务器软件中。它只在 Apache 1.1 或更高版本中可用。本模块以与匿名访问 ftp 站点相同的方式对访问进行控制；即有一个特殊的用户名'anonymous' 和一个 email 地址作为口令。用户提供的 email 地址是可以记入日志的。
  若与其它基于数据库的访问控制方法结合使用，则可在实现高效用户跟踪和根据用户 profile 文件定制的同时，仍对非注册（'unregistered'）用户开放。使用基于身份认证的用户跟踪的一大优点在于：与设置 cookies 和特殊的 URL 前缀/后缀不同，它完全是不依赖于浏览器的，并且使用户可以共享 URL 地址。
- mod_auth_db：
  提供基于 Berkeley DB 文件的用户身份认证，其实现存在于 mod_auth_db.c 文件中，缺省时没有编译进服务器中。在某些支持 DB 而不支持 DBM 的系统中，可以用此模块替代 mod_auth_dbm 模块。该模块只在 Apache 1.1 或更高版本中可用。
  在某些 BSD 系统中（如 FreeBSD 和 NetBSD），dbm 可以自动映像为 Berkeley DB。这时你既可以使用 mod_auth_dbm，也可以使用 mod_auth_db。而在其它系统中，你要想使用 DB 库，必须首先安装上它。请访问 http://www.sleepycat.com/获取相关信息。
- mod_auth_dbm：
  提供了使用 DBM 文件进行用户身份认证的手段。其实现存在于 mod_auth_dbm.c 文件中，缺省时没有编译进服务器中。
- mod_digest：
  提供了基于 MD5 摘要认证机制的用户身份认证。其实现存在于 mod_digest.c 文件中，缺省时未被编译进服务器程序中，且只在 Apache 1.1 及以后版本中可用。

它们分别定义在 mod_auth.c、mod_auth_anon.c、mod_auth_db.c、mod_auth_dbm.c、mod_digest.c 文件中。

7.9.1. HTTP 认证机制

HTTP 中，为了对访问用户进行权限控制，引入了认证机制。

认证机制概述

HTTP 提供了一种简单的询问/响应认证机制（challenge-response authentication mechanism），可供服务器询问用户请求、用户提供认证信息。该机制使用一种可扩展的、大小写不敏感的标号（token）来标识（识别）该认证机制，其后是用逗号分开的各条属性值，各属性带有实现该认证机制所必须的参数。

```
auth-scheme    = token
auth-param     = token "=" quoted-string
```

401（未授权）响应消息是原始服务器用来询问一客户端程序的身份和权限的的。该响应必须包含一 WWW-Athenticate 头域，该域至少包含适用于被请求资源的质询。

```
challenge     = auth-scheme 1*SP realm *( "," auth-param )
realm         = "realm" "=" realm-value
realm-value    = quoted-string
```

域（realm）属性（大小写不敏感）是所有基于询问的认证机制所必须的。realm 属性的值（是大小写敏感的），与被访问服务器的根（root）URL 一起，可确定一保护空间（protection space）。这样通过定义不同的域，就可将一台服务器上的受保护资源分为一系列的保护空间，每一空间有自己的认证机制和认证数据库。域属性值为一字符串，通常由原始服务器指定的。

客户端程序若想从服务器那里获得授权，通常是在第一次请求的响应的状态码为 401 或 411（未定义 Content-Length 头域）后，在后续请求中再加入 Authorization 头域实现的。但这并不是必然的情况，可以想象的是，客户程序在第一次请求时就可能且可以发送 Authorization 头域。该 Authorization 头域由包含针对指定域的客户程序身份认证信息的证书（credentials）构成。

```
credentials    = basic-credentials
                       | auth-scheme #auth-param
```

上述证书的适用范围是由保护空间决定的。若先前的请求已经过认证，则一定时期内，同一证书可在对同一保护空间的请求中再用，具体的时间长短，由认证机制、参数、或使用者的配置决定。除非认证机制专门定义，否则一个保护空间是不能超出其服务器范围的。

如果该服务器不接受（同意）随某一请求送来的证书，则应该返回 403（禁止访问）响应。该响应必须包含一 WWW-Authenticate 头域，其中包含适用于被请求资源的质询（可能是新的）和对拒绝的说明。

在涉及代理服务器的 HTTP 请求/响应中，代理服务器必须原封不动地转发 WWW-Authenticate 及 Authorization 头域，其工作对客户端程序来说应是透明的。注意这里的认证仍是针对 WWW 服务器的，与 HTTP/1.1 中阐述的代理服务器对其服务的访问认证不同，后者允许一客户通过代理认证及代理授权头，来与代理之间互传认证信息。

需要说明的是，HTTP 协议并不要求身份认证一定使用上述简单的询问/响应机制。您完全可以使用其它机制，如在传输层的加密或消息封装，使用附加的说明认证信息的头域。详细内容参阅相关资料，本书不介绍这些机制。

询问/响应机制可分为两种：基本认证机制和摘要认证机制，它们分别由 HTTP1.0 和 HTTP1.1 引入。

基本认证机制

基本（basic）认证机制是建立在客户程序必须用每一域的用户 ID 及密码来使自己获得

身份认可这样一种模型上的。域值应该被认为是一个密文字符串（opaque string），该字符串只用于与该服务器上的其它域进行区分。服务器只有在证实提供的用户 ID 及密码在请求 URL 所属保护空间的合法性后才会服务请求。

当收到对保护空间的 URI 的未授权请求时，服务器应给出如下的询问：

```
WWW-Authenticate: Basic realm="WallyWorld"
```

这里，"WallyWorld"是服务器指派的标识该请求 URI 所属保护空间的字符串。

为了获得授权，客户端程序应发送经 64 基方法编码的身份证书，原文应包含用户 ID 及密码（以“ :”字符分隔）：

```
basic-credentials = "Basic" SP basic-cookie
basic-cookie   = <base64 encoding of user-pass,
                 except not limited to 76 char/line>
user-pass   = userid ":" password
userid      = *<TEXT excluding ":">
password    = *TEXT
```

用户 ID（userid）可为大小写敏感的。

如客户程序希望发送 userid （用户 ID）——"Aladdin"和 password（密码）——"open sesame"，则应发送如下头域：

```
Authorization: Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==
```

其中“ QWxhZGRpbjpvcGVuIHNlc2FtZQ== ”是将字符串“ Aladdin: open sesame ”进行 64 基编码后得到的。

摘要认证机制

虽然通过基本访问认证满足了一定程度的安全要求，但由于用户名和密码是明文传输的（64 基编码对水平一般的黑客来说也不能算是障碍，因为加密/解密一目了然），所以必须要有一种新的机制来提供更高程度的安全，因此产生了摘要认证机制。

尽管摘要访问认证机制与基本认证机制一样，服务器和客户双方都保存一个共同的口令，但不同之处在于，摘要认证机制中不需要传送口令的明文，而这恰是基本认证机制的最大问题。

但需要说明的是，摘要认证机制并非最终解决方案，比如其中传输的对象内容就根本没有加密。我们只能说它的作用仍很弱，只不过是弥补了基本认证机制的重大缺陷。

HTTP 的摘要认证机制在 HTTP1.1 中引入，但单独由 RFC 2069 进行描述。象基本访问认证一样，摘要访问认证也基于简单询问/响应。

1. WWW-Authenticate 头域

在摘要认证机制中，如果服务器收到了对受保护对象的访问请求，而该请求中并没有提供满足条件的 Authorization 头域，则服务器应返回 401（未授权）响应并包含 WWW-Authenticate 头域。WWW-Authenticate 头域的具体定义如下：

```
WWW-Authenticate    = "WWW-Authenticate" ":" "Digest"
                      digest-challenge
digest-challenge    = 1#( realm | [ domain ] | nonce |
                    [ digest-opaque ] |[ stale ] | [ algorithm ] )
realm               = "realm" "=" realm-value
realm-value         = quoted-string
```

```
domain              = "domain" "=" <"> 1#URI <">
nonce               = "nonce" "=" nonce-value
nonce-value         = quoted-string
opaque              = "opaque" "=" quoted-string
stale               = "stale" "=" ( "true" | "false" )
```

其中，

realm：**是一个字符串，用来告诉用户应该使用什么用户名和口令，从而可以避免由于有多个名字同时存在而使用户会手足无措的情况。该串应包含当前进行认证的服务器的名字，最好还能体现有访问权的用户集，比如“**registered_users@gotham.news.com**”。该串可用引号括起来以便可以使用空格。**

domain：**是用逗号分隔开的一系列** URI **地址。目的是要告诉客户端同样的认证信息可以适用的** URI **集。列出的** URI **地址可以分布在不同的服务器上，而若没有** domain **项的设置，则缺省的** URI **集是当前响应服务器上的所有** URI **地址。**

nonce：**是服务器给出的一个数据字符串，它在产生** 401 **响应时创建，同时必须满足唯一性，即与其它时刻产生的值不能相同。**HTTP **协议建议该串是经** 64 **基编码的或是十六进制的。具体的** nonce **值的确定是依赖于实现的，产生该值的算法决定了实现的好坏。**HTTP **协议中建议使用如下方法：**

```
H(client-IP ":" time-stamp ":" private-key )
```

其中 client-IP **是指提出亲请求的客户端的数字** IP **地址，**time-stamp **是服务器取的一个时间值，**private-key **则是只为服务器所知的值。这样产生的** nonce **值传给客户端，在其重新提出请求时，必须携带此** nonce **值，而服务器则重新计算** nonce **值，并与送来的值进行比较，若不匹配则拒绝该请求。用这种方法，服务器便可以控制** nonce **值的时效，避免来自同一** IP **地址的客户多次使用同一** nonce **值来达到多次访问的目的，这实际上就使黑客通过窃取合法用户的** nonce **值来伪装自己的梦想完全破灭。**

opaque：**是一个数据字符串，由服务器给出，客户必须原封不动地返回之。**HTTP **协议中建议该串是** 64 **基编码的或是十六进制数据。**

stale **：是一个标志，指示客户的先前的亲请求由于给的** nonce **值已失效而被拒绝。当** stale **值为** TRUE**（无论是大写还是小写）时，客户端可以不再提示用户输入用户名和口令，而仅重新进行加密计算并提出新请求；服务器只有在接收到的请求中包含无效的** nonce **值但对应的摘要却正确时（这说明用户知道正确的用户名和口令），才在响应中设置值为** TRUE**。**

algorithm：**是一个字符串，指出使用的摘要与校验和生成算法。若没有给出，则缺省值是** MD5 **算法。这里，我们将用密钥** secret **对数据** data **进行摘要的生成，表示为** KD(secret, data)**，而将对数据** data **施加校验和算法得到的校验和字符串表示为** H(data)**。对** MD5 **算法来说就是：**

```
H(data) = MD5(data)
KD(secret, data) = H(concat(secret, ":", data))
```

即摘要是密钥和数据经“:**”连接后进行** MD5 **加密的结果。**

2. Authorization **请求头域**

客户在第一次请求被拒绝后，若重发请求，则需包含 Authorization **头域，具体定义如下：**

```
Authorization       = "Authorization" ":" "Digest" digest-response
digest-response     = 1#( username | realm | nonce | digest-uri |
                          response | [ digest ] | [ algorithm ] |
```

```
                    opaque )
username           = "username" "=" username-value
username-value      = quoted-string
digest-uri         = "uri" "=" digest-uri-value
digest-uri-value    = request-uri        ; As specified by HTTP/1.1
response           = "response" "=" response-digest
digest            = "digest" "=" entity-digest
response-digest     = <"> *LHEX <">
entity-digest      = <"> *LHEX <">
LHEX               = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" |
```

其中的 response-digest **和** entity-digest **分别表示对应值的编码值，其计算方法如下：**

```
response-digest    =
     <"> < KD ( H(A1), unquoted nonce-value ":" H(A2) > <">
A1           = unquoted username-value ":" unquoted realm-value
                                       ":" password
password      = < user's password >
A2           = Method ":" digest-uri-value
```

需要说明的是，这里的 username-value **是一个带引号的字符串，在生成** A1 **时引号应被去掉。比如，若** Authorization **头域中包含如下内容：**

```
username="Mufasa", realm="myhost@testrealm.com"
```

而用户 Mufasa **的口令为“** CircleOfLife **”，则** H(A1) **就变成了** H (Mufasa : myhost@testrealm.com : CircleOfLife)**，其中的引号被去掉。**

上面的 Method **为** HTTP **请求方法，**request-uri **值是请求行中给出的** Request-URI **地址。**

可选的 digest **域是实体内容和相关实体头域的摘要，它可以有效地确保请求和响应的完整性。对请求来说，可以是** POST **或** PUT **数据；对响应来说，则是返回的文档。**digest **域的值——**entity-digest **由如下方法确定：**

```
entity-digest = <"> KD (H(A1), unquoted nonce-value ":" Method ":"
                      date ":" entity-info ":" H(entity-body)) <">
      ; format is <"> *LHEX <">
date = = rfc1123-date
entity-info = H(
        digest-uri-value ":"
        media-type ":"
        *DIGIT ":"
        content-coding ":"
        last-modified ":"
        expires
        )
last-modified   = rfc1123-date
expires        = rfc1123-date
```

如上所示，entity-info **的计算包含了请求实体的** URI **地址及相关头域——**Content-type、Content-length、Content-encoding、Last-modified **和** Expires**。这些头域在传输中不应被可能存在的代理服务器所篡改。**

注意，并非所有的请求都包含上面列出的头域，不存在的头域变为空即可，但作为分隔符的“:”则保留，

3. AuthenticationInfo 头域

当认证成功时，服务器可以（但不必须）发送给客户一个 Authentication-info 头域，指示服务器希望与客户就成功认证的信息（比如实体摘要与用于下一次交互的新的 nonce 值）进行交流。Authentication-info 头域包含两个域：digest 和 nextnonce。但两个都是可选的。

```
AuthenticationInfo = "Authentication-info" ":"
                                1#( digest | nextnonce )
nextnonce      = "nextnonce" "=" nonce-value
digest = "digest" "=" entity-digest
```

digest 可以被客户用来验证接收到的实体是否在传输中被更改。

nextnonce 参数值则指示服务器希望客户在下一次提出认证请求时使用该 monce 值。

4. 摘要操作

在接收到 Authorization 头域后，服务器根据客户给出的用户名去取对应的口令，然后进行与客户端相同的摘要计算，并将结果与客户给出的值做比较，从而确定用户的合法性。

注意，在服务器上没有必要知道用户的正文口令，实际上只要知道 H(A1)值，Authorization 头域的合法性就完全可以确定。

客户端可以保存对应域的用户名、口令和 nonce 值，这样在将来对某个域提出请求时，可直接给出合法的 Authorization 头域。服务器可以接受老的 Authorization 头域，即使 nonce 值已过时。若服务器较为“严厉”的话，完全可以返回一个 401 响应，并给出一个新的 nonce 值，指示客户端应重新发送请求。若响应中给出 stale=TRUE，则是明确提示客户端应使用新的 nonce 值提出请求，且不必在让用户重新输入用户名和口令。

5. 示例

如下给出了对服务器上的受保护文档进行请求的过程。被请求文档为：http://www.nowhere.org/dir/index.html。客户和服务器均知道合法的用户名和口令分别是：Mufasa 和 CircleOfLife。

在客户提出第一次请求时，没有给出 Authorization 头域，所以服务器给出如下响应：

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Digest realm="testrealm@host.com",
                    nonce="dcd98b7102dd2f0e8b11d0f600bfb0c093",
                    opaque="5ccc069c403ebaf9f0171e9517f40e41"
```

客户端接收到响应后，提示用户输入用户名和口令，然后发送新请求，其中包含如下 Authorization 头域：

```
Authorization: Digest username="Mufasa",
                realm="testrealm@host.com",
                nonce="dcd98b7102dd2f0e8b11d0f600bfb0c093",
                uri="/dir/index.html",
                response="e966c932a9242554e42c8ee200cec7f6",
                opaque="5ccc069c403ebaf9f0171e9517f40e41"
```

代理认证

上面我们是给出了 WWW 服务器的认证方法。还有一个问题是，若服务过程中使用了代理服务器，则如何实现对请求代理服务的认证。实际上，上述的摘要认证机制同样适用于用户与代理之间、代理与代理之间、代理与目的服务器之间的认证，具体是使用 HTTP/1.1 给出的 Proxy-Authenticate 和 Proxy-Authorization 头域。

代理认证的过程与上述客户与目的服务器之间的认证很相似。在接收到需进行认证的请求后，代理/服务器给出“ HTTP/1.1 401 Unauthorized ”响应，并附加如下形式的 Proxy-Authenticate 头域：

```
Proxy-Authentication = "Proxy-Authentication" ":" "Digest"
digest-challenge
```

这里的 digest-challenge 与上文中的出现意义相同。客户/代理接收到上述响应后，重发带有如下 Proxy-Authenticate 头域的请求：

```
Proxy-Authorization = "Proxy-Authorization" ":" digest-response
```

这里的 digest-response 与上文意义相同。若认证成功，服务器可能给客户返回如下形式的 Proxy-Authentication-info 头域：

```
Proxy-Authentication-info = "Proxy-Authentication-info" ":" nextnonce
```

这里的 nextnonce 与上面 Authentication-info 头域中的 nextnonce 意义相同。

7.9.2. MD5 算法

本节集中讲述在 Apache 服务器的身份认证和 cache 生成（属于 mod_proxy 部分）中起重要作用的 MD5 算法。MD5 是一种单向散列算法，它不像 CRC 校验码，很难找到两份不同的输入具有一样的输出结果。

该算法在 RFC 1321 中有详细的介绍，并且有参考的 C 语言实现。MD5 的提出者是 Ron Rivest。

单向 hash 函数

一个典型的单向 hash 函数，输入的任意长的消息，将产生一定长的 hash（乱码）。对给定的 hash 值，在计算上不可能找出具有该 hash 值的消息，事实上，从一个消息的 hash 值，得不出一个消息的有用信息，甚至是一比特也不能。对一些单向 hash 函数，要根据两个产生相同 hash 值的函数产生的 hash 值鉴别这两个消息，在计算上也是不可能的。

一个单向 hash 函数可以是私有的，也可是公开的，象加密函数一样。MD5 算法便是一种这样的函数。以下便是公开单向 hash 函数的的一个应用：大多数公开密钥数字签名系统相对说来都较慢，为了对一个长消息签名，其耗时是用户不能忍受的。解决方法便是：先计算出消息的单向 hash 函数值，而后对 hash 值签名。

单向 hash 函数也叫消息摘要函数。目前有多种消息摘要函数，MD5 用得最广。

MD5 算法的描述

设我们有 b 比特长的消息作为输入，欲求其摘要。这里 b 为任意非负整数，可为零，不必为 8 的整数倍，可为任意大。将该消息记为：m_0 m_1 ... m_{b-1}

计算步骤如下：

步骤 1：追加填充比特

将该消息（在末尾）填充，使其填充后的长度模上 512 的结果是 448，即是将其扩充为 512 的整数倍少 64 比特。注意，算法要求，即使消息长度已是 n 倍 512 减 64，填充必须进行。要追加的比特总数为 1 到 512 比特之间。

步骤 2：追加长度

将 b 的 64 位形式的表达式追加到前一步结果末尾，若 b 大于 2^64（极不可能）的情况下，只用低端 64 位。注意，是按低字（32 比特）序将其当作两个 32 比特字追加的。

到此，消息长度正好是 512 比特的整数倍。即是，正好是 16 个字（32bits）长的整数倍。我们结果消息表示为 M[0…N-1]，这里，N 是 16 的整数倍。

步骤 3：初始化 MD 缓冲区

为计算消息摘要，需用四字缓冲区（A、B、C、D）。这里，A、B、C、D 均为 32 比特的寄存器。这些寄存器被初始化为如下 16 进制值（低字节序）：

```
word A: 01 23 45 67
word B: 89 ab cd ef
word C: fe dc ba 98
word D: 76 54 32 10
```

步骤 4：，分块处理消息，每块 16 字（256bits）长

首先，定义四个辅助函数，该四个函数均使用三个 32 比特字为参数，并产生一个 32 比特字为输出，其定义为：

```
F(X,Y,Z) = XY v not(X) Z（v 表示按位或，XY 表示按位与）
G(X,Y,Z) = XZ v Y not(Z)
H(X,Y,Z) = X xor Y xor Z
I(X,Y,Z) = Y xor (X v not(Z))
```

```
In each bit position F act as a conditional: if X then Y else Z.
```

由于 XY 和 not（X）Z 决不会在同一位置为 1，因而，函数 F 也可以用“ + ”号来定义，而不用“ v ”，有趣的是，您可以注意到，如果 X、Y、Z 的各比特独立不相关，则 F（X，Y，Z）的各比特也独立不相关。

函数 G，H，I 与 F 相似，也是由 X，Y，Z 比特对位计算产生输出，如此方法，则若 X，Y，Z 各比特独立，G（X，Y，Z），H(X,Y,Z), 和 I(X,Y,Z)的个位也相互独立无关。注意，函数 H 是其输入的按位"xor"（异或）或"parity"（偶）函数。

这一步用到一具有 64 元素的表 T[1 ... 64]，该表由正弦函数构造产生。T[i]表示该表的第 i 个元素, T[i]为 4294967296 与 abs(sin(i))乘积的整数部分,i 为弧度数。

然后，如下计算：

```
/* 处理每一 16 字长的块 */
For i = 0 to N/16-1 do

 /* 将块 i 拷贝到 X. */
 For j = 0 to 15 do
```

```
      Set X[j] to M[i*16+j].
    end /* 结束 j 循环*/

    /* 将 A 存为 AA, B 存为 BB, C 为 CC, D 为 DD */
    AA = A
    BB = B
    CC = C
    DD = D

    /* Round 1. */
/* 设[abcd k s i]表示操作：
a = b + ((a + F(b,c,d) + X[k] + T[i]) <<< s)，其中<<<s 表示左环移 s 位*/
    /* 做以下 16 步计算. */
    [ABCD  0  7  1] [DABC  1 12  2] [CDAB  2 17  3] [BCDA  3 22  4]
    [ABCD  4  7  5] [DABC  5 12  6] [CDAB  6 17  7] [BCDA  7 22  8]
    [ABCD  8  7  9] [DABC  9 12 10] [CDAB 10 17 11] [BCDA 11 22 12]
    [ABCD 12  7 13] [DABC 13 12 14] [CDAB 14 17 15] [BCDA 15 22 16]

    /* Round 2. */
    /* 设[abcd k s i]表示以下操作：
        a = b + ((a + G(b,c,d) + X[k] + T[i]) <<< s). */
    /* 做以下各步：*/
    [ABCD  1  5 17] [DABC  6  9 18] [CDAB 11 14 19] [BCDA  0 20 20]
    [ABCD  5  5 21] [DABC 10  9 22] [CDAB 15 14 23] [BCDA  4 20 24]
    [ABCD  9  5 25] [DABC 14  9 26] [CDAB  3 14 27] [BCDA  8 20 28]
    [ABCD 13  5 29] [DABC  2  9 30] [CDAB  7 14 31] [BCDA 12 20 32]

    /* Round 3. */
    /* [abcd k s t]表示
        a = b + ((a + H(b,c,d) + X[k] + T[i]) <<< s). */
    /*做以下操作 16 步*/
    [ABCD  5  4 33] [DABC  8 11 34] [CDAB 11 16 35] [BCDA 14 23 36]
    [ABCD  1  4 37] [DABC  4 11 38] [CDAB  7 16 39] [BCDA 10 23 40]
    [ABCD 13  4 41] [DABC  0 11 42] [CDAB  3 16 43] [BCDA  6 23 44]
    [ABCD  9  4 45] [DABC 12 11 46] [CDAB 15 16 47] [BCDA  2 23 48]

    /* Round 4. */
    /*[abcd k s t]表示：
        a = b + ((a + I(b,c,d) + X[k] + T[i]) <<< s). */
    /* Do the following 16 operations. */
    [ABCD  0  6 49] [DABC  7 10 50] [CDAB 14 15 51] [BCDA  5 21 52]
    [ABCD 12  6 53] [DABC  3 10 54] [CDAB 10 15 55] [BCDA  1 21 56]
    [ABCD  8  6 57] [DABC 15 10 58] [CDAB  6 15 59] [BCDA 13 21 60]
    [ABCD  4  6 61] [DABC 11 10 62] [CDAB  2 15 63] [BCDA  9 21 64]
```

```
/* 然后完成以下加法(即将该块开始计算以前的值加到四个寄存器)。 */
A = A + AA
B = B + BB
C = C + CC
D = D + DD

end /* 结束 i 的循环 */
```

至此，步骤 4 结束。

步骤 5：输出

A, B, C, D 便是要输出的摘要。输出结果应以 A 的低字节开始，以 D 的高字节结束。

到此，MD5 算法的描述便结束。

MD5 在 cache 组织上的应用

在 Apache（1.3.6 版本）中的 proxy 中，采用 MD5 算法将要进行 cache 的文件的文件名 HASH 后得到 128 比特，再将二进制数进行文本化编码，得到一定长的字符串（对大小写不敏感文件系统，采用类似 MIME 中的 64 基编码的 32 基编码，得到 26 个字母的串），将此字符串，作为文件名保存。在查询时，先将文件名进行同样的转换，即可查找。

这里，主要是利用了 MD5 算法的结果是定长的特性，使得 cache 中的文件名为定长，从而使得对 cache 的管理方便、快捷。同时，其结果的唯一性，是该法有效的保证。

当然，MD5 转换及将其结果进行文本化编码，都带来了额外的开销。但是，MD5 算法及 32 基编码本身速度快，尤其是 cache 容量特别大的时候，其优势将会愈是明显。

MD5 在身份认证中的应用

PGP 加密中 MD5 的应用

在介绍 PGP 之前，首先介绍 RSA（Rivest-Shamir-Adleman）算法。

RSA 是一种基于大数不可能质因数分解假设的公匙体系。简单使它非常适合用来满足两个要求：保密性（Privacy）和认证性（Authentication）。地说就是找两个很大的质数，一个公开给世界，一个不告诉任何人。一个称为“公匙”，另一个叫“私匙”（Public key & Secret key or Private key）。这两个密匙是互补的，就是说用公匙加密的密文可以用私匙解密，反过来也一样。假设甲要寄信给乙，他们互相知道对方的公匙。甲就用乙的公匙加密邮件寄出，乙收到后就可以用自己的私匙解密出甲的原文。由于没别人知道乙的私匙所以即使是甲本人也无法解密那封信，这就解决了信件保密的问题。另一方面由于每个人都知道乙的公匙，他们都可以给乙发信，那么乙就无法确信是不是甲的来信。认证的问题就出现了，这时候数字签名就有用了。

在说明数字签名前先要解释一下什么是“邮件文摘”（messagedigest），简单地讲就是对一封邮件用某种算法算出一个能体现这封邮件“精华”的数来，一旦邮件有任何改变这个数

都会变化，那么这个数加上作者的名字（实际上在作者的密匙里）还有日期等等，就可以作为一个签名了。著名的 PGP 就是用一个 128 位的二进制数作为“邮件文摘”的，用来产生它的算法就是 MD5。

回到数字签名上来，甲用自己的私匙将上述的 128 位的“精华”加密，附加在邮件上，再用乙的公匙将整个邮件加密。这样这份密文被乙收到以后，乙用自己的私匙将邮件解密，得到甲的原文和签名，乙的 PGP 也从原文计算出一个 128 位的“精华”来再用甲的公匙解密签名得到的数比较，如果符合就说明这份邮件确实是甲寄来的。这样两个要求都得到了满足。

PGP 还可以只签名而不加密，这适用于公开发表声明时，声明人为了证实自己的身份（在网络上只能如此了），可以用自己的私匙签名。这样就可以让收件人能确认发信人的身份，也可以防止发信人抵赖自己的声明。还它可以防止发信人抵赖和信件被途中篡改，这一点在商业领域有很大的应用前途。

那么为什么说 PGP 用的是 RSA 和传统加密的杂合算法呢？因为 RSA 算法计算量极大在速度上不适合加密大量数据，所以 PGP 实际上用来加密的不是 RSA 本身，而是采用了一种叫 IDEA 的传统加密算法。我先解释一下什么叫传统加密，简单地说就是用一个密匙加密明文，然后用同样的密匙解密。这种方法的代表是 DES（US Federal Data Encryption Standard），也就是乘法加密，它的主要缺点就是密匙的传递渠道解决不了安全性问题，不适合网络环境邮件加密需要。IDEA 是一个有专利的算法，专利持有者是 ETH 和一个瑞士公司：Ascom-TechAG。非商业用途的 IDEA 实现不用向他们交纳费用。IDEA 的加（解）密速度比 RSA 快得多，所以实际上 PGP 是用一个随机生成密匙（每次加密不同）用 IDEA 算法对明文加密，然后用 RSA 算法对该密匙加密。这样收件人同样是用 RSA 解密出这个随机密匙，再用 IDEA 解密邮件本身。这样的链式加密这样的链式加密就做到了既有 RSA 体系的保密性，又有 IDEA 算法的快捷性。

MD5 算法的安全性

根据上面给出的 MD5 的算法可以知道：一次提出两个消息，要其具有同一摘要，需 2^64 次操作。任意提出一个消息，要其具有给定的摘要，其概率是 1/（2^128）。也即是用枚举法由摘要得出消息，需进行 2^128 次 MD5 算法的计算，更加上产生任意消息的算法时间，完全就是个天文时间。

7.9.3. 实现机制

如下给出了各个认证模块中处理方法的详尽描述和分析。

由于 mod_auth_db 和 mod_auth_dbm 模块与 mod_auth 模块相比，仅是在口令的存放方式上有所不同，其它机制则基本相同，所以下文中将此两模块的分析略去，读者可参考 mod_auth 模块。

MD5 摘要算法方法

为了实现MD5算法，定义了AP_MD5_CTX结构，该结构定义如下：

```
typedef struct {
```

```
        /* MD5算法缓冲区(ABCD)，其中UINT4是自定义无符号长整数 */
        UINT4 state[4];
        /* 已参与计算的比特数(必为2^64 的模)(低字节序) */
        UINT4 count[2];
        /*每次输入的字符的缓冲区*/
        unsigned char buffer[64];
    } AP_MD5_CTX;
```

MD5算法实现过程的可简单描述为：摘要算法以512比特为单位分块进行；计算时，先调用ap_MD5Init()初始化以上结构及缓冲区，然后将数据一次一块地送入ap_MD5Update()（每块不一定是512比特的整数倍）；其间，每次该函数处理结束时，buffer中保存着不足512比特的二进制数，待到要进行摘要计算的数送完后，调用ap_MD5Final()，进行MD5结束工作。

三个摘要算法方法为：

ap_MD5Init 方法

```
ap_MD5Init(AP_MD5_CTX *contex)
```

初始化contex内容，将ABCD置入初值,将count置0。

调用此方法前，需要申请一个AP_MD5_CTX结构，将其指针作为参数来使用。

ap_MD5Update 方法

```
ap_MD5Update(AP_MD5_CTX * context, const unsigned char *input, unsigned int
inputLen)
```

继续处理下一消息块，根据处理结果更新context 内容。其中input 为本次将要处理的无符号字符串快指针，inputLen是其长度。

ap_MD5Final 方法

```
ap_MD5Final(unsigned char digest[16], AP_MD5_CTX * context)
```

本方法完成结束操作,完成整个消息文摘算法的比特填充操作和长度填充操作并完成最后一次块计算，并将结果传给digest，同时将context归0。

本方法在digest中返回MD5算法最终结果。

MD5 用于认证的方法

如下方法为用户身份认证提供直接的支持。

ap_MD5Encode 方法

```
ap_MD5Encode(const char *pw, const char *salt, char *result, size_t nbytes)
```

将输入的密码 pw 及 salt 盐信息用基于 MD5 的加密算法编码产生 HASH 字符串。其中 result 为输出缓冲指针，nbytes 为其长度，一般设得较大（120）。

ap_validate_password 方法

```
ap_validate_password(const char *passwd, const char *hash)
```

验证用户送来的 passwd 密码是否与服务器配置文件保存的密码（以 hash 码形式保存）匹配。其中 passwd 是用户送来的密码，hash 是在服务器保存的用户密码。若匹配则返回 NULL，否则返回出错信息串。

MD5 算法的用户 API 方法

如下方法定义在 util_md5.c 文件中，它们依赖于上面列出的定义在 ap_md5c.c 中的低层方法。

ap_md5_binary 方法

```
ap_md5_binary(pool *p, const unsigned char *buf, int length)
```

本方法完成完整的 MD5 转换，它将 buf 字符串的值进行 MD5 摘要变换，并将结果转换为 16 基编码的 32 字符，并在末尾加'\0'字符串结束符。

具体的操作过程如下：

ap_MD5Init ()

ap_MD5Update ()

ap_MD5Final（hash，&my-md5）（结果存于 hash 中，16bytes）

调用 hex（）函数将 hash 中值按每 4 位取出，变为 0……f 之间的字符（即进行 16 基编码），存于 result 中（共 32 字符）

加'\0'于末尾

调用 ap_pstrdup（p, result）将结果保存于 p 中。

ap_md5 方法

```
ap_md5(pool *p, const unsigned char *string)
```

本方法只是对 ap_md5_binary 方法的进一步封装。

ap_md5contextTo64 方法

```
ap_md5contextTo64(pool *a, AP_MD5_CTX * context)
```

将 context **中的** MD5 **计算结果完成最后结束工作，并将结果进行** 64 **基字符化编码，得** 22 **字符，并添**'\0'**、** '='**、** '='**于末尾得到一个长为** 25 **的字符串。**

操作过程如下：

```
unsigned char digest[18]
encodeddigest=ap_pcalloc(a, 25 * sizeof(char))分配 25 字符空间
ap_MD5Final(digest, context)结束 context 的 MD5 计算，
结果于 digest（前 16 字符）中
digest 后两字符赋值为 0
将 digest 中的二进制数按 64 基字符化编码，结果（22 字符）放于
encodedDigest 中，并将其末三字符依次置为'\0'，'\0'，'\0'
返回 encodedDigest 头指针。
```

ap_md5digest 方法

```
ap_md5digest(pool *p, FILE *infile)
```

本方法将文件 infile **的内容进行** MD5 **摘要计算并** 64 **基编码为** 25 **字符。**

具体过程如下：

```
AP_MD5_CTX context；
unsigned char buf[1000]；
ap_MD5Init(&context)；
while (fread (buff,1,sizeoff (buf) ,infile))；
ap_MD5Update ()；
rewind(infile)；
```

返回 ap_md5contextTo64(p, &context)。

mod_auth 中实现的方法

mod_auth **模块中定义了如下结构来记录与认证有关的配置信息，来方便方法调用中信息的传递：**

```
auth_config_struct {
    //用户帐号文件名，对应 AuthUserFile 配置命令。
    char *auth_pwfile;
    //组帐号文件名，对应 AuthGroupFile 配置命令。
    char *auth_grpfile;
    int auth_authoritative;
} auth_config_rec;
```

其中 auth_authoritative **可取** 0 **或** 1**，表示在有一个以上认证模块存在时，前一个是否在自身不能确定用户能访问的情况下还需将判定的权力前传给下一个模块。缺省值为** 1**，表示不前传，可提高安全性，对应** AuthAuthoritative **配置命令。**

- 与初始化相关的方法有：

```
create_auth_dir_config
```

- 与配置命令处理相关的方法有：

```
set_auth_slot
ap_set_flag_slot
```

- 与请求处理有关的方法有：

```
get_pw
groups_for_user
authenticate_basic_user
check_user_access
```

create_auth_dir_config 方法

```
create_auth_dir_config(pool *p, char *d)
```

创建一个缺省的 auth_config_rec 结构，并返回之。

set_auth_slot 方法

```
set_auth_slot(cmd_parms *cmd, void *offset, char *f, char *t)
```

本方法处理配置命令 AuthUserFile 及 AuthGroupFile，由配置命令后的字串 f 参数值获得用户或组文件的全路径名入口，即往前加上服务器根。其中，offset 指出要处理的配置信息在 auth-config-rec 结构中的偏移，在一个方法要处理多个配置命令时需要此参数把配置命令和记录信息的数据结构的各成员之间对应起来。

若 t 不为空且不为"standard"，则返回出错信息“ Invalid auth file type ”，否则，调用 http_config.c 中的 ap_set_file_slot 方法完成具体的配置。

ap_set_flag_slot 方法

```
ap_set_flag_slot(cmd_parms *cmd, char *struct_ptr, int arg)
```

本方法实际定义在 http_config.c 中，但实际只被各个认证模块使用来在指定的数据结构的指定位移处设置一个整数标志。

get_pw 方法

```
get_pw(request_rec *r, char *user, char *auth_pwfile)
```

本方法用来从口令文件中取得当前请求用户对应的口令。auth-pwfile 为记录用户授权信息（ID 及口令）的配置文件，返回值为指向口令字符串的指针。它在 authenticate_basic_user 方法中被调用。

具体处理过程为：

首先打开口令文件，出错则记录文件操作错误，并返回空指针；否则继续；

然后逐次从配置文件中读取一行，查看是否为当前请求用户的信息，若是则取出口令字

段返回之。若一直到文件尾还未找到对应行，则返回 NULL，表示用户提供的用户名非法。

groups_for_user 方法

```
groups_for_user(pool *p, char *user, char *grpfile)
```

本方法从组信息（包含组名及组所含的成员的 ID）文件 grpfile 中获得用户 user 所属的组，返回指定用户所在的所有组的组名的列表数组。它在 check_user_access 方法中被调用。

具体操作过程如下：

打开组信息文件，若失败则记录错误 "Could not open group file"，并返回空指针。

正常时，循环读取文件的每一行，且跳过注释行。

每一行包含一个组的名称及成员的定义。依次从一行中取出以逗号分开的用户 ID，检查是否与指定的 user 相同。若相同，则将当前组名放入返回的组列表中。

最后，关闭文件，释放资源，返回组列表。

authenticate_basic_user 方法

```
authenticate_basic_user(request_rec *r)
```

本方法用于 HTTP 基本认证，检查用户 ID 及其口令是否正确。

具体过程是：首先取得 AuthUserFile 命令设置的用户口令文件信息，然后检查客户端给出的用户名是否为本服务器上的合法用户及是否在本服务器有授权。若验证成功则返回 OK，否则若是口令文件没有设置或找不到，返回 DECLINED，否则返回 AUTH_REQUIRED，即或者用户不存在或口令不对，提示用户重新输入。

具体过程如下：

调用 ap_get_basic_auth_pw 方法从用户请求中取得认证口令，若返回值（非 OK）则返回；

若无用户授权信息文件，则返回 DECLINED，拒绝访问；

调用 get_pw 方法，从口令文件获取当前请求用户的口令字。若 get_pw 方法返回 NULL，发现该用户名并不存在，则看 auth_config_rec 结构的配置信息块中的 auth_authoritative 值，若其为 1（on），表示无需可能存在的下一个认证模块处理当前请求，直接将用户不存在的错误信息记入日志，并返回 AUTH_REQUIRED，提示用户重新提供认证用户名及口令；若为 0，则返回 DECLINED，表示认证的权利前传给下一个可能存在的认证模块（参阅 AuthAuthoritative 配置命令）。

若从口令文件取到了用户的口令，则调用 ap_md5c.c 中的 ap_validate_password 方法对用户提供的口令和本地存放的口令进行比较。若用户提供了错误的口令，则记录出错信息，返回 AUTH_REQUIRED；否则说明用户身份得到认证，返回 OK。

check_user_access 方法

```
check_user_access(request_rec *r)
```

本方法主要根据 limit、require 的组信息进行文件/目录级的授权检查，即使是本服务器的合法用户，也须通过此检查被授权后才可进行访问。若用户 ID 符合要求，则返回 OK。

本方法与 authenticate_basic_user 方法一起，完成全部 mod_auth 模块所负责的认证工作。

操作过程如下：

调用 ap_requires **方法获取** require **配置信息。若无** requires **配置，则表示任何用户均可访问，返回** OK。

若有 requires **配置，则循环处理每一条配置，检查当前请求用户是否在指定的用户或组范围内。若在内，则返回** OK（**即是该用户** ID **是被允许访问的**）。

否则检查 sec->auth_authoritative **的值。若为** 0，**表示认证的权力前传给下一个可能存在的认证模块，则返回** DECLINED（**参阅** AuthAuthoritative **配置命令**）。**否则为** 1，**表示已可定论不能访问，则进行日志，最终返回** AUTH_REQUIRED，**提示重新认证。**

mod_auth_anon 中实现的方法

本模块中定义了如下数据结构分别描述匿名用户和匿名认证配置信息：

```
typedef struct auth_anon {
        char *password;
        struct auth_anon *next;
} auth_anon;//由于匿名用户 ID 可多个，故定义为链结构的元素
typedef struct {
    auth_anon *auth_anon_passwords;//匿名用户名，而非是变量名所表达的匿名口令
    int auth_anon_nouserid;//是否要求用户输入 ID
    int auth_anon_logemail;//是否记录下 email 地址
    int auth_anon_verifyemail;//是否效验 email 地址
    int auth_anon_mustemail;//用户是否必须输入 email 地址
    int auth_anon_authoritative;//是否前传认证权力给后面的认证模块
} anon_auth_config_rec;//用于记录匿名认证配置信息
```

- **与初始化相关的方法有：**

```
create_anon_auth_dir_config
```

- **与配置命令处理相关的方法有：**

```
anon_set_passwd_flag
anon_set_userid_flag
anon_set_logemail_flag
anon_set_verifyemail_flag
anon_set_authoritative_flag
anon_set_string_slots
```

- **与请求处理有关的方法有：**

```
anon_authenticate_basic_user
check_anon_access
```

create_anon_auth_dir_config 方法

```
create_anon_auth_dir_config(pool *p, char *d)
```

创建一个 anon_auth_config_rec **结构，并按匿名认证的缺省配置值进行初始化，返回指向该结构的指针，失败则为** NULL。

anon_set_passwd_flag 方法

```
anon_set_passwd_flag(cmd_parms *cmd, anon_auth_config_rec * sec, int arg)
```

本方法设置 anon_auth_config **结构中的** auth_anon_mustemail **成员变量值为** arg**。**

anon_set_userid_flag 方法

```
anon_set_userid_flag(cmd_parms *cmd, anon_auth_config_rec * sec, int arg)
```

本方法设置 anon_auth_config_rec **结构中的** auth_anon_nouserid **成员变量值为** arg**，** auth_anon_nouserid **用于保存是否要求用户输入用户** ID **的配置信息。**

anon_set_logemail_flag 方法

```
anon_set_logemail_flag(cmd_parms *cmd, anon_auth_config_rec * sec, int arg)
```

本方法将 arg **赋给** anon_auth_config_rec **结构** sec **中的** auth_anon_logemail **成员（控制是否登记用户** email**）。**

anon_set_verifyemail_flag 方法

```
anon_set_verifyemail_flag(cmd_parms *cmd, anon_auth_config_rec * sec, int arg)
```

本方法设置 anon_auth_config_rec **结构** sec **的** auth_anon_verifyemail**（是否效验** email**）取** arg **值。**

anon_set_authoritative_flag 方法

```
anon_set_authoritative_flag(cmd_parms *cmd, anon_auth_config_rec * sec, int arg)
```

本方法设置 anon_auth_config_rec **结构** sec **的** auth_anon_authoritative **成员变量取** arg **值。**

anon_set_string_slots 方法

```
anon_set_string_slots(cmd_parms *cmd, anon_auth_config_rec * sec, char *arg)
```

本方法将 arg **指向的匿名用户** ID **加入** anon_auth_config_rec **结构** sec **的** auth_anon_passwords **变量所在链的链头。**

具体操作过程如下：

若 arg **空，则以提示“** Anonymous string cannot be empty, use Anonymous_NoUserId instead **”返回，否则继续执行。**

保留 sec->auth_anon_passwords **到** first**。**

调用 ap_palloc()，在 cmd-pool 中申请一 auth_anon 结构空间，将指针赋给 sec->auth_anon_passwords；将 arg 赋给 sec->auth_anon_passwords 中的 password。若以上两步失败则以提示"Failed to claim memory for an anonymous password..." 返回，否则继续

将 first 赋给 sec->auth_anon_passwords->next，即调整链表结构，并返回 NULL。

anon_authenticate_basic_user 方法

```
anon_authenticate_basic_user(request_rec *r)
```

本方法对匿名访问用户按基本认证机制进行访问认证及身份认证。

操作过程如下：

调用 ap_get_basic_auth_pw 方法取得请求中的用户名及口令。若返回了非 0 值，则直接将其返回，说明或者请求中没有认证信息或者认证机制不是基本认证。

检查 sec->auth_anon_passwords（合法的匿名名字链）。若为空则返回 DECLINED；否则继续，验证用户名是否合法。

若请求中用户名为空且 sec->auth_anon_nouserid 为 1（即允许请求不提供 ID）则记录 OK 到变量 res 中，表示在用户名上已通过检查；否则，在 sec->auth_anon_passwords （匿名名字链表）中查找当前请求用户名，若有则 res=OK。

下一步验证口令。若用户名 OK 且在要求 email 地址作为口令的情况下使用了合法的地址，则返回 OK，通过认证；否则若 sec->auth_anon_authoritative 为 1，表示不前传认证权力，则记录未通过认证的错误信息，返回 AUTH_REQUIRED；若为 0，则返回 DECLINED，允许下一模块处理。

check_anon_access 方法

```
check_anon_access(request_rec *r)
```

本方法验证匿名访问。

当前版本中直接返回 DECLINED，以后可能在此扩展功能。

mod_digest 中实现的方法

本模块中定义了如下的数据结构分别记录配置信息及请求处理时的摘要信息。

```
typedef struct digest_config_struct {
   char *pwfile;                    //用户信息文件，其中密码是以摘要形式保存的
} digest_config_rec;
typedef struct digest_header_struct {
    char *username;     //用户名
    char *realm;            //用户在服务器上所属的域（由//服务器划定的）
    char *nonce;            //现时值（与现在时间有关），参阅 RFC2069 协议内容
    char *requested_uri;    //被请求资源的 uri
    char *digest;           //摘要值
} digest_header_rec;    //记录摘要认证体制的头的信息
```

- **与初始化相关的方法有：**

```
create_digest_dir_config
```

- **与配置命令处理相关的方法有：**

```
set_digest_slot
```

- **与请求处理有关的方法有：**

```
get_hash
get_digest_rec
find_digest
authenticate_digest_user
digest_check_auth
```

create_digest_dir_config 方法

```
create_digest_dir_config(pool *p, char *d)
```

创建一个 digest_config_rec **结构，返回之。其中调用** ap_pcalloc **方法来实现空间申请。**

set_digest_slot 方法

```
set_digest_slot(cmd_parms *cmd, void *offset, char *f, char *t)
```

本方法处理配置命令 AuthDigestFile**。**

若配置出错，则返回"Invalid auth file type"**出错信息；否则调用** ap_set_string_slot **来保存文件名信息。**

get_hash 方法

```
get_hash(request_rec *r, char *user, char *auth_pwfile)
```

本方法在口令文件中查找当前请求用户的口令摘要信息，返回对应的信息行字符串。若未找到，则返回空指针。auth-pwfile **参数给出口令文件的路径。本方法在** authenticate_digest_user **方法中被调用。**

具体过程如下：

首先打开口令文件。若出错，则记录错误信息"Could not open password file "**，并返回** NULL**；否则继续。**

依次读出文件各行，并跳过注释及空行。

若该行中用户名及认证域与请求中对应信息相匹配，则返回当前行；若最终未找到，则返回空指针。

get_digest_rec 方法

```
get_digest_rec(request_rec *r, digest_header_rec * response)
```

本方法从请求中解析出摘要认证信息，在 authenticate_digest_user **方法中被调用。**

若客户端使用的认证机制不是摘要机制，则返回 DECLINED；若服务器错则返回 SERVER-ERROR，并记录错误；若认证信息不足或用户未提供认证信息，则返回 AUTH-REQUIRED；否则获得用户提供的摘要信息，返回 OK。

具体处理过程为：

若服务器配置中未给出认证类型 AuthType 或给出了但非摘要认证,则返回 DECLINED；

若服务器配置中未给出认证名称 AuthName，则记录出错信息，返回 SERVER_ERROR；

若用户请求中，没有认证信息，则调用 ap_note_digest_auth_failure 方法设置提示用户提供认证信息的头域，返回 AUTH_REQUIRED；

若客户请求非摘要认证机制，则记录"client used wrong authentication scheme "出错信息，返回 AUTH_REQUIRED，要求用户重新提供认证信息；

进而对请求中提供的认证信息进行解析，分别得到用户名 username、认证域 realm、nounce 值、请求地址 requested_uri、认证机制标志等信息。若其中任一为空，则返回 AUTH_REQUIRED，要求重新进行认证；否则正常，返回 OK。

find_digest 方法

```
find_digest (request_rec *r, digest_header_rec * h, char *a1)
```

本方法根据用户送来的认证头域信息（在 h 里），及本地的用户口令文件中查出的密码信息，按摘要认证机制，算出用于验证用户摘要值的正确摘要值。返回 32 字符长的 16 基编码字符串。具体则是调用 util_md5.c 文件中的 ap_md5 方法来实现的。

authenticate_digest_user 方法

```
authenticate_digest_user(request_rec *r)
```

本方法对用户进行摘要认证，检验用户身份的真实性。

具体处理过程如下：

调用 get_digest_rec 方法，取用户请求中的摘要认证信息，若发现错误，则直接返回错误代码。

若用户口令信息文件不存在，则返回 DECLINED。

调用 get_hash 方法，从口令文件中取得请求用户的口令信息，不成功则记录错误信息，返回 AUTH_REQUIRED，要求重新进行请求认证；

若取出了服务器出存放的口令信息，则调用 find_digest 方法计算 hash 值并将其与用户提供的口令进行比较。若两值不等，则记录"user password mismatch"出错信息，并返回 AUTH_REQUIRED；否则成功通过认证，返回 OK。

digest_check_auth 方法

```
digest_check_auth(request_rec *r)
```

本方法检验是否能访问该资源目录。

本方法主要根据 limit、require 的组信息进行文件/目录级的授权检查，即使是本服务器的合法用户，也须通过此检查被授权后才可进行访问。若用户 ID 符合要求，则返回 OK。

本方法与 authenticate_digest_user 方法一起，完成全部 mod_digest 模块所负责的摘要认

证工作。

操作过程如下：

调用 ap_requires 方法获取 require 配置信息。若无 requires 配置，则表示任何用户均可访问，返回 OK。

若有 requires 配置，则循环处理每一条配置，检查当前请求用户是否在指定的用户或组范围内。若在内，则返回 OK（即是该用户 ID 是被允许访问的）。

否则返回 AUTH_REQUIRED，提示重新认证。

第10节 log 模块

一般的软件都有日志功能，尤其是那些请求/响应式服务器软件更是如此，以便于日常统计、维护和监控。

在 Apache 中，mod_log_config 完成对请求的日志功能，日志的格式可以定制。在 Apache 1.2 及更高版本中，mod_log_config 缺省被编译进服务器中，用于替代 mod_log_common 模块；而在先前的版本中它只是可选模块。

本模块提供了三个配置命令：TransferLog 用来创建一个日志文件；LogFormat 用来对日志格式进行定制；CustomLog 则包含了以上两者的功能。TransferLog 和 CustomLog 配置命令可以多次使用，从而使请求可以分别记录到多个文件中。

本模块还实现了 CookieLog 配置命令，从而可以记录 mod_usertrack 模块生成的用户访问轨迹信息。但需要注意 CookieLog 已不建议使用，你应该使用 CustomLog 代替之。从 Apache 1.3.5 版本开始，本模块支持基于环境变量的有条件日志功能。这样你就可以根据某个环境变量是否被定义来控制是否对请求进行日志记录，而且这一功能是文件级别上的。从 Apache 1.3.5 开始，mod_log_config 模块还通过 SetEnvIf 命令和有条件 CustomLog 命令可以实现 mod_log_referer 模块支持的 RefererIgnore 类似的功能。

对于服务器运转中随时可能出现的错误的日志，则在主程序中专门设置了一个文件 http_log.c 进行处理，请参阅第 4 节中的相关部分。

7.10.1. 概述

日志可以用来记录服务器执行过程中发生的动作，既可是正常行为也可是异常错误。

日志信息一般写到文件中。Apache 服务器中主要用到的日志文件有四个，分别是：

- Access.log：记录请求主机、请求时间、请求行等信息，记录内容可自定义。如：

```
127.0.0.1 - - [06/Nov/1999:22:09:44 +0800] "GET / HTTP/1.0" 200 1673
127.0.0.1 - - [06/Nov/1999:22:09:45 +0800] "GET /apache_pb.gif HTTP/1.0" 200 2326
127.0.0.1 - - [06/Nov/1999:22:09:47 +0800] "GET /manual/index.html HTTP/1.0" 200 2352
127.0.0.1 - - [06/Nov/1999:22:09:47 +0800] "GET /manual/images/index.gif HTTP/1.0" 200 1540
127.0.0.1 - - [06/Nov/1999:22:09:47 +0800] "GET /manual/images/sub.gif HTTP/1.0" 200 6083
```

- Agent.log：记录代理用户产生请求的应用程序的名称，一般为浏览器名。如：

```
Mozilla/2.0 (compatible; MSIE 3.01; Windows NT)
Mozilla/2.0 (compatible; MSIE 3.01; Windows NT)
Mozilla/4.04 [en] (WinNT; I)
Mozilla/4.04 [en] (WinNT; I)
```

- Referer.log：**记录产生请求** URI **的源的** URI, **它可用于使服务器产生一个到源的逆向链表。如：**

```
- -> /
http://localhost/ -> /apache_pb.gif
- -> /
http://localhost/ -> /apache_pb.gif
http://localhost/ -> /manual/index.html
http://localhost/manual/index.html -> /manual/images/sub.gif
http://localhost/manual/index.html -> /manual/images/index.gif
```

- error.log：**记录各种错误信息。如：**

```
%% [Tue Jul 06 09:54:45 1999] GET /cgi-bin/status_report.exe&restart=Apache
HTTP/1.0
%%               404               c:/program               files/apache
group/apache/cgi-bin/status_report.exe&restart=apache
%error
script not found or unable to stat
```

日志信息还可以通过管道发送给由配置命令指定的应用程序，由后者可以进行更高级的分类和处理。请参阅 TransferLog **配置命令。**

7.10.2. 日志文件格式

除非用 LogFormat **配置命令特别指明，一般由** TransferLog **命令定义生成的日志文件都使用所谓的标准“普通日志格式（**Common Log Format，CLF**）；而在有特殊要求时（比如使用了多个日志文件，每个都有不同的格式），可使用** LogFormat **配置命令和** CustomLog **配置命令对日志的格式进行专门定义。**

普通日志格式

在普通日志格式文件——CLF **文件中每一行内容的含义描述如下：**

CLF **文件中每一行表示一次请求，其中包含如下由空格分隔的若干标记符：**

host**（机器地址）** ident**（标识）**authuser**（认证用户名）** date**（日期）** request**（请求）** status**（响应状态码）** bytes**（传输的字节数）**

若某个标记没有对应的值，则用连字符“-”代替。各个标记的含义如下：

host**：客户端机器的完整域名；域名不存在时取** IP **地址。**

ident**：若设置了** IdentityCheck **检查且客户端机器运行了** identd **例程，则此应是客户提供的标识信息。**

authuser**：若请求文档是受密码保护的，则此处为请求中给出的用户标识名。**

date**：请求的日期和时间。具体的格式如下：**

```
date = [day/month/year:hour:minute:second zone]
```

```
day = 2*digit
month = 3*letter
year = 4*digit
hour = 2*digit
minute = 2*digit
second = 2*digit
zone = (`+' | `-') 4*digit
```

request：客户请求行，用双引号括起来。

status：返回给客户的三位状态码数字。

bytes：返回给客户的不包括头域的数据的字节数。

日志格式的定制

LogFormat 命令和 CustomLog 命令中使用特定字符串作为格式化参数。该字符串在每次请求到来时被记录进日志文件。其中既可以包含原样复制到日志中的文本字符，也可以使用如下将被实际内容替代的`%'配置命令：

%...b：	发送的字节数，不包括 HTTP 头域；
%...f：	文件名；
%...{FOOBAR}e：	环境变量 FOOBAR 的内容；
%...h：	远程主机；
%...a：	远程主机的 IP 地址；
%...{Foobar}i：	请求中 Foobar 头域行的内容；
%...l：	远程的日志名（来自可能存在的 identd）；
%...{Foobar}n：	其他模块中 Foobar 注释的内容；
%...{Foobar}o：	响应中 Foobar 头域行的内容；
%...p：	服务器接收请求的端口号；
%...P：	处理请求的子进程 ID 号；
%...r：	请求的首行；
%...s：	响应状态码。对内部重定向的请求来说，此处为最初请求的状态码，可以用 %...>s 表示需要最后一个返回码；
%...t：	时间，使用普通日志时间格式；
%...{format}t：	时间，但使用 format 参数给出的格式，其中的格式遵循 strftime(3)标准。；
%...T：	处理请求所用时间，以秒为单位；
%...u：	远程用户名（来自用户认证，当响应状态码为 401 时此值可能非真值）；
%...U：	请求文档的 URL 路径；
%...v：	处理请求的服务器的 ServerName 名字；
%...V：	由 UseCanonicalName 命令设置的服务器名。

其中的`...'可能不对应任何内容（例如“ %h %u %r %s %b ”），也可能是某种条件（当条件不满足时用`-'做为具体值）。

条件表达式表现为一系列可能带有前缀`!'的 HTTP 状态码。这样，`%400,501{User-agent}i'指示只记录 400 和 501 类出错响应时的 User-agent 头域信息；`%!200,304,302{Referer}i' 则指示对所有未返回正常响应状态码的请求的 Referer 头域进行日志记录。

注意普通日志格式是通过串“ %h %l %u %t \"%r\" %s %b ”定义的，可以在此基础上进行扩展或调整（例如，在末尾增加额外的域）。NCSA 服务器的扩展/组合日志格式则为“ %h %l%u %t \"%r\" %s %b \"%{Referer}i\" \"%{User-agent}i\" ”。

注意处理请求的服务器的 ServerName 配置和 Port 配置分别取做%v 和 %p 的值，而不论是否有 UseCanonicalName 设置，因为不这样的话，日志分析程序将不得不重复使用 theentire 虚拟主机匹配算法来确定究竟是哪个主机处理了请求。

7.10.3. 多个日志文件的使用

可以多次使用 TransferLog 命令和 CustomLog 命令将请求访问记录到多个日志文件中。每个请求都会在如上两个配置命令定义的日志文件中被记录。

虚拟主机的使用

若<VirtualHost> 配置片段不包含任何 TransferLog 和 CustomLog 配置，则将使用为主服务器定义的日志进行记录；而若包含一个或多个日志配置，则由此虚拟主机处理的请求将仅记录到这些配置命令指定的日志文件中，而不会写入主服务器的任何日志文件。请参阅如下的例子。

7.10.4. mod_log_config 的实现

由于日志记录本身就是请求处理的一个阶段，所以 http_config.c 文件中专门定义了 ap_log_transaction 方法去完成日志记录阶段的工作，实际则是调用各个模块中定义的 logger 句柄。

```
int ap_log_transaction(request_rec *r)
{
    return run_method(r, offsets_into_method_ptrs.logger, 1);
}
```

而诸多模块中只有 mod_log_config 模块定义了 logger 句柄——multi_log_transaction 方法（参阅附录 Apache 各模块配置接口定义），所以 ap_log_transaction 方法实际的处理都在 multi_log_transaction 方法中。相关方法有：

```
multi_log_transaction
config_log_transaction
process_item
```

在服务器启动时，对配置文件中日志相关命令的处理也是由本模块完成的。相关方法有：

```
log_format
parse_log_string
parse_log_item
find_log_func
log_remote_host
log_remote_address
log_remote_logname
```

```
log_remote_user
log_request_line
log_request_file
log_request_uri
log_status
log_bytes_sent
log_header_in
log_header_out
log_note
log_env_var
log_request_time
log_request_duration
log_virtual_host
log_server_port
log_server_name
log_child_pid
```

又由于日志记录涉及客户请求各个处理阶段的方方面面，随时需要记录必要的日志信息，如出错情况等，表现为调用日志记录方法，生成日志条目，因此 Apache **服务器中在主程序部分专门安排了一个文件——**http_log.c**，负责出错日志记录过程的实现。相关方法有：**

multi_log_transaction 方法

```
multi_log_transaction(request_rec *r)
```

针对传入的请求 r **为每个（虚拟）服务器执行日志。**

首先获得 multi_log_state **结构，它保存着要用到的日志文件队列，然后用一循环分别取出该队列中的每一日志文件状态结构（文件名，**fd**），然后以默认格式对每一日志文件进行日志。具体的日志操作在** config_log_transaction **方法中。**

multi_log_state **结构的定义如下：**

```
typedef struct {
    char *default_format_string;
    array_header *default_format;
    array_header *config_logs;
    array_header *server_config_logs;
    table *formats;
} multi_log_state;
```

每个服务器（无论是主服务器还是虚拟服务器）均具有一个该结构类型的数据块。对主服务器而言，config_logs **包含一个日志文件的列表，**server_config_logs **为空。对虚拟服务器而言，**server_config_logs **指向与主服务器中** config_logs **相同的列表，**config_logs **则指向在该虚拟服务器中定义的日志列表。这样如果该虚拟服务器没有定义的日志，就可以用主服务器的日志。**

LogFormat **配置命令给出的默认日志格式保存在** default_format **中。**

每个日志文件的信息用 config_log_state **结构来定义。**

```
typedef struct {
    char *fname;
```

```
    char *format_string;
    array_header *format;
    int log_fd;
    char *condition_var;
#ifdef BUFFERED_LOGS
    int outcnt;
    char outbuf[LOG_BUFSIZE];
#endif
} config_log_state;
```

fname **存放日志文件名；**log_fd **为日志文件句柄；**format_string **存放日志格式串，**format **存放解析后的结果，具体说是元素为** log_format_item **类型的数组，若为空，则使用默认格式。**

log_format_item **类型定义如下：**

```
typedef struct {
    item_key_func func;
    char *arg;
    int condition_sense;
    int want_orig;
    array_header *conditions;
} log_format_item;
```

func **为对应某格式串参数的记录日志信息的函数。**

config_log_transaction 方法

```
config_log_transaction(request_rec *r, config_log_state * cls, array_header *default_format)
```

首先判定格式串中是否含有环境变量存在条件。若有则判定是否成立，以确定是否继续进行处理。

若条件成立或无条件，则按指定的格式进行记录。具体是对每一格式字符参数调用 process_item **函数，从而执行对应该参数的日志函数，得到相应的日志记录字符串。**

将所有的日志记录串连接起来，最后写到指定的日志文件中。

process_item 方法

```
static const char *process_item(request_rec *r, request_rec *orig, log_format_item *item)
```

本方法是日志记录的实际处理过程。

item **参数为** log_format_item **结构类型，其中包含一** item_key_func **类型变量** func**，它被定义为指向一带有**(request_rec *, char *)**变量，返回** char **类型值的函数的指针，通过该函数可得到对应日志参数的日志字符串，最后返回之。**

格式字符参数与处理函数的对应关系定义在日志格式串参量数组 log_item_keys **中：**

```
static struct log_item_list {
    char ch;
```

```
    item_key_func func;
    int want_orig_default;
} log_item_keys[] = {
    { 'h', log_remote_host, 0   },
    { 'a', log_remote_address, 0},
    { 'l', log_remote_logname, 0    },
    { 'u', log_remote_user, 0   },
    { 't', log_request_time, 0  },
    { 'T', log_request_duration, 1  },
    { 'r', log_request_line, 1  },
    { 'f', log_request_file, 0  },
    { 'U', log_request_uri, 1   },
    { 's', log_status, 1},
    { 'b', log_bytes_sent, 0},
    { 'i', log_header_in, 0 },
    { 'o', log_header_out, 0},
    { 'n', log_note, 0  },
    { 'e', log_env_var, 0   },
    { 'V', log_server_name, 0   },
    { 'v', log_virtual_host, 0  },
    { 'p', log_server_port, 0   },
    { 'P', log_child_pid, 0 },
    { '\0'  }
};
```

其中 ch **为格式字符参数，**func **为对应的函数。**

在服务器启动读入配置文件时，若读到 LogFormat **命令，则调用** log_format **方法对格式字符串进行解析，最后可得到一个** array_header* **类型的数组，其中每一项均为** log_format_item **结构类型，对应格式字符串中的一个字符参数，其中对应的日志记录函数即是从** log_item_keys **中查到的。**

log_format 方法

```
static const char *log_format(cmd_parms *cmd, void *dummy, char *fmt, char
*name)
```

处理定义的 LogFormat **配置命令（参阅前面的** LogFormat **命令讲解）。**

对命令中定义的格式字符串调用 parse_log_string **方法进行解析。然后根据是否是在定义格式串别名，来进行不同处理：**

若是格式串别名定义，仅将该别名定义记录到当前模块的 multi_log_state **结构的** formats **表中；若是定义日志格式串，则将解析的结果记录到** multi_log_state **结构的** default_format **中。**

parse_log_string 方法

```
static array_header *parse_log_string(pool *p, const char *s, const char **err)
```

循环调用 parse_log_item 方法对日志格式串 s 进行解析，返回结果。该结果给出了定义的格式字符参数与对应日志记录函数之间的映射关系。

parse_log_item 方法

```
static char *parse_log_item(pool *p, log_format_item *it, const char **sa)
```

对格式串 sa 进行解析，直至开头的一个`%'配置命令被解析。

解析出的格式字符参数对应的日志函数由 find_log_func 方法获得。

find_log_func 方法

```
static struct log_item_list *find_log_func(char k)
```

从定义格式字符与处理函数映射关系的 log_item_keys 中查找与指定格式字符 k 对应的处理函数，将对应映射项返回。

log_remote_host 方法

```
static const char *log_remote_host(request_rec *r, char *a)
```

调用 http_core.c 中的 ap_get_remote_host 方法从 r 中取得远程主机名返回之。与格式字符参数'h'对应。

log_remote_address 方法

```
static const char *log_remote_address(request_rec *r, char *a)
```

返回远程主机 IP 地址。与格式字符参数'a'对应。

log_remote_logname 方法

```
static const char *log_remote_logname(request_rec *r, char *a)
```

调用 http_core.c 中的 ap_get_remote_logname 方法取得远程用户登录名返回之。与格式字符参数'l'对应。

log_remote_user 方法

```
static const char *log_remote_user(request_rec *r, char *a)
```

返回远程用户名。与格式字符参数'u'**对应。**

log_request_line 方法

```
static const char *log_request_line(request_rec *r, char *a)
```

返回用户请求行，将可能存在的密码用"XXXXXX"**屏蔽掉。与格式字符参数**'r'**对应。**

log_request_file 方法

```
static const char *log_request_file(request_rec *r, char *a)
```

返回请求的文件名。与格式字符参数'f'**对应。**

log_request_uri 方法

```
static const char *log_request_uri(request_rec *r, char *a)
```

返回请求 URI **地址。与格式字符参数**'U'**对应。**

log_status 方法

```
static const char *log_status(request_rec *r, char *a)
```

返回应答状态码。与格式字符参数's'**对应。**

log_bytes_sent 方法

```
static const char *log_bytes_sent(request_rec *r, char *a)
```

返回应答返回的字节数。与格式字符参数'b'**对应。**

log_header_in 方法

```
static const char *log_header_in(request_rec *r, char *a)
```

返回指定请求头域内容。与格式字符参数'i'**对应。**

log_header_out 方法

```
static const char *log_header_out(request_rec *r, char *a)
```

返回指定应答头域内容。与格式字符参数'o'**对应。**

log_note 方法

```
static const char *log_note(request_rec *r, char *a)
```

返回记录在 r->notes 中的指定请求处理信息（如错误信息，请参阅本节的 ap_log_rerror 方法）。与格式字符参数'n'对应。

log_env_var 方法

```
static const char *log_env_var(request_rec *r, char *a)
```

返回指定的环境变量值。与格式字符参数'e'对应。

log_request_time 方法

```
static const char *log_request_time(request_rec *r, char *a)
```

返回请求时刻。首先用 ap_get_gmtoff(&timz)获得请求接收时刻，然后以定制格式或 CLF%d/%b/%Y:%H:%M:%S 格式转换时间串。与格式字符参数't'对应。

log_request_duration 方法

```
static const char *log_request_duration(request_rec *r, char *a)
```

返回请求处理所用时间。与格式字符参数'T'对应。

log_virtual_host 方法

```
static const char *log_virtual_host(request_rec *r, char *a)
```

返回处理请求的虚拟主机名。与格式字符参数'v'对应。

log_server_port 方法

```
static const char *log_server_port(request_rec *r, char *a)
```

返回服务器端口号。与格式字符参数'p'对应。

log_server_name 方法

```
static const char *log_server_name(request_rec *r, char *a)
```

返回本服务器的名称。调用 http_core.c 中的 ap_get_server_name 方法实现。与格式字符参数'V'对应。

log_child_pid 方法

```
static const char *log_child_pid(request_rec *r, char *a)
```

返回请求处理子进程的进程 ID。与格式字符参数'P'对应。

ap_open_logs 方法

```
void ap_open_logs(server_rec *s_main, pool *p)
```

调用 open_error_log 方法创建日志文件或日志子进程，并使之替代标准错误输出。

open_error_log 方法

```
static void open_error_log(server_rec *s, pool *p)
```

打开错误日志文件或启动负责处理出错信息流的子进程。

ap_log_error 方法

```
API_EXPORT(void) ap_log_error(const char *file, int line, int level, const server_rec *s, const char *fmt, ...)
```

是错误日志记录的主方法，具体的实现在 log_error_core 方法中。

log_error_core 方法

```
static void log_error_core(const char *file, int line, int level, const server_rec *s, const request_rec *r, const char *fmt, va_list args)
```

本方法是错误日志核心函数。具体处理分如下三种情况：

- 如果在使用标准错误日志（stderr logging），则不记录级别高于指定日志级别的消息，除非是关于启动／关闭的；
- 如果在使用正常错误日志（normal logging），则不记录级别高于指定日志级别的消息，除非是关于启动／关闭的；
- 如果在进行系统日志（syslog logging），则不记录级别高于指定日志级别的消息，包括关于启动／关闭的。

关于错误信息的具体情况，请参阅前面关于 error.log 的示例。

ap_log_rerror 方法

```
API_EXPORT(void) ap_log_rerror(const char *file, int line, int level, const request_rec *r, const char *fmt, ...)
```

基本功能与 ap_log_error 相似，也是调用 log_error_core 方法来实现具体功能。但增加

了如下部分：

当错误级别是或高于警告级别（warning），且还没有记录与此类错误相关的信息时，则将此处的出错信息记录到 r 的 notes 域中。

第11节 status 模块

本模块支持网站管理员可以远程获得服务器运行状况,从而可以及时发现并消除隐患和问题。本模块的实现在 mod_status.c 文件中，且只在 Apache 1.1 及以后版本中可用。

7.11.1. mod_status 的使用

在本模块的支持下，服务器可以返回一个 HTML 页面，它给出了当前服务器运行状况的统计信息，包括如下内容：

- 处理请求的子进程数目；
- 空闲的子进程数目；
- 每个子进程的状态、已处理请求的数目及已返回响应的总字节数；
- 客户请求及响应的字节总数目；
- 服务器启动或重新启动的时间及已运行的时间
- 每秒平均处理的请求数、每秒响应的字节数及每个请求响应的平均字节数；
- 每个子进程当前占用的 CPU 时间及 Apache 占用的总时间；
- 当前正在处理的请求。

根据客户的要求，该页面还可以在自动更新。

由于 mod_status 模块缺省情况下没有被配置到服务器中，所以要使用本模块需要进行一些配置工作。具体就是将配置文件中的如下命令行前的“#”去掉，使之有效。

```
#LoadModule status_module modules/ApacheModuleStatus.dll

#<Location /server-status>
#    SetHandler server-status
#    Order deny,allow
#    Deny from all
#    Allow from .your_domain.com
#</Location>
```

其中第一行是告诉服务器将 mod_status 模块动态装入，后一个片段则是指示由 mod_status 模块中定义的 server-status 句柄来处理客户的 server-status 请求。其中的.your_domain.com 应改成被允许进行 server-status 请求的客户所在的网络地址。

客户提出的 server-status 请求具体有如下不同形式：

- /server-status：

返回表格方式的页面；

- /server-status?notable：

返回非表格方式的页面，此方式针对不支持表格的浏览器；

- /server-status?refresh：

返回的页面每 1 秒自动更新；

- /server-status?refresh=N：

返回的页面每 N 秒自动更新；

- /server-status?auto：

返回适于程序自动解析的页面。

7.11.2. mod_status 的实现

- 初始化方法有：

```
status_init
```

- 配置命令处理方法有：

```
set_extended_status
```

- 客户请求处理有关的方法有：

```
format_byte_out
format_kbyte_out
show_time
status_handler
```

status_init 方法

```
status_init(server_rec *s, pool *p)
```

对 status_flags 数组进行初始化，设置上表示服务器各方面参数的标志字符。

set_extended_status 方法

```
set_extended_status(cmd_parms *cmd, void *dummy, char *arg)
```

记录 ExtendedStatus 配置命令的设置。arg 为该命令的具体配置参数。

format_byte_out方法

```
format_byte_out(request_rec *r, unsigned long bytes)
```

根据 bytes 的数目大小以单位至少为 bytes 的字符串表示将其写入 r 指示的连接中，被 status_handler 方法调用。

format_kbyte_out方法

```
format_kbyte_out(request_rec *r, unsigned long kbytes)
```

根据 kbytes 的数目大小以单位至少为 kbytes 的字符串表示将其写入 r 指示的连接中，被 status_handler 方法调用。

show_time 方法

```
show_time(request_rec *r, time_t tsecs)
```

将单纯的秒数换算成包括天、日、小时、分、秒的字串表示，并写入请求 r 表示的连接中，被 status_handler 方法调用。

status_handler 方法

```
status_handler(request_rec *r)
```

根据配置和请求的情况，向请求 r 表示的连接中写入关于服务器的一些统计信息，在客户端的浏览器中将显示一 HTML 文件。

首先查看请求的参数，若是 refresh，则将指明的刷新频率（即多长时间刷新一次）作为“Refresh”头域的值写入 r->headers_out。而若是 notable 或 auto 模式，则进行记录。

调用 ap_send_http_header 发送 HTTP 头。并判断若只发送头，则返回 0。

调用 ap_sync_scoreboard_image 完成内存与硬盘中内容的同步。

对所有服务器（虚拟）的状态进行计数，包括就绪个数、正忙个数、访问次数、传输的字节数等。

若不是 auto 模式，则向客户发送服务器的版本、运行时间等信息。

若需要显示扩展 status 信息，则发送扩展信息，包括：访问总次数和传输总字节数、响应频率、每秒负载、每次请求传输的字节数等。但若是 auto 模式，仅发送简单信息，否则发送详细的内容。

继续发送忙闲服务器次数信息。

发送诸服务器的 Scoreboard 信息。

若不是 auto 模式，则发送关于 Scoreboard 的注释。

若若需要显示扩展 status 信息，则发送每个服务器的详细信息。否则，仅发送“若要了解详细信息，需要使用 ExtendedStatus On”的提示。

若不是 auto 模式，发送服务器的标识串。

最后返回 0。

第12节 headers 模块

本模块是可选的，用来支持对 HTTP 响应头域的定制，包括合并、替换或删除。只有将 mod_headers.c 编译到 Apache 中后才可以使用。

7.12.1. mod_headers 的实现方法

- 配置初始化方法有：

```
create_headers_dir_config
create_headers_config
merge_headers_config
```

- **配置命令处理方法有：**

```
header_cmd
```

- **与客户请求处理相关的方法有：**

```
do_headers_fixup
fixup_headers
```

create_headers_config 方法

```
create_headers_config(pool *p, server_rec *s)
```

本方法创建并初始化本 headers **模块的** headers_conf **类型的信息块，返回块结构的指针。**

create_headers_dir_config 方法

```
create_headers_dir_config(pool *p, char *d)
```

目录信息 d **被忽略，实际完成与** create_headers_config **同样的工作。返回信息块结构的指针。**

merge_headers_config 方法

```
merge_headers_config(pool *p, void *basev, void *overridesv)
```

本方法将两个配置信息块 basev **和** overridesv **合并，返回结果信息块指针。**

header_cmd 方法

```
header_cmd(cmd_parms *cmd, headers_conf * dirconf, char *action, char *hdr,
char *value)
```

对在配置文件中定义的 Header **命令进行处理。其中** action、hdr、value **分别指示** header_entry **结构的各个参数。**

根据 action **中的值是** set、unset、add **或** append**，分别进行标记。若值非法，则返回错误信息串。**

根据识别的动作类型，再对其它参数的值进行判断。有非法现象，则返回错误信息。否则创建 header_entry **结构记录之。**

若 cmd->path **非空，则将上述** header_entry **结构放到特定目录的对应信息配置块中，否则放到整个服务器的信息块中。**

do_headers_fixup 方法

```
do_headers_fixup(request_rec *r, array_header *headers)
```

本方法针对请求 r **的响应头，执行** headers **中记录的** Header **命令，完成** HTTP **头域的增加、删除、附加和设置。**

fixup_headers 方法

```
fixup_headers(request_rec *r)
```

本方法分别用 r 中的 server->module_config 和 per_dir_config 记录的 Header 命令对 r 中的响应头进行处理。具体调用 do_headers_fixup 实现。

总返回 DECLINED。

第13节 info 模块

本模块支持向客户提供对服务器模块安装和配置命令的设置等情况的描述，在 mod_info.c 文件中实现。本模块缺省时并没有被编译到服务器软件中，且只在 Apache 1.1 及以后版本中可用。

7.13.1. mod_info 使用方法

由于该模块默认没编译到服务器中，所以要使用该模块，还要做一些额外的工作。有两种方法：

- 将 mod_info.c 直接编译到服务器程序中。你需要在 modules.c 中若干位置增加对应 mod_info 模块的项，然后对整个服务器程序进行编译。
- 将 mod_infoc 单独编译成动态链接库，在服务器的配置文件中通过 LoadModule 命令将其装入。

在配置加载该模块后，你还需要在配置文件中加入：

```
<Location /server-info>
SetHandler server-info
</Location>
```

这样，你就可以在客户端通过输入 http://your.host.dom/server-info 地址来访问服务器，获得由 mod_info 中的 server-info 句柄生成的描述服务器的页面了。

你可以使用的请求有：

- /server-info：

返回对服务器及所有模块的完整配置描述页面；

- /server-info?server：

只返回服务器配置信息；

- /server-info?module_name：

返回指定模块的配置信息；

- /server-info?list：

返回所链入的模块的简单列表。

需要注意的是：配置文件是在运行时被 mod_info 模块读入的，因此访问 http://your.host.dom/server-info 显示的信息未必就是当前服务器所使用的配置，因为配置文件可能在服务器启动后被更改；而且，服务器启动所基于的用户必须有读配置文件的权利，否则你将在 mod_info 生成的页面中看不到配置信息。

为了提高安全性，你还可以在<Location>片段中使用<Limit>配置片段来限制外界对上述特殊地址的访问。

7.13.2. mod_info 的实现

mod_info 中的处理方法可分为如下三类：

- 初始化方法有：

```
create_info_config
merge_info_config
```

- 配置命令处理方法有：

```
add_module_info
```

- 客户请求处理有关的方法有：

```
find_more_info
mod_info_html_cmd_string
mod_info_load_config
mod_info_module_cmds
display_info
```

create_info_config 方法

```
create_info_config(pool *p, server_rec *s)
```

创建 info 模块的类型为 info_svr_conf 的配置信息块，其中仅包含一个存放各模块附加描述信息的数组。

merge_info_config 方法

```
merge_info_config(pool *p, void *basev, void *overridesv)
```

将类型为 info_svr_conf 的配置信息块 basev 和 overridesv 合并，实际只是将其中的存放模块附加信息的数组合并。

add_module_info 方法

```
add_module_info(cmd_parms *cmd, void *dummy, char *name, char *info)
```

将由 AddModuleInfo 定义的关于模块 name 的附加信息 info 放到 info_svr_conf->more_info 数组中。

find_more_info 方法

```
find_more_info(server_rec *s, const char *module_name)
```

从配置中找出本模块的扩展的信息，返回对应字串。与 AddModuleInfo 配置命令有关。

mod_info_html_cmd_string 方法

```
mod_info_html_cmd_string(const char *string, char *buf, size_t buf_len)
```

将字串 string **进行处理，返回放在** buf **中。**

mod_info_load_config 方法

```
mod_info_load_config(pool *p, const char *filename, request_rec *r)
```

从由 filename **指定的文件中读出每条有效配置信息(即非注释行),生成以** info_cfg_lines **结构类型量为链节的链表，返回头指针。**

mod_info_module_cmds 方法

```
mod_info_module_cmds(request_rec *r, info_cfg_lines *cfg, const command_rec
*cmds, char *label)
```

将实际配置信息 cfg **中涉及** cmds **的内容发送给由** r **指示的客户端。**

display_info 方法

```
display_info(request_rec *r)
```

根据配置和请求的情况，向请求 r **表示的连接中写入关于服务器的当前信息，在客户端的浏览器中将显示一** HTML **文件。**

首先检查请求的方法是 M_GET**，否则直接退出，返回** DECLINED**。**

调用 ap_send_http_header **将** HTTP **头发出。且若** r->header_only **为** 1**，即只需发送头，则也直接退出，返回** 0**。**

首先发送 HTML **文件头、标题等。**

然后，若请求是 GET /server-info?list**，则将向连接中写入当前模块的列表，写** HTML **文件尾，然后返回。否则，将三个全局性文件中的配置命令读出。**

进一步地，若请求是 GET /server-info?server**，则将服务器的版本号、运行模式、主机名、**ServerRoot**、状态信息文件等内容以** HTML **格式发给客户。**

否则若请求是 GET /server-info?module_name**，则将对应** module_name **的模块的非空处理句柄信息、能处理的配置命令信息、配置文件中相关命令的实际配置信息、附加的模块信息等写入与客户的连接中。**

最后发送 HTML **文件尾，返回。**

第14节 其它若干问题

7.14.1. 事件控制

在 Windows 平台下，Apache 利用了其事件机制（Event）进行运行的控制，包括通知正在运行的 Apache 进程终止或重新启动。具体 Windows 下事件的工作机制请参阅相关资料(如 Visual Studio 帮助)。

在 Apache 启动时，在 http_main.c 中的 master_main 方法（主进程体）会执行如下一段代码：

```
    ap_snprintf(signal_prefix_string, sizeof(signal_prefix_string), "ap%d",
getpid());
    setup_signal_names(signal_prefix_string);
    signal_shutdown_event = CreateEvent(NULL, TRUE, FALSE,
signal_shutdown_name);
    signal_restart_event = CreateEvent(NULL, TRUE, FALSE, signal_restart_name);
```

即完成了用于外界控制自身终止或重启的两个事件：signal_shutdown_event 与 signal_restart_event。此两个事件有一个据一定规则构造的名字，具体构造过程为：

两个事件名的前缀相同，均是最开头为“ ap ”，然后紧跟此次运行进程的 ID 号，通过 getpid 方法可获取到。然后在 setup_signal_names 方法中，分别加上‘ _shutdown ’和‘ _restart ”，构成完整的名字。

在事件创建完成后，在 master_main 方法的循环中，通过如下代码：

```
    process_handles[current_live_processes] = signal_shutdown_event;
    process_handles[current_live_processes+1] = signal_restart_event;
    rv = WaitForMultipleObjects(current_live_processes+2,
            (HANDLE *)process_handles, FALSE, INFINITE);
```

对各个子进程和上述的两个事件进行无限等待，直到有任何一个进程终止或任何一个事件为信号态（参阅第 2 节中对 master_main 方法的介绍），后一种情况表明外部有进程将某事件置为了信号态，表示通知当前进程终止或重启，于是通过如下判定：

```
    cld = rv - WAIT_OBJECT_0;
    if (cld == current_live_processes) {
        shutdown_pending = 1;
        break;
    }
    if (cld == current_live_processes+1) {
        restart_pending = 1;
        break;
    }
```

退出当前循环，并进行最终的终止或重启处理。

上面介绍了通过事件受控的一方。发出控制的一方则有如下过程：

在 http_main.c 的 REALMAIN 方法中处理程序启动时的命令行参数时，若有 k 参数，表示此次运行是为了对正在运行的 Apache 进程进行控制。因此随后执行：

```
    if (signal_to_send) {
```

```
        send_signal(pconf, signal_to_send);
        exit(0);
    }
```

调用 send_signal 方法发送信号，然后退出。

在 send_signal 方法中，首先是按照上述事件名构造规则，构造针对受控制进程的事件名字，然后根据控制信号的不同，分别调用 ap_start_shutdown 方法和 ap_start_restart 方法，后两个方法则进一步调用 signal_parent 方法（由参数来区分控制信号），通过 SetEvent 调用完成具体的事件信号态设置，从而达到控制 Apache 进程的目的。

7.14.2. 注册表项

在 Windows 平台下，安装 Apache 会在注册表中创建对应的注册项，即在 HKEY_LOCAL_MACHINE 中生成 HKLM\SOFTWARE\Apache Group\Apache\version 项，其中 version 是随版本的变化而变化的，如对 Apache1.3.6 来说就是“1.3.6”。在该项中创建名为 ServerRoot 的值，对应的数据为 Apache 的安装目录。

处理注册表的工作是在 registry.c 中实现的，具体实现方法如下：

ap_registry_create_apache_key 方法

```
static int ap_registry_create_apache_key(void)
```

本方法创建 Apache 注册项 HLKM\SOFTWARE\Apache Group\Apache\version。若成功，返回 0；否则返回-1。

在 ap_registry_store_key_int 方法中设置注册键值时若发现 Apache 注册项不存在，会调用本方法创建之。

ap_registry_get_key_int 方法

```
static int ap_registry_get_key_int(pool *p, char *key, char *pBuffer, int
nSizeBuffer, char **ppValue)
```

本方法读取注册表中 Apache 注册项中指定注册键值的数据。key 中为注册键值的名字。若 ppValue 为 NULL，则申请空间存放注册键值的数据于*pValue 中；若 ppValue 非空，则认为其为长度是 nSizeValue 的缓冲区。

成功时返回值为结果串的长度，否则失败记录日志，且：

要取的值不存在时，返回-1；

若运行时内存空间不足，返回-2；

若 pValue/ nSizeBuffer 指示的缓冲区不足够大，返回-3；

若出现错误，返回-4。

ap_registry_store_key_int 方法

```
static int ap_registry_store_key_int(char *key, DWORD type, void *value, int
```

```
value_size)
```

本方法在 Apache **注册项中存放一个注册键值和数据对。若成功，返回** 0**；否则若键值不存在，返回**-1**，或出现其它错误，返回**-4**。**

若 Apache **注册项不存在，则首先调用** ap_registry_create_apache_key **方法创建之。**

ap_registry_get_server_root方法

```
int ap_registry_get_server_root(pool *p, char *dir, int size)
```

本方法读取注册表中 Apache **的** ServerRoot **注册值的数据。若成功或数据没找到，返回** 0 **；否则表示取注册数据时出错，返回**-1**。参数** dir **中存放获取的数据。调用** ap_registry_get_key_int **方法具体实现。**

ap_registry_set_server_root方法

```
int ap_registry_set_server_root(char *dir)
```

本方法完成 Apache **的** ServerRoot **注册键值的设置。成功时返回** 0**，否则返回**-1**。调用** ap_registry_store_key_int **方法具体实现。**

7.14.3. 服务运行模式

细心的读者可能会发现在前文中提到，当运行参数指示以 NT **下的服务方式运行** Apache **服务器时，会启动** Apache **服务，但并未对此条路径做详细的介绍。之所以如此安排，是为了使整个思路清晰简洁，避免过多的分支。在本节，将“补”上这一课，对与** NT **服务相关的内容进行阐述。**

在第 2 **节的** service_main **方法介绍中，实际涉及了所有的** Apache **中与服务运行模式相关的内容，包括服务的安装、卸载、运行及控制等，这些内容均在** os\win32\service.c **文件中定义。**

- **安装：**

由 InstallService **方法实现：**

```
GetModuleFileName( NULL, szPath, 512 )
```

取得当前执行程序的路径，存于 szPath **中。**

```
schSCManager = OpenSCManager(
             NULL,            // machine (NULL == local)
             NULL,            // database (NULL == default)
            SC_MANAGER_ALL_ACCESS   // access required
            );
```

打开服务控制管理器。

```
schService = CreateService(
    schSCManager,             // SCManager database
    globdat.name,       // 服务名
    globdat.name, // 显示名称
```

```
    SERVICE_ALL_ACCESS,          // 访问方式
    SERVICE_WIN32_OWN_PROCESS,  // 服务类型
    SERVICE_AUTO_START,          // 启动类型
    SERVICE_ERROR_NORMAL,       // 错误控制类型
    szPath,                        // 服务对应的执行代码路径
    NULL,
    NULL,
    NULL,
    NULL,                       // 使用本地系统帐号执行
    NULL);                      // 没有口令
```

创建名为 Apache **的服务。其中** globdat.name **在** service_main **方法中被设为“** Apache"**，且** Apache **服务为自动启动的服务。**

```
CloseServiceHandle(schService);
```

关闭服务句柄。

```
CloseServiceHandle(schSCManager);
```

关闭服务控制管理器句柄。

退出。

- **卸载：**

由 RemoveService **方法实现。具体是通过服务控制管理器，首先用** ControlService **方法停止可能的** Apache **服务的执行，然后再调用** DeleteService **方法将其从服务注册表中删除。代码实现参阅** service.c **文件。**

- **运行：**

关于运行的相关内容已在本章第 2 **节中进行了介绍，请参阅** service_main **及** service_main_fn **方法。**

- **控制：**

由 service_ctrl **方法实现。该方法完成用户对** Apache **服务的控制操作的响应处理。此方法是通过在** service_main_fn **方法中用**

RegisterServiceCtrlHandler(globdat.name, service_ctrl)))

被注册为服务控制方法。

在该方法中，按照用户的操作，进行相应的处理。目前的情形较简单，即只有 SERVICE_CONTROL_STOP **一种控制操作，接收到此要求后，调用** ap_start_shutdown **方法通知** Apache **进程退出。**

相信读者到此处，已经对 Apache **服务器无论是外部的使用还是内部的实现均有了全面的了解。考虑到** Apache **的实际使用情况，下一章特别地对两个分别支持** Perl **和** Java Servlet **的模块进行介绍，以期对读者的实际工作有所帮助。**

第8章 重要的 Apache 扩展模块

第1节 mod_perl 模块

8.1.1. 什么是 Perl

Perl（Practical Extraction and Report Language）叫做文字分析报告语言，是一个高级程序设计语言，由 Larry Wall 发明，融合了许多语言的特性。

Perl 的设计遵循实用原则，要求易用高效，而且功能全面。Perl 语言中包含了 C、C++、shell 脚本、sed、awk 这几种语言的语法，它原始的目的就是用来取代 UNIX 原来 sed/awk 与 script 的组合，用来汇总数据，产生报表。随着版本的改进，其功能越来越强。现在的功能已经超乎原先设计时的想象，几乎任何事都可以做到，也变成 PC 必备的标准工具了。

因为它对字符串与数据分析方面有很强的处理能力，尤其是在利用关联数组解析数据串方面，功能非常强大！现在许多信息都存放在数据库中，均可方便地用 Perl 来沟通。

在 Perl5.0 版本中进一步增加了面向对象的用法，增强了对字符串的处理能力，可以把整个文件当成一个字符串来处理、不受任何数据的大小限制而只受存储器的大小，能够处理 DBM 数据库格式的数据也能够处理二进制的数据等等。更重要的是，Perl 还可以在 Unix、WindowsNT、Windows95、Dos、Linux、FreeBSD、OS2、Macitosh 等操作系统中使用。目前还可以用 Perl 来和一些数据库软件沟通，在国外一些功能强大的 CGI 应用程序几乎都是用 Perl 语言来当主角！

若您看到这里就觉得功能这样强大的语言 Perl 一定很难学，那您就大错特错了。Perl 是很容易上手的。它的语法和 C 语言很接近，对于原本熟悉 C 语言的人来说，Perl 非常易学。只要有一点 C 语言的基础，再加上几个小时的学习时间就可以很轻易地用 Perl 语言写一个简单的 CGI 应用程序。至于那些没有 C 语言基础的人，只要多花一点时间，也是可以很快地掌握 Perl 的！到那时，您就会发觉原来 Perl 是那么的“平易近人”，而写出来的 CGI 应用程序是那么地简洁有力。

Perl 的编程不是本书要讨论的内容，请读者参考有关书籍。

8.1.2. 什么是 mod_perl

由于 Perl 在文本处理方面的出色表现，近年来，Perl 已成为最流行的 Web 客户程序开发语言。它主要用于开发 CGI 脚本。但是，它是一种解释语言，执行时需要启动解释程序，从而带来较大的工作负担。尤其在 CGI 比较多，负载比较重的 Web 站点上，这一点在给用户造成的“慢牵牛”的感觉中“贡献”最大。

我们常说，需求是社会进步的不竭源泉。正是上面的问题，促成了 mod_perl 的诞生。

Mod_perl 模块开发的目的是使 Apache 服务器能直接执行 perl 代码，从而避免频繁启动解释器带来的低效。Apache/Perl 集成计划实现了 Perl 和 Apache 的强强联合。采取的方法是把 Perl 的运行库连接进 Apache 服务器中，并且为服务器的 C 语言 API 提供面向对象的 Perl

接口。

Mod_perl是一个软件包。其中一部分被用于和 Apache 一起编译和连接。其余部分是 perl 代码，为支持 perl 的（perl-enabled）服务器提供面向对象接口。

为能更容易地运行已有的 Perl CGI 脚本，mod_perl 包含了通用目标模块（Apache::Registry）。该模块对 CGI 环境进行了模拟，支持程序员可以编写在普通 CGI 环境和 mod_perl 环境下均可执行的 CGI。但已有的 CGI 脚本恐怕要进行一些修改，

Mod_perl 的最大优点是功能强大和神奇的速度。你可以充分领略服务器内部的工作过程，接触各个处理阶段。这使得你可以对 URI 到文件名的转换、认证、响应生成和日志各阶段进行定制。用 mod_perl 编写的 Apache 模块能够完成用 C 写的 Apache 模块一样的功能。在这种模式下，你几乎不需要增加运行负载。

如下是众多志愿者开发的在 mod_perl 下可以使用的 Apache/Perl 模块：

Apache::Registry	运行不变的 CGI 脚本
Apache::Status	嵌入的解释程序的运行时状况
Apache::Embperl	在 HTML 文档中嵌入 Perl 代码
Apache::SSI	在 Perl 中实现 SSI
Apache::DBI	维护持久的 DBI 连接
Apache::Gateway	实现 HTTP/1.1 网关
Apache::GzipChain	压缩无用输出
Apache::Filter	过滤文档和脚本输出
Apache::Sandwich	生成输出页的页眉和页脚
Apache::TransLDAP	通过 LDAP 查询解析 URI 地址
Apache::ASP	实现 ASP
Apache::AuthenDBI	通过 DBI 实现基于数据库的身份认证
Apache::PHLogin	基于 PH 数据库的身份认证
Apache::DBILogger	通过 DBI 实现基于数据库的日志记录
Apache::Session	持久会话管理
Apache::Throttle	基于连接速度的内容协商

另外，嵌入服务器的常驻解释程序避免启动外部解释程序的开销，同时也避免了附加的 Perl 启动时间。例如：不需要像通常的 WWW 服务器 扩展那样启动一个独立的进程。扩展机制，例如通用网关接口（CGI），完全可以用 Perl 代码取代。这些代码实现请求处理的响应产生部分。

如果您对 Apache/Perl 感兴趣，或有什么建议或问题，可以加入 Apache/Perl 邮件列表组。给 modperl-request@apache.org 发邮件，在内容部分中写入"subscribe modperl" 字样。而标明"subscribe modperl-announce"，则会加入消息公告列表。

如下是有关的一些重要链接：

Ken Williams 追随者的邮件列表文档：
http://forum.swarthmore.edu/epigone/modperl

Frank Cringle 的 mod_perl FAQ：
http://perl.apache.org/faq/

Stas Bekman 的 mod_perl 开发者指南：
http://perl.apache.org/guide/

Lincoln Stein 和 Doug MacEachern 的书—— "Writing Apache Modules with and C"（用 Perl 和 C 书写 Apache 模块）：
http://www.modperl.com/

Andrew Ford 的 mod_perl 快速参考卡片：
http://www.refcards.com/

Vivek Khera 的 mod_perl 性能调整指南：
http://perl.apache.org/tuning/

Apache/Perl 模块列表：
http://www.perl.com/CPAN/modules/by-module/Apache/apache-modlist.html

Apache/Perl 主页：
http://perl.apache.org/ 或 http://apache.perl.org/

8.1.3. 编译和安装 mod_perl

Mod_perl 可以在 http://www.perl.com/CPAN/modules/by-module/Apache/ 找到。
在安装 Mod_perl 之前，你首先要具备如下条件：

win32 平台

尽管在 win32 平台上 Mod_perl 并非太稳定，但毕竟已能提供支持了，对 Windows 用户无疑是个福音，对各位练手足够了。
安装 Mod_perl 有一些必备条件：MSVC++ 5.0+、Apache 1.3-dev 或更高版本及 Perl 5.004_02 或更高版本。
mod_perl 不能与 ActiveState 版本兼容，而只能与“正式”的 Perl 一起使用，后者可以从 http://www.perl.com/CPAN/src/5.0/latest.tar.gz 拿到。
Jeffey Baker 现在维护着 mod_perl 的二进制版：
`http://www.perl.com/CPAN/authors/Jeffrey_Baker/`
在首先完成了 Apache 和 Perl 的安装之后，再安装 mod_perl。

1. 在 mod_perl 软件包目录下，运行命令：

```
perl Makefile.PL
nmake install
```

这将安装 mod_perl 的 Perl 部分，并二进制版本的安装文件。

2. 编译生成 ApacheModulePerl.dll
在 MS developer studio 集成环境下，
选择 "File -> Open Workspace ..."，

选择 "Files of type [Projects (*.dsp)]"
找到并打开 mod_perl-x.xx/src/modules/ApacheModulePerl/ApacheModulePerl.dsp

选择 "Tools -> Options -> [Directories]"
选择 "Show directories for: [Include files]"

你需要在其中增加上如下目录：

```
C:\apache_x.xx\src\include
```

（实际的路径可能是 C:\...\mod_perl-x.xx\src\modules\perl）

```
C:\perl\lib\Core
```

选择 "Project -> Add to Project -> Files"，增加下面的内容：

```
perl.lib        （即 C:\perl\lib\Core\perl.lib）
ApacheCore.lib（即 C:\Apache\ApacheCore.lib）
```

选择 "Build -> Set Active Configuration... -> [ApacheModulePerl - Win32 Release]"
选择 "Build -> Build ApacheModulePerl.dll"

你可能会看到有一些其实无大碍的警告错误，你可以进行如下设置消除之：

"Project -> Settings -> [C/C++] -> Category: [Code Generation] -> Use runtime library: [Multithreaded DLL]

3. 测试

一旦编译连接完 ApacheModulePerl.dll 并安装了 apache.exe 之后，你可以对 mod_perl 进行测试。具体做法如下：

执行 nmake test

4. 配置

你不必将 apache 进行重新编译，而只将 ApacheModulePerl.dll 拷贝到目录

```
$SERVER_ROOT/modules
```

中即可。

在 httpd.conf 配置文件中增加如下配置行：

```
LoadModule perl_module modules/ApacheModulePerl
```

务必保证'C:\perl\bin' 已在你的搜索路径（PATH）中，这样 apache 才能找到 perl.dll。

5. 调整

CGI::Switch 模块在 apache_1.3a1 下不能正常工作，因为 basic_http_header() 没有被 ApacheCore.dll 输出（export）。你需要将如下的片段：

```
use CGI::Switch ();
 my $q = CGI::Switch->new;
```

变成：

```
use CGI ();
 my $q = CGI->new;
```

这样在 Windows 平台下，你就可以享受 mod_perl 的强大功能了。

UNIX 平台

在 UNIX 平台下安装 Mod_perl，也有一些必备条件：Apache 1.2.6 或更高版本、Perl 5.004 或更高版本。

在 mod_perl 的源目录下，运行命令：

```
perl Makefile.PL
make
make test (optional)
make install
```

其中 Makefile.PL 会搜索 apache 源代码树进行配置，如果未发现，会提示你输入一个路径。

当被问及：

```
"Configure mod_perl with ../apache_xxx/src ?"
```

时，回答 'y' 表示告诉 Makefile.PL 在指示的位置可以找到 src/*.h 文件。

当被问及：

```
"Shall I build httpd in $adir for you?"
```

时，回答 'y' 会去运行../apache_xxx/src/Configure。

运行 'make' 会对 Apache 进行编译和连接。

若想避免这些提示，并给出 apache 源代码所在位置，可以使用如下命令：

```
perl Makefile.PL DO_HTTPD=1
```

要避免提示和对 httpd 的编译，使用如下命令：

```
perl Makefile.PL NO_HTTPD=1
```

任何情况下，你都需要执行 'make install' 来安装 mod_perl 的 Perl 端。

缺省情况下，所有的回调句柄除了 PerlHandler 均被关闭。

你可以编辑 src/modules/perl/Makefile，或在运行 Makefile.PL 时指示打开开关：

可以使用的参数有：

```
PERL_POST_READ_REQUEST
PERL_TRANS
PERL_INIT

PERL_HEADER_PARSER
PERL_AUTHEN
PERL_AUTHZ
PERL_ACCESS
PERL_TYPE
PERL_FIXUP
PERL_LOG
PERL_CLEANUP
PERL_CHILD_INIT
PERL_CHILD_EXIT
PERL_DISPATCH

PERL_STACKED_HANDLERS
```

```
PERL_METHOD_HANDLERS
PERL_SECTIONS
PERL_SSI
```

例如：要支持 PerlAuthenHandler **和** PerlFixupHandler**，采用如下方法：**

```
perl Makefile.PL PERL_AUTHEN=1 PERL_FIXUP=1
```

要支持回调：

```
perl Makefile.PL ALL_HOOKS=1
```

要支持上述所有的功能设置 EVERYTHING=1**。**

```
perl Makefile.PL EVERYTHING=1
```

要支持跟踪功能，设置 PERL_TRACE=1**。**

```
perl Makefile.PL PERL_TRACE=1
```

如果文件“ makepl_args.mod_perl **”存在于与** mod_perl **相同的目录中，则它将被** Makefile.PL **读取。**

例如：

```
% ls -1 .
apache_1.3/
makepl_args.mod_perl
mod_perl-1.12/

% cat makepl_args.mod_perl
EVERYTHING=1 DO_HTTPD=1

% cd mod_perl-1.12
% perl Makefile.PL && make test && make install
```

你会发现所有的选项均被允许，你不会再被要求输入 apache **所在路径，缺省值是**../apache_1.3**。**

在 eg/**目录中，有一个名为** makepl_args.mod_perl **的示例文件，其中你可以发现使用了一些控制新增的实验性功能的指令，可是非常的酷！**

8.1.4. 在 mod_perl 上运行 Perl CGI

如上文所说，mod_perl **可以替代普通的** CGI **工作方式。尽管这仅是** mod_perl **中很小一部分功能，但却是应用最广的。本部分对该功能进行描述。**

读者可能认为要进行复杂的配置。实际上，一般地你只需要执行如下的命令即可。

```
perl Makefile.PL && make && make install
```

实现对 CGI **的替代的具体的配置细节描述如下：**

```
Alias /perl/  /real/path/to/perl-scripts/

<Location /perl>
SetHandler  perl-script
PerlHandler Apache::Registry
Options +ExecCGI
</Location>
```

这里的“ Location **”指** URI **地址，而不是指一个目录，可以认为上面的路径就是：**

```
<Location http://www.yourname.com/perl>
```

在该路径下的任何文件（在你本地文件系统的/real/path/to/perl-scripts/下）将会被 Apache::Registry 模块处理，由后者还模拟一个 CGI 环境。该文件必须存在并是可执行的，而且 'Options ExecCGI' 必须打开。

如果你想基于文件名后缀来用 mod_perl 执行脚本，则可以进行如下设置：

```
<Files ~ "\.pl$">
 SetHandler perl-script
 PerlHandler Apache::Registry
 Options ExecCGI
 </Files>
```

注意，ScriptAlias 对 mod_perl 没有影响。

如果你一直使用 Perl 5.004，那么大多数 CGI 脚本都可以不需任何修改而在 mod_perl 下运行。但若你使用的是 5.003，则需要进行一些修改，因为 Perl 内嵌的 read() 和 print() 函数在 mod_perl 下无法正常工作。如果你使用的是 CGI.pm，则使用$query->print 来取代 print()。

缺省情况下，mod_perl 不发送任何 HTTP 头域，但你可以改变它：

```
   PerlSendHeader On
```

这样 mod_perl 会以与 mod_cgi 相同的方式发送响应行和普通 HTTP 头域。而且和 mod_cgi 一样，PerlSendHeader 也不发送换行符，你必须在脚本中自己给出。例如：

```
print "Content-type: text/html\n\n";
```

如果你使用 CGI.pm 且用了 'print $q->header',那么你就不必再用 PerlSendHeader On 了。

要运行一个`nph' 类型的非分析头标的 CGI 脚本，你需要在每个输出文件处理的写或打印操作后强制进行输出清除，这时增加下面一行即可：

```
local $| = 1;
```

如果你设置了 PerlSendHeader On，你需要把如下语句加到 httpd.conf 文件中：

```
<Files */nph-*>
 PerlSendHeader Off
 </Files>
```

曾编过 CGI 程序的读者可能会感觉非常的琐碎（sloppy programming），而 mod_perl 则不然。因为 CGI 脚本作为单独的一个进程执行，它的生命周期就对应一个 HTTP 请求。当请求结束时，进程退出，一切占用的资源都会被清除，如全局变量空间、打开的文件等。在 mod_perl 下运行的脚本则有更长的生命期，可以跨越多个请求，在同一进程中可以存在不同的脚本。这意味着你必须自己指定清除它。

你最好用严格的方式去写脚本，写干净利落的语句，同时不要忘记使用-w 开关!!!

这一点对于 mod_perl 来说比其它都重要。它并不是必须的，但强烈建议你这样做，它会节省你很多时间。当然，显式地清除脚本的操作在普通 CGI 环境下也能正常工作。

8.1.5. 基于 mod_perl 写 Perl 模块

安装和配置好了 mod_perl，就可以开始开发新的基于 mod_perl 的脚本了。先举一例，讲述如何创建一个很简单的 Perl 模块（一个面向对象的 Perl 脚本）。如下为模块 Test.pm，其功能非常简单，只是打印出环境变量。

```
package Apache::Test;
```

```
# Tell Perl to show warning when a variable is used without
# a declaration. This is done using 'use strict.'
use strict;

sub handler {
#
# Purpose: this method gets called by mod_perl. So this is
# really the entry point into this module. In other words,
# this is the traditional main function.
#

# Assign the request object (passed as a parameter) to a
# local variable.
My $r = shift;

# Local temporary variables
my $key;

# Assign a local associative array variable to the hash
# returned by the request object variable's cgi_env()
# methods
#
my %ENV = $r->cgi_env;

# Set the content-type of the output to text/html
#
$r->content_type("text/html");

# Print out the HTTP headers
$r->send_http_header;

# Now loop through the %ENV array and print out the
# key=value pair for each environment variable.
#
foreach $key (keys %ENV) {
        $r->print("$key = $ENV{$key} <BR>.");
}
# Done
return 1;
}

# The following line is required for a Perl module.
1;
_END_
```

在这个例子中，只有一个称为 handler 的子程序。如果象下面那样指定配置行：

```
<Location /test>
SetHandler perl-script
PerlHandler Apache::Test
</Location>
```

那么无论什么时候当 Apache 接到包含/test/的请求时，都将调用 handler 方法。如果计划管理 Apache 请求过程，那么需要非常了解 Apache API。在若干请求阶段中，Apache API 允许模块进入并做一些事情。Apache 文档描述了请求的不同阶段和你的模块能做什么。在默认情况下，这些 hooks 在编译时是被禁止的。关于激活这些 hooks 的信息请参阅有关的安装文档。

8.1.6. 与 SSI 的结合使用

mod_perl 还可以和 SSI 脚本集成。 需要做的只是确保在建立 Apache 二进制文件时，在配置文件中有如下内容：

```
EXTRA_CFLAGS=-DUSE_PERL_SSL  -I. 'perl  -MextUtils::Embed -ccopts'
```

或者确保在建立 mod_perl 时使用如下命令：

```
perl Makefile.PL PERL_SSI=1
```

一旦建好了 Apache，就可以利用 mod_perl 的优点为你的 Perl SSI 脚本服务。如果想取代下面的 SSI 命令：

```
<!-#exec cmd="/some_directory_alias/your_perlscript" ->
```

可以在 SSI 页中使用如下内容：

```
<!-#perl sub="Apache::Include" arg="/some_directory_alias/your_perlscript"
->
```

同时需要在 httpd.conf 配置文件中用如下命令来预装脚本：

```
PerlScript /path/to/your_perlscript
```

在前面的例子中，SSI 调用中的 sub 键值可以是子程序名、模块名、某类方法名，或者是一个匿名的子程序调用（sub{}）。例如：

```
<!-#perl sub="Apache::Test" ->
```

在默认情况下，这条命令将调用 Test::handler 方法，只要 Test.pm 在服务器启动时通过 PerlModule 或其他方法预装。

如果想调用的不是缺省的 handler 方法，需要同时给出方法名。

还有一个传参数的问题。例如将如下的语句加入 html 文件中：

```
  <!--#perl sub="Apache::Include" arg="/perl/ssi.pl" -->
```

要想给 ssi.pl 脚本传递参数，怎么办呢？

仅象如上的格式 Apache::Include 是无法正常工作的。不用脚本，而应该定义一个通过 PerlRequire 或 PerlModule 引入的方法，例如：

```
  sub My::ssi {
     my($r, $one, $two, $three) = @_;
     ...
  }
```

则在 html 文件中的格式是：

```
  <!--#perl sub="My::ssi" arg="one" arg="two" arg="three" -->
```

8.1.7. 使用 Perl 来配置 Apache

除了在 CGI 或 SSI 脚本中使用 mod_perl所用到的优点外，mod_perl还有不寻常的特征，热心的 Perl 程序员和 Apache 管理者一定会喜欢这个特征——它具有用 Perl 写 Apache 服务器配置的能力。

可以在 Apache 配置文件中加入<Perl>配置片段，如下所示：

```
<Perl>
#插入你的 Perl 代码在此
1；
</Perl>
```

其中，最后一行（包含 1；）是需要的。你的代码可以是任何 Perl 脚本。<Perl>配置片段中的代码在特殊的软件包中编译，mod_perl 把配置信息传给 Apache 内核配置模块。在使用这个特征前，需要确保正确地建立了 Apache 和 mod_perl。为了给 Apache 服务器配置这个特征，需要这样配置 mod_perl：

```
perl Makefile.PL PERL_SECTIONS=1
```

Apache 和 mod_perl 被配置和重新编译后，就可在 Apache 配置文件中写 Perl 代码了。下面说明用于描述配置的语法。

取单个值的命令可以作为标量。例如：

```
User httpd
```

这个 User 命令的值是个字符串，因此可被写为：

```
<Perl>
$User = "httpd";
1;
</Perl>
```

下面是配置的例子：

```
<Perl>
$User = "httpd";
$ServerAdmin = "kabir@nitec.com";
$MinSpareServers = 5;
$MaxSpareServers = 5;
$MaxClients = 40;
1;
</Perl>
```

需要多值的命令可以用列表表示。例如，PerlModule Apache::TestOne Apache::TestTwo 可以表示为：

```
@PerlModule = qw(Apache::TestOne Apache::TestTwo);
```

配置片段可以改头换面来表示，例如：

```
<VirtualHost 206.171.50.50>
ServerName www.nitec.com
ServerAdmin kabir@nitec.com
</VirtualHost>
```

可以被表示为：

```
$VirtualHost{"206.171.50.50"} = {
ServerName => ' www.nitec.com',
```

```
ServerAdmin => 'kabir@nitec.com'
}
```

下面是另一个稍复杂的例子：

```
$Location{"/some_dir_alias/"} = {
AuthUserFile => '/www/nitec/secret/htpasswd',
AuthType => 'Basic',
AuthName => 'Subscribers Only Access',
DirectoryIndex => [qw(welcome.html welcome.html)],
Limit => {
            METHODS => POST GET',
        Require => 'user reader'
}
};
```

可以用类似的方法定义其他的配置片段，例如<Directory>、<File>**等。注意：为了使**<Perl>**片段能工作，需要在配置文件中定义** PerlScript **命令。如果不需要装入任何** Perl **脚本，可以象下面这样设置** PerlScript **命令来利用**<Perl>**配置片段：**

```
PerlScript /dev/null
```

建议只在出现下列一种或几种要求时才在配置文件种使用<Perl>**配置片段：**

你非常喜欢 Perl**，不能想象既然在** Apache **配置中能用**<Perl>**为什么不用它。**

运行了多个 Apache **服务器，并希望创建一组能用在所有** Apache **服务器系统上的配置文件（这正是希望使用的地方）。一个使用脚本的配置非常灵活。它可以读外部文件，使用循环来自动配置等等。这样的配置只是受限于** Apache **管理者的编程能力。**

如果为第一种情况这样做，那么应该在你的配置文件中用自己的风格写了有用的脚本。如果处于第二种情况下，并且不知道怎样利用 Perl **的这一特征，可以看一看下面的例子：**

假设有三个 Web **服务器系统：**host_a, host_b, host_c **，其中** host_a **比** host_b **更加强有力，**host_b **比** host_c **更强有力。现在，希望定义一个**<Perl>**容器，该容器允许为这三个主机创建单一的配置。下面就是这样一个** httpd.conf **文件：**

```
<Perl>
# Get the host name using the UNIX hostname utility and store it
# in the $thisHost variable.
my $thisHost = '/bin/hostname';
if ($thisHost =~ /host_a/) {
        # configuration for host_a goes here
        $MinSpareServers = 10;
        $MaxSpareServers = 20;
    $StartServers = 30;
    $MaxClients = 256;
}
elsif ($thisHost =~ /host_b/) {
        # configuration for host_b goes here
        $MinSpareServers = 5;
        $MaxSpareServers = 10;
    $StartServers = 10;
    $MaxClients = 50;
```

```
}
else ($thisHost =~ /host_c/) {
        # configuration for host_c goes here
        $MinSpareServers = 3;
        $MaxSpareServers = 5;
    $StartServers = 5;
    $MaxClients = 30;
}
1;
</Perl>
```

为了使这个例子更有趣，假设这三个主机中的每一个都有不同的虚拟主机，我们希望这些虚拟主机以简练的方式配置。看看这个例子：

```
<Perl>
# Get the host name using the UNIX hostname utility and store it
# in the $thisHost variable.
my $thisHost = '/bin/hostname';
my $thisDomain = 'mydomain.com';
my @vHosts = ();
my $anyHost;

if ($thisHost =~ /host_a/) {
        # configuration for host_a goes here
    @vHosts = qw(gaia, athena, romeo, juliet, shazam);
}
elsif ($thisHost =~ /host_b/) {
    @vHosts = qw(catbart, ratbart, dilbert);
}
else ($thisHost =~ /host_c/) {
    @vHosts = qw(lonelyhost);
}

for $anyHost (@vHosts) {
    %{$VirtualHost{"$anyHost.$domainName"}} = {
        "ServerName" => "$anyHost.$domainName",
        "ServerAdmin" => "webmaster\@$anyHost.$domainName"
    }
}

1;
</Perl>
```

一旦为 Apache 服务器创建了合适的基于 Perl 的配置，就可以用下面的命令检查代码语法，确保代码的语法正确性。

```
perl -cx httpd.conf
```

8.1.8. 安全性问题

mod_perl 脚本的安全性如何呢？

因为 mod_perl 在 httpd 的子进程中运行，具有 httpd.conf 中指定的 user-id 和 group-id。这里的 user/group 应该具有可能的最低的权限。它应该只能有读被允许读的文件的权限。即便如此，粗心大意也会带来麻烦。例如你不应让 /etc/passwd 由于此原因向整个网络洞开。

不同 mod_perl 脚本可以使用同一 Perl 解释器实例。所以除了经典的 CGI 问题之外，一个恶意的 mod_perl 脚本可以重新定义任何 Perl 对象并改变其它 mod_perl 脚本的行为。

如果你把 tainting checks 的开关打开，perl 可以帮助你避免使用来自网络的数据的危险。在 mod_perl 脚本的第一行设置 -T 开关并不足以进行 tainting checks。你还必须在 httpd.conf 文件中包含配置命令 PerlTaintCheck On。

如果某个脚本需要有较高的级别，怎么办呢？

你必须重新启动一个进程，使用相应合适的 user-id （或 group-id）。如果所有处理的请求都需要较高的权限，则你也可以写一个具有 suid 权限的 CGI 脚本。请参阅 Apache 文档中关于 suEXEC 的部分。或者，也可创建一个具有 suid 权限的子进程来专门处理需要特别处理的任务。

第2节 Java Servlet 扩展模块

8.2.1. Servlet 基础

什么是 Servlet

读者想必对 Java applet 都很熟悉了。applet 是在客户端的浏览器中运行来提供动态性的，而 servlet 则是在 WWW 服务器端可用来取代 CGI 的一种技术，两者都是基于 Java 平台的。原来有个说法是：Java 性能不好，只在客户端有一定市场，在服务器端并不适用，现在看来并非如此，因为有了 servlet。servlet 可以认为是 Java 在服务器端占据的一席之地。

Servlet 可以定义为：基于 Java Servlet API 开发的在支持 Java 的服务器端运行的类 CGI 的 Java 程序，它支持页面动态生成等。

servlet 能够象 CGI 脚本一样扩展 WWW 服务器功能，但是 servlet 占用资源很少，有很多用 CGI 程序支持的一些站点由于访问量剧增，性能迅速下降，这是 CGI 的一大缺点。同时由于 servlet 是用 Java 编写的，因此是跨平台的。实际上，servlet 是电子商务真正的开始。Servlet 可以相当有效地替代 CGI 脚本，它可以方便地产生容易编写而且运行快的动态文本，可以方便地调试，找出程序中的问题。

Servlet API，是用来写 servlet 的，是外部环境的抽象，使得 servlet 不必象 CGI 脚本那样再去关心诸如 servlet 是怎样被装载的，servlet 运行的服务器环境是什么，或者用来传输数据的协议是什么等等问题，这样 servlets 就可以应用到融合在不同的 web 服务器中。但 Java Servlet API 不是 Java 核心框架的一部分，而是一个标准的 Java 扩展包，可以作为通用的附加产品包免费或付极少费用购买使用。

下面是一些 servlet 应用范围：

- 用于处理 HTML FORM 通过 HTTPS 产生 POSTed 数据，包括买卖订单或信用卡数据。 因此 servlet 可以成为订单处理系统的一部分，和产品存货数据库一道工作，也许可以用在在线支付系统上。
- 允许人们之间的合作。 一个 servlet 能并发处理多个请求；它们可以使用在诸如在线会议这样的同步请求支持系统上。
- 转送请求。Servlet 可以转送请求给其他的服务器和 servlets。 这就允许在镜像同样内容的几个服务器之间平衡负载。按照任务类型或组织范围，可以支持在几个服务器中划分逻辑上的服务器。

Servlet 结构透视

在具体掌握 servlet 之前，须对 Java 语言有所了解。下面假设您已经有了 Java 基础。

在 Servlet API 中最重要的是 Servlet interface（接口）。所有的 servlet 必须 implement（实现）这个接口， 但具体的方式多种多样。或者是直接的，或者通过 extending（继承）实现了这个接口的某个类，如 HttpServlet。这个 servlet 接口提供了 servlet 与客户端通信的方法。Servlet 程序员可以在他们开发 servlet 程序时使用某些或所有方法。

当一个 servlet 接收来自客户端的请求时，它接收两个对象： 一个是 ServletRequest 接口的实例，另外一个是 ServletResponse 接口的实例。ServletRequest 接口抽象了从客户端到服务器之间的联系，而 ServletResponse 接口则抽象了从 servlet 返回客户端的联系。

ServletRequest 接口可以提供这样一些信息，如提出请求的客户名称、客户端正在使用的协议、产生请求并且接收请求的服务器远端主机名等。它也提供获取数据流的 servlet——ServletInputStream，其中的数据是客户端使用 HTTP 协议的 POST 和 PUT 方法递交的。ServletRequest 的子接口可以让 servlet 获取更多的协议特性数据。例如 HttpServletRequest 接口包含了获取 HTTP 协议特定头域信息的方法。

ServletResponse 接口给出了响应客户端的 servlet 方法。它允许 servlet 设置响应内容的长度和 MIME 类型，并且提供输出流 ServletOutputStream 对象。开发者可以通过该对象返回响应数据。ServletResponse 子类或接口可以给出更多的协议相关的方法或操作。例如 HttpServletResponse 包含允许 servletHTTP 对响应头域信息进行操作的方法。上面有关类和接口的描述构成了一个基本的 Servlet 框架。HTTP servlets 有一些可提供会话跟踪能力的方法。servlet 开发者可以用这些 API 在会话进行时维护 servlet 与客户端之间的状态。

8.2.2. Servlet 的工作方式

Servlet 的工作方式或说它的整个生命周期可简单描述为：服务器装载运行 servlet；servlet 接收来自客户端的多个请求并且返回数据给客户端；最后服务器卸载装入的 servlet。

详细的过程如下所述：

当一个服务器装载 servlet 时，它运行 servlet 的 init 方法。这个方法不能反复调用，一旦调用就是再装载 servlet。只有到服务器调用 destroy 方法卸载 servlet 后才能再调用 init 方法。

在服务器装载 servlet 并进行初始化后，servlet 就能够处理客户端的请求了，用 service 方法可以做到这一点。根据每个客户端请求的情况分别使用不同的 service 方法来完成处理：接收客户端请求， 并且发回相应的响应。

Servlet 同时运行多个请求的能力非常重要，但这也需要按多线程（thread-safe）样式编写 service 方法。 如：service 方法更新 servlet 对象中的一个属性 field，这个属性可以同时被存取。假如某个服务器不能同时并发运行 service 方法，也可以用 SingleThreadModel 接口来实现 Servlet。这个接口保证不会有两个以上的线程并发运行。

Servlet 一直运行到它们被服务器卸载。在 servlet 的生命周期中，编写多线程形式的代码要求特别注意 servlet 的卸载的处理。

8.2.3. 编写 Servlet

对于 servlet 的编写本不是本书的内容，但为了能使读者有一个完整的和直接的体验。这里简单地谈一下 servlet 的编程。

Servlet 实现了 javax.servlet.Servlet 接口。Servlet 开发者可以通过直接实现该接口来开发，但这样通常没有必要。因为大多数 servlet 是针对用 HTTP 协议的 WWW 服务器，这样开发 servlet 的最通用方法是去继承 javax.servlet.http.HttpServlet 类。

HttpServlet 类通过继承基类 GenericServlet 实现了 Servlet 接口。HttpServlet 提供了处理 HTTP 协议信息的功能，它的 service 方法支持标准 HTTP/1.1 请求。

HttpServlet 类中有几个缺省的方法，你可以自己重载这些方法来实现一些特定的处理：

doGet： 用于处理 GET、有附加条件的 GET 和 HEAD 请求

doPost： 用于处理 POST 请求

doPut： 用于处理 PUT 请求

doDelete： 用于处理 DELETE 请求

缺省情况下，HttpServlet 的 service 方法，当接收到一个 OPTIONS 请求时，会调用 doOptions 方法；当接收到一个 TRACE 请求时调用 doTrace 方法。doOptions 缺省执行方式是自动决定何种 HTTP 选项支持且返回该信息。

与客户端的交互

在你使用上述方法时，必须带两个参数：第一个包含来自客户端的数据 HttpServletRequest；第二个参数包含对客户端的响应 HttpServletResponse。在下例中是这样的情况。

HttpServletRequest 对象提供 HTTP 请求头信息。具体怎样获取这些数据取决于具体的 HTTP 请求方法。

- 不管任何 HTTP 方法，你可以用 getParameterValues 方法，来获取特定名称的参数值。
- 对于 HTTP GET 请求，使用 getQueryString 方法可以返回一个可用来解析的字符串。
- 对于 HTTP POST、PUT 和 DELETE 请求，你有两种方法可以选择。如果是文本数据，你可通过调用 getReader 方法返回的 BufferedReader 对象来获取；如果是二进制数据，则可通过调用 getInputStream 方法返回的 ServletInputStream 对象来获取。

HttpServletResponse 对象提供了响应客户返回数据的两种方法。你可以用 getWriter 方法返回的 Writer 对象，也可用 getOutputStream 方法返回的 OutputStream 对象。你应该用前者返回文本数据，而用后者返回二进制数据。

在使用 Writer 或 OutputStream 返回具体数据之前，应该先设置 HTTP 头域。HttpServletResponse 提供了设置 Content-type、Content-encoding 和 Content-length 等头域的

方法。而后，你可以用 writer 或 outputstream 将响应主体部分发回用户。完成后要关闭 writer 或 output stream 以便让服务器知道响应已经完毕。

GET 和 HEAD 方法的处理示例——SimpleServlet

如下是一个处理 GET 和 HEAD 方法的 HTTP Servlet 的例子（你可以在 JSDK 包的 examples 子目录中找到本例及下例代码）：

```
public class SimpleServlet extends HttpServlet {
    public void doGet(HttpServletRequest req, HttpServletResponse res)
      throws ServletException, IOException
    {
      // 首先设置头部
      res.setContentType("text/html");

      // 用 writer 方法返回响应数据
      PrintWriter out = res.getWriter();
      out.println("<HEAD><TITLE>SimpleServlet
Output</TITLE></HEAD><BODY>");
      out.println("<h1> SimpleServlet Output </h1>");
      out.println("<P>This is output is from SimpleServlet.");
      out.println("</BODY>");
      out.close();
    }

    public String getServletInfo() {  return "A simple servlet";    }
}
```

POST 方法的处理示例——SurveyServlet

首先给出对应的客户端包含 POST FORM 的 HTML 文件：

```
<html>
  <head><title>JdcSurvey</title></head>
  <body>
    <form action=http：//localhost:8080/servlet/survey method=POST>
    <input type=hidden name=survey value=Survey01Results>
        <BR><BR>How Many Employees in your Company?<BR>
          <BR>1-100<input type=radio name=employee value=1-100>
          <BR>100-200<input type=radio name=employee value=100-200>
          <BR>200-300<input type=radio name=employee value=200-300>
          <BR>300-400<input type=radio name=employee value=300-400>
          <BR>500-more<input type=radio name=employee value=500-more>

        <BR><BR>General Comments?<BR>
```

```
          <BR><input type=text name=comment>

        <BR><BR>What IDEs do you use?<BR>
         <BR>JavaWorkShop<input type=checkbox name=ide value=JavaWorkShop>
         <BR>J++<input type=checkbox name=ide value=J++>
         <BR>Cafe'<input type=checkbox name=ide value=Cafe'>

        <BR><BR><input type=submit><input type=reset>
    </form>
  </body>
</html>
```

下面给出处理上述 POST 方法的 HTTP Servlet：

```
public void doPost(HttpServletRequest req, HttpServletResponse res)
  throws ServletException, IOException
{
  // 首先设置响应的 "content type" 头部
  res.setContentType("text/html");

  //得到响应的 PrintWriter 以返回文本给客户端
  PrintWriter toClient = res.getWriter();
  try {
    //打开一个文件写入 Survey 的结果。
    String surveyName = req.getParameterValues("survey")[0];
    FileWriter resultsFile = new FileWriter(resultsDir
        + System.getProperty("file.separator")
        + surveyName + ".txt", true);
    PrintWriter toFile = new PrintWriter(resultsFile);

    // 从客户端得到表单数据 & 存贮在这个文件中
    toFile.println("<BEGIN>");
    Enumeration values = req.getParameterNames();
    while(values.hasMoreElements()) {
        String name = (String)values.nextElement();
        String value = req.getParameterValues(name)[0];
        if(name.compareTo("submit") != 0) {
          toFile.println(name + ": " + value);
        }
    }
    toFile.println("<END>");

    //关闭文件。
    resultsFile.close();
    // 用一个 thank you 返回客户端
    toClient.println("<html>");
```

```
        toClient.println("<title>Thank you!</title>");
        toClient.println("Thank you for participating");
        toClient.println("</html>");
      } catch(IOException e) {
        e.printStackTrace();
        toClient.println(
            "A problem occured while recording your answers. "
            + "Please try again。");
      }

      // 关闭 writer; 响应完成。
      toClient.close();
    }
```

该 Survey servlet **将** FORM **数据写入一个文件，并且用信息“** thank you **”响应用户。**

其中 doPost **方法用** getParameterNames **和** getParameterValues **方法来从** FORM **中获取数据。因为它返回给客户的是文本信息，故** doPost **调用** getWriter **方法。在写响应体部分之前，它进行了响应头域的设置。整个响应完成后，关闭** Writer **对象。**

生命周期方法

重载 init 方法

在初始化过程中，servlet **要准备好它所需要的资源，以便能够接收请求。这里不必考虑多线程，因为** servlet **的初始化过程是单线程的。一旦** init **方法完成，**servlet **就能接收客户端的请求了。而如果初始化不能成功，这个方法会抛出** UnavailableException **异常。**

Init **方法使用** ServletConfig **对象作为参数。这个对象应该保存起来，以便能在调用** getServletConfig **方法时返回之。实现的最简单的办法是，在新的** init **方法中调用** super.init**。不这样的话，你必须自己编写保存** ServletConfig **对象的代码，并且重载** getServletConfig **方法，以便它能从新的位置取到** ServletConfig **对象。**

下面是一个初始化方法的例子。它来自上面的 Survey Servlet**，该** servlet **从** FORM **中接收输入然后存到文件中。为了存储调查信息，它需要接收一个目录名为初始化参数。**

```
    public void init(ServletConfig config)
      throws ServletException
    {
      super.init(config);
      //获取目录
      resultsDir = getInitParameter("resultsDir");

      //如果没有目录，不处理客户请求
      if (resultsDir == null) {
        throw new UnavailableException (this,
          "Not given a directory to write survey results!");
```

```
        }
    }
```

上述 init 方法调用 super.init 方法来处理 ServletConfig 对象，并且取以初始化参数提供的目录名来设置 resultsDir 值。如果没有提供目录名，该 servlet 抛出一个“unavailable（不存在）”的异常。如果 init 方法成功完成，该 servlet 就能处理客户端请求了。

初始化参数

初始化参数的规范是与服务器的调用方式相关的。但无论初始化参数如何指定，获取的方法却是相同的：以要取的参数名为参数调用 getInitParameter 方法。上面的例子中便有体现。你若要获取所有的参数的名字，可以用 getParameterNames 方法。

重载 destroy 方法

当服务器卸载一个 servlet，它将调用 servlet 的 destroy 方法。destroy 方法将清除初始化方法中的动作。本部分首先讲述如何编写一个简单的 destroy 方法，然后给出当 destroy 方法被调用时 service 方法仍在运行情况的处理办法。

若在 servlet 中重载了 init 方法，则一般也要重载 destroy 方法，以便将对应的初始化操作清除（如释放资源等）。虽然并不总是如此。

比如一个 servlet 在初始化时打开了一个数据库并建立了连接，则对应的 destroy 方法需要去关闭这个连接。该 destroy 方法如下所示：

```
/**
 * 关闭数据库连接
 */
public void destroy() {
   try {
      con.close();
   } catch (SQLException e) {
      while (e != null) {
           log("SQLException: " + e.getSQLState() + '\t' +
              e.getMessage() + '\t' +
              e.getErrorCode() + '\t');
          e = e.getNextException();
      }
   } catch (Exception e) {
      e.printStackTrace();
   }
}
```

Servlet 终止时对 service 方法的处理

当服务器卸载 servlet 时，会去调用对应的 destroy 方法。但在此之前，需要等待该 servlet

所有的 service 方法调用执行完，或者等待服务器规定的某一段时间（称为宽限期），只要两者有一个完成，destroy 方法便会被调用。如果 servlet 操作完成需时较多（比宽限期长），则在 destroy 方法被调用时还会有线程在运行。因此一个严谨的 servlet 程序员需要确保调用 destroy 方法时所有处理客户请求的线程都已运行结束。下面将讲述有关的处理技术。

对于处理客户请求时间可能较长的 servlet 来说，应当跟踪其当前运行的所有 service 方法。所有这些方法应周期性地循环执行，以便它们都能够取得进展。而若 servlet 被销毁（destroy 方法被调用），则这些耗时长的 service 方法应该立即停止工作，并进行必要的清除工作，然后返回。

比如，可以用称为 serviceCounter 的变量来统计有多少个 service 方法在运行，用称为 shuttingDown 变量来指示 servlet 是否正被销毁（即 destroy 方法是否正在被调用）。每个变量可以有各自的存取方法：

```
public ShutdownExample extends HttpServlet {
    private int serviceCounter = 0;
    private Boolean shuttingDown;
    ...
    // serviceCounter 的存取方法
    protected synchronized void enteringServiceMethod() {
        serviceCounter++;
    }
    protected synchronized void leavingServiceMethod() {
        serviceCounter--;
    }
    protected synchronized int numServices() {
        return serviceCounter;
    }
    //shuttingDown 的存取方法
    protected setShuttingDown(Boolean flag) {
        shuttingDown = flag;
    }
    protected Boolean isShuttingDown() {
        return shuttingDown;
    }
}
```

每次进入 service 方法时均要增加计数，而在退出时则减少计数。

```
protected void service(HttpServletRequest req, HttpServletResponse
resp)
    throws ServletException, IOException
{
    enteringServiceMethod();
    try {
        super.service(req, resp);
    } finally {
        leavingServiceMethod();
```

```
        }
    }
```

同时，destroy 方法也应当检查 serviceCounter。而且如果存在运行耗时长的方法的话，要设置 shuttingDown 变量。这个变量可以让正在处理请求的线程知道：该结束了，关闭吧！destroy 方法应当等待运行的 service 方法完成，从而实现一个“干净”的关闭。

```
public void destroy() {
    /* 检查是否有线程在运行，如果存在，告诉它们结束运行*/
    if (numServices() > 0) {
        setShuttingDown(true);
    }

    /* 等待它们结束 */
    while(numService() > 0) {
        try {
            thisThread.sleep(interval);
        } catch (InterruptedException e) {
        }
    }
}
```

长耗时的请求处理方法在必要时应当检查这个变量，并且适时结束自己的工作：

```
public void doPost(...) {
    ...
    for(i = 0; ((i < lotsOfStuffToDo) && !isShuttingDown()); i++) {
        try {
            partOfLongRunningOperation(i);
        } catch (InterruptedException e) {
        }
    }
}
```

提供关于 Servlet 的信息

某些 applet 和应用能显示关于 servlet 的某些信息，其中可以包含 servlet 编写意图的简单描述、作者及其版本号。Servlet API 中提供了 getServletInfo 方法来获取上述信息。缺省情况下，该方法返回 null。你可以重载该方法来提供一定的信息。如下所示：

```
/**
 * This is a simple example of an HTTP Servlet.  It responds to the GET
 * and HEAD methods of the HTTP protocol.
 */
public class SimpleServlet extends HttpServlet {
  ...
    public String getServletInfo() {
        return "A simple servlet";
    }
```

```
    }
```

当然重载该方法并不是必须的。

8.2.4. 用 servletrunner 来运行 Servlet

一旦写好 servlet**，你可以在很多** web **服务器上运行它，也可以用** JSDK **中提供的** servletrunner **工具来运行。**

使用 servletrunner **运行** servlet**，你必须使用属性。**

属性

属性是一个键-值对，用作 servlet **的配置、创建和初始化。比如** servlet.phone.code = PhoneServlet **就是一个属性，它的键是** servlet.phone.code**，值是** PhoneServlet**。**

一个 servlet **有两个属性。一个是** servlet.name.code**，它的值是** servlet **的类名。另一个是** servlet.name.initargs**，它的值存放** servlet **的初始化参数。**

code 属性的使用

servlet.*name*.code **属性通过在其中的** *name* **部分和** servlet **类间建立关联来命名** servlet**。如果你的** servlet **要使用初始化参数，就必须定义这个属性。它使服务器在** servlet **对象和初始化参数间建立关联：这是通过在它们的名字（**servlet.name.code **和** servlet.name.initargs**）中使用相同的** *name* **部分来实现的。即使你的** servlet **没有使用初始化参数，也推荐使用这个属性，以便客户端能用名字来访问** servlet**。**

Initargs 属性的语法

servlet.name.initArgs **属性的值存放** servlet **初始化参数的值。单个参数的语法是：**

参数名=参数值

一个完整的属性（整个键-值对）必须占单独的一行。为方便阅读，你可以用“ \ **”来在连续的多行中定义一个属性。下面是用于名为** phone **的** servlet **的参数定义：**

```
servlet.phone.initArgs=\
  phonelist=servlets/phonelist
```

如果有多个初始化参数，用“，”间隔开，比如一个访问数据库的名为 dbdemo **的** servlet **的参数定义形式如下：**

```
servlet.dbdemo.initArgs=\
  username=fill_in_the_user, \
  password=fill_in_the_password, \
  owner=fill_in_the_name
```

属性文件

你可以用文件存放需要定义的所有属性。该文件通常命名为“servlet.properties”，它应保存所有将用 servletrunner 运行的 servlet 的属性。 该文件应是文本格式的，可以在编辑器中创建、编辑。如下是一个 servlet.properties 文件的片段：

```
# phone servlet (sample.html)
servlet.phone.code=PhoneServlet
servlet.phone.initArgs=\
     phonelist=servlets/phonelist

# bulletin board servlet
servlet.bboard.code=BBoardServlet

# order entry servlet
servlet.dbdemo.code=OrderEntryServlet
servlet.dbdemo.initArgs=\
       username=fill_in_the_user,\
       password=fill_in_the_password,\
       owner=fill_in_the_name
```

ServletRunner 工具的使用

本部分讲解如何使用 JSDK 中包含的 servletrunner 工具运行 servlet。

servletrunner 是一个用于测试的小工具。它是多线程的，所以能同时运行多个 servlet，或者运行一个还要调用其它 servlet 来处理客户请求的 servlet。和其它服务器不同，它不支持被修改的 servlet 的自动重新加载。但是由于 servletrunner 非常小，只有很少的资源开销，你完全可以随时终止或启动它，以便加载新版本的 servlet。

这个 servletrunner 在 <JSDK>/bin 目录中。 用-help 调用它会显示如下的使用帮助信息：

```
% ./bin/servletrunner -help
Usage: servletrunner [options]
Options:
  -p port     the port number to listen on
  -b backlog  the listen backlog
  -m max      maximum number of connection handlers
  -t timeout  connection timeout in milliseconds
  -d dir      servlet directory
  -r root     document root directory
  -s filename servlet property file name
  -v         verbose output
%
```

为了了解这些选项的缺省值，你可以用-v 开关调用 servletrunner。servletrunner 启动后，显示如下信息：

```
% ./bin/servletrunner -v
Server settings:
  port = 8080
  backlog = 50
  max handlers = 100
  timeout = 5000
  servlet dir = .
  document dir = .
  servlet propfile = .:servlet.properties
```

你可以改变其中的缺省值。

servletrunner 启动后，你就能运行各个 servlet 了，你可以在浏览器中直接敲入，也可以通过 FORM 来调用 servlet。调用 servlet 的一般形式为：

```
http://machine-name:port/servlet/servlet-name
```

这里 servlet-name 对应你给 servlet 取的名字。比如，为了运行名为 phone 的 servlet，定义属性 servlet.phone.code = PhoneServlet 后，你可以用下面的 URL。（这里假设 servletrunner 运行在你自己的主机——localhost 的端口 8080 上，该 phone servlet 位于运行 servletrunner 时指定的 servlet 目录中。）

```
http://localhost:8080/servlet/phone
```

在举个例子。上面提到的 Survey servlet，是在提交 FORM 时运行。相应的 servlet 的 URL 地址是：

```
http://demo:8080/servlet/survey
```

如下是 survey servlet 运行的结果：

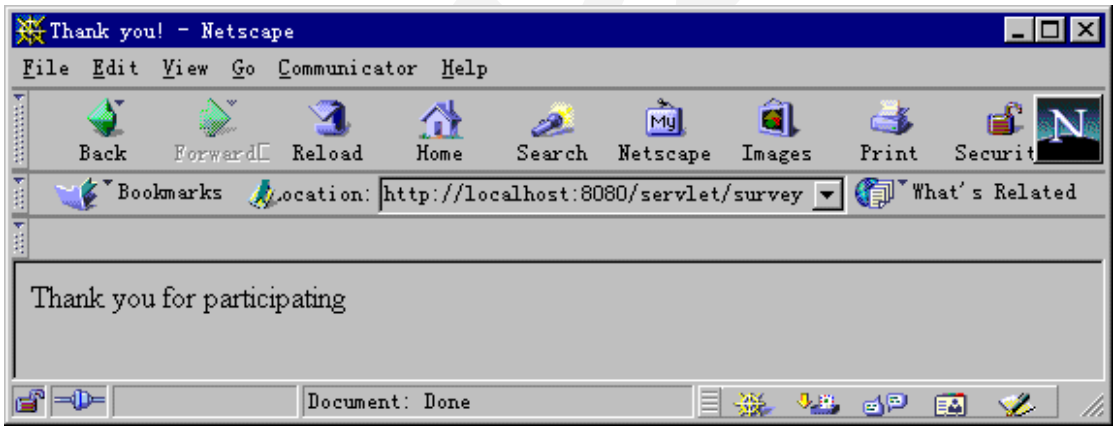


图 8.1

读者在使用时，需要首先启动 servletrunner，然后用浏览器打开 JSDK 包的 examples 子目录中的 JdcSurvey.html 文件。在其中的 FORM 中输入数据后提交请求，将显示出图 8.1 所示的结果。实际的处理和显示便是由 Survey servlet——examples 子目录中的 SurveyServlet.class 完成的。

注意，你可能需要将原始 JdcSurvey.html 文件中的如下行：

```
<form action=http://demo:8080/servlet/survey method=POST>
```

改为：

```
<form action=http://localhost:8080/servlet/survey method=POST>
```

因为“ localhost ”才是本地机器的缺省别名，而“ demo ”则无法被解析。不知是否是 JSDK 包的一个错误？！

8.2.5. 什么是 Apache JServ

概貌

Apache JServ 是完全符合 JavaSoft Java Servlet APIs 2.0 规范的 100%纯 Java servlet 执行平台(engine)。为了和其它 servlet 平台保证完全的兼容,Apache JServ 的开发者们和 JavaSoft 及其它 JVM 开发商密切合作,为 servlets 和 Java 语言本身提供完整而准确的支持。

因此 Apache JServ 作为在任何 Java1.1 兼容 JVM 上的纯 servlet 引擎,可以执行任何符合 JavaSoft Java Servlet APIs 2.0 规范的 Java servlet。

为了不依赖于底层,实现完全的抽象,Apache JServ 1.0 被设计成一个服务器端的应用,可以处理遵循特定协议(即 Apache JServ Protocol,AJP 协议)的请求。其中提供的模块可以支持 WWW 服务器对 servlet 请求进行转换并传递给对应的 servlet 执行平台。

当前,Apache JServ 包只包含一个用于 Apache Web 服务器的模块(mod_jserv)。

Apache JServ 特征

- 完全符合 JavaSoft Servlet API 2.0 specification,支持 100%纯 Java servlets 应用的执行,支持完全的 servlet 执行平台的移植。
- 以与 WWW 服务器分离的多线程方式工作,可提高服务器的稳定性和节省资源。
- 集成化的负载平衡机制充分满足您的需求。
- 请求的智能重定向支持负载平衡请求会话。
- 使用 AJP 协议可支持复杂的网络环境(比如:很多 WWW 服务器要与位于不同机器上的许多 servlet 执行平台通信)。
- Servlets 可以被分成几个逻辑区(servlet 区),从而方便 servlet 的管理并支持不同 servlets 的相互分离。
- Servlets 可以放在目录中也可放在打包文件(jar or zip)中。
- Servlets 类在更改后可以自动装入。
- 基于 MD5 的连接身份确认和 IP 过滤保证完全可信的 servlet 的执行。
- mod_jserv 模块完全支持 Apache Web 服务器 1.2.x 和 1.3.x。
- 配置文件和显示状态及配置信息的动态页面的管理方便。
- 完全支持页面编译(page compilation)、JSP(Java server pages)、SSI(server side include)和模板系统(需要有其它模块的支持)
- 扩展的日志和跟踪功能支持细节控制和调试。
- 当然,还有就是开放源码带来的一切优点……

8.2.6. 如何在 win32 平台下安装 Apache Jserv

在 win32 平台下安装 Apache Jserv,既可以使用自动方式,直接取得并执行安装程序按照每一步的提示进行操作,也可以自己进行定制安装。

自动安装

首先执行安装程序——Apache_JServ_1_0.exe**，会出现下列画面：**

图 8.2

该画面说明制作 Apache_JServ **安装包的工具。用鼠标点击一下该画面和击一下空格键，安装即会继续。出现：**

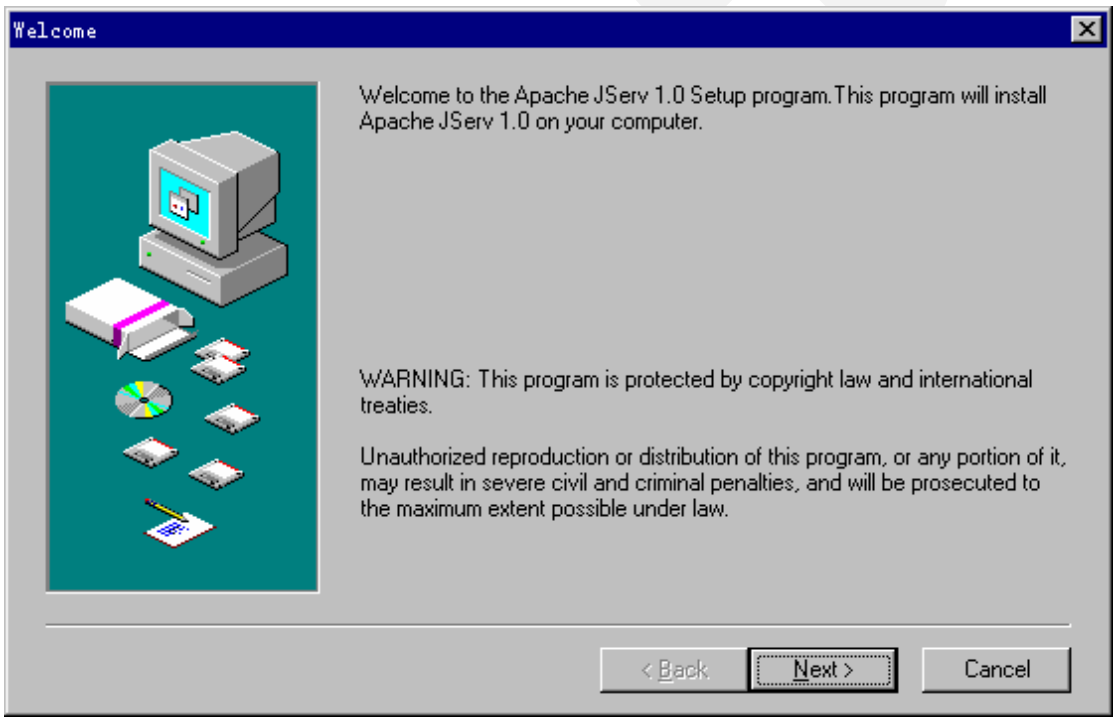


图 8.3

点击“ Next **”按钮，出现：**

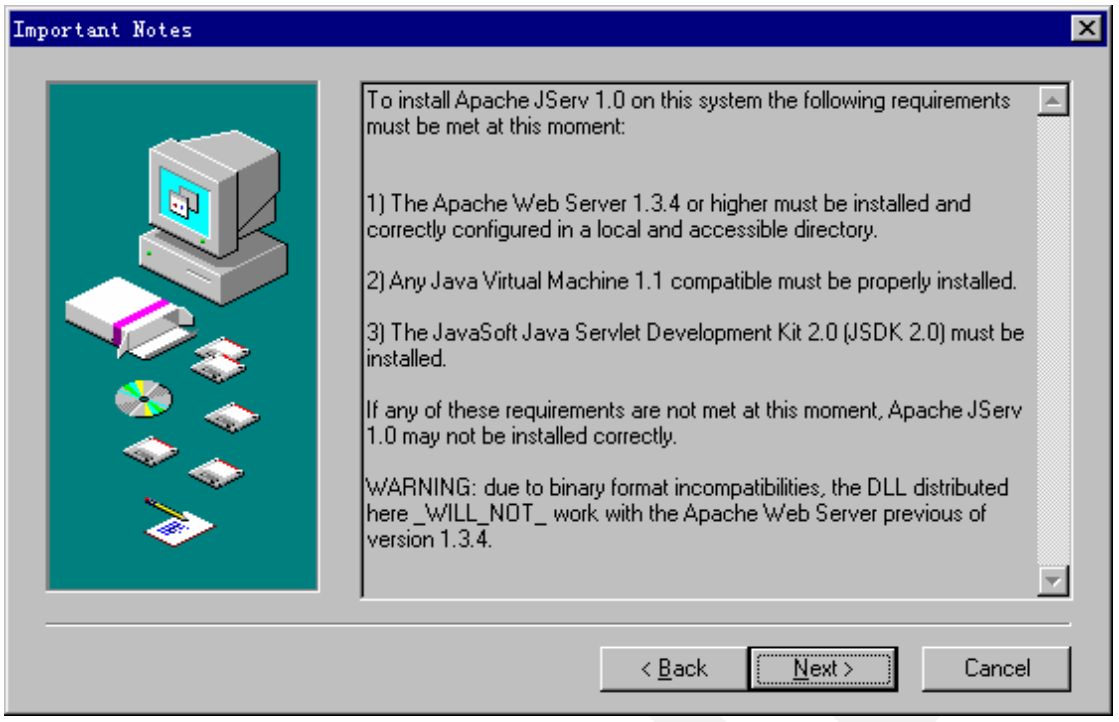


图 8.4

点击“ Next **”按钮，出现：**

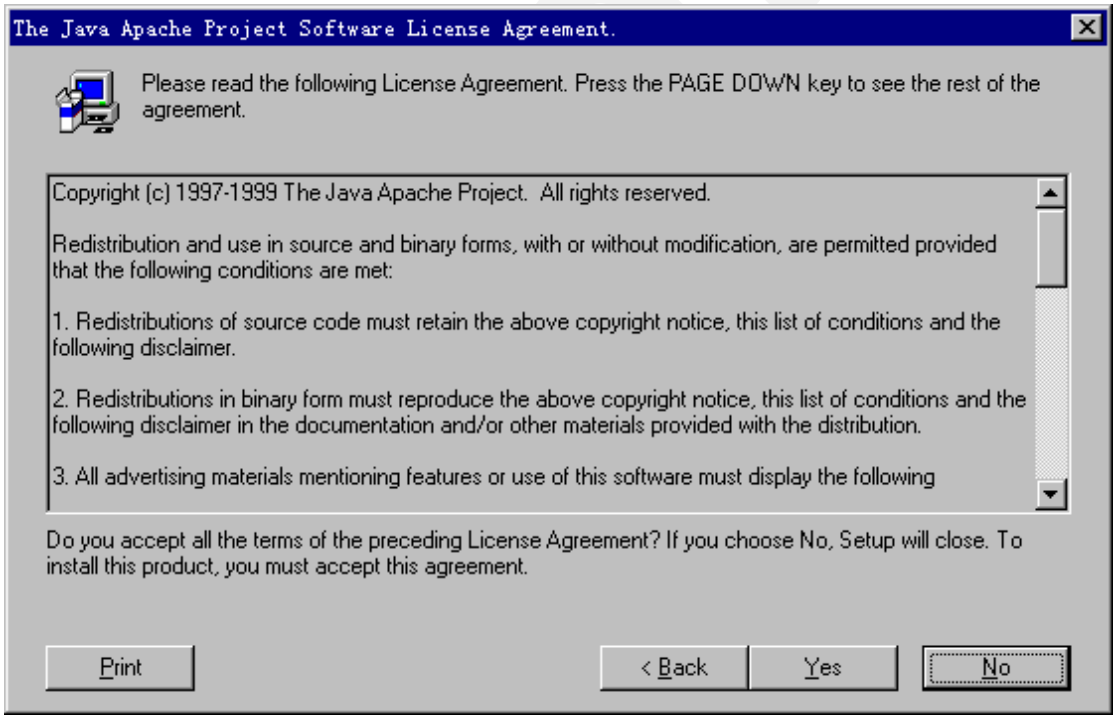


图 8.5

点击“ Yes **”同意给出的软件使用协定，则出现：**

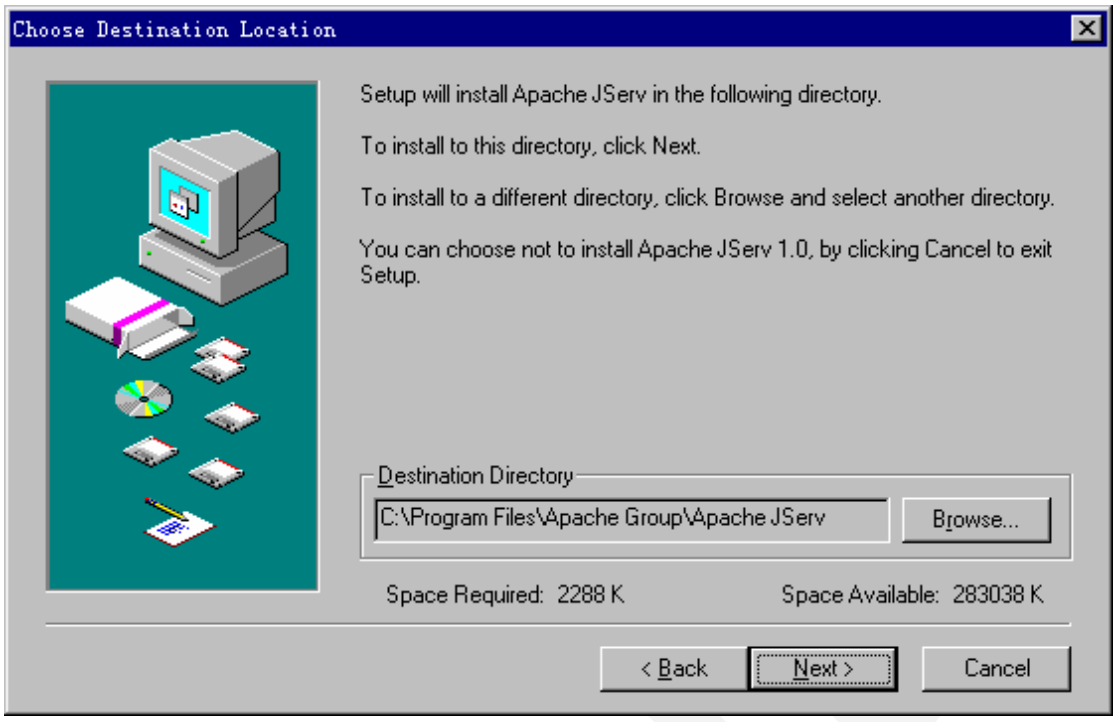


图 8.6

其中用户可以选择 Apache_Jserv **的安装目录，缺省情况下安装到“** C:\Program Files\Apache Group\Apache JServ **”中。若用户要安装到其它目录，可以点击“** Browse **”，在出现的如下窗口中选择新位置：**

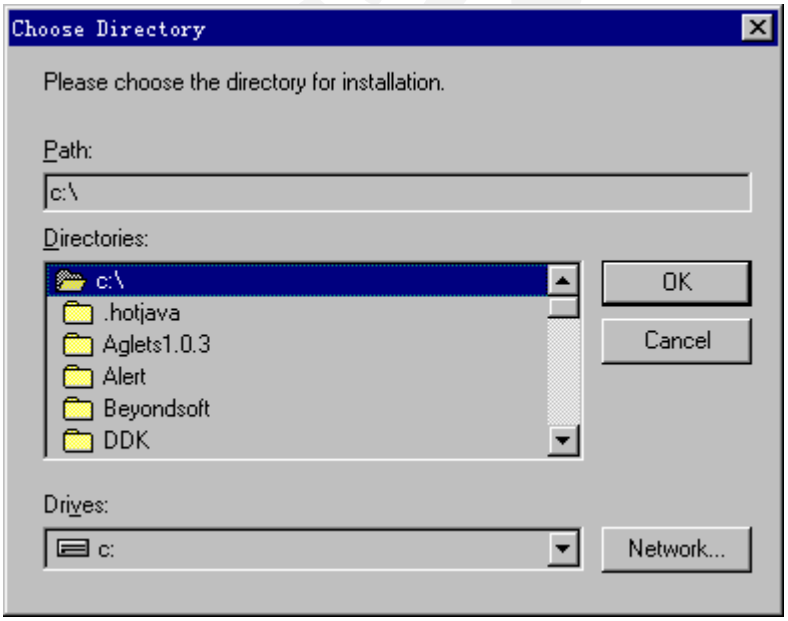


图 8.7

在选好目标目录后点击“ OK **”，将返回先前的画面。继续点击“** Next **”出现如下画面：**

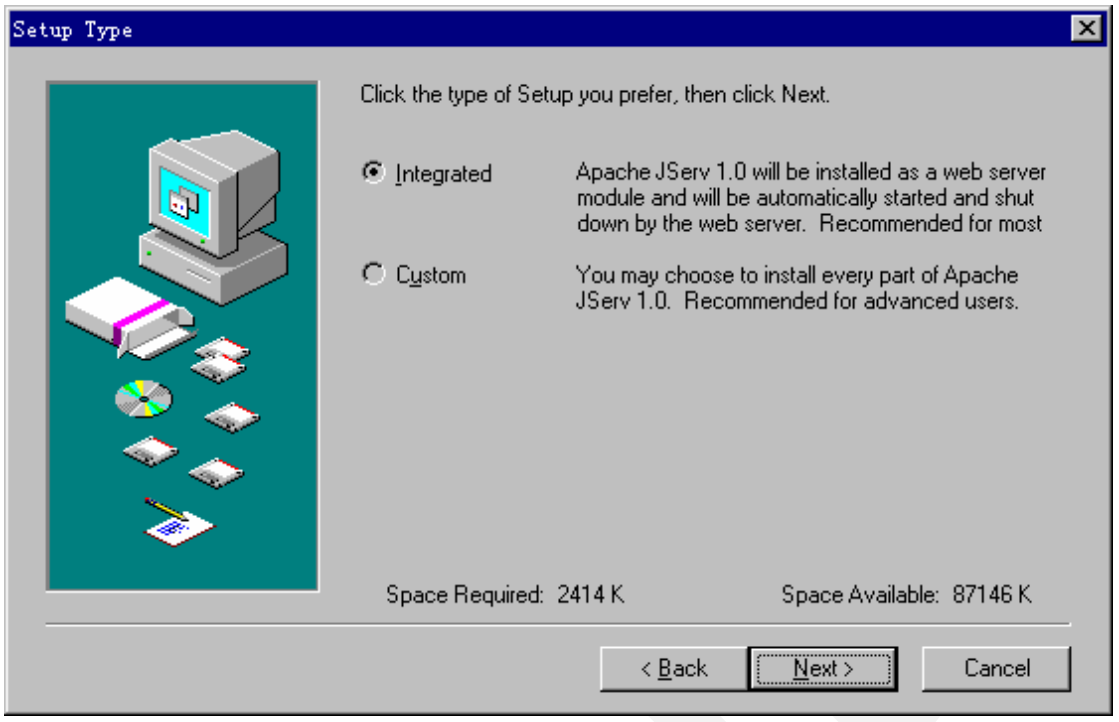


图 8.8

其中给出安装 Apache_Jserv 的两种方式：集成（Integrated）和用户定制（Custom）。前者表示 Apache_Jserv 将作为 Apache 服务器的一个模块，随着 Apache 服务器的启动而工作，此是缺省选择；后者表示用户可以安排 Apache_Jserv 的安装形态。

若选择后者，则用户可以对安装的各成分进行选择，如图 8.9 所示。Apache_Jserv 包括四部分：

- Servlet 引擎
- 与 Apache 服务器连接的扩展模块；
- 文档
- 源代码

选择完毕，点击“ Next ”即可开始安装。

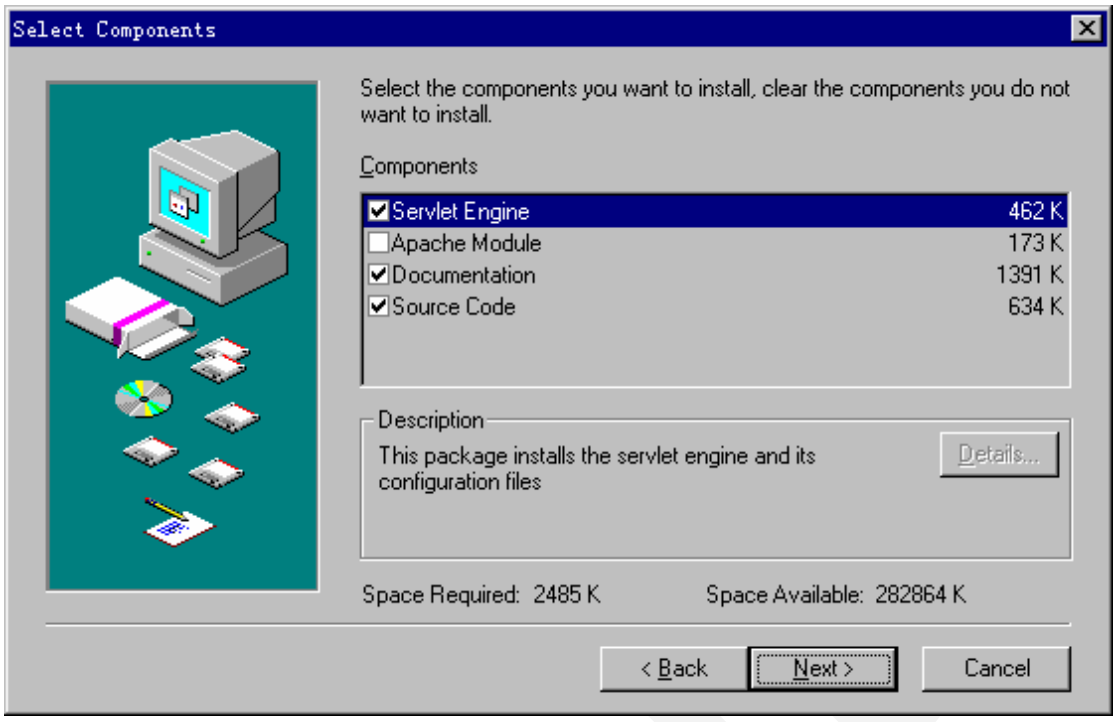


图 8.9

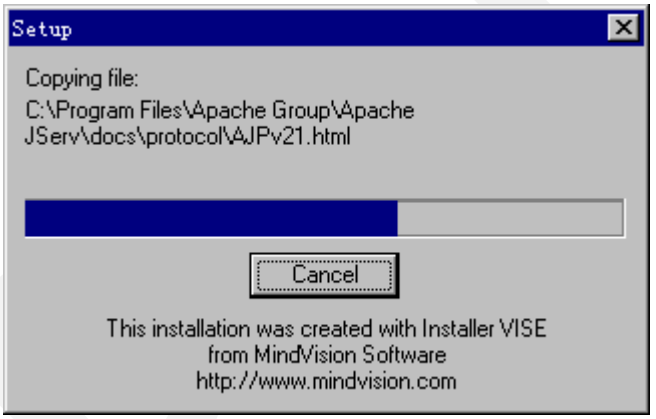


图 8.10

一般应选择前者，以便能自动完成 Apache **服务器和** Apache_Jserv **的结合。系统进行安装的过程，如上所示。在程序拷贝完成后，显示如下画面：**

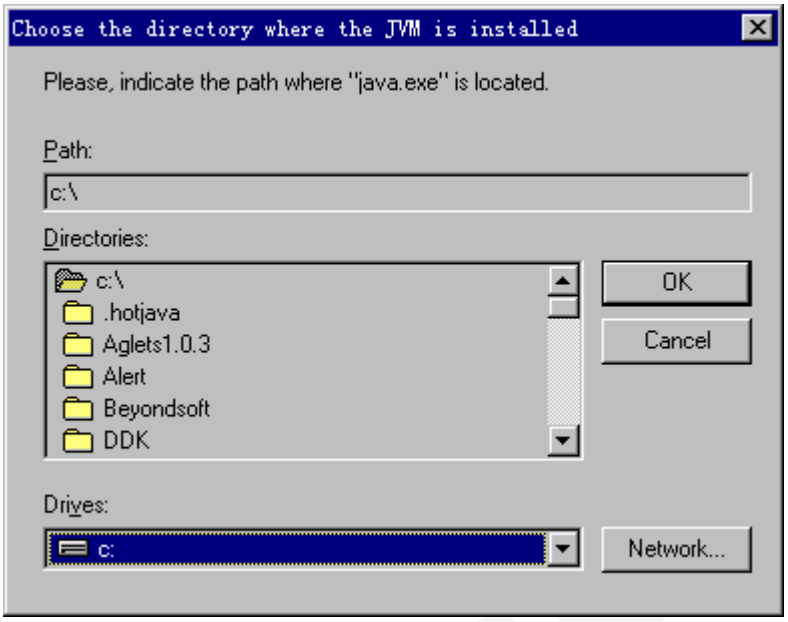


图 8.11

要求提供 Apache_Jserv **运行所需要的** Java **虚拟机（**JVM**）的位置信息。指定系统中已安装的** JVM **的位置，如“**c:\jdk1.1.6\bin**”后，选“**OK**”，则继续提示输入** JSDK **的类库位置信息：**

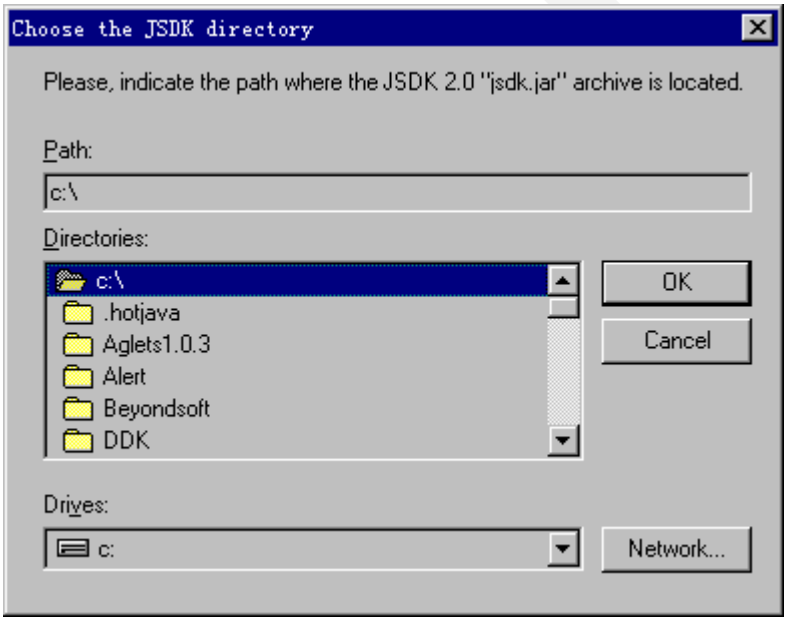


图 8.12

指定系统中已安装的 JSDK **工具包中包含的类库位置，如“**c:\jsdk2.0\lib**”后选“**OK**”，出现画面：**

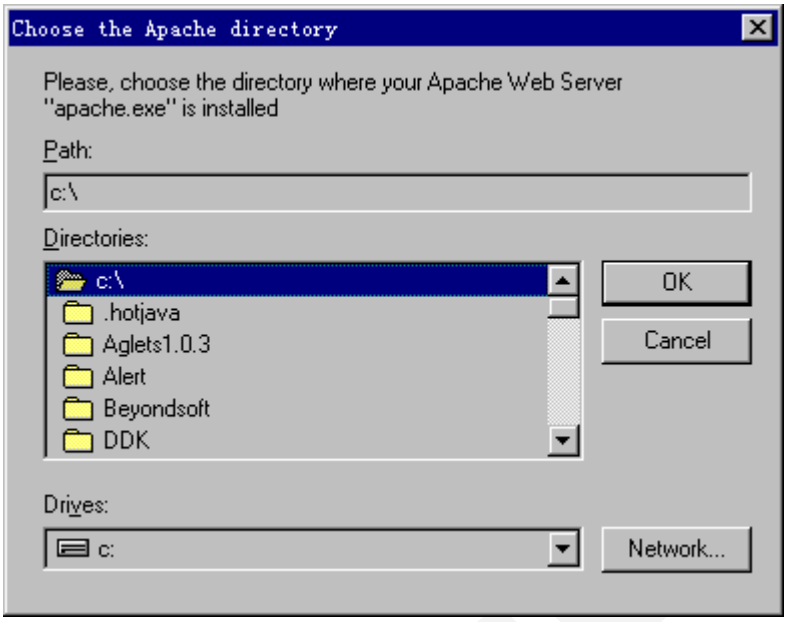


图 8.13

该画面提示输入 Apache **服务器可执行文件的位置，如“**c:\program files\apache group\apache**”后选“**OK**”，出现：**

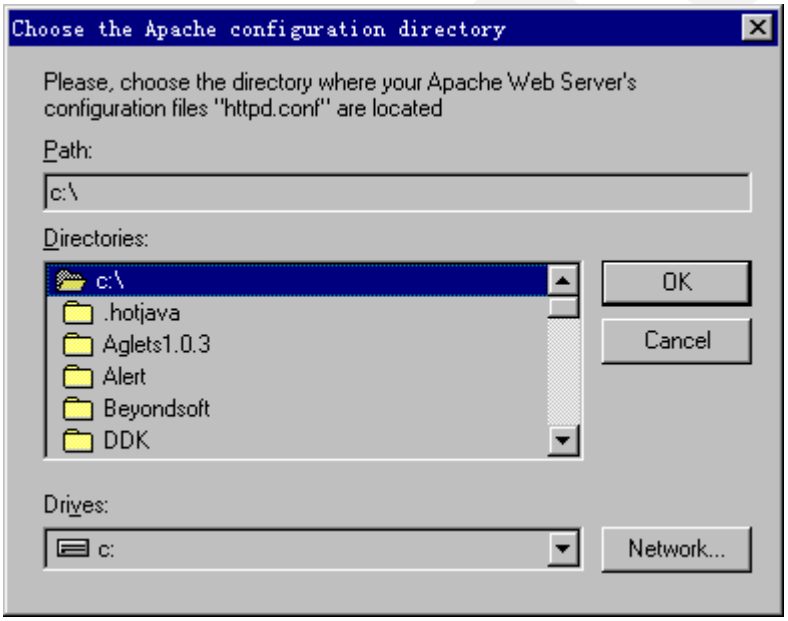


图 8.14

该画面提示输入 Apache **服务器主配置文件的位置，如“**c:\program files\apache group\apache\conf **”，然后选“**OK**”，安装程序给出警告信息：将更改该配置文件，且会备份原来文件。**

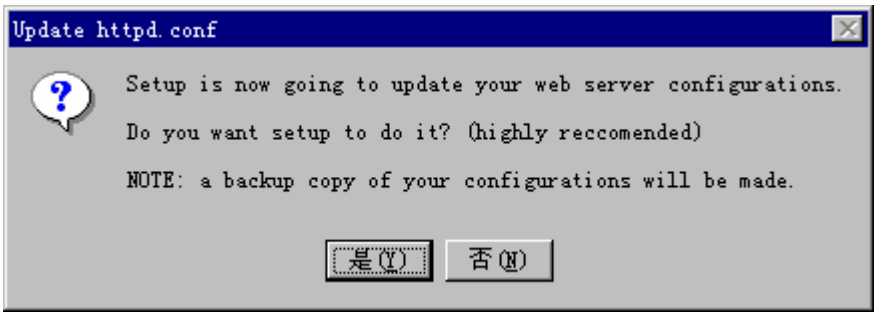


图 8.15

选“是”继续，安装程序实际完成该更改工作，最后显示创建一个程序组窗口：

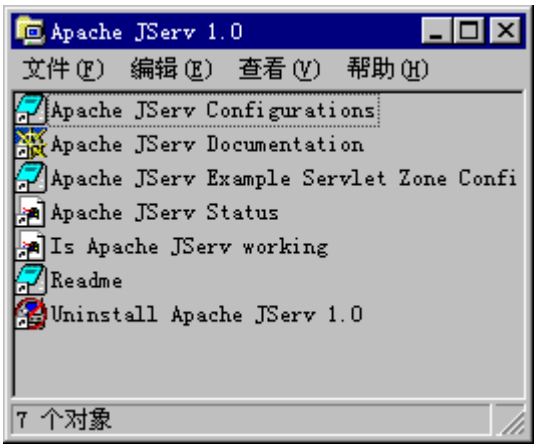


图 8.16

至此 Apache_Jserv 已完全安装完毕。

正常情况下，以控制台方式运行 Apache 服务器应出现如下的画面，其中的提示说明 Apache_Jserv 已与 Apache 服务器一起正常工作。

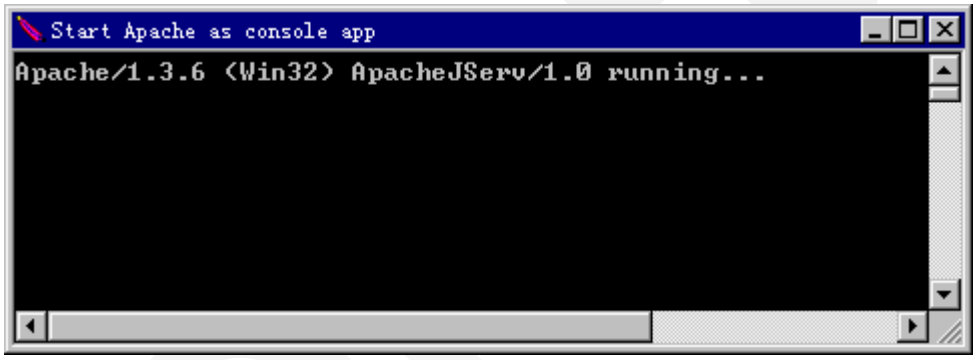


图 8.17

非自动安装

介绍

当您没有使用自动安装方式，或您在自动安装时遇到了麻烦，或者在定制安装时 Apache JServ 没有和 Apache Web 服务器自动连接起来时，您需要阅读本节来学习如何在 win32 平台下手工地一步步安装 Apache JServ。

需求

要使用 Apache JServ，需要满足如下条件：

- 磁盘空间：
  完整的 Apache JServ 安装需要不到 2Mb 的硬盘空间。编译过的非压缩 jar 文件包则

大约 100Kb。

- Apache HTTP 服务器：
  Apache JServ 1.0 必须在 Apache HTTP 服务器 1.3.1 或更高版本上运行。
- Java 运行环境（Java Runtime Environment）：
  需要有 Java 1.1 兼容的运行环境。正式的版本可以在 JavaSoft 站点上找到，还有支持的平台和其它的类似产品等信息。
- Java Servlet 开发包（Java Servlet Development Kit 2.0，JSDK）：
  需要有 JavaSoft 的 JSDK 2.0 来支持 servlet 的执行。该包可以从 JavaSoft 站点（http://java.sun.com/products/servlet/index.html）免费拿到。
  注意：仅有 JSDK 2.0 满足要求，其它版本现在均不支持。
- Java 编译器（Java Compiler）：
  既然 Apache JServ 中已经包含了一个预编译好的 Java 类库，所以对 Java 代码的编译是选择性，不是必需的。
  注意：如果你要自己写 servlet 的话，就必须有 Java 编译器。
- C 编译器（C Compiler）：
  即使 win32 平台下已经含有了预编译好的包，mod_jserv 模块和本地化包装也仍需要 C 编译器，以便能正常地工作。当前，只有 Microsoft Visual C++（version 5.0 或更高版本）符合要求。

概貌

Apache JServ servlet 执行平台包括两部分：servlet 引擎（用 100% 纯 Java 代码编写）和 WWW 服务器模块（mod_jserv，用 C 代码编写）。该模块是 WWW 服务器和 servlet 执行平台间的通信层。

尽管 Apache JServ 有多种安装方式，本节只讨论基本安装方式（也称为集成或自动安装），这种方式下，servlet 执行平台进程由 WWW 服务器创建并受其控制，因此它们共享同样的系统资源。

这里我们假设你已经安装和配置了 Apache Web 服务器。

安装 servlet 引擎

servlet 引擎是一个 100%纯 Java 服务器端应用程序，它其自己的配置文件。jserv.properties 文件是它的主配置文件，zone.properties 则是 servlet 区配置文件的示例。你需要更改 jserv.properties 文件中的某些配置以使启动 JVM 的模块知道 JVM 在哪里。

如下是你所需要编辑的属性：

```
wrapper.bin=@JAVA@ （JVM 软件名，如果不在 PATH 中需要指定绝对路径。）
wrapper.classpath=@JSERV_CLASSES@（ApacheJServ.jar 的路径）
wrapper.classpath=@JSDK_CLASSES@ （jsdk.jar 的路径）
root.properties=@ZONE_CONF@ （./conf/zone.properties 的路径）
log.file=@JSERV_LOG@ （./logs/jserv.log 的路径）
```

安装 WWW 服务器扩展模块

Apache web 服务器通过文件名为 ApacheModuleJServ.dll（在 Apache JServ 软件包的\bin\c 目录下）的 Apache JServ 模块与 servlet 引擎通信。该文件要放到已安装的 Apache 软件包的\modules 目录下。

在拷贝完该文件后，你需要将配置文件模板\conf\httpd.conf 附加到 Apache 配置文件（通常是 httpd.conf）后。基于该模板，你可以对 Apache JServ 模块和 WWW 服务器进行配置。

现在，首先将对应的 loadmodule 配置命令前的“#”去掉，以使 Apache 知道要装入该模块：

```
# Tell Apache on win32 to load the Apache JServ communication module
LoadModule jserv_module modules/ApacheModuleJServ.dll
```

然后，你需要通过如下的配置命令指定 servlet 引擎配置文件（通常是 jserv.properties）和模块日志文件：

```
ApJServProperties <./conf/jserv.properties 的完整路径>
ApJServLogFile <./logs/jserv.module.log 的完整路径>
```

若该日志文件不存在，则将被创建；否则新内容将附加到原有内容后。

检查安装情况

现在你可以测试以下你的安装情况：启动 Apache Web 服务器，提出请求“http://127.0.0.1/jserv/”。

如果 WWW 服务器返回了一个“File not found”的错误代码，则说明你没有正确安装该模块，或者你禁止了 jserv-status 句柄的使用。否则你应该可以看到动态生成的 Apache JServ 配置信息页面，告诉你当前 servlet 环境的情况。

你看到的页面是是由 ApacheModuleJServ.dll 模块生成的。要知道 servlet 引擎是否正常工作，点击“servlet engine section”链接。如果成功，则你会看到 Apache JServ 动态生成的显示配置信息的页面；否则你会被告之服务器内部错误，这说明 servlet 引擎没有正常工作。

问题的解决方法

如果在连接 servlet 引擎时遇到服务器内部错误，你需要去查看日志文件以发现问题的所在。

若 jserv.log 文件没有创建，则意味着 Apache JServ 根本没有启动。通常这是由于传递给虚拟机的 classpath 有问题。查看 jserv.module.log 文件和 Apache 的 error.log 文件来确定问题的所在。你可以用绝对路径来替代相对路径来试一试，因为不同的进程的工作目录会有所不同。

当 classpath 设置正确，Apache JServ 也正常启动后，jserv.log 便会创建。查看该文件的描述可解决其它问题。

Servlets 的安装

在 servlet 引擎启动并运行后，你就该安装 servlets 了。但你首先要了解 servlet 区及其用法。进一步地你还需要了解 servlets 和 servlet 引擎的安全性等问题。这些你可在下文找到。

8.2.7. Servlet 区

要充分发挥 Apache JServ 的强大功能，你必须对 servlet 区及其设计思想有一个很好的理解。

定义

如下给出几个定义：

Servlet 是实现了 javax.servlet.Servlet 的 Java 类，即是一个文件。

Servlet 库（Servlet repository）是 servlets 的集合。它可能是一个目录，或是一个类文件包（zip 或 jar 文件）。

Servlet 区（Servlet zone）是 Servlet 库的集合。

既然在同一个 Servlet 区中的 Servlet 库共享相同的逻辑上下文环境，因此可以把 Servlet 区看作 servlet 引擎的中虚拟主机。这使得你可以将 servlets 逻辑地分开（如基于所属的虚拟服务器等），将来在安全性方面，可以将每个 Servlet 区映射成一个安全沙箱（security sandbox）。

创建 servlet 区

每个 Servlet 区有独立的配置文件，使创建和配置非常的容易。

第一步，在/conf/zone.properties 文件中增加包含 servlet 的 servlet 库。你需要做的只是：增加一条列有全部 servlet 库的属性定义：

```
# The list of servlet repositories controlled by this servlet zone
# Syntax: repositories=<repository>,<repository>...
repositories=/usr/local/apache/servlets,/usr/local/java/servlets.jar
```

另外，你最好能把该文件改名为[zone name].properties ，并把它放到 Apache JServ 可以访问的目录中。注意：servlet 初始化参数和特定的 servlet 区配置（类自动重新加载和会话超时等）都存在这个文件中，因此，你需要让该 servlet 区的所有者可以访问和配置该区。

声明一个 servlet 区

一旦创建了 servlet 区，你还必须让 Apache JServ 知道它才能使之工作。具体的做法是：将如下几行加入到 jserv.properties 文件中：

```
# List of servlet zones JServ manages
# Syntax: zones=<servlet zone>,<servlet zone>...
```

```
zones=bob,alice

# Configuration file for each servlet zone
# Syntax: <zone name>.properties=<full path to configFile>
bob.properties=/home/bob/servlets/servlets.properties
alice.properties=/home/alice/servlets/servlets.properties
```

在启动时，Apache JServ 为使用对应的配置文件对各 servlet 区进行设置。因此，必须让 Apache JServ 有访问 servlet 区配置文件及其中库的权限。

装配 servlet 区

在 Apache JServ 和各个 servlet 区配置完后，你还需要将各个 servlet 区装配（mount）到 WWW 服务器文件空间中。你只需要在服务器配置文件中增加寥寥一行即可。假设你已经使 Apache JServ 运行在 jserv.yourDomain.com 上，并用 9009 端口来处理如上定义的 servlet 区。

就象你在 UNIX 系统中装配磁盘一样，你要将各个 servlet 区逻辑地分开，就需要将这些 servlet 区装配到特定的 WWW 服务器 URI 上。你需要使用 ApJServMount 配置命令：

```
ApJServMount /bob/servlets ajpv11://jserv.yourDomain.com:9009/bob
ApJServMount /alice/servlets ajpv11://jserv.yourDomain.com:9009/alice
```

现在，Bob 和他的客户就可以用 http://www.yourDomain.com/bob/servlets 连接到他的 servlets 上，对 Alice 也一样。

和其它 servlet 引擎不同，Apache JServ 的突出优势是可以装配不仅是本地的 servlet 区，还可以是远程的 servlet 区。实际上，Apache JServ 扩展模块起到了一个“AJP 浏览器”的作用，而不仅仅是其它 servlet 引擎“粘合”本地进程和 Java 进程的的作用。

这使你可以构造一个完整的三层网络应用环境，使 WWW 服务器可以“集中精力”管理 URI 空间。你甚至可以使用网络无关的 JNI servlets（如传真机、本地数据库、甚至烤箱和微波炉）

未来计划

在 Apache JServ 的将来版本中，每个 servlet 区将有独自的安全管理器和沙箱，就象浏览器处理 Java applets 一样。这样会使同一 JVM 执行的 servlets 在逻辑上完全分开，取消了现在必须用不同的 UID/GID 来执行多个虚拟机的需要。

8.2.8. Apache JServ 状态查询

Apache JServ 的一个非常有趣的地方是：它的核心类 JServ.java 本身就是一个 servlet！这虽然看起来很奇怪，但却使 ApacheJServ 可以作为 servlet 运行，接收客户的请求，输出当时的动态信息，如配置情况和将来增加的远程管理和统计数据信息。

Apache JServ 通过动态地创建一些页面，使管理员可以对系统的当前情况有所了解，不仅是对于 mod_jserv 模块，还有连接的各个 servlet 引擎。

访问 Apache JServ 的状态页面

Apache JServ 状态页面在缺省情况下是不被禁止的，它被映射到 WWW 服务器地址空间的/jserv/地址上，如 http://127.0.0.1:80/jserv/，请注意一定要有最后的一个“/”。

这些页面使你可以远程查看配置信息，并对 servlet 引擎施加控制。

警告：为了不让无关用户搜集系统的安全信息，最好禁止他人对状态查询的访问。

对状态显示页面控制

如下是一段将 Apache JServ 状态页面映射到服务器地址空间中的配置定义。它可以放到服务器三个配置文件的任何一个中：

```
<Location /status/jserv/>
SetHandler jserv-status
order deny,allow
deny from all
allow from dummy.org
</Location>
```

上面的配置将 Apache JServ 状态页面（SetHandler jserv-status）映射到/status/jserv/地址上（<location/status/jserv/>），并只允许来自"dummy.org" 域的用户访问。

关于访问控制的更复杂的例子和文档请参考服务器相关在线说明。

删除 Apache JServ 状态页面

要删除该页面只需将上面的配置命令片段从配置文件中删除即可。

8.2.9. 如何安装 servlets

在 Apache JServ servlet 执行平台安装完毕后，你还需要将你的 servlets 放进去。本节将介绍如何安装、配置和管理 servlets。

对 servlet 环境分区

Apache JServ 支持将其执行空间分成几个独立的区域，称为 servlet 区，就好象是独立的虚拟 servlet 执行平台一样。与 WWW 服务器中的虚拟主机机制类似，你最好基于所有者、安全访问权限等对 servlet 空间进行划分。但是，大多数配置并不需要这些高级的特征，所有 servlets 都放在一个区中即可。

注意：Apache JServ 中，servlets 是由 servlet 区来处理的，而不是直接由 servlet 引擎完成，因此，至少要有一个 servlet 区。

现在，假设你需要两个 servlet 区，一个用于生产（production），一个用于开发（development）。我们简单地将它们分别称为 production 区和 development 区。我们还假设

servlet 执行平台已经启动并运转起来了。下面我们仔细来看 servlet 区的建立过程。

创建 servlet 区

Apache JServ 有一个通常命名为 jserv.properties 的主配置文件，用来存放一般性信息。每个 servlet 区也有一个配置文件，由 servlet 引擎统一管理。因此，我们生成区配置文件示例(zone.properties)的两个拷贝,并分别命名为 production.properties 和 development.properties。

然后，对 jserv.properties 文件进行编辑，增加这样两条配置：

```
# List of servlet zones Apache JServ manages
zones=production,development

# Configuration file for each servlet zone (one per servlet zone) production.
properties=/servlets/production/production.properties
development.properties=/servlets/development/development.properties
```

当 Apache JServ 启动时，它会读取 servlet 区，并查看对应的区配置文件。

配置 servlet 区

除了要告诉 servlet 引擎有哪些 servlet 区外，你还应告诉它到哪儿去找其中的 servlets。你需要在区配置文件中加入下面的配置：

```
# The list of servlet repositories controlled by this servlet zone
repositories=/servlets/production/project1/
repositories=/servlets/production/project2.zip
repositories=/servlets/production/shared_servlets.jar
```

在本例中,production 区有三个 servlet 库：一个目录(project1)、一个 zip 文件(project2.zip)和一个 java 压缩文件（shared_servlets.jar)。这些配置告诉 servlet 区的类加载器到哪里去找被请求的 servlet。

servlets 的定位

既然 servlets 是 Java 类而且每个 servlet 区有其自己的类加载器，所以 servlets 的访问、定位和命名都使用包名加类名的 Java 类全称。注意，在定位时并没有象 WWW 服务器那样考虑 servlet 在文件系统中的实际位置。实际上，在不同 servlet 库但有相同包名的 servlet 被类加载器认为在同一个目录中。

例如，org.dummy.project1 包中的 servlet /servlets/production/project1/Hello.class 的 Java 全称为 org.apache.project1.Hello，与它的实际存放位置无关。

这种基于包名的绝对名称序列可以避免 servlet 区内的类名冲突。而如果两个 servlet 区共享同一个 servlet 库，由于它们使用不同的类加载器，故也不会有冲突。注意：为了避免与名为 class 的包冲突，在构造全称时总是将扩展名.class 去掉。

Servlet 别名

尽管类的命名在排序和避免冲突方面非常成功，但这却给 servlets 带来非常繁琐的名字。因此 servlet 区中引入了别名机制来简化名称，可以使用别名来取代原来的全称来指代原来的 servlet。但是，这样的两个 servlets 被认为是两种东西，两种称谓引入了同一 servlet 类的不同实例。

例如，将类 org.dummy.project1.Hello 简称为 hello， 需要在区配置文件中增加如下命令：

```
servlet.hello.code=org.dummy.project1.Hello
```

启动 servlets

正常情况下，servlet 只在第一次请求时才实例化和初始化。这虽然使内存消耗较低，但也使第一次请求的处理比后续请求慢。为了避免这一问题，你可以让你的 servlet 在 servlet 引擎启动时即运行，你需要把该 servlet 放在该 servlet 区的 servlet 启动列表中。

具体做法是，在 servlet 区配置文件中增加如下命令：

```
servlets.startup=org.dummy.project1.Hello
```

这里可以使用 servlet 的全称，也可以使用别名。

Servlet 初始化参数（initArgs）

和其它可执行代码类似（application，applets），servlets 也需要某些初始化参数。每个 servlet 可以有无限数目的初始化参数，可以放在 servlet 区配置文件中，也可单独存放于某个文件。例如：servlet “hello”需要两个参数：message 和 color，可以通过在 servlet 区配置文件中增加如下内容实现：

```
servlet.hello.initArgs=message="Hello world to everyone"
servlet.hello.initArgs=color=red
```

全局初始化参数(default initArgs)

有时，在某个 servlet 区的所有 servlet 必须共享同一初始化参数。方法是：

```
servlets.default.initArgs=path_to_docs=/usr/local/docs/html
```

8.2.10. 保证 servlet 环境的安全

本节将介绍在各种情况下如何保证 servlet 执行平台的安全。

外部安全性

Apache JServ 1.0 在通信协议上进行了很重要的增强，减少了不可信 servlet 执行的危险。

让我们举例说明之。假设你有一个 servlet，SQLquery，用来连接数据库。该 servlet 是为远程管理和数据库调试而设计的，它的安全由 WWW 服务器来保证。

但如果在 servlet 引擎上没有安全限制，一个聪明的攻击者会直接向 servlet 引擎请求这个 servlet，从而绕过了 WWW 服务器的安全控制。这称为外部攻击。

为了避免外部攻击，Apache JServ 1.0 引入了功能强大的安全限制新机制：外部连接的 IP 过滤和基于 MD5 的连接身份认证。

IP 过滤

减少外部攻击最简单有效的办法是基于客户的 IP 地址对请求进行过滤。需要注意的是，这对 WWW 服务器的用户没有任何的影响，因为只有 WWW 服务器才可以连接 servlet 引擎。因此，为了允许从特定 IP 地址来的连接，你必须在 Apache JServ 主配置文件的允许列表中指定它。

例如，你有两个 WWW 服务器，分别是 194.39.283.233 和 194.39.283.234，为了限制 servlet 请求只允许从这两个地址发出，需要在 jserv.properties 文件中增加如下行：

```
security.allowedAddresses=194.39.283.233,194.39.283.234
```

在该过滤器的作用下，任何不是来自表中所列地址的连接请求均会被忽略。

连接认证

有时，仅有 IP 过滤并不充分，因为某些不可信用户可以发出伪装成来自被允许 IP 地址的请求。这时，可以使用连接认证来最大限度减少外部攻击的可能性。这种方法要求在 WWW 服务器和 servlet 引擎均存有同一文件的拷贝，该文件称为安全密钥。

密钥文件可以是任意格式和任意长度（甚至可以是一幅图象），但我们建议你胡乱地敲大约一百个字符作为密钥。我们认为没有必要使用过长的密钥，因为密钥长度和安全认证时间成正比，多输入几个字符不会使安全性有多大提高。

注意：你系统的安全性是与你的密钥密切相关的。任何可以猜出密钥并能伪装来自被允许地址范围的人都是潜在的攻击者。

要实现 servlet 引擎上的安全限制，你可以在 jserv.properties 文件中增加如下两行：

```
security.authentication=false
security.secretKey=/etc/jserv/jserv.secret.key
```

然后，你还要在每个合法的 WWW 服务器的配置文件 httpd.conf 中增加如下内容：

```
ApJServSecretKey /etc/jserv/jserv.secret.key
```

同时，你要保证如上的两个密钥是完全相同的，WWW 服务器和 servlet 引擎均有权限读取它。

内部安全性

外部安全性是针对不可信 servlet 的执行的，而内部安全性则针对 servlet 的自身行为。如果你是自己编写的 servlets 或者完全清楚它的源码，则不会存在任何内部安全性。但如果你使用的是他人提供的预编译好的 servlets，你就需要注意了。

和使用不同的 UID/GID 来执行 CGI 类似，一个可能的办法是用不同的 UID/GID 来执行

几个 servlet 引擎。这样就将不同 servlet 引擎执行的 servlet 分离开来。

另一种解决办法是使用将来功能更强的 Java 安全管理器，保护系统资源不受异常 servlet 行为的影响。这会在将来版本中实现。

第3节 基于 Apache JServ 的可扩展性、负载平衡与容错

8.3.1. 概述

可扩展性、负载平衡、容错是 Internet 内容提供商特别关心的问题：随着市场的成熟，静态 web 页面的需求下降，动态内容生成、个性化和事务处理成为必需。

我们认为 Java 不仅在客户端有效，在服务器端也是核心技术。它在建立原型和验证阶段的方便快捷，将为垂青者在占领市场的时间上得到丰厚回报。

然而，由于 Java 虚拟机变得越来越快，人们已倾向于要求更加个性化的动态页面，而这种应用则要求 CPU 有更强的能力。Apache JServ 正是面向这一需求，使你可以采取“分而治之”的策略，可以根据所需用大量的机器分担你的业务。

同时，基于 HTTP 会话的应用和用户访问事务不断涌现，Apache JServ 在这方面则可以确保系统运行无忧。当然整个系统必须是容错的。

而且当然，所有的这一切都是免费的 ;-)

本部分将描述如何利用 Apache JServ 的负载平衡和容错特征。

8.3.2. 特征

可扩展性

随着服务器端应用的增长，使用传统的两层结构（www 服务器和 ServerAPI）已非常困难，而 CGI 只会使情况变得更糟。Apache JServ 是一个 100%纯 Java 多线程 servlet 引擎，使用 N 层模型，可以进行方便地扩展。它小可至一个微型站点（如一台 Pentium 120 MHz 机器，48MB RAM，运行 Linux，配备了 Apache 和 JServ），大可基于大量强劲机器。

而界于两个极端之间，Apache JServ 体系结构让你也可以选择从一台 64-CPU 机器到 64 个廉价 PC 之间的任何配置，后者可以方便实现负载平衡：它可以将负载由众多 servlet 引擎分担。

在 Apache JServ 体系中，核心在 AJP，它是一个基于 socket 的协议，使多台机器可以分成两类：一边是运行 Apache 和 mod_jserv，另一边则是运行 servlet 引擎。

例如，Apache JServ 系统可以，

小到：

1 台 PC 机，运行一个 Apache 服务器和一个 JServ 引擎（可以有 1 到 z 个区）。

大至：

10 个运行 Apache 的主机（使用 round-robin DNS 或硬件来实现负载平衡）

33 个 JServs 引擎运行在：

15 个运行 Linux 和 JDK1.2 的 PC 机、
或 5 个 Sun E450 工作站、
或 10 个运行 NT 和 JDK1.1.6 的 PC 机、
或 3 台 AS400、
... （你有太多的选择）

Apache HTTP/HTTPS 服务器的伸缩

每个标准的 Apache 服务器都可以启动 1..256 个 httpd 进程（通过重新编译可以改变这个限制）。这意味着一个 Apache 服务器可以同时处理 256 个客户请求。

因此，对于要处理成千上万并行请求的 web 站点，就需要运行多个这样的服务器，并且使用负载平衡机制来将负载分担在不同的 Apache 服务器上。

各个 Apache 服务器可以运行在同一机器上，但分别使用不同的 IP 地址和端口号，也可以分布在不同的机器上。

负载平衡机制可以基于一个 round robin DNS 服务器，也可以是专用硬件。实际的选择与具体的部署有关，但与 Apache 和 JServ 却没有直接关系。我们并不依赖于这种负载平衡机制来处理应用层的会话。

某些 Apache 服务器是支持 SSL 的，这样 SSL 会话也必须被负载平衡所支持，这时通常使用专用的硬件来实现。

正如我们可以把 Apache 服务器看作 JServ servlet 引擎的网关一样，我们可以想象所有的 apache 服务器将被客户认为是一个主机簇，它们有共同的唯一的主机名，并使用 cookies 机制在接收端标记该主机簇的名字。这使得系统可以跟踪基于 SSL 或非 SSL 的 URL 访问的 HTTP 会话。

我们在下文将会看到这些 Apache 服务器将共享各个 JServ 识别信息（如路由和 TCP 信息等）。

JServ 服务器的伸缩

Jserv 是一个 100%纯 Java 程序，其中，每个 servlet 以独立的线程运行。对付大量的并发请求，很自然地想到用各个 JServ 来处理。有如下原因支持：

现在没有任何一个 JVM 可以同时运行上千个实际进行事务处理地线程。

某些 Java 类有同步方法，增加 JServs 的数目可以增加并行的程度。

冗余可以增加系统的安全性。如果一个 JVM 崩溃，或者某个 servlet 执行了 System.exit()，或任何其它的问题发生，其它的 Apache JServ 都将照常工作。

Servlet 区可以分布在不同的 JServ 引擎上。由于 Jserv 可以不同的 userid 来运行，从而也可增强安全性。

每个 JServ 都可以有自己特有的 CLASSPATH 环境变量。

每个 JServ 可以分别使用不同版本的 JDK 或 JRE 选项（如-nojit、- verbosegc、...）。

一个 JServ 可以运行在一个 64*CPU 的机器上，而在 64 个廉价 PC 上则可以运行任意数目（64，128，或更多）的 JServs。

当然，你可以在一台机器上运行多个 JServ，每个仅需要一个 IP 地址和 TCP 端口号对。

现在在 Apache 上对指明的 JServs 数目有限制，这个限制是在编译时定好的，可以通过重新编译来改变之。施加该限制仅是为了避免内存的过度紧张。现在的限制值是 25。

开放性

Jserv 由 100%纯 Java 代码实现，它可以运行在任何具有 JVM 的机器上。所以同一时刻，你可以使用若干 OS/2 平台、若干 Linux、若干 Solaris、若干 NT ...

Apache 本身也能够运行在不同的硬件和操作系统平台上，它使用基于 TCP/IP 的 Apache Java Protocol (AJP)来发送请求给 JServs。

众所周知，Apache 在一些免费的 OS 如 xBSD 或 Linux 上是非常高效的。

安全性

- 你可以用不同的用户身份和权限来在相同或不同的机器上运行 Apache 和 JServ。
- 你可以在各个 JServ 上分别设置多个 servlet 区。
- JServ 支持防火墙的设置（AJP 协议仅使用一个 TCP 端口）。
- JServ 支持 ACL 机制，只允许来自指明的特定主机的 AJP 请求。
- Apache 和 JServ 之间的通信可以使用密钥来实现身份认证。
- Apache+SSLeay 支持对 https URL 的访问进行 128 位的 SSL 加密访问（界于浏览器和 www 服务器之间）。
- 服务器的冗余（重复设置的 Apache 和 JServ）可以提高连续服务的程度。
- Apache + JServ 不依赖于任何外部的部件。整个系统中不存在“牵一发而动全身”的要害点。

负载平衡

第 0 级：

一台机器运行 Apache，一台运行 JServ，该 JServ 包含了所有的 servlet 区。

这是 Apache JServ（没有负载平衡）当前的缺省模式。JServ 可以以自动模式启动。

第 1 级：

一台机器运行 Apache，N 台运行 JServ，每个 JServ 负责各自的 servlet 区。这是 1.0 版本下的一个可能模式。你必须指定一个 JServ/zone。Apache 不能自动启动不同的 JServ，你必须把手工模式设为 true。

第 2 级：

一台机器运行 Apache，m*n 台运行 JServ。

在各个 JServ 上，每个 servlet 区均负责一定的负载。你必须使用新的"balance"参数。

对于每个区，你都必须指定一系列地位相同的 JServ 主机，基于一个简单但高效的算法，既考虑各主机负载，又引入随机因素，来实现各个 JServ 之间的负载平衡。

每个 httpd 进程（在一个 Apache 服务器中）随机地（取模）为每个 servlet 区选择一个缺省的目标，并且提出对该目标的请求（此时会话尚未建立）。底层的操作系统确保各个进程共享 CPU 资源。

第 3 级：

p 台机器运行 Apache，m*n 台运行 JServ。和第 2 级相同，所有的 p 个 Apache 服务器必须使用相同的配置文件来使会话在所有服务器上保持活动（必须使用相同的路由参数）。

会话处理

一个会话一旦建立，是限于一个特定 JServ 内部的。这样就可保证在同一会话中的下一个请求是传递给适当 JServ 的，而不管请求来自何处（即：任何 Apache 均可）。Cookies 被用来分辨各个会话，其中含有设置该 cookies 的 JServ 的标记。

这确保在 Apache 出现问题时系统能容错。

会话的工作过程

会话是基于 servlet 设置的 cookie 实现的。

每当请求从平衡的 ServletMountPoint 发出，mod_jserv 都要选择一个 JServ 来进行处理。

mod_jserv 把 cookie trailer（尾）(the ApJServRoute = eg：JS3 for PC3)加入到传给 JServ 的环境变量中。

JServ 则又在将会话 cookie 返回给浏览器前把该 cookie trailer 附加到其尾部。

下一次在访问时，httpd 服务器（可能与先前设置会话 cookie 的主机相同，也可能不同）将接收请求，查看头域来找到会话 cookie，并确定是哪个 JServ 拥有该 cookie trailer，然后把该请求再发送给对应的 servlet 引擎。

如果在一个给定域（一个 round-robin DNS 所辖范围）中有多于一个 Apache 服务器，则所有的 Apache 必须使用相同的循环参数。

容错

没有要害点

若你有多个 Apache 服务器，一个终止了，整个系统仍然正常工作，且仍在所有的 https 服务器中平均分配 SSL 会话。

当然你有权选择是否使用负载平衡机制（round-robin DNS 或专用硬件），但若使用，用户的会话将不会因一个 Apache 的垮掉而中断。

若你有多个 JServ 主机，一个终止，整个系统仍然正常工作，涉及的会话会由其它 JServ 主机平均分担。

任何一个 Apache 服务器都可以将请求路由到任意一个 JServ 上（以后会话的维护不依赖于与负载平衡相关的硬件和 Apache 服务器。

只要还有一个 Apache 和一个 JServ 在运行，你的系统就仍在工作。

负载平衡实现了，容错也就自然具备了。

所有的 httpd 共同拥有一块内存区，其中存有每个 JServ 的最新状态。

只要一个 JServ 被某 httpd 进程标记为“不可用”，同一主机上的其它进程也会知道该信息，从而可以避免重复的连接失败或 TCP 连接超时。

对应的当前请求及后续请求将会被重定向到同一 JServ 集合中其它的 JServ 上。

另外专门有一个“watchdog”程序在默默地工作，它不断地试图去连该 JServ，直至成功。一旦成功，该 JServ 便又可以被 httpd 进程所使用。

如果请求被重定向到另外的 JServ 上，则会话将被中断。已有会话将认为是非法的，因为会话不能跨越网络（不遵循 Servlet API 规范）。

例 1——Apache 容错

不妨假设我们有 2 个 Apache 服务器，分在不同的主机上。

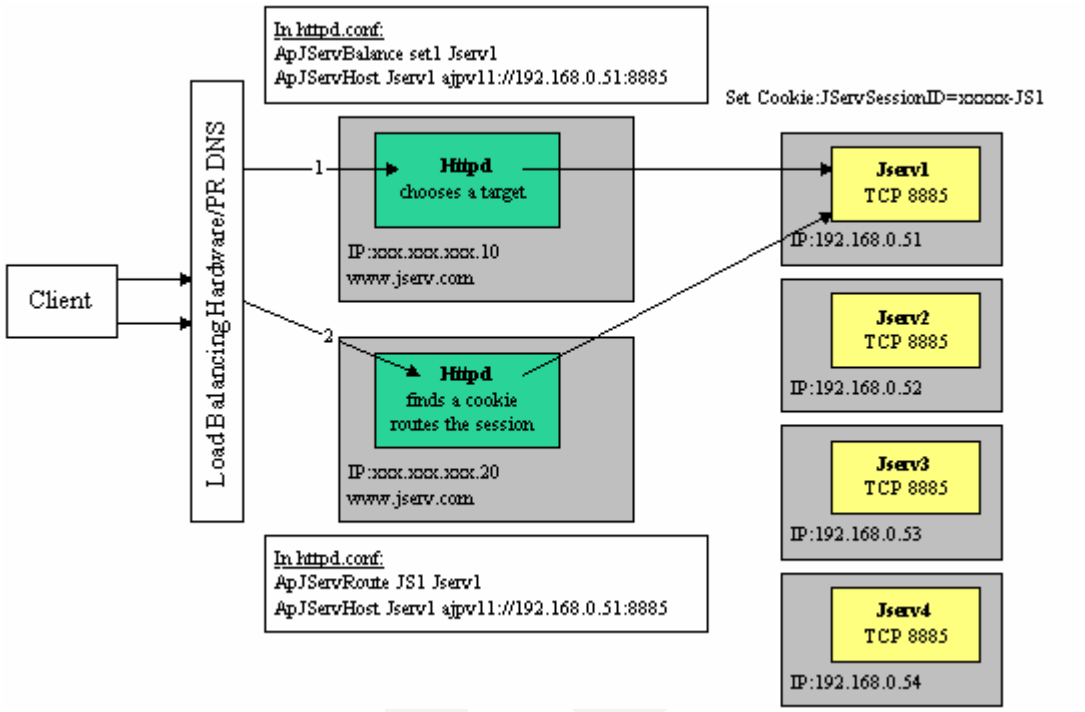


图 8.18

第一步：

a. Web 客户请求 www.jserv.com 的 80 端口。

b. www.jserv.com 是一个负载平衡的系统，它实际将请求传给监听 3000 端口的服务器 111.222.333.10。

c. 在 111.222.333.10 的 3000 口上监听的 Apache 服务器随机选择一个 Jserv 主机——192.168.0.51 上的 8885 口来处理该请求。

d. 在 192.168.0.51 的 8885 口上的 Jserv 机器响应该请求，返回对应页面，并附加上带有 JServ 会话 ID 和“xxxx-JS1”值的 cookie。

第二步：

a. Web 客户请求 www.jserv.com 的 80 口，索取另一页面。

b. www.jserv.com 将该请求解析给 111.222.333.20 的 3000 口上的 Apache 服务器（一个与上次不同的机器）。

c. 该 Apache 服务器识别到请求附带的 JServ 会话 ID cookie，并注意到在 cookie 最后有一个 Jserv 标识“JS1”。

d. 该 Apache 服务器知道 JS1 来自 192.168.0.51 的 8885 口，于是将该请求传递给该先前的 Jserv 服务器。

例 2——JServ 负载平衡及容错

第一步：JServ **负载平衡**

a. **客户** A **请求** servlet A1。

b. httpd **服务器选择一个目标** JServ **引擎** A1'。

c. **结果：设置了一个带** JS1 **标的** cookie。

d. **客户** B **请求** servlet B1。

e. httpd **服务器选择了一个目标** JServ **引擎** B1'。

f. **结果：设置了一个带** JS2 **标的** cookie。

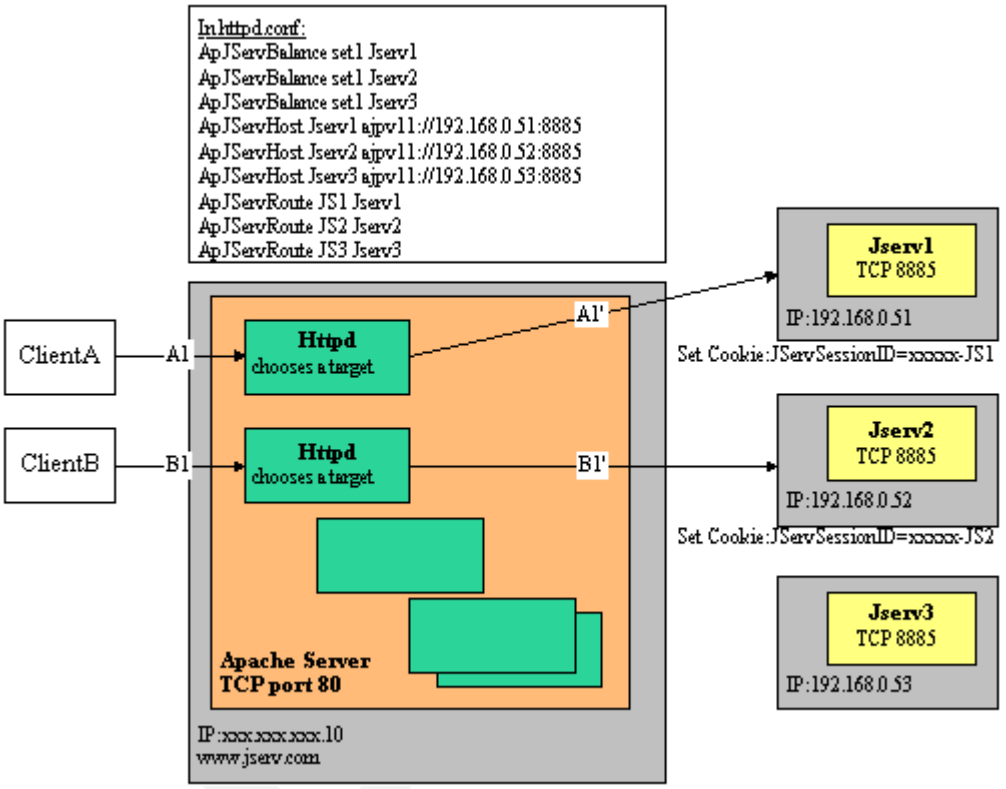


图 8.19

第二步：会话处理

a. **客户** B **请求** servlet B2

b. httpd **服务器识别到** cookie JS2。

c. **该请求被传递给** JServ **引擎** B2'，**而现在** JServ A1'**发生严重问题而垮掉**...

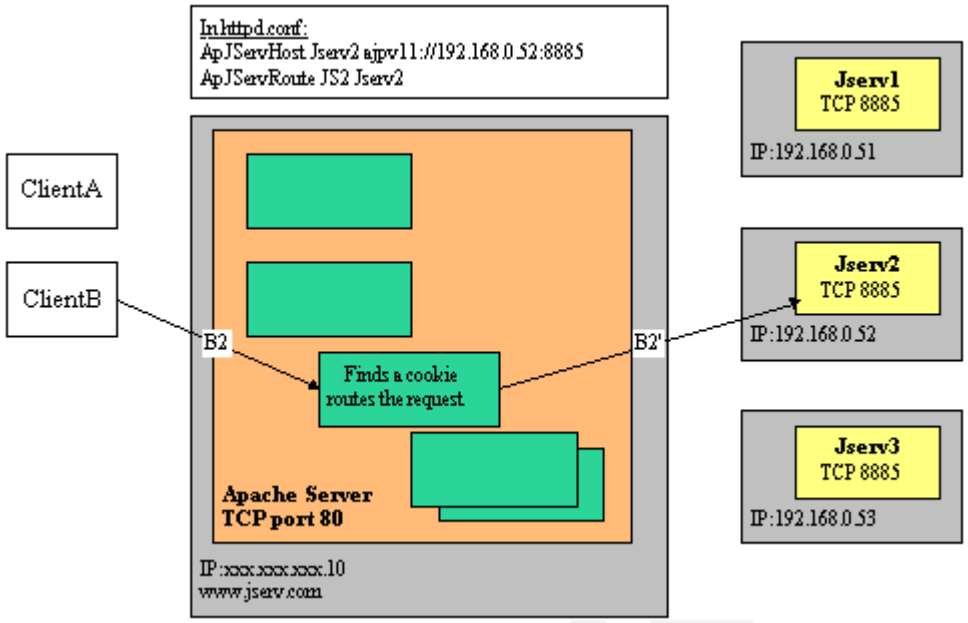


图 8.20

第三步：JServ **容错**

a. **客户** A **请求** servlet A2 **（附加有先前设置的** cookie JS2**）。**

b. httpd **服务器识别到** cookie JS1**。**

c. **该请求被传递给** JServ **引擎** A1'**时出现问题（**httpd **进程在共享内存中标记** JServ1 **已死掉）。**

d. http **服务器寻找替代者，把请求传给** JServ A2"**。**

e. A2"**的处理结果：若需建立会话，则在建立会话时会设置** cookie JS3**（**JS2 cookie **则被删除掉）。**

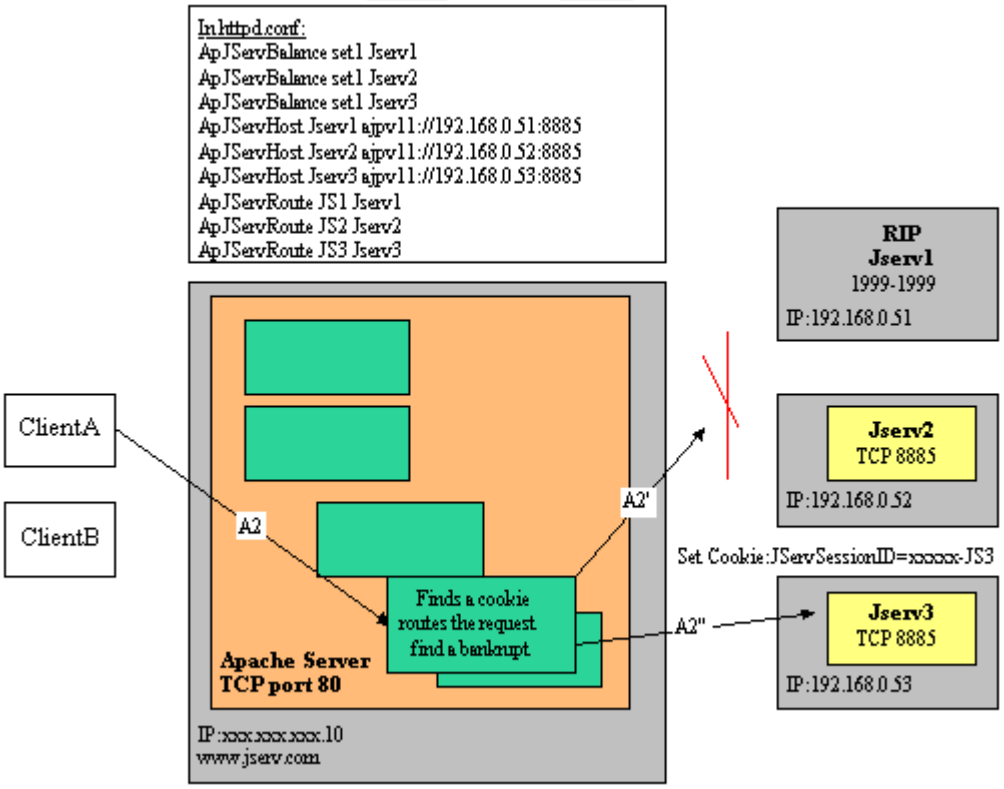


图 8.21

易管理

关于 JServ 状态的所有信息存在共享内存中，可方便的访问和更新。

你可以随时停掉一个 JServ 来进行维护，而不会影响你的客户，因为他们的请求会被传递给其它的 JServ 来处理。

管理员可以更新共享内存中特定 JServ 的状态，把它标记为 SHUTDOWN，从而使 httpd 进程察觉到该信息，将后来的请求定向到其它的 JServ。

Monitoring tasks（监控任务）可以读取在共享内存中 JServ 的状态，并给出报告（该监控任务甚至可以是一个受保护的 CGI 程序）。

8.3.3. 配置

你不需要在 Java 端做任何特别的事，也不需要在属性文件中设置什么新参数。

管理员在安装了 JServ 后，可以选择是否在 Apache 上使用负载平衡。

具体的配置示例见下文。

1 - 定义主机和路由参数

附带包含 JS2 标记的 cookies 的请求将基于 ajpv11 协议被路由到 PC2——192.168.0.52. 的 8888 口上。

2 - 定义主机 weight

```
ApJServBalance set1 SPARK 4
```

缺省 weight 为 1。本例（SPARK 机）则是一个 4CPU 机。

3 - 定义内部相互等价的 JServs 集合（这里称为 set1）

设置 "set1" 包含 PC1 + PC2 + PC3 + SPARK。

4 - 定义负载平衡的 servlet mount point

对/servlet 的请求将在 Jserv 集合 "set1"上进行负载平衡。

SPARK 机器应该比其它机器要承担 4 倍的请求负载。

若一个 JServ 垮掉，请求会在"set1"集合中进行重定向。

5 - 定义共享内存文件

定义在各个 Apache 进程间作为共享内存使用的文件的路径（需使 JServ 的 userid 有读+写+创建的能力）。

该路径可以是绝对的也可是相对的。

```
ApJServShmFile log/jserv_shm


<IfModule mod_jserv.c>
######################################################
# Apache JServ Configuration File #
######################################################
# Note： this file should be appended to httpd.conf
ApJServManual on
ApJServMount /oldservlet ajpv11：//192.168.0.1：7777/zone2//old style config
ApJServMount /servlet balance：//set1/zone1
ApJServBalance set1 PC1
ApJServBalance set1 PC2
```

```
ApJServBalance set1 PC3
ApJServBalance set1 SPARK 4
ApJServHost PC1 ajpv11：//192.168.0.51：7777
ApJServHost PC2 ajpv11：//192.168.0.52：8888
ApJServHost PC3 ajpv11：//192.168.0.53：9999
ApJServHost SPARK ajpv11：//192.168.0.54：7777
ApJServRoute JS1 PC1
ApJServRoute JS2 PC2
ApJServRoute JS3 PC3
ApJServRoute sp1 SPARK
ApJServShmFile log/jserv_shm
ApJServSecretKey DISABLED
</IfModule>
```

8.3.4. 内部实现

负载平衡算法

所有工作都是在 mod_jserv **中完成的。**

当 Apache **重新启动时，配置文件（**httpd.conf**）被解析。对于每个** servlet **引擎集合，均要定义一个所涉及引擎的循环序列。每个引擎可以在序列中出现** n **次，这里的** n **是上面所描述的引擎的权重。**

在本例中，循环序列为：

```
PC1 - PC2 - PC3 - SPARK - SPARK - SPARK - SPARK
```

具体的负载平衡算法如下：

```
for each HTTP request coming for a balanced Servlet mount point：
do
    if is the first HTTP request for a balanced Servlet mount point
    then
    process_mount_default_target = randomly chosen in set
    endif

    if a session cookie is found
    then
        the process finds the JServ which owns thesession (using ApJServRoute
    parameters)
        if the JServ is not dead/stopped (in shared memory)
        then
            the process sends the request to this JServ.
            if the JServ replies
            then
                return
```

```
            else
                mark the JServ dead (in shared memory)
            endif
        endif
    endif

    # here we have either got a request without session cookie
    # or a broken session. In this last case， we send te request
    # to another target in the list without saying it to the client.
    # (the application will have to notice the broken session anyway)

    if process_target != process_mount_default_target AND
     our process_mount_default_target is alive (in shared memory)
    then
        process_target = process_mount_default_target
    endif
    while (process_target exists)
    do
        if process_target is alive (in shared memory)
        then
            the process sends the request to this JServ.
            if the JServ replies
            then
                return
            else
                mark the JServ dead (in shared memory)
            endif
        endif
        process_target = next in set
        done
    done
```

Watchdog 进程

在 Apache **启动时，会有一个特殊的** httpd **进程被创建。它并不监听** HTTP **请求。该进程每** 10 **秒睡眠一次，读共享内存并试图去连接死掉的** JServ **引擎。当某个** JServ **有应答时，它便会在共享内存中被标记为“正在运行”。**

具体的算法如下：

```
while true
do
    sleep(10)
    if this process is not the default watchdog (in shared memory )
       break
    endif
```

```
    for each JServ present in shared memory list
    do
       if JServ.state = down
        connect JServ
       if failed
        continue
       else
        mark it alive in shared memory
       endif
    endif
    done
done
```

共享内存

共享内存用来完成 httpd **进程之间的通信。**

每个 httpd **进程都以读写方式打开一个普通磁盘文件，并** mmap**（进行与内存间映象）它。这使该文件可以象内存一样被使用。由于所有进程并发地运行，所以某进程对该内存区的任何改动均会被其它进程很快觉察。这就为各** httpd **进程间共享关于** Jserv **服务器的状态情况提供了一种简便的方式，可避免在目的** JServ **垮掉时大量的徒劳的连接。**

该共享内存可以允许任何进程读写之（通过文件访问权限），这样我们完全可以想象到：使用 CGI **程序、**SNMP **代理及由** inetd **启动的程序访问该内存区，可以为我们完成一些管理任务。**

需要注意，该内存区中的信息不是 MT-**安全（**Multi-Thread Safe**）的，也不是** MP-**安全（**Multi-Process Safe**）的，因为访问不是同步的。换句话说，此共享机制实际并不保证多个线程或进程可以安全地并发访问。**

为什么？这是因为容错部分并无此需求。如果我们使用共享计数器，那将是非常烦人的（以点击次数为例：LB **算法虽可用来完成负载的分配，但它只适合于** 1 **个** Apache **及** N **个** JServ **这样的方式。而随机+权重也并非是一个太坏的选择，且还容易实现** ;-)

该内存区文件不是可直观读写的，你应该用 C/Perl **程序去访问它（在** jserv.h **文件中详细的格式定义）。**

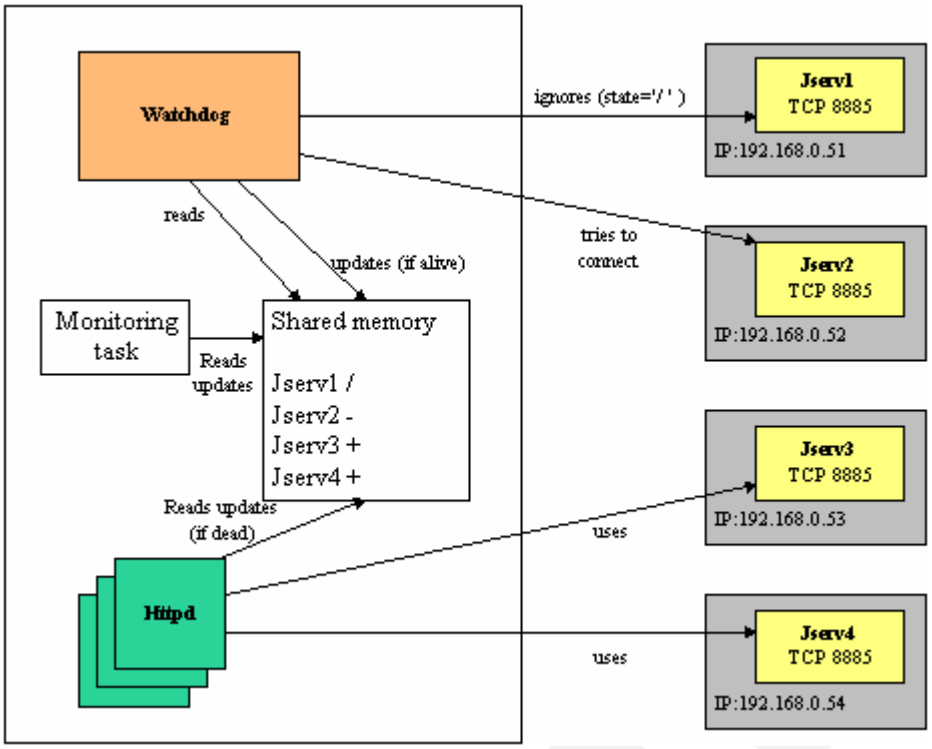


图 8.22

8.3.5. 内部状态- 管理任务

每个 JServ 都在共享内存中进行了描述。其中一个参数就是服务器的状态。该状态可以是：

- UP（内部表示为 '+'）：表示 JServ 正在运行。
- DOWN （内部表示为 '-'）:表示 JServ 无法应答（Watchdog 可以将该状态改为 UP）。
- SHUTDOWN_IMMEDIATE（内部表示为 'X'）：JServ 被管理员所终止，不允许再有请求。
- SHUTDOWN_GRACEFUL （内部表示为 '/'）：JServ 被管理员所终止，不允许再有请求，但现存的会话仍可继续（比如要求它可以 10 分钟后终止）。

要进行一个不太紧急的管理任务时，管理员可以把 JServ 的状态从 UP 改为 SHUTDOWN_GRACEFUL，等几分钟直至其又从该状态变成 SHUTDOWN_IMMEDIATE，然后就可以进行管理操作了。管理员应该等多长依赖于你的应用。

一旦设为 SHUTDOWN*，watchdog 将不会再去连接它并设置其状态了。

若要进行的管理非常紧急，则管理员可以直接将状态从 UP 改为 SHUTDOWN_IMMEDIATE，然后进行操作。这确保在进行管理时不会再有请求被路由到该 JServ 上。

如果整个站点有 10 个 Apache，则管理员必须逐个地修改每个服务器地共享内存文件。

在完成管理任务后，管理员便可将状态由 SHUTDOWN_IMMEDIATE 改回 UP。

那么如何读取/修改共享内存文件呢？现在还没有现成的工具。该文件包含有可读的二进制数据。你可以用自编的程序（C 或 Perl），或许以后也会出现这样的工具。emacs 或 vi 或 Win32 下的 Notepad 都不太合适做这件事。

其实编一个类似工具并不复杂，就和读一个普通文件类似，你可以以 src/modules/jserv/jserv_mmap.c 为参照。

8.3.6. 大型站点

使用 Apache 和 JServ 构建一个没有要害点的大型站点没有一点问题。基本思想是对每个 Apache 服务器创建不同的 JServ 集合，而不是每个 Apache 都有 n 个（JServ 的数目）可能的选择。因为那将是对内嵌的负载平衡能力的限制。

举一例，比如 Apache1 的引擎集合中有 5 个 JServ：JServ1，JServ2，JServ3，JServ4，JServ5。Apache2 则有另外的 5 个 JServ： JServ11， JServ12， JServ13， JServ14，JServ15。

我们假设你已有 Round Robin DNS 来完成 www.jserv.org 在 2 个 IP 地址上的静态分配及一个 fail-over（容错设备）（某些能同时支持两者）。

- 一个客户提出连接请求，并从 DNS 处得到对应 www.jserv.com 的 IP1。
- 该请求被送到 Apache1 上。
- Apache1 使用它的负载平衡算法在它的 JServ 集合中选择一个引擎（比如说 JServ1）。
- JServ1 设置一个 cookie （JServ 会话 Id=xxxx.JS1）到客户的浏览器中。
- 基于已建立会话的客户的后续请求，也被路由到 JServ1。
- 这时 Apache1 死掉了。
- 按预计（由于我们有负载平衡设备），该请求又送到 Apache2。
- 原来的 cookie（JServ 会话 Id=xxxx.JS1）被识别到，于是 Apache2 将请求交给 JServ1 来处理。
- 每个 Apache 从不把请求传给在其 JServ 集合之外的任何引擎，除非是已经建立的会话。

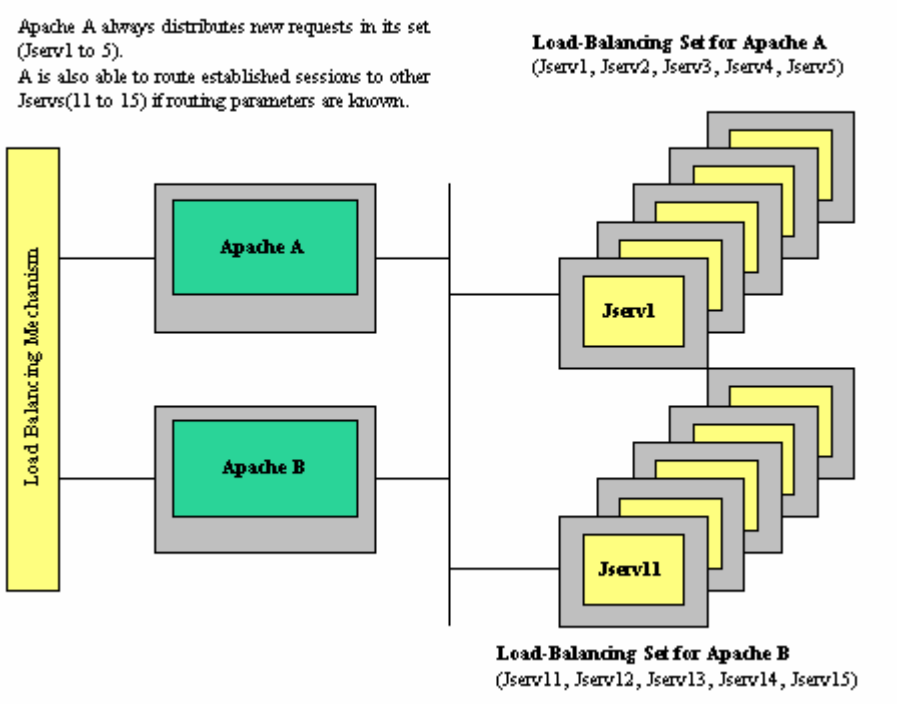


图 8.23

```
#######################################################
# Apache JServ Configuration File for Apache A #
#######################################################
# Note： this file should be appended to httpd.conf
```

```
ApJServManual on

ApJServMount /servlet balance：//set1/zone1

# my own set of JServs for servlet zone "zone1"
ApJServBalance set1 JServ1
ApJServBalance set1 JServ2
ApJServBalance set1 JServ3
ApJServBalance set1 JServ4
ApJServBalance set1 JServ5

ApJServHost JServ1 ajpv11：//192.168.0.51：7777
ApJServHost JServ2 ajpv11：//192.168.0.52：8888
ApJServHost JServ3 ajpv11：//192.168.0.53：9999
ApJServHost JServ4 ajpv11：//192.168.0.54：7777
ApJServHost JServ5 ajpv11：//192.168.0.55：7777
ApJServHost JServ11 ajpv11：//192.168.0.61：7777
ApJServHost JServ12 ajpv11：//192.168.0.62：7777
ApJServHost JServ13 ajpv11：//192.168.0.63：7777
ApJServHost JServ14 ajpv11：//192.168.0.64：7777
ApJServHost JServ15 ajpv11：//192.168.0.65：7777

ApJServRoute JS1 JServ1
ApJServRoute JS2 JServ2
ApJServRoute JS3 JServ3
ApJServRoute JS4 JServ4
ApJServRoute JS5 JServ5
ApJServRoute JS11 JServ11
ApJServRoute JS11 JServ12
ApJServRoute JS11 JServ13
ApJServRoute JS11 JServ14
ApJServRoute JS11 JServ15

ApJServShmFile log/jserv_shm

ApJServSecretKey DISABLED
</IfModule>

########################################################
# Apache JServ Configuration File for Apache B #
########################################################
# Note： this file should be appended to httpd.conf
ApJServManual on
```

```
ApJServMount /servlet balance：//set1/zone1

# my own set of JServs for servlet zone "zone1"
ApJServBalance set1 JServ11
ApJServBalance set1 JServ12
ApJServBalance set1 JServ13
ApJServBalance set1 JServ14
ApJServBalance set1 JServ15

ApJServHost JServ1 ajpv11：//192.168.0.51：7777
ApJServHost JServ2 ajpv11：//192.168.0.52：8888
ApJServHost JServ3 ajpv11：//192.168.0.53：9999
ApJServHost JServ4 ajpv11：//192.168.0.54：7777
ApJServHost JServ5 ajpv11：//192.168.0.55：7777
ApJServHost JServ11 ajpv11：//192.168.0.61：7777
ApJServHost JServ12 ajpv11：//192.168.0.62：7777
ApJServHost JServ13 ajpv11：//192.168.0.63：7777
ApJServHost JServ14 ajpv11：//192.168.0.64：7777
ApJServHost JServ15 ajpv11：//192.168.0.65：7777

ApJServRoute JS1 JServ1
ApJServRoute JS2 JServ2
ApJServRoute JS3 JServ3
ApJServRoute JS4 JServ4
ApJServRoute JS5 JServ5
ApJServRoute JS11 JServ11
ApJServRoute JS11 JServ12
ApJServRoute JS11 JServ13
ApJServRoute JS11 JServ14
ApJServRoute JS11 JServ15

ApJServShmFile log/jserv_shm

ApJServSecretKey DISABLED
</IfModule>
```

8.3.7. 需注意的问题

不要使用过大的 servlet **引擎权重，因为在循环列表中必须确切地出现对应次该引擎（例如对一个配有** 2*400MHz CPU **的机器设一个** 800 **的权重就不是一个好主意）。**

活动的 JServ **的最大数目是** 25**。要增加之，可以增加** jserv.h **中** NB_MAX_JSERVS **常量的值，并重新编译。**

另外遗憾地告诉各位：全部的功能在 Unix **类的平台均已实现（已在** Solaris/Linux **上进行了测试）。但** NT **版本却是部分的实现：现在还没有** watchdog **的功能。哪个志愿者感兴趣**

不妨一展身手。

现在 NT 已实现了负载平衡和共享内存。

Watchdog 进程是 Apache 服务器启动时由主程序创建的，所以它继承（太多的 IMHO）了 Apache 服务器的特征：以 Apache 的 userid 身份运行（当 TCP 端口号< 1024 时，userid 必须是 root）。这实际是非常危险的，虽然利用这一点并不那么困难，因为该进程并不监听 HTTP 请求。将来这一问题会尽快解决。

8.3.8. 小技巧

- **不激活负载平衡**

你当然可以在你系统的一部分或全部范围内不使用负载平衡机制。

在上面的配置例子中：

```
ApJServMount /svlet1 ajpv11：//192.168.0.1：7777/sv <<<<< 这与负载平衡无关。
ApJServMount /servlet balance：//set1/zone1 <<<<<此是相关处。
```

- **在没有共享内存和 watchdog 时使用负载平衡**

没有共享内存和 watchdog 也仍然可以使用负载平衡。

你只需将 httpd.conf 文件中的 ApJServShmFile 行注释掉：

```
######ApJServShmFile log/jserv_shm
```

仍使用负载平衡的语法时：

```
ApJServMount /servlet balance：//set1/zone1
.../...
```

这时将不再使用各 httpd 进程共享的内存，你也不再能用 watchdog 来监测 JServ 的状态。

- **移去所有的负载平衡代码**

只需去掉 src/modules/jserv/jserv.h 文件中下行的注释：

```
#define LOAD_BALANCE
```

并对 JServ 和 Apache 进行重新编译：

```
make clean; make; make install
```

第9章 HTTP1.1 协议

网络，简单地说，是一些相互通信的自主实体的集合，如各种网络服务器软件和客户端软件，其中也包括了 WWW 服务器和浏览器。这些实体要相互通信，就必须遵守一些“礼节”和“规则”，这在网络范畴中称为协议。具体到以 Apache 为代表的 WWW 服务器与诸多浏览器之间的协议就是 HTTP 协议。

协议是网络的灵魂，协议设计的好坏直接关系到网络应用发展的现状和前景。目前在 WWW 中广泛采用的是超媒体传输协议——HTTP 的 1.0/1.1 版本，即 RFC1945 和 RFC2068，

本章在本书中相对独立，只对 HTTP 协议感兴趣的读者可以只读本章。

本部分较全面地介绍了 HTTP1.1 协议的基本原理、重要概念和术语，包括 HTTP 连接、HTTP 消息和消息代码、HTTP 操作、HTTP 头域等，同时穿插一些和 HTTP 实现有关的内容，期望能对读者在 WWW 方面的使用和应用开发有所帮助。

第1节 概貌

9.1.1. HTTP 协议的特征

HTTP 位于 TCP/IP 协议体系结构的应用层，是一种面向分布、协作的超媒体信息系统协议，HTTP 的本质特征是数据表示的构造和协商，此外，它还具有元数据表示、无状态、面向对象等特点。

HTTP 协议自 1990 年便应用于 WWW 环球信息网中。HTTP 的第一个版本称为 HTTP/0.9，是一个在 Internet 上进行基本数据传输的简单协议。由 RFC1945 定义的 HTTP/1.0 则进一步支持 MIME 格式消息的传输，支持关于所传输数据的元信息的使用，支持基于请求/响应语义的修饰符的使用。但 HTTP/1.0 没有充分考虑对级联代理、缓冲、持续连接和虚拟主机等机制的支持。而 HTTP/1.1 则在 HTTP/1.0 基础上又进行了增强，确保各种功能的实现更加可靠。

HTTP 提供开放的操作方法（Method）集合，因此除了用于 Web 中超媒体资源获取、用户前端更新和资源注解等功能外，通过对请求操作（Request Method）的扩充，还可以广泛用于其它系统应用，比如域名服务器和分布式对象管理系统等等。资源标识是 HTTP 的核心问题，在 HTTP 中通过统一资源标识符（Uniform Resource Identifier）来标识被操作的资源。

HTTP 是一种元协议（Generic Protocol），它所描述的是一种通用或一般的语义和语法结构，称之为元结构（Generic Structure），因此除了 Web 本身，它还可用于用户软件（或用户进程，英文称为 User Agent，有别于下文提到的用户代理——User Proxy）、网关（Gateway）与其它 Internet 应用系统之间的通信，例如 SMTP、NNTP、FTP、Gopher 和 WAIS 等，从而可实现各类应用资源超媒体访问的集成。

HTTP 具有面向对象的特性，资源对象和相应的操作方法（Method）可以一起传送。一个典型的例子是 Java 类代码的传送，它实现了软件在网络上的动态执行，刻划了一种新的网络计算模式。HTTP 还提供了丰富的执行语义，资源的操作方式有 OPTIONS、GET、HEAD、

POST、PUT、DELETE、TRACE，它们不实指某种具体的媒体类型，可以认为是一种面向用户的元操作，具有很好地可扩充性。

HTTP1.1 的目标是支持现实世界中已广泛使用的各种配置，为高可靠性或至少错误可靠检测的 Web 应用开发提供协议支撑。

9.1.2. HTTP 基本术语和概念

HTTP1.1 规范说明中大量使用了 RFC822（Internet 消息报文格式标准）中的扩展 BNF（巴科斯瑙尔范式）和元结构表示，同时也借用了许多有关 MIME 的定义，因此 HTTP 和 Internet 电子报文有密切的联系。

下面就对在 HTTP 连接中涉及的一些基本概念做阐述：

- 连接——Connection

两个程序之间为了通信而建立的传输层虚电路。

- 消息——Message

类 MIME 格式的消息包，是 HTTP 通信的基本单元。HTTP 只有两种类型的消息，分别为请求消息（Request）和应答消息（Response）。

- 实体——Entity

实体是请求或应答消息携带的数据，包括实体头域（Rntity-Header Field）和实体内容（Entity-Body），即通常所说的头和体两个部分。实体头域包含了数据的元信息。

- 资源——Resource

资源是一种可被 URI 地址标识的网络数据或服务对象。某一资源可以有多种表示（Representation），例如不同的语言、数据格式、大小和分辨率等。这些不同的表示都叫做该资源的变体（Variant）。

- 内容协商——Content Negotiation

内容协商指为了响应请求而在多种实体表示（变体）中进行选择的机制。对任何响应的实体的表示的选择都可以进行选择。

- 客户（或称为客户端，客户进程，客户程序等等）——Client

一个为了发送请求信息而建立连接的程序或进程。

- 用户软件（用户进程）——User Agent

发起请求的客户软件或进程。在 HTTP 中它们通常指浏览器、具有 Web 浏览功能的文字编辑器和蜘蛛人（Spider，一种能漫游 Web 世界的软件“机器人”，又叫 Robot）或其它的类似用户端工具软件。

- 服务器（或称为服务进程，服务器程序等等）——Server

为了以发送应答消息方式对各种请求消息进行服务而建立连接的应用程序。注意，任何程序都可能既是客户程序又是服务程序，所以这里所说的服务器仅仅指在某次特定连接中应用程序所“扮演”的角色。服务器的种类有原始服务器、客户代理、信关和隧道（Tunnel）等。

- 客户代理——Proxy

一种中继程序，它代表其它客户发送请求，从其行为看，既是服务器又是客户端。它对到来的请求进行处理，或经过适当处理后前传给其它服务器。一个代理必须既实现了 HTTP 客户端的功能，也实现了服务器的功能。

- 网关——Gateway

网关指为其它服务器提供中继功能的服务器。与代理不同，网关在客户端程序看来，就是原始服务器，客户程序并不能意识到它的存在。

- **隧道——**Tunnel

隧道指在两个连接之间进行中继且不进行任何中间干涉和改变的程序。隧道一旦工作后，可以认为是对 HTTP **连接来说不存在的。而只有当两个连接都关闭后隧道程序才结束工作。**

- **缓存——**Cache

缓存是对程序中存放响应消息的本地存储区和控制消息缓冲机制、检索和删除消息的子系统的统称。缓存存放可缓冲的（Cachable**）响应，从而在将来客户对相同资源进行请求时，直接将该响应返回，从而减少响应时间和网络带宽消耗。客户或服务器中都可以包含一个缓存，但对隧道服务器则不适用。**

- **可缓冲的——**Cachable

响应是可缓冲的，是指该响应可以存储到缓存中以备在以后有相同请求时可快速给出应答。对于 HTTP **响应的可缓冲性的具体判定，在下文适当地方将逐个提到。**

- **原始服务器——**Origin Server

Web **应用的典型环境是客户程序通过和服务器程序建立的可靠链路连接访问服务器上的资源。这种服务器称为原始服务器，它存放着被请求的资源，有别于其它的服务器。**

上面讲述的术语将在下面频繁出现。

9.1.3. 操作概貌

HTTP **协议是一个请求/响应协议。客户首先给服务器发送请求，请求由请求方法、**URI **地址、协议版本、**MIME **格式的包含请求修饰符、客户信息和可选的具体内容的消息构成。服务器对请求做出响应。响应中有一状态行，包括了所使用的协议版本信息、成功或出错代码。状态行后，是一个** MIME **格式的消息，其中含有服务器信息、实体元信息和可能有的实体内容。**

大多数 HTTP **通信都是由客户端程序发起的，表现为对原始服务器上的某个资源的请求。最简单的情况是，只建立了客户端程序（**UA**）和原始服务器（**O**）之间的一个连接（**V**）。我们称这个连接为虚电路，也称为** HTTP **请求/应答链路，如图** 9.1 **所示。另外，**HTTP **还支持可靠链路中继的概念。**

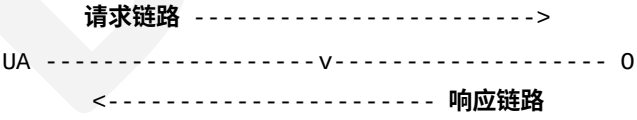


图 9.1

中继结点有代理、网关和隧道三种。网关代表其它服务器接收客户请求，对用户来说是透明的，如果可能还会对请求进行协议转换。隧道对客户和服务器都是不可见，它对通过的 HTTP **数据流不作任何改变，通常用于穿过防火墙。代理则作为一种转发代理具有行为的双重性，既作为客户端代表用户向目的服务器发出请求，又代表服务器向用户返回应答。**

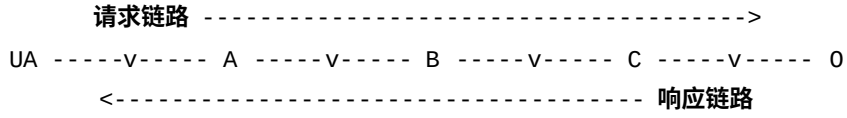


图 9.2

在图 9.2 **中，客户端程序和原始服务器之间有三个中继结点（**A**、**B **和** C**），则** HTTP **请求、应答链路将由四段彼此独立的连接构成。弄明白这一点非常重要，因为有些** HTTP **通信参数或只适用于最近的连接，或适用于非隧道类型的近邻，或只适用于整个链路的两端点，或应用于链上的所有连接。尽管该图表达的是一条线形的链，但实际上其中的每一方都有可**

能还涉及其它的通信连接。例如，B 在处理 A 的请求的同时，可能还接收来自不同于 A 的请求，并向不同于 C 的服务器转发。

除隧道方式外，通信的任何一方（包括客户程序、原始服务器和代理、网关）都可以拥有本地的缓冲区，以处理重复的请求消息。缓冲的重要性就在于缩短了 HTTP 请求/应答链路，因此减少了系统响应时间。比如若 B 中缓冲了来自原始服务器的先前对同一请求的应答，而 A 和客户端程序又没有缓存时，图 9.2 就可简化为图 9.3：

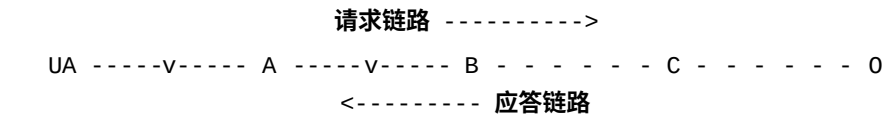


图 9.3

现实 Web 世界已有各种系统结构和配置的缓冲和代理的例子，例如有为了节省跨洋线路带宽而设立的国家级层次代理缓冲存储、广播或多点播放缓冲内容的服务系统，甚至还有通过 CD-ROM 发放缓冲内容代理机构等等。HTTP/1.1 协议中有关缓冲的原理、算法和实现机制是一个较为复杂的课题，在此我们不作详细介绍。

HTTP 通信一般是基于 TCP/IP 连接进行的。使用的端口缺省是 80，其它端口也可以用。同时，HTTP 并未排除基于其它 Internet 协议或在其它网络中实现。HTTP 协议仅假设底层可提供可靠的传输；任何可以满足此要求的协议都可做为 HTTP 实现的基础。HTTP/1.1 请求和应答结构向底层协议数据单元的转换不在本书的范围之内，请感兴趣的读者参阅其它书籍。

第2节　HTTP 协议参数

9.2.1. HTTP 版本

HTTP 协议使用“〈主版本号〉.〈次版本号〉”模式来表达协议的版本号。发送者可以使用协议版本机制来指示消息的格式和它处理 HTTP 协议通信的能力。仅仅增加消息组成而不影响通信行为，或者仅仅扩展头域的取值，并不会引起版本的变化。当对协议的改变没有影响消息解析算法时，将提高次版本号，这意味着消息语义的增加和发送方通信能力的扩展。只有当协议中消息的格式发生变化时才会提高主版本号。

HTTP 消息的版本是通过包含在消息首行中的 HTTP-Version 头域来指示的。

```
HTTP-Version   = "HTTP" "/" 1*DIGIT "." 1*DIGIT
```

注意，主版本号和次版本号必须是整数，有可能不只是一位数。这样，比如 HTTP/2.4 就比 HTTP/12.3 版本低。发送方不应发出前导的 0，万一发出，接收方应将其忽略掉。

应用程序在发送请求或响应消息时必须包含 HTTP-Version 头域，其中的版本号应是该程序兼容的最高版本。

代理和网关服务器在转发与自身兼容版本不同的 HTTP 消息时应进行特殊处理。既然协议版本反映了发送方的协议支持能力，所以代理/网关绝对不能发送一个版本高于自身的消息；如果接收到了较高版本的请求，则代理/网关或者降低请求的版本并响应出错信息，或者以隧道方式工作。而比代理/网关版本低的请求转发前可以进行版本的提升，并且给出的响应必须与请求的主版本号一致。

注意：HTTP 版本的转换可能涉及对应版本不支持的头域的改变和删除。

9.2.2. 统一资源标识符（Uniform Resource Identifiers）

统一资源标识符（以下简称 URI**）是一种格式化字符串，可用名字、地点或其它特征标识某个资源，因此** URI **有许多别名，如** WWW **地址、全局文档标识符（**Universal Document Identifiers**）、统一资源名（**Uniform Resource Names**）以及大家很熟悉的统一资源定位符（**Uniform Resource Locators**，**URL**）。**

- **基本语法**

HTTP **协议中使用的** URI **可以是绝对形式，也可以是相对某已知基** URI **的（这与使用的具体环境有关）。两种形式的不同在于绝对** URI **总是以一个模式名（**scheme**）加一个“：”开头。具体的格式定义如下：**

```
URI           = ( absoluteURI | relativeURI ) [ "#" fragment ]
absoluteURI   = scheme ":" *( uchar | reserved )
relativeURI   = net_path | abs_path | rel_path

net_path      = "//" net_loc [ abs_path ]
abs_path      = "/" rel_path
rel_path      = [ path ] [ ";" params ] [ "?" query ]

path          = fsegment *( "/" segment )
fsegment      = 1*pchar
segment       = *pchar

params        = param *( ";" param )
param         = *( pchar | "/" )

scheme        = 1*( ALPHA | DIGIT | "+" | "-" | "." )
net_loc       = *( pchar | ";" | "?" )

query         = *( uchar | reserved )
fragment      = *( uchar | reserved )

pchar         = uchar | ":" | "@" | "&" | "=" | "+"
uchar         = unreserved | escape
unreserved    = ALPHA | DIGIT | safe | extra | national

escape        = "%" HEX HEX
reserved      = ";" | "/" | "?" | ":" | "@" | "&" | "=" | "+"
extra         = "!" | "*" | "'" | "(" | ")" | ","
safe          = "$" | "-" | "_" | "."
unsafe        = CTL | SP | <"> | "#" | "%" | "<" | ">"
national      = <any OCTET excluding ALPHA, DIGIT,
                reserved, extra, safe, and unsafe>
```

关于 URL 的语法和语义等详细信息，读者可以参考 RFC1738 和 RFC1808。

HTTP 协议对 URI 的长度没有预先的限制。服务器必须能够处理它所提供的任何资源的 URI 地址，并且能处理几乎无限长度的 URI，在提供基于 GET 方法的 FORM 机制时便可能遇到这种情况。若一个 URI 太长服务器无法处理，则它应该返回 414（请求 URI 太长）的响应状态码。

注意：服务器应尽量不要使用长度超过 255 的 URI，因为一些老式的客户程序和代理服务器不能很好支持。

- http URL

“http”模式被用来基于 HTTP 协议定位网络资源。下面给出 http URL 的模式相关的语法和语义：

```
http_URL      = "http:" "//" host [ ":" port ] [ abs_path ]
host          = <A legal Internet host domain name
                 or IP address (in dotted-decimal form),
                 as defined by Section 2.1 of RFC 1123>
port          = *DIGIT
```

若端口为空或没有给出，则认为取缺省值 80。

上面定义的 http URL 的语义是：该资源所在的服务器监听该主机的指定 TCP 端口，对应资源的 URI 是绝对路径。

- URI 的比较

当比较两个 URI 地址看它们是否匹配时，客户应该使用大小写敏感的逐字节的比较。但下列情况除外：

1. 空的或没有给出的端口与该 URI 的缺省端口等价；
2. 主机名的比较是大小写不敏感的；
3. 模式名的比较是大小不敏感的；
4. 空的绝对路径与路径“/”等价。

除保留（reserved）字符和不安全（unsafe）字符外的其它字符均与它们的""%" HEX HEX"编码等价。

例如，如下的三个 URI 是等价的：

```
http://abc.com:80/~smith/home.html
http://ABC.com/%7Esmith/home.html
http://ABC.com:/%7esmith/home.html
```

9.2.3. 日期和时间

- 完整日期格式

HTTP 应用程序曾经总共支持了三种日期/时间格式：

```
Sun, 06 Nov 1994 08:49:37 GMT  ; 在 RFC822 中定义，后被 RFC1123 修改
Sunday, 06-Nov-94 08:49:37 GMT ; 在 RFC850 中定义，后被 RFC1036 废除
Sun Nov  6 08:49:37 1994       ; 是 ANSI C 中 asctime()使用的格式
```

第一种格式作为 Internet 标准，是由 RFC1123（RFC822 的更新版）定义的定长表示。第二种格式是常用格式，但它基于废除的 RFC850 日期格式，未使用 4 位数计年。HTTP/1.1 客户和服务器必须支持上述全部三种日期格式（为了与 HTTP/1.0 兼容），尽管它们在头域中

表达时间必须只使用 RFC1123 的格式。

注意：可能需要接收日期值的程序还应尽量能接收来自非 HTTP 应用程序的日期值，因为这种情况在通过代理/网关基于 SMTP 或 NNTP 进行消息的检索或粘贴时经常遇到。

所有在 HTTP 协议中使用的时间必须是格林威治标准时间（GMT）。上面前两种格式中的“GMT”标志更可以清楚地表达这个意思。而第三种实际也隐含是 GMT 时间。

严格的语法和语义定义如下：

```
HTTP-date    = rfc1123-date | rfc850-date | asctime-date
rfc1123-date = wkday "," SP date1 SP time SP "GMT"
rfc850-date  = weekday "," SP date2 SP time SP "GMT"
asctime-date = wkday SP date3 SP time SP 4DIGIT
date1        = 2DIGIT SP month SP 4DIGIT
               ; day month year (e.g., 02 Jun 1982)
date2        = 2DIGIT "-" month "-" 2DIGIT
               ; day-month-year (e.g., 02-Jun-82)
date3        = month SP ( 2DIGIT | ( SP 1DIGIT ))
               ; month day (e.g., Jun  2)
time         = 2DIGIT ":" 2DIGIT ":" 2DIGIT
               ; 00:00:00 - 23:59:59
wkday        = "Mon" | "Tue" | "Wed"
             | "Thu" | "Fri" | "Sat" | "Sun"
```

注意：HTTP 对日期/时间格式的要求仅限于通信协议部分，客户和服务器在用户界面和请求的日志记录等活动中则不受此限制。

- 秒

某些 HTTP 头域还允许使用十进制的秒数。格式为：

```
delta-seconds  = 1*DIGIT
```

9.2.4. 内容编码（Content Codings）

内容编码的值指示对实体进行的编码转换。内容编码的目的主要是为了压缩或避免实际媒体类型标志和信息的丢失。一般，实体都是以一定编码形式存储和传输，而只在接收处解码。内容编码的取值格式为：

```
content-coding   = token
```

所有的内容编码取值都是大小写不敏感的。HTTP/1.1 使用 Accept-Encoding 和 Content-Encoding 头域中定义的编码值。该值不仅用来描述内容编码，更重要的是接收方可以利用它来进行相应的解码。

Internet 特定值授权组织（Internet Assigned Numbers Authority ，IANA）负责编码取值的注册。最初的注册值有：

1. gzip：文件压缩工具“gzip”使用的编码格式。

2. compress：UNIX 通用压缩程序“compress”使用的压缩格式。

注意：使用程序名作为编码方式的标识并不是一个好习惯，上述情况是由历史的原因造成的。为了于先前的 HTTP 的实现相兼容，各种程序应该认为“x-gzip”和“x-compress”分别与“gzip”和“compress”等价。

3. deflate：由 RFC 1950 定义的“zlib”格式。

新的内容编码值应该进行注册，以保证客户和服务器之间的互操作性。

9.2.5. 传输编码（Transfer Codings）

传输编码通常被使用对实体内容进行处理从而实现网络上信息的“安全传输”。与内容编码不同，传输编码是消息的属性，而不是实体的属性。

```
transfer-coding         = "chunked" | transfer-extension
transfer-extension      = token
```

所有的传输编码都是大小写不敏感的。HTTP/1.1 使用 Transfer-Encoding 头域中定义的编码值。

传输编码类似于 MIME 中的 Content-Transfer-Encoding 值，后者是为了实现在 7 位传输服务上进行二进制数据的安全传输。然而安全传输对于 8 位的传输协议来说则有不同的含义。在 HTTP 中，消息体的唯一不安全特征在于很难确定体的确切长度，或在共享传输信道中加密数据的传输。

分块编码（chunked encoding）是最常见的一种传输编码。它将消息体分成一系列的小块，每个小块开始都有大小指示。然后还有可选的包含实体头域的注解信息（footer）。这就使得接收方能正确地接收动态生成的信息。

```
Chunked-Body   = *chunk
                 "0" CRLF
                 footer
                 CRLF
chunk          = chunk-size [ chunk-ext ] CRLF
                 chunk-data CRLF
hex-no-zero    = <HEX excluding "0">
chunk-size     = hex-no-zero *HEX
chunk-ext      = *( ";" chunk-ext-name [ "=" chunk-ext-value ] )
chunk-ext-name = token
chunk-ext-val  = token | quoted-string
chunk-data     = chunk-size(OCTET)
footer         = *entity-header
```

分块编码方式是以长度为 0 的小块为结尾的，而最后的注解信息则以一个空行为结束标志。使用的注解信息是为了能充分描述动态生成的实体。应用程序不能在注解中使用不适于做注解的头域。

所有的 HTTP/1.1 应用程序必须能够接收和解除分块编码，且能忽略掉它们不认识的传输编码扩展。一个服务器在接收到一个它以不认识的传输编码处理过的实体时，应返回 501（未实现）状态码，并关闭连接。服务器不能向 HTTP/1.0 客户发送经传输编码的内容。

代理/网关在传递一个基于 MIME 兼容协议的经过传输编码的消息之前，必须除去使用的编码。

- 处理分块编码的消息体的过程

处理分块编码的消息体的过程可用如下伪代码表示：

```
length := 0
read chunk-size, chunk-ext (if any) and CRLF
while (chunk-size > 0) {
   read chunk-data and CRLF
   append chunk-data to entity-body
   length := length + chunk-size
```

```
   read chunk-size and CRLF
}
read entity-header
while (entity-header not empty) {
   append entity-header to existing header fields
   read entity-header
}
Content-Length := length
 Remove "chunked" from Transfer-Encoding
```

9.2.6. 媒体类型（Media Types）

HTTP 协议在 Content-Type 和 Accept 头域中使用 Internet 媒体类型来提供开放的和可扩展的数据类型和类型协商。

```
media-type     = type "/" subtype *( ";" parameter )
type          = token
subtype       = token
```

在“类型/子类型”后面可以使用“属性/值”对形式的参数。

```
parameter     = attribute "=" value
attribute     = token
value         = token | quoted-string
```

类型、子类型和参数的属性名都是大小写不敏感的。参数值可能是大小写区分的，也可能不是，具体要视参数的语义。在类型与子类型、属性与值之间不能出现空格。客户端程序必须处理对应由“类型/子类型”描述的 MIME 类型的各个参数（或借助外部的应用程序来处理)），并在出现问题时通知用户。

注意：某些较老的 HTTP 应用程序不能识别媒体类型参数，所以当发送数据给这些程序时，应该只使用当时媒体类型必须的参数。

媒体类型值的登记由 IANA 组织负责。媒体类型的登记过程在 RFC2048 中有述。使用为登记的媒体类型是应该尽量避免的。

9.2.7. 多部类型（Multipart Types）

MIME 中提供了许多多部类型，即在一个体中封装了一个或多个实体。所有的多部类型都有相同的语法（具体请参阅 RFC2045 中的定义)，其中必须包含一个边界参数。消息体本身是协议元素，因此必须只使用回车换行来表示各个部分之间的体分隔符。与 MIME 不同，这里每个多部消息的结尾（epilogue）都必须是空的；HTTP 应用程序不能发送结尾（即使原来的多部类型数据中包含一个结尾)。

HTTP 多部类型数据中的每个部分都可以包含针对该部分的头域。比如，每个可以用 URL 地址标识的所含部分均需要带一个 Content-Location 头域。

总之，HTTP 客户端程序应遵循与 MIME 客户端程序相同的方式来接收多部类型的数据。若程序接收到一个无法识别的多部子类型，则应认为是“multipart/mixed”。

注意：“multipart/form-data”类型被专门用来传送基于 POST 请求方法的 FORM 数据。

9.2.8. 产品标识（Product Tokens）

产品标识用来使通信的应用程序双方能通过软件名和版本号相互识别对方。使用产品标识的大多数域还支持列出各个子产品，即应用程序中的重要部分，各个产品标识之间用空格分开。习惯上，产品的次序是按重要性从大到小排列的。

```
product         = token ["/" product-version]
product-version = token
```

例如：

```
User-Agent: CERN-LineMode/2.15 libwww/2.17b3
Server: Apache/0.8.4
```

产品标识应尽可能简短，尤其不能出现用于广告或不必要的信息。尽管在产品版本部分可以出现任何字符，但该部分应仅用来作为版本分辨（即同一产品的不同版本应该只在产品版本部分不同）。

9.2.9. 实体标记（Entity Tags）

实体标记用于区分同一个被请求资源的不同实体。头域 ETag、If-Match、If-None-Match 和 If-Range 中都使用了实体标记。实体标记由一个难懂的引用字符串（opaque quoted string）构成，可能前面加上一个弱指示符（weakness indicator）。

```
entity-tag = [ weak ] opaque-tag

weak       = "W/"
opaque-tag = quoted-string
```

强实体标记（strong entity tag）可以被同一资源的不同两个实体共享，但两个实体必须每个字节都等价。

弱实体标记使用“W/”前缀来指示。它可以被同一资源的不同两个实体共享，但两个实体必须等价，相互可以替代且语义上没有显著改变。弱实体标记只能用于弱比较（weak comparison）。

实体标记在与某资源相关的所有实体的所有版本中都必须是唯一的。当对不同 URI 的请求没有特别指明实体的等价性时，实体标记可以用于对应的实体。

9.2.10. 实体片段单位（Range Units）

HTTP/1.1 允许客户端请求服务器发送的应答消息中只包含原应答实体的一部分（实体域）。Range 和 Content-Range 头域中用到了该协议参数。一个实体可以按不同的单位分成若干个子片段。

```
range-unit       = bytes-unit | other-range-unit

bytes-unit       = "bytes"
other-range-unit = token
```

在当前版本中，HTTP/1.1 定义实体片段单位为一个字节。HTTP/1.1 程序可以忽略用其它单位指示的片段范围。

第3节 HTTP 消息

HTTP 消息包括客户端请求消息和服务器应答消息。

HTTP-message = Request | Response

这两种类型的消息都由一个起始行、一个或多个头域、一个空行（表示头域部分结束）和一个可选的消息体组成。

```
generic-message = start-line
                  *message-header
                  CRLF
                  [ message-body ]

start-line      = Request-Line | Status-Line
```

为了更加健壮，服务器应该忽略掉请求行前接收到的空行。换句话说，如果服务器读输入流等待消息到来时先读到了一个回车/换行，则应将它忽略掉。

9.3.1. 消息头域

HTTP 的头域（Header Fields）是一种重要的语法结构。它有各种类型，包括通用头域、请求头域、应答头域和实体头域等。它们均由一个域名，加上冒号，后跟一个域值构成。域名是大小写不敏感的。域值前可以有任意数目的空格或制表符，但最好只有一个。头域可以占多行，后面的每一行开头必须有一个空格或制表符。

```
message-header = field-name ":" [ field-value ] CRLF

field-name     = token
field-value    = *( field-content | LWS )

field-content  = <the OCTETs making up the field-value
                 and consisting of either *TEXT or combinations
                 of token, tspecials, and quoted-string>
```

这里的 message-header 表示消息头域，*message-header 表示消息头域可以出现任意次（可以为 0 次）；方括号表示可选；CRLF 表示回车、换行符，下面同。

不同类型头域的接收顺序并不重要，然而一般情况下最好按下列顺序发送：首先是通用头域，然后是请求头域或应答头域，最后是实体头域。

具有相同名字的消息头域可以出现在同一个消息中，当且仅当域值是“，”分隔的列表的形式（即 value0，value1，...），且在将多个头域合并成一个时不会改变消息的语义。合并的方法是将后一个域值附加到前一个域中，且用“，”分隔开。因此同名的多个域的次序会影响接收方对组合域值的解释，故而代理服务器在传递消息时不能擅自改变头域的次序。

9.3.2. 消息体（Message Body）

HTTP 消息中除了头域，还有体，用来携带与请求或应答有关的实体内容。实体内容本身或进行了某种编码后的实体内容都叫做消息体。

```
message-body = entity-body
             | <entity-body encoded as per Transfer-Encoding>
```

对实体内容进行传输编码可以保证消息安全、可靠传输。注意，传输编码是消息的特性，而不是消息体的特性，所以传输编码可以被请求/访问链路上的任何一个应用进程添加或去除。

HTTP 请求消息中 Content-Length 或 Transfer-Encoding 头域的出现说明了消息内容的存在。通常只是在请求操作允许的情况下，请求消息才能携带消息体。

对应答消息来说，是否携带消息体由请求操作和应答状态代码共同决定。比如所有对请求操作 HEAD 所作的应答消息都不能有消息体，尽管该应答消息包中可以存在实体头域。还有，所有代码为 1xx（提示），204（无内容），304（没有被修改）的应答消息也都不能携带消息体。注意，长度为 0 的消息体是可能存在的。关于响应代码的详细含义请参阅本章第 7 节。

9.3.3. 消息长度

消息中可以有消息体，体的长度的确定依据如下方法（依次进行）：

1. 任何不能带消息体的响应（如 1xx、204 和 304 响应及对 HEAD 请求的响应）必须在头域部分后紧跟一个空行，表示消息的结束。

2. 若消息中使用了 Transfer-Encoding 头域指示使用了分块传输编码，则长度要根据该编码的定义计算得到。

3. 若使用 Content-Length 头域，则该域的值就是消息体的长度。

4. 若消息使用了“ multipart/byteranges ”媒体类型，且其可以自行确定界限，则从该类型知道的长度就是体的长度。除非知道接收方能解析，否则发送方不应使用某种媒体类型。

5. 根据服务器关闭连接时已传递的内容确定（关闭连接不能用来确定请求体结束，因为这样以来服务器就无法返回响应了）。

对于 HTTP/1.0 兼容的应用程序，发送带有消息体的 HTTP/1.1 请求时必须包含一个合法的 Content-Length 头域，除非已经知道服务器是基于 HTTP/1.1 的。如果请求中有消息体但却未给出 Content-Length 头域，则在无法确定消息长度时服务器应响应 400（错误的请求）或若它坚持要接收到一个合法的 Content-Length 头域应响应 411（缺少长度信息）。

所有的 HTTP/1.1 应用程序必须能够接收经分块编码的数据，从而允许在无法预先确定消息长度时可广泛地使用该机制。

但消息也不能既有 Content-Length 头域，又进行了分块编码。若两者都指定，则应忽略前者。

当消息中使用了 Content-Length 头域时，其值应与消息体的字节数完全匹配。HTTP/1.1 客户端程序在接收并识别到不正确的长度信息时应通知用户。

9.3.4. 通用头域

通用头域既可用于请求，也可用于响应，但却不适用于传输的实体。这些头域只对传输的消息适用。

```
general-header = Cache-Control
               | Connection
               | Date
```

```
                | Pragma
                | Transfer-Encoding
                | Upgrade
                | Via
```

不被识别的头域认为是实体头域。

9.3.5. 请求消息

HTTP 请求消息的 BNF 表示如下：

```
Request       = Request-Line
                *( general-header
                 | request-header
                 | entity-header )
                CRLF
                [ message-body ]
```

- 请求行

上面的 Request-Line 表示请求行。请求行依次由请求方法、被请求资源的 URI 地址、使用的协议版本号构成，最后以回车换行结束。各部分之间用空格分开，除了最后的回车换行外，其它地方不能出现回车或换行。

```
Request-Line   = Method SP Request-URI SP HTTP-Version CRLF
```

- 请求方法

上面的 Method 表示客户端要求服务器对以 Request-URI 为地址的资源进行的操作。方法名是大小写敏感的。

```
Method         = "OPTIONS"
               | "GET"
               | "HEAD"
               | "POST"
               | "PUT"
               | "DELETE"
               | "TRACE"
               | extension-method

extension-method = token
```

适用于某资源的方法可以在 Allow 头域中指明。服务器使用状态码来通知客户某个方法是否适用。若请求方法为服务器所知但却是对应资源所不允许的，那么服务器应返回 405（方法不允许）响应；而若方法无法识别或服务器没有实现它时，则返回 501（没有实现）响应。一个服务器所知的请求方法可以被列在 Public 响应头域中被客户程序获取。

GET 和 HEAD 方法是所有服务器都应该支持的。所有其它方法则是可选的。

- 请求 URI 地址

上面的 Request-URI 是统一资源标识符的形式，用来确定请求针对的资源。

```
Request-URI    = "*" | absoluteURI | abs_path
```

其中的三个选择依赖于具体的请求。星号表示请求不是针对任何特定的资源的，而是针对服务器本身的，只有当请求方法不是必须指定资源时才可以使用。例如：

```
OPTIONS * HTTP/1.1
```

当请求要经过代理时使用上面的第二种方式，即绝对地址方式。代理服务器将前传该请求，并从缓存中查找对应的资源，并返回之。注意代理有可能再把请求传给另一个代理，或者直接送给 Request-URI 中指明的目的服务器。为了避免请求的死循环，代理服务器必须能识别它本身的所有名字，包括别名及数字的 IP 地址。如下是一个请求行的例子：

```
GET http://www.w3.org/pub/WWW/TheProject.html HTTP/1.1
```

为了将来的 HTTP 版本中所有请求都使用绝对的请求地址，要求所有的 HTTP/1.1 服务器必须能接收带绝对地址的请求，即使现在的 HTTP/1.1 客户仅在对代理的请求中使用该形式。

Request-URI 地址的最常见形式是用于确定原始服务器或网关上的资源。这种情况下，URI 地址的绝对路径作为 Request-URI，而 URI 的网络地址（机器地址）则必须放在 Host 头域中传输。例如，直接访问原始服务器“www.w3.org”上资源的客户应建立与该服务器主机 80 端口之间的 TCP 连接，并发送：

```
GET /pub/WWW/TheProject.html HTTP/1.1
Host: www.w3.org
```

然后自然还有请求的其它部分。注意绝对路径不能为空；若原来的 URI 地址中没有给出，则应补上“/”(服务器的根)。

若代理服务器接收到一个 Request-URI 中没有路径信息的请求，且请求方法能支持星号方式，则请求链路上的最后一个代理应将星号作为最终的请求地址，然后再前传。例如请求：

```
OPTIONS http://www.ics.uci.edu:8001 HTTP/1.1
```

在代理服务器与主机“www.ics.uci.edu”的 8001 端口连接后被转换成如下形式：

```
OPTIONS * HTTP/1.1
Host: www.ics.uci.edu:8001
```

Request-URI 地址有时是经过一定编码处理的，具体请参阅本章第 2 节。原始服务器必须能解析该地址，并对非法的请求地址给出适当的响应。

代理服务器在前传请求时，除了上面所提到的用星号替换空的绝对路径外，不能对请求的请求地址部分进行任何的更改，无论代理内部是如何实现的。

注意：这里的“不准更改”的原则禁止代理更改请求的本意，因为有时原始服务器会用非保留（non-reserved）的 URL 字符来表示一个实现约定的含义。

- 请求资源

HTTP/1.1 原始服务器应意识到 Internet 请求指明的确切资源是由 Request-URI 和 Host 头域共同决定的。

某些不允许到来的请求中使用与己不同的主机地址的服务器会忽略掉请求的 Host 头域。而某些原始服务器接收的请求中使用不同的主机地址会对应不同的资源，其中的每个主机有时被称为虚拟主机（virtual hosts）或虚主机名（vanity hostnames）。这时，要使用如下的规则来确定被请求的资源：

1. 若 Request-URI 是一个绝对地址，则主机地址是其中的一部分。这时使用的任何 Host 头域都应该被忽略掉。
2. 若 Request-URI 不是一个绝对地址，且请求中包含一个 Host 头域，则主机由该头域的值来确定。
3. 若由上面两步确定的主机地址不是服务器上的合法主机地址，则服务器应返回 400（错误的请求）响应消息。

而对于接收到未含 Host 头域的 HTTP/1.0 请求的服务器则应使用启发式规则(heuristics)来确定究竟是请求哪个资源。读者可以联系 Apache 服务器的虚拟主机模式下资源的确定过程来理解此节。

- **请求头域**

客户使用请求头域来传递关于请求和客户的一些额外信息给服务器。这些头域作为请求的修饰符，语义与编程语言中方法调用中的参数类似。

```
request-header = Accept
               | Accept-Charset
               | Accept-Encoding
               | Accept-Language
               | Authorization
               | From
               | Host
               | If-Modified-Since
               | If-Match
               | If-None-Match
               | If-Range
               | If-Unmodified-Since
               | Max-Forwards
               | Proxy-Authorization
               | Range
               | Referer
               | User-Agent
```

请求头域只有在协议版本提高时才可以进行扩展，但有时会使用一些公认的新的或实验性的头域。不能识别的头域将被认为时实体头域。

9.3.6. 应答消息

服务器在接收和解析请求消息后，要返回响应消息。

```
Response     = Status-Line
               *( general-header
                | response-header
                | entity-header )
               CRLF
               [ message-body ]
```

- **状态行**

响应消息的第一行是状态行 Status-Line**。**

```
Status-Line = HTTP-Version SP Status-Code SP Reason-Phrase CRLF
```

其中首先是协议的版本号，然后是一个数字形式的状态码及对应的文本形式的原因短语。每一部分均用空格隔开，且除了最后有回车换行外，不允许再出现其它的回车换行。

- **状态码和原因短语**

响应状态码 Status-Code **是一个** 3 **位数整数；原因短语** Reason-Phrase **是对状态代码的简短文本陈述。状态码供协议自动机（指实现协议的程序）使用，而原因短语是可读的为人类用户使用的。我们有时可以在浏览器软件的服务器返回状态显示中看到状态代码，但这并非是对客户端程序的强制性要求。**

状态代码的第一个数字定义了应答的类别，其余的两位数字只起编号作用。一共有 5 **个类别：**

1xx：Informational-**提示，收到请求，继续**...；

2xx：Success-**请求成功，操作指令被成功地收到、理解和接受；**

3xx：Redirection-**请求未完成，为了完成请求，必须采取进一步的操作；**

4xx：Client Error-**客户错，请求语法错或无法完成；**

5xx：Server Error-**服务器错，服务器无法完成有效的请求。**

如下列出了 HTTP/1.1 **协议使用的所有的响应状态码，其中给出的原因短语仅作为建议而非强制性的，在具体的客户端实现中完全可以使用自己的表达。**

```
Status-Code    = "100"   ; Continue
               | "101"   ; Switching Protocols
               | "200"   ; OK
               | "201"   ; Created
               | "202"   ; Accepted
               | "203"   ; Non-Authoritative Information
               | "204"   ; No Content
               | "205"   ; Reset Content
               | "206"   ; Partial Content
               | "300"   ; Multiple Choices
               | "301"   ; Moved Permanently
               | "302"   ; Moved Temporarily
               | "303"   ; See Other
               | "304"   ; Not Modified
               | "305"   ; Use Proxy
               | "400"   ; Bad Request
               | "401"   ; Unauthorized
               | "402"   ; Payment Required
               | "403"   ; Forbidden
               | "404"   ; Not Found
               | "405"   ; Method Not Allowed
               | "406"   ; Not Acceptable
               | "407"   ; Proxy Authentication Required
               | "408"   ; Request Time-out
               | "409"   ; Conflict
               | "410"   ; Gone
               | "411"   ; Length Required
               | "412"   ; Precondition Failed
               | "413"   ; Request Entity Too Large
               | "414"   ; Request-URI Too Large
               | "415"   ; Unsupported Media Type
               | "500"   ; Internal Server Error
               | "501"   ; Not Implemented
               | "502"   ; Bad Gateway
               | "503"   ; Service Unavailable
               | "504"   ; Gateway Time-out
               | "505"   ; HTTP Version not supported
```

```
                | extension-code

extension-code = 3DIGIT

Reason-Phrase  = *<TEXT, excluding CR, LF>
```

HTTP 协议中的响应状态码是可扩展的。HTTP 应用程序并非必须支持上述所有的状态码，此仅是最好的期望。但应用程序必须能理解任何状态码所属类别的含义，该含义通过码的首位数字可以得到。任何不能识别的状态码应该当作同一类别的 x00 来看待，唯一的不同之处在于不能识别的状态码不能缓冲。客户收到一个不能识别的状态码 431，则它应该认为自己收到了 400 响应，即在请求中存在问题。这种情况下，客户端程序应该将响应中的实体呈现给用户，因为通常其中都含有用户可读的一些信息，该信息可解释出现的异常。

- 响应头域

服务器可以通过设置响应头域来传递额外的无法放在响应行中的信息给客户。这些头域可以是关于服务器的信息，也可以是关于请求针对的资源的。

```
response-header = Age
                | Location
                | Proxy-Authenticate
                | Public
                | Retry-After
                | Server
                | Vary
                | Warning
                | WWW-Authenticate
```

请求头域只有在协议版本提高时才可以进行扩展，但有时会使用一些公认的新的或实验性的头域。不能识别的头域将被认为是实体头域。

第4节　实体

除非请求方法和响应状态码禁止，请求和响应消息中可以带有实体。实体由若干实体头域和一个体（内容）构成，有些响应中可能只包含实体头域。

9.4.1. 实体头域

实体头域定义了关于实体内容的元信息，若没有体，头域则是针对请求指定的资源的。

```
entity-header  = Allow
               | Content-Base
               | Content-Encoding
               | Content-Language
               | Content-Length
               | Content-Location
               | Content-MD5
               | Content-Range
               | Content-Type
```

```
                    | ETag
                    | Expires
                    | Last-Modified
                    | extension-header

        extension-header = message-header
```

实体头域可以扩展，而不会影响协议。但扩展的实体头域未必为接收方所知，所以无法识别的头域应该被接收者忽略掉，对代理服务器来说则只管前传即可。

- **实体内容**

HTTP **请求和应答中的实体内容是按照实体头域指示的方式进行编码的。**

```
        entity-body    = *OCTET
```

只有在含有消息体的消息中才会存在实体内容。后者是通过对消息体按 Transfer-Encoding **头域的指示进行解码得到的。**

- **类型**

消息中含有的实体内容可能各种类型，类型信息可以通过 Content-Type **和** Content-Encoding **头域来获得。它们构成一个有层次的编码模型：**

```
        entity-body := Content-Encoding( Content-Type( data ) )
```

Content-Type **头域是指数据的媒体类型，而** Content-Encoding **头域则指示对该数据又进行了何种编码，通常是某种数据压缩方式。**

包含实体内容的 HTTP/1.1 **消息应包含一个** Content-Type **头域来定义实体内容的媒体类型。当且仅当** Content-Type **头域不存在时，接收方才可以试着通过数据内容和** URL **地址的文件扩展名来猜测媒体类型。若仍无法确定数据的类型，接收芳应将其当作“**application/octet-stream**”类型。**

- **长度**

实体内容的长度就是消息体被按传输编码解码后的长度。

第5节 HTTP 连接

9.5.1. 持续连接

在 HTTP/1.1 **以前，嵌入网页的图形以及相关数据的使用经常要求客户程序在一段较短的时间内向同一个服务器发出多个请求，每取回一个** URL **地址指示的资源都需要重新建立一个独立的** TCP **连接，从而增加了** HTTP **服务器一端的开销，并且可能导致网络阻塞。**

HTTP/1.1 **则提供了持续连接功能，持续** HTTP **连接有如下好处：**

1. **由于减少了** TCP **连接建立和关闭的次数，从而节约了** CPU **时间和** TCP **协议控制块消耗的内存；**

2. **在一次** TCP **连接之上就可以实现** HTTP **请求和应答的流水线作业，这意味着客户程序可以不必等待应答就可以连续发出多个请求，减少了系统延迟，提高了单个** TCP **连接的使用效率。**

3. **减少了** TCP **连接建立时所发送的报文数，并给** TCP **控制进程足够的时间以确定网络的阻塞状态，从而减少整个网络阻塞的发生次数；**

4. **没有关闭** TCP **连接的负担就可以报告差错，从而促进** HTTP **平稳地发展。未来版本**

的 HTTP 客户可能为了优化而采用新的协议特性，如果这些客户和旧版本的服务器通信并收到差错报告，由于有了持续性连接，就可以在不关闭连接的情况下，采用旧的语义和语法规范重试。

协商

关于持续连接，HTTP/1.1 与先前版本的 HTTP 的最大不同在于，它将持续连接作为建立的 HTTP 连接的缺省特性，即除非特别声明，客户假设服务器将维持一个持续连接。

持续连接还提供给客户和服务器使用 Connect 头域来通知关闭当前 TCP 连接的机制。一旦指示关闭连接的 Connect 头域（域值中含有一个 close 标记）发出，则客户将不能再发送任何请求。

一个遵循 HTTP/1.1 的服务器缺省认为 HTTP/1.1 客户将维持一条持续连接，除非请求中带有指示关闭连接的 Connect 头域。若服务器想在发送完响应后立即关闭连接，它应该应答客户一个带标识字“ close ”的 Connect 头域。

一个 HTTP/1.1 客户应认为建立的连接是持续性的，且根据来自服务器的响应中是否有带 close 标记的 Connect 头域来决定是否保持网络连接。在客户程序不打算再维持连接以发送更多请求时，它应该发送一个带 close 标记的 Connect 头域给服务器。当然不是单独发送该 Connect 头域，而是将其附到最后一次请求中通知服务器。

当无论是客户还是服务器发送了带 close 标记的 Connect 头域后，连接都将被关闭，而对应的请求也将是最后一次请求。

而基于比 1.1 版本低的 HTTP 协议的客户和服务器都不应假定持续连接的存在，除非其被显式指示。

流水工作

支持持续连接的客户可以以流水线方式发送请求（即连续提出多个请求，而不是等上一个应答返回后再发送后一个请求）。在这种情况下，服务器必须按接收请求的次序给出应答。

以持续连接和流水方式工作的客户在第一个流水请求失败后应重新去建立连接。如果真遇到上述情况时，客户程序在确定连接是持续的之前，不能进行流水工作。而当服务器在没有完全返回所有应答之前关闭了连接时，客户也应重新发送未响应的请求。

代理服务器的使用

代理服务器中持续连接的实现对这一功能的使用影响重大。

代理必须分别向客户和原始服务器（或其它代理服务器）指示持续连接。每个持续连接都只适用于一端连接，而非针对整个链路。

另外，代理服务器不能与 HTTP/1.0 客户建立持续连接。

有关实现上的考虑

服务器通常设置一些超时值，用于撤销那些超时的不活动连接。对代理服务器而言，超

时值一般要定得大一些，因为客户程序有可能同时向同一个服务器发出大量的请求消息。不管客户端还是服务器，如果想利用超时功能的话，都应该逐渐地关闭传输连接。客户程序和服务器都应该在接收到对方的传输关闭请求之后才作出适当的回应。

客户程序和服务器可以在任何时刻关闭传输连接。例如，在客户刚发送完一个新请求的同时，服务器决定关闭该“空闲”的连接。从服务器的角度来看，连接正被关闭，而在客户程序看来，则是请求在继续。

这就客户程序、服务器和代理都必须能从异步的关闭事件中恢复。只要请求操作是幂等的（N＞0 个相同请求的效果和单个同样请求的效果相同），客户软件就应该在没有用户干涉的情况下自动重新打开传输层连接，重发被取消的请求。除此之外的其他请求操作（非幂等类型），都不应该自动重试，但可以人工干预重试。然而，如果第二次请求仍然失败，则不应该再试。

服务器应该在每个连接上都至少响应一次请求。服务器不应该在发送响应的过程中关闭连接，除非已经察觉网络或客户方出现了问题。

使用持续连接的客户应该限制与特定服务器之间并存的连接的数目。一个单用户客户端程序与任何一个服务器应该最多有 2 个连接。而一个代理服务器与另一个服务器之间最多不应超过 2*N 个连接，这里的 N 指并存的活动用户数目。这样一些原则主要是为了提高 HTTP 响应速度，避免网络拥塞。

9.5.2. 有关消息传输的要求

一般的要求有：

- HTTP/1.1 的服务器应该保持持续连接而且利用 TCP 的流量控制来解决暂时的过载问题，而不是简单地解除连接以等待客户重试。
- HTTP/1.1（或以上）客户如果要发送消息实体内容，应该在传输请求消息的同时检查网络连接的情况，如果发现了错误状态，应该立即停止发送。
- HTTP/1.1（或以上）客户必须准备好接收跟在普通消息之后的一个 100 类（继续）状态码。
- HTTP/1.1（或以上）服务器接收到来自 HTTP/1.0（或以下）客户的请求时，不能发送 100（继续）响应；而是应该或者等待请求正常完成（从而避免一个请求被意外中断）或是尽可能早地关闭连接。

在接收到来自 HTTP/1.1（或以上）客户的涉及上述情况的请求时，一个 HTTP/1.1（或以上）服务器必须或者响应 100（继续）状态码并继续读取输入流，或者响应一个出错状态码。如果服务器在应答中指示出错，则它可以立即关闭传输层（TCP）连接，也可以继续读取并扔掉请求的其余部分，而绝对不能去执行请求方法。

客户端程序至少应记下最近访问过的服务器的版本号；如果 HTTP/1.1 客户在接收到响应状态码之前曾收到过 HTTP/1.1（或以上）版本的应答，则当它遇到连接被对方关闭时，应在不需用户参与的情况下重试该请求，只要该请求方法是幂等的；而其它方法则不能自动重试，尽管可能提供一个用户界面给用户来做是否重试的选择。如果客户重新发送了请求，则它必须首先发送请求头域，然后必须等待服务器的应答，可能是 100（继续）响应，也可能是出错响应，若是前者，则客户应继续发送。

如果 HTTP/1.1 客户从来没有收到过来自服务器的 HTTP/1.1（或以上）版本的应答，则它应该假设服务器实现的 HTTP 协议是 1.0 甚至更早版本，且它不会使用 100（继续）响应。若在这种情况下，客户在收到任何状态码前遇到连接被关闭，则它应该重发请求。而在重发时，应使用如下的“二进制指数后退”算法来确保获取一个可靠的响应：

- 初始化与服务器的新连接；
- 传输请求头域；
- 将估计的应答返回延迟时间（比如参考建立连接所花时间）赋给变量 R，若不好估计，可以取一定值，比方说 5 秒钟；
- 计算 T=R*（2N），这里的 N 是前面重试连接的次数；
- 一直等待，直到有出错响应从服务器返回，或 T 秒钟间隔已过；
- 若没有收到出错响应，则在 T 秒钟后发送请求体；
- 若客户发现连接被关闭，则重新从第 1 步开始直到请求被接收，最终客户可能会收到来自服务器的出错应答，或者客户没有耐心再试则退出发送过程。

无论服务器的版本如何，客户收到一个出错响应后，绝对不能继续再发送未完请求且要关闭连接。

若一个 HTTP/1.1（或以上）客户遇到连接被关闭，而这时它仅收到 100（继续）状态码，则它应该重试请求，而不必等 100 响应完全接收到（但如果这样便于实现的话，也可如此）。

第6节　HTTP 方法

客户通过在请求中使用不同的 HTTP 请求方法（操作），提请服务器执行不同的操作。

9.6.1. 安全和幂等方法

- 安全方法（Safe Methods）

协议的实现程序必须意识到它是代表实际用户的，应该让用户知道他进行的操作可能给自己或别人带来什么样的后果。

一般认为，GET 和 HEAD 方法除了引起检索操作外，不应造成其它任何后果。这些方法被认为是安全的。而执行其它的如 POST、PUT 和 DELETE 等方法时则应该让用户知道正在执行“不安全的”动作。

但话又说回来，执行 GET 等操作而不对服务器造成任何副作用是不可能的；实际上，某些动态资源还把此看作一种特征。

- 幂等方法（Idempotent Methods）

我们有时关注某个请求方法是否是幂等的，即 N（N > 0）个相同请求的副作用与单个请求的副作用相同。GET、HEAD、PUT 和 DELETE 方法均是幂等的。

HTTP1.1 中支持的操作方法有：OPTIONS、GET、HEAD、POST、PUT、DELETE 和 TRACE 等。

9.6.2. OPTIONS

OPTIONS 操作说明了由 Request-URI 地址所标识的 HTTP 请求/应答链路上有关通信功能选项。客户使用该操作可以在不对实体实际操作或传送实体的情况下获得某个实体的操作要求、功能选项，或服务器的能力。

对 OPTIONS 请求的响应中不能包含任何实体信息，而只能包含通信的功能选项，除非该响应是出错信息（例如可以包含 Allow 头域却不能带有 Content-Type 头域）。而且对 OPTIONS 请求的应答是不能被缓存的。

若请求中的 URI 是一个星号，则 OPTIONS 请求是针对整个服务器的。一个 200 响应应包含必要的头域来指示服务器实现的一些可选的特征（如 Public 头域）及一些功能扩展。通过代理服务器的不带任何路径信息的 OPTIONS 请求，将被自动转换成“ OPTIONS * ”请求。

若请求 URI 不是一个星号，则 OPTIONS 请求是仅针对指定的资源的。一个 200 响应应包含必要的头域来指示服务器实现的可对该资源应用的一些可选的特征（如 Public 头域）及一些功能扩展。若 OPTIONS 请求是通过代理提出的，则代理服务器必须编辑响应，删去针对代理服务器的但其自身不支持的选项。

9.6.3. GET

GET 操作表示以实体的形式取回由 Request-URI 地址所标识的任何信息。如果 Request-URI 指向一个数据处理进程，那么该进程运行的结果将作为应答消息包中的实体而返回。

如果请求消息带有 If-Modified-Since，If-Unmodified-Since，If-Match，If-None-Match 或 If-Range 等头域，则 GET 就变成了“条件 GET”。条件 GET 请求要求实体只有在满足指定条件时才传输，而不满足时，一般表示本地缓存中存有有效备份，而直接使用，避免重复的数据传输。

如果请求消息中含有 Range 头域，这相当于一个“部分 GET”。部分 GET 只要求传送实体的一部分内容，这样就不必传送客户已拥有的数据，从而减少了网络使用。

9.6.4. HEAD

HEAD 操作和 GET 语义基本相同，但是 HEAD 规定应答消息包中不能含有消息体。使用该操作可以在不传输实体内容的情况下获取实体的元信息，因此它经常被用来测试超媒体链的有效性、可访问性和最近的修改状态。

如果对 HEAD 请求的响应信息可以被用来更新缓存中对应资源实体数据，则该响应可以认为是可缓冲的。若新的头域值与已有备份不同（如 Content-Length、Content-MD5、ETag 或 Last-Modified 等头域），则应认为该备份已过时。

9.6.5. POST

POST 操作用于请求目的服务器接收请求消息包中的实体，作为由 Request_URI 地址所标识资源的一个新的从属。有了 POST，可以在实际应用中完成如下的功能：

- 已存在资源的注解；
- 向 BBS、新闻组或邮件列表张贴消息；
- 向数据处理进程提供一组数据，比如表格（form）提交的结果；
- 实现数据库的添加操作。

POST 操作的实际功能由服务器决定，这通常取决于 Request-URI。被 POST 的实体从属于某个 URI 就好象文件从属于包含它的一个目录，新闻稿从属于某个目的新闻组，或者记录属于某个数据库一样。

POST 请求要求的动作可能不一定带来可用 URI 定位的资源的产生。这种情况下，或着 200（OK）或者 204（No Content）都是适当的响应状态码，这依赖于响应中是否包含一个

描述结果的实体。

若请求使某资源在原始服务器上创建，则响应应该是 201（Created）并包含一个描述请求状态指向新资源的实体和一个 Location 头域。

对 POST 请求的响应不是可缓冲的，除非响应中包含了适当的 Cache-Control 或 Expires 头域。但是，303（See Other）响应仍可用来引导用户程序去何处查找可缓冲的资源。

9.6.6. PUT

PUT 操作请求把携带的实体存放于 Request-URI 处，请注意它和 POST 的区别。

如果 Request-URI 指向一个已经存在的资源，请求消息包中携带的实体将作为原始服务器上原资源的一个更新版本；如果 Request-URI 所指的资源不存在，而且该 URI 能被请求用户进程定义为一个新资源，那么原始服务器将用该 URI 创建一个新资源。

若新资源创建，服务器必须发 201（Created）响应通知用户程序；若已有资源被更新，则应发 200（OK）或者 204（No Content）响应来指示请求的成功完成；若资源无法创建或更新，则应使用适当的出错响应来反映之。实体的接收者不能忽略它无法识别或没有实现的任何 Content-*形式的头域，而必须返回 501（Not Implemented）响应。

若请求经过了缓存且请求 URI 对应一个或多个当前被缓冲的实体，则它们应被认为已过时。对 PUT 请求的响应不是可缓冲的。

POST 和 PUT 操作最根本的差别反映在 Request-URI 的含义中。POST 的 URI 标识了将对被携带实体进行处理的某个资源，该资源可能是一个数据处理进程、一个多协议网关或者一个接收注解信息的独立实体。与此相反，PUT 请求中的 URI 标识了该请求所携带的实体，只有用户软件知道 URI 的确切含义，而服务器绝对不能对其它资源实施被请求的操作。若服务器希望请求应用于其它 URI 地址，则它必须发送 301（Moved Permanently）响应，由用户程序来决定是否进行请求的重定向。

一个资源可能被多个不同的 URI 所标识，因此一个普通的 URI 请求可能导致原始服务器定义多个其它的新 URI。

9.6.7. DELETE

DELETE 操作请求原始服务器删除由 Request-URI 标识的资源。注意，该操作可能受到原始服务器一端的人为干预（或其它因素），所以即使从服务器返回的状态代码表明操作成功完成，客户也不能保证操作被执行。不过一般情况下服务器不应该发出成功的 DELETE 确认信息，除非发出应答消息的同时，服务器的确要删除该资源或把该资源移到不能被客户访问到的地方。

若响应中包含了描述状态的实体，则应使用状态码 200（OK）；若对应的动作还未执行，则应使用状态码 202（Accepted）；若响应要表达成功但未包含任何实体，则应使用状态码 204（No Content）。

若请求经过了缓存且请求 URI 对应一个或多个当前被缓冲的实体，则它们应被认为已过时。对 DELETE 请求的响应不是可缓冲的。

9.6.8. TRACE

TRACE 操作用于引发一次请求消息包的远程应用层自环测试。如果成功，请求消息的最后一个接收者应该把整个请求消息包作为应答消息的实体内容部分发还给客户端，其 Content-Type 为“message/http”。对 TRACE 的应答不能被缓存。

TRACE 使客户可以了解到请求/应答链路另一端数据接收的情况，并把接收到的数据作为测试或诊断信息回送。头域 Via 的值可用来跟踪请求链路，而 Max-Forwards 头域允许客户限制请求链的长度，从而可以检测出一组循环转发该请求消息包的客户代理来，从而有助于发现网络中存在的使消息空转的闭环。

第7节　状态码

下面对各个状态码进行详细的描述，包括它所针对的请求方法及响应中使用的各种元信息等。

9.7.1. 提示类状态码——1xx

此类状态码指示临时性响应，其中只包含一个状态行和可选的头域，以一空行结束。既然 HTTP/1.0 中没有定义 1xx 形式的状态码，所以服务器不能发送 1xx 响应给 HTTP/1.0 客户程序。

- 100 Continue

通知客户应继续发送请求。此临时性响应用来通知客户请求的初始部分已被接收并未被拒绝。客户应该继续发送请求的其余部分，或在请求已完成时，忽略掉此响应。服务器在接收到完整的请求后，必须发送一个最终的响应。

- 101 Switching Protocols

表示服务器理解并将完成客户的请求，但是用 Upgrade 消息头域来指示当前连接上使用的协议要进行更换。要更换的协议在 Upgrade 头域中给出。

只有在对通信有益时才应该进行协议的切换。例如，从旧版本的 HTTP 协议更换到新版本上，或更换到实时同步协议上将有助于提高传输的速度和质量。

9.7.2. 成功类状态码——2xx

- 200 OK

表示请求成功。响应中返回的信息依赖于请求方法，例如：

GET：响应中包含对应被请求资源的实体；

HEAD：响应中包含与被请求资源对应的实体头域，但没有任何消息实体内容；

POST：返回描述或带有请求动作结果的实体；

TRACE：返回作为响应实体的被最终服务器接收的请求消息。

- 201 Created

表示请求已成功执行，新资源已被创建。该资源可以用响应中包含的 URI 地址来指引，通常都是使用 Location 头域。原始服务器在返回 201 响应前，必须已创建资源；若动作无法

快速完成，则应返回 202 响应。

- 202 Accepted

表示请求已被接收，但处理尚未完成。最终请求或成功或失败，本协议中没有定义其它的机制来表达此异步操作的最终完成。

此状态码是专门用于非履行情况（non-committal）的。它的意图在于使服务器可以接收一个针对其它进程的请求，同时不必让与客户端的连接一直持续至处理的结束。响应中的实体应指示请求的当前状态，或者包含一个对状态监视器的指针，或者包含请求动作最终完成的一个时间估计。例如从客户端发送一个特定的 HTTP 请求给服务器，要求服务器对日志信息进行归档，这时服务器就可返回 202 Accepted 响应，表示请求已接收，并同时启动一个日志归档进程进行相应的操作。该操作是异步的，即 202 响应并非等到归档工作完成才响应给客户端。

- 203 Non-Authoritative Information

表示实体头域中返回的元信息并不是源于原始服务器，而是来自一个本地或第三方的拷贝。提供的集合或是原始版本的子集，或是超集。例如，包含关于资源的本地注解信息会最终得到一个原始服务器所知的元信息的超集。任何时候都不必须使用此状态码，而且只有在响应也可用 200 的时候，才可以用此状态码。

- 204 No Content

表示服务器已经执行了请求的动作，但没有进一步的新信息返回。若客户是一个用户程序，则它不应改变引发请求发送的文档视图。响应中可以包含实体头域形式的元信息，该信息应该是应用于程序正打开的文档的。204 响应不能包含消息体。

- 205 Reset Content

表示服务器已经执行了请求的动作，用户程序应该重新设置引起请求发送的文档视图。此类响应主要用于在用户输入一定内容后发出请求，而后将表格清空以便于再次输入的情况。此类响应中不能包含实体。

- 206 Partial Content

表示服务器已执行了部分 GET 请求。对应的请求中一定含有一个 Range 头域来指示期望的区间。响应必须或者包含一个 Content-Range 头域来指示响应中包含的内容区间，后者包含类型为 multipart/byteranges 的多部内容，每一部分包含一个 Content-Range 头域。若没有使用 multipart/byteranges 类型，则响应中的 Content-Length 头域的值必须与响应消息体中实际的字节数匹配。

9.7.3. 重定向类状态码——3xx

此类状态码指示用户程序必须采取进一步的动作来完成请求。当且仅当再次请求是 GET 或 HEAD 方法时，不必有实际用户的参与。用户程序自动进行的请求的重定向不能超过 5 次，因为有时重定向存在无限循环的情况。

- 300 Multiple Choices

请求的资源对应一组表示，其中的每一个都有自己的位置。响应中含有用户驱动的协商信息使得用户可以选择一个最佳表示，并将请求重定向到对应的位置。

除非请求是 HEAD 方法，否则响应中应该包含一个实体，实体中应给出资源特征和位置的列表供用户选择。实体格式由 Content-Type 头域定义的媒体类型给出。用户程序可以基于此格式和自身能力，选择最佳的表示。

若服务器在多种表示中有缺省的最佳选择，则它应该在 Location 头域中包含对应的 URL 地址；用户程序可以使用该头域值来进行自动的重定向。

若非特别说明，此类响应是可缓冲的。

- 301 Moved Permanently

被请求的资源已移动，与一新的永久 URI 对应，将来引用该资源应使用响应中返回的 URI 地址。若非特别说明，此类响应是可缓冲的。

若新的 URI 是一个位置指示，则它的 URL 地址应该在响应的 Location 头域中给出。除非请求方法是 HEAD，否则响应的实体部分应包含一个超文本形式的链接指向新的 URI。

若 301 响应对应的请求不是 GET 或 HEAD 方法，则用户程序不能自动进行重定向，除非用户已肯定其做法。因为这可能会改变请求发出的条件。

注意：当接收到 301 状态码后自动重定向 POST 请求时，某些现存的 HTTP/1.0 用户程序会错误地处理成 GET 请求。

- 302 Moved Temporarily

被请求的资源临时性地移动到一个不同的 URI 下。只有当响应中给出了 Cache-Control 或 Expires 头域时，响应才是可缓冲的。

若新的 URI 是一个位置指示，则它的 URL 地址应该在响应的 Location 头域中给出。除非请求方法是 HEAD，否则响应的实体部分应包含一个超文本形式的链接指向新的 URI。

若 302 响应对应的请求不是 GET 或 HEAD 方法，则用户程序不能自动进行重定向，除非用户已肯定其做法。因为这可能会改变请求发出的条件。

注意：当接收到 302 状态码后自动重定向 POST 请求时，某些现存的 HTTP/1.0 用户程序会错误地处理成 GET 请求。

- 303 See Other

表示被请求的资源在不同的 URI 地址下，应使用 GET 方法对该地址进行查询。此状态码主要用于支持 POST 方式的执行脚本输出使用户程序重定向到一个被选择的资源。303 响应不能进行缓冲，但对第二个请求（重定向请求）的响应则可能是可缓冲的。

若新的 URI 是一个位置指示，则它的 URL 地址应该在响应的 Location 头域中给出。除非请求方法是 HEAD，否则响应的实体部分应包含一个超文本形式的链接指向新的 URI。

- 304 Not Modified

若客户提出的是条件 GET 请求且访问是允许的，但文档未被修改，则服务器应返回此状态码。响应中不能含有消息体。

响应必须包含下列头域：

- Date；
- ETag 或/和 Content-Location，若在对相同请求的 200 响应中会发送这些头域的话；
- Expires、Cache-Control 和/或 Vary，若域值和先前针对同一变体的响应不同的话。

若 304 响应指示实体当前未被缓冲，则缓存应丢掉响应，并重新发出非条件的请求。

若缓存使用 304 响应来更新缓存项，则它应该更新响应中给出的所有头域的值。

- 305 Use Proxy

表示被请求的资源必须通过由 Location 头域给出的代理服务器来访问。Location 头域给出了代理的 URL 地址。此响应的接收者应再通过指定的代理重新提出请求。

9.7.4. 客户端出错类状态码——4xx

形如 4xx 的状态码用于客户端出错的指示。除非对应 HEAD 请求，服务器应该在响应中包含一个实体，来解释错误情形及此情形是暂时的还是永久性的。这些状态码对任何请求

方法均适用。用户程序应将响应中包含的带解释信息的实体显示给用户。

- 400 Bad Request

请求语法错，服务器无法理解。客户端程序不应再发相同的请求。

- 401 Unauthorized

请求需要进行用户身份认证。响应中必须包含一个 WWW-Authenticate 头域，其中包含一个针对被请求资源的提问。客户在附加上一个 Authorization 头域后可以重新发送先前的请求。若先前的请求中已经含有 Authorization 内容，则 401 响应表示认证未通过。若 401 响应中包含与先前响应相同的提问，即用户程序至少已经试图认证过一次，这时该程序应把响应实体内容呈现给用户，以便用户在看到相关的解释或诊断信息后能给出正确的认证回答。身份认证的细节请参阅第 7 章第 9 节。

- 402 Payment Required

保留使用。

- 403 Forbidden

服务器理解请求，但拒绝执行之。若请求方法不是 HEAD 且服务器希望公开请求不能执行的原因，则它应该在响应实体中描述原因。此状态码通常用于服务器不想告诉拒绝请求的原因或其它类型的响应不适用的情况。

- 404 Not Found

服务器没有找到与被请求 URI 匹配的资源。若服务器不想让客户知道此信息，则也可以用状态码 403 代替。若服务器知道是由于进行过一些内部的变更，使指定的 URI 不再有效且不能给出新地址建议时，应返回 410（已过时）状态码。

- 405 Method Not Allowed

表示请求方法对指定的资源不适用。响应中必须包含 Allow 头域给出可以使用的方法。

- 406 Not Acceptable

表示指定的资源无法产生客户指定可接收的响应实体。

除非请求是 HEAD 方法，响应中应该包含一个实体，其中带有一系列实体特征和位置信息，以便客户可以从中选择。实体的格式由 Content-Type 头域给出的媒体类型定义。

- 407 Proxy Authentication Required

此与 401 类似，但指示的是应首先通过代理的认证。代理服务器应返回一个 Proxy-Authenticate 头域，给出针对被访问资源的一个询问。客户可以继续提出带有适当 Proxy-Authorization 头域的请求。

- 408 Request Timeout

客户没有在服务器规定的等待时间内提出请求。客户可以随后重发原始请求。

- 409 Conflict

表示由于与资源的当前状态冲突而无法完成请求动作。仅当用户可能有解决冲突的能力且希望重发请求时才使用此状态码。响应体中应该包含足够的信息使用户能确定冲突的原因。此类响应在处理 PUT 请求的时候最常出现。

- 410 Gone

被请求的资源已不存在，且不知道进一步的地址信息。这里出现的问题应理解为永久性的。若服务器无法确定问题是否永久存在，则应该使用 404 作为响应状态码。此类响应是可缓冲的，除非特别声明。

本状态码主要为了通知接收方由于站点维护资源已不存在，远程的链接应该被删除。

- 411 Length Required

由于请求中没有定义 Content-Length 头域，所以服务器拒绝处理请求。客户在增加上 Content-Length 头域后可重发请求。

- 412 Precondition Failed

表示服务器判定一个或多个请求头域中给出的条件表达式为假。此状态码使客户可以针对资源的元信息设置一定条件，避免请求定位到一个非是所期望的资源上。

- 413 Request Entity Too Large

由于请求实体太大，服务器拒绝处理请求。服务器可以关闭连接阻止客户继续发送请求。

若问题是暂时性的，则服务器应在响应中设置一个 Retry-After 头域通知客户稍后可再试。

- 414 Request-URI Too Long

由于请求 URI 地址过长，服务器拒绝处理请求。这种情况一般仅在客户将 POST 请求转换成带有很长查询信息的 GET 请求时才发生。

- 415 Unsupported Media Type

表示服务器不支持请求实体格式。

9.7.5. 服务器端出错类状态码——5xx

形如 5xx 的状态码被服务器用来指示发觉本身出错或无法执行请求的情况。除非请求方法是 HEAD，服务器一般都应该在响应实体中给出问题出现的原因，及其是永久性的还是临时性的。用户程序应该向用户显示这些信息。这些状态码可用在对任何请求方法的响应中。

- 500 Internal Server Error

服务器遇到了意外情况无法完成请求的处理。

- 501 Not Implemented

服务器不支持请求所要求的功能。当服务器不能识别请求方法时应使用此响应。

- 502 Bad Gateway

表示网关或代理服务器接收到的上一级服务器返回的响应不合法。

- 503 Service Unavailable

由于临时过载或维护等原因，暂时无法处理请求。其含义是此问题是临时性的。如果可能，可以在响应的 Retry-After 头域中给出可重发请求的等待时间。若响应中没给出 Retry-After 头域，则客户应将其等同于一个 500 响应。

注意：提供此状态码并不意味着在服务器过载时必须使用它。某些服务器实际上仅通过拒绝进一步的网络连接来表示自己负载过重。

- 504 Gateway Timeout

表示网关或代理服务器没有等到从上一级服务器返回的响应。

- 505 HTTP Version Not Supported

表示服务器不支持或拒绝支持请求使用的 HTTP 协议版本。通过此状态码，服务器可表达自己不能或不愿使用与请求相同版本的协议来完成请求的处理。响应实体中应包含该版本不被支持的原因，及支持那些版本。

第8节　HTTP 访问身份认证

HTTP 提供了一种简单的“提问-回答”式身份鉴别方法。先由服务器向客户发出身份认证请求，再由客户提供确认信息。

HTTP 并不仅限于这种简单的身份鉴别方式，还可使用其它机制，包括传输层加密或消息包的封装。

详细的身份认证机制请参阅第 7 章第 9 节。

第9节　内容协商

大多数的 HTTP 应答消息包带有用户请求的信息，显然，最好的情况是提供满足用户要求的最佳实体。很多时候服务器或缓冲存储之间并没有统一的最佳标准，并非所有用户软件有能力处理所有的实体类型，因此，HTTP 提供了一种“内容协商”机制，能够使服务器在给定的应答有多个表示的情况下，挑选出其中最好的一种资源变体。

注意，内容协商并不是“格式协商”(Format Negotiation)，这是因为可选的表示可能属于同一种媒体类型，却具备了该类型的不同能力，比如不同的自然语言表示。

HTTP 定义了两种类型的内容协商，分别为服务器驱动（server-drivern）和用户软件驱动（agent-driven）的内容协商。这两类协商是互不影响的，所以可以独立或结合使用。

9.9.1. 服务器驱动的内容协商

若响应中给出的最佳表示的选择是服务器基于一定算法完成的，则称为服务器驱动内容协商。进行的选择依赖于存在的变体种类、请求消息特定头域的值及与请求相关的其它信息（如客户端的网络地址等）。

当选择算法很难向用户程序描述或服务器希望直接给出一个应答而避免过多的来回延迟时，服务器驱动是较好的内容协商方式。为了提高选择的准确度，用户程序可以发送一些头域（如 Accept、Accept-Language、Accept-Encoding 等）来描述自己的喜好。

服务器驱动内容协商方式同时有如下缺点：

- 服务器很难确定什么是给定用户的最佳选择，因为这需要对用户的能力和意图有充分的了解（例如，用户是想在屏幕上看还是要打印到纸上？）。
- 让客户每次请求都提供自身能力的信息将使效率降低（实际上，只有很少的情况下才需要进行选择），而且也是对用户私有信息的“粗暴践踏”。
- 这使服务器软件中生成响应的算法实现的复杂度提高。
- 它限制了缓存用一次响应应答多次用户请求的优势的发挥。

HTTP/1.1 原始服务器在发送基于服务器驱动内容协商的可缓冲响应时应包含 Vary 头域，用它来描述响应可能存在的多种表示的范围。

9.9.2. 用户软件驱动的内容协商

在用户软件驱动的内容协商方式下，最佳表示的选择权在用户程序一方。它在接收到服务器的初始应答后作出决定。该应答的头域和实体中给出了所有存在的变体的信息及它们各自的 URL 地址。对它们的选择可以自动完成，也可由用户通过生成的菜单手工选择完成。

当服务器无法确定客户端的能力且变体是常见的基于类型、语言或编码等方面时，最好使用用户软件驱动的内容协商。

这种方式的缺点则在于它需要额外的请求来获取最佳表示变体。只有当缓存起作用时这一问题才可忽略不计。

为支持用户软件驱动的内容协商，HTTP/1.1 中定义了 300（有多种选择）和 406（不是

可接受的）两种状态码。

9.9.3. 透明内容协商

透明内容协商是前两种内容协商方式的综合。当缓存中含有某一响应的多个变体，且它对各个变体的含义也都清楚时，缓存可以代替原始服务器来完成服务器驱动方式的内容协商。

透明内容协商的优势在于协商的负担被分担，且在缓存能作出选择的情况下也可避免用户软件驱动方式下的多次请求。

进行透明内容协商的 HTTP/1.1 缓存必须在响应中给出一个 Vary 头域描述表示变化的范围。

第10节 HTTP 头域

这一节给出了标准 HTTP/1.1 头域的语法和语义说明。

对实体头域来说，文章中的术语发送方（Sender）和接收方（Recipient）可能指客户或服务器任何一方，这取决于发送或接收实体的当前实际行为。

通用头域既可用于请求消息，也可用于应答消息，但是不能作用于被传输的实体。消息包中未被识别的头域可以认为是实体头域。

下面我们挑选一些常见的重要头域介绍给读者，侧重于设计原理和用途。每个头域说明中均给出了规范的语法结构，以方便用户对照。

9.10.1. Accept

Accept 请求头域用来指示客户端可以接收的响应中使用的媒体类型。

```
Accept         = "Accept" ":"
                 #( media-range [ accept-params ] )
media-range    = ( "*/*"
                 | ( type "/" "*" )
                 | ( type "/" subtype )
                 ) *( ";" parameter )
accept-params  = ";" "q" "=" qvalue *( accept-extension )
accept-extension = ";" token [ "=" ( token | quoted-string ) ]
```

其中的星号表示一定的可变范围，“ */* ”表示所有媒体类型，而“ type/* ”表示该类型的所有子类型。media-range 部分还可以带有针对特定媒体类型的参数。

每个 media-range 部分中的参数都以名为“ q ”的参数开头，用它来表示相对质量因子。例如：

```
Accept: audio/*; q=0.2, audio/basic
```

9.10.2. Accept-Charset

Accept-Charset 头域用来指示客户端可以接收的响应中使用的字符集。它使客户可以告

诉服务器其对字符集的支持能力。

```
Accept-Charset = "Accept-Charset" ":"
        1#( charset [ ";" "q" "=" qvalue ] )
```

一般认为 ISO-8859-1 字符集所有的用户程序都能支持。

若没有给出任何 Accept-Charset 头域，则认为客户可以接收基于任何字符集的数据。若服务器无法提供基于客户指定字符集的信息，则应返回 406（不是可接受的）出错响应。

9.10.3. Accept-Encoding

Accept-Encoding 头域与 Accept 头域类似,但它是用来限制在响应中使用的内容编码的。

```
Accept-Encoding  = "Accept-Encoding" ":"
                    #( content-coding )
```

例如：

```
Accept-Encoding: compress, gzip
```

若请求中没有给出本头域，则服务器可以认为客户可以接受任何内容编码。若服务器无法满足客户提出的此类要求，则应返回 406 响应。

若 Accept-Encoding 头域取空值，则表示任何编码都无法接受。

9.10.4. Accept-Language

Accept- Language 头域与 Accept 头域类似,但它是用来限制在响应中使用的自然语言的。

```
Accept-Language = "Accept-Language" ":"
            1#( language-range [ ";" "q" "=" qvalue ] )
language-range  = ( ( 1*8ALPHA *( "-" 1*8ALPHA ) ) | "*" )
```

language-range 参数用来描述用户对某种语言的喜好程度，缺省取值为 1。例如：

```
Accept-Language: da, en-gb;q=0.8, en;q=0.7
```

表示“我最喜欢丹麦语（Danish），但也能接受英国英语和其它种类的英语。”

9.10.5. Accept-Ranges

Accept-Ranges 响应头域指示服务器支持对资源的一部分提出请求：

```
Accept-Ranges     = "Accept-Ranges" ":" acceptable-ranges
acceptable-ranges = 1#range-unit | "none"
```

例如：

```
Accept-Ranges: bytes
```

但若其不支持，则应返回：

```
Accept-Ranges: none
```

来告诉客户不要发送部分资源的请求。

9.10.6. Age

Age 响应头域表达服务器响应已产生的时间。

```
Age = "Age" ":" age-value
age-value = delta-seconds
```

9.10.7. Allow

Allow 实体头域列出请求 URI 指示的资源所支持的操作方法。Allow 头域必须出现在 405（方法不允许）响应中。

```
Allow        = "Allow" ":" 1#method
```

例如：

```
Allow: GET, HEAD, PUT
```

9.10.8. Authorization

用户程序在接收到来自服务器的 401（没有授权）响应后，可以发送带有 Authorization 头域的请求来进行认证。Authorization 头域的值由用户程序针对请求资源域的认证信息构成。

```
Authorization  = "Authorization" ":" credentials
```

HTTP 协议的认证细节参阅第 7 章第 9 节。

9.10.9. Connection

Connection 通用头域用来定义与某次连接相关的某些信息。

```
Connection-header = "Connection" ":" 1#(connection-token)
connection-token  = token
```

HTTP/1.1 代理服务器必须对接收的消息的 Connection 头域进行解析，并将与其中包含的标识符同名的头域从消息中删除，然后再将消息转发。

HTTP/1.1 中还定义了“close”标识符用来在 Connection 头域指示在响应完成后连接将被关闭。如：

```
Connection: close
```

9.10.10. Content-Base

实体头域 Content-Base 用来为实体内部的相对 URL 地址指定基地址（Base URI）。URI 基地址说明了资源的起始（根）位置，其格式为：

Content-Base=“Content-Base” “ :” absoulteURI

如果 Content-Base 不存在，实体的 URI 基地址由 Content-Location（如果该 Content-LocationURI 是一个绝对 URI 地址）或者用来发起请求的 URI 决定。但是请注意：消息包中实体内容的 URI 基地址可能在实体内容内部被重新定义。

9.10.11. Content-Encoding

Content-Encoding 实体头域用来作为媒体类型的修饰说明。其值指示对实体内容进行何种编码，这样在接收端应进行何种解码。该头域主要用于指示对文档的压缩。

```
Content-Encoding  = "Content-Encoding" ":" 1#content-coding
```

例如：

```
Content-Encoding: gzip
```

9.10.12. Content-Language

Content-Language 实体头域用来描述被封装实体面向说何种自然语言的用户。

```
Content-Language  = "Content-Language" ":" 1#language-tag
```

例如：

```
Content-Language: da
```

表示实体内容面向说丹麦语的用户。

9.10.13. Content-Length

实体头域用八位串的十进制数字表示说明了发送给接收方的消息内容大小。格式为：

```
Content-Length=“Content-Length”“:”1*DIGIT
```

例如：

```
Content-Length:3495
```

这里的 DIGIT 表示从 0 到 9 的十进制数字。

9.10.14. Content-Location

Content-Location 头域可用来为消息包所携带的实体指定资源位置。假如某个资源有多个实体和它相连，而且这些实体都具有独立的位置可被分别地访问，那么服务器应该为这个资源的每个将被回送的变体提供一个 Content-Location。除此以外，服务器还应该为与应答实体相关的资源提供一个 Content-Location。格式为：

```
Content-Location=“Content-Location”“:” (absoluteURI|relativeURI)
```

9.10.15. Content-MD5

Content-MD5 实体头域是实体内容的 MD5 文摘，用来支持端到端的消息完整性检查。

```
Content-MD5   = "Content-MD5" ":" md5-digest
md5-digest   = <base64 of 128 bit MD5 digest as per RFC 1864>
```

MD5 文摘的计算是针对已经过 Content-Encoding 编码处理的实体内容，但是在应用 Transfer-Encoding 编码之前。

9.10.16. Content-Range

当服务器回送一个部分应答给客户时，服务器必须给出这个应答所覆盖的实体域范围，以及原实体内容的长度。Content-Range 头域和这个部分实体内容一起发送，表明该部分内容在整个实体内容中的插入位置，它同时也说明了整个实体内容的大小。格式为：

```
Content-Range = "Content-Range" ":" content-range-spec
content-range-spec     = byte-content-range-spec
byte-content-range-spec = bytes-unit SP first-byte-pos "-"
                        last-byte-pos "/" entity-length
entity-length          = 1*DIGIT
```

和 byte-range-specifier 值不同的是，上面的 byte-content-range-spec 最多只能指明一个域，而且必须包含域的第一个和最后一个字节的绝对偏移。

当一条 HTTP 消息包含单一内容区域时（例如，对单一内容区域请求的应答，或者是对多个相互重合且无空隙留下的内容区域的请求的应答），伴随着该内容就有一个 Content-Range 头域同时发送，而用 Content-Length 头域说明实际发送的字节数。例如：

```
HTTP/1.1 206 Partial content
Date: Wed, 15 Nov 1995 06:25:24 GMT
Last-modified: Wed, 15 Nov 1995 04:58:08 GMT
Content-Range: bytes 21010-47021/47022
Content-Length: 26012
Content-Type: image/gif
```

9.10.17. Content-Type

实体头域 Content-Type 说明发送给接收方的实体的媒体类型。格式为：

```
Content-Type   = "Content-Type" ":" media-type
```

例如：

```
Content-Type: text/html; charset=ISO-8859-4
```

9.10.18. Date

通用头域 Date 代表了消息生成的日期和时间。理论上讲，该时间应该代表了实体刚要生成前的那一时刻，而具体实现时，该时间可以取值为消息生成过程中的任何时刻而不影响它的语义。格式为：

```
Date  = "Date" ":" HTTP-date
```

例如：

```
Date: Tue, 15 Nov 1994 08:12:31 GMT
```

9.10.19. ETag

实体头域 ETag 定义了关联实体（Associate Entity）的实体标记，可用于比较来自同一

资源的不同实体。格式为：

```
Etag=“Etag”“:”entity-tag
```

例如：

```
ETag:“xyzzy”
ETag:W/“xyzzy”
ETag:“”
```

9.10.20. Expires

Expires 实体头域给出响应的过期时间。已过期的缓存内容，除非经过与原始服务器的联系重新验证有效，否则不应发送给客户作为响应。

```
Expires = "Expires" ":" HTTP-date
```

例如：

```
Expires: Thu, 01 Dec 1994 16:00:00 GMT
```

9.10.21. From

From 请求头域包含控制客户端程序的用户的 Email 地址。

```
From   = "From" ":" mailbox
```

例如：

```
From: webmaster@w3.org
```

该头域可以用于日志记录，确定不合法请求的来源。一般安全起见，应用程序在未经用户同意时不应发送此头域，以免侵犯个人隐私。

9.10.22. Host

请求头域 Host 定义了被请求资源的 Internet 主机地址和端口号，它可以由用户或有关资源（一般为 HTTP URL）给出的初始 URL 得到。Host 头域的值必须代表初始 URL 给出的原始服务器或网关的网络地址。一个客户则必须在用于 Internet 的 HTTP/1.1 请求消息中包含 Host 头域。格式为：

```
Host = "Host" ":" host [ ":" port ]
```

举个例子，对 URL 为＜http://www.w3.org/pub/WWW/＞的原始服务器上资源的请求必须写成：

```
GET /pub/WWW/ HTTP/1.1
Host: www.w3.org
```

9.10.23. If-Modified-Since

请求头域 If-Modified-Since 可以使 GET 操作变成条件 GET，其含义是：如果被请求变体自从该域中指定的时间以来没有被修改过，那么服务器就不会返回一个实体。一个带有 If-Modified-Since 头域而没有 Range 头域的 GET 操作的语义是请求传输某个被标识的实体，条件是只有在该实体自从 If-Modified-Since 头域给出的日期以来被修改过时才传输。有了这

个功能，就可以用最少的数据流量对缓存信息进行有效地更新。头域 If-Unmodified-Since 和 If-Modified-Since 的语义正好相反。

格式为：

```
If-Modified-Since=“If-Modified-Since””:”HTTP-date
```

例如：

```
If-Modified-Since:Sat,29Oct199419:43:31GMT
```

9.10.24. If-Match

If-Match 头域用来指定一些条件，但条件成立时，才去执行对应的请求方法，对相应的实体进行处理。

```
If-Match = "If-Match" ":" ( "*" | 1#entity-tag )
```

若其中的 entity-tag 与正常 GET 请求返回的实体的某标识匹配，或者其中给定的是“*”，则去执行请求方法，否则不执行。

例如：

```
If-Match: "xyzzy"
If-Match: "xyzzy", "r2d2xxxx", "c3piozzzz"
If-Match: *
```

当无法匹配时，应返回 412（前提条件不满足）响应。

If-None-Match 头域与本头域类似。

9.10.25. If-Range

如果一个客户在其缓存中有某个实体的一个部分拷贝，并且希望在缓存中存放一个整个实体的最新拷贝，那么可以使用带 Range 请求头域的部分 GET 操作（比如含有 If-Unomodified-Since 的 GET）。然而，如果因为实体被修改而导致条件失败，客户只有再次发出第二次请求以获得完整的当前实体内容。

有了 If-Range 就可以实现“短路”请求，即：如果实体没有被改变，就发送缓存中当前没有的那个部分；否则，就发送完整的新实体。

格式为：

```
If-Range=“If-Range””:”（entity-tag|HTTP-date）
```

9.10.26. If-Unmodified-Since

请求头域 If-Unmodified-Since 可使一个操作成为有条件的操作。其语义是，如果被请求的资源自该域指定的时间起没有被修改过，那么服务器应该执行被请求的操作，就好象 If-Unmodifed-Since 头域不存在一样。

格式为：

```
If-Unmodified-Since=“If-Unmodified-Since””:”HTTP-date
```

例如：

```
If-Unmodified-Since:Sat,29Oct199419:43:41GMT
```

9.10.27. Last-Modified

实体头域 Last-Modified **用来说明被服务器所承认的某个资源变体上次被修改的日期和时间。**

格式为：

```
Last-Modified=“Last-Modified”“:”HTTP-date
```

例如：

```
Last-Modified:Tue,15Nov199412:45:26GMT
```

请注意，该头域的确切含义取决于原始服务器的实现和初始资源的性质。对文件来说，该域的值可能就是文件系统上次修改时间；对于含有动态变化部分（Dynamical parts**）的实体来说，它可能取所有组成部件上次修改时间集合中的最近一个值；而对数据库网关而言，该值则可能是记录的上次更新时间戳；另外，对于虚拟的对象而言，则可能是对象内部状态的上次改变时间。**

9.10.28. Location

应答头域 Location **用来为接收方重定向一个不同于** Request-URI **地址的新地址，以完成相应的请求操作或进行一个新资源的确认。对** 201 **类（创建）应答来说，**Location **属于该请求创建的新资源；在** 3xx**（重定向）响应中则用于指定重定向的地址。**Content-Location **和** Location **头域的不同之处在于** Content-Location **指明了请求消息所携带的实体的初始位置，所以** Content-Location **和** Location **可能同时存在于一个应答消息中。**

格式为：

```
Location=“Location”“:”absoluteURI
```

例如：

```
Location:http://www.w3.org/pub/WWW/People.html
```

9.10.29. Max-Forwards

Max-Forwards **请求头域与** TRACE **方法配合使用，用来限制请求前传经过的最大代理或网关服务器数目。**

```
Max-Forwards   = "Max-Forwards" ":" 1*DIGIT
```

没有此头域的限制，请求消息有可能在一个代理/网关构成的闭环中死循环，而得不到响应。

9.10.30. Pragma

通用头域 Pragma **包含了可能对请求/应答链路上任何接收者产生作用并与** HTTP **实现相关的配置信息（**Directives**）。所有的配置信息可以指定不同于** HTTP **协议规范的可选行为。**

格式为：

```
Pragma=“Pragma”“:”1#pragma-directive
pragama-directive=“no-cache”|extension-pragma
extension-pragma=token[“=”（token|quoted-string）]
```

上面的“ no-cache ”如果出现在某个请求消息中，则任何请求/应答链上的应用进程都应该向原始服务器转发该请求，即使这个应用进程已经有了被请求内容的缓存拷贝。

9.10.31. Proxy-Authenticate

应答头域 Proxy-Authenticate 必须被包含在一个 407 类（代理身份认证请求）的应答消息中。该头域的值由认证策略和可作用于由相应 Request-URI 地址标识的客户代理的参数构成。和 WWW-Authenticate 不同的是，头域 Proxy-Authenticate 只对本次连接起作用，而不应该传给下游的客户进程。

格式为：

```
Proxy-Authentication="Proxy-Authentication"":"challenge
```

这里的 challenge 是来自发送方的提问信息。

9.10.32. Proxy-Authorization

请求头域 Proxy-Authorization 可以让客户向一个要求身份认证的代理说明自己或(用户)的身份。

格式为：

```
Proxy-Authorization="Proxy-Authorization"":"credentials
```

这里的 credentials 是客户发出的身份认证信息。

9.10.33. Range

由于所有的 HTTP 实体都在 HTTP 消息包中以字节序列的形式表示，所以对任何 HTTP 实体来说字节区域范围的概念都是有意义的。HTTP 中字节区域的定义作用于实体内容（注意和消息内容的不同）中的字节序列。一个字节区域操作可能指定单个实体内部的字节单一区域或者实体域的集合。

格式为：

```
ranges-specifier=byte-ranges-specifier
byte-ranges-specifier=bytes-unit"="byte-range-set
byte-range-set=1#(byte-range-spec|suffix-byte-range-spec)
byte-range-spec=first-byte-pos"-"[last-byte-pos]
first-byte-pos=1*DIGIT
last-byte-pos=1*DIGIT
```

这里的 suffix-byte-range-spec 可以指定实体内容尾部的若干个字节。

```
suffix-byte-range-spec="-"suffix-length
```

9.10.34. Server

应答头域 Server 包含了处理客户请求的原始服务器软件的信息。该头域可含有多个产品标识符（Token）和说明服务器或任何相关子产品的注释信息。

请注意暴露服务器软件的特定版本信息，可能使服务器更易受到对存有安全漏洞软件的

攻击，因此，建议服务器的实现最好使 Server 域成为可配置的选项。

格式为：

```
Server=“Server””:”1*（product|comment）
```

例如：

```
Server:CERN/3.0libwww/2.17
```

9.10.35. Transfer-Encoding

通用头域 Transfer-Encoding 用来说明为了在发送方和接收方之间安全地传输消息内容从而对其采用了哪种编码（如果有的话）。和 Content-Encoding 不同的是传输编码是消息包的特征，而不是实体的特征。

格式为：

```
Transfer-Encoding=“Transfer-Encoding””:”1#transfer-coding
```

例如：

```
Transfer-Encoding:chunked
```

这里的“chunked”（分块传输）是编码方式的一种。

9.10.36. Upgrade

通用头域 Upgrade 允许客户说明它还支持其它哪些通信协议，以便当服务器认可协议转换时使用新的交互协议。服务器必须在 101 类应答消息中使用一个 Upgrade 域来说明转换的协议。

Upgrade 头域的目的是提供一种从 HTTP/1.1 转换到其它不兼容（Incompatible）协议的简单机制。通过让客户采用一种被普遍接受的协议向服务器发出请求，传达自己期望使用比现有“更优”协议的意愿，这种机制减小了在不兼容协议之间转换的难度。

Upgrade 头域只能用于已经存在的传输层连接之上的应用层协议转换，而且只对当前连接有效，所以 HTTP/1.1 消息包中任何 Upgrade 出现的地方，Connection 头域的值中必须提供关键词 upgrade。

格式为：

```
Upgrade=“Upgrade””:”1#product
```

例如：

```
Upgrade:HTTP/2.0,SHTTP/1.3,IRC/6.9,RTA/x11
```

9.10.37. User-Agent

请求头域 User-Agent 包含了有关发起请求的客户端（UserAgent）的软件信息。设置该头域是为了进行统计、协议冲突跟踪以及用户软件的自动识别，以便为了避免特定用户软件的限制而进行应答消息跟踪。

格式为：

```
User-Agent=“User-Agent””:”1*（product|comment）
```

例如：

```
User-Agent:CERN-LineMode/2.15libwww/2.17b3
```

9.10.38. Vary

服务器通过应答头域 Vary 发出信号，表明它采用了服务器驱动的内容协商方式，从应答的可选表示中选出应答实体。Vary 域的值表示给定的头域包含了当前表示可能变化的尺度（Dimensions），或者变化尺度没有确定（用“*”表示），从而可因将来请求消息的不同而变化。

格式为：

```
Vary=“Vary””:” (“*”|1#field-name)
```

9.10.39. Via

Via 通用头域用来指示在客户端软件和原始服务器之间的请求/应答链路上存在的中继点，包括使用的协议及自身标识。因此该头域的使用者是代理和网关服务器。

```
Via =  "Via" ":" 1#( received-protocol received-by [ comment ] )

received-protocol = [ protocol-name "/" ] protocol-version
protocol-name     = token
protocol-version  = token
received-by       = ( host [ ":" port ] ) | pseudonym
pseudonym         = token
```

例如：

```
Via: 1.0 fred, 1.1 nowhere.com (Apache/1.1)
```

9.10.40. WWW_Authenticate

应答头域 WWW_Authenticate 必须被包含在一个 401 类（未授权）应答消息中。头域值中的 chanllenge 说明了身份认证策略和作用于相应 Request URI 地址的参数。

格式为：

```
WWW-Authenticate=“WWW-Authenticate””:”1#chanllenge
```

第11节 与早期版本的兼容

HTTP/1.1 协议具有设计精巧特点的同时，也采取各种措施来使各种实现软件能更容易地支持 HTTP 的先前版本。HTTP/1.1 规范中对各种实现软件包括服务器和客户端程序提出了要求。

对服务器软件的要求包括：

- 能够识别 HTTP0.9、1.0 或 1.1 请求中请求行的各种格式；
- 能够理解 HTTP0.9、1.0 或 1.1 格式的任何合理请求；
- 能够以与客户端程序所使用协议相同的主版本返回响应。

要求客户端程序：

- 能够识别 HTTP1.0 或 1.1 响应中状态行的格式；

- **能够理解** HTTP0.9、1.0 **或** 1.1 **格式的任何合理响应。**

附录 Apache 各模块配置接口定义

各模块的相关的控制结构及配置结构总结如下，可方便查询。

http_core

core_handlers

```
static const handler_rec core_handlers[] = {
{ "*/*", default_handler },
{ "default-handler", default_handler },
{ NULL, NULL }
};
```

core_cmds

```
static const command_rec core_cmds[] = {

/* Old access config file commands */

{ "<Directory", dirsection, NULL, RSRC_CONF, RAW_ARGS,
  "Container for directives affecting resources located in the specified "
  "directories" },
{ end_directory_section, end_nested_section, NULL, ACCESS_CONF, NO_ARGS,
  "Marks end of <Directory>" },
{ "<Location", urlsection, NULL, RSRC_CONF, RAW_ARGS,
  "Container for directives affecting resources accessed through the "
  "specified URL paths" },
{ end_location_section, end_nested_section, NULL, ACCESS_CONF, NO_ARGS,
  "Marks end of <Location>" },
{ "<VirtualHost", virtualhost_section, NULL, RSRC_CONF, RAW_ARGS,
  "Container to map directives to a particular virtual host, takes one or "
  "more host addresses" },
{ end_virtualhost_section, end_nested_section, NULL, RSRC_CONF, NO_ARGS,
  "Marks end of <VirtualHost>" },
{ "<Files", filesection, NULL, OR_ALL, RAW_ARGS, "Container for directives "
  "affecting files matching specified patterns" },
{ end_files_section, end_nested_section, NULL, OR_ALL, NO_ARGS,
  "Marks end of <Files>" },
```

```
{ "<Limit", ap_limit_section, NULL, OR_ALL, RAW_ARGS, "Container for "
  "authentication directives when accessed using specified HTTP methods" },
{ "</Limit>", endlimit_section, NULL, OR_ALL, NO_ARGS,
  "Marks end of <Limit>" },
{ "<LimitExcept", ap_limit_section, (void*)1, OR_ALL, RAW_ARGS,
  "Container for authentication directives to be applied when any HTTP "
  "method other than those specified is used to access the resource" },
{ "</LimitExcept>", endlimit_section, (void*)1, OR_ALL, NO_ARGS,
  "Marks end of <LimitExcept>" },
{ "<IfModule", start_ifmod, NULL, OR_ALL, TAKE1,
  "Container for directives based on existance of specified modules" },
{ end_ifmodule_section, end_ifmod, NULL, OR_ALL, NO_ARGS,
  "Marks end of <IfModule>" },
{ "<IfDefine", start_ifdefine, NULL, OR_ALL, TAKE1,
  "Container for directives based on existance of command line defines" },
{ end_ifdefine_section, end_ifdefine, NULL, OR_ALL, NO_ARGS,
  "Marks end of <IfDefine>" },
{ "<DirectoryMatch", dirsection, (void*)1, RSRC_CONF, RAW_ARGS,
  "Container for directives affecting resources located in the "
  "specified directories" },
{ end_directorymatch_section, end_nested_section, NULL, ACCESS_CONF, NO_ARGS,
  "Marks end of <DirectoryMatch>" },
{ "<LocationMatch", urlsection, (void*)1, RSRC_CONF, RAW_ARGS,
  "Container for directives affecting resources accessed through the "
  "specified URL paths" },
{ end_locationmatch_section, end_nested_section, NULL, ACCESS_CONF, NO_ARGS,
  "Marks end of <LocationMatch>" },
{ "<FilesMatch", filesection, (void*)1, OR_ALL, RAW_ARGS,
  "Container for directives affecting files matching specified patterns" },
{ end_filesmatch_section, end_nested_section, NULL, OR_ALL, NO_ARGS,
  "Marks end of <FilesMatch>" },
{ "AuthType", ap_set_string_slot,
  (void*)XtOffsetOf(core_dir_config, ap_auth_type), OR_AUTHCFG, TAKE1,
  "An HTTP authorization type (e.g., \"Basic\")" },
{ "AuthName", set_authname, NULL, OR_AUTHCFG, TAKE1,
  "The authentication realm (e.g. \"Members Only\")" },
{ "Require", require, NULL, OR_AUTHCFG, RAW_ARGS,
  "Selects which authenticated users or groups may access a protected space" },
{ "Satisfy", satisfy, NULL, OR_AUTHCFG, TAKE1,
  "access policy if both allow and require used ('all' or 'any')" },
#ifdef GPROF
{ "GprofDir", set_gprof_dir, NULL, RSRC_CONF, TAKE1,
  "Directory to plop gmon.out files" },
#endif
```

```
/* Old resource config file commands */

{ "AccessFileName", set_access_name, NULL, RSRC_CONF, RAW_ARGS,
  "Name(s) of per-directory config files (default: .htaccess)" },
{ "DocumentRoot", set_document_root, NULL, RSRC_CONF, TAKE1,
  "Root directory of the document tree"  },
{ "ErrorDocument", set_error_document, NULL, OR_FILEINFO, RAW_ARGS,
  "Change responses for HTTP errors" },
{ "AllowOverride", set_override, NULL, ACCESS_CONF, RAW_ARGS,
  "Controls what groups of directives can be configured by per-directory "
  "config files" },
{ "Options", set_options, NULL, OR_OPTIONS, RAW_ARGS,
  "Set a number of attributes for a given directory" },
{ "DefaultType", ap_set_string_slot,
  (void*)XtOffsetOf (core_dir_config, ap_default_type),
  OR_FILEINFO, TAKE1, "the default MIME type for untypable files" },

/* Old server config file commands */

{ "ServerType", server_type, NULL, RSRC_CONF, TAKE1,
  "'inetd' or 'standalone'"},
{ "Port", server_port, NULL, RSRC_CONF, TAKE1, "A TCP port number"},
{ "HostnameLookups", set_hostname_lookups, NULL, ACCESS_CONF|RSRC_CONF, TAKE1,
  "\"on\" to enable, \"off\" to disable reverse DNS lookups, or \"double\" to "
  "enable double-reverse DNS lookups" },
{ "User", set_user, NULL, RSRC_CONF, TAKE1,
  "Effective user id for this server"},
{ "Group", set_group, NULL, RSRC_CONF, TAKE1,
  "Effective group id for this server"},
{ "ServerAdmin", set_server_string_slot,
  (void *)XtOffsetOf (server_rec, server_admin), RSRC_CONF, TAKE1,
  "The email address of the server administrator" },
{ "ServerName", set_server_string_slot,
  (void *)XtOffsetOf (server_rec, server_hostname), RSRC_CONF, TAKE1,
  "The hostname of the server" },
{ "ServerSignature", set_signature_flag, NULL, ACCESS_CONF|RSRC_CONF, TAKE1,
  "En-/disable server signature (on|off|email)" },
{ "ServerRoot", set_server_root, NULL, RSRC_CONF, TAKE1,
  "Common directory of server-related files (logs, confs, etc.)" },
{ "ErrorLog", set_server_string_slot,
  (void *)XtOffsetOf (server_rec, error_fname), RSRC_CONF, TAKE1,
  "The filename of the error log" },
{ "PidFile", set_pidfile, NULL, RSRC_CONF, TAKE1,
```

```
    "A file for logging the server process ID"},
{ "ScoreBoardFile", set_scoreboard, NULL, RSRC_CONF, TAKE1,
    "A file for Apache to maintain runtime process management information"},
{ "LockFile", set_lockfile, NULL, RSRC_CONF, TAKE1,
    "The lockfile used when Apache needs to lock the accept() call"},
{ "AccessConfig", set_server_string_slot,
  (void *)XtOffsetOf (server_rec, access_confname), RSRC_CONF, TAKE1,
  "The filename of the access config file" },
{ "ResourceConfig", set_server_string_slot,
  (void *)XtOffsetOf (server_rec, srm_confname), RSRC_CONF, TAKE1,
  "The filename of the resource config file" },
{ "ServerAlias", set_server_alias, NULL, RSRC_CONF, RAW_ARGS,
  "A name or names alternately used to access the server" },
{ "ServerPath", set_serverpath, NULL, RSRC_CONF, TAKE1,
  "The pathname the server can be reached at" },
{ "Timeout", set_timeout, NULL, RSRC_CONF, TAKE1, "Timeout duration (sec)" },
{ "KeepAliveTimeout", set_keep_alive_timeout, NULL, RSRC_CONF, TAKE1,
  "Keep-Alive timeout duration (sec)"},
{ "MaxKeepAliveRequests", set_keep_alive_max, NULL, RSRC_CONF, TAKE1,
  "Maximum number of Keep-Alive requests per connection, or 0 for infinite" },
{ "KeepAlive", set_keep_alive, NULL, RSRC_CONF, TAKE1,
  "Whether persistent connections should be On or Off" },
{ "IdentityCheck", set_idcheck, NULL, RSRC_CONF|ACCESS_CONF, FLAG,
  "Enable identd (RFC 1413) user lookups - SLOW" },
{ "ContentDigest", set_content_md5, NULL, OR_OPTIONS,
  FLAG, "whether or not to send a Content-MD5 header with each request" },
{ "UseCanonicalName", set_use_canonical_name, NULL,
  OR_OPTIONS, FLAG,
  "Whether or not to always use the canonical ServerName : Port when "
  "constructing URLs" },
{ "StartServers", set_daemons_to_start, NULL, RSRC_CONF, TAKE1,
  "Number of child processes launched at server startup" },
{ "MinSpareServers", set_min_free_servers, NULL, RSRC_CONF, TAKE1,
  "Minimum number of idle children, to handle request spikes" },
{ "MaxSpareServers", set_max_free_servers, NULL, RSRC_CONF, TAKE1,
  "Maximum number of idle children" },
{ "MaxServers", set_max_free_servers, NULL, RSRC_CONF, TAKE1,
  "Deprecated equivalent to MaxSpareServers" },
{ "ServersSafetyLimit", set_server_limit, NULL, RSRC_CONF, TAKE1,
  "Deprecated equivalent to MaxClients" },
{ "MaxClients", set_server_limit, NULL, RSRC_CONF, TAKE1,
  "Maximum number of children alive at the same time" },
{ "MaxRequestsPerChild", set_max_requests, NULL, RSRC_CONF, TAKE1,
  "Maximum number of requests a particular child serves before dying." },
```

```
{ "RLimitCPU",
#ifdef RLIMIT_CPU
  set_limit_cpu, (void*)XtOffsetOf(core_dir_config, limit_cpu),
#else
  no_set_limit, NULL,
#endif
  OR_ALL, TAKE12, "Soft/hard limits for max CPU usage in seconds" },
{ "RLimitMEM",
#if defined (RLIMIT_DATA) || defined (RLIMIT_VMEM) || defined (RLIMIT_AS)
  set_limit_mem, (void*)XtOffsetOf(core_dir_config, limit_mem),
#else
  no_set_limit, NULL,
#endif
  OR_ALL, TAKE12, "Soft/hard limits for max memory usage per process" },
{ "RLimitNPROC",
#ifdef RLIMIT_NPROC
  set_limit_nproc, (void*)XtOffsetOf(core_dir_config, limit_nproc),
#else
  no_set_limit, NULL,
#endif
   OR_ALL, TAKE12, "soft/hard limits for max number of processes per uid" },
{ "BindAddress", set_bind_address, NULL, RSRC_CONF, TAKE1,
  "'*', a numeric IP address, or the name of a host with a unique IP address"},
{ "Listen", set_listener, NULL, RSRC_CONF, TAKE1,
  "A port number or a numeric IP address and a port number"},
{ "SendBufferSize", set_send_buffer_size, NULL, RSRC_CONF, TAKE1,
  "Send buffer size in bytes"},
{ "AddModule", add_module_command, NULL, RSRC_CONF, ITERATE,
  "The name of a module" },
{ "ClearModuleList", clear_module_list_command, NULL, RSRC_CONF, NO_ARGS,
  NULL },
{ "ThreadsPerChild", set_threads, NULL, RSRC_CONF, TAKE1,
  "Number of threads a child creates" },
{ "ExcessRequestsPerChild", set_excess_requests, NULL, RSRC_CONF, TAKE1,
  "Maximum number of requests a particular child serves after it is ready "
  "to die." },
{ "ListenBacklog", set_listenbacklog, NULL, RSRC_CONF, TAKE1,
  "Maximum length of the queue of pending connections, as used by listen(2)" },
{ "CoreDumpDirectory", set_coredumpdir, NULL, RSRC_CONF, TAKE1,
  "The location of the directory Apache changes to before dumping core" },
{ "Include", include_config, NULL, (RSRC_CONF | ACCESS_CONF), TAKE1,
  "Name of the config file to be included" },
{ "LogLevel", set_loglevel, NULL, RSRC_CONF, TAKE1,
  "Level of verbosity in error logging" },
```

```
{ "NameVirtualHost", ap_set_name_virtual_host, NULL, RSRC_CONF, TAKE1,
  "A numeric IP address:port, or the name of a host" },
#ifdef _OSD_POSIX
{ "BS2000Account", set_bs2000_account, NULL, RSRC_CONF, TAKE1,
  "Name of server User's bs2000 logon account name" },
#endif
#ifdef WIN32
{ "ScriptInterpreterSource", set_interpreter_source, NULL, OR_FILEINFO, TAKE1,
 "Where to find interpreter to run Win32 scripts (Registry or script shebang line)" },
#endif
{ "ServerTokens", set_serv_tokens, NULL, RSRC_CONF, TAKE1,
  "Determine tokens displayed in the Server: header - Min(imal), OS or Full" },
{ "LimitRequestLine", set_limit_req_line, NULL, RSRC_CONF, TAKE1,
  "Limit on maximum size of an HTTP request line"},
{ "LimitRequestFieldsize", set_limit_req_fieldsize, NULL, RSRC_CONF, TAKE1,
  "Limit on maximum size of an HTTP request header field"},
{ "LimitRequestFields", set_limit_req_fields, NULL, RSRC_CONF, TAKE1,
  "Limit (0 = unlimited) on max number of header fields in a request message"},
{ "LimitRequestBody", set_limit_req_body,
  (void*)XtOffsetOf(core_dir_config, limit_req_body),
  OR_ALL, TAKE1,
  "Limit (in bytes) on maximum size of request message body" },
{ NULL }
};
```

core_module

```
API_VAR_EXPORT module core_module = {
    STANDARD_MODULE_STUFF,
    NULL,            /* initializer */
    create_core_dir_config, /* create per-directory config structure */
    merge_core_dir_configs, /* merge per-directory config structures */
    create_core_server_config,   /* create per-server config structure */
    merge_core_server_configs,   /* merge per-server config structures */
    core_cmds,           /* command table */
    core_handlers,       /* handlers */
    core_translate,      /* translate_handler */
    NULL,            /* check_user_id */
    NULL,            /* check auth */
    do_nothing,          /* check access */
    do_nothing,          /* type_checker */
    NULL,            /* pre-run fixups */
    NULL,            /* logger */
    NULL,            /* header parser */
```

```
    NULL,            /* child_init */
    NULL,            /* child_exit */
    NULL         /* post_read_request */
};
```

mod_proxy

proxy_handlers

```
static const handler_rec proxy_handlers[] =
{
    {"proxy-server", proxy_handler},
    {NULL}
};
```

proxy_cmds

```
static const command_rec proxy_cmds[] =
{
    {"ProxyRequests", set_proxy_req, NULL, RSRC_CONF, FLAG,
     "on if the true proxy requests should be accepted"},
    {"ProxyRemote", add_proxy, NULL, RSRC_CONF, TAKE2,
     "a scheme, partial URL or '*' and a proxy server"},
    {"ProxyPass", add_pass, NULL, RSRC_CONF, TAKE2,
     "a virtual path and a URL"},
    {"ProxyPassReverse", add_pass_reverse, NULL, RSRC_CONF, TAKE2,
     "a virtual path and a URL for reverse proxy behaviour"},
    {"ProxyBlock", set_proxy_exclude, NULL, RSRC_CONF, ITERATE,
     "A list of names, hosts or domains to which the proxy will not connect"},
    {"ProxyReceiveBufferSize", set_recv_buffer_size, NULL, RSRC_CONF, TAKE1,
     "Receive buffer size for outgoing HTTP and FTP connections in bytes"},
    {"NoProxy", set_proxy_dirconn, NULL, RSRC_CONF, ITERATE,
  "A list of domains, hosts, or subnets to which the proxy will connect directly"},
    {"ProxyDomain", set_proxy_domain, NULL, RSRC_CONF, TAKE1,
     "The default intranet domain name (in absence of a domain in the URL)"},
    {"AllowCONNECT", set_allowed_ports, NULL, RSRC_CONF, ITERATE,
     "A list of ports which CONNECT may connect to"},
    {"CacheRoot", set_cache_root, NULL, RSRC_CONF, TAKE1,
     "The directory to store cache files"},
    {"CacheSize", set_cache_size, NULL, RSRC_CONF, TAKE1,
     "The maximum disk space used by the cache in Kb"},
```

```
    {"CacheMaxExpire", set_cache_maxex, NULL, RSRC_CONF, TAKE1,
     "The maximum time in hours to cache a document"},
    {"CacheDefaultExpire", set_cache_defex, NULL, RSRC_CONF, TAKE1,
     "The default time in hours to cache a document"},
    {"CacheLastModifiedFactor", set_cache_factor, NULL, RSRC_CONF, TAKE1,
     "The factor used to estimate Expires date from LastModified date"},
    {"CacheGcInterval", set_cache_gcint, NULL, RSRC_CONF, TAKE1,
     "The interval between garbage collections, in hours"},
    {"CacheDirLevels", set_cache_dirlevels, NULL, RSRC_CONF, TAKE1,
     "The number of levels of subdirectories in the cache"},
    {"CacheDirLength", set_cache_dirlength, NULL, RSRC_CONF, TAKE1,
     "The number of characters in subdirectory names"},
    {"NoCache", set_cache_exclude, NULL, RSRC_CONF, ITERATE,
     "A list of names, hosts or domains for which caching is *not* provided"},
    {"CacheForceCompletion", set_cache_completion, NULL, RSRC_CONF, TAKE1,
     "Force a http cache completion after this percentage is loaded"},
    {"ProxyVia", set_via_opt, NULL, RSRC_CONF, TAKE1,
     "Configure Via: proxy header header to one of: on | off | block | full"},
    {NULL}
};
```

proxy_module

```
module MODULE_VAR_EXPORT proxy_module =
{
    STANDARD_MODULE_STUFF,
    proxy_init,          /* initializer */
    NULL,           /* create per-directory config structure */
    NULL,           /* merge per-directory config structures */
    create_proxy_config, /* create per-server config structure */
    NULL,           /* merge per-server config structures */
    proxy_cmds,          /* command table */
    proxy_handlers,      /* handlers */
    proxy_trans,     /* translate_handler */
    NULL,           /* check_user_id */
    NULL,           /* check auth */
    NULL,           /* check access */
    NULL,           /* type_checker */
    proxy_fixup,     /* pre-run fixups */
    NULL,           /* logger */
    NULL,           /* header parser */
    NULL,           /* child_init */
    NULL,           /* child_exit */
    proxy_detect     /* post read-request */
```

```
};
```

mod_header

headers_cmds

```
static const command_rec headers_cmds[] =
{
    {"Header", header_cmd, NULL, OR_FILEINFO, TAKE23,
     "an action, header and value"},
    {NULL}
};
```

headers_module

```
module MODULE_VAR_EXPORT headers_module =
{
    STANDARD_MODULE_STUFF,
    NULL,                    /* initializer */
    create_headers_dir_config,  /* dir config creater */
    merge_headers_config,      /* dir merger --- default is to override */
    create_headers_config,     /* server config */
    merge_headers_config,      /* merge server configs */
    headers_cmds,             /* command table */
    NULL,                    /* handlers */
    NULL,                    /* filename translation */
    NULL,                    /* check_user_id */
    NULL,                    /* check auth */
    NULL,                    /* check access */
    NULL,                    /* type_checker */
    fixup_headers,            /* fixups */
    NULL,                    /* logger */
    NULL,                    /* header parser */
    NULL,                    /* child_init */
    NULL,                    /* child_exit */
    NULL                     /* post read-request */
};
```

mod_info

info_cmds

```
static const command_rec info_cmds[] =
{
    {"AddModuleInfo", add_module_info, NULL, RSRC_CONF, TAKE2,
     "a module name and additional information on that module"},
    {NULL}
};
```

info_handlers

```
static const handler_rec info_handlers[] =
{
    {"server-info", display_info},
    {NULL}
};
```

info_module

```
module MODULE_VAR_EXPORT info_module =
{
    STANDARD_MODULE_STUFF,
    NULL,                   /* initializer */
    NULL,                   /* dir config creater */
    NULL,                   /* dir merger --- default is to override */
    create_info_config,       /* server config */
    merge_info_config,        /* merge server config */
    info_cmds,               /* command table */
    info_handlers,            /* handlers */
    NULL,                   /* filename translation */
    NULL,                   /* check_user_id */
    NULL,                   /* check auth */
    NULL,                   /* check access */
    NULL,                   /* type_checker */
    NULL,                   /* fixups */
    NULL,                   /* logger */
    NULL,                   /* header parser */
    NULL,                   /* child_init */
```

```
    NULL,                    /* child_exit */
    NULL                     /* post read-request */
};
```

mod_status

status_handlers

```
static const handler_rec status_handlers[] =
{
    {STATUS_MAGIC_TYPE, status_handler},
    {"server-status", status_handler},
    {NULL}
};
```

status_module_cmds

```
static const command_rec status_module_cmds[] =
{
    { "ExtendedStatus", set_extended_status, NULL, RSRC_CONF, TAKE1,
      "\"On\" to enable extended status information, \"Off\" to disable" },
    {NULL}
};
```

status_module

```
module MODULE_VAR_EXPORT status_module =
{
    STANDARD_MODULE_STUFF,
    status_init,     /* initializer */
    NULL,            /* dir config creater */
    NULL,            /* dir merger --- default is to override */
    NULL,            /* server config */
    NULL,            /* merge server config */
    status_module_cmds,      /* command table */
    status_handlers,     /* handlers */
    NULL,            /* filename translation */
    NULL,            /* check_user_id */
    NULL,            /* check auth */
    NULL,            /* check access */
```

```
    NULL,            /* type_checker */
    NULL,            /* fixups */
    NULL,            /* logger */
    NULL,            /* header parser */
    NULL,            /* child_init */
    NULL,            /* child_exit */
    NULL         /* post read-request */
};
```

mod_access

access_cmds

```
static const command_rec access_cmds[] =
{
    {"order", order, NULL, OR_LIMIT, TAKE1,
     "'allow,deny', 'deny,allow', or 'mutual-failure'"},
    {"allow", allow_cmd, &its_an_allow, OR_LIMIT, ITERATE2,
     "'from' followed by hostnames or IP-address wildcards"},
    {"deny", allow_cmd, NULL, OR_LIMIT, ITERATE2,
     "'from' followed by hostnames or IP-address wildcards"},
    {NULL}
};
```

access_module

```
module MODULE_VAR_EXPORT access_module =
{
    STANDARD_MODULE_STUFF,
    NULL,            /* initializer */
    create_access_dir_config,    /* dir config creater */
    NULL,            /* dir merger --- default is to override */
    NULL,            /* server config */
    NULL,            /* merge server config */
    access_cmds,
    NULL,            /* handlers */
    NULL,            /* filename translation */
    NULL,            /* check_user_id */
    NULL,            /* check auth */
    check_dir_access,        /* check access */
    NULL,            /* type_checker */
```

```
    NULL,            /* fixups */
    NULL,            /* logger */
    NULL,            /* header parser */
    NULL,            /* child_init */
    NULL,            /* child_exit */
    NULL         /* post read-request */
};
```

mod_digest

digest_cmds

```
static const command_rec digest_cmds[] =
{
    {"AuthDigestFile", set_digest_slot,
  (void *) XtOffsetOf(digest_config_rec, pwfile), OR_AUTHCFG, TAKE12, NULL},
    {NULL}
};
```

digest_module

```
module MODULE_VAR_EXPORT digest_module =
{
    STANDARD_MODULE_STUFF,
    NULL,            /* initializer */
    create_digest_dir_config,    /* dir config creater */
    NULL,            /* dir merger --- default is to override */
    NULL,            /* server config */
    NULL,            /* merge server config */
    digest_cmds,     /* command table */
    NULL,            /* handlers */
    NULL,            /* filename translation */
    authenticate_digest_user,    /* check_user_id */
    digest_check_auth,       /* check auth */
    NULL,            /* check access */
    NULL,            /* type_checker */
    NULL,            /* fixups */
    NULL,            /* logger */
    NULL,            /* header parser */
    NULL,            /* child_init */
    NULL,            /* child_exit */
```

```
    NULL        /* post read-request */
};
```

mod_auth

auth_cmds

```
static const command_rec auth_cmds[] =
{
    {"AuthUserFile", set_auth_slot,
     (void *) XtOffsetOf(auth_config_rec, auth_pwfile), OR_AUTHCFG, TAKE12,
     "text file containing user IDs and passwords"},
    {"AuthGroupFile", set_auth_slot,
     (void *) XtOffsetOf(auth_config_rec, auth_grpfile), OR_AUTHCFG, TAKE12,
     "text file containing group names and member user IDs"},
    {"AuthAuthoritative", ap_set_flag_slot,
     (void *) XtOffsetOf(auth_config_rec, auth_authoritative),
     OR_AUTHCFG, FLAG,
     "Set to 'no' to allow access control to be passed along to lower modules if
the UserID is not known to this module"},
    {NULL}
};
```

auth_module

```
module MODULE_VAR_EXPORT auth_module =
{
    STANDARD_MODULE_STUFF,
    NULL,            /* initializer */
    create_auth_dir_config, /* dir config creater */
    NULL,            /* dir merger --- default is to override */
    NULL,            /* server config */
    NULL,            /* merge server config */
    auth_cmds,           /* command table */
    NULL,            /* handlers */
    NULL,            /* filename translation */
    authenticate_basic_user,/* check_user_id */
    check_user_access,      /* check auth */
    NULL,            /* check access */
    NULL,            /* type_checker */
    NULL,            /* fixups */
```

```
    NULL,            /* logger */
    NULL,            /* header parser */
    NULL,            /* child_init */
    NULL,            /* child_exit */
    NULL         /* post read-request */
};
```

mod_anon_auth

anon_auth_cmds

```
static const command_rec anon_auth_cmds[] =
{
    {"Anonymous", anon_set_string_slots, NULL, OR_AUTHCFG, ITERATE,
     "a space-separated list of user IDs"},
    {"Anonymous_MustGiveEmail", anon_set_passwd_flag, NULL, OR_AUTHCFG, FLAG,
     "Limited to 'on' or 'off'"},
    {"Anonymous_NoUserId", anon_set_userid_flag, NULL, OR_AUTHCFG, FLAG,
     "Limited to 'on' or 'off'"},
{"Anonymous_VerifyEmail", anon_set_verifyemail_flag, NULL, OR_AUTHCFG, FLAG,
 "Limited to 'on' or 'off'"},
    {"Anonymous_LogEmail", anon_set_logemail_flag, NULL, OR_AUTHCFG, FLAG,
     "Limited to 'on' or 'off'"},
    {"Anonymous_Authoritative", anon_set_authoritative_flag, NULL, OR_AUTHCFG,
FLAG,
     "Limited to 'on' or 'off'"},

    {NULL}
};
```

anon_auth_module

```
module MODULE_VAR_EXPORT anon_auth_module =
{
    STANDARD_MODULE_STUFF,
    NULL,            /* initializer */
    create_anon_auth_dir_config,/* dir config creater */
    NULL,            /* dir merger ensure strictness */
    NULL,            /* server config */
    NULL,            /* merge server config */
    anon_auth_cmds,      /* command table */
```

```
    NULL,            /* handlers */
    NULL,            /* filename translation */
    anon_authenticate_basic_user,   /* check_user_id */
    check_anon_access,      /* check auth */
    NULL,            /* check access */
    NULL,            /* type_checker */
    NULL,            /* fixups */
    NULL,            /* logger */
    NULL,            /* header parser */
    NULL,            /* child_init */
    NULL,            /* child_exit */
    NULL         /* post read-request */
};
```

mod_log_config

config_log_cmds

```
static const command_rec config_log_cmds[] =
{
    {"CustomLog", add_custom_log, NULL, RSRC_CONF, TAKE23,
     "a file name, a custom log format string or format name, "
     "and an optional \"env=\" clause (see docs)"},
    {"TransferLog", set_transfer_log, NULL, RSRC_CONF, TAKE1,
     "the filename of the access log"},
    {"LogFormat", log_format, NULL, RSRC_CONF, TAKE12,
     "a log format string (see docs) and an optional format name"},
    {"CookieLog", set_cookie_log, NULL, RSRC_CONF, TAKE1,
     "the filename of the cookie log"},
    {NULL}
};
```

config_log_module

```
module MODULE_VAR_EXPORT config_log_module =
{
    STANDARD_MODULE_STUFF,
    init_config_log,          /* initializer */
    NULL,                    /* create per-dir config */
    NULL,                    /* merge per-dir config */
    make_config_log_state,     /* server config */
```

```
    merge_config_log_state,     /* merge server config */
    config_log_cmds,           /* command table */
    NULL,                     /* handlers */
    NULL,                     /* filename translation */
    NULL,                     /* check_user_id */
    NULL,                     /* check auth */
    NULL,                     /* check access */
    NULL,                     /* type_checker */
    NULL,                     /* fixups */
    multi_log_transaction,      /* logger */
    NULL,                     /* header parser */
    NULL,                     /* child_init */
#ifdef BUFFERED_LOGS
    flush_all_logs,            /* child_exit */
#else
    NULL,
#endif
    NULL                      /* post read-request */
};
```

mod_alias

alias_cmds

```
static const command_rec alias_cmds[] =
{
    {"Alias", add_alias, NULL, RSRC_CONF, TAKE2,
     "a fakename and a realname"},
    {"ScriptAlias", add_alias, "cgi-script", RSRC_CONF, TAKE2,
     "a fakename and a realname"},
    {"Redirect", add_redirect, (void *) HTTP_MOVED_TEMPORARILY,
     OR_FILEINFO, TAKE23,
  "an optional status, then document to be redirected and destination URL"},
    {"AliasMatch", add_alias_regex, NULL, RSRC_CONF, TAKE2,
     "a regular expression and a filename"},
    {"ScriptAliasMatch", add_alias_regex, "cgi-script", RSRC_CONF, TAKE2,
     "a regular expression and a filename"},
    {"RedirectMatch", add_redirect_regex, (void *) HTTP_MOVED_TEMPORARILY,
     OR_FILEINFO, TAKE23,
     "an optional status, then a regular expression and destination URL"},
    {"RedirectTemp", add_redirect, (void *) HTTP_MOVED_TEMPORARILY,
     OR_FILEINFO, TAKE2,
```

```
     "a document to be redirected, then the destination URL"},
    {"RedirectPermanent", add_redirect, (void *) HTTP_MOVED_PERMANENTLY,
     OR_FILEINFO, TAKE2,
     "a document to be redirected, then the destination URL"},
    {NULL}
};
```

alias_module

```
module MODULE_VAR_EXPORT alias_module =
{
    STANDARD_MODULE_STUFF,
    NULL,            /* initializer */
    create_alias_dir_config,/* dir config creater */
    merge_alias_dir_config, /* dir merger --- default is to override */
    create_alias_config,/* server config */
    merge_alias_config,     /* merge server configs */
    alias_cmds,          /* command table */
    NULL,            /* handlers */
    translate_alias_redir,   /* filename translation */
    NULL,            /* check_user_id */
    NULL,            /* check auth */
    NULL,            /* check access */
    NULL,            /* type_checker */
    fixup_redir,     /* fixups */
    NULL,            /* logger */
    NULL,            /* header parser */
    NULL,            /* child_init */
    NULL,            /* child_exit */
    NULL         /* post read-request */
};
```

mod_dir

dir_cmds

```
static const command_rec dir_cmds[] =
{
    {"DirectoryIndex", add_index, NULL,
     DIR_CMD_PERMS, ITERATE,
     "a list of file names"},
```

```
    {NULL}
};
```

dir_handlers

```
static const handler_rec dir_handlers[] =
{
    {DIR_MAGIC_TYPE, handle_dir},
    {NULL}
};
```

dir_module

```
module MODULE_VAR_EXPORT dir_module =
{
    STANDARD_MODULE_STUFF,
    NULL,                   /* initializer */
    create_dir_config,        /* dir config creater */
    merge_dir_configs,        /* dir merger --- default is to override */
    NULL,                   /* server config */
    NULL,                   /* merge server config */
    dir_cmds,                /* command table */
    dir_handlers,             /* handlers */
    NULL,                   /* filename translation */
    NULL,                   /* check_user_id */
    NULL,                   /* check auth */
    NULL,                   /* check access */
    NULL,                   /* type_checker */
    NULL,                   /* fixups */
    NULL,                   /* logger */
    NULL,                  /* header parser */
    NULL,                   /* child_init */
    NULL,                   /* child_exit */
    NULL                    /* post read-request */
};
```

mod_log_agent

agent_log_cmds

```
static const command_rec agent_log_cmds[] =
{
    {"AgentLog", set_agent_log, NULL, RSRC_CONF, TAKE1,
     "the filename of the agent log"},
    {NULL}
};
```

agent_log_module

```
 module agent_log_module =
{
    STANDARD_MODULE_STUFF,
    init_agent_log,           /* initializer */
    NULL,                    /* create per-dir config */
    NULL,                    /* merge per-dir config */
    make_agent_log_state,       /* server config */
    NULL,                    /* merge server config */
    agent_log_cmds,           /* command table */
    NULL,                    /* handlers */
    NULL,                    /* filename translation */
    NULL,                    /* check_user_id */
    NULL,                    /* check auth */
    NULL,                    /* check access */
    NULL,                    /* type_checker */
    NULL,                    /* fixups */
    agent_log_transaction,      /* logger */
    NULL,                    /* header parser */
    NULL,                    /* child_init */
    NULL,                    /* child_exit */
    NULL                     /* post read-request */
};
```

mod_log_referer

referer_log_cmds

```
static const command_rec referer_log_cmds[] =
{
    {"RefererLog", set_referer_log, NULL, RSRC_CONF, TAKE1,
     "the filename of the referer log"},
    {"RefererIgnore", add_referer_ignore, NULL, RSRC_CONF, ITERATE,
     "referer hostnames to ignore"},
    {NULL}
};
```

log_module

```
module referer_log_module =
{
    STANDARD_MODULE_STUFF,
    init_referer_log,          /* initializer */
    NULL,                     /* create per-dir config */
    NULL,                     /* merge per-dir config */
    make_referer_log_state,     /* server config */
    NULL,                     /* merge server config */
    referer_log_cmds,          /* command table */
    NULL,                     /* handlers */
    NULL,                     /* filename translation */
    NULL,                     /* check_user_id */
    NULL,                     /* check auth */
    NULL,                     /* check access */
    NULL,                     /* type_checker */
    NULL,                     /* fixups */
    referer_log_transaction,    /* logger */
    NULL,                     /* header parser */
    NULL,                     /* child_init */
    NULL,                     /* child_exit */
    NULL                      /* post read-request */
};
```

mod_mime

mime_cmds

```
static const command_rec mime_cmds[] =
{
    {"AddType", add_type, NULL, OR_FILEINFO, ITERATE2,
     "a mime type followed by one or more file extensions"},
    {"AddEncoding", add_encoding, NULL, OR_FILEINFO, ITERATE2,
     "an encoding (e.g., gzip), followed by one or more file extensions"},
    {"AddLanguage", add_language, NULL, OR_FILEINFO, ITERATE2,
     "a language (e.g., fr), followed by one or more file extensions"},
    {"AddHandler", add_handler, NULL, OR_FILEINFO, ITERATE2,
     "a handler name followed by one or more file extensions"},
    {"ForceType", ap_set_string_slot_lower,
     (void *)XtOffsetOf(mime_dir_config, type), OR_FILEINFO, TAKE1,
     "a media type"},
    {"RemoveHandler", remove_handler, NULL, OR_FILEINFO, ITERATE,
     "one or more file extensions"},
    {"SetHandler", ap_set_string_slot_lower,
     (void *)XtOffsetOf(mime_dir_config, handler), OR_FILEINFO, TAKE1,
     "a handler name"},
    {"TypesConfig", set_types_config, NULL, RSRC_CONF, TAKE1,
     "the MIME types config file"},
    {"DefaultLanguage", ap_set_string_slot,
     (void*)XtOffsetOf(mime_dir_config, default_language), OR_FILEINFO, TAKE1,
     "language to use for documents with no other language file extension" },
    {NULL}
};
```

mime_module

```
module MODULE_VAR_EXPORT mime_module =
{
    STANDARD_MODULE_STUFF,
    init_mime,                /* initializer */
    create_mime_dir_config,     /* dir config creator */
    merge_mime_dir_configs,     /* dir config merger */
    NULL,                    /* server config */
    NULL,                    /* merge server config */
    mime_cmds,                /* command table */
```

```
    NULL,                   /* handlers */
    NULL,                   /* filename translation */
    NULL,                   /* check_user_id */
    NULL,                   /* check auth */
    NULL,                   /* check access */
    find_ct,                /* type_checker */
    NULL,                   /* fixups */
    NULL,                   /* logger */
    NULL,                   /* header parser */
    NULL,                   /* child_init */
    NULL,                   /* child_exit */
    NULL                    /* post read-request */
};
```

mod_userdir

userdir_cmds

```
static const command_rec userdir_cmds[] = {
    {"UserDir", set_user_dir, NULL, RSRC_CONF, RAW_ARGS,
   "the public subdirectory in users' home directories, or 'disabled', or 'disabled
username username...', or 'enabled username username...'"},
    {NULL}
};
```

userdir_module

```
module userdir_module = {
    STANDARD_MODULE_STUFF,
    NULL,                   /* initializer */
    NULL,                   /* dir config creater */
    NULL,                   /* dir merger --- default is to override */
    create_userdir_config,      /* server config */
    NULL,                   /* merge server config */
    userdir_cmds,             /* command table */
    NULL,                   /* handlers */
    translate_userdir,         /* filename translation */
    NULL,                   /* check_user_id */
    NULL,                   /* check auth */
    NULL,                   /* check access */
    NULL,                   /* type_checker */
```

```
    NULL,                   /* fixups */
    NULL,                   /* logger */
    NULL,                   /* header parser */
    NULL,                  /* child_init */
    NULL,                   /* child_exit */
    NULL                    /* post read-request */
};
```

mod_so

so_cmds

```
static const command_rec so_cmds[] = {
    { "LoadModule", load_module, NULL, RSRC_CONF, TAKE2,
      "a module name and the name of a shared object file to load it from"},
    { "LoadFile", load_file, NULL, RSRC_CONF, ITERATE,
      "shared object file or library to load into the server at runtime"},
    { NULL }
};
```

so_module

```
module MODULE_VAR_EXPORT so_module = {
   STANDARD_MODULE_STUFF,
   NULL,         /* initializer */
   NULL,         /* create per-dir config */
   NULL,         /* merge per-dir config */
   so_sconf_create,     /* server config */
   NULL,         /* merge server config */
   so_cmds,          /* command table */
   NULL,         /* handlers */
   NULL,         /* filename translation */
   NULL,         /* check_user_id */
   NULL,         /* check auth */
   NULL,         /* check access */
   NULL,         /* type_checker */
   NULL,         /* fixer_upper */
   NULL,         /* logger */
   NULL,         /* header parser */
   NULL,         /* child_init */
   NULL,         /* child_exit */
```

```
    NULL          /* post read-request */
};
```

参考文献

1. RFC931: Authentication Server
2. RFC1321: The MD5 Message-Digest Algorithm
3. RFC1521: MIME: Mechanisms for Specifying and Describing the Format of Internet Message Bodies
4. RFC1630: Universal Resource Identifiers in WWW
5. RFC1738: Uniform Resource Locators (URL)
6. RFC1867: Form-based File Upload in HTML
7. RFC1945: Hypertext Transfer Protocol -- HTTP/1.0
8. RFC2068: Hypertext Transfer Protocol -- HTTP/1.1
9. RFC2069: An Extension to HTTP : Digest Access Authentication
10. RFC2109: HTTP State Management Mechanism
11. 《Web Server Technology》, Nancy J. Yeager and Robert E.McGrath，1996
12. 《CGI 编程指南》，蔡奇玉等
13. 《Servlet 简介》，板桥斋：http://www.banqiao.org/ JSDK
14. 《PC Magazine》，http://www.zdnet.com/
15. “Apache API”，http://www.apache.org/
16. “Using Apache JServ 1.0”，Ari Haberstadt，http://www.servletcentral.com/
17. “How to : Scalability - Load-Balancing - Fault tolerance”，Bernard Bernstein，Jean-Luc Rochat，http://java.apache.org/jserv/howto.load-balancing.html
18. “CGI 技术及其开发”，林辉，http://www.swm.com.cn/
19. “IEDnet 技术文摘：Web 的灵魂——新版 HTTP/1.1 协议草案介绍”，张恂
20. http://java.sun.com/
21. http://www.netcraft.com/Survey/
22. http://webcompare.internet.com/