超文本传输协议-HTTP/1.1（修订版）

---译者：孙超进
---email：sunchaojin@163.com

说明

本文档规定了互联网社区的标准组协议，并需要讨论和建议以便更加完善。请参考“互联网官方协议标准”（STD 1）来了解本协议的标准化状态。本协议不限流传发布。

版权声明

摘要

超文本传输协议（HTTP）是一种为分布式，协作式的，超媒体信息系统。它是一种通用的，无状态（stateless）的协议，除了应用于超文本传输外，它也可以应用于诸如名称服务器和分布对象管理系统之类的系统，这可以通过扩展它的请求方法，错误代码和消息头[47]来实现 。HTTP 的一个特性就是是数据表现形式是可以定义的和可协商性的，这就允许系统能独立于于数据传输被构建。

HTTP 在 1990 年 WWW 全球信息刚刚起步的时候就得到了应用。本说明书详细阐述了 HTTP/1.1 协议，是 RFC 2068 的修订版[33]。

目录（略）

1 引论

1.1 目的

超文本传输协议（HTTP）是一种为分布式的，协作的，超媒体信息系统，它是面向应用层的协议。在 1990 年 WWW 全球信息刚刚起步的时候 HTTP 就得到了应用。HTTP 的第一个版本叫做 HTTP/0.9,是一种为互联网原始数据传输服务的简单协议。由 RFC 1945[6]定义的 HTTP/1.0 进一步完善了这个协议。它允许消息以类 MIME 消息的格式传送，它包括传输数据的元信息和对请求/响应语义的修饰。但是，HTTP/1.0 没有充分考虑到分层代理，缓存的，以及持久连接和虚拟主机的需求的影响。并且随着不完善的 HTTP/1.0 应用程序的激增，这就迫切需要一个新的版本，以便能使两个通信程序能够确定彼此的真实能力。

此规范定义的协议叫做“HTTP/1.1”，.这个协议与 HTTP/1.0 相比，此规范更为严格，以确保各个协议的特征得到可靠实现。

实际的信息系统除了简单的获取信息之外，还要求更多的功能，包括查找（search），终端更新（front-end update）和注解（annotation）。HTTP 为请求提供可扩充方法集和消息头集[47]。HTTP 是建立在统一资源标识符（URI）[3]的约束上的，作为一个地址（URL）[4]或名称（URN）[20]，以指定被一个方法使用的资源。消息以一种类似于互联网邮件[9]消息格式来传输的，互联网消息格式定义于多目的互联网邮件扩展（MIME）[7]里。

HTTP 也是用于用户代理（user agents）和其它互联网系统的代理/网关之间通信的通信协议，这些互联网系统可能由 SMTP[16],NNTP[13],FTP[18]，Gopher[2]和 WAIS[10]协议支持。通过这种方式，HTTP 允许不同的应用程序对资源进行基本的超媒体访问。

1.2 要求

本文的关键词“必须”（"MUST"）,，“不能”（"MUST NOT"），“需要”（"REQUIRED"），“应该”（"SHALL"），“不应该”（"SHALL NOT"），“应该”（"SHOULD"），“不应该”（"SHOULD NOT"），“建议的”（"RECOMMENDED"），“可能”（"MAY"），和“可选的”（"OPTIONAL"）将由 RFC 2119[34]解释。

一个应用程序如果不能满足协议提供的一个或多个 MUST 或 REQUIRED 等级的要求，是不符合要求的。一个应用程序如果满足所有必须（MUST）或需要的（REQUIRED）等级以及所有应该（SHOULD）等级的要求，则被称为非条件遵循（unconditionally compliant）的；若满足所有必须（MUST）等级的要求但不能满足所有应该（SHOULD）等级的要求则被称为条件遵循的（conditionally compliant）。

1.3 术语

本说明用到了若干术语，以表示 HTTP 通信中各参与者和对象扮演的不同角色。

连接（connection）
为通信而在两个程序间建立的传输层虚拟电路。

消息（message）
HTTP 通信中的基本单元。它由一个结构化的八比特字节序列组成，与第 4 章定义的句法相匹配，并通过连接得到传送。

请求（request）
一种 HTTP 请求消息，参看第 5 章的定义。

响应（response）
一种 HTTP 响应消息，参看第 6 章的定义。

资源（resource）
一种网络数据对象或服务，可以用第 3.2 节定义的 URI 指定。资源可以以多种表现方式（例如多种语言，数据格式，大小和分辨率）或者根据其它方面而而不同的表现形式。

实体（entity）
实体是请求或响应的有效承载信息。一个实体包含元信息和内容，元信息以实体头域（entity-header field）形式表示，内容以消息主体（entity-body）形式表示。在第 7 章详述。

表现形式 （representation）
一个响应包含的实体是由内容协商（content negotiation）决定的。如第 12 章所述。有可能存在一个特定的响应状态码对应多个表现形式。

内容协商（content negotiation）
当服务一个请求时选择资源的一种适当的表示形式的机制（mechanism），如第 12 节所述。任何响应里实体的表现形式都是可协商的（包括错误响应）。

变量（variant）
在某个时刻，一个资源对应的表现形式（representation）可以有一个或多个（译注：一个 URI 请求一个资源，但返回的是此资源对应的表现形式，这根据内容协商决定）。每个表现形式（representation）被称作一个变量。 ‘变量’这个术语的使用并不意味着资源（resource）是由内容协商决定的.。

客户端（client）
为发送请求建立连接的程序.。

用户代理（user agent）
初始化请求的客户端程序。常见的如浏览器，编辑器，蜘蛛（可网络穿越的机器人），或其他的终端用户工具.

服务器（Server）
服务器是这样一个应用程序，它同意请求端的连接，并发送响应（response）。任何给定的程序都有可能既做客户端又做服务器；我们使用这些术语是为了说明特定连接中应用程序所担当的角色，而不是指通常意义上应用程序的能力。同样，任何服务器都可以基于每个请求的性质扮演源服务器，代理，网关，或者隧道等角色之一。

源服务器（Origin server）
存在资源或者资源在其上被创建的服务器（server）被成为源服务器（origin server）。

代理（ Proxy）
代理是一个中间程序，它既可以担当客户端的角色也可以担当服务器的角色。代理代表客户端向服务器发送请求。客户端的请求经过代理，会在代理内部得到服务或者经过一定的转换转至其他服务器。一个代理必须能同时实现本规范中对客户端和服务器所作的要求。透明代理（transparent proxy）需要代理认证和代理识别，而不修改请求或响应。非透明代理（non-transparent proxy）需修改请求或响应，以便为用户代理（user agent）提供附加服务，附加服务包括组注释服务，媒体类型转换，协议简化，或者匿名过滤等。除非透明行为或非透明行为经被显式地声明，否则，HTTP 代理既是透明代理也是非透明代理。

网关（gateway）
网关其实是一个服务器，扮演着代表其它服务器为客户端提供服务的中间者。与代理（proxy）不同，网关接收请求，仿佛它就是请求资源的源服务器。请求的客户端可能觉察不到它正在同网关通信。

隧道（tunnel）
隧道也是一个中间程序，它一个在两个连接之间充当盲目中继（blind relay）的中间程序。一旦隧道处于活动状态，它不能被认为是这次 HTTP 通信的参与者，虽然 HTTP 请求可能已经把它初始化了。当两端的中继连接都关闭的时候，隧道不再存在。

缓存（cache）
缓存是程序响应消息的本地存储。缓存是一个子系统，控制消息的存储、获取和删除。缓存里存放可缓存的响应（cacheable response）为的是减少对将来同样请求的响应时间和网络带宽消耗。任一客户端或服务器都可能含有缓存，但缓存不能存在于一个充当隧道（tunnel）的服务器里。

可缓存的（cacheable）
我们说响应（response）是可缓存的，如果这个响应可以被缓存（cache）保存其副本，为的是能响应后续请求。确定 HTTP 响应的缓存能力（cacheability）在 13 节中有介绍。即使一个资源（resourse）是可缓存的，也可能存在缓存是否能利用此缓存副本为某个特定请求的约束。

第一手的（first-hand）
如果一个响应直接从源服务器或经过若干代理（proxy），并且没有不必要的延时，最后到达客户端，那么这个响应就是第一手的（first-hand）。
如果响应通过源服务器（origin server）验证是有效性（validity）的，那么这个响应也同样是第一手的。

显式过期时间（explicit expiration time）
是源服务器认为实体（entity）在没有被进一步验证（validation）的情况下，缓存（cache）不应该利用其去响应后续请求的时间（译注：也就是说，当响应的显式过期时间达到后，缓存必须要对其缓存的副本进行重验证，否则就不能去利用此副本去响应后续请求）。

启发式过期时间（heuristic expiration time）
当没有显式过期时间（explicit expiration time）可利用时，由缓存指定过期时间.

年龄（age）
一个响应的年龄是从被源服务器发送或被源服务器成功验证到现在的时间。

保鲜寿命（freshness lifetime）
一个响应产生到过期之间的时间。

保鲜（Fresh）
如果一个响应的年龄还没有超过保鲜寿命（freshness lifetime），那么它就是保鲜的.。

陈旧（Stale）
一个响应的年龄已经超过了它的保鲜寿命（freshness lifetime），那么就是陈旧的.

语义透明（semantically transparent）
缓存（cache）可能会以一种语意透明（semantically transparent）的方式工作。这时，对于一个特定的响应，使用缓存既不会对请求客户端产生影响也不会对源服务器产生影响，缓存的使用只是为了提高性能。当缓存（cache）具有语意透明时，客户端从缓存接收的响应跟直接从源服务器接收的响应完全一致（除了使用 hop-by-hop 头域）。

验证器（Validator）
验证器其实是协议元素（例如：实体标签（entity tag）或最后修改时间（last-modified time）等），这些协议元素被用于识别缓存里保存的副本（即缓存项）是否等价于源服务器的实体的副本。

上游/下游（upstream/downstream）
上游和下游描述了消息的流动：所有消息都是从上游流到下游。

内向/外向（inbound/outbound）
内向和外向指的是消息的请求和响应路径：“内向”即“移向源服务器”，“外向”即“移向用户代理（user agent）”。

1.4 总体操作

HTTP 协议是一种请求/响应型的协议。 客户端给服务器发送请求的格式是一个请求方法

（request method），URI，协议版本号，然后紧接着一个包含请求修饰符（modifiers），客户端信息，和可能的消息主体的类 MIME（MIME-like）消息。服务器对请求端发送响应的格式是以一个状态行（status line），其后跟随一个包含服务器信息、实体元信息和可能的实体主体内容的类 MIME（MIME-like）的消息。其中状态行（status line）包含消息的协议版本号和一个成功或错误码。HTTP 和 MIME 之间的关系如附录 19.4 节所阐述。

大部分的 HTTP 通信是由用户代理（user agent）发起的，由应用于一个源服务器资源的请求构成。最简单的情形，这可以通过用户代理（UA）和源服务器（O）之间的单一连接（v）来实现。

请求链（Request chain）---------- ---------→
用户代理（UA）----------单一连接（v）----------源服务器（O）
<----------响应链（response chain）

有可能在请求/响应链中出现一个或多个中间者（intermediares），这是比较复杂的情形。常见的中间者（intermediares）有三种：代理（proxy），网关（gateway）和隧道（tunnel）。代理（proxy）是一种转发代理（a forwarding agent），它接收绝对 URI（absoulute url，相对于相对 url）请求，重写全部或部分消息，然后把格式化后的请求发送到 URI 指定的服务器上。网关是一种接收代理（receiving agent），它充当一个在服务器之上的层（layer），必要时它会把请求翻译成为下层服务器的协议。隧道不改变消息而充当两个连接之间的中继点；它用于通信需要穿过中间者（如防火墙）甚至当中间者不能理解消息内容的时候。

请求链（request chain）----------→
UA-----v-----A-----v-----B-----v-----C----------v----------O
<----------响应链（response chain）

上图显示了用户代理（user agent）和源服务器之间的三个中间者（A，B 和 C）。整条链的请求或响应将会通过四个被隔离开的连接。这个不同点很重要，因为某些 HTTP 通信选项有可能只能采用最近的非隧道邻接点的连接，有可能只采用链的端点（end-point），或者也有可能只采用于链上所有连接。图表尽管是线性的，每个参与者可能忙于多个并发的通信。例如，B 可以接收来自不是 A 的许多客户端的请求，并且 / 或者可以把请求转发到不是 C 的服务器，与此同时 C 正在处理 A 的请求。

通信中任何非隧道成员都可能会采用一个内部缓存（internal cache）来处理请求。如果沿着链的成员有请求已缓存的响应，请求/响应链就会大大缩短。下图阐明了一个最终请求响应链，假定 B 拥有一个来自于 O（通过 C）的以前请求响应的缓存副本，并且此请求的响应并未被 UA 或 A 缓存。

请求链（request chain）---------->
UA-----v----------A-----v-----B-----C----O
<---------响应链 （response chain）

并不是所有的响应都能有效地缓存，一些请求可能含有修饰符（modifiers），这些修饰符对缓存动作有特殊的要求。HTTP 对缓存行为（behavior）和可缓存响应（cacheable responses）的定义在第 13 章定义。

实际上，目前万维网上有多种被实践和部署的缓存和代理的体系结构和配置。这些系统包括节省带宽的缓存代理（proxy cache）层次（hierarchies）系统，可以广播（broadcast）或多播（multicast）缓存数据的系统，通过 CD-ROM 发布缓存数据子集的机构，等等。HTTP 系统（http system）会被应用于宽带连接的企业局域网中的协作，并且可以被用于 PDAs 进行低耗无线断续连接访问。HTTP1.1 的宗旨是为了支持各种各样的已经部署的配置，同时引进一种协议结构，让它满足可以建立高可靠性的 web 应用程序，即使不能达到这种要求，也至少可以可

靠的定位故障。

HTTP 通信通常发生在 TCP/IP 连接上。默认端口是 TCP 80，不过其它端口也可以使用。但并不排除 HTTP 协议会在其它协议之上被实现。HTTP 仅仅期望的是一个可靠的传输（译注：HTTP 一般建立在传输层协议之上）；所以任何提供这种保证的协议都可以被使用；协议传输数据单元（transport data unit）与 HTTP/1.1 请求和响应的消息结构之间的映象已经超出了本规范的范围。

大部分 HTTP/1.0 的实现都是对每个请求/响应交换（exchange）产生一个新的连接。而 HTTP/1.1 中，一个连接可以用于一个或更多请求/响应交换，虽然连接可能会因为各种原因中断（见第 8.1 节）。

2 符号习惯和一般语法

2.1 扩充的 BNF（扩充的 巴科斯-诺尔范式）

本文档规定的所有机制都用两种方法描述：散文体（prose）和类似于 RFC 822 的扩充 Backus-Naur Form（BNF）。要理解本规范，使用者需熟悉符号表示法。扩充 BNF 结构如下：

名字（name）=定义（definition）
名字（name）就是代表规则的名字，规则名里不能包含“<”和“>”，通过等号把规则名和规则定义（definiation）分离开。空格只有在采用延续行缩进来指定跨度多于一行的规则定义的时候才有意义。某些基本规则（basic rules）使用大写字母包含在规则定义里， 如 SP，LWS，HT，CRLF，DIGIT，ALPHA，等等。**尖括号**可以包含在规则定义里，只要它们的存在有利于区分规则名的使用。

“字面文本”（“literal”）
字面文本（literal text）两边用引号。除非声明，字面文本大小写不敏感（译注：如，HEX = "A" | "B" | "C" | "D" | "E" | "F" | "a" | "b" | "c" | "d" | "e" | "f" | DIGIT 里的 A，B,C，D 等等都是字面文本（literal text））。

规则 1 | 规则 2
由竖线（“|”）分开的元素是可选的，例如，“yes | no”表示 yes 或 no 都是可接受的。

(规则 1 规则 2)
围在括号里的多个元素视作一个元素。所以，“(elem (foo | bar) elem)”符合的字符串是“elem foo elem”和“elem bar elem”。

*规则
前面的字符“*”表示重复。完整的形式是“<n>*<m>元素”，表示元素至少出现<n>次，至多出现<m>次。默认值是 0 和无穷大，所以"*（元素）"允许任何数值，包括零；"1*元素"至少出现一次；"1*2element"允许出现一次或两次。

[规则]
方括号里是任选元素；“[foo bar]”相当于“*1（foo bar）”。

N 规则
特殊的重复：“<n>（元素）”与“<n>*<n>（元素）”等价；就是说，（元素）正好出现<n>次。这样 2DIGIT 是一个两位数字，3ALPHA 是一个由三个字符组成的字符串。

#规则
类似于“*”，结构“#”是用来定义一系列元素的。完整的形式是<n>#<m>元素，表示至少<n>个元素，至多<m>个元素，元素之间被一个或多个逗号（“,”）以及可选的线性空白（LWS）隔开了。这就使得表示列表这样的形式变得非常容易；像
(*LWS element *(*LWS ","*LWS element))
就可以表示为
1#element
无论在哪里使用这个结构，空元素都是允许的，但是不计入元素出现的次数。换句话说 ，“（element ）, ,（element） ”是允许的，但是仅仅视为两个元素。因此，在至少需要一个元素的地方，必须存在至少一个非空元素。默认值是 0 和无穷大，这样，“#element”允许任意零个或多个元素；“1# element”需要至少一个；“1#2element”允许一个或两个元素。

注释（comment）
用分号引导注释。

隐含的*LWS
本规范所描述的语法是基于字（word-based）的。除非特别注明，线性空白（LWS）可以出现在任何两个相邻字之间（标记（token）或引用字符串（quoted-string）），以及相邻字和间隔符之间，但是这并没有改变对一个域的解释。任何两个标记（token）之间必须有至少一个分割符，否则将会被理解为只是一个标记。

2.2 基本规则 （basic rule）

下面的规则贯穿于本规范的全文，此规则描述了基本的解析结构。US-ASCII（美国信息交换标准码）编码字符集是由 ANSI X3.4-1986[21]定义的。

```
OCTET（字节）      = <任意八比特的数据序列>
CHAR          = <任意 ASCII 字符（ascii 码值从 0 到 127 的字节）>
UPALPHA       = <任意大写字母"A"..."Z">
LOALPHA       = <任意小写字母"a"..."z">
ALPHA         = UPALPHA | LOALPHA
DIGIT         = <任意数字 0，1，...9>
CTL           = <任意控制字符（ascii 码值从 0 到 31 的字节）及删除键 DEL（127>
CR            = <US-ASCII CR, 回车（13）>
LF            = <US-ASCII LF, 换行符（10）>
SP            = <US-ASCII SP, 空格（32）>
HT            = <US-ASCII HT, 水平制表 （9）>
<">           = <US-ASCII 双引号（34）>
```

HTTP/1.1 将 CR LF 的序列定义为任何协议元素的行尾标志，但这个规定对实体主体（endtity-body）除外（要求比较松的应用见附录 19.3）。实体主体（entity-body）的行尾标志是由其相应的媒体类型定义的，如 3.7 节所述。

```
CRLF          = CR LF
```

HTTP/1.1 的消息头域值可以折叠成多行，但紧接着的被折叠行由空格（SP）或水平制表（HT）折叠标记开始。所有的线性空白（LWS）包括被折叠行的折叠标记（空格 SP 或水平制表键 HT），具有同 SP 一样的语义。接收者在解析域值并且将消息转送到下游（downstream）之前可能会将任何线性空白（LWS）替换成单个 SP（空格）。

```
LWS           = [CRLF] 1*(SP | HT)
```

下面的 TEXT 规则仅仅适用于头域内容和值的描述，不会被消息解释器解析。TEXT 里的字可以包含不仅仅是 ISO-8859-1[22]里的字符集，也可以包含 RFC 2047 里规定的字符集。

```
TEXT        = <除 CTLs 以外的任意 OCTET，但包括 LWS>
```

一个 CRLF 只有作为 HTTP 消息头域延续的一部分时才在 TEXT 定义里使用。

十六进制数字字符用在多个协议元素（protocol element）里。

```
HEX         = "A" | "B" | "C" | "D" | "E" | "F"
            | "a" | "b" | "c" | "d" | "e" | "f" | DIGIT
```

许多 HTTP/1.1 的消息头域值是由 LWS 或特殊字符分隔的字构成的。这些特殊字符必须先被包含在引用字符串（quoted string）里之后才能用于参数值（如 3.6 节定义）里。

```
token （标记）          = 1*<除 CTLs 与分割符以外的任意 CHAR >
separators（分割符）     = "(" | ")" | "<" | ">" | "@"
                        | "," | ";" | ":" | "\" | <">
                        | "/" | "[" | "]" | "?" | "="
                        | "{" | "}" | SP | HT
```

通过用圆括号括起来，注释（comment）可以包含在一些 HTTP 头域里。注释只能被包含在域值定义里有“comment”的域里。在其他域里，圆括号被视作域值的一部分。

```
comment （注释）= "(" *(ctext | quoted-pair | comment )” )"
ctext       = <除"(" 和 ")"以外的任意 TEXT >
```

如果一个 TEXT 若被包含在双引号里，则当作一个字。

```
quoted-string  =  （  <"> *（qdtext | quoted-pair ）  <"> ）
qdtext      = <any TEXT except <">>
```

斜划线（"\"）可以被作为单字符的引用机制，但是必须要在 quoted-string 和 comment 构造之内。

```
quoted-pair = "\" CHAR
```

3 协议参数

3.1 HTTP 版本

HTTP 使用一个“<major>.<minor>”数字模式来指明协议的版本号。为了进一步的理解 HTTP 通信，协议的版本号指示了发送端指明消息的格式和能力，而不仅仅是通过双方通信而获得的通信特性。当消息元素的增加不会影响通信行为或扩展了域值时，协议版本是不需要修改的。当协议会因为添加一些特征而做了修改时，<minor>数字就会递增。这些修改不会影响通常的消息解析算法，但它会给消息添加额外的语意（semantic）并且会暗示发送者具有额外的能力。协议的消息格式发生变化时，<major>数字就会增加。

HTTP 消息的版本在 HTTP-Version 域被指明，HTTP-Version 域在消息的第一行中。

HTTP-Version = "HTTP" "/" 1*DIGIT "." 1*DIGIT

注意 major 和 minor 数字必须被看成两个独立整数，每个整数都可以递增，并且可以增大到大于一位数的整数，如 HTTP/2.4 比 HTTP/2.13 低，而 HTTP/2.4 又比 HTTP/12.3 低。前导 0 必须被接收者忽略并且不能被发送者发送。

一个应用程序如果发送或响应消息里的包含 HTTP-Version 为“HTTP/1.1”的消息，那么此应用程序必须至少条件遵循此协议规范。最少条件遵循此规范的应用程序应该把“HTTP/1.1”包含在他们的消息的 HTTP-Version 里，并且对任何不兼容 HTTP/1.0 的消息也必须这么做。关于何时发送特定的 HTTP-Version 值的细节，参见 RFC2145[36]。

应用程序的 HTTP 版本是应用程序最少条件遵循的最高 HTTP 版本。

当代理或网关应用程序转发（forwarding）消息的协议版本不同于代理或网关应用程序本身协议版本的时候，代理（proxy）和网关（gateway）应用程序就要小心。因为消息里协议版本说明了发送者处理协议的能力，所以一个代理/网关千万不要发送一个高于该代理/网关应用程序协议版本的消息。如果代理或网关接收了一个更高版本的消息，它必须要么使协议的版本降低，要么以一个错误响应，或者要么切换到隧道行为（tunnel behavior）。

自从 RFC 2068[33]发布后，由于存在与 HTTP/1.0 代理（proxy）的互操作问题，所以缓存代理（caching proxies）必须能提升请求的版本到他们能支持的程度，但网关（gateway）可以这么做也可以不这么做，而隧道（tunnels）却不能这么做。代理（Proxy）/网关（gateway）的响应（Response）必须和请求（request）的主版本（major version）号保持一致。

注意：HTTP 版本间的转换可能会对消息头域（header fields）在版本里有或没有而进行改变。

3.2 通用资源标识符（URI）

URIs 有许多名字已为人所知：WWW 地址，通用文档标识符，通用资源标识符[3]，以及后来的统一资源定位器（URL）[4]和统一资源名称（URN）[20]。就 HTTP 而言，通用资源标识符（URI）只是简单的格式化字符串---通过名称，位置，或其它特征---识别一个资源。

3.2.1 一般语法

根据使用的背景，HTTP 里的 URIs 可以表示成绝对（absoulute）形式或相对形式（相对 URI 基于根 URI[11]）。两种形式的区别是根据这样的事实：绝对 URI 总是以一个模式（scheme）名作为开头，其后是一个冒号。关于 URL 更详尽的语法和含义请参看“统一资源标识符（URI）:一般语法和语义”，RFC 2396 [42]（代替了 RFCs 1738 [4]和 RFC 1808 [11]）。本规范采用了 RFC 2396 里的”URI-reference”， "absoluteURI"， "relativeURI"， "port"， "host"， "abs_path"， "rel_path", 和"authority"的定义格式。

HTTP 协议不对 URI 的长度作事先的限制，服务器必须能够处理任何他们提供资源的 URI，并且应该能够处理无限长度的 URIs，这种无效长度的 URL 可能会在客户端以基于 GET 方式的请求时产生。如果服务器不能处理太长的 URI 的时候，服务器应该返回 414 状态码（此状态码代表 Request-URI 太长）。

注:服务器在依赖大于 255 字节的 URI 时应谨慎，因为一些旧的客户或代理实现可能不支持这些长度。

3.2.2 HTTP URL

在 HTTP 协议里，http 模式（http scheme）被用于定位网络资源（resourse）的位置。本节定义了 http URLs 这种特定模式（scheme）的语法和语义。

```
http_URL = "http:" "//" host [ ":" port ] [ abs_path [ "?" query ]]
```

如果端口为空或未给出，就假定为 80。它的语义即：已识别的资源存放于正在监听 tcp 连接的那个端口的服务器上，并且请求资源的的 Request-UR 为绝对路径（5.1.2 节）。无论什么可能的时候，URL 里使用 IP 地址都是应该避免的（参看 RFC 1900 [24]）。如果绝对地址（abs_path）没有出现在 URL 里，那么应该给出"/"。如果代理（proxy）收到一个主机（host）名，但是这个主机名不是全称域名（fully quanlified domain name），则代理应该把它的域名加到主机名上。如果代理（proxy）接收了一个全称域名，代理不能改变主机（host）名称。

3.2.3 URI 比较

当比较两个 URI 是否匹配时，客户应该对整个 URI 比较时应该区分大小写，并且一个字节一个字节的比较。 但下面有些特殊情况：

- 一个为空或未给定的端口等同于 URI-refernece（见 RFC 2396）里的默认端口;

- 主机（host）名的比较必须不区分大小写;

- 模式（scheme）名的比较必须是不区分大小写的;

- 一个空绝对路径（abs_path）等同于"/"。

除了“保留（reserved）”和“不安全（unsafe）”字符集里的字符（参见 RFC 2396 [42]），其它字符和它们的"%HEXHEX"编码的效果一样。

例如,以下三个 URI 是等同的:

```
http://abc.com:80/~smith/home.html
http://ABC.com/%7Esmith/home.html
http://ABC.com:/%7esmith/home.html
```

3.3 日期/时间格式（Date/Time Formats）

3.3.1 完整日期 （Full Date）

HTTP 应用曾经一直允许三种不同日期/时间格式：

```
Sun, 06 Nov 1994 08:49:37 GMT  ; RFC 822, updated by RFC 1123
Sunday, 06-Nov-94 08:49:37 GMT ; RFC 850, obsoleted by RFC 1036
Sun Nov  6 08:49:37 1994       ; ANSI C's asctime（） format
```

第一种格式是作为 Internet 标准提出来的，它是一个国定长度的，由 RFC 1123 [8]（RFC 822[9]的升级版本）定义的一个子集。第二种格式使用比较普遍，但是基于废弃的 RFC 850 [12]协议，并且没有年份。如果 HTTP/1.1 客户端和服务器要解析日期，他们必须能接收所有三

种格式（为了兼容 HTTP/1.0），但是它们只能用 RFC 1123 里定义的日期格式来填充头域（header field）的值里用到 HTTP-date 的地方。

注:日期值的接收者被鼓励能可靠地接收来自于非 HTTP 应用程序发送的的日期值，例如有时可以通过代理（proxy）/网关（gateway）向 SMTP 或 NNTP 获取或提交消息。

所有的 HTTP 日期/时间都必须以格林威治时间（GMT）表示。对 HTTP 而言，GMT 完全等同于 UTC（世界协调时间）。前两种日期/时间格式里包含“GMT”，它是时区的三个字面的简写，并且当读到一个 asctime 格式时必须先被假定是 GMT 时间。HTTP 日期（HTTP-date）区分大小写，不能在此语法中除 SP 之外包含一个多余的 LWS。

```
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"
weekday      = "Monday" | "Tuesday" | "Wednesday"
                 | "Thursday" | "Friday" | "Saturday" | "Sunday"
month        = "Jan" | "Feb" | "Mar" | "Apr"
                 | "May" | "Jun" | "Jul" | "Aug"
                 | "Sep" | "Oct" | "Nov" | "Dec"
```

注意：HTTP 对日期/时间格式的要求仅仅应用在协议流的使用。客户和服务器不必把这种格式应用于用户呈现（user presentation），请求记录日志，等等。.

3.3.2 Delta Seconds （秒间隔）

一些 HTTP 头域（header field）允许时间值以秒为单位，以十进制整数表示，此值代表消息接收后的时间。

```
delta-seconds  = 1*DIGIT
```

3.4 字符集 （Character Sets）

HTTP 使用术语“字符集”的定义，这和 MIME 中所描述的是一样.

本文档中的术语“字符集”涉及到一种方法，此方法是用单个或多个表将一个字节序列转换成一个字符序列（译注：从这里来看，这应该是一种映射关系，表保存了映射关系）。注意在反方向上无条件的转换是不成立的，因为并不是所有的字符都能在一个给定的字符集里得到，一个字符集里可能提供多个字节序列表征一个特定的字符。这个定义为的是允许不同种类的字符编码从单一简单表映射（如 US-ASCII）到复杂表的转换方法，例如利用 ISO-2022 技术。然而，相关于 MIME 字符集名字的定义必须要充分指定从字节到字符的映射。特别是利用外部外围信息来精确确定映射是不允许的.

注：这里使用的术语“字符集”一般的被称作一种“字符编码”。不过既然 HTTP 和 MIME 在

同一机构注册，术语统一是很重要的。

HTTP 字符集的标记（token）是用不区分大小写的。所有的标记由 IANA 字符集注册机构[19]定义。

```
charset = token
```

尽管 HTTP 允许用任意标记（token）作为字符集（charset）值，但这个标记已经在 IANA 字符集注册机构注册过了，那么这个标记必须代表在该注册机构定义的字符集。对那些非 IANA 定义的字符集，应用程序应该限制使用。

HTTP 协议的实现者应该注意 IETF 字符集的要求[38][41].

3.4.1 丢失字符集（Missing Charset）

一些 HTTP/1.0 应用程序当他们解析 Content-Type 头时，当发现没有字符集参数（charset parameter，译注: Content-Type: text/plain; charset=UTF-8，此时 charset=UTF-8 就是字符集参数）可用时，这意味着接收者必须猜测实体主体（entity body）的字符集到底是什么。如果发送者希望避免这种情况，他应该在 Content-Type 头域里包含一个字符集参数，即使字符集是 ISO-8859-1 的也应该指明，这样就不会让接收者产生混淆。

不幸的是，一些旧的 HTTP/1.0 客户端不能处理在 Content-Type 头域里明确指定的字符集参数。HTTP/1.1 接收端必须要认真对待发送者提供的字符集；并且当用户代理（user agent，译注：如浏览器）开始呈现一个文档时，虽然用户代理可以猜测文档的字符集，但如果 content-type 头域里提供了字符集，并且用户代理也支持这种字符集的显示，不管用户代理是否愿意，它必须要利用这种字符集。参见 3.7.1 节。

3.5 内容编码（Content Codings）

内容编码（content coding）的值表示一种曾经或能被应用于一个实体的编码转换（encoding transformation）。内容编码主要用于文档的压缩或其它有效的变换，但这种变换必须不能丢失文档的媒体类型的特性，并且不能丢失文档的信息（译注：就像有损压缩和无损压缩，前者不会丢失信息，后者会丢失信息）。实体经常被编码后保存，然后传送出去，并且在接收端被解码。

```
content-coding   = token
```

所有内容编码（content-coding）的值是不区分大小写的。HTTP/1.1 在接受译码 （Accept-Encoding， 14.3 节）和内容译码（Content-Encoding）（14.11 节）头域里使用内容编码（content-coding）的值。尽管该值描述了内容编码，更重要的是它指出了一种解码机制，利用这种机制对实体的编码进行解码。

网络分配数字权威（ （IANA）充当内容编码的值标记（token）注册机构。最初，注册表里包含下列标记:

gzip（压缩程序）
一种由文件压缩程序"gzip"（GNU zip）产生的编码格式（在 RFC 1952 中描述）。这种编码格式是一种具有 32 位 CRC 的 Lempel-Ziv 编码（LZ77）。

compress（压缩）
一种由 UNIX 文件压缩程序"compress"产生的编码格式。这种编码格式是一种具有可适应性的 Lempel-Ziv-Welch 编码（LZW）。

对于将来的编码，用程序名识表征编码格式是不可取。在这里用到他们是因为他们在历史的作用，虽然这样做并不好。为了同以前的 HTTP 实现相兼容，应用程序应该将"x-gzip"和"x-compress"分别等同于"gzip"和"compress"。

deflate（缩小）
deflate 编码是由 RFC 1950 [31]定义的"zlib"编码格式与 RFC 1951 [29]里描述的"deflate"压缩机制的组合的产物。

identity（一致性）
Identity 是缺省编码；指明这种编码表明不进行任何编码转换。这种内容编码仅被用于接受译码（Accept-Encoding）头域里，但不能被用在内容译码（Content-Encoding）头域里。.

新的内容编码的值标记（token）应该被注册；为了实现客户和服务器间的互操作性，实现新值的内容编码算法规范应该能公开利用并且能独立实现，并且与本节中被定义的内容编码目的相一致。

3.6 传输编码 （Transfer Codings）

传输编码（transfer-coding ，译注：transfer coding 和和 transfer-coding 这两个术语在本协议规范里所表达的意思其实没什么太大区别，“transfer-coding”可能更能表达语意，因为它是规则中的规则名）的值被用来表示一个曾经，能够，或可能应用于一个实体的编码转换，传输编码是为了能够确保网络安全传输。这不同于内容编码（content coding），因为传输编码（transfer coding）是消息的属性而不是实体的属性。

```
transfer-coding = "chunked" | transfer-extension
transfer-extension      = token *（ ";" parameter ）
```

参数（parameter）采用属性/值对的形式.

```
parameter               = attribute  "=" value
attribute               = token
value                   = token  |   quoted-string
```

所有传输编码的值是大小写不敏感。传输编码的值在 TE 头域（14.39 节）和在传输译码（Transfer-encoding） 头域中（14.41 节）被运用。

无论何时，传输编码（transfer-coding）应用于一个消息主体（message body）时，如果存在多个传输编码，则这些传输编码中必须包括“块”（"chunked"）传输编码，除非通过关闭连接而使消息结束。当“块”（“chunked”）传输编码被用于传输编码时，它必须是应用于消息主体的最后传输编码。"块"（"chunked"）传输编码最多只能用于消息主体（message-body）一次。规定了上述规则后，接收者就可以确定消息的传输长度（transfer-length）（4.4 节）

传输编码与 MIME[7]的内容传输译码（Content-Transfer-Encoding，MIME [7]）的值相似，它被定义能够实现在 7 位传输服务上保证二进制数据的安全传输。不过，传输编码与内容传输译码（Content-Transfer-Encoding）对纯 8 位传输协议有不同的侧重点。在 HTTP 中，消息主体存在不安全的特性是因为有时候很难确定消息主体的长度（7.2.2 节）和在共享的传输上加密数据。

网络分配数字权威（IANA）担任注册传输编码的值标记（token）的角色。起初,注册包含如下标记:"块"（3.6.1 节）,"身份"（3.6.2 节）,"gzip"（3.5 节）,"压缩"（3.5 节）,和"缩小"（3.5 节）.

新的传输编码的值标记应该注册，这同新的内容编码的值标记也需要注册一样。.

如果接收端接收到一个经过传输编码编码过的实体主体（entity body）但它不能对这个编码后的实体主体进行解码，那么它应返回 501（不能实现），并且要关闭连接。服务器不能向 HTTP/1.0 客户端发送传输编码.。

3.6.1 块传输编码（Chunked Transfer Coding）

块编码（chunked encoding）改变消息主体使消息主体（message body）成块发送。每一个块有它自己的大小（size）指示器，在所有的块之后会紧接着一个可选的包含实体头域的尾部（trailer）。这种编码允许发送端能动态生成内容，并能携带能让接收端判断消息是否接收完整的有用信息。

```
Chunked-Body（块正文）      = *chunk（块）
                                last-chunk（最后块）
                            trailer（尾部）
                            CRLF
chunk（块）                  = chunk-size [ chunk-extension ] CRLF
                           chunk-data CRLF
chunk-size     = 1*HEX
last-chunk     = 1*（"0"） [ chunk-extension ] CRLF
chunk-extension= *（ ";" chunk-ext-name [ "=" chunk-ext-val ] ）
chunk-ext-name = token
chunk-ext-val  = token | quoted-string
chunk-data     = chunk-size（OCTET）
trailer        = *（entity-header CRLF）
```

chunk-size 是用 16 进制数字字符串。块编码（chunked encoding）以大小为 0 的块结束，紧接着是尾部（trailer），尾部以一个空行终止。

尾部（trailer）允许发送端在消息的末尾包含额外的 HTTP 头域（header field）。Trailer 头域（Trailer header field，在 14.40 节阐述）来指明哪些头域被包含在块传输编码的尾部（trailer） （见 14.40 节）

如果服务器要使用块传输编码进行响应，除非以下至少一条为真时它才能包含尾部（trailer）：

a）如果此响应的对应请求包括一个 TE 头域，并且利用 “trailers”指明了块传输编码响应的尾部是可以接受的（TE 头域在 14.39 节中描述；或者

b）如果是源服务器进行响应，响应里 trailer 字段里全部包含的是可选的元信息，并且接收端接收此块传输编码响应时可能不会理会响应的尾部（以一种源服务器是可以接受的方式）。换句话说，源服务器原意接受尾部（trailer）可能会在到达客户端时被丢弃的可能性。

当消息被一个 HTTP/1.1（或更高版本）的代理（proxy）接收并转发到一个 HTTP/1.0 接收端的时候，此要求防止了一种互操作性的失败。

在附录 19.4.6 节介绍了一个例子，这个例子介绍怎样对一个块主体（chunked-body）进行解码。

所有 HTTP/1.1 应用程序必须能接收和解码以块（chunked）传输编码进行编码的消息主体，并且必须能忽略它们不能理解的块扩展（chunk-extentsion）。

3.7 媒体类型（Media Type）

为了提供开放的，可扩展的数据类型和类型协商，HTTP 在 Content-Type（14.17 节）实体头域和 Accept 请求头域里利用了网络媒体[17]类型。

```
media-type      = type "/" subtype *（ ";" parameter ）
type            = token
subtype         = token
```

参数（parameter）以一种 属性/值（attribute/value）形式（如 3.6 节定义）跟随 类型/子类型（type/subtype）。

类型（type），子类型（subtype），和参数（parameter）里属性名称是大小写不敏感的。参数值有可能是大小写敏感的，也可能不是，这根据参数里属性名称的语意。线性空白（LWS）不能被用于类型（type）和子类型（subtype）之间，也不能用于参数的属性和值之间。参数的出现或不出现对处理媒体类型（media-type）可能会有帮助，这取决于它在媒体类型注册表里的定义。

注意一些旧的 HTTP 应用程序不能识别媒体类型的参数（parameter）。当向一个旧 HTTP 应用程序发送数据时，发送端只有在被 type/subtype 定义里需要时才使用类型参数（parameter）。

媒体类型（media-type）值需要被注册到网络数字分配权威（IANA[19]）里。媒体类型的注册程序在 RFC 1590[17]中大概描述。使用未经注册的媒体类型是不被鼓励的。

3.7.1 规范化和文本缺省 （Canonicalization and Text Defaults）

网络媒体类型以一种规范化格式被注册。一个实体主体（entity-body）通过 HTTP 消息传输，在传输前必须以一种合适的规范化格式来表示，但除了文本类型（text type），文本类型将会在下一段阐述。

当消息以一种规范化格式表现时，文本类型的子类型（subtype）会运用 GRLF 作为文本里的换行符。HTTP 放松了这个要求，允许文本媒体以一个 CR 或 LF 代表一个换行符传输，并且如果这样做的话就要贯穿整个实体主体（entity-body）。HTTP 应用程序必须能接收 CRLF，CR 和 LF 作为在文本媒体一个换行符。另外，如果文本里所属的字符集（character set）不能利用字节 13 和 10 来分别地表示 CR 和 LF，这是因为存在一些多字节字符集，HTTP 允许应用字符集里等价于 CR 和 LF 的字节序列来表示换行符。对换行符的灵活处理只能应用于实体主体里的文本媒体；在 HTTP 消息控制结构（如头域和多边界体（multipart boundaries））里，一个纯粹的 CR 或 LF 都不能代替 CRLF 的作用。

如果一个实体主体（entity-body）用内容编码（content-coding）进行编码，原始数据在被编码前必须是一种以上定义的媒体类型格式。.

"charset"参数（parameter）被应用于一些媒体类型，来定义数据的字符集（见 3.4 节）。当发送端没有指明 charset 参数（parameter）时，“text”类型的子媒体类型（subtype）被接收端接收后会被认为是缺省的 ISO-8859-1 字符集。非“ISO-8859-1”字符集和它的子类（subsets）的数据必须被指示恰当的字符集。3.4.1 节描述了兼容性问题。

3.7.2 多部分类型（Multipart type）

MIME 提供了一系列“多部分”（multipart）类型---在单个消息主体内包装一个或多个实体。所有的多部分类型共享一个公共的语法（这在 RFC 2046[40]的 5.1.1 节中描述），并且包含一个边界（boundary）参数作为多部分媒体类型的值的一部分。多部分类型的消息主体是一个协议元素，并且必须用 CRLF 来标识体部分（body-part，译注：见 RFC 2046 的 5 节）之间的换行。

不同于 RFC 2046 里的多部分消息类型的描述，HTTP1.1 规定任何多部分类型的消息尾声（epilogue，译：见 RFC 2046 对多部分消息类型的规则描述）必须不能存在；HTTP 应用程序不能传输尾声（epilogue）（即使原始的多部分消息尾部包含一个尾声）。存在这些限制是为了保护多部分消息主体的自我定界的特性，因为多部分边界的结束（译注：根据 RFC2046 中定义，多部分边界结束后可能还会有尾声）标志着消息主体的结束。

通常，HTTP 把一个多部分类型的消息主体（message-body）和任何其它媒体类型的消息主体等同对待：严格看作有用的负载体。有一个例外就是“ multipart/byterange”类型（附录 19.2），当它出现在 206（部分内容）响应时，此响应会被一些 HTTP 缓存机制解析，缓存机制将会在 13.5.4 节和 14.16 节介绍。在其它情况下，一个 HTTP 用户代理会遵循 MIME 用户代理一样或者相似的行为，这依赖于接收何种多部分类型。一个多部分类型消息的每一个体部分（body-part）里的 MIME 头域对于 HTTP 除了 MIME 语意并没有太大意义。

通常， 一个 HTTP 用户代理应该遵循与一个 MIME 用户代理相同或相似的行为。如果一个应用程序收到一个不能识别的多部分子类型，这个应用程序必须将它视为"multipart/mixed"。

注："multipart/form-data"类型已经被特别地定义用来处理 Post 请求方法传送的窗体数据，这在 RFC 1867[15]里定义。

3.8 产品标记 （product Tokens）

产品标记用于使通信应用软件能通过软件名称和版本来标识自己。很多头域都会利用产品标记，这些头域允许构成应用程序重要部分的子产品能以空白分隔去列举。通常为了识别应用程序，产品以应用程序的重要性的顺序来列举的。

```
product             = token  ["/" product-version]
product-version     = token
```

例:

```
User-Agent:CERN-LineMode/2.15 libwww/2.17b3
Server: Apache/0.8.4
```

产品标记应言简意赅。它们不能用来做广告或其他不重要的信息。虽然任一标记可能出现 product-version 里，但这个标记仅能用来做一个版本 （i.e., 同产品中的后续版本应该在 product-version 上有区别）

3.9 质量值（Quality Values）

HTTP 内容协商（content negotiation，12 节介绍）运用短“浮点”数字（short floating point number）来表示不同协商参数的相对重要性。重要性的权值被规范化成一个从 0 到 1 的实数。0 是最小值，1 是最大值。如果一个参数的质量值（quanlity value）为 0，那么这个参数的内容对客户端来说是不被接受。HTTP/1.1 应用程序不能产生多于三位小数的实数。下面规则限定了这

些值。

```
qvalue         = （ "0" [ "." 0*3DIGIT ] ）
                       | （ "1" [ "." 0*3（"0"） ] ）
```

"质量值" 是一个不当的用词，因为这些值仅仅表示相对等级。

3.10 语言标签 （Language Tags）

一个语言标签表征一种自然语言，这种自然语言能说，能写，或者被用来人与人之间的沟通。计算机语言明显不包括在内的。HTTP 在 Accept-Language 和 Content-Language 头域里应用到语言标签（language tag）。

HTTP 语言标签的语法和注册和 RFC 1766[1]中定义的一样。总之，一个语言标签是由一个部分或多部分构成：一个主语言标签和可能为空的多个子标签。

```
Language-tag      = primary-tag*（"-" subtag）
primary             = 1*8ALPHA
subtag            = 1*8ALPHA
```

标签中不允许出现空格，标签大小写不敏感（case-insensitive）。由 IANA 来管理语言标签中的名字。典型的标签包括:

```
en, en-US, en-cockney, i-cherokee, x-pig-latin
```

上面的任意两个字母的主标签是一个 ISO-639 语言的缩写，并且两个大写字母的子标签是一个 ISO-3166 的国家代码。（上面的最后三个标签是未经注册的标签；但是除最后一个之外所有的标签都会将来注册）。

3.11 实体标签 （Entity Tags）

实体标签被用于比较相同请求资源中两个或更多实体。HTTP/1.1 在 ETag（14.19 节），If-match（14.24 节），If-None-match（14.26 节）和 If-Rang（14.27 节）头域中运用实体标签。关于它们怎样被当作一个缓存验证器（cache validator）被使用和比较在 13.3.3 节被定义。一个实体标签由一个给定的晦涩引用字符串（opaque quoted string），还可能前面带一个弱指示器组成。

```
entity-tag = [ weak ] opaque-tag
weak       = "W/"
opaque-tag = quoted-string
```

一个“强实体标签”如果被一个资源的两个实体里共享，那么这两个实体必须在字节上等价。

一个“弱实体标签”是以"W/"前缀的，它可能会被一个资源的两个实体共享，如果这两个实体是等价的，并且能彼此替换，并且替换后也不会在语义上发生太大改变。一个弱实体标签只能用于弱比较（weak comparison）。

在一个特定资源的所有实体版本里，一个实体标签必须能唯一。一个给定的实体标签值可以被用于不同的 URI 请求的实体。相同实体标签的值应用于不同 URI 请求的实体，并不意味着这些实体是等价的。

3.12 范围单位（Range Units）

HTTP/1.1 允许客户请求响应实体的一部分。HTTP/1.1 在 Range（14.35 节）和 Content-Range（14.16 节）头域里应用范围单位（range units）。任何实体根据不同结构化单元都能被分解成子范围

range-unit = bytes-unit | other-range-unit
bytes-unit = "bytes"
other-range-unit = token

HTTP/1.1 中定义的唯一的范围单位是"bytes"。HTTP/1.1 实现可能忽略其他单位指定的范围。

HTTP/1.1 被设计允许应用程序实现不依赖于对范围的了解。

4 HTTP 消息

4.1 消息类型（Message Types）

HTTP 消息由从客户到服务器的请求消息和从服务器到客户的响应消息两部分组成.

HTTP-message = Request|Response ;HTTP/1.1

请求（第 5 节）和响应（第 6 节）消息利用 RFC 822[9]定义的常用消息的格式，这种消息格式是用于传输实体（消息的负载）。两种类型的消息都由一个开始行（start-line），零个或更多个头域（经常被称作“头”），一个指示头域结束的空行（也就是以一个 CRLF 为前缀的什么也没有的行），最后一个可有可无的消息主体（message-body）组成。

generic-message = start-line
*（message-header CRLF）
CRLF
[message-body]
start-line = Request-Line | Status-Line

为了健壮性，服务器应该忽略任意请求行（Request-Line）前面的空行。换句话说，如果服务器开始读消息流的时候发现了一个 CRLF，它应该忽略这个 CRLF。

一般一个存在问题的 HTTP/1.0 客户端会在 POST 请求消息之后添加额外的 CRLF。为了重新声明被 BNF 明确禁止的行为，一个 HTTP/1.1 客户端不能在请求前和请求后附加一些不必要的 CRLF。

4.2 消息头 （Message Headers）

HTTP 头域包括常用头域（4.5 节），请求头域（5.3 节），响应头域（6.2 节）和实体头域（7.1 节）。它们遵循的是 RFC822[0]3.1 节中给出的同一个常用格式。每一个头域由一个名字（域名）跟随一个":"和域值构成。域名是大小写不敏感的。域值前面可能有任意数量的 LWS 的。但 SP（空格）是首选的。头域能被延伸多行，这通过在这些行前面加一些 SP 或 HT。应用程序当产生 HTTP 消息时，应该遵循“常用格式”，因为可能存在一些应用程序，他们不能接收任何常用形式之外的形式。.

message-header = field-name ":" [field-value]

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, separators, and quoted-string>

filed-content 不包括任何前导或后续的 LWS（线性空白）：线性空白出现在域值（filed-value）的第一个非空白字符之前或最后一个非空白字符之后。前导或后续 LWS 可能会在不会改变域值语意情况下被删除。任何出现在 filed-content 之间的 LWS 可能在解析域值之前或把这个消息往下流传递时会被一个 SP 代替。.

不同域名的头域被接收的顺序是不重要的。然而，首先发送常用头域，然后紧接着是请求头域或者是响应头域，然后是以实体头域结束，这样做是一个好的的方法。

如果一个头域的域值被定义成一个以逗号隔开的列表，那么使用同一个域名（filed-name）的多个消息头域可能会出现在一些消息中。不改变消息的语义，可以把相同名的多个头域结合成一个“域名:域值”对的形式，这可以通过把每一个后续的域值加到第一个里，每一个域值用逗号隔开的算法实现。同名头域的接收顺序对合并的域值的解释是有重要意义的，所以代理（proxy）当把消息转发时不能改变域值的顺序。

4.3 消息主体 （Message Body）

HTTP 消息的消息主体用来承载请求和响应的实体主体（entity-body）的。这些消息主体（message-body）仅仅当被传输译码头域（Transfer-Encoding）指明的传输编码（transfer-coding）应用于实体主体（entity-body）时才和实体主体相区别，其它情况消息主体和实体主体相同。传输译码头域在 14.41 节阐述。

message-body=entity-body|<entity-body encoded as per Transfer-Encoding>

传输译码头域被用来指明应用程序的传输编码，它是为了保证消息的安全和合适的传输。传输译码（Transfer-Encoding）头域是消息的属性，而不是实体的属性，因此可能会沿着请求/响应链被添加或删除。（然而，3.6 节描述了一些限制当使用某个传输编码时）

什么时候消息主体（message-body）允许出现在消息中，这根据不同请求和响应来决定的。

请求中消息主体（message-body）的存在是被请求中消息头域中是否存在内容长度（Content-Length）或传输译码（Transfer-Encoding）头域来通知的。一个消息主体（message-body）不能被包含在请求里如果某种请求方法（见 5.1.1 节）不支持请求里包含实体主体（entity-body）。一个服务器应该能阅读或再次转发请求里的消息主体；如果请求方法不允许包含一个实体主体（entity-body），那么当服务器处理这个请求时消息主体应该被忽略。

对于响应消息，消息里是否包含消息主体依赖相应的请求方法和响应状态码。所有 HEAD 请求方法的请求的响应消息不能包含消息主体，即使实体头域出现在请求里。所有 1XX（信息的），204（无内容的）和 304（没有修改的）的响应都不能包括一个消息主体（message-body）。所有其他的响应必须包括消息主体，即使它长度可能为零。

4.4 消息的长度（Message Length）

当消息主体出现在消息中时，一条消息的传输长度（transfer-length）是消息主体（message-body）的长度；也就是说在实体主体被应用了传输编码（transfer-coding）后。当消息中出现

消息主体时，消息主体的传输长度（transfer-length）由下面（以优先权的顺序）决定：：

1。任何不能包含消息主体（message-body）的消息（这种消息如 1xx，204 和 304 响应和任何 HEAD 方法请求的响应）总是被头域后的第一个空行（CRLF）终止，不管消息里是否存在实体头域（entity-header fields）。

2。如果 Transfer-Encoding 头域（见 14.41 节）出现，并且它的域值是非”“dentity”传输编码值，那么传输长度（transfer-length）被“块”（chunked）传输编码定义，除非消息因为通过关闭连接而结束。

3。如果出现 Content-Length 头域（属于实体头域）（见 14.13 节），那么它的十进制值（以字节表示）即代表实体主体长度（entity-length，译注：实体长度其实就是实体主体的长度，以后把 entity-length 翻译成实体主体的长度）又代表传输长度（transfer-length）。Content-Length 头域不能包含在消息中，如果实体主体长度（entity-length）和传输长度（transfer-length）两者不相等（也就是说，出现 Transfer-Encodind 头域）。如果一个消息即存在传输译码（Transfer-Encoding）头域并且也 Content-Length 头域，后者会被忽略。

4。如果消息用到媒体类型“multipart/byteranges”，并且传输长度（transfer-length）另外也没有指定，那么这种自我定界的媒体类型定义了传输长度（transfer-length）。这种媒体类型不能被利用除非发送者知道接收者能怎样去解析它； HTTP1.1 客户端请求里如果出现 Range 头域并且带有多个字节范围（byte-range）指示符，这就意味着客户端能解析 multipart/byteranges 响应。

一个 Range 请求头域可能会被一个不能理解 multipart/byteranges 的 HTTP1.0 代理（proxy）再次转发；在这种情况下，服务器必须能利用这节的 1，3 或 5 项里定义的方法去定界此消息。

5。通过服务器关闭连接能确定消息的传输长度。（请求端不能通过关闭连接来指明请求消息体的结束，因为这样可以让服务器没有机会继续给予响应）。

为了与 HTTP/1.0 应用程序兼容，包含 HTTP/1.1 消息主体的请求必须包括一个有效的内容长度（Content-Length）头域，除非服务器是 HTTP/1.1 遵循的。如果一个请求包含一个消息主体并且没有给出内容长度（Content-Length），那么服务器如果不能判断消息长度的话应该以 400 响应（错误的请求），或者以 411 响应（要求长度）如果它坚持想要收到一个有效内容长度（Content-length）。

所有的能接收实体的 HTTP/1.1 应用程序必须能接受"chunked"的传输编码（3.6 节），因此当消息的长度不能被提前确定时，可以利用这种机制来处理消息。

消息不能同时都包括内容长度（Content-Length）头域和非 identity 传输编码。如果消息包括了一个非 identity 的传输编码，内容长度（Content-Length）头域必须被忽略.

当内容长度（Content-Length）头域出现在一个具有消息主体（message-body）的消息里，它的域值必须精确匹配消息主体里字节数量。HTTP/1.1 用户代理（user agents）当接收了一个无效的长度时必须能通知用户。

4.5 常用头域（General Header Fields）

有一些头域即适用于请求消息也适用于响应消息，但是这些头域并不适合传输实体。这些头域只能应用于传输消息。

general-header = Cache-Control ; Section 14.9

| Connection ; Section 14.10
| Date ; Section 14.18
| Pragma ; Section 14.32
| Trailer ; Section 14.40
| Transfer-Encoding ; Section 14.41
| Upgrade ; Section 14.42
| Via ; Section 14.45
| Warning ; Section 14.46

常用头域名能被扩展，但这要和协议版本的变化相结合。然而，如果通信里的所有参与者都认同新的或实践性的头域是常用头域，那么它们可能就被赋于常用头域的语意。不被识别的头域会被作为实体头（entity-header）头域来看待。

5 请求（Request）

一个请求消息是从客户端到服务器端的，在消息首行里包含方法，资源指示符，协议版本。

Request = Request-Line ; Section 5.1
*（（ general-header ; Section 4.5
| request-header ; Section 5.3
| entity-header ） CRLF） ; Section 7.1
CRLF
[message-body] ; Section 4.3

5.1 请求行 （Request-Line）

请求行（Request-Line）是以一个方法标记开始，后面跟随 Request-URI 和协议版本（HTTP-Version），最后以 CRLF 结束。元素是以 SP 字符分隔。除了最后的 CRLF，CR 或 LF 是不被允许的。

Request-Line =Method SP Request-URL SP HTTP-Version CRLF

5.1.1 方法 （Method）

方法标记（token）指明了在被 Request-URI 指定的资源上执行的方法。这种方法是大小写敏感的。

Method = "OPTIONS" ;9.2 节
| "GET" ;9.3 节
| "HEAD" ;9.4 节
|"POST" ;9.5 节
|"PUT" ;9.6 节
|"DELETE" ;9.7 节
|"TRACE" ;9.8 节
|"CONNECT" ;9.9 节
| extension-method

Extension-method = token

资源所允许的方法由 Allow 头域指定（14.7 节）。响应的返回码总是通知客户某个方法对当前

资源是否是被允许的，因为被允许的方法能被动态的改变。如果服务器能理解某方法但此方法对请求资源不被允许的，那么源服务器应该返回 405 状态码（方法不允许）；如果源服务器不能识别或没有实现某个方法，那么服务器应返回 501 状态码（没有实现）。方法 GET 和 HEAD 必须被所有一般的服务器支持。所有其它的方法是可选的；然而，如果上面的方法都被实现，这些方法遵循的语意必须和第 9 章指定的相同。

5.1.2 请求 URL（Request-URI）

Request-URI 是一种通用资源标识符（ 3 .2 节），并且它用于指定请求的请求资源。

Request-URI ="*" | absoluteURI | abs_path | authotity

Request-URI 的四个选项依赖于请求的性质。星号 “*”意味着请求不能应用于一个特定的资源，只能应用于服务器本身，并且只能在方法不应用于一个资源的时候才被允许。举例如下

OPTIONS * HTTP/1.1

当向代理（proxy）提交请求时，绝对 URI（absoluteRUI）格式是不可缺少的。代理（proxy）可能会被要求再次转发请求或者从一个有效的缓存（cache）里构造响应去响应请求。注意：代理可能转发请求给另一个代理或直接给被 absoluteURI 指定的源服务器。为了避免循环请求，代理（proxy）必须能识别所有的服务器名字，包括任何别名，本地的变化值，数字 IP 地址。一个请求行（Request-Line）的例子如下：

G ET http://www.w3.org/pub/www/TheProject.html HTTP/1.1

为了未来 HTTP 版本的所有请求能迁移到 absoluteURI 地址，所有基于 H TTP/1.1 的服务器必须接受绝对 absoluteURI 形式的请求，即使 HTTP/1.1 客户端只为代理产生绝对 absoluteURI 请求。

authority 格式只被用于 CONNECT 方法（ 9 .9 节）.

Request-URI 大多数情况是被用于指定一个源服务器或网关（gateway）上的资源。这种情况下，URI 的绝对路径（abs_path，见 3.2.1 节）作为 Request-URI 被传输，并且 URI 网络位置（authority）必须在 Host 头域里指出。例如：客户希望直接从源服务器获取资源，这种情况下，它可能会建立一个 TCP 连接，此连接是特定于主机 “www.w3.org”的 80 端口的，这时会发送下面行：

GET /pub/WWW/TheProject.html HTTP/1.1
Host：www.w3.org

接下来是请求的其他部分。注意绝对路径（absolute path）不能是空的；如果在原始 URI 里没有出现绝对路径，必须给出" / "（服务器的根）。

Request-URI 是以 3.2.1 节里指定的格式传输。如果 Request-URI 用"%HEX HEX"[42]编码，源服务器为了解析请求必须能对它进行解码。服务器接收到一个无效的 Request-URI 时必须以一个合适的状态码响应。

透明代理（proxy）不能重写接收到的 Request-URI 里的 “abs_path”部分，当它转发此请求到下一个内向服务器（inbound server，见 1.3 术语）时，但除了上面说的把一个空的 abs_path 用一个 ”/”代替之外。

注：不重写的规则是为了防止代理（proxy）改变请求的原意，因为存在源服务器利用非保留

（non-reserved）的 URI 字符为相反的目的 。实现者应该知道某些低于 HTTP/1.1 版本的代理（proxy）会重写 Request-URI。

5.2 请求资源的识别 （The Resource Identified by a Request）

请求资源的精确定位是由请求里的 Request-URI 和 Host 头域决定的。

如果源服务器不允许资源根据请求的不同主机来区分时，那么它可以会忽略 Host 头域的值，当它决定通过 HTTP/1.1 请求来识别资源的时候。（在 HTTP/1.1 里关于对 Host 的支持在 19.6.1.1 节描述了其他的要求）。

一个源服务器如果必须基于请求主机来区分资源（这是因为存在虚拟主机（virtual hosts）和虚拟主机名（vanity host names））， 那么， 对 HTTP/1.1 请求，它必须遵循下面的规则去决定请求的资源：

1． 如果 Request-URI 是绝对地址（absoluteURI），那么主机（host）是 Request-URI 的一部分。任何出现在请求里 Host 头域的值应当被忽略。

2． 假如 Request-URI 不是绝对地址（absoluteURI），并且请求包括一个 Host 头域，则主机（host）由该 Host 头域的值决定.

3． 假如由规则 1 或规则 2 定义的主机（host）对服务器来说是一个无效的主机（host），则应当以一个 400（坏请求）错误消息返回。

缺少 Host 头域的 HTTP/1.0 请求的接收者可能会尝试去利用启发式的规则（例如：检查 URI 路径是否是特定某个主机）去决定被请求的真正资源。

5.3 请求头域 （Request Header Fields）

请求头域允许客户端传递请求的附加信息和客户端自己的附加信息给服务器。这些头域作为请求的修饰，这和程序语言方法调用的参数语义是等价的。

```
请求头（request-header） = Accept           ;14.1 节
                         | Accept-Charset    ;14.2 节
                          |Accept-Encoding     ;14.3 节
                          |Accept-Language     ;14.4 节
                          |Authorization       ;14.8 节
                          |Expect              ;14.20 节
                          |From          ;14.22 节
                          |Host            ;14.23 节
                          |If-Match        ;14.24 节
                          |If-Modified-Since     ;14.25 节
                          | If-None-Match        ;14.26 节
                          | If-Range             ;14.27 节
                          | If-Unmodified-Since  ;14.28 节
                          | Max-Forwards        ;14.31 节
                          | Proxy-Authorization  ;14.34 节
                          | Range           ;14.35 节
                          | Referer         ;14.36 节
                          |TE                ;14.39 节
```

| User-Agent ;14.43 节

请求头域的名字是能被扩展，但只能随协议版本改变而被扩展。然而新的或实践性的头域可以给定请求头域语义，如果所有通信方都把它看作请求头域。不能识别的头域会被看作实体头域（entity-header）。

6 响应 （Response）

接收和解析一个请求消息后，服务器发出一个 HTTP 响应消息。

```
response   =Status-Line                    ;6.1 节
        *（（ general-header）                              ; 4.5 节
                    | response-header    ;6.2 节
                    | entity-header）CRLF）     ;7.1 节
                   CRLF
                   [ message-body ]       ;7.2 节
```

6.1 状态行 （Status-Line）

响应消息的第一行是状态行（stauts-Line），由协议版本以及数字状态码和相关的文本短语组成，各部分间用空格符隔开，除了最后的 CRLF 序列，中间不允许有 CR 或 LF。

```
Status-Line = HTTP-Version SP Status-Code SP Reason-Phrase CRLF
```

6.1.1 状态码与原因短语 （Status Code and Reason Phrase）

Status-Code 元素是一个试图理解和满足请求的三位数字整数码，这些码的完整定义在第十章。原因短语（Reason-Phrase）是为了给出关于状态码的简单的文本描述。状态码用于控制，而原因短语（Reason-Phrase）是让用户便于阅读。客户端不需要检查和显示原因短语。

状态码的第一位数字定义响应类别。后两位数字没有任何分类角色。第一位数字有五种值：

-1xx ： 报告的 -请求被接收到，继续处理
-2xx ： 成功 - 被成功地接收（received），理解（understood），接受（accepted）的动作 。
-3xx ： 重发 - 为了完成请求必须采取进一步的动作。
-4xx ： 客户端出错 - 请求包括错的语法或不能被满足。
-5xx ： 服务器出错 - 服务器无法完成显然有效的请求。

下面列举了为 HTTP/1.1 定义的态码值，和对应的原因短语（Reason-Phrase）的例子。原因短语在这里例举只是建议性的----它们也许被一个局部的等价体代替而不会影响此协议的语义。

```
Status-Code ＝
        "100" ; 10.1.1 节: 继续
       |"101"  ; 10.1.2 节: 转换协议
       |"200"  ; 10.2.1 节: OK
       |"201"  ; 10.2.2 节:  已创建
        |"202"  ;  10.2.3 节:  接受
       |"203"  ;  10.2.4 节: 非权威信息
```

|"204" ; 10.2.5 节: 无内容
|"205" ; 10.2.6 节: 重置内容
|"206" ; 10.2.7 节: 部分内容
|"300" ; 10.3.1 节: 多个选择
|"301" ; 10.3.2 节: 永久移动
|"302" ; 10.3.3 节: 发现
|"303" ; 10.3.4 节: 见其它
|"304" ; 10.3.5 节: 没有被改变
|"305" ; 10.3.6 节: 使用代理
|"307" ; 10.3.8 节 临时重发
|"400" ; 10.4.1 节: 坏请求
|"401" ; 10.4.2 节: 未授权的
|"402" ; 10.4.3 节: 必要的支付
|"403" ; 10.4.4 节: 禁用
|"404" ; 10.4.5 节: 没有找到
|"405" ; 10.4.6 节: 方式不被允许
|"406" ; 10.4.7 节: 不接受的
|"407" ; 10.4.8 节: 需要代理验证
|"408" ; 10.4.9 节: 请求超时
|"409" ; 10.4.10 节; 冲突
|"410" ; 10.4.11 节: 不存在
|"411" ; 10.4.12 节: 长度必需
|"412" ; 10.4.13 节；先决条件失败
|"413" ; 10.4.14 节: 请求实体太大
|"414" ; 10.4.15 节; 请求 URI 太大
|"415" ; 10.4.16 节: 不被支持的媒体类型
|"416" ; 10.4.17 节: 请求的范围不满足
|"417" ; 10.4.18 节: 期望失败
|"500" ; 10.5.1 节: 服务器内部错误
|"501" ; 10.5.2 节: 不能实现
|"502" ; 10.5.3 节: 坏网关
|"503" ; 10.5.4 节: 服务不能获得
|"504" ; 10.5.5 节: 网关超时
|"505" ; 10.5.6 节: HTTP 版本不支持
|扩展码

extension-code =3DIGIT
Reason-Phrase = *<TEXT,excluding CR,LF>

HTTP 状态码是可扩展的。HTTP 应用程序不需要理解所有已注册状态码的含义，尽管那样的理解是很希望的。但是，应用程序必须了解由第一位数字指定的状态码的类别，任何未被识别的响应应被看作是那个类别的 x00 状态码，未被识别的响应不能被缓存除外。例如，如果客户端收到一个未被识别的状态码 431，则可以安全的认为请求有错，并且它会对待此响应就像它接收了一个状态码是 400 的响应。在这种情况下，用户代理（user agent）应当把响应的实体展现给用户，因为实体有可能包括人类可读的信息，这些信息也许能解释非正常状态的原因。

6 .2 响应头域 （Response Header Fields）

响应头域允许服务器传送响应的附加信息，这些信息不能放在状态行（Status-Line）里.。这些头域给出有关服务器的信息以及请求 URI（Request-URI）指定资源的更进一步访问信息。

```
response-header  =  Accept-Ranges          ; 14.5 节
                   |Age          ; 14.6 节
                   |Etag           ; 14.19 节
                   |Location          ; 14.30 节
                   |Proxy-Autenticate     ; 14.33 节
                   |Retry-After     ; 14.37 节
                   |Server      ; 14.38 节
                   |Vary        ; 14.44 节
                   |WWW-Authenticate  ; 14.47 节
```

响应头域的名字能依赖于协议版本的变化而扩展。然而，新的或者实践性的头域可能会给予响应头域的语义如果通信所有成员都能识别他们并把他们看作响应头域。不被识别的头域被看作实体头域。

7 实体（Entity）

如果不被请求方法或响应状态码所限制，请求和响应消息都可以传输实体。 实体包括实体头域（entity-header）与实体主体（entity-body），而有些响应只包括实体头域（entity-header）。

在本节中的发送者和接收者是否是客户端或服务器，这依赖于谁发送或谁接收此实体。

7.1 实体头域（Entity Header Fields）

实体（entity-header）头域定义了关于实体主体的的元信息，或在无主体的情况下定义了请求的资源的元信息。有些元信息是可选的；一些是必须的。

```
entity-header = Allow ; Section 14.7
       | Content-Encoding ; Section 14.11
       | Content-Language ; Section 14.12
       | Content-Length ; Section 14.13
       | Content-Location ; Section 14.14
       | Content-MD5 ; Section 14.15
       | Content-Range ; Section 14.16
       | Content-Type ; Section 14.17
       | Expires ; Section 14.21
       | Last-Modified ; Section 14.29
       | extension-header

extension-header = message-header
```

扩展头机制允许在不改变协议的前提下定义额外的实体头域，但不保证这些域在接收端能够被识别。未被识别的头域应当被接收者忽略，且必须被透明代理（transparent proxy）转发。

7.2 实体主体（Entity Body）

由 HTTP 请求或响应发送的实体主体（如果存在的话）的格式与编码方式应由实体的头域决定。

```
Entity-body= *OCTET
```

如 4。3 节所述，实体主体（entity-body）只有当消息主体存在时时才存在。实体主体（entity-body）从消息主体根据传输译码头域（Transfer-Encoding）解码得到，传输译码用于确保消息的安全和合适传输。

7.2.1 类型（Type）

当消息包含实体主体（entity-body）时，主体的数据类型由实体头域的 Content-Type 和 Content-Encoding 头域确定。这些头域定义了两层顺序的编码模型：

Entity-body：=Content-Encoding（ Content-Type（ data） ）

Content-Type 指定了下层数据的媒体类型。Content-Encoding 可能被用来指定附加的应用于数据的内容编码，经常用于数据压缩的目的，内容编码是请求资源的属性。没有缺省的编码。

任一包含了实体主体的 HTTP/1.1 消息都应包括 Content-Type 头域以定义实体主体的媒体类型。如果只有媒体类型没有被 Content-Type 头域指定时，接收者可能会尝试猜测媒体类型，这通过观察实体主体的内容并且/或者通过观察 URI 指定资源的扩展名。如果媒体类型仍然不知道，接收者应该把类型看作”application/octec-stream”。

7.2.2 实体主体长度（Entity Length）

消息的实体主体长度指的是消息主体在被应用于传输编码（transfer-coding）之前的长度。4.4 节定义了怎样去确定消息主体的传输长度。

8 连接

8.1 持久连接（Persistent Connection）。

8.1.1 目的

在没有持久连接之前，为获取每一个 URL 指定的资源都必须建立了一个独立的 TCP 连接，这就加重了 HTTP 服务器的负担，易引起互联网的阻塞。嵌入图片与其它相关数据通常使用户在短时间内对同一服务器进行多次请求。目前针对这些性能问题的分析以及一个原型实现的结果是可以获得的[26][30]。 HTTP/1.1（RFC2068）实现的实践过程和度量展现了一个满意的结果。对其他方法也有了初步探究，如 T/TCP [27]。

HTTP 持久连接有着诸多的优点：

--- 通过建立与关闭较少的 TCP 连接，不仅节省了路由器与主机（客户端，服务器，代理，网关，隧道或缓存）的 CPU 时间，还节省了主机用于 TCP 协议控制块（TCP protocol control blocks）的内存。

--- HTTP 请求和响应能在连接上进行管线请求方式。 管线请求方式能允许客户端执行多次请求而不用等待每一个请求的响应（译注：即客户端可以发送请求而不用等待以前的请求的响应到来后再发请求），并且此时只有一个 TCP 连接被使用，从而提高了效率，减少了时间。

--- 网络阻塞会被减少，这是由于减少了因 TCP 连接产生的包的数量并且也由于允许 TCP 有充

分的时间去决定网络阻塞的状态。

--- 因为无须在创建 TCP 连接时的握手上耗费时间，而使后续请求的等待时间减少。

---HTTP 改进的越来越优雅，因为报告错误不需要关闭 tcp 连接的代价。将来的 HTTP 版本客户端可以乐观的尝试利用一个新特性，但是如果和老服务器通信时错误被报告，那么就要用旧的语义进行重新尝试。

HTTP 实现应该实现持久连接。

8.1.2 总体操作

HTTP/1.1 与早期 HTTP 版本的一个显著区别在于持久连接是 HTTP/1.1 的缺省方式。也就是说，除非另有指定，客户端总应当假定服务器会保持持久连接，即便在收到服务器的出错响应时也应如此。

持久连接提供了一种可以由客户端或服务器通知终止 TCP 连接的机制。利用 Connection 头域（14。10 节）可以产生终止连接信号。一旦出现了终止连接的信号，客户端便不可再向此连接提出任何新请求。

8.1.2.1 协商（Negotiation）

除非请求里 Connection 头域中包含“close”连接标记（connection-token），HTTP/1.1 服务器总可以认为 HTTP/1.1 客户端想要维持持久连接（persistent connection）。如果服务器想在发出响应后立即关闭连接，它应当发送一个含“close”的 Connection 头域。

一个 HTTP/1.1 客户端可能期望连接一直保持开着，但这必须是基于服务器响应里是否包含一个 Connection 头域并且此头域里是否包含“close”。如果客户端不想再继续维持连接来发送更多请求，那么它应发送一个值为“close”的 Connection 头域。

如果客户端或服务器中的任一方在 Connection 头域里包含“close”，那么那个请求就成为这个连接的最后一个请求。

客户端和服务器不应认为持久连接是低于 1.1 的 HTTP 版本所拥有的，除非它被显式地指明 。19.6.2 节指出了跟 HTTP/1.1 客户端兼容的更多信息。

8.1.2.2 管线（pilelining）

支持持久连接（persistent conncetion）的客户端可以以管线的方式发送请求（即无须等待响应而发送多个请求）。服务器必须按接收请求的顺序发送响应。

假定以持久连接方式进行连接，并且假定在连接建立后进行管线方式请求的客户端应该准备去重新尝试连接如果首次管线请求方式尝试失败。如果客户端重新去尝试连接，那么，只有在客户端知道连接是持久连接之后，客户端才能进行管线发送请求。如果服务器在响应所有对应的请求之前关闭了连接，客户端必须准备去重新发送请求。

客户端不应该利用非等幂的方法或者非等幂的方法序列（见 9.1.2 节）进行管线方式的请求。否则一个过早的传输层连接的终止可能会导致不确定的结果。希望发送非等幂方法请求的客户端

只有接收了上次它发出请求的响应后才能再次发送请求给服务器。

8.1.3 代理 （Proxy Servers）

代理是非常重要的，因为代理正确地实现了 Connection 头域的属性，这在 14.10 节指出了。

代理必须分别向它相连的客户端或源服务器（或其他的代理）指明持久连接。每一个持久连接只能应用于一个传输层连接。

代理不能和一个 HTTP/1.0 客户端建立一个 HTTP/1.1 持久连接（但是参见 RFC2068[33]里的关于许多 HTTP/1.1 客户端利用 Keep-Alive 头域的问题的讨论）。

8.1.4 实际考虑 （Practical Considerations）

服务器通常有一个时限值，超过一定时间即不再维持处于非活动的连接。代理会选一个较高的值，因为客户端很可能会与同一服务器建立多个连接。持久连接方式的采用对于客户端与服务器来说均未提出任何必须存在超时或给定多长时间的要求。

当客户端或服务器希望超时时， 它应该优雅的去关闭传输连接。客户端与服务器端应始终注意对方是否终止了传输层连接，并适当的予以响应。若客户端或服务器未能及时检测到对方已终止了连接，将会造成不必要的网络资源浪费。

客户端，服务器，或代理可能在任意时刻会终止传输连接。比如，客户端可能正想发出新的请求，而此时服务器却决定关闭“闲置”的连接。在服务器看来，连接已经因为闲置被关闭了，但客户端认为我正在请求。

这表明客户端，服务器与代理必须有能力从连接的异步终止事件中恢复。只要请求是等幂的（见 9.1.2 节），客户端软件应该能重新打开传输层连接并重试传输遗弃的请求序列而不需要用户进行去交互。对非等幂方法的请求就不能自动重试请求，尽管用户代理（user agent）可能提供一个人工操作去重试这些请求。用户代理（user-agent）对应用程序语义理解的确认应该替代用户的确认。如果再次重试请求序列失败，那么就不能再进行自动重试请求了。

服务器尽可能应该至少在每次连接中响应一个请求。除非出于网络或客户端的故障，服务器不应在传送响应的中途断开连接。

使用持久连接的客户端应限制与某一服务器同时连接的个数。单用户客户端不应与任一服务器或代理保持两个以上的连接。代理应该与其它服务器或代理之间维护 2*N 个连接，其中 N 是同时在线的用户数。这些准则是为了改善响应时间和和避免阻塞。

8.2 消息传送的要求（Message Transmission Requirements）

8.2.1 持久连接与流量控制 （Persistent Connections and Flow Control）

HTTP/1.1 服务器应保持持久连接并使用 TCP 流量控制机制来解决临时过载，而不是在终止连接后指望客户端的重试。后一方法会恶化网络阻塞。

8.2.2 监视连接中出错状态的消息

HTTP/1.1（或更新）客户端应该当在发送请求的消息主体时同时监视网络连接是否处于出错状态。若客户端发现了错误，它应当立即停止消息主体的的传送。若消息主体是以块传输编码方式发送的（3.6 节），可以用长度为零的块和空尾部（trailer）来提前标记报文结束。若消息主体之前有 Content-Length 头域，那么客户端必须关闭连接。

8.2.3 100 状态码的用途

100 状态码（继续，见 10.1.1 节）的目的在于允许客户端，在发送此请求消息主体前，判定服务器是否愿意接受发送消息的主体（基于请求的头域）。 在有些情况下，如果服务器拒绝查看消息主体，这时客户端发送消息主体是不恰当的或会大大降低效率。

HTTP/1.1 客户端的要求：

--- 若客户端想要在发送请求消息主体之前等候 100（继续）响应，则它必须发送一个 Expect 请求头域（见 14.20 节），并且值是“100-continue”。

--- 客户端不能发送一个值是“100-continue”的 Expect 请求头域，如果此客户端不打算发送带消息主体的请求。

由于存在旧实现，协议允许二义性的情形存在，这在客户端在发送“Expect:100-continue”后而不一定要接收一个 417（期望失败）状态码或着 100（继续）状态码时。因此，当一个客户端发送 Expect 请求头域给一个源服务器（可能通过代理），此服务器也没有以 100（继续）状态码响应，那么客户端不应该在发送请求消息的主体前无限等待。

HTTP/1.1 源服务器的要求：

--- 当接收一个包含值为“100-contitue”的 Expect 请求头域的请求时，源服务器必须或者以 100（继续）状态码响应从而继续从输入流里接收数据，或者以一个最终的状态码响应。源服务器不能在发送 100（继续）状态码响应之前接收请求主体。如果服务器以一个终结状态码响应后，它可能会关闭传输层连接或者它也可能会继续接收或遗弃剩余的请求。但是既然它返回了一个终结状态码的响应，它就不能再去执行那个请求的方法（如：POST 方法，PUT 方法）。

--- 如果请求消息不含值为“100-continue”的 Expect 请求头域， 源服务器不应发送 100（继续）响应，并且，当请求来自 HTTP/1.0（或更早）的客户端时，服务器也不得发送 100（继续）响应。对此规定有一例外：为了与 RFC 2068 兼容，源服务器可能会发送一个 100（继续）状态响应以响应 HTTP/1.1 的 PUT 或 POSt 请求，虽然这些请求中没有包含值为”100-continue”的 Expect 请求头域。这个例外的目的是为了减少任何客户端因为等待 100（继续）状态响应的延时，但此例外只能应用于 HTTP/1.1 请求，并不适合于其他 HTTP 版本的请求。

--- 若源服务器已经接收到部分或全部请求的消息的主体，源服务器可以不需要发 100（继续）响应。

---一旦请求消息主体被接收和被处理，发送 100（继续）响应的源服务器必须最终能发送一个终结状态响应，，除非源服务器过早切断了传输层连接。

--- 若源服务器接收到不含值为“100-contitue”的 Expect 请求头域的请求，但该请求含有请求消息主体，而服务器在从传输层连接上接收整个请求消息主体前返回一个终结状态响应，那么此源服务器不能关闭传输层连接直到它接收了整个请求或者直到客户端关闭了此连接。否则客

户端可能不会信任接收此响应消息。然而，这一要求不应该被解释为防止服务器免受拒绝服务攻击，或者防止服务器被客户端攻击。

对 HTTP/1.1 代理的要求：

--- 若代理接到一个请求，此请求包含值为“100-continue”的 Expect 请求头域，并且代理可能不能确定下一站服务器是否遵循 HTTP/1.1 或更高版协议，那么它必须转发此请求时包含此 Expect 头域。

--- 若代理知道下一站服务器版本是 HTTP/1.0 或更低，则它不能转发此请求，并且它必须以 417（期望失败）状态响应。

--- 代理应当维护一个缓存，以记录最近访问下一站点服务器的 HTTP 版本号。

--- 若接收到的请求来自于版本是 HTTP/1.0（或更低）的客户端，并且此请求不含值为“100-continue”的 Expect 请求头域，那么代理不能转发 100（继续）响应。 这一要求可覆盖 1xx 响应转发的一般规则（参见 10.1 节）。

8.2.4 服务器过早关闭连接时客户端的行为

如果 HTTP/1.1 客户端发送一条含有消息主体的请求消息，但不含值为“100-continue”的 Expect 请求头域，并且如果客户端没有直接与 HTTP/1.1 源服务器相连，并且客户端在接收到服务器的状态响应之前看到了连接的关闭，那么客户端应该重试此请求。在重试时，客户端可以利用下面的算法来获得可靠的响应。

1． 向服务器发起一新连接。

2． 发送请求头域。

3． 初始化变量 R，使 R 的值为通往服务器的往返时间的估计值（比如基于建立连接的时间），或在无法估计往返时间时设为一常数值 5 秒。

4． 计算 T=R*（2**N），N 为此前重试请求的次数。

5． 等待服务器出错响应，或是等待 T 秒（两者中时间较短的）。

6． 若没等到出错响应，T 秒后发送请求的消息主体。

7． 若客户端发现连接被提前关闭，转到第 1 步，直到请求被接受，接收到出错响应，或是用户因不耐烦而终止了重试过程。

在任意点上，客户端如果接收到服务器的出错响应，客户端

--- 不应再继续发送请求， 并且

--- 应该关闭连接如果客户端没有完成发送请求消息。

9 方法定义（Method Definitions）

HTTP/1.1 常用方法的定义如下。虽然方法可以被展开，但新加的方法不能被认为与扩展客户端和服务器共享同样的语义。

Host 请求头域（见 14.23 节）必须能在所有的 HTTP/1.1 请求里出现。

9.1 安全和等幂（Idempotent）方法

9.1.1 安全方法（Safe Methods）

实现者应当知道软件是代表用户在互联网上进行交互，并且应该小心地允许用户知道任何它们可能采取的动作(action)，这些动作可能给他们自己或他人带来无法预料的结果。

GET 和 HEAD 方法只是执行没有影响的动作那就是获取资源。这些方法应该被考虑是“安全”的。可以让用户代理调用的其它方法，如：POST，PUT，DELETE，这些方法是特殊的，用户代理应该知道这些方法可能会执行不安全的动作。

自然的，保证当服务器由于执行 GET 请求而不能产生副作用是不可能的；实际上，一些动态的资源会考虑这个特性。用户并没有请求这些副作用，因此这些副作用对用户应该是不受影响的。

9.1.2 等幂方法（Idempotent Mehtods）

方法可以有等幂的性质因为（除了出错或终止问题）N>0 个相同请求的副作用同单个请求的副作用的效果是一样（译注：等幂就是值不变性，相同的请求得到相同的响应结果，不会出现相同的请求出现不同的响应结果）。方法 GET，HEAD，PUT，DELETE 都有这种性质。同样，方法 OPTIONS 和 TRACE 不应该有副作用，因此具有内在的等幂性。然而，有可能有多个请求的请求序列是不等幂的，即使在那样的序列中所有方法都是等幂的。（如果整个序列整体的执行的结果总是相同的，并且此结果不会因为序列的整体，部分的再次执行而改变，那么此序列是等幂的。）例如，一个序列是非等幂的如果它的结果依赖于一个值，此值在以后相同的序列里会改变。

根据定义，一个序列如果没有副作用，那么此序列是等幂的（假设在资源集上没有并行的操作）。

9.2 OPTIONS（选项）

OPTIONS 方法表明请求想得到请求/响应链上关于此请求里的 URI（Request-URI）指定资源的通信选项信息。此方法允许客户端去判定请求资源的选项和/或需求，或者服务器的能力，而不需要利用一个资源动作（译注：使用 POST，PUT，DELETE 方法）或一个资源获取（译注：用 GET 方法）方法。

此方法的响应是不能缓存的.。

如果 OPTIONS 请求消息里包括一个实体主体（当请求消息里出现 Content-Length 或者 Transfer-Encoding 头域时），那么媒体类型必须通过 Content-Type 头域指明。虽然此规范没有定义如何使用此实体主体，将来的 HTTP 扩展可能会利用 OPTIONS 请求的消息主体去得到服务器得更多信息。一个服务器如果不支持 OPTION 请求的消息主体，它会遗弃此请求消息主体。

如果请求 URI 是一个星号（"*"）,，OPTIONS 请求将会应用于服务器的所有资源而不是特定资源。因为服务器的通信选项通常依赖于资源，所以”*”请求只能在“ping”或者“no-op”方法时

才有用；它干不了任何事情除了允许客户端测试服务器的能力。例如：它能被用来测试代理是否遵循 HTTP/1.1。

如果请求 URI 不是一个星号（"*"）,，OPTIONS 请求只能应用于请求 URI 指定资源的选项。

200 响应应该包含任何指明选项性质的头域，这些选项性质由服务器实现并且只适合那个请求的资源（例如，Allow 头域），但也可能包一些扩展的在此规范里没有定义的头域。如果有响应主体的话也应该包含一些通信选项的信息。这个响应主体的格式并没有在此规范里定义，但是可能会在以后的 HTTP 里定义。内容协商可能被用于选择合适的响应格式。如果没有响应主体包含，响应就应该包含一个值为“0”的 Content-Length 头域。

Max-Forwards 请求头域可能会被用于针对请求链中特定的代理。当代理接收到一个 OPTIONS 请求，且此请求的 URI 为 absoluteURI，并且此请求是可以被转发的，那么代理必须要检测 Max-Forwards 头域。如果 Max-Forwards 头域的值为“0”，那么此代理不能转发此消息；而是代理应该以它自己的通信选项响应。如果 Max-Forwards 头域是比 0 大的整数值，那么代理必须递减此值当它转发此请求时。如果没有 Max-Forwards 头域出现在请求里，那么代理转发此请求时不能包含 Max-Forwards 头域。

9.3 GET

GET 方法意思是获取被请求 URI（Request-URI）指定的信息（以实体的格式）。如果请求 URI 涉及到一个数据生成过程，那么这个过程生成的数据应该被作为实体在响应中返回而不是过程的源文本，除非源文本恰好是过程的输出。

如果请求消息包含 If-Modified-Since,，If-Unmodified-Since，If-Match，If-None-Match 或者 If-Range 头域，GET 的语义将变成“条件（conditionall） GET”。一个条件 GET 方法会请求满足条件头域的实体。条件 GET 方法的目的是为了减少不必要的网络使用，这通过允许利用缓存里仍然保鲜的实体而不用多次请求或传输客户端已经拥有的实体来实现的。.

如果请求方法包含一个 Range 头域，那么 GET 方法就变成“部分 Get”（partial GET）方法。一个部分 GET 会请求实体的一部分，这在 14.35 节里描述了。 部分 GET 方法的目的是为了减少不必要的网络使用，可以允许客户端从服务器获取实体的部分数据，而不需要获取客户端本地已经拥有的部分实体数据。

GET 请求的响应是可缓存的（cacheable）如果此响应满足第 13 节 HTTP 缓存的要求。

看 15.1.3 节关于 GET 请求用于表单时安全考虑。

9.4 HEAD

HEAD 方法和 GET 方法一致，除了服务器不能在响应里返回消息主体。HEAD 请求响应里 HTTP 头域里的元信息（译注：元信息就是头域信息）应该和 GET 请求响应里的元信息一致。此方法被用来获取请求实体的元信息而不需要传输实体主体（entity-body）。此方法经常被用来测试超文本链接的有效性，可访问性，和最近的改变。.

HEAD 请求的响应是可缓存的，因为响应里的信息可能被缓存用于更新以前那个资源对应缓存的实体.。如果出现一个新的域值指明缓存的实体和当前源服务器上的实体有所不同（可能因为 Content-Length，Content-MD5，ETag 或 Last-Modified 值的改变），那么缓存（cache）必须认为缓存项是过时的（stale）。

9.5 POST

POST 方法被用于请求源服务器接受请求中的实体作为请求资源的一个新的从属物。 POST 被设计涵盖下面的功能。

--已存在的资源的注释；

--发布消息给一个布告板，新闻组，邮件列表，或者相似的文章组。

--提供一个数据块，如提交一个表单给一个数据处理过程。

--通过追加操作来扩展数据库。

POST 方法的实际功能是由服务器决定的，并且经常依赖于请求 URI（Request-URI）。POST 提交的实体是请求 URI 的从属物，就好像一个文件从属于一个目录，一篇新闻文章从属于一个新闻组，或者一条记录从属于一个数据库。

POST 方法执行的动作可能不会对请求 URI 所指的资源起作用。在这种情况下，200（成功）或者 204（没有内容）将是适合的响应状态，这依赖于响应是否包含一个描述结果的实体。

如果资源被源服务器创建，响应应该是 201（Created）并且包含一个实体，此实体描述了请求的状态。并且引用了这个新资源和一个 Location 头域（见 14.30 节）。

POST 方法的响应是不可缓存的。除非响应里有合适的 Cache-Control 或者 Expires 头域。然而，303（见其他）响应能被用户代理利用去获得可缓存的响应。

POST 请求必须遵循 8.2 节里指明的消息传送的要求。

参见 15.1.3 节关于安全性的考虑.

9.6 PUT

PUT 方法请求服务器去把请求里的实体存储在请求 URI（Request-URI）标识下。如果请求 URI（Request-URI）指定的的资源已经在源服务器上存在，那么此请求里的实体应该被当作是源服务器关于此 URI 所指定资源实体的最新修改版本。如果请求 URI（Request-URI）指定的资源不存在，并且此 URI 被用户代理定义为一个新资源，那么源服务器就应该根据请求里的实体创建一个此 URI 所标识下的资源。如果一个新的资源被创建了，源服务器必须能向用户代理（user agent） 发送 201（已创建）响应。如果已存在的资源被改变了，那么源服务器应该发送 200（Ok）或者 204（无内容）响应。如果资源不能根据请求 URI 创建或者改变，一个合适的错误响应应该给出以反应问题的性质。实体的接收者不能忽略任何它不理解和不能实现的 Content-*（如：Content-Range）头域，并且必须返回 501（没有被实现）响应。

如果请求穿过一个缓存（cache），并且此请求 URI（Request-URI）指示了一个或多个当前缓存的实体，那么这些实体应该被看作是旧的。PUT 方法的响应是不可缓存的。

POST 方法和 PUT 方法请求最根本的区别是请求 URI（Request-URI）的含义不同。POST 请求里的 URI 指示一个能处理请求实体的资源（译注：此资源可能是一段程序，如 jsp 里的 servlet） 。此资源可能是一个数据接收过程，一个网关（gateway，译注：网关和代理的区别是：网关可以进行协议转换，而代理不能，只是起代理的作用，比如缓存服务器其实就是一个代理），或者一个单独接收注释的实体。对比而言，PUT 方法请求里的 URI 标识请求里封装的

实体一一用户代理知道 URI 意指什么，并且服务器不能把此请求应用于其它资源（resource）。如果服务器期望请求被应用于一个不同的 URI，那么它必须发送 301（永久移动）响应；用户代理可以自己决定是否重定向请求。

一个单独的资源可能会被许多不同的 URI 指定。如：一篇文章可能会有一个 URI 指定当前版本，而这个 URI 区别于这篇文章其它特殊版本的 URI。这种情况下，对一个通用 URI 的 PUT 请求可能会导致其资源的其它 URI 请求被源服务器重定义。

HTTP/1.1 没有定义 PUT 方法对源服务器的状态影响。

PUT 请求必须遵循 8.2 节中的消息传送的要求。

除非特别指出，PUT 方法请求里的实体头域应该被用于资源的创建或修改。

9.7 DELETE（删除）

DELETE 方法请求源服务器删除请求 URI 指定的资源。此方法可能会在源服务器上被人为的干涉（或通过其他方法）。客户端不能保证此操作能被执行，即使源服务器返回成功状态码。然而，服务器不应该指明成功除非它打算删除资源或把此资源移到一个不可访问的位置。

如果响应里包含描述成功的实体，响应应该是 200（OK）；如果 DELETE 动作还没有执行，应该以 202（已接受）响应；如果 DELETE 请求方法已经执行但响应不包含实体，那么应该以 204（无内容）响应。

如果请求穿过缓存，并且请求 URI（Request-URI）指定了一个或多个缓存当前实体，那么这些缓存项应该被认为是旧的。DELETE 方法的响应是不能被缓存的。

9.8 TRACE

TRACE方法被用于激发一个远程的，应用层的请求消息回路（译注：TRACE方法让客户端测试到服务器的网络通路，回路的意思如发送一个请返回一个响应，这就是一个请求响应回路，）。最后的接收者也许是源服务器，也许是接收到包含Max-Forwards头域值为0请求的代理或网关。TRACE请求不能包含一个实体。

TRACE方法允许客户端去了解数据被请求链的另一端接收的情况，并且利用那些数据信息去测试或诊断。Via头域值（见14.45）有特殊的用途，因为它可以作为请求链的跟踪信息。利用Max-Forwards头域允许客户端限制请求链的长度，这是非常有用的，因为可以利用此去测试代理链在无限循环里转发消息。

如果请求是有效的，响应应该在实体主体里包含整个请求消息，并且响应应该包含一个Content-Type头域值为”message/http”的头域。此方法的响应不能被缓存。

9.9 CONNECT（连接）

HTTP1.1 协议规范保留了 CONNECT 方法，此方法是为了能用于能动态切换到隧道的代理（(如 SSL tunneling [44]).）。

10.状态码定义

每一个状态码在下面定义，包括此状态码依赖于方法的描述和响应里需要的任何元信息的描述。

10.1 通知的 1xx

这类状态代码指明了一个临时性的响应，包含一个 Status-Line 和可选的头域，并且被一个空行结束（译注：空行就是 CRLF）。这类状态码响应没有必须的头域。因为 HTTP/1.0 没有定义任何 1xx 状态码，所以服务器不能发送一个 1xx 响应给一个 HTTP/1.1 客户端，除了实验性的目的。

客户端必须能在一个常规响应之前接受一个或多个 1xx 状态，即使客户端不期望 100（继续）状态响应。不被客户端期望的 1xx 状态响应可能会被用户代理忽略。

代理必须能转发 1xx 响应，除非代理和它的客户端的连接关闭了，或者除非代理自己响应请求并产生 1xx 响应。（例如：如果代理添加了“Expect:100-continue”头域当转发请求时，那么它不必转发相应的 100（继续）状态响应。）

10.1.1 100 继续 （Continue）

100 状态响应告诉客户端应该继续请求。100 响应是个中间响应，它被用于通知客户端请求的初始部分已经被接收了并且此请求还没有被服务器丢弃。客户端应该继续发送请求的剩余部分，或者，如果此请求已经完成了客户端会忽略此 100 响应。服务器在接收请求后必须发送一个终结响应。见 8.2.3 节关于此状态码的讨论和使用。

10.1.2 101 切换协议 （Switching Protocols）

服务器理解和愿意遵循客户端这样的请求，此请求通过 Upgrade 消息头域（见 14.42 节）指明在连接上应用层协议的改变。 服务器将会切换到响应里 Upgrade 头域里指明的协议，它会以一个空行结束此 101 响应。

只有协议切换时能受益协议才应该切换。例如，当传输资源时，切换到一个新的 HTTP 版本比旧的版本要好，或者切换到一个实时的，同步的协议会带来好处时，这时我们都应该考虑切换。

10.2 成功 2xx

这类状态码指明客户端的请球已经被服务器成功的接收，理解，并且接受了。

10.2.1 200 OK

此状态码指明客户端请求已经成功了。响应返回的信息依赖于请求里的方法，例如：

GET 请求资源的相应的实体已经包含在响应里并返回给客户端。

HEAD 相应于请求资源实体的实体头域已经被包含在无消息主体的响应里。

POST 响应里已经包含一个实体，此实体描述或者包含此 POST 动作执行的结果

TRACE　响应里包含一个实体，此实体包含终端对服务器接收的请求消息。

10.2.2 201 已创建（Created）

请求已经被服务器满足了并且已经产生了一个新的资源。新创建的资源的 URI 在响应的实体里返回，但是此资源最精确的 URI 是在 Location 头域里给出的。响应应该含有一实体，此实体包含此资源的特性和位置，用户或用户代理能从这些特性和位置里选择最合适的。实体格式被 Content-Type 头域里媒体类型指定。源服务器必须能在返回 201 状态码之前建立资源。如果动作（译注：这里指能创建资源的方法，如 POST 方法）不能被立即执行，那么服务器应该以 202（接受）响应代替。

一个 201 响应可以包含一个 ETag 响应头域，此头域的值指明了当前请求变量（译注：变量的含义见第 1.3 节“变量”的解释）也即刚刚创建的资源的实体标签（entity tag）值，见 14.19 节。

10.2.3 202 接受（Accepted）

请求已经被接受去处理，但是还没有处理完成。请求可能会或者不会处理完成，因为存在当处理的过程中拒绝处理的情况。

202 响应是有意非担保性的。它是为了允许服务器可以为其它处理（如：每天执行一次的批处理）接收请求而不需要用户代理在处理没有完成之前长期连接到服务器。响应里的实体应该包含请求当前状态的声明并且应该包含一个状态监视指针或一些用户期望何时请求被满足的评估值。

10.2.4 203 非权威信息（Non-Authoritative information）

此状态码响应指明响应里实体头域元信息不能从源服务器获而是从本地的或第三方响应副本里收集的。这些元信息可能是源服务器版本的子集或超集。如，包含一个存在本地的资源注释信息就可以产生一个源服务器能理解的元信息的超集。利用此响应状态码不是必须但是比 200（Ok）响应却更加合适。

10.2.5 204 无内容 （No Content）

服务器已经满足了请求但并没有返回一个实体而是返回更新的元信息。此响应可能包含新的或更新的元信息以实体头域的形式，这些元信息应该相关于请求变量。

利用此 204 响应，客户端如果是一个用户代理，它就可以不用改变引起请求发送的文档视图（译注：如一篇 html 文档在浏览器里呈现的样子）。204 状态响应主要的目的是允许输入，而不必引起用户代理当前文档视图的改变，尽管一些新的或更新了的元信息可能会应用于用户代理视图里的当前文档。

204 响应不能包含一个消息主体，并且在头域后包含一个空行结束。

10.2.6 205 重置内容（Reset Content）

205 状态响应是服务器告诉用户代理应该重置引起请求被发送的文档视图。此响应主要的目的是清空文档视图表单里的输入框以便用户能输入其它信息。此响应不能包含一个实体。

10.2.7 206 部分内容（Partial Content）

服务器已经完成了客户端对资源的部分 GET 请求。请求必须包含一个 Range 头域（14.35 节）用来指出想要的范围，并且也有可能包含一个 If-Range 头域（见 14.27 节）来使请求成为一个条件请求。

206 状态的响应必须包含以下的头域：

- 或者含有一个 Content-Range 头域，此头域指明了响应里的范围；或者含有一个值为“multipart/byteranges”的 Content-Type 头域并且每部分包含 Content-Range 头域。如果一个 Content-Length 头域出现在响应里，它的值必须是实际传输的消息主体的字节数。

- Date 头域

- ETag 和/或 Content-Location 头域，如果这些头域假设在相同请求的 200 响应里也会出现的话。

- Expire，Cache-Control，和/或者 Vary 头域，如果这些头域的域值与以前同一变量响应中的不一样。

如果 206 响应是使用了强缓存验证（见 13.3.3）的 If-Range 请求的结果，那么此响应不应该包含其他的实体头域。如果响应是使用了弱缓存验证的 If-Range 请求的结果，那么响应必须不能包含其他的实体头域；这能防止缓存里缓存的实体主体与更新头域之间的不一致性。另外，响应必须包含假设在相同请求的 200 响应里的所有实体头域。

缓存不能把 206 响应和以前的缓存内容相合并如果 ETag 或 Last-Modified 头域并不能精确匹配，见 13.5.4。

一个不能支持 Range 和 Content-Range 头域的缓存不能缓存 206（部分的）响应。

10.3 重新定向 3xx.

这类状态码指明用户代理需要更进一步的动作去完成请求。进一步的动作可能被用户代理自动执行而不需要用户的交互，并且进一步动作请求的方法必须为 GET 或 HEAD。一个客户端应该发现无限的重定向循环，因为此循环能产生网络拥挤。

注意：以前此规范版本建议一个最多能有五个重定向。内容开发者应该知道客户端可能存在这个限制。

10.3.1 300 多个选择.（Multiple Choices）

请求资源对应于众多表现形式中的一个，每个表现形式都有一个特定的位置(location)，并且代理驱动协商（agent-driven negotiation）信息（见 13 章）被提供以便用户（或用户代理）能选择一个更适的表现形式并重定向它的请求到那个表现形式的位置。

除非是 HEAD 请求，否则 300 状态响应应该包含一个实体，此实体包含一个资源特性和位置列表，从这个列表里用户或用户代理能选择最合适的资源的表现形式。实体格式被 Content-Type 头域里的媒体类型指定。用户代理选择最合适的表现形式的行为可能会被自动执行，这依赖于实体格式和自己的能力。然而，此规范并没有定义自动执行行为的标准。

如果服务器能确定更好的表现形式，它应该为此表现形式在 Location 头域里包含一个特定的

URI 来指明此表现形式的位置；用户代理可能会利用此 Location 头域自动重定向。300 状态响应是可缓存的除非被特别指明。

10.3.2 301 永久移动 （Moved Permanently）

请求资源被赋于一个新的永久的 URI，并且任何将来对此资源的引用都会利用此 301 状态响应返回的 URI。具有链接编辑能力的客户端应该能自动把请求 URI 的引用转到到服务器返回的新的引用下。此响应是能缓存的除非另外声明。

新的永久 URI 应该在响应中被 Location 头域给定。除非请求方法是 HEAD，否则此响应应该包含一个超文本提示和一个指向新 URI 的超文本链接。

如果客户端接收了一个来自非 GET 或 HEAD 请求方法的 301 响应，那么用户代理不能自动重定向请求除非它能被用户确认，因为这可能会改变请求提交的条件。

注意：当客户端在接收了 301 状态码响应后，会重定向 POST 请求，一些已经存在的 HTTP/1.0 用户代理会错误的把此请求变成一个 GET 请求。

10.3.3 302 发现(Found)

请求的资源暂时地存放在一个不同的 URI 下。因为重定向的地址可能有时会被改变，客户端应该继续为将来的请求利用请求 URI(Request-URI)。302 响应是只有在 Cache-Control 或 Expires 头域指明的情况下才能被缓存。

临时的 URI 应该在 Location 头域里指定。除非请求方法是 HEAD，否则此响应应该包含一个超文本提示和一个指向新 URI 的超文本链接。

如果客户端接收了一个来自非 GET 或 HEAD 请求方法的 302 响应，那么用户代理不能自动重定向请求除非它能被用户确认，因为这可能会改变请求提交的条件。

注意：RFC1945 和 RFC2068 指定客户端不能在重定向请求的时候改变请求方法。然而，大多数用户代理实现会把 302 响应看成是 303 响应，从而根据 Location 头域值的 URI 执行 GET 请求，不管原始的请求方法是什么。303 和 307 状态响应的目的是为使服务器明白客户端期望哪种类型的重定向。

10.3.4 303 见其他(See Other)

请求的响应被放在一个不同的 URI 下，并且应该用 GET 方法获得那个资源。此方法的存在主要是让 POST 调用脚本的输出能使用户代理重定向到一个选择的资源。新的 URI 并不是原始请求资源的代替引用。303 响应不能被缓存，但是再次重定向请求的响应应该被缓存。

不同的 URI 应该在 Location 头域里指定。除非请求方法是 HEAD，除非请求方法是 HEAD，否则此响应应该包含一个超文本提示和一个指向新 URI 的超文本链接。

注意：许多 HTTP/1.1 以前版本的用户代理不能理解 303 状态响应。当这些客户端比较关注于互操作性的时候，302 状态码应该被代替利用，因为大多用户代理对 302 响应的理解就是 303 响应。

10.3.5 304 没有改变（Not Modified）

如果客户端已经执行了条件 GET 请求，并且访问服务器的资源是允许的，但是服务器上的文档并没有被改变，那么服务器应该以此状态码响应。304 响应不能包含一个消息主体(message-body)，并且在头域后面总是以一个空行结束。

此响应必须包含下面的头域：

- Date，除非 14.18.1 指明的那些规则下 Date 是可以遗漏的。如果时钟不准确的源服务器遵循这些规则，并且代理和客户端在接收了一个没有 Date 头域的响应后加上了自己的 Date（这在 RFC 2086 里声明了，见 14.19 节），缓存将会正确操作。

- ETag 和/或 Content-Location 头域，如果这些头域应在相同请求的 200 响应里出现的话。

- Expire，Cache-Control，和/或者 Vary 头域，如果这些头域值与以前同一变量响应中的不一致。

如果条件 GET 请求使用强缓存验证（见 13.3.3 节）时，那么响应不应包含其它实体头域。当条件 GET 使用弱缓存验证时，那么响应必须不能包含其它实体头域；这能防止缓存的实体主体与更新的头域之间的不一致性。

如果一个 304 响应指示一个没有被缓存的实体，那么此缓存必须不用理会此响应，并且以无条件请求重试请求。

如果缓存利用一个接收到的 304 响应去更新一个缓存项，那么缓存必须用此响应响应里任何最新的域值更新缓存项。

10.3.6 305 使用代理 （Use Proxy）

请求资源必须能通过代理访问，代理的地址在响应的 Location 头域里指定。Location 头域指定了代理的 URI。接收者被期望通过代理重试此请求，305 响应必须被源服务器产生。

注意：RFC 2068 并没有说明 305 响应必须重定向一个单独请求并且只能被源服务器产生。不注意这些限制会有重要的安全后果。

10.3.7 306 没有使用的（unused）

306 状态码被用于此规范以前的版本，是不再使用的意思，并且此状态码被保留。

10.3.8 307 临时重发（Temporary Redirect）

请求的资源临时存在于一个不同的 URI 下。由于重新向可能有时会改变，所以客户端应该继续利用此请求 URI（Request-URI）为将来的请求。307 响应只有被 Cache-Control 或 Expire 头域指明时才能被缓存。

临时 URI 应该在响应的 Location 头域里给定。否则此响应应该包含一个超文本提示和一个指向新 URI 的超文本链接，因为许多 HTTP/1.1 以前的用户代理不能理解 307 状态响应。因此，此提示应该包含用户在新的 URI 上重试原始请求的必需信息。

如果 307 状态响应.对应的请求的方法不是 GET 或 HEAD，那么用户代理不能自动重定向此请求除非它能被用户确认，因为因为这可能会改变请求提交的条件。

10.4 客户端错误 4xx

状态码 4xx 类的目的是为了指明客户端出现错误的情况。除了当响应一个 HEAD 请求，服务器应该包含一个实体，此实体包含一个此错误请求的解释。此状态码对所有请求方法都是适合的。用户代理应该展示任何响应里包含的实体给用户。

如果客户端发送数据，利用 TCP 的服务器实现应该小心地确保客户端确认包含了响应的包（packets）的接收，在服务器关闭此输入连接前。如果在关闭连接后，客户端继续发送数据给服务器，那么服务器的 TCP 栈将发送一个重置包给客户端，这能擦除客户端非确认的输入缓冲（input buffers）在这些缓冲被 HTTP 应用程序读和解析之前。

10.4.1 400 坏请求（Bad Request）

请求不能被服务器理解，由于错误的语法。客户端不应该在没有改变请求的情况下重试请求。

10.4.2 401 未授权的 （Unauthorized）

服务器需要对请求进行用户认证。响应必须包含一个 WWW-Authenticate 头域（见 14.47），此头域包含一个适用于请求资源的授权的激励（challenge）。客户端会以一个 Authorization 头域重试请求。如果请求包含了授权证书，那么 401 响应指明对这些证书的授权失败。如果 401 响应包含一个和以前响应的同样激励，并且用户代理已经尝试至少一次的授权，那么用户应该被呈现包含在响应里的实体，因为这些实体可能包含相关的诊断信息。HTTP 授权访问在“HTTP Authentication：Basic and Digest Access Authentication”[43]里解释。

10.4.3 402 必需的支付 （Payment Required）

此状态码为将来的应用保留。

10.4.4 403 禁用 （Forbidden）

服务器理解此请求，但拒绝满足此请求。认证是没有作用的，并且请求不应该被重试。如果请求方法是 HEAD 并且服务器想让客户端知道请求为什么不能被满足，那么服务器起应该在响应实体里描述此拒绝的原因。如果服务器不希望告诉客户端拒绝的原因，那么 404 状态码(Not Found)响应将被使用。

10.4.5 404 没有找到（Not Found）

服务器并没有找到任何可以匹配请求 URI 的资源。没有迹象表明条件是暂时或永久的 。410（Gone）状态响应应该被使用，如果服务器通过内部配置机制知道一个旧资源永远不能获得并且也没有转发地址。此状态码通常被使用，当服务器不希望精确指出请求为何被拒绝，或者当没有任何其它响应可用时。

10.4.6 405 方法不被允许（Method Not Allowed）

此状态码表示请求行（Request-Line）里的方法对此资源来说不被允许。响应必须包含一个 Allow 头域，此头域包含以一系列对此请求资源有效的方法。

10.4.7 406 不可接受的 （Not Acceptable）

根据客户端请求的接受头域（译注：如：Accept, Accept-Charset, Accept-Encoding, 或者 Accept-Language），服务器不能产生让客户端可以接受的响应。

除非是 HEAD 请求，否则响应应该包含一个实体，此实体应该包含一个可得的实体特性和位置列表，通过它用户或用户代理能选择最合适自己的。实体格式被媒体类型指定。依赖于此格式和用户代理的本身能力，选择最合适的可能会被自动执行。然而，此规范并没有定义自动执行选择的标准。

注意：HTTP/1.1服务器被准许根据请求里的接受头域会返回不可接受的响应。在一些情况下，这可能更倾向于发送一个406响应。用户代理被鼓励观察到来的响应的头域来确定此响应是否是可接受的。

如果响应是不可接受的，用户代理应该暂时停止剩余数据的接收并且询问用户然后去决定进一步的动作。

10.4.8 407 需要代理验证（Proxy Authentication Required）

此状态码和 401（Unauthorized）相似，但是指示客户端首先必须利用代理对自己验证。代理必须返回一个 Proxy-Authenticate 头域（见 14.33 节），此头域包含一个适用于代理的授权激励。客户端可能利用一个合适的 Proxy-Autorization 头域去重试此请求（见 14.34 节）。HTTP 访问授权在“HTTP Authentication；Basic and Digest Access Authentication”[43]。

10.4.9 408 请求超时（Request Timeout）

客户端在服务器等待的时间里不能产生请求。客户端可能在以后会重试此请求。

10.4.10 409 冲突 （Confilict）

请求不能完成由于和当前资源的状态冲突。此状态码只被允许出现在期望用户也许能解决此冲并且能重新提交此请求的情况下。响应主体应该包含足够的为用户认识此资源冲突的信息。理想的情况下，响应实体应该包含足够为用户或用户代理解决此问题的信息；然而，这是也许没有可能并且也没有必要。

冲突最可能发生在响应 PUT 请求的时候。例如，如果版本被使用并且被 PUT 的实体包含资源的改变，而这些改变会和以前的（第三方的）请求的相冲突，那么服务器应该使用 409 响应去指明它不能完成此请求。在这种情况下，此响应的实体可能包含这两个版本的差异点，响应的实体格式以 Content-Type 头域指定。

10.4.11 410 不存在（gone）

请求资源在源服务器上不再可得并且也没有转发地址可用。此条件被认为是永久的。具有链接编辑能力的客户端应该在用户确认后删除请求 URI 的引用。如果服务器不知道或不容易去确定条件是否是永久的，那么此 404（没有发现）状态响应将被代替利用。响应是可缓存的，除非另外申明。

410 响应主要的目的是为了 web 维护任务，这通过告诉接收者资源已经不可得了并且告诉接收者服务器拥有者已经把那个资源的远程连接给移除了。对有时间限制的，推销性的服务，和对不再继续工作在服务器站点人员的资源，这个事件（410 响应）是非常普遍的。它不需要把所有

长久不可得的资源标记为“gone”或者保持任意长时间—这需要服务器拥有者自己的判断

10.4.12 411 长度必需 （Length Required）

服务器拒绝接受请求里没有包含 Content-Length 头域的请求。客户端可以重试此请求如果它添加了一个有效的 Content-Length 头域，此头域值指定了请求消息里消息主体的长度。

10.4.13 412 先决条件失败 （Precondition Failed）

在一个或多个请求头域里指定的先决条件当在服务器上测试为 false 时返回的响应。此响应允许客户端把先决条件放放到当前资源的元信息（头域数据）之上，这样能防止请求方法被应用于一个非目的性的资源。

10.4.14 413 请求实体太大

服务器拒绝处理请求因为请求实体太大以致达到服务器不愿意去处理。服务器可能关闭此连接去防止客户端继续请求。

如果条件是暂时的，服务器应该包含一个 Retry-After 头域用来指明此条件是暂时的并且指明客户端应该什么时候重试。

10.4.15 414 请求 URI 太长（Request-URI Too Long）

服务器拒绝为请求服务因为此请求 URI 太长了以至于服务器不能解析。这种情况是很少的，只发生在当客户端把 POST 请求不合适地转换为带有大量查询信息的 GET 请求时。

10.4.16 415 不被支持的媒体类型（Unsupported Media Type）

服务器拒绝为请求服务，因为请求的实体的格式不能被此方法的请求资源所支持。

10.4.17 416 请求范围不满足 （Requested Range Not Satisfiable）

服务器返回一个此状态码的响应，如果请求包含一个 Range 请求头域（见 14.35 节），并且此头域里 range-specifier 值没有和已选资源的当前 extent 值重叠，并且请求没有包含一个 If-Range 请求头域。（对 byte-ranges 来说，这意味着 byte-range-spec 的所有 first-byte-pos 值大于选择的资源的当前长度）。

当此状态码响应是在 byte-range 请求返回时，响应应该包含一个 Content-Range 实体头域用来指定已选资源的当前长度（见 14.16 节）。响应不能使用 multipart/byteranges 媒体类型。

10.4.18 417 期望失败（Expectation Failed）

Expect 请求头域里指定的希望不能被服务器满足，或者，如果服务器是代理，那么能确定请求不能被下一站（next-hop）服务器满足。

10.5 服务器错误 5xx （Server Error）

这类状态码指明服务器处理请求时产生错误或不能处理请求。除了 HEAD 请求，服务器应该包含一个实体，此实体用来解释错误，和是否是暂时或长期条件。用户代理应该展示实体给用户。此响应状态码能应用于任何请求方法。

10.5.1 500 服务器内部错误 （Internal Server Error）

服务器遇到了一个意外条件，此条件防止服务器满足此请求。

10.5.2 501 不能实现 （Not Implemented）

服务器没有能力去满足请求。当服务器不能识别请求方法并且不支持它请求资源的时候，这个响应是很合适的。

10.5.3 502 坏网关 （Bad Gateway）

此响应说明：作为网关或代理的服务器从上游（upstream）服务器接收了一个无效的响应。

10.5.4 503 服务不能获得（Service Unavailable）

由于服务器暂时地过载或维护，服务器不能处理请求。这就是说这是暂时条件，此条件将会在一些延时后被减轻。延迟的长度可以在 Retry-After 头域里指定。如果没有 Retry-After 被给，那么客户端应该处理此响应就像它处理 500 响应一样。

注意：503 状态码的存在并不是意指服务器当产生过载时必须利用它。一些服务器可能希望拒绝此连接。

10.5.5 504 网关超时（Gateway Timeout）

作为网关或代理的服务器在不能及时地接收一个从 URI 指定的上游（upstream）服务器（例如：HTTP，FTP，LDAP 服务器）或者其他的辅助性服务器（例如：DNS 服务器）的响应。

注意：当 DNS 查找超时时，一些部署的代理将会返回 400 或 500 响应。

10.5.6 505 HTTP 版本不支持 （HTTP version Not Supported）

服务器不能支持，或拒绝支持此 HTTP 协议版本消息。505 响应指明服务器不能或不愿意完成这样的请求，这在 3.1 中描述了。此响应应该包含一个实体，此实体描述了为什么此协议版本不被支持和其他能被服务器支持的协议版本。

11.访问认证（Access Authentication）

HTTP 提供一些可选的激励响应（challenge-response）认证（authentication）机制，这些机制能被用于服务器去激励客户端请求，使客户端提供认证信息。常用访问认证框架，还有“basic”和“digest”认证规范，都在“HTTP Authentication： basic and Digest Access Authentication”[43]规范里指定。HTTP/1.1 规范采用了“激励（chanllenge）”和“证书（credentials）”的定义。

12.内容协商 （Content Negotiation）

大多数响应包含一个实体，此实体包含人类用户能理解的信息。通常，希望响应能提供给用户相应请求的“最可得”（best available）的实体。对服务器和缓存来说，不幸的是，并不是所有的用户都对这个“最可得”的实体有相同的喜好，并且并不是所有的用户代理（如 web 浏览器）都能一致的呈现这些实体。所以，HTTP 提供了几个“内容协商”的机制 — 当有多个可得的表现形式的时候，对给定响应去选择最好的表现形式的过程。

注意：没有称做“格式协商”（译注：“格式”指的是“媒体类型”）的，这是因为选择的表现形式可能会有相同的媒体类型，但却根据其它能力（capabilities），例如一种不同的语言。

任何包含一个实体主体的响应应该由受协商来决定，包括错误响应。

在 HTTP 中，有两种类型的内容协商：服务器驱动协商和代理驱动协商。这两种类型的协商具有正交性，因此能被单独使用或联合使用。一个联合使用的协商会被叫做透明协商，这发生在当缓存利用源服务器提供的代理驱动协商信息为后续请求提供服务器驱动协商的时候。

12.1 服务器驱动协商（Server-driven Negotiation）

如果响应的最好的表现形式的选择是通过服务器上的算法来实现，那么这种方式的协商称做服务器驱动协商。选择是基于响应可得的表现形式（根据不同的维度，响应会不同；例如，语言，内容编码，等等）和请求消息里特定的头域或附属于请求的其他信息（如：网络客户端的地址）。

服务器驱动协商是有优点的，当从可行的表现形式里进行选择的算法对用户代理来说描述是比较难的（译注：代理驱动协商），或者当服务器期望通过第一个响应发送“最好的猜测”（“best guess”）给客户端时（如果“最好的猜测”对用户是合适的，能避免后续请求的回路延迟）的时候。为了改善服务器的猜测，用户代理应该包含请求头域（Accept，Accept-Language，Accept-Encoding，等等），这些头域能描述它对响应的喜好。

服务驱动协商有如下缺点：

1. 对服务器来说不可能确切地决定什么对用户来说是最好的，因为那需要对用户代理和响应目的的全面理解（如：用户到底想把响应展示到屏幕还是打印到纸上？）。

2. 使用户代理在而每一个请求里描述它的能力即不高效（假定只有少量的响应却拥有多个表现形式）也潜在地侵犯用户的隐私。

3. 使源服务器的实现变得复杂，并且使为请求产生响应的算法实现变得复杂。

4. 可能会限制公有缓存（public cache）为多个客户请求利用相同响应的能力

HTTP/1.1 包含下面的请求头域来使服务器驱动协商启动，这些请求头域描述了用户代理的能力和用户喜好：Accept（14.1 节），Accept-Charset（14.2 节），Accept-Encoding（14.3 节），Accept-Language（14.4 节），和 User-Agent（14.43 节）。然而，一些源服务器并不只局限于这些维度，这些服务器能基于请求的任何方面来让响应不同，这些方面包括请求头域之外的信息或在此规范里没有定义的扩展头域信息。

Vary 头域能被用来表达服务器选择表现形式（representation）利用的参数，表现形式受服务器驱动协商决定的。见 14.44 节描述了 Vary 头域如何被服务器的使用，13.6 节描述了 Vary 头域如何被缓存的使用。

12.2 代理驱动协商 （Agent-driven Negotiation）

对代理驱动协商来说，在源服务器一个初始响应后，响应的最好表现形式的选择是被用户代理执行的。选择是基于响应的一系列可得的表现形式，这些表现形式被包含在初始响应的头域或初始响应的实体主体（entity-body）里，每个表现形式被一个属于自己的 URI 指定。从这些表现形式中选择可能被自动执行（如果用户代理有能力这样做）或者被用户从超文本连接菜单中手工选择。

代理驱动协商是有优点的，当响应可能只能根据一般用途的维度（如：类型，语义，编码）而不同的时候，当源服务器不能通过查看请求而判定用户代理能力的时候，当共有缓存（public cache）被用来分派服务器的承载和减少网络使用的时候。

代理驱动协商也需要第二次请求去获得最好表现形式的缺点。第二次请求只有当缓存被使用时才是有效率的。另外，此规范没有定义用户代理自动选择表现形式的机制，虽然它也没有避免这样的机制被作为一个扩展使用于 HTTP/1.1。

HTTP/1.1 定义了 300（多个选择）和 406（不接受的）状态响应来进行代理驱动协商，当服务器不能或不愿意利用服务器驱动协商来提供一个不同响应（a varying response）的是时候。

12.3 透明协商（Transparent Negotiation）

透明协商是服务器驱动协商和代理驱动协商的结合体。当一个缓存接收到服务器的响应，并且响应里提供了一系列可得的表现形式（就像在代理驱动协商里的响应一样）并且缓存能理解维度的差异，那么此缓存变得有能力代表源服务器为那个资源的后续请求去执行服务器驱动协商。

透明协商的优点在于它能分发源服务器的协商工作并且能移除代理驱动协商的第二次请求的延迟，因为缓存能正确的猜测到合适的响应并返回给请求端。

此规范没有定义透明协商的机制，虽然它也没有避免任何这样的机制作为扩展被用于 HTTP/1.1。

13 HTTP 中的缓存

HTTP 通常应用于能通过采用缓存技术提高性能的分布式信息系统。HTTP/1.1 协议包括的许多使缓存尽可能的工作的元素。因为这些元素与协议的其他方面有着千丝万缕的联系，而且他们相互作用、影响，因此有必要分别从方法（methods），头（headers），响应状态码来介绍缓存的基本设计。

如果缓存不能改善性能，他将一无用处。HTTP/1.1 中缓存可以在许多情况下排除发送请求和发送完整响应。前者减少了网络回路的数量；我们利用一个“过期（expiration）”机制来为此目的（见 13.2 节）。后者减少了网络应用的带宽；我们用“验证（validation）”机制来为此目的（见 13.3）。

对行为，可行性，和关闭的操作的要求需要我们能放松了语义透明性。HTTP/1.1 协议允许服务器，缓存，和客户端在必要的时候能显式地降低透明性。然而，因为不透明的操作能混淆非专业的用户，同时可能和某个服务器应用程序不兼容（例如订购商品），因此协议透明性在下面情况下才能被放松：

-- 当被客户端或源服务器放松时，只能被一个显式的协议层（protocol-level）请求放松
-- 当被缓存或被客户端放松时，只能在出现一个对终端用户的显式的警告时才能被放松

因此，HTTP/1.1 协议提供这些重要的元素：

1．提供了完全语义透明性（full semantic transparency）的特性当被通信所有方需要时

2. 允许源服务器或用户代理显式地请求和控制非透明性操作的协议特性

3. 允许缓存给响应绑定警告信息（通过 Warning 头域来实现）的协议特性，当这些响应没有保持语义透明的请求近似性（requested approximation）时

一个基本的原则是客户端必须能够发现任何语义透明性的潜在放松规则。

注意：服务器，缓存，或者客户端的实现者可能会面对设计上的决策，它并没有明确地在此规范里讨论。如果一个决策影响了语义透明性，那么实现者将会在维持语义透明性上犯错，除非一个仔细且完整的分析能向我们展示打破透明性带来的好处。

13.1.1 缓存正确性（Cache Correctness）

一个正确的缓存必须能用缓存里最新的（up-to-date）响应来响应请求（见 13.2.5，13.2.6，和 13.12），已缓存的响应必须满足下面的条件：

1．已缓存的响应已被检测应与通过源服务器重验证后返回的响应等价。

2．已缓存的响应是足够保鲜(fresh)的（见 13.2 节）。缺省的情况下，这意味着此响应必须满足客户端，源服务器，和缓存（见 14.9 节）的最少严格（least restrictive freshness）保鲜要求；如果源服务器指定了保鲜要求，那么它是源服务器自己的保鲜要求。

   如果一个已缓存的响应，由于客户端和源服务器的最多严格（most restrictive freshness）保鲜要求，而不是足够保鲜的，那么在仔细考虑的情况下，缓存也许仍然会根据一个合适的警告头域（见 13.15 和 14.46）返回此已缓存的响应，除非这个响应不被允许（例如：通过”no-store” cache-directive，或者通过一个”no-cache” cache-request-directive；见 14.9 节）。

3．已缓存的响应是一个合适的 304（没有被改变），305（代理重定向），或 错误（4xx 或者 5xx）响应消息。

如果缓存不能同源服务器通信，如果已缓存的响应能正确的服务于请求，那么一个正确的缓存应该用它缓存的响应去响应请求；如果不能服务器于此请求，那么缓存必须能返回一个错误或一个警告指示通信失败。

如果缓存从服务器端接收到一个响应（或者是一个完整的响应，或者一个 304（没有改变）的响应），此响应应该是服务器正常情况下发送到请求的客户端的响应，并且此响应并不再新鲜，那么此缓存应该把此响应转发给请求客户端而不需要添加一个新的 Warning 头域（而不需要移去已经存在的 Warning 头域）。缓存并不是简单的因为传输中响应变得陈旧而去尝试重验证响应；这可能会导致一个无限的循环。用户代理接收一个陈旧的但没有 Warning 头域的响应，它应该提示用户一个警告信息。

13.1.2 警告信息（Warnings）

无论何时，缓存返回一个响应，此响应既不是第一手的(first-hand)也不是足够保鲜（在 13.1.1 节的条件 2），那么缓存必须利用一个 Warning 常用头域来警告产生的影响。Warning 头域和当前定义的警告在 14.46 节里描述。这些警告允许客户端去采取合适的动作。

Warning 头域可能被用于缓存相关的和其他的目的。使用 Warning 头域而不是错误状态码，区别于是否是真正的错误。

警告被赋予三位数字警告码（warn-codes）。第一个数字指明：在一个成功的重验证之后，警告是否必须或不必从缓存项里删除。

1xx 警告描述了响应的保鲜或重验证状态，并且这些状态必须在一个成功重验证之后删除。1xx 警告码（warn-codes）可能是由缓存产生的，当验证一个缓存项时。此警告码不能被客户端产生。

2xx 警告描述了实体主体或实体头域的某些方面的信息，这些信息不能被重验证修改（例如，一个实体主体的有损压缩），并且这些信息不能在成功重验证之后被删除。

见 14.46 节关于警告码的定义。

HTTP/1.0 缓存将缓存所有响应中的警告，并且不会删除第一类警告。被要通往到 HTTP/1.0 缓存的响应，存在其响应中的警告携带一个额外的 warning-date 域可防止将来的 HTTP/1.1 接收端信任一错误的已缓存的警告。

警告同样携带一个警告文本。此文本可能是任何合适的自然语义（可能基于客户端请求的 Accept 头域），同时包含一个可选择的关于何种字符集被使用的声明。

多个警告可能会绑定一个响应（或者被源服务器发送或者被一个缓存发送），并且包括多个警告可以共用一个警告码。例如，服务器可能会以英语和法语提供相同的警告。

当多个警告绑定一个响应时，有时候不可能把所有的警告都展示给客户。HTTP 版本没有指定严格优先规则去决定哪个警告优先展示和以何种顺序去展示，但是可以探索一些方法。

13.1.3 缓存控制机制 （Cache-control Mechanism）

HTTP/1.1 基本缓存机制（服务器指定过期时间和验证器）对缓存是隐式指令。某些情况下，服务器或客户端可能需要给 HTTP 缓存提供显式指令。我们利用 Cache-Control 头域为此目的。

Cache-Control 头域允许客户端或服务器在请求或响应里传输多个指令。这些指令常常覆盖缺省的缓存算法。作为一个常用规则，如果头域值中存在一个明显的冲突，那么最多严格解释（most restrictive interpretation）的头域值会被应用（也就是说，能保留语义透明性的值）。然而，一些情况下，cache-control 指令被显式地用来削弱语义透明性的近似性（例如，”max-stale” 或者 “public”）。

Cache-control 指令在 14.9 节被描述。

13.1.4 显式用户代理警告（Explicit User Agent Warnings）

很多用户代理允许用户覆盖基本缓存机制。例如，用户代理可能允许用户指定缓存实体（即使实体明显是陈旧的）永远不要被验证。或者，用户代理可能习惯于给任何请求加上 “ Cache-Control：max-stale=3600”。用户代理不应该缺省的执行非透明行为或导致非正常且无效率的缓存行为，但是可以通过一个显式的用户动作，代理可能会被显式地设置去这样做。

如果用户已覆盖基本缓存机制，那么用户代理无论何时遇到不能满足服务器透明性要求（server’s transparency requirements）的信息被展现时，用户代理应该显式地给用户以指示

（特别是，如果被展现实体被认为是陈旧的）。因为协议通常允许用户代理去判定响应是否是陈旧，所以此指示只需要实际发生时才被展现。此指示不必是对话框；它应该是一个图标（例如，一个正在腐烂的鱼）或者一些其他的指示器。

如果用户以某种方式已覆盖缓存机制，同时这种方式可能非正常地降低了缓存效率，那么用户代理应该继续指示此状态给用户（例如，通过一个图片显示）以便用户能谨慎地消费额外资源或忍受额外延迟。

13.1.5 规则和警告的例外情况（Exceptions to the Rules and Warnings）

在某些情况下，缓存的操作者可以选择去设置返回陈旧的响应，即使此响应没有被客户端请求。这个决定不应该被轻易决定，但是为了实用性或性能方面的要求可能会这样做，特别是当缓存和源服务器连接不好时。无能何时当缓存会返回一个陈旧的响应时，缓存必须给此响应做个标记（利用 Warning 头域），这样能使客户端软件能提醒用户可能会存在潜在的问题。

同样可以允许用户代理去采取步骤获得第一手的或保鲜的响应。由于这个原因，如果客户端显式地请求第一手的或保鲜的响应，缓存就不应该返回一个陈旧的响应，除非由于技术或策略方面的原因而不能遵守。

13.1.6 由客户控制的行为（Client-controlled Behavior）

当源服务器（退一步讲，可以是通过在响应里添加年龄的中间缓存）是过期信息的主要来源时，在某些情况下，客户端可能需要控制缓存决定是否返回一个已缓存的响应而不需要通过服务器验证它。客户端通过利用一些 Cache-Control 头域的指示命令（directives）来达到此目的。

客户端的请求可能会指定自己愿意接受一没有经过验证响应的最大年龄（age）；指定 0 值会强迫缓存重验证所有的响应。客户端同样也会指定在响应过期之前的最小保持时间。这两个选项会增强对缓存行为的限制，因此不能进一步地放松缓存语义透明的近似性（approximation of semantic transparency）。

客户端同样可能会指定它愿意接受（accept）到达的陈旧响应。这放松了对缓存的限制，同时这可能违反了源服务器对语义透明性的限制，但是这可以支持脱机操作（ disconnected operation），或者高可用性（high availability）当连接不好时。

13.2 过期模型 （Expiration Model）

13.2.1 服务器指定过期（Server-Specified Expiratiion）

HTTP 缓存会工作的很好，这是因为缓存能完全地避免客户端对源服务器的请求。避免请求的主要机制是使服务器通过提供一个将来的显式过期时间（explicit expiration time）以指示响应可以满足后续的请求。也就是说，缓存能返回一个保鲜（fresh）的响应而不需要和源服务器接触。

服务器可以给响应赋一个将来的显式过期时间（explicit expiration time），并确信在过期时间之前实体不会改变。这通常能保持语义透明性，只要服务器对过期时间仔细斟酌。

过期机制只能应用于从缓存构建响应，而不能应用于立即转发给客户端的第一手（first-hand，见 1.3 节术语）响应。

如果源服务器希望强制语义透明的缓存去验证每个请求，它给显式过期时间（explicit expiration time）设为过去。这就是说响应总是陈旧的，所以当缓存利用此响应去服务于后续请求时，缓存应该验证此响应。见 14.9.4 节，有更严格的方式来进行重验证。

如果源服务器希望强迫任何 HTTP/1.1 缓存（不管此缓存是怎样设置的）去验证每一个请求，源服务器应该利用 “must-revalidate”缓存控制指令（见 14.9 节）。

服务器指定**显式过期时间**是通过利用 Expires 头域或 Cache-Control 头域里的 max-age 缓存控制指令。

过期时间不能被用于强制客户代理去刷新显示或重载资源；过期的语义只能应用于缓存机制，并且当对资源的发起新请求时，此机制只需检测此资源的过期状态。见 13.13 节，关于缓存和历史机制的差异。

13.2.2 启发式过期

因为源服务器不能总是提供一个显式过期时间（explicit expiration time），HTTP 缓存通常会赋予一个启发式过期时间（heuristic expiration time），它采用一种算法，此算法利用其它头域的值（例如 Last-Modified 时间）去估计一个合理的过期时间。HTTP/1.1 规范没有提供特定的算法，但是却加强了算法结果最坏情况的限制。因为启发式过期时间可能会损坏语义透明性，他们应该被谨慎地使用，并且我们鼓励源服务器尽可能提供显式过期时间（explicit expiration times）。

13.2.3 年龄（Age）计算

为了了解缓存项（译注：缓存项是用来响应请求的，它包含已缓存的响应实体）是否是保鲜的（fresh），缓存需要知道其年龄是否已超过保鲜寿命（freshness lifetime）。我们在 13.2.4 节中讨论如何计算保鲜寿命，本节讨论如何计算响应或缓存项的年龄。

在讨论中，我们利用术语“now”来表示“执行计算主机上时钟的当前值”。使用 HTTP 协议的主机，特别是运行于源服务器和缓存的主机，应该使用 NTP[28] 或其他类似协议来同步其时钟到全球（globally）精确时间标准上。

HTTP1.1 协议要求源服务器尽可能在每个响应里附加一个 Date 头域，并且赋予响应产生的时间（见 14.18 节）。我们利用术语“date_value”去表示 Date 头域的值，这是一种适合于运算操作的表示方法。

当从缓存里获取响应消息时，HTTP/1.1 利用 Age 响应头域来表达响应消息的估计年龄。Age 响应头域值是缓存对响应自从被源服务器产生或重验证到现在的时间估计值。

实际上，年龄的值是响应从源服务器途径每一个缓存的逗留时间的总和，再加上响应在网络路径上传输的时间。

我们用“age_value”来表示 Age 头域的值，这是一种适于算术操作的表示方法。

一个响应的年龄(age)可以通过两种完全独立的途径来计算:：

1． now - date_value，如果本地时钟与源服务器时钟同步的相当好。若结果为负，则取零。

2． age_value，如果途径响应路径（response path）的所有缓存均遵循 HTTP1.1 协议。

如果我们有两种不同的方法计算响应的年龄，我们可以合并二者如下：

corrected_received_age = max（now – date_value，age_value）

并且只要我们有近似同步时钟或全 HTTP/1.1（all-HTTP/1.1）路径，就能得到一个可信赖的（保守的）结果。

由于网络附加延时，会在服务器产生响应与下一个缓存或客户端接收响应之间产生时间延迟。如果不经修订，这一延迟会带来不正常的低年龄。

由于导致产生年龄值的请求是早于年龄值的产生，我们能校正网络附加延迟通过记录请求产生的时间。然后，当一个年龄值被接收后，它必须被解释成相对于请求产生的时间，而不是相对响应接收的时间。不管有多少延迟，此算法会导致保守的结果。所以，我们计算：

corrected_initial_age = corrected_received_age + （now - request_time）

这里“request_time”是请求的发送时间。

当缓存收到响应时，年龄计算的算法总结如下：

```
/*
 * age_value
 *      is the value of Age：  header received by the cache with this response.
 * date_value
 *      is the value of the origin server's Date: header
 * request_time
 *      is the （local） time when the cache made the request
 *             that resulted in this cached response
 * response_time
 *      is the （local） time when the cache received the response
 * now
 *      is the current （local） time
 */

apparent_age = max（0, response_time - date_value）; //缓存收到响应时响应的年龄
corrected_received_age = max（apparent_age, age_value）;
response_delay = response_time - request_time;
corrected_initial_age = corrected_received_age + response_delay;
resident_time = now - response_time; //即收到响应到现在的时间间隔
current_age   = corrected_initial_age + resident_time;
```

缓存项（cache entry）的 current_age 是缓存项从被源服务器最后验证到现在的时间间隔（以秒记）再加上 corrected_initial_age。当从缓存项里产生一条响应时，缓存必须在响应里包含一个 Age 头域，它的值应该等于此缓存项的 current_age 值。

Age 头域出现在响应里说明响应不是第一手的（first-hand）（译注：第一手的说明，响应是直接来自于源服务器到达接收端的，而不是来自于缓存里保存的副本）。然而相反的情况并不成立，因为响应里缺少 Age 头域并不能说明响应是第一手的(fisrt-hand)，除非所有请求路径上的缓存都遵循 HTTP/1.1 协议（也就是说，以前 HTTP 版本缓存没有定义 Age 头域）。

13.2.4 过期计算（Expiration Calculations）

为了确定一条响应是保鲜的（fresh）还是陈旧的（stale），我们需要将其保鲜寿命（freshness lifetime)和年龄(age)进行比较。年龄的计算见 13.2.3 节，本节讲解怎样计算保鲜

寿命，以及判定一个响应是否已经过期。在下面的讨论中，数值可以用任何适于算术操作的形式表示。

我们用术语“expires_value”来表明 Expires 头域的值。我们用术语“max_age_value”来表示 Cache-Control 头域里“max-age”控制指令的值（见 14.9.3 节）。

max-age 指令优于 Expires 头域执行，所以如果 max-age 出现在响应里，那么定义如下：

```
freshness_lifetime = max_age_value
```

否则，若 Expires 头域出现在响应里，定义如下：

```
freshness_lifetime = expires_value - date_value
```

注意上述运算不受时钟误差影响，因为所有信息均来自源服务器。

如果 Expires， Cache-Control：max-age, 或 Cache-Control：s-maxage （见 14.9.3） 均未在响应中出现，且响应没有包含对缓存的其他控制，那么缓存可以用启发式算法计算保鲜寿命（freshness lifetime）。缓存必须对年龄大于 24 小时的响应附加 113 警告，如果此响应不带这种警告。

同样，如果响应有最后修改时间（Last-Modified time），启发式过期值应不大于从那个时间开始到现在这段时间间隔的某个分数。典型设置为间隔的 10% 。

计算响应是否过期非常简单：

```
response_is_fresh = （freshness_lifetime > current_age）
```

13.2.5 澄清过期值（Disambiguation Expiration Values）

由于过期值容易被任意设置，有可能两个缓存包含同一资源的不同保鲜值（fresh values）。

如果客户端执行获取请求接收到一个非第一手的响应，此请求响应在客户端缓存里仍然是保鲜的，并且缓存项里的 Date 头域的值比新响应的 Date 头域值要新，那么客户端应该忽略此新响应。如果这样的话，它可以以“Cache-Control：max-age=0”指令（见 14.9 节）重试请求，从而去强制和源服务器重验证。

如果一个缓存对同一个表现形式（representation）拥有两个保鲜响应却有不同的验证器，那么缓存必须利用 Date 头域值最近的响应。这种情况可能发生由于缓存会缓存来自其它缓存的响应，或者由于客户端已要求重载或重验证一个显然保鲜的缓存项的。

13.2.6 澄清多个响应（Disambiguating Multiple Response）

因为客户端可能收到经多个路径而来的响应，所以某些响应会经过一些缓存集，某些响应会经过其它缓存集，客户端收到响应的顺序可能与源服务器发送响应的顺序不同。我们希望客户端利用最新的响应，即使旧响应仍然是保鲜的。

实体标签（entity tag）和过期值（expiration value）都不能决定响应的顺序，因为可能会出现晚一点的响应故意携带过早的过期时间（expiration time）。日期值的精度被规定只有一秒。

当客户端尝试重验证一个缓存项的时，并且接收到的响应里 Date 头域晚于已存在的缓存项，

那么客户端应该无条件的重试请求，并且包含

Cache-Control: max-age=0

去强制任何中间缓存通过源服务器来验证（validate）它们的缓存副本，或者

Cache-Control: no-cache

去强制任何中间缓存去从源服务器获得一个新的副本。

13.3 验证模型（Validation Model）

当缓存拥有一个旧缓存项并且想利用它来作为客户端请求的响应时，缓存必须首先通过源服务器（或者可能通过一个有保鲜响应的中间缓存）对其进行验证看是否此缓存项可用。我们称做“验证（validating）”此缓存项。我们可以不必花代价来重传完整响应（full response），并且在缓存项无效时可以不必产生额外的回路，因此 HTTP1.1 协议支持使用条件方法（conditional methods）。.

协议支持条件方法（conditional methods）的关键特征是围绕“缓存验证器（cache validator）”展开的。当源服务器产生一个完整响应（full response）时，它同时会附加一些验证器给响应，这些验证器和缓存项一起保存。当客户端（用户代理或缓存）对有缓存项的资源执行条件请求时，客户端在请求里包含一个相关的验证器（validator）。

服务器（有可能是源服务器或缓存服务器）核对请求里的验证器和当前此实体本地验证器是否匹配，如果匹配（见 13.3.3），则返回一个特定状态码（通常为 304（没有改变））的响应并不包含实体主体（entity body）。如果不匹配，服务器就返回完整响应（包含实体主体）。这样，如果验证器匹配，我们就避免了传输完整响应（full response）；同时，如果验证器不匹配，也避免了额外的回路。

在 HTTP1.1 协议中，一个条件请求和普通的请求相似，除条件请求携带一些特殊的头域（这些头域包含验证器），包含这些特殊的头域就隐含地表明请求方法（通常是 GET 方法）为条件请求方法。

协议中缓存验证条件有正向条件和负向条件。也就是说有存在验证器匹配，请求方法会执行；验证器不匹配，请求方法也可能会执行。

注意：缺少验证器的响应可能会被缓存，而且会被缓存用来为请求提供服务直到缓存副本过期，除非用缓存控制指令显式地禁止缓存这样去做。然而，如果缓存没有实体的验证器，那么缓存不能执行条件方法来获取资源，这就意味着在缓存副本过期之后不会得到刷新。

13.3.1 最后修改日期 （Last-Modified Dates）

Last-Modifed 实体头域值经常被用作一个缓存验证器。简而言之，如果实体自从 Last-Modifed 值之后没有改变，那么缓存项被认为是有效的。

13.3.2 实体标签缓存验证器（Entity Tag Cache Validators）

ETag 响应头域值是实体标签，它提供了一个“晦涩（opaque）”缓存验证器。当在不方便保存修改日期时，当在 HTTP 日期值的一秒精度不能满足需要时，或当源服务器希望避免使用修改日期产生的冲突时，通过实体标签能得到更可靠的验证，。

实体标签在 3.11 节描述了。使用了实体标签的头域在 14.19，14.24，14.26 和 14.44 节里描述了。

13.3.3 强,弱验证器 （Weak and Strong Validators）

由于源服务器和缓存会比较两个验证器来确定他们是否代表相同的实体，所以通常希望实体发生任何变化时验证器也相应变化，这样的验证器为强验证器。.

然而，可能存在这样的请求，服务器倾向于仅在实体发生重要的语义变化时才改变验证器，而在实体的某些方面不发生重大改变时就不改变验证器。在资源变化时验证器未必变化的验证器称为弱验证器.。

实体标签通常是强验证器，但协议提供一种机制来使实体标签变成弱验证器。可以认为强验证器在实体的每一字节变化时而变化，而弱验证器仅在实体的语义变化时才变化。换言之，我们能认为强验证器是特定实体的标识，而弱验证器是同一类语义等价实体的标识。

注: 强验证器的例子：一个整数他会随着每次实体发生变化而递增。一个实体的修改时间，如果以秒为精度，能被看作为弱验证器，因为在一秒内资源可能改变两次。对弱验证器的支持是可选择的。然而，弱验证器允许更有效地缓存等价对象。

.

客户端产生请求并把验证器包含在一个验证头域（validating header fiield）里时或服务器比较两个验证器的时候均用到验证器。强验证器可在任何情况下使用，而弱验证器仅在不依赖于严格相等时才可用。当客户端产生条件 GET 请求来请求一个完整实体时，任何类型的验证器都可以使用。然而，子范围（sub-range）请求时只能使用强验证器，否则客户端可能会得到一个不一致的实体。

客户端可以在发出简单（非子范围）GET 请求里既可以利用弱验证器也可以利用强验证器。客户端不能利用弱验证器在其它的请求形式里。

HTTP1.1 协议定义验证的唯一功能就是比较。有两种验证器比较方法，这依赖于比较的背景是否允许利用弱验证器。

.

- 强比较方法：如果相等，两验证器必须完全一致，并且两个验证器都是强验证器。.
- 弱比较方法：如果相等，两验证器必须完全一致，但它们中的任何一个或全部被标明“弱”（“weak”）的不会影响结果。

实体标签是强验证器除非它被显式地标记为弱(weak)的。3.11 节给出了实体标签的语法。

最后修改时间（Last-Modifed 头域的值）被用作请求的验证器时默认为弱验证器，除非满足下列规则才判定它是强验证器：.

- 此验证器正在被源服务器用来和当前实体验证器进行比较，并且
- 源服务器知道相关的实体不会在当前验证器涵盖的秒内改变两次，

或者

- 此验证器即将被客户端用于 If-Modified-Since 或者 If-Unmodified-Since 头域里，因为客户端有一个关于实体的缓存项，并且
- 缓存项包含一个日期值（Date value），日期值给出了源服务器发送源响应的时间，并且
- Last-Modifed 头域值至少提前于日期值（Date value）60 秒。

或者

- 此验证器正在被中间缓存通过与其缓存项里保存的验证器相比较 ，并且
- 缓存项包含日期值（Date value），它指明了源服务器发送源响应（original response）的时间，并且
- Last-Modifed 头域值至少提前于日期值（Date value）60 秒。

此种方法依赖于以下事实，如果两个不同响应被源服务器在同一秒内被发出，但这两个响应都有相同的最后修改时间（Last-Modified time），那么其中至少有一个响应的日期值和最后修改时间的值（Last-Modifed 的值）相等。60 秒的限制能保证 Date 和 Last-Modifed 的值在不同时钟或在响应准备期间的不同时刻产生。一个实现可能会利用大于 60 秒的值，如果它认为 60 秒太短。

如果客户端希望执行子范围(sub-range)请求来请求一个只有最后修改（Last-Modifed）时间但没有晦涩（opaque）验证器时，它可能会认为只有最后修改（Last-Modified）时间是强的。

若缓存或源服务器接收到一条件请求，而不是完整响应 GET 请求时，他必须使用强比较方法去计算条件。

此规定允许 HTTP1.1,缓存和客户端能安全地执行子范围（sub-range）请求获得来自 HTTP/1.0 得来的值。

13.3.4 关于何时使用实体标签和最后修改时间的规则

我们对源服务器，客户端和缓存采用一套规则和建议来规定不同的验证器何时被使用，出于何种目的被使用。

HTTP/1.1 源服务器：

- 应发送一个实体标签验证器，除非产生此实体标签不可行。

- 可以发送弱实体标签来替代强实体标签，如果使用弱实体标签能提高性能或者不能发送强实体标签。

- 应发送一 Last-Modifed 值如果可以的话，除非打破语义透明性（这可能由于利用此日期于 if-Modified-Since 头域里）会导致严重的后果。

换句话说，对 http1.1 源服务器来说，比较好的做法是同时发送强实体标签和 Last-Modified 值。.

为了合法，强实体标签必须随相关联的实体值改变而改变。弱实体标签应该随相关联的实体在语义上发生改变而改变。

注意：为保证语义透明缓存，源服务器必须避免为两个不同的实体重用某个特定的强实体标签值，也不能重用某特定弱实体标签为两个语义不同的实体。缓存项应该能保持任意长的时间，而不管过期时间（expiraton time），所以不能假设缓存从来不会尝试利用以前获得的验证器来验证缓存项。

HTTP/1.1 客户端：

- 若实体标签被源服务器提供，HTTP/1.1 客户端必须在任何缓存条件请求（利用 If-Match 或 If-None-Match）里利用实体标签.。

- 仅 Last-Modified 值被源服务器提供时，HTTP/1.1 客户端应在非子范围缓存条件请求（利

用 If-Modified-Since）里利用此值。

- 仅 Last-Modified 值被 HTTP/1.0 源服务器提供时，HTTP/1.1 客户端可能会在子范围缓存条件请求（利用 If-Unmodified-Since）里利用此值。

- 如果实体标签和 Last-Modified 值都被源服务器提供，HTTP/1.1 客户端应该在缓存条件请求里利用这两个验证器。这允许 HTTP/1.0 和 HTTP/1.1 缓存能合适地进行响应。

HTTP/1.1 源服务器，当接收到一个条件请求并同时包含 Last-Modifed 日期（例如，在 If-Modified-Since，或 If-Unmodified-Since 头域里）和一个或多个实体标签（例如在 If-Match，If-None-Match，或 If-Range 头域里）作为缓存验证器时，源服务器不能返回一个 304 状态响应（没有改变）除非这样作能与请求里所有的条件头域一致。

HTTP/1.1 缓存，当接收到一个条件请求并且此请求里同时包含 Last-Modified 日期和一个或多个实体标签作为缓存验证器时，它不能返回一个本地已缓存的响应给客户端，除非已缓存的响应与请求里所有条件头域一致。

注意：这些原理背后的规则是 HTTP/1.1 服务器和客户端应在请求和响应里尽可能传输非冗余的信息。接收这些信息的 HTTP/1.1 系统将会对这些接收到的验证器作最保守的假设。

HTTP/1.0 客户端和缓存会忽略实体标签。通常，Last-Modified 值会被这些系统接收和利用，以提供透明性和高效缓存行为，因此 HTTP/1.1 源服务器这时应提供 Last-Modified 值。在一下很少的情况，一个 Last-Modified 值作为验证器被 HTTP/1.0 系统使用时会带来严重的后果，这时 HTTP/1.1 服务器不应提供一个 Last-Modified 值。

13.3.5 非验证条件（Non-validating Conditionls）

实体标签背后的原则是只有服务作者才知道资源的语义从而去选择一个合适的缓存验证机制，并且任何比字节相等（byte-equality）比较方法复杂的验证器比较方法都会带来风险。所以，任何其他的头域的比较（除了 Last-Modified，为了兼容 HTTP/1.0）从来不会被用于验证一个缓存项。

13.4 响应的可缓存性（Response Cacheability）

除非被缓存控制（见 14.9 节）指令明确地限制，缓存系统可以将一成功响应作为缓存项，可以返回缓存项里的响应副本而不需要验证它如果此副本是保鲜的，并且也可以在验证成功后返回它。如果响应既没有缓存验证器也没有显式过期时间，我们认为它不能被缓存，但是某些缓存可能会违反这个约定（例如，当处于脱机时）。客户端能经常发现这种来自于缓存的响应，只需通过把 Date 头域值同当前时间作比较。

注意：某些 HTTP1.0 缓存可能违反这一假设而没有提示任何警告。

然而，在一些情况下，缓存不适合保存实体，或使缓存的实体服务于后续请求。这可能因为服务的作者认为绝对的语义透明性是有必要的，或着出于安全和隐私考虑。某些缓存控制指令因此被提供是为了让服务器能指示某些资源实体，或部分实体，不能被缓存。

注意在 14.8 节里描述了防止一个共享缓存（shared cache）去保存和返回一个以前包含 Authorization 头域请求的响应。

除非缓存控制指令防止响应被缓存，一个接收的响应如果它的状态码是 200，203，206，300，301 或 410，那么此响应应该被缓存保存而且可用于后续的请求，但

这必须受过期机制决定。然而，如果缓存不支持 Range 和 Content-Range 头域，那么它不能缓存 206 响应（部分内容）响应。

缓存接收到的响应如果是其他的状态码（如，302 和 307），那么此响应不能被用于服务于后续的请求，除非缓存控制指令或其他的头域显式地允许它能这样做。例如，这些头域包含下面的头域：Expires 头域（见 14.21）；“max-age”，“s-maxage”，“must-revalidate”，“prox-revalidate”，“public”或“private”缓存控制指令（见 14.9）。

13.5 从缓存里构造响应

缓存的目的是为了响应将来的请求而缓存请求的响应信息。在很多情况下，缓存简单地返回响应的某部分给请求者。然而，如果缓存拥有一个基于以前响应的缓存项，它可能必须把新响应的部分和它缓存项里的内容合起来。

13.5.1 End-to-end 和 Hop-by-hop 头域

为定义缓存和非缓存代理的行为，我们将 HTTP 头域分成两类：

- end-to-end 头域，他们被传输给最终请求或响应的接收者。响应里 end-to-end 头域必需作为缓存项的一部分存储，并且必须在任何缓存项构成响应里被传输。
- hop-by-hop 头域，他们只对传输层连接有意义，并且即不能被缓存保存也不能被代理转发。

下面的 HTTP/1.1 头域是 hop-by-hop 头域：

- Connection
- Keep-Alive
- Proxy-Authenticate
- Proxy-Authorization
- TE
- Trailers
- Transfer-Encoding
- Upgrade

所有其它被 HTTP/1.1 定义的头域均为 end-to-end 头域。

其它 hop-by-hop 头域必须在 Connection 头域（14.10 节）里列出，并将被引进于 HTTP/1.1（或后来的版本）里。

13.5.2 不可更改的头域 （Non-modifiable Headers）

HTTP1.1 的某些特征，如数字认证（Digest Authentication），是基于某些 end-to-end 头域。一个透明代理不应该改变 end-to-end 头域，除非这些头域的定义要求或允许这样做。

一个透明代理不能改变请求或响应里的下面头域，而且它不能添加这些头域到没有这些头域的请求或响应里：

- Contents-location
- Content-MD5
- ETag
- Last-Modified

一个透明代理不能改变响应里的下面头域：

- Expires

但它可以添加这些头域如果响应里没有这些头域时。如果一个 Expires 头域被添加，它必须等于响应里 Date 头域的值。

一个代理不能在包含 no-transform 缓存控制指令的消息中改变或添加下面的头域。

- Content-Encoding
- Content-Range
- Content-Type

一个非透明代理可能会在一个消息里改变或添加这些头域如果消息不包含 no-transform 缓存控制指令，但是如果代理这样做了，它必须添加一个警告 214（转换被应用）（见 14.46 节）。

注意：end-to-end 头域的不必要更改可能会导致认证失败，如果更强的认证机制被应用于后续的 HTTP 版本中。此认证机制可能依赖于没有在此列举的头域值。

请求或响应里的 Content-Length 头域会根据 4.4 节的规则被添加或被删除。一个透明代理必须保留实体主体的 entity-length（见 7.2.2），虽然它可以改变 transfer-length（4.4 节）。

13.5.3 联合头域（Combining Headers）

当一个缓存对服务器发出验证请求时，而且服务器提供 304（没有改变）响应或 206（部分内容）响应时，那么缓存将构造一个响应发送给请求客户端。

如果状态码是 304（没有改变），缓存利用缓存项里的实体主体（entity-body）作为客户端请求响应的实体主体。如果响应状态码是 206（部分内容）并且 Etag 或 Last-Modified 头域能精确匹配，那么缓存可能把缓存项里的内容和接收到的响应里的新内容合并并且利用最后合并的结果作为输出响应（见 13.5.4）。

缓存项里的 end-to-end 头域被用于构造响应，除了：

- 任何保存的警告码是 1xx（见 14.46）的 Warning 头域必须从缓存项和转发的响应里删除。
- 任何保存的警告码是 2xx 的 Warning 头域必须要在缓存项和转发的响应里保留。
- 任何 304 或 206 响应提供的 end-to-end 头域必须替换缓存项的相应头域

除非缓存决定去删除缓存项，否则它必须同样能用接收的响应里的相应 end-to-end 头域去替换缓存项里的头域，除了上面描述的 Warning 头域。如果输入响应里的一个头域匹配缓存项里多个头域，那么所有这些旧的头域必须被替换。

也就是说，输入响应的所有 end-to-end 头域会覆盖缓存项里所有相应的 end-to-end 头域（除了缓存的警告码是 1xx 的 Warning 头域，它将会被删除即使没有被覆盖）。

注意：此规则允许源服务器去利用 304（没有改变）或一个 206（部分内容）响应去更新以前同一实体或子范围实体响应的任何头域，虽然它也许不总是有意义或正确。这条规则不允许源服务器去利用 304（没有改变）或 206（部分内容）响应去完全地删除一个以前响应的头域。

13.5.4 联合字节范围（Combing Byte Ranges）

一条响应可能仅传送一个实体主体的某一部分，这是由于请求包含一个或多个 Range 指定的范围，或者由于连接会被过早地断开。在几次这样得传输后，缓存可能已经接收了同一个实体

主体的多个范围部分。

如果缓存有一个实体的非空子范围，并且一个输入（incoming）响应（译注：输入响应是进入缓存的响应，输出响应是从缓存出去的响应）携带了另一子范围，那么缓存可能会把新的子范围和已经存在的子范围联合起来如果下面两者同时满足：

- 输入响应和缓存项都有缓存验证器。
- 利用强比较方法使两个缓存验证器匹配（见 13.3.3）。

如果任何要求不能满足，缓存必须利用最接近现在（most recent）的部分响应（基于任何响应的 Date 头域值，并且会利用输入响应如果这些 Date 头域值相等或丢失了），而且必须丢弃其它的部分信息。

13.6 缓存已协商响应（Caching Negotiated Responses）

使用服务器驱动内容协商（12.1 节），会在响应里添加 Vary 头域，并改变缓存利用响应去服务后续请求的条件和过程。见 14.44 节关于服务器利用 Vary 头域的描述。

服务器应利用 Vary 头域告诉缓存哪些请求头域被服务器用于从响应（可缓存的）对应的多个表现形式里进行选择，响应由服务器驱动协商决定。Vary 头域里指定的头域被称做选择请求头域（selecting request-header）。

当缓存接收到一个后续请求，并且此请求的 URI 指定了一个或多个（包含一个 Vary 头域的）缓存项时，缓存不能利用这个缓存项去构造响应从而服务于新的请求，除非所有出现在新请求里的选择请求头域匹配来自于源请求里已被缓存保存的选择请求头域。

在两个请求里，我们定义这两个选择请求头域匹配，如果并且只有第一个请求的选择请求头域能被转换为第二个请求里的头域通过添加或删除线性空白（被允许出现在相应的 BNF 里的线性空白）和/或把多个消息头域结合成一个头域通过 4.2 节里的规则。

一个 Vary 头域值是“*”总是不能匹配的，并且后续对那个资源的请求只能合适地被源服务器理解。

如果缓存项里的选择请求头域（selecting request-header）不能匹配新请求的选择请求头域，那么缓存不能利用缓存项去满足请求除非它能以条件请求把此新请求接力到源服务器并且源服务器返回一个 304（没有改变）的状态响应，并包含一个实体标签或一个指明被使用实体的 Content-Location 头域。

如果一个缓存的表现形式已拥有实体标签，那么转发请求应是以条件请求发送并且应包含资源对应的所有缓存项的实体标签于 If-None-Match 头域（不匹配则执行方法）里。这告诉了服务器当前缓存拥有的实体集，以便如果实体集里的任何实体匹配请求的实体，那么服务器能利用 Etag 头域于 304（没有改变）响应里，从而去告诉缓存哪个缓存项是合适的。如果新响应的实体标签匹配已存在的缓存项，那么新响应应被利用去更新已经存在的缓存项的头域，而且更新的最后结果必须返回给客户端。

如果任何已存的缓存项只包含相关实体的部分内容，那么它的实体标签不应被包含于 If-None-Match 头域里，除非这个请求是范围请求并且缓存项能完全满足请求。

如果缓存接收到一成功响应，响应的 Content-Location 头域匹配同一请求 URI 已存缓存项的 Content-Location 头域，并且它的实体标签不同于已存缓存项的实体标签，而且它的 Date 头域值比已存缓存项更接近当前，那么已经存在的缓存项不能被返回去响应将来的请求并且将会

从缓存里删除。

13.7 共享和非共享缓存 （Shared and Non-Shared Caches）

出于安全和保密考虑，有必要区分共享和非共享缓存。非共享缓存是仅供一个用户访问，此情况下，访问性（accessibility）应由适当的安全机制控制。所有其它缓存均被认为是共享的。此协议的其它部分对共享缓存的操作添加限制以防止隐私丢失或访问控制的失败。

13.8 错误和不完全的响应缓存行为

缓存收到不完整响应（例如响应的字节数比 Content-Length 头域指定的值要小）时也可以保存，但是必须把此响应看作部分响应。部分相应可以合并（见 13.5.4）；合并结果可能是完整响应或仍是部分响应。.缓存在没有显式标明响应是部分响应情况下（例如利用 206（部分内容）状态响应）不能把部分响应返回给客户端。缓存不能使用一个 200（OK）状态码返回一个部分响应。

如果缓存当试图重验证一个缓存项时收到一 5xx 响应，那么它既可以将此响应转发给请求的客户端，或者执行跟服务器响应失败一样。在后面的情况下，它可以返回一个以前的接收的响应，除非缓存项包含一个“must-revalidate”缓存控制指令（见 14.9 节）。

13.9 GET 和 HEAD 的副作用（Side Effects of GET and HEAD）

除非源服务器显式地禁止缓存它们的响应，否则缓存对任何资源的 GET 和 HEAD 方法不应有导致错误的行为的副作用（side effects）如果这些响应来自于缓存。他们可以仍然有副作用，但缓存在决定缓存时不必考虑这些副作用。缓存总是被期望去观察一源服务器对缓存的显式限制（explicit restrictions）。

我们注意到此规则的一个例外：有些应用程序习惯于在在 GETs 和 HEADs 方法里使用查询 URLs（在 rel_path_part 里包含一个“?”）从而执行带来很大的副作用的操作，缓存不能把此 URIs 的响应看作一个保鲜的，除非服务器提供一个显式过期时间（explicit expiration time）。这就意味着，对这样的 URIs 的，来自 HTTP/1.0 服务器的响应不应被缓存保存。见 9.1.1 节相关的信息。

13.10 在更新或删除后的无效性

在源服务器上，某些方法对某资源执行的影响，可能会引起一个或多个已存缓存项的非透明的无效。也就是说，虽然他们可能会继续是保鲜的，但是他们不能准确的反应出源服务器对那个资源的新请求的响应。

HTTP 协议无法保证所有此类缓存项均被标明无效。例如，引起源服务器上资源变化的请求可能不会穿过存有一个缓存项的代理。然而，却有一些规则帮助减少可能的错误行为。

在此节里，短语“使实体无效（invalidata an entity）”意味着缓存要么可以删除它保存的所有的实体的实例，要么可以把这些实体实例标记为“无效”并在它们可作为后续请求的响应之前进行重验证。

一些 HTTP 方法必须让缓存去使一个实体无效（invalidate an entity） 。这些实体要么被请求 URI 指定，要么在 Location 或在 Content-Location 头域里被指定（如果出现的话）。这些方法

是：

- PUT
- DELETE
- POST

为了防止服务器攻击拒绝，一个基于 Location 或 Content-Location 头域里的 URI 的无效处理必须只有在 URI 的 host 部分和请求 URI 里的 host 部分相同时才被执行。

一个缓存如果不能理解请求里的方法，那么它应该使请求 URI 指定的任何实体无效。

13.11 强制写通过（ Write-Through Mandatory）

所有可能对源服务器资源进行修改的方法都要写通过（written through）给源服务器。这通常包括所有除了 GET 和 HEAD 的方法。缓存在将此种请求转发给服务器并获得相应响应前不能对请求客户端做出响应。 但这个不能阻碍代理缓存在服务器已发送最终响应（final replay）之前发送 100（继续）响应。

相反的情况（通常叫“写回”或“拷贝回”缓存）在 HTTP1.1 中是不允许的，这是由于保持一致的更新是非常困难的，并且在写回之前也会存在服务器，缓存和网络故障的问题。

13.12 缓存替换 （Cache Replacement）

如果一个新可缓存（见 14.9.2, 13.2.5, 13.2.6 和 13.8)响应从一资源被缓存接收，并且同一资源的已存响应已经被缓存保存，那么缓存应该利用新响应去响应当前请求。缓存可能会把这一新响应放进存储里，并且可以，如果它满足所有其它要求，利用此响应来响应任何将来的请求。如果缓存想把此新响应加进缓存存储，13.5.3 的规则必须应用。

说明：一个新响应如果其 Date 头域值比已存已缓存的响应的 Date 头域值要旧，那么它是不可缓存的。

13.13 历史列表 （History Lists）

用户代理经常使用历史机制（history mechanisms），如“后退”按钮和历史列表，来重新展示一个会话的先前实体。

历史机制和缓存机制是不同的。特别是历史机制不应尝试给资源当前状态展示一语义透明视图。其历史机制只是为了展示资源获取当时用户看到的东西。

默认情况，过期时间（expiration time）没有应用于历史机制（history mechanisms）。如果实体仍然在存储里，历史机制应该对其显示即使其实体已经过期了，除非用户专门地设置代理去刷新已过期的历史文档。

这不能防止历史机制告诉用户某视图可能过期。

注意：如果历史机制没必要地阻止了用户查看陈旧资源，那么这会强制服务作者避免利用 HTTP 过期控制和缓存控制当他们想利用时。服务作者可以认为用户不被呈现错误消息或警告消息是非常重要的，当他们利用导向按钮（如回退按钮）去看以前获得的资源时。即使有时这些资源本不应被缓存保存，或应很快过期，用户界面的考虑可能会强制服务作者去寻求其它防止缓存的方法（例如，“一次性”URLs）为了避免不正确的历史机制功能的影响。

14 头域定义

本节定义了所有 HTTP/1.1 种标准头域的语法和语义。对于实体头域来说，发送者和接收者都既可以指客户端也可以指服务器，取决于谁发送和谁接收此实体。

14.1 Accept

Accept 请求头域被用于指定哪些媒体类型的响应对请求端是可接受的。Accept 头域被用于指明请求只对某些期望的媒体类型有效，例如请求一个内嵌的图像。

```
Accept        = "Accept" ":"
                          #（ media-range [ accept-params ] ）

media-range    = （ "*/*"
                          | （ type "/" "*" ）
                          | （ type "/" subtype ）
                          ） *（ ";" parameter ）
accept-params  = ";" "q" "=" qvalue *（ accept-extension ）
accept-extension = ";" token [ "=" （ token | quoted-string ） ]
```

星号”*”字符用于把媒体类型组合成一个范围，“*/*”指明了所有的媒体类型而“type/*”指明 type 类型的所有子类型。Media-range 可能包含一个媒体类型参数。

每一个 media-range 可能会跟随一个或多个 accept-params，以“q”参数指明一个相对的喜爱程度的质量因子。通过第一个“q”参数（如果有的话）把 accept-params 和 media-range 参数分离。喜爱程度质量因子允许用户或用户代理去指明对那个 media-range 的相对喜爱程度，qvalue 的范围是从 0 到 1（见 3.9 节）。缺省是 q=1。

注意：利用“q”参数名字将媒体类型参数（译注：media-range 里的 parameter）和 accept-extension 分离开来是基于历史的实践。尽管这能防止任何以“q”命名的媒体类型参数应用于 media-range 里，但在一个 media-range 里使用“q”被认为是不可能发生的，这是因为在 IANA 的媒体类型注册表里是没有“q”参数的并且在 Accept 头域里利用媒体类型参数也是很少见。将来的媒体类型不被鼓任何以“q”命名的参数注册。

例子:：

```
Accept :audio/*;q=0.2 , audio/basic
```

该例应该被解释成“我喜欢 audio/basic，但是可以给我发送任何 audio 类型如果它最容易得到，但在喜爱程度质量要下降 80%”。

如果没有 Accept 头域出现，那么会假设客户端能接受所有媒体类型。如果 Accept 头域在请求消息里出现，并且如果服务器根据联合 Accept 头域值发现它不能发送客户端可接受的响应，那么服务器应发送 406（不可接受的）响应。

一个更加详尽的例子如下：

```
Accept: text/plain; q=0.5, text/html, text/x-dvi; q=0.8, text/x-c
```

这可能被口头地解释成“text/html 和 text/x-c 是更喜爱的媒体类型，但是如果他们不存在，那么发送 text/x-dvi 实体，但如果 text/x-dvi 也不存在，那么发送 text/plain 实体。”

Media-range 能被更具有特殊性的 media-range 或媒体类型覆盖。如果多个 media-range 应用了一个特指的类型，那么最具有特殊性的应该优先。例如：

Accept: text/*， text/html, text/html; level=1, */*

拥有下面的优先顺序：
1） text/html; level=1
2） text/html
3） text/*
4） */*

一个媒体类型的喜爱程度质量因子是和一个给定的媒体类型联系在一起的，它是由查找能最高优先匹配那个媒体类型的 media-range 决定的。例如：

Accept:: text/*; q=0.3, text/html; q=0.7, text/html; level=1,
text/html; level=2; q=0.4, */*; q=0.5

可能会引起下面值被联系：

text/html;level=1 = 1
text/html = 0.7
text/plain = 0.3
image/jpeg = 0.5
text/html;level=2 = 0.4
text/html;level=3 = 0.7

注意：一个用户代理可能会为一个特定的 media-range 提供一个缺省的质量值的集合。然而，除非用户代理是一个不能和其他的呈现代理交互的封闭的系统，否则这个缺省的集合应该可以被用户可设置的。

14.2 Accept-Charset

Accept-Charset 请求头域可以用来指名哪些字符集的响应对请求端是可接受的。Accept-Charset 头域使客户端能通知服务器产生哪些能让客户端更理解的字符集响应。

Accept-Charset = "Accept-Charset" ":"
1#（ （ charset | "*" ）[";" "q" "=" qvalue] ）

字符集值在 3.4 节里描述。每一个字符集可能被给于一个相联系的质量值用来表示用户对那个字符集的喜爱程度。缺省值是 q=1.例如：

Accept-Charset:: iso-8859-5, unicode-1-1;q=0.8

如果特殊值“*”出现在 Accept-Charset 头域里，那么将匹配任何 Accept-Charset 头域里没有的字符集（包含 ISO-8859-1）。如果 Accept-Charset 头域里没有出现“*”出现，那么所有没有在 Accept-Charset 头域里显式声明的字符集的质量值都为 0，但是有个例外，那就是如果 ISO-8859-1 没有被显式声明，那么它的质量值为 1。

如果 Accept-Charset 头域没有出现，那么缺省情况是任何字符集会接受。如果 Accept 头域出现在请求消息里，并且如果服务器不能发送请求端期望的字符集（Accept-Charset 头域指定的）的响应，那么服务器应发送一个 406（不能接受的）错误状态响应，尽管发送一个不可接受的响应也是允许的。

14.3 Accept-Encoding

Accept-Encoding 请求头域和 Accept 头域相似，但 Accept-Encoding 是限定服务器返回给客户端可以接受的内容编码（content-coding，见 3.5 节）。

```
Accept-Encoding  = "Accept-Encoding" ":"
                   1#（ codings [ ";" "q" "=" qvalue ] ）
codings          = （ content-coding | "*" ）
```

使用的例子如下：

```
Accept-Encoding: compress, gzip
Accept-Encoding:
Accept-Encoding: *
Accept-Encoding: compress;q=0.5, gzip;q=1.0
Accept-Encoding: gzip;q=1.0, identity; q=0.5, *;q=0
```

服务器判断一个内容编码（content-coding）是否是可接受的，是根据 Accept-Encoding 头域，并利用下面的规则来决定：

1. 如果一个内容编码（content-coding）在 Accept-Encoding 头域里出现，那么它是可以接受的（acceptable），除非它的 qvalue 为 0。（这在 3.9 节里定义，一个 qvalue 为 0 说明是“不可接受的”）
2. 如果“*”出现在 Accept-Encoding 头域里，那么它匹配任何没有出现在 Accept-Encoding 头域里的可得内容编码。
3. 如果多个内容编码是可接受的，那么 qvalue 为最高的且非 0 的内容编码是最喜欢的。
4. “identity”内容编码总是可接受的，除非 qvalue 为 0，或者 Accept-Encoding 头域包含“*;q=0”并且同时没有包含“identity”内容编码。如果 Accept-Encoding 头域值为空，那么只有“identity”编码是可接受的。

如果 Accept-Encoding 头域在请求里出现，并且如果服务器不能发送一个 Accept-Encoding 头域里指定的编码响应，那么服务器应该发送一个 406（不接受的）错误的响应。

如果没有 Accept-Encdong 头域出现在请求消息里，服务器应该假设客户端将接受任何内容编码。在这种情况下，如果“identity”是这些可得的内容编码中的一个，那么服务器应利用“identity”内容编码，除非服务器有附加信息指明其它内容编码对客户端是有意义的。

注意：如果请求没有包含 Accept-Encoding 头域，并且如果“identity”内容编码是不可得，那么通常能被 HTTP/1.0 客户端容易理解的内容编码（也就是说，“gzip”和“compress”）是更喜爱的；一些不能合适展示消息的老客户端会发送其它内容编码。服务器也许同样能以特定用户代理或客户端的信息来做决定。

注意：大多数 HTTP/1.0 应用程序不能识别或遵循一个内容编码跟随一个 qvalue。这意味着 qvalue 可能不能工作，并且不能被允许和 x-gzip 或 x-compress 在一起。

14.4 Accept-Language

Accept-Language 请求头域和 Accept 请求头域类似，但是它是限定服务器返回给客户端喜爱的自然语言。

```
Accept-Language = "Accept-Language" ":"
                  1#（ language-range [ ";" "q" "=" qvalue ] ）
language-range  = （ （ 1*8ALPHA *（ "-" 1*8ALPHA ） ） | "*" ）
```

每个 language-range 均被赋以一个质量值，它代表用户对此 language-range 里涵盖语言的喜爱程度。质量值缺省为 1，例如：

Accept-Language: da, en-gb;q=0.8, en;q=0.7

好像在说：“我更喜欢 Danish，但是也可以接收 British English 和其他的 English 的类型的语言。”一个 language-range 匹配一个语言标签如果它能精确和语言标签相等，或者它能精确匹配标签前缀，标签前缀是语言标签里字符“-”之前的部分。特殊的“*”字符出现在 Accept-Language 头域里，表明能匹配任何不在此头域里的语言标签。

注意：前缀匹配规则并不是意味着：给语言标签赋值时，如果用户理解某一个标签，那么他同样会理解所有以这个标签作为前缀的所有标签。这种情况下，前缀规则只是表明可以允许使用前缀标签匹配。

语言标签的质量因子是 Accept-Language 头域里匹配此语言标签的 language-range 的质量值。如果没有 Accept-Language 头域里 language-range 匹配的语言标签，那么此语言的质量因子被赋予 0。如果没有 Accept-Language 头域出现在请求里，那么服务器应该假设所有语言将是请求端可接受的。如果一 Accept-Language 头域出现在请求里，那么所有质量因子大于 0 的语言是可接受的。

如果发送一个和用户喜爱的语言相反，这将在 15.1.4 里讨论。

由于各个用户的理解程度不一样，建议客户端应用程序应让用户对语言的偏好进行选择。如果客户不能进行选择，那么 Accept-Language 头域不能在请求里给出。

注意：当让用户作出选择时，用户可能不熟悉上述语言匹配的细节，所以应该提供合适的向导。例如，用户可能会认为选择“en-gb”会提供任何类型的英语文档如果 British English 不可得。在这种情况下，用户代理应该能建议用户添加一个“en”去得到最佳匹配行为。

14.5 Accept-Range

Accept-Range 响应头域允许服务器向客户指明服务器对范围请求的接受度。

```
Accept-Ranges     = "Accept-Ranges" ":" acceptable-ranges
acceptable-ranges = 1#range-unit | "none"
```

源服务器如果接受字节范围请求(byte-range request)那么可以发送

Accept-Ranges: bytes

但是不是必须这样做。客户端在没有接收此头域时也可以产生字节范围请求（byte-range request）。范围单位（range units）被定义在 3.12 节。

服务器如果不能接受任何类型的范围请求（range request），将会发送

Accept-Ranges：none

去劝告客户不要尝试范围请求（range request）。

14.6 Age

Age 响应头域表示发送者对响应产生（或重验证）时刻后经过的时间的估计。一个已缓存的响应是保鲜的（fresh）如果此响应的年龄没有超过它的保鲜寿命（freshness response）。Age 值怎样计算在 13.2.3 节里描述了。

```
Age = "Age" ":" age-value
age-value = delta-seconds
```

Age 值是十进制非负整数，并且以秒为单位.。

如果缓存接收到一个 Age 值大于它所能表示的上限，或它的年龄计算出现溢出，那么它必须传送 Age 头域的值为 2147483648 （2^31）。一个包含缓存的 HTTP/1.1 服务器必须在来自于自身缓存的响应里包含一个 Age 头域。缓存应利用一个至少 31 位的运算类型。

14.7 Allow

Allow 实体头域中列出了被请求 URI（Request-URI）指定的资源所支持的方法。此头域的目的是严格地让接收端知道资源所适合的方法。在 405（方法不被允许）响应中必须出现 Allow 头域。

```
Allow = "Allow" ":"  #Method
```

使用示例：

```
Allow: GET, HEAD, PUT
```

这一头域不能阻止客户端使用其他方法。但在 Allow 头域域中给出的方法应被执行。Allow 头域里指定的方法是每次请求时被源服务器定义的方法。

Allow 头域里可以和一 PUT 请求一起使用，而为了说明新的或改变的资源支持这些方法。服务器不需要去支持这些方法，服务器应包含一个 Allow 头域在响应里并且给出实际支持的方法。

代理（proxy）不能改变 Allow 头域即使它没有理解此头域里指定的所有方法，因为用户代理可能和源服务器通信有其他的意图。

14.8 Authorization （授权）

用户代理往往希望通过服务器给自己授权，用户代理这样做是通过在请求里包含一个 Authorization 请求头域，但是通常在接收了一个 401 响应后就没有必要再让服务器给自己授权了。Authorization 头域由包含用户代理对请求资源域的授权信息的证书（credentials）组成。

```
Authorization  = "Authorization" ":" credentials
```

HTTP 访问授权在“HTTP Authenticatiion：Basic and Digest Access Authentication”[43]中描述。如果一个请求被授权并且一个域（realm）被指定，那么这个证书应对此域里所有的其他请求是有效的（假设在此授权模式本身不需要其它例如根据激发值或利用同步时钟而变化的证书）。

当一个共享缓存（shared cache）（见 13.7 节）接收一个请求，并且此请求包含一个 Authorization 头域时，那么缓存不能返回此请求相应的响应来响应任何其它的请求，除非下面指定的例外之一发生：

1. 如果此响应包含“s-maxage”缓存控制指令，那么此缓存可以利用此响应来响应后续请求。但是（如果指定的最大年龄过期了）代理缓存必须首先通过源服务器来重验证此响应，同时利用新请求里的 Authorization 请求头域去让源服务器给新请求授权。（这是 s-maxage 定义的行为）。如果响应包含“s-maxage=0”，那么代理在重利用此响应之前必须总是重验证它。
2. 如果此响应包含“must-revalidate”缓存控制指令，那么缓存可以利用此响应来响应后续请求。但是如果此响应是陈旧的，那么所有缓存必须首先通过源服务器重验证那个响应，同时利用新请求里的 Authorization 请求头域去让源服务器去给此新请求授权。
3. 如果此响应包含“public”缓存控制指令，那么此响应可以用来响应任何后续的请求。

14.9 Cache-Control

Cache-Control 常用头域被用于指定必须在请求/响应链上的被所有缓存机制遵守指令。这些指令指定了防止缓存去干涉请求或响应的行为。这些指令经常覆盖缺省的缓存算法。缓存指令是单方向的，因为请求中指令的存在并不意味着同样的指令必须在响应中出现。

请注意 HTTP/1.0 缓存可能没有实现 Cache-Control，并且也没有实现 Pragma: no-cache（参见 14.32 节）。

缓存指令必须被代理或网关通过，不管这些指令对应用程序有多重要，因为这些指令可能对请求/响应链上的所有接收者都适用。不可能为一个特定缓存去指定一个缓存指令。

```
Cache-Control = "Cache-Control" ":" 1#cache-directive

cache-directive = cache-request-directive
| cache-response-directive

cache-request-directive =
"no-cache" ; Section 14.9.1
| "no-store" ; Section 14.9.2
| "max-age" "=" delta-seconds ; Section 14.9.3, 14.9.4
| "max-stale" [ "=" delta-seconds ] ; Section 14.9.3
| "min-fresh" "=" delta-seconds ; Section 14.9.3
| "no-transform" ; Section 14.9.5
| "only-if-cached" ; Section 14.9.4
| cache-extension ; Section 14.9.6

cache-response-directive =
"public" ; Section 14.9.1
| "private" [ "=" <"> 1#field-name <"> ] ; Section 14.9.1
| "no-cache" [ "=" <"> 1#field-name <"> ]; Section 14.9.1
| "no-store" ; Section 14.9.2
| "no-transform" ; Section 14.9.5
| "must-revalidate" ; Section 14.9.4
| "proxy-revalidate" ; Section 14.9.4
| "max-age" "=" delta-seconds ; Section 14.9.3
| "s-maxage" "=" delta-seconds ; Section 14.9.3
| cache-extension ; Section 14.9.6

cache-extension = token [ "=" ( token | quoted-string ) ]
```

当指令不伴有 1#field-name 参数出现时，该指令适用于整个请求或响应。当一个指令伴有一个 1#field-name 参数时，此指令仅应用于被命名的头域，而不能应用于请求或响应的其他部分。

这一机制支持可扩展性；HTTP 协议将来的版本的实现可以通过将指令应用于 HTTP/1.1 中未定义的头域。

缓存控制指令可分为如下几类：

- 对什么是可缓存的限制；这可能只由源服务器指定。
- 对什么能被缓存保存的限制；这可由源服务器或用户代理指定。
- 对基本过期机制的改进；这可能由源服务器或用户代理指定。
- 对缓存重验证及重载的控制；这可能仅由用户代理指定。
- 对实体传输的控制
- 缓存系统的扩展。

14.9.1 什么是可缓存的

缺省情况下，若请求方法、请求头域和响应状态码指明响应为可缓存的，则此响应就是可以缓存的。13.4 节总结了可缓存性的这些缺省情况。下列缓存控制响应指令（Cache-Control response deirctives）允许源服务器覆盖缺省的响应可缓存性：

public

指明响应可被任何缓存保存，即便该响应通常是不可缓存的或只在非共享缓存里是可缓存的。（参见 14.8 节，关于 Authorization 头域的更多详述。）

private

指明响应消息的部分或所有部分是为一个用户准备的并且不得被共享缓存保存。可以使源服务器可以声明响应的特定部分来针对某一用户并且对其他用户的请求是无效的。一个私有（非共享）缓存可以缓存此响应。

注：词语“私有”的使用仅用来控制响应在何处可被缓存，并且不能保证消息内容的隐私。

no-cache

如果 no-cache 缓存控制指令没有指定一个 field-name，那么一个缓存不能利用此响应在没有通过源服务器对它进行成功重验证的情况下去满足后续的请求。这允许源服务器去防止响应被缓存保存，即使此缓存已被设置可以返回陈旧响应给客户端。

如果 no-cache 缓存控制指令指定一个或多个 field-name，那么缓存可以利用此响应去满足后续的请求，但这要受限于对缓存的其它限制。然而，指定的 filed-name 必须不能在后续请求的响应里被发送如果此响应没有在源服务器那里得到成功重验证。这允许源服务器能防止缓存去重利用响应里的某些头域，但仍然允许缓存能保存响应剩余部分。

注意：大多数 HTTP/1.0 缓存将不能识别或遵循这个指令。

14.9.2 什么能被缓存保存

no-store

no-store 缓存控制指令的目的在于防止不经意地释放或保留敏感信息（比如存放在备份磁带的信息）。 no-store 缓存控制指令应用于整个消息，并且可以在响应里或在请求里被发送。如果在请求里被发送，缓存不能保存此请求或此请求响应的任何部分。如果在响应里被发送，缓存不能保存此响应或保存此响应请求的任何部分。此缓存控制指令能应用于非共享缓存和共享缓存。“不能保存”在这个背景里的意思是指缓存不能有意地把信息保存在非易失性存储里，而且必须尽力去删除易失性存储上的信息当它转发完毕后。

即使当此指令在一个响应里时，用户也可能会显式地在缓存系统之外保存这个响应（例如利用一个“另存为”对话框）的地方。历史缓冲（History buffers）（见 13.13 节）可能保存这个响应作为它们正常操作的一个部分。

此指令的目的是为了满足某些用户声明的要求，还有就是为那些比较在意通过访问缓存数据结构而发生信息泄漏的作者提供方便。在一些情况下，利用此缓存控制指令可能会增强隐私，但是我们注意到它并不是在任何情况下都是可信任的或都能充分地保护隐私的。特别是，恶意的或被损坏的缓存可能不能识别到或遵循此指令，并且网络通信也容易随时受到窃听。

14.9.3 对基本过期机制的改进

实体的过期时间可由源服务器利用“Expires”头域（参见 14.21 节）指定。 或者，也可以在响应里利用 max-age 缓存控制指令指定。当 max-age 缓存控制指令出现在一个已缓存的响应里时，如果此响应的当前年龄（current age，译注：见 13.2.3 节关于 current age 的定义）大于为那个资源的一个新请求时 max-age 里给定的年龄值（以秒），那么这个已缓存的响应是陈旧的（stale）。对在一个响应里应用 max-age 缓存控制指令意味着此响应是可缓存的（也就是说“公有的”）除非出现其它更具限制性的缓存控制指令于响应里。

若响应同时含有 Expires 头域和 max-age 缓存控制指令，那么 max-age 缓存控制指令应该覆盖 Expires 头域，即使 Expires 头域更具限制性。此规则允许源服务器可以为一给定响应来为一 HTTP/1.1（或更迟）缓存提供一个比一 HTTP/1.0 缓存更长的过期时间（expiration time）。这个规则可能会很有用，如果某个 HTTP/1.0 缓存不恰当地计算了年龄（ages）或过期时间（expiration times），可能由于不同步的时钟。

许多 HTTP/1.0 缓存实现可能会把响应里小于或等于该响应里 Date 头域值的 Expires 头域值看成与“no-cache”缓存控制响应指令等效。如果一个 HTTP/1.1 缓存接收到这样一个响应，并且此响应没有包含一个 Cache-Control 头域，缓存应把此响应看成一个不可缓存的，这是为了保持和 HTTP/1.0 服务器兼容。

注意：一个源服务器可能希望把一个相对较新的 HTTP 缓存控制特性，例如“private”缓存控制指令，用于一个存有不能理解此特性的旧的缓存的网络上。源服务器应该需要把新特性和响应里值小于或等于 Date 值的 Expires 头域联合起来，这样以便防止旧缓存不恰当地保存此响应。

s-maxage
: 如果一响应包含一 s-maxage 缓存控制指令，那么对于一共享缓存（不能对私有缓存）来说，s-maxage 指定的值将会覆盖 max-age 缓存控制指令或 Expires 头域。**s-maxage 缓存控制指令照样意指 proxy-revalidate 缓存控制指令（见 14.9.4 节）的语义**，也就是说，共享缓存在没有通过源服务器首先重验证这个已陈旧的缓存项时不能利用它来响应后续的请求。s-maxage 缓存控制指令总是被私有缓存忽略。

注意：大多数不遵循此规范的老的缓存没有实现任何缓存控制指令。 一个源服务器如果希望利用一缓存控制指令去限制（但不能阻止）遵循 HTTP/1.1 的缓存去进行缓存处理，那么它可能会采用 max-age 控制指令去覆盖 Expires 头域，并且它会承认 HTTP/1.1 以前版本的缓存不会去观察 max-age 缓存控制指令。

其它缓存控制指令允许一个用户代理（user agent）去改变基本的过期机制。这些指令可能会被指定在请求里：

max-age
: 表明客户端愿接受这样一个响应，此响应的年龄不大于客户端请求里 max-age 指定时间

（以秒为单位）。除非 max-stale 缓存控制指令也包含在请求里，否则客户端是不能接收一个陈旧响应的。

min-fresh
表明客户端愿接受一个这样的响应，其保鲜寿命不小于响应当前年龄 (current age，见 13.2.3 节关于 current_age 的定义)与客户端请求里的 min-fresh 指定的时间之和（以秒为单位）。也就是说，客户端想要一个响应至少在 min-fresh 指定的时间内是保鲜的。

max-stale
表明客户端愿接受已经过期的响应。 若客户端请求为 max-age 指定了一个值，则表明客户端愿意接受过期时间不超过在 max-stale 里指定秒数的响应。若 max-stale 没有赋值，则客户端愿接受任意年龄的陈旧响应。

由于 max-stale 缓存控制指令出现在请求里，或由于此缓存被设置成能覆盖响应的过期时间，导致缓存返回了一个陈旧响应，那么缓存必须把一个 Warning 头域放进这个陈旧响应里，此 Warning 头域里应该是 110 警告码（响应是陈旧的）。

一个缓存可以被设置为可以不需要验证就可以返回陈旧的响应，但这不应与任何关于缓存验证（例如，一个“must-revalidate”缓存控制指令）的“必须”等级的要求相冲突。

若新请求与缓存项都包含一个“max-age”缓存控制指令，那么取两个值的小者来为此请求决定缓存项的保鲜程度。

14.9.4 缓存重验证和加载控制（Cache Revalidation and Reload Controls）

有时，用户代理可能希望或出于需要，坚持要求某缓存通过源服务器去重验证其缓存项，或者要求其缓存从源服务器那里重新加载其缓存项。End-to-end 重验证也许是有必要的，如果缓存或源服务器已过高估计已缓存的响应（cached response）的过期时间。End-to-end 重新加载可能是必要的，如果缓存项由于某原因已经变得陈旧了。

End-to-end 重验证可能被请求，当客户端没有本地缓存副本，此时我们称之为“未指定的 end-to-end 重验证（unspecified end-to-end revalidation）”，或者，当客户端存有本地缓存副本，此时我们称之为“指定的 end-to-end 重验证（specific end-to-end revalidation）”。

利用缓存控制请求指令，客户端能指定下面三种动作：

End-to-end reload （End-to-end 重新加载）
请求包含“no-cache”缓存控制指令，或，为了兼容 HTTP/1.0 客户端，“Pragma: no-cache”缓存控制指令。头域名不能被包含在请求的 no-cache 缓存控制指令里。服务器不能利用一个缓存副本来响应这样一个请求。

Specific end-to-end revalidation（指定的 end-to-end 重验证）
请求包含一个“max-age=0”缓存控制指令，它强制每个途径源服务器的缓存必须通过下一缓存或服务器来重验证它所拥有的缓存项（如果有的话）。此初始请求包含一带有客户端当前验证器的缓存验证条件。

Unspecified end-to-end revalidation（未指定的 end-to-end 重验证）
请求包含一个“max-age=0”缓存控制指令，它强制每个途径源服务器的缓存必须通过下一缓存或服务器来重验证它所拥有的缓存项（如果有的话）。此初始请求没有包含一缓存验证条件；沿着路径上的第一个拥有此资源缓存项的缓存（如果有的话）在请求里包含一带有

其缓存当前验证器的缓存验证条件。

max-age

当一个中间缓存被一个 max-age=0 的缓存控制指令强迫去重验证它所拥有的缓存项，并且客户端已经在请求里包含了其本身拥有的验证器，此验证器可能不同于当前缓存项里保存的验证器。在这种情况下，缓存应该在不影响语义透明性的情况下利用任一验证器去执行请求。

然而，验证器的选择可能会影响性能。最好的办法对中间缓存来说，就是当执行请求时利用它自己的验证器。如果服务器以 304（没有改变）回复，那么此缓存能返回一个它自己现在已经验证了的副本给客户端同时以一 200（OK）状态码。如果服务器以一新实体和一新缓存验证器来回复请求，那么此中间缓存会把返回的验证器同客户端请求里的验证器作比较并利用强比较方法。如果客户端的验证器和源服务器的相等，那么此中间缓存器只是简单的返回 304（没有改变）响应。否则，它返回一个新的实体并且状态码是 200 的响应。

如果一个请求包含一个 no-cache 缓存控制指令，那么它不应包含 min-fresh，max-stale，或 max-age 缓存控制指令。

only-if-cache

在一些情况下，例如糟糕的网络连接，一客户端可能希望一缓存只返回缓存当前保存的响应，并且不需要缓存通过源服务器对其缓存项进行重新加载或重验证。如果这样作，客户端可能会包含一个 only-if-cached 缓存控制指令于请求里。如果缓存接收了这样的指令，那么缓存应利用缓存项（但必须满足请求的其它方面的限制）去响应，或以 504（网关超时）状态码响应。然而，如果一组缓存作为一统一的具有良好内部连接性的系统来操作，那么这个请求可能会转发到缓存组的内部。

must-revalidate

由于一个缓存可能被设置去忽略服务器指定的过期时间，并且又由于一个客户端请求可能包含一个 max-stale 缓存控制指令（具有相似的作用），所以此协议同样包含一个让源服务器强迫缓存项被重验证的机制。当 must-revalidate 缓存控制指令出现在已被缓存接收的响应里时，那么此缓存不能利用此缓存项，如果在它变得陈旧并且没有通过源服务器对它进行重验证的情况下，去响应一个后续的请求。（也就是说，缓存必须每次进行 end-to-end 重验证，如果单独地基于源服务器的 Expire 或 max-age 值，已缓存的响应是陈旧的话）

must-revalidate 缓存控制指令可以为某些协议特性提供可信赖性操作。在所有情况下，一个 HTTP/1.1 缓存必须遵循 must-revalidate 缓存控制指令；特别地，如果此缓存不能直接和源服务器通信，那么它必须产生一个 504（网关超时）响应。

服务器应发送 must-revalidate 缓存控制指令，如果并且只有在服务器对实体的验证请求失败而导致不正确的操作时，例如一个不动声息的未执行的金融事务。接收端不能采取任何违反此缓存控制指令的自动的行为，并且不能自动地提供一个被验证无效的实体副本如果此副本通过源服务器重验证失败。

尽管这不被推荐，用户代理（user agent）如果在糟糕的网络连接限制下，可能会违反此缓存控制指令，但是，如果这样的话，它必须显式地警告用户这是一个未验证的响应。警告必须在每次未验证的访问时被提供，而且用户代理应需要用户显式地确认信息。

proxy-revalidate

proxy-revalidate 缓存控制指令和 must-revalidate 缓存控制指令具有相同的语义，但它不能应用于非共享（non-shared）的用户代理缓存。它能被用于一已被授权请求的响应，去允许用户缓存能保存或者过后能返回此响应而不需要去重验证它（因为它已经被那个用户授权了一次），但是服务于多个用户的代理仍然需要每次去重验证它（为了保证每个用户已被

授权）。注意这样的授权响应同样需要 public 缓存控制指令，这是为了允许响应能完全被缓存。

14.9.5 No-Transform 缓存控制指令

no-ransform

中间缓存（代理）的实现者们已发现转换某个实体主体的媒体类型转是很有用的。一个非透明代理可能，例如，会在不同图像格式之间进行转换，这是为了节约空间或在低速的连接上减少通信流量。

然而，当这些转换应用于某些应用的实体主体时，会引发严重操作问题。比如，医学图象应用程序，科学数据分析应用程序和利用 end-to-end 认证的应用程序都必须保证接收到的实体主体与原实体主体每一 bit 都是一致的。

所以，如果消息包括了 no-ransform 缓存控制指令， 那么中间缓存或代理不能改变 13.5.2 节中列出的受限于 no-transform 缓存控制指令的头域。这意味着缓存或代理不能改变由这些头域指定的实体主体的任何方面（aspect），包括实体主体本身的值。

14.9.6 缓存控制扩展（Cache control Extendions）

Cache-Control 头域能被扩展，可通过一个或多个 cache-extension 标记，每个标记可以被赋于一个值。信息扩展（这些扩展无须缓存行为的改变）可以不经改变其它缓存控制指令的语义而被添加。行为扩展是通过现有缓存控制指令的基本行为的修饰来实现的。 新缓存控制指令与标准缓存控制指令两者都被提供，这样，不理解新缓存控制指令的应用程序会缺省地采用标准缓存控制指令规定的行为，而那些能理解新指令的应用程序会将其看做是修改了标准缓存控制指令的要求。这样，缓存控制指令的扩展可以在无须改变基本协议的情况下就能够实现。

扩展机制依赖于一 HTTP 缓存，此缓存遵从所有本地 HTTP 版本缓存定义的缓存控制指令，遵从一定的扩展，并且会忽略所有它不能理解的缓存控制指令。

例如，考虑一个假设的名为“community”的新缓存响应控制指令，此指令被看成是对 private 缓存控制指令的修饰。我们定义此新缓存控制指令以表明：除共享缓存，任何被 communit 值命名的社区成员共享的缓存也可缓存此响应。例如，如果一个源服务器希望允许 UCI 社区里的成员在他们共享缓存里可以使用一私有响应，那么此源服务器应该包含：

Cache-Control: private, commuity="UCI"

一个见到此头域的缓存将会正确执行，即使此缓存不能理解这个 community 缓存扩展，因为缓存同样能看到并且能理解 private 缓存控制指令所以这样能导致缺省的安全行为。

不能识别的缓存控制指令必须被忽略；我们认为不能被 HTTP/1.1 缓存识别的任一缓存控制指令会和标准缓存控制指令（或者响应的缺省缓存能力）联合在一起这样缓存行为会保持最小正确性即使缓存不理解此扩展。

14.10 Connection

Connection 常用头域允许发送者指定某些专属于某特定连接的选项，并且 Connection 头域不能被代理（proxy）在以后的连接中传送。

Connection 头域遵循如下语法：

Connection = “Connection” “:” 1#(connection-token)
connection-token = token

HTTP/1.1 代理必须在转发消息之前解析 Connection 头域并且，为此头域中每一个 connection-token，从消息中删除任何与 connection-token 里同名头域。 连接选项是由 Connection 头域中出现 connection-token 而指明的，而不是由任何相应的附加头域，因为附加头域可以不被发送如果对应的那个连接没有参数。

Connection 头域里列出的消息头域不得包含 end-to-end 头域，例如 Cache-Control 头域。

HTTP/1.1 定义了“close”连接选项，这是为了让发送者指明在完成响应后连接将被关闭。

例如
Connection：close

无论是出现在请求或响应的头域里都表明：在完成现有请求/响应后连接不应被视是“持续的（persistent）”（参见 8.1 节）。

不支持持久连接的 HTTP/1.1 应用程序必须在每一消息中都加上“close”连接选项。

接收到含有 Connection 头域的 HTTP/1.0（或更低版本）消息的系统必须要为每一个 connection-token 去删除或忽略消息中与之同名的头域。这避免以前版本的 HTTP/1.1 代理错误转发这些头域。

14.11 Content-Encoding

“Content-Encoding”实体头域是对媒体类型的修饰。当此头域出现时，其值表明对实体主体采用了何种内容编码，从而可以知道采用何种解码机制以获取 Content-Type 头域中指出的媒体类型。Content-Encoding 头域主要目的是可以在不丢失下层媒体类型的标识下对文档进行压缩。

Content-Encoding = "Content - Encoding" ":" 1#content-coding

内容编码在 3.5 节里定义。下面是一个应用的例子：

Content-Encoding：gzip

内容编码（content-coding）是请求 URI 指定实体的特性。通常，实体主体以内容编码（content-coding）的方式存储，然而只有在此实体主体被呈现给用户之前才能被解码。然而，非透明代理可能会把实体主体的内容编码（content-coding）改成接收端能理解的内容编码（content-coding），除非“no-transform”缓存控制指令出现在消息里。

如果实体的内容编码不是“identity”，那么此响应必须包含一个 Content-Encoding 实体头域（见 14.11 节）并且列出非 dentity 的内容编码。

若实体的内容编码（content-coding）是一不被源服务器接受的请求消息，则响应必须以 415 状态码响应（不支持的媒体类型）。

若实体采用多种编码，则内容编码必须在 Content-Encdoing 头域里列出，而且还必须按他们

被编码的顺序列出。额外的关于编码参数的信息可以在其它实体头域里提供，这在此规范里没有定义。

14.12 Content-Language

Content-Language 实体头域描述了实体面向用户的自然语言。请注意，这不一定等同于实体主体中用到的所有语言。

Content-Language = “Content-Language” “:” 1#language-tag

语言标签由 3.10 节定义。Content-Language 头域的主要目的在于让用户根据自己喜爱的语言来识别和区分实体。这样，如果实体主体的内容是面向丹麦语言的用户，那么下面的头域是适合的：

Content-Language: da

若未指明 Content-Language 头域，那么此内容缺省是对所有语言的用户都支持。这既可能意味着发送者认为实体主体的内容与任意自然语言无关，也可能是发送者不知道内容该针对哪种语言。

在 Content-Language 头域里可以为内容（content）列出多种语言。例如，同时用毛里土语和英语呈现“Treaty of Waitangi”就可以用下面表示：

Content-Language: mi,en

然而，有多种语言呈现于实体中并不代表此实体一定是为多个国家语言的用户准备的。比如《初学拉丁文》之类的语言启蒙教程，显然是针对英语用户的。这里，合适的 Content-Language 头域里应只包括“en”。

Content-Language 可应用于任意媒体类型（media type） -- 它不限于文本式的文档。

14.13 Content-Length

Content-Length 实体头域用于指明发送给接收者实体主体（entity-bidy）的大小（以十进制的字节数表示），或是在使用 HEAD 方法时，指明实体主体本应在 GET 方法时发送实体主体的大小。

Content-Length = “Content-Length” “:” 1*DIGIT

示例：

Content-Length: 3495

除非被 4.4 节里规定的规则禁止，否则应用程序应该利用此头域指明消息主体（ message-body）的传输长度。

任何大于或等于 0 的 Content-Length 均为有效值。如果一个 Content-Length 没有在消息里给定，4.4 节描述了如何判断消息主体的长度。

请注意 Content-Length 头域的含义与 MIME 中的关于此头域的定义有很大的区别，MIME 中，它在 content-type 类型为“message/external-body”的消息里是可选的。在 HTTP 中，在消息被传输之前，如果消息的长度能被确定，那么消息里应该包含 Content-Length 头域，除非不被

4.4 节里的规则允许。

14.14 Content-Location

Content-Location 实体头域可用来为消息里的实体提供对应资源的位置，当此实体的访问位置独立于请求 URI 时。一服务器应该为响应实体的变量（variant，译注：见 1.3 节 术语）提供一个 Content-Location 头域；尤其是在资源有多个对应的实体时，并且这些实体拥有各自不同的位置，并且可以通过这些位置单独地访问到各个实体，这时服务器应该为一个特定的变量（variant）提供一个 Content-Location 头域。

Content – Location = “Content-Location” “:” (absoluteURI | relativeURI)

Content-Location 的值同样为实体定义了基 URI（base URI）。

Content-Location 的值并不能作为源请求 URI（original requested URI）的替代物；它只能是陈述了请求时相应于特定实体的资源位置。将来的请求也许会用 Content-Location 里的 URI 作为请求 URI，如果请求期望指定那个特定实体源。

如果一个实体含有一个 Content-Location 头域，并且此头域里的 URI 不同于获得此实体的 URI，那么缓存不能认为此实体能被用于去响应基于那个 Content-Location 里 URI 的后续请求。然而，Content-Location 能被用于区分同一请求资源的多个实体，这在 13.6 节里描述了。

若 Content-Location 拥有的是相对 URI（relative URI），则此相对 URI（relative URI）是相对于请求 URI 来解析的（Request-URI）。

PUT 或 POST 请求中含有 Content-Location 头域是没有定义的；服务器可自由忽略它。

14.15 Content-MD5

Content-MD5 实体头域，正如 RFC1864[23]里定义的一样，提供实体主体（entity-body）的 MD5 摘要（digest），为的是提供 end-to-end 消息完整性检测（MIC）。（注：一 MIC 有利于检测实体主体传输过程中的偶然性变动，但不一定能防范恶意攻击。）

Content-MD5 = "Content-MD5" ":" md5-digest
MD5-digest=< 由 RFC 1864 定义的 base64 的 128 位 MD5 摘要>

Content-MD5 头域可由源服务器或客户端产生，用作实体主体的完整性检验。只有源服务器或客户端可生成 Content-MD5 头域；不得由代理和网关生成，否则会有悖于其作为端到端完整性检验的价值。任何实体主体的接收者，包括代理和网关，都可以检查此头域里的摘要值与接收到的实体主体的摘要值是否相符。

MD5 摘要的计算基于实体主体的内容，包括任何已应用的内容编码（content-coding），但不包括应用于消息主体的任何传输编码。若接收到的消息具有传输编码，那么传输编码必须在用 Content-MD5 值与接收到的实体相检测之前被解除。

这样做的结果是：摘要的计算是基于实体主体（entity-body）的字节的，就像一条命令：如果没有传输编码被应用，它们将会被发送。

HTTP 将 RFC 1864 拓宽到允许对 MIME 复合媒体类型（如 multipart/*,message/rfc822）计算摘要，但这并不改变如前所述的摘要计算方法。

有一些关于这个的后果。复合媒体类型（composite types）的实体主体可能包含许多 body-

part，每一个 body-part 都有它自己的 MIME 和 HTTP 头域（包括 Content-MD5，Content-Transfer-Encoding，和 Content-Encoding 头域），如果一个 body-part 有一个 Content-Transfer-Encoding 或 Content-Encoding 头域，那么应该认为此 body-part 的内容已被应用此编码，并且认为此 body-part 被包含于 Content-MD5 摘要里，就是说在应用编码之后 。Transfer-Encoding 头域不被允许出现在 body-part 里。

不可在计算或核对摘要之前就将任何其它换行转换为 CRLF：实际传输的文本中使用的换行必须原封不动的参与摘要计算。

注：虽然 HTTP 的 Content-MD5 的定义和 RFC 1864 中关于 MIME 实体主体的完全一样， 但 HTTP 实体主体在对 Content-MD5 的应用上仍然有几处与 MIME 实体主体有所区别。首先，HTTP 不象 MIME 会用 Content-Transfer-Encoding 头域，而是会使用 Transfer-Encoding 和 Content-Encoding 头域。 其次，HTTP 比 MIME 更多地使用二进制内容类型，所以这种情况要注意用于计算摘要的字节顺序是由那个类型所定义的传输字节顺序（transmission byte order）。最后，HTTP 允许文本类传输时采用数种换行，而不只是规范的使用 CRLF 的的标准形式。

14.16 Content-Range

Content-Range 实体头域与部分实体主体一起发送，用于指明部分实体主体在完整实体主体里哪部分被采用。 范围的单位（Range unit）在 3.12 节中定义。

```
Content-Range = "Content-Range" ":" content-range-spec

content-range-spec = byte-content-range-spec
byte-content-range-spec = bytes-unit SP
                                byte-range-resp-spec "/"
                                ( instance-length | "*" )

byte-range-resp-spec = (first-byte-pos "-" last-byte-pos)
                                | "*"
instance-length = 1*DIGIT
```

除非无法或很难判断，此头域应指明完整实体主体的总长度。星号“ *”表示生成响应时的 instance-length 未知。

不像 byte-ranges-specifier 值（参见 14.35.1 节），byte-range-resp-spec 必须只能指明一个范围，并且必须包含首字节和尾字节的绝对位置。

一个带有 byte-range-resp-spec 的 byte-content-range-spec，如果它的 last-byte-pos 值小于 first-byte-pos 值，或它的 instance-length 值小于或等于它的 last-byte-pos 值，那么就说明是无效的。收到无效的 byte-content-range-spec 将被忽略，并且任何随其传输的内容都将被忽略

响应时发送状态码 416（请求的范围无法满足）的服务器应包含一个 Content-Range 头域，且里面的 byte-range-resp-spec 的值为“*”。instance-length 指定当前选定资源的长度。状态码为 206（部分内容）的响应不应该包含一个 byte-range-resp-sepc 为“*”的 Content-Range 头域。

假定实体共含 1234 字节，byte-content-range-spec 值的例子如下：

```
. The first 500 bytes:
   bytes 0-499/1234
```

```
. The second 500 bytes:
   bytes 500-999/1234

. All except for the first 500 bytes:
   bytes 500-1233/1234

. The last 500 bytes:
   bytes 734-1233/1234
```

当 HTTP 消息里包含单个范围时（比如，对单个范围请求的响应，或对一组无缝相连的范围请求的响应），那么此内容必须跟随一个 Content-Range 头域，并且还应该包含一个 Content-Length 头域来表明实际被传输字节的数量。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
```

当 HTTP 报文包含多个范围时（比如，对多个无重叠范围请求的响应），它们会被当作多部分类型的消息来传送。基于为此目的的多部分媒体类型为“ multipart/byteranges”，它在附录 19.2 里介绍了。见 19.6.3 里关于兼容性的问题描述。

对单个范围请求的响应不能使用 multipart/byteranges 媒体类型。若对多个范围请求的响应结果为一单个范围，那么可以以一个 multipart/byteranges 媒体类型发送并且此媒体类型里只有一个部分（part）。一个客户端如果无法对 multipart/byteranges 消息解码，那么它不能在一请求中请求多个字节范围。

当客户端在一请求中申请多个字节范围时，服务器应按他们在请求中出现顺序的范围返回他们所指定的范围。

若服务器出于句法无效的原因忽略了 byte-range-spec，它应把请求里无效的 Range 头域视为不存在。（正常情况下，这意味着返回一个包含完整实体的 200 响应。）

如果服务器接收到一请求，此请求包含一无法满足的 Range 请求头域（也即，所有 byte-range-spec 里的 first-byte-pos 值大于当前选择资源的长度），那么它将返回一个 416 响应（请求的范围无法满足）（参见 10.4.17 节）。

注： 客户端对无法满足 Range 请求头域不能指望服务器一定返回 416（请求的范围无法满足）响应而非 200（OK）的响应，因为不是所有服务器都能处理 Range 请求头域。

14.17 Content-Type

Content-Type 实体头域指明发给接收者的实体主体的媒体类型，或在 HEAD 方法中指明若请求为 GET 时将发送的媒体类型。

```
Content-Type = "Content-Type" ":" media-type
```

媒体类型有3.7节定义。 此头域的示例如下：

```
Content-Type: text/html; charset=ISO-8859-4
```

7.2.1 节提供了关于确定实体媒体类型方法的进一步论述。

14.18 Date

Date 常用头域表明产生消息的日期和时间，它和 RFC822 中的 orig-date 语义一样。此头域值是一个在 3.3.1 里描述的 HTTP-date；它必须用 RFC1123[8]里的 date 格式发送。

Date="Date""："HTTP-date

举个例子

Date:Tue,15 Nov 1994 08:12:31GMT

源服务器在所有的响应中必须包括一个日期头域，除了下面这些情况：

1. 如果响应的状态代码是 100（继续）或 101（转换协议），那么响应根据服务器的需要可以包含一个 Date 头域。

2. 如果响应状态代码表达了服务器的错误，如 500（内部服务器错误）或 503（难以获得的服务），那么源服务器就不适合或不能去产生一个有效的日期。

3. 如果服务器没有时钟，不能提供合理的当前时间的近似值，这个响应没必要包括 Date 头域，但在这种情况下必须遵照 14.18.1 节中的规则。

一个收到的消息如果没有 Date 头域的话就会被接收者加上一个，如果这条消息将要被接收者缓存或者将要通过一需要日期的网关。一个没有时钟的 HTTP 实现不能在没有重验证响应时去缓存（保存）此响应。一个 HTTP 缓存，特别是一个共享缓存，应该使用一种机制，例如 NTP[28]，让它的时钟与外界可靠的时钟保持同步。

客户端在包括实体主体（entity-body）的消息中应该包含一个 Date 头域，例如在 PUT 和 POST 请求里，即时这样做是可选的。一个没有时钟的客户端不能在请求中发送 Date 头域。

一个 Date 头域中的 HTTP-date 不应该是一个消息产生时刻之后的日期和时间。它应该表示与消息产生时的日期和时间的最近似值，除非没有办法产生一个合理的精确日期和时间。理论上说，日期应该是在实体（entity）产生之前的那一刻，实际上，日期是在不影响其语义值的情况下消息产生期间的任意时刻。

14.18.1 没有时钟的源服务器运作

一些源服务器实现可能没有可得时钟。一个没有可得时钟的源服务器不能给一个响应指定 Expires 或 Last-Modified 头域值，除非通过一个具有可信赖时钟的系统或用户，把此值与此资源联系在一起。可以给 Expires 赋予一值，此值在服务器的配置时间之时或之前将被视为过去。（这允许响应提前过期而不需要为每个资源保存单独的 Expires 值）。

14.19 ETag

Etag 响应头域提供了请求对应变量（variant）的当前实体标签。与实体标签一起使用的头域在 14.24，14.26 和 14.44 节里描述。实体标签可用于比较来自同一资源的不同实体。（参见 13.3.3 节）

Etag = "Etag" ":" entity-tag

例：

ETag: "xyzzy"
ETag: W/"xyzzy"
ETag: ""

14.20 Expect

Expect 请求头域用于指明客户端需要的特定服务器行为。

Expect = "Expect" ":" 1#expectation

expectation = "100-continue" | expectation-extension
expectation-extension = token ["=" (token | quoted-string)
*expect-params]
expect-params = ";" token ["=" (token | quoted-string)]

一个服务器如果不能理解或遵循一个请求里 Expect 头域的任何 expectation 值，那么它必须以合适的错误状态码响应。如果服务器不能满足任何 expectation 值，服务器必须以 417（期望失败）状态码响应，或者如果服务器满足请求时遇到其它问题，服务器必须发送 4xx 状态码。

本头域为将来的扩展被定义成一个扩展的语法。若服务器接收到的请求含有它不支持的 expectation-extension，那么它必须以 417（期望失败）状态响应。

expectation 值的比较对于未引用标记（unquoted token）（包括“100-contine”标记）是而言是不区分大小写的，对引用字符串（quoted-string）的 expectation-extension 而言是区分大小写的。

Expect 机制是 hop-by-hop 的：即 HTTP/1.1 代理（proxy）必须返回 417（期望失败）响应如果它接收了一个它不能满足的 expectation。 然而，Expect 请求头域本身是 end-to-end 头域；它必须要随请求一起转发。

许多旧版的 HTTP/1.0 和 HTTP/1.1 应用程序并不理解 Expect 头域。

参见 8.2.3 节中 100（继续）状态的使用。

14.21 Expires

Expires 实体头域（entity-header）给出了在何时之后响应即被视为陈旧的。一个陈旧的缓存项不能被缓存（一个代理缓存或一个用户代理的缓存）返回给客户端，除非此缓存项被源服务器（或者被一个拥有实体的保鲜副本的中间缓存）验证。见 13.2 节关于过期模型的进一步的讨论。

Expires 头域的出现并不意味着资源在 Expire 指定时间时、之前或之后将会改变或不存在。

Expires 头域里日期格式是绝对日期（absolute date）和时间，由 3.3.1 节中 HTTP-date 定义；它必须是 RFC1123 里的日期格式：

Expires="Expires " ":" HTTP-date

使用示例为：

```
Expires: Thu, 01 Dec 1994 16:00:00 GMT
```

注：若响应包含一个 Cache-Control 头域，并且含有 max-age 缓存控制指令（参见 14.9.3 节），则此指令覆盖 Expires 头域。

HTTP/1.1 客户端和缓存必须把其它无效的日期格式，特别是包含“0”的日期格式看成是过去的时间（也就是说，“已经过期”）。

为了将响应标为“已经过期”，源服务器必须把 Expires 头域里的日期设为与 Date 头域值相等。（参见 13.2.4 节里关于过期计算的规则。）

为标记响应为“永不过期”，源服务器必须把 Expires 头域里的日期设为晚于响应发送时间一年左右。HTTP/1.1 服务器不应发送超过将来一年的过期日期。

除非另外被 Cache-Control 头域（见 14.9 节）指明，否则如果存在 Expires 头域且头域里的日期值为某响应（可能缺省是不可缓存的）将来时间，那么就表明此响应是可缓存的。

14.22 From

From 请求头域，如果有的话，应该包含用户代理当前操作用户的 email 地址。这个地址应该是机器可用的地址，这被 RFC 822 [9]里的“mailbox”定义的同时也在 RFC 1123 [8]里作了修订：

```
From   = "From" ":" mailbox
```

例如：

```
From: webmaster@w3.org
```

From 头域可以被用于记录日志和作为识别无效或不期望请求的来源。他不应该被用作访问保护的不可靠方法。这个头域的解释是：此请求是代表所指定人执行，此人应该承担这个方法执行的责任。特别的，机器人代理（robot agents）应该包含这个头域，这样此人应该对运行此机器人代理程序负责，并且应该能被联系上如果在接收端出现问题的话。

此头域里的网络 email 地址是可以和发出请求的网络主机（host）不同。例如，当一个请求被通过一代理（proxy）时，源作者的地址应被使用。

客户端在没有用户的允许时是不应发出 From 头域的，因为它可能和用户的个人隐私或他们站点的安全策略（security policy）相冲突。强烈建议在任何一请求之前，用户能取消，授权，和修改这个头域的值。

14.23 Host

Host 请求头域指明了请求资源的网络主机和端口号，这可以从用户或相关资源给定的源 URI 获得（通常是一个 HTTP URL，在 3.2.2 节描述）。Host 头域值必须代表源服务器或网关（由那个源 URL 指定）的命名权限（naming authority）。这允许源服务器或网关去区分有内在歧义的 URLS，例如，拥有一个 IP 地址对应有多个主机名服务器，它的根“/”URL。

```
Host = "Host" ":" host [ ":" port ] ; 3.2.2 节
```

一个“host”如果没有跟随的端口信息，那么就采用是请求服务的的默认端口（例如，对一个 HTTP URL来说，就是80端口）。例如，一个对源服务器“http://www.w3.org/pub/WWW/”的请求，可以用下面来表示：

```
GET /pub/WWW/HTTP/1.1
```

Host: www.w3.org

一个客户端必须在所有 HTTP/1.1 请求消息里包含一个 Host 头域。如果请求 URI 没有包含请求服务的网络主机名，那么 Host 头域必须给一个空值。一个 HTTP/1.1 代理必须确保任何它转发的请求消息里必须包含一个正确的 Host 头域，用于指定代理请求服务。所有基于网络的 HTTP/1.1 服务器必须响应 400（坏请求）状态码，如果请求消息里缺少 Host 头域。

见 5.2 和 19.6.1.1 节里有针对 Host 头域的其他要求。

14.24 If-Match

If-Match 请求头域是用来让方法成为条件方法。如果一个客户端已经从一个资源获得一个或多个实体（entity），那么它可以通过在 If-Match 头域里包含相应的实体标签（entity tag）来验证实体是否就是服务器当前实体。实体标签（entity tag）在 3.11 节里定义。这个特性使更新缓存信息只需要一个很小的事务开销。当更新请求时，它照样被用于防止对资源错误版本的不经意修改。作为一种特殊情况，“*”匹配资源的当前任何实体。

If-Match = "If-Match" ":" （ "*" | 1#entity-tag ）

如果 If-Match 头域里任何一个实体标签假设与一个相似 GET 请求（没有 If-Match 头域）返回响应里实体的实体标签相匹配，或者如果给出“*”并且请求资源的当前实体存在，那么服务器可以执行请求方法就好像 If-Match 头域不存在一样。

服务器必须用强比较方法（见 13.3.3）来比较 If-Match 里的实体标签（entity tag）。

如果没有一个实体标签匹配，或者给出了“*”但服务器上没有当前的实体，那么服务器不能执行此请求的方法，并且返回 412 响应（先决条件失败）。这种行为是很有用的，特别是在当客户端希望防止一更新方法（updating method）（例如 PUT 方法）去修改此客户端上次获取但现已改变的资源时，

如果请求假设在没有 If-Match 头域的情况下导致了除 2XX 或 412 以外的其他状态码响应，那么 If-Match 头域必须被接收端忽略。

“If-Match: *” 的含义是：此方法将被执行，如果源服务器（或缓存，很可能使用 Vary 机制，见 14.44 节）选择的表现形式（representation）存在的话，但是如果此表现形式不存在，那么此方法不能被执行。

如果一个请求想要更新一个资源（例如 PUT）那么它可以包含一个 If-Match 头域来指明：当相应于 If-Match 值（一个实体标签）的实体不再是那个资源的表现形式时，此请求方法不能被采用。这允许用户表明：如果那个资源已经改变了而他们不知道的话，他们不希望请求成功。
例如：

If-Match: "xyzzy"
If-Match: "xyzzy", "r2d2xxxx", "c3piozzzz"
If-Match: *

既有 If-Match 头域又有 If-None-Match 或 If-Modified-Since 头域的请求的结果在本规范没有定义。

14.25 If-Modified-Since

If-Modified-Since 请求头域被用来让方法成为条件方法：如果请求变量（variant）自从此头域里指定的时间之后没有改变，那么服务器不应该返回实体；而是应该以 304（没有改变）状态码进行响应，同时返回的消息不需要消息主体（message-body）。

If-Modified-Since = "If-Modified-Since" ":" HTTP-date

一个例子是：

If-Modified-Since: Sat, 29 Oct 1994 19:43:31 GMT

如果一个 GET 请求方法含有 If-Modified-Since 头域但无 Range 头域，那么此方法请求的实体只有请求里 If-Modified-Since 头域中指定日期之后被改变才能被服务器返回。决定这个算法包括下列情况：

a）如果请求假设会导致除状态 200 之外的任何其它状态码，或者如果 If-Modified-Since 日期是无效的，那么响应就和正常的 GET 请求的响应完全一样。比服务器当前时间晚的日期是无效的。

b）如果自从一个有效的 If-Modified-Since 日期以来，变量已经被修改了，那么服务器应该返回一个响应同正常 GET 请求一样。

c）如果自从一个有效的 If-Modified-Since 日期以来，变量没有被修改，那么服务器应该返回一个 304（没有改变）响应。

这种特征的目的是以一个最小的事务开销来更新缓存信息。

注意：Range 请求头域改变了 If-Modified-Since 的含义；详细信息见 14.35。

注意：If-Modified-Since 的时间是由服务器解析的，它的时钟可能和客户端的不同步。

注意：当处理一个 If-Modified-Since 头域的时候，一些服务器使用精确的日期比较方法，而不是小于（less-than）比较方法，来决定是否发送 304（没有改变）响应。当为缓存验证而发送一个 If-Modified-Since 头域的时候，为了得到最好的结果，客户端被建议尽可能地去利用以前 Last-Modified 头域里被接收的日期字符串。

注意：如果客户端，对同一请求，在 If-Modified-Since 头域中使用任意日期代替 Last-Modified 头域里得到的日期，那么客户端应该知道这个日期应该能被服务器理解。由于客户端和服务器之间时间编码的不同，客户端应该考虑时钟不同步和舍入的问题。如果在客户端第一次请求时与后来请求里头域 If-Modified-Since 指定的日期之间，文档发生改变，这就可能会出现竞争条件，还有，如果 If-Modified-Since 从客户端得到的日期没有得到服务器时钟的矫正，就有可能出现时钟偏差等问题的。客户端和服务器时间的偏差最有可能是由于网络的延迟造成的。

既有 If-Modified-Since 头域又有 If-Match 或 If-Unmodified-Since 头域的请求的结果在本规范没有定义。

14.26 If-None-Match

If-None-Match 头域被用于一个方法使之成为条件的。一个客户端如果拥有一个或多个从某资源获得的实体，那么它能验证在这些实体中有不存在于服务器当前实体中的实体，这通过在 If-

None-Match 头域里包含这些实体相关的实体标签（entity tag）来达到此目的。这个特性允许通过一个最小事务开销来更新缓存信息。它同样被用于防止一个更新方法（如，PUT）不经意的改变一个客户端认为不存在但事实却存在的资源。

作为特殊情况，头域值“*”匹配资源的任何当前实体。

If-None-Match = "If-None-Match" ":" （ "*" | 1#entity-tag ）

如果 If-None-Match 头域里的任何实体标签（entity tag）假设与一个相似的 GET 请求（假设没有 If-None-Match 头域）返回实体的实体标签相匹配，或者，如果“ *”被给出并且服务器关于那个资源的任何当前实体存在，那么服务器不能执行此请求方法，除非资源的修改日期和请求里 If-Modified-Since 头域（假设有的话）里提供的日期匹配失败（译注：匹配失败说明资源改变了）。换言之，如果请求方法是 GET 或 HEAD，那么服务器应以 304（没有改变）来响应，并且包含匹配实体的相关缓存头域（特别是 Etag） 。对于所有其它方法，服务器必须以 412（先决条件失败）状态码响应。

13.3 节说明了如何判断两实体标签是否匹配。弱比较方法只能用于 GET 或 HEAD 请求。

如果 If-None-Match 头域里没有实体标签匹配，那么服务器可以执行此请求方法就像 If-None-Match 头域不存在一样，但是必须忽略请求里的任何 If-Modified-Since 头域。也就是说，如果没有实体标签匹配 ，那么服务器不能返回 304（没有改变）响应。

如果假设在没有 If-None-Match 头域存在的情况下，请求会导致除 2xx 及 304 状态码之外响应，那么 If-None-Match 头域必须被忽略。（见 13.3.4 节关于假如同时存在 If-Modified-Since 和 If-None-Match 头域时服务器的行为的讨论）

“If-None-Match: *”的意思是：如果被源服务器（或被缓存，可能利用 Vary 机制，见 14.44 节）选择的表现形式（representation）存在的话，请求方法不能被执行，然而，如果表现形式不存在的话，请求方法是能被执行的。这个特性可以防止在多个 PUT 操作中的竞争。

例:
If-None-Match: "xyzzy"
If-None-Match: W/"xyzzy"
If-None-Match: "xyzzy", "r2d2xxxx", "c3piozzzz"
If-None-Match: W/"xyzzy", W/"r2d2xxxx", W/"c3piozzzz"
If-None-Match: *

如果一个请求含有 If-None-Match 头域，还含有一个 If-Match 或 If-Unmodified-Since 头域的话，此请求的指向结果在此规范里没有定义。

14.27 If-Range

如果客户端在其缓存中有一实体的部分副本，并希望其整个缓存项是及时更新的，那么客户端可以在一条件 GET（conditional GET）（利用了 If-Unmodified-Since 和 If-Match 头域两者或其中之一）请求里利用 Range 请求头域。然而，如果由于实体被改变而使条件失败，那么客户端可能会发出第二次请求从而去获得整个当前实体主体（entity-body）。

If-Range 头域允许客户端使第二次请求短路（short-circuit）。说的通俗一点，这意味着：如果实体没有改变，发送我想要的部分；如果实体改变了，那就把整个新实体发过来。

If-Range = “if-Range” “:”(entity-tag | HTTP-date)

若客户端没有一实体标签（entity tag），但有一个最后修改日期（Last-Modified date），它可以在 If-Range 头域里利用此日期。（服务器通过检查一两个字符即可区分合法 HTTP-date 与任意形式的 entity-tag。）If-Range 头域只应该与一 Range 头域一起使用，并且必须被忽略如果请求不包含一个 Range 头域或者如果服务器不支持子范围（sub-range）操作。

如果 If-Range 头域里给定的实体标签匹配服务器上当前实体的实体标签（entity tag），那么服务器应该提供此实体的指定范围，并利用 206（部分内容）响应。如果实体标签（entity tag）不匹配，那么服务器应该返回整个实体，并利用 200（ok）响应。

14.28 If-Unmodified-Since

If-Unmodified-Since 请求头域被用于一个方法使之成为条件方法。如果请求资源自从此头域指定时间开始之后没有改变，那么服务器应该执行此请求就像 If-Unmodified-Since 头域不存在一样。

如果请求变量（variant，译注：见术语）在此头域指定时间后以后已经改变，那么服务器不能执行此请求，并且必须返回 412（前提条件失败）状态码。

If-Unmodified-Since = "If-Unmodified-Since" ":" HTTP-日期

此域的应用实例:

If-Unmodified-Since: Sat, 29 Oct 1994 19:43:31 GMT

如果请求正常情况下（即假设在没有 If-Unmodified-Since 头域的情况下）导致任何非 2xx 或 412 状态码，那么 If-Unmodified-Since 头域将会忽略。

如果此头域中指定的日期无效，那么此头域会被忽略。

在请求里有 If-Unmodified-Since 头域并且有 If-None-Match 或者 If-Modified-Since 头域中的一个，这种请求的处理结果在此规范中没有被定义。

14.29 Last-Modified

Last-Modified 实体头域（entity-header）指定了变量（variant）被源服务器所确信的最后修改的日期和时间。

Last-Modified = “Last-Modified” “:” HTTP-date

应用示例如下：

Last-Modified : Tue, 15 Nov 1994 12:45:26 GMT

此头域的确切含义取决于源服务器的实现和源资源（original resource）的性质。 对文件而言，它可能仅仅指示文件上次修改的时间。对于包含动态部分的实体而言，它可能是组成其各个部分中最后修改时间最近的那个部分。对数据库网关而言，它可能是记录的最新修改时间戳。对虚拟对象来说，它可能是最后内部状态改变的时间。

源服务器不能发送一个迟于消息产生时间的 Last-Modified 日期。假如资源最后修改日期可能指示将来的某个时间，那么服务器应该用消息产生的时间替换那个日期。

源服务器获得实体 Last-Modified 值应尽量靠近服务器产生响应的 Date 值。这允许接收者对实

体修改时间作出准确的估计，特别是如果实体的改变时间接近响应产生的时间。

HTTP/1.1 服务器应该尽可能地发送 Last-Modified 头域。

14.30 Location

Location 响应头域被用于为了完成请求或识别一个新资源，使接收者能重定向于 Location 指明的 URI 而不是请求 URI。对于 201（Created）响应而言，Location 是请求建立新资源的位置。对于 3xx 响应而言，Location 应被指定服务器为自动重定向资源所喜爱的 URI。Location 头域值由一个绝对 URI 组成。

Location = “Location” “:” absoluteURI

一个例子如下：

Location : http://www.w3.org/pub/WWW/People.html

注: Content-Location 头域（14.14 节）不同于 Location 头域，Content-Location 头域指定了请求里封装实体的源位置。有可能一个响应即包含 location 也包含 Content-Location 头域。在 13.10 节里有关于一些方法的要求。

14.31 Max-Forwards

Max-Forwards 请求头域为 TRACE（9.8 节）和 OPTIONS（9.2 节）提供一种机制去限制转发请求的代理或网关的数量。当客户端尝试去跟踪一个好像陷入失败或陷入循环的请求链时，这是非常有帮助的。

Max-Forwards = “Max – Forwards” “:” 1*DIGIT

Max-Forwards 值是十进制的整数，它指定了请求消息可以剩余转发的次数。

对于一个 TRACE 或 OPTIONS 请求，如果包含一个 Max-Forwards 头域，那么接收此请求的代理或网关必须能在转发（forwarding）此请求之前检查和更新 Max-Forwards 头域值。如果接收的值为 0，那么接收者不能转发此请求；而是，它必须作为最后的接收者响应。如果接收的 Max-Forwards 值比 0 大，那么此转发的消息必须包含一个更新了的 Max-Forwards 头域，更新的值是在接收时的值上减去 1。

对本规范定义的所有其它方法以及任何没有明确作为方法定义部分的扩展方法的请求里，Max-Forwards 头域可能会被忽略。

14.32 Pragma

Pragma 常用头域被用于包含特定执行指令，这些指令可能被应用于请求/响应链中任何接收者。从协议的观点来看，pragma 指令指定的行为是可选的；然而，一些系统可能要求行为必须满足指令的要求。

Pragma	= “Pragma” “:” 1#pragma-directive
pragma-directive	=”no-cache” | extension-pragma
extension-pragma	=token [“=” (token | quoted-string)]

当 no-cache 指令出现在请求消息中，应用程序应该转发（forward）此请求到源服务器，即使它拥有此请求响应的缓存副本。pragma 指令和 no-cache 缓存控制指令（见 14.9）有相同的语义，并且它是为同 HTTP/1.0 向后兼容而被定义。当一个 no-cache 请求发送给一个不遵循

HTTP/1.1 的服务器时，客户端应该既包含 pragma 指令也包含 no-cache 缓存控制指令。

pragma 指令必须能穿过代理和网关应用程序，不管对于那些应用程序有没有意义。因为这些指令可能对请求/响应链上的所有接受者有用。不可能为一个特定的接收者定义一个 pragma；然而，任何对接收者不相关的 pragma 指令都应该被接收者忽略。

HTTP/1.1 缓存应该把“Pragma：no-cache”当作好像客户端发送了“cache-control:no-cache”。在 HTTP 中不会有新的 pragma 指令会被定义。

14.33 Proxy-Authenticate

Proxy-Authenticate 响应头域必须被包含在 407 响应（代理授权）里。此头域值由一个 challenge 和 parameters 组成，challenge 指明了授权模式，而 parameters 应用于请求 URI 的代理。

```
Proxy-Authenticate  = "Proxy-Authenticate" ":" 1#challenge
```

关于HTTP访问授权过程的描述在“HTTP Authentication：Basic and Digest Access Authentication”[43]中介绍了。不像WWW-Authenticate头域，Proxy-Authenticate头域只能应用于当前连接，并且不应该传递给下游（downstrem）客户端。然而，一个中间代理可能需要从请求下游客户端而获得它自己的证书（credentials），这在一些情况下就好像代理正在转发Proxy-Authenticate头域一样。

14.34 Proxy-Authorization

Proxy-Authorization 请求头域允许客户端让一代理能给客户端自己（或客户端的用户）授权 。Proxy-Authorization 头域值由包含用户代理（为代理和/或请求资源域）的授权信息的证书组成。

```
Proxy-Authorization = “Proxy-Authorization” “:” credentials
```

HTTP 访问授权过程在“HTTP Authentication: Basic and Digest Access Authentication”[43]中描述。不像 Authorization 头域，Proxy-Authorization 头域只能应用于下一个期望利用 Proxy-Authenticate 头域授权的外向（outbound）代理。。

14.35 Range

14.35.1 字节范围 （Byte Ranges）

既然所有的 HTTP 实体都以字节序列形式的 HTTP 消息表示，那么字节范围的概念对任何 HTTP 实体都是有意义的.（不过并不是所有的客户和服务器都需要支持字节范围操作.。）

HTTP 里的字节范围应用于实体主体的字节序列（不必和消息主体一样）。

字节范围操作可能会在一个实体里指定一个字节范围或多个字节范围。

```
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

byte-range-spec 里的 first-byte-pos 值给出了一个范围里第一个字节的偏移量.。last-byte-pos 值给出了这个范围里最后一个字节的偏移量；也就是说，确定的字节位置必须在实体的范围之内。字节偏移是以 0 为基准（译注：0 代表第一个字节，1 代表第二个字节）。

如果存在 last-byte-pos 值，那么它一定大于或等于那个 byte-range-spec 里的 first-byte-pos，否则 byte-range-spec 在句法上是非法的。接收者接收到包括一个或多个无效的 byte-range-spec 值的 byte-range-set 时，它必须忽略包含那个 byte-range-set 的头域.。

如果 last-byte-pos 值不存在，或者大于或等于实体主体的当前长度，则认为 last-byte-pos 等于当前实体主体长度减一。

通过选择 last-byte-pos，客户能够限制获得实体的字节数量而不需要知道实体的大小。

suffix-byte-range-spec = “-“ suffix-length
suffix-length = 1*DIGIT

suffix-byte-range-spec 用来表示实体主体的后缀，其长度由 suffix-length 值给出.。（也就是说，这种形式规定了实体正文的最后 N 个字节。）如果实体短于指定的 suffix-length，则使用整个实体主体。

如果一个句法正确的 byte-range-set 至少包括一个这样的 byte-range-spec，它的 first-byte-pos 比实体主体的当前长度要小，或至少包括一个 suffix-length 非零的 suffix-byte-range-spec，那么 byte-range-set 是可以满足的，否则是不可满足的。如果 byte-range-set 不能满足，那么服务器应该返回一个 416 响应（请求范围不能满足）。否则，服务器应该返回一个 206 响应（部分内容）

byte-ranges-specifier（字节-范围-说明符）值的例子（假定实体主体的长度为 10000）:

-- 第一个 500 字节（字节偏移量 0-499,包括 0 和 499）: bytes=0-499

-- 第二个 500 字节（字节偏移量 500-999,包括 500 和 999）: bytes=500-999

-- 最后 500 字节（字节偏移量 9500-9999,包括 9500 和 9999）: bytes=-500 或 bytes=9500-

-- 仅仅第一个和最后一个字节（字节 0 和 9999）: bytes=0-0,-1

-- 关于第二个 500 字节（字节偏移量 500-999,包括 500 和 999）的几种合法但不规范的叙述:
bytes=500-600,601-999
bytes=500-700,601-999

14.35.2 范围请求（Range Retrieval Requests）

使用条件或无条件 GET 方法可以请求实体的一个或多个字节范围，而不是整个实体，这利用 Range 请求头域，请求返回的结果就是 Range 头域指示的请求资源实体的范围。

Range = "Range" ":" ranges-specifier

服务器可以忽略 Range 头域。然而，.HTTP/1.1 源服务器和中间缓存应该尽可能支持字节范围，因为 Range 支持从部分失败传输中有效地恢复，并且支持从大的实体中有效地获取部分内容。

如果服务器支持 Range 头域，并且指定的范围或多个范围对实体来说是适合的：

1. 如果在无条件 GET 请求里出现 Range 头域，那么这将会改变没有 Range 头域时的 GET 请求返回的结果。换句话说，返回的状态码不是 200（ok）而是 206（部分响应）。

2. 如果在条件 GET（请求里利用了 If-Modified-Since 和 If-None-Match 中任意一个或两者，或者利用了 If-Unmodified-Since 和 If-Match 中的任意一个或两者）请求里出现 Range 头域，那么这将改变返回的结果，如果 GET 请求假设在没有 Range 头域时被服务器成功响应并且条件为真。但如果条件为假，它不会影响 304（没有改变）响应被返回。

某些情形下，除了使用 Range 头域外，可能还要同时使用 If-Range 头域（见 14.27 节）。

如果支持范围请求的代理接收了一个范围请求，并转发（forward）请求到内向（inbound）服务器，并且接收到了一个完整实体，那么它只应该返回给客户请求的范围。代理将接收的整个响应存储到它的缓存里如果此响应满足缓存分配策略。

14.36 Referer

Referer 请求头域允许客户端，为了让服务器受益，指定请求 URI 来源的资源 URI。（Referer 头域的 Referer 本应该写成 Referrer，出现了笔误）。Referer 请求头域允许服务器为了个人兴趣，记录日志，优化缓存等来产生回退链接列表。它照样允许服务器为维护而跟踪过时或写错的链接。Referer 头域不能被发送如果请求 URI 从一个没有自身 URI 的资源获得，例如用户从键盘输入。

Referer = "Referer" ":" （ absoluteURI | relativeURI ）

例如:

Referer: http://www.w3.org/hypertext/DataSources/Overview.html

如果 Referer 头域的域值是相对 URI，那么它将被解析为相对于请求 URI。URI 不能包含一个片段（fragment）。见 15.1.3 关于安全的考虑。

14.37 Retry-After

Retry-After 响应头域能被用于一个 503（服务不可得）响应，服务器用它来向请求端指明服务不可得的时长。此头域可能被用于 3xx（重定向）响应，服务器用它来（如 web 浏览器）指明用户代理再次提交已重定向请求之前的最小等待时间。Retry-After 头域值可能是 HTTP-date 或者也可能是一个响应时间后的十进制整数秒。

Retry-After = “Retry-After” “:” (HTTP-date | delta-seconds)

下面是它的两个例子

Retry-After: Fri,31 Dec 1999 23:59:59 GMT
Retry-After:120

在后一例子中，延迟时间是 2 分钟。

14.38 Server

Server 响应头域包含了源服务器用于处理请求的软件信息。 此域可包含多个产品标记（3.8 节），以及鉴别服务器与其他重要子产品的注释。产品标记按它们的重要性来排列，并鉴别应用程序。

Server = “Server”

例：

服务器：CERN/3.0 libwww/2.17

若响应是通过代理转发的，则代理程序不得修改 Server 响应头域。作为替代，它应该包含一个 Via 头域（在 14.45 节里描述）。

注：揭示特定的软件版本可能会使服务器易于受到那些针对已知安全漏洞的软件的攻击。 建议服务器实现者将此域作为可设置项。

14.39 TE

TE 请求头域指明客户端可以接受哪些传输编码（transfer-coding）的响应，和是否愿意接受块（chunked）传输编码响应的尾部（trailer）（译注：TE 头域和 Accept-Encoding 头域与 Content-Encoding 头域很相似，但 TE 应用于传输编码（transfer coding），而 Content-Encoding 应用于内容编码（content coding，见 3.5 节））。TE 请求头域的值可能由包含关键字“trailers” 和/或用逗号分隔的扩展传输编码名（扩展传输编码名可能会携带可选的接受参数的列表）（在 3.6 节描述）组成。

```
TE       = "TE" ":" #（ t-codings ）
t-codings = "trailers" |（ transfer-extension [ accept-params ] ）
```

如果出现关键字“trailers”，那么它指明客户端愿意接受（chunked）传输编码响应的尾部（trailer）。此关键字为传输编码（transfer-coding）值而保留，但它本身不代表一种传输编码。

举例：

TE: deflate
TE:
TE: trailers, deflate;q=0.5

TE 请求头域仅适用于立即连接。所以无论何时，只要在 HTTP/1.1 消息中存在 TE 头域，连接头域（Connection header filed）（参见 14.10 节）中就必须指明。

通过 TE 头域，服务器能利用下述规则来测试传输编码（transfer-coding）是否是可被客户端接受的：

1. 块（chunked）传输编码总是可以接受的。如果在 TE 头域里出现关键字“trailers”，那么客户端指明它愿意代表自己或任意下游（downstream）客户端去接受块（chunked）传输编码响应里的尾部（trailer）。这意味着，如果“trailers”给定，客户端正在声明所有下游（downstream）客户端愿意接收块（chunked）传输编码响应里的尾部（trailer），或声明它愿意代表下游接收端去尝试缓存响应。

   注意：HTTP/1.1 并没有定义任何方法去限制块传输编码响应的大小，这是为了方便客户端能缓存整个响应。

2．只要是出现在 TE 头域里的传输编码都是可被请求端接受的，除非此传输编码跟随的 qvalue 值为 0（根据 3.9 节中定义，qvalue 为 0 表明是“不可接受的”（not acceptable））

3．如果在 TE 头域里有指明多个传输编码是可接受的，那么传输编码（transfer-coding）的 qvalue 值最大的是最容易被被接受的。块传输编码的 qvalue 值为 1。

如果 TE 头域值是空的或者 TE 头域没有出现在消息里，那么服务器只能认为块（chunked）传输编码的响应是请求端可以接受的。没有传输编码的消息总是可接受的。

14.40 Trailer

Trailer 常用头域值指明了以块（chunked）传输编码的消息里尾部（trailer）用到的头域。

Trailer = "Trailer" ":" 1#field-name

一个 HTTP/1.1 消息会包含一个 Trailer 头域，如果消息利用了块（chunked）传输编码并且编码里的尾部（trailer）不为空。这样做是为了使接收端知道块（chunked）传输编码响应消息尾部（trailer）有哪些头域。

如果具有块传输编码的消息，但没有 Trailer 头域存在，则此消息的尾部（trailer）将不能包括任何头域。3.6.1 节展示了块传输编码的尾部（trailer）的利用限制。

Trailer 头域中指示的消息头域不能包括下面的头域：
.Transfer-Encoding
.Content-Length
.Trailer

14.41 Transfer-Encoding

传输译码（Transfer-Encoding）常用头域指示了消息主体（message body）的编码转换，这是为了实现在接收端和发送端之间的安全数据传输。它不同于内容编码（content-coding），传输代码是消息的属性，而不是实体（entity）的属性。

Transfer-Encoding = "Transfer-Encoding" ":" 1#transfer-coding

传输编码（transfer-coding）在 3.6 节中被定义了。一个例子是：

Transfer-Encoding: chunked

如果一个实体应用了多种传输编码，传输编码（transfer-coding）必须以应用的顺序列出。传输编码（transfer-coding）可能会提供编码参数（译注：看传输编码的定义,3.6 节），这些编码参数额外的信息可能会被其它实体头域（entity-header）提供，但这并没有在规范里定义。

许多老的HTTP/1.1应用程序不能理解传输译码（Transfer-Encoding）头域。

14.42 Upgrade

Upgrade 常用头域允许客户端指定它所支持的附加通信协议，并且可能会使用如果服务器觉得可以进行协议切换。服务器必须利用 Upgrade 头域于一个 101（切换协议）响应里，用来指明哪个协议被切换了。

Upgrade = “Upgrade” “:” 1#product

例如，

Upgrade: HTTP/2.0，SHTTP/1.3, IRC/6.9, RTA/x11

Upgrade 头域的目的是为了提供一个从 HTTP/1.1 到其它不兼容协议的简单迁移机制。这通过允许客户端告诉服务器客户端期望利用另一种协议，例如主版本号更高的最新 HTTP 协议，即使当前请求仍然使用 HTTP/1.1。这能降低不兼容协议之间迁移的难度，只需要客户端以一个更普遍被支持协议发起一个请求，同时告诉服务器客户端想利用“更好的”协议如果可以的话（“更好的”由服务器决定，可能根据方法和/或请求资源的性质决定）。

Upgrade 头域只能应用于应用程序层（application –layer）协议之间的切换，应用程序层协议在传输层（transport-layer）连接之上。Upgrade 头域并不意味着协议一定要改变；并且服务器接受和使用是可选的。在协议改变后应用程序层（apllication-layer）的通信能力和性质，完全依赖于新协议的选择，尽管在改变协议后的第一个动作必须是对初始 HTTP 请求（包含 Upgrade 头域）的响应。

Upgrade 头域只能应用于立即连接（immediate connection）。因此，upgrade 关键字必须被提供在 Connection 头域里（见 14.10 节），只要 Upgrade 头域呈现在 HTTP/1.1 消息里。

Upgrade 头域不能被用来指定切换到一个不同连接的协议。为这个目的，使用 301，302，303 重定向响应更合适。

这个规范定义了本协议的名字为“HTTP”，它在 3.1 节的 HTTP 版本规则中定义的超文本传输协议家族中被使用。任何一个标记都可被用来做协议名字，然而，只有当客户端和服务器认为这个名字对应同一协议才有用。

14.43 User-Agent

User-Agent 请求头域包含关于发起请求的用户代理的信息。这是为了统计，跟踪协议违反的情况，和为了识别用户代理从而为特定用户代理自动定制响应。用户代理应该包含 User-Agent 头域在请求中。此头域包含多个识别代理和子产品的产品标记（见 3.8 节）和解释。通常，为了识别应用程序，产品标记按重要性排列。

User-Agent = "User-Agent" ":" 1*（ product | comment ）

例子:

User-Agent: CERN-LineMode/2.15 libwww/2.17b3

14.44 Vary

Vary 响应头域值指定了一些请求头域，全部去决定某缓存是否被允许去利用此响应（并且此响应仍然保鲜）去回复后续请求而不需要重验证（revalidation）。对于不可缓存或已陈旧的响应，Vary 头域值用于告诉用户代理（user agent）选择表现形式（reprentation）的标准。一个 Vary

头域值是“*”意味着缓存不能根据后续请求的请求头域来决定此响应是合适的表现形式。见13.6节关于缓存如何利用Vary头域。

```
Vary  = "Vary" ":" （  "*" | 1#field-name ）
```

一个HTTP/1.1的服务器应该包含一个Vary头域于任何可缓存的受限于服务器驱动协商的响应里。这样做是允许缓存恰当去理解关于那个资源的将来请求，并通知用户代理关于那个资源的协商出现。一个服务器可能包含一个Vary头域于一个不可缓存的受限于服务器驱动协商的响应里，因为这样做可能为用户代理提供有用的关于响应变化的维度的信息。

一个Vary头域值由域名（filed-name）组成，响应的表现形式是基于Vary头域里列举的请求头域来选择的。一个缓存可能会假设为将来请求进行相同的选择，如果Vary头域例举了相同的域名，但必须是此响应在此期间是保鲜的。

Vary头域里的域名并不是局限于本规范里定义的标准请求头域。域名是大小写不敏感的。

Vary域值为“*”指明不受限于请求头域的非特指参数（例如，客户端的网络地址）作用于响应表现形式中进行选择　。“*”值不能被代理产生；它可能只能被源服务器产生。

14.45 Via

Via常用头域必须被网关（gateways）和代理（proxies）使用，用来指明在用户代理和服务器之间关于请求的中间协议和接收者，和在源服务器和客户端之间关于响应的中间协议和接收者。它和RFC822[9]里的“Received”头域相似，并且它用于跟踪消息的转发，避免请求循环，和指定沿着请求/响应链的所有发送者的协议能力。

```
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
```

received-protocol指出沿着请求/响应链每一段的服务器或客户端所接收消息的协议版本　。protocol-version被追加于Via头域值后面，当消息被转发时。

只要协议是HTTP，那么protocol-name是可有可无的。received-by头域通常是接收的转发服务器的host（主机）和可选的port（端口）号，或接收的转发客户端的host（主机）和可选的port（端口）号。然而，如果真实host（主机）被看作是信息敏感的，那么此主机可能会被别名代替。如果port（端口号）没有被给定，那么它可能被假设为received-protocol的缺省port（端口）号。

Via头域里如果有多个域值，则每个值分别代表一个已经转发消息的代理或网关。每一个接收者必须把它的信息追加到最后，所以最后的结果是按照转发应用程序的顺序来的。

comment（注释）可能被用于Via头域是为了指定接收者代理或网关的软件，这个好比User-Agent和Server头域。然而，Via头域里所有的comment是可选的（译注：可有可无的），并且可以被接收者在转发消息之前移去。

例如，有一个请求消息来自于一个HTTP/1.0用户代理，被发送到代号为“fred”的内部代理，此内部代理利用HTTP/1.1协议转发此请求给一个站点为nowhere.com的公共代理，而此公共代理为了完成此请求通过把它转发到站点为“www.ics.uci,edu”的源服务器。被

“www.ics.uci.edu”站点接收后的请求这时可能有下面的 Via 头域：

Via: 1.0 fred, 1.1 nowhere.com (Apache/1.1)

被用作通向网络防火墙的入口的代理和网关在缺省情况下不应该转发 host（主机）的名字和端口到防火墙区域里。如果这些信息显示地指定要被传送，那么就应该被传送。如果此信息显示地指定不能被传送，那么任何穿过防火墙而被接收的 host（主机）应该用一个合适的别名替换。

为了隐藏组织结构的内部结构需要，一个代理（proxy）可能会在一个 Via 头域中把相同 received-protocal 值的项合成一个项。例如，

Via: 1.0 ricky, 1.1 ethel, 1.1 fred, 1.0 lucy

将被折叠成

Via: 1.0 ricky, 1.1 mertz, 1.0 lucy

应用程序不应该合并多个项，除非他们都在相同组织的控制下并且 host（主机）已经被别名代替了。应用程序不能合并不同 received-protocaol 值的项。

14.46 Warning

Warning 常用头域被用于携带额外关于消息状态或消息转换的信息，而这些信息是不能在消息里反应出来的。这些信息通常被用于去警告由于缓存操作或消息主体转换带来的透明性（semantic transparency）的缺失。

Warning 头域被用于响应里，这里有如下语法：

```
Warning    = "Warning" ":" 1#warning-value

warning-value = warn-code SP warn-agent SP warn-text
                                                [SP warn-date]

warn-code  = 3DIGIT
warn-agent =  （  host [ ":" port ] ）  | pseudonym
                    ; the name or pseudonym of the server adding
                    ; the Warning header, for use in debugging
warn-text  = quoted-string
warn-date  = <"> HTTP-date <">
```

一个响应可能携带多个 Warning 头域。

warn-text 必须使用对于接收响应的用户来说尽可能理解的自然语言和字符集。找到用户自能理解的自然语言和字符集，必须基于任何可能的知识，如缓存或用户的位置，请求里的 Accept-Language 头域，响应里的 Content-Language 头域，等等。缺省语言是英语，缺省字符集是 ISO-8859-1。

如果字符集不是 ISO-8859-1，那么它必须利用 RFC2047 里描述的来在 warn-text 里进行编码。

Warning 头域能被应用于任何消息，然而，一些 warn-codes 是特定于缓存的，并且只能被应用于响应消息。新的 Warning 头域应添加到任何已存 Warning 头域的后面。缓存不能删除任何它接收到的消息里的 Warning 头域。然而，如果一缓存成功验证一缓存项，那么它应移除任何以前依附于那个缓存项的 Warning 头域除了特定警告码（Warning codes）的 Warning 头域。然后，它必须添加这个验证响应里的任何 Warning 头域。换句话说，Warning 头域是依附于最近

相关响应的 Warning 头域。

当多个 Warning 头域被附加于一个响应里，那么用户代理应该通知用户尽可能多的警告，并且以它们呈现在响应里的顺序。如果用户代理不能通知用户所有的警告，那么用户代理应该按照下面的规则：

- 前面的响应里的警告优于后面响应的警告
- 用户偏爱的字符集的警告优于其它字符集的警告，但这除了 warn-codes 和 warn-agents 一致的情况。

产生多个 Warning 头域的系统应该时刻记住利用用户代理行为来安排警告。

关于警告的缓存行为的要求在 13.1.2 里描述。

下面是当前定义的 warn-codes，每一个 warn-code 都有一个建议性的以英语表示的 warn-text，和它的意思的描述。

110 Response is stale
无论何时当返回响应是陈旧的时候，必须被包含。

111 Revalidation failed
如果一个缓存因为尝试去重验证响应失败而返回一个陈旧的响应（由于不能到达服务器），必须被包含。

112 Disconnected operation
如果缓存在一段时间被有意地断开连接，应该被包含。

113 Heuristic expiration
如果缓存探索性地选择了一个保鲜寿命大于 24 小时并且响应的年龄大于 24 小时时，必须被包含。

199 Miscellaneous warning
警告文本可能包含任意信息呈现给用户。除了呈现给用户警告，接收警告的系统不能采取任何自动行为。

214 Transformation applied
如果一个中间缓存或代理采用任何对响应的内容编码（content-coding）（在 Content-Encoding 头域里指定）或媒体类型（media-type）（在 Content-Type 头域里指定）的改变变，或响应的实体主体（entity-body）的改变，那么此响应码必须被中间缓存或代理添加，除非此警告码（warning code）已经在响应里出现。

299 Miscellaneous persistent warning
警告文本应该包含任意呈现给用户的任意信息。接收警告的系统不能采取任何自动行为。

如果一个实现在一个消息里发送多个版本是 HTTP/1.0 或更早的 HTTP 协议版本的 Warning 头域，那么发送者必须包含一个和响应日期（date）相等的 warn-date 到每一个 Warning 头域值中。

如果一个实现收到一条 warning-value 里包含一个 warn-date 的消息，并且那个 warn-date 不同于响应里的 Date 值，那么 warning-value 必须在保存，转发，或利用消息之前从消息里删除。（这回防止本地缓存去缓存 Warning 头域的恶果。）如果所有 warning-value 因为这个原因而被删除，Warning 头域必须也要被删除。

14.47 WWW-Authenticate

WWW-Authenticate 响应头域必须包含在 401（没有被授权）响应消息中。此域值至少应该包含一个 callenge，此 callenge 指明授权模式（schemes）和适用于请求 URI 的参数。

WWW-Authenticate =“WWW-Authenticate” “:” 1#challenge

HTTP 访问授权过程在“HTTP Authentication: Basic and Digest Access Authentication”[43]里描述。用户代理被建议特别小心去解析 WWW-Authenticate 头域值，当此头域值包含多个 challenge，或如果多个 WWW-Authenticate 头域被提供且 challenge 的内容能包含逗号分隔的授权参数的时候。

15．安全考虑 （Security Consideration）

这一部分是用来提醒程序开发人员，信息提供者，和用户关于 HTTP/1.1 安全方面的限制。讨论并不包含对被披露问题的明确的解决办法，然而，它却对减少安全风险提供了一些建议。

15.1 个人信息 （Personal Information）

HTTP 的客户端经常要对大量的个人信息保密（例如用户的名字，域，邮件地址，口令，密匙等。），并且应当非常小心地防止这些信息无意识地通过 HTTP 协议泄露到其他的资源。我们非常强烈地建议应该给用户提供一个方便的界面来控制这种信息的传播，并且设计者和实现者在这方面应该特别注意。历史告诉我们在这方面的错误经常引起严重的安全和/或者隐私问题，并导致对设计或实现者的公司产生非常不利的影响。

15.1.1 服务器日志信息的滥用 （Abuse of Server Log Information）

服务器是用来保存用来指定用户读模型或感兴趣主题的请求的。这些信息通常显然是需保密的，并且它的使用在某些国家被法律保护。利用 HTTP 协议提供数据的人们必须保证在这些数据被许可的情况下分发。

15.1.2 敏感信息的传输 （Transfer of Sensitive Inforamtion）

就像任何数据传输协议一样，HTTP 不能调整传输数据的内容，也没有任何经验方法在任意给定请求的背景里去判断特定信息的敏感性。因此，应用程序应该尽可能为此信息提供者提供对此信息的控制。在此背景里，有四个头域需要提出来，这四个头域是：Server，Via，Referer 和 From。

揭露服务器特定软件版本信息可能会使服务器的机器更容易受到通过软件安全漏洞来进行攻击的攻击。实现者应该使 Server 头域成为可设置的选项。

用作穿过网络防火墙入口的代理应该特别小心防火墙后可以辨别主机信息的头域信息被传输。特别是，代理应移除在防火墙之后产生的任何 Via 头域。

Referer 头域允许被学习的读模式和反向链接的跟踪。虽然它非常有用，但也会被滥用如果用户细节没有从包含在 Referer 头域里信息里分离开来。即使当个人信息已经被移除了，Referer 头域也可能指定私有文档的 URI（不能被公开）。

From 头域里的信息可能会和用户的私有兴趣或他们站点的安全策略相冲突，因此 From 头域不应被传输在用户没有能使此头域的内容失效、有效和更改的情况下。用户必须能在用户的喜爱

或应用程序的缺省设置范围内设置此头域的内容。

我们建议，尽管不需要，给用户提供一个方便的开关界面来使发送 From 和 Referer 头域有效或失效。

User-Agent（14.4 节）或 Server（14.38 节）头域有时候能被用来去判断一个特定的客户端或服务器是否存在安全漏洞。不幸的是，同样的信息经常被用于其它有价值的目的，因为 HTTP 当前没有更好的机制。

15.1.3 URI 中敏感信息的编码（Encoding Sensitive Information in URI's）

因为一个链接的源可能是私有信息或者可能揭露其它私有信息源，所以强烈建议用户能选择是否需要发送 Referer 头域。例如，浏览器客户端可能为了开放/匿名方式提供一个触发开关，此开关可能使 Referer 头域和 From 头域信息的发送有效/无效。

如果参考页面在一个安全的协议上传输，客户端不应该包含一个 Referer 头域在一个非安全 HTTP 请求里。

利用 HTTP 协议的服务作者不应该利用基于窗体 GET 提交敏感数据，因为这个能引起数据在请求 URI 里被编码。许多已存在的服务，代理，和用户代理将在对第三方可见的地方记录请求 URI。服务器能利用基于窗体 POST 提交来取代基于窗体 GET 提交。

15.1.4 连接到 Accept 头域的隐私问题

Accept 请求头域能揭露用户的信息给所有被访问的服务器。Accept-Language 头域能揭露用户的私有信息，因为能理解特定语言的人经常被认为就是某个特定种族里的成员。提供选项在每次请求里去设置 Accept-Language 头域的用户代理被强烈鼓励让设置过程应包含一个让用户知道自己隐私可能被会泄漏的消息。

一种限制隐私丢失的方法可能是缺省为用户代理去遗漏 Accept-Language 头域的发送，并且询问用户是否给开始给服务器发送 Accept-Language 头域，如果用户代理通过查看任何由服务器产生的 Vary 响应头域时发现这个发送能提高服务的质量。

每一个请求里的用户可配置的接受头域（accept header fields），特别是如果这些接受头域（accept header fileds）包含质量值，那么应该被服务器用作相对信赖和长久的用户标识符（user identifiers）。这样的用户标识符将会允许内容提供者进行点击跟踪以及允许合作内容提供者匹配跨服务器点击跟踪或者形成单个用户窗体提交。注意对于许多并不在代理后面的用户，运行用户代理的主机的网络地址也将作为长久用户的标识符。在代理被用作增强隐私的环境里，用户代理应保守地提供接受（accept）头域配置选项给终端用户上。作为一种高度隐私的方式，用户代理可能在接力的（relayed）的请求里的过滤接受头域。提供高度可配置性的用户代理应警告用户会有隐私的泄漏。

15.2 基于文件和路径名称的攻击

HTTP 的源服务器的实现应该限制 HTTP 请求返回的文档应是服务器管理员有意图的文档。如果 HTTP 服务器要把 HTTP URIs 翻译成文件系统的调用，那么服务器必须小心去对待提供给 HTTP 客户端的文件传输。例如，UNIX，微软 Windows，和其他操作系统都利用“..”去指示当前的父目录。对于这样一个系统，一 HTTP 服务器不允许任何这样的构造（construct）在请求 URI 里，如果这个构造能在通过 HTTP 服务器访问之外能访问这个资源。同样的，用作对服务

器内部引用的文件（如访问控制文件，配置文件，脚本代码）必须受到保护而不让其被不合适的获取，因为他们可能包含敏感的信息。经验告诉我们一个在 HTTP 服务器实现里的一个小小的错误会带来安全风险。

15.3 DNS 欺骗

使用 HTTP 的客户端严重依赖于域名服务，因此这会导致基于 IP 和 DNS 名称的不关联的攻击。客户端需要小心关注 IP 地址/DNS 名称关联的持久合法性。

特别是，HTTP 客户端为了确认 IP 地址/DNS 名称关联性，应该依赖于客户端自己的的名称解析器，而不应依赖缓存以前主机（host）名称查找（host name lookups）结果。许多平台在恰当的时候可在本地缓存主机名称查找（host name lookups），并且他们应被配置为可这样做。然而，只有当被名称服务器报告的 TTL（Time To Live）信息指明被缓存的信息仍然有用时，才可以缓存查找（lookups）。

如果 HTTP 客户端为了提高性能去缓存主机名称查找（host name lookups）的结果，那么他必须观察被 DNS 报告的 TTL 信息。

如果 HTTP 客户端不能看到这条规则，那么，当以前访问的 IP 地址改变时，他们就会被欺骗。因为网络地址的改变变得很平常，所以这种形式的攻击在不断增加。看到这个规则能减少潜在的安全攻击的可能性。

此要求同样能改进客户端负载均衡（load-balancing）行为，因为重复的服务器能利用同一个 DNS 名称，此要求能降低用户在访问利用策略（strategy）的站点中的体验失败。

15.4 Location 头域和欺骗

如果单个的服务器支持互不信任的多个组织，那么它必须检查响应（在据称的组织的控制下产生）里 Location 和 Content-Location 头域值，以确信这些头域不会使它们没有权限的资源无效。

15.5 Content-Disposition 的问题

RFC 1806 [35]，在 HTTP 中经常使用的 Content-Disposition（见 19.5.1 节）头域就源于此文档，有许多非常认真的安全考虑在此文档里说明。Content-Disposition 并不是 HTTP 标准版本中的一部分，但自从它被广泛应用以来，我们正在证明它对使用者的价值和风险。详细资料见 RFC 2183 [49]（对 RFC 1806 的升级）。

15.6 授权证书和空闲客户端

现有的 HTTP 客户端和用户代理通常会不明确地保留授权信息。HTTP/1.1 并没有为服务器提供一个方法让服务器去指导客户端丢弃这些缓存的证书（credentials）。这是一个重大缺陷，此缺陷需要扩展 HTTP 协议来解决。在某些情形下，缓存证书会干涉应用程序的安全模型，这些情形包含但不局限于：

- 这样的客户端。此客户端空闲已到达一定时间，服务器可能希望再次使客户端让用户出示证书。
- 这样的应用程序。此应用程序包括了一个会话中断指令（例如在一页上有“退出”或者“提交”的按钮），根据此指令，服务器端“知道”不需要更多理为客户端保留证书。

这是作为当前独立研究的。有很多解决这个问题的社区，并且我们鼓励在屏幕保护程序、空闲超

时、和其他能减轻安全问题的方法里利用密码保护。特别是，能缓存证书的用户代理被鼓励去提供一个容易的访问控制机制，让在用户控制下丢弃缓存的证书。

15.7 代理和缓存 （Proxies and Caching）

本质上说，HTTP 代理是中间人（men-in-the-middle），并且存在中间人攻击（man-in-the-middle attacks）危险。系统（代理运行于其上）的缺陷能导致严重的安全和隐私问题。代理拥有对相关安全信息、用户和组织的个人信息、和属于用户和内容提供者的专有信息的访问权限。一个有缺陷的代理，或一个没有考虑安全性和隐私性的代理可能会被委托用来攻击。

代理操作者应该保护代理运行其上的系统，正如他们保护任何包含或传输敏感信息的系统一样。特别是，代理上收集的日志信息经常包含较高的个人敏感信息，和/或关于组织的信息。日志信息应该被小心的保护，并且要合适地开发利用。（见 15.1.1）节。

代理的设计者应当考虑到设计和编码判定所涉及到的隐私和安全问题，以及他们提供给代理操作人员配置选项（尤其是缺省配置）所牵涉到的隐私和安全问题。

代理的用户需要知道他们自比运行代理的操作员更不值得信赖；HTTP 协议自身不能解决这个问题。

在合适的时候，对密码学的正确应用，可能会保护广泛的安全和隐私攻击。密码学的讨论不在本协议文档的范围内。

15.7.1 关于代理的服务攻击的拒绝

代理是存在的。代理很难被保护。关于此研究正在进行。

16 感谢（Acknowledgment）

这份规范大量使用了扩展 BNF 和 David 为 RFC 822 [9]定义的常用结构。同样的，它继续使用了很多 Nathaniel Borenestein 和 Ned Freed 为 MIME [7]提供的定义。我们希望在此规范里他们的结论有助于减少过去在 HTTP 和互联网邮件消息格式关系上的混淆。

HTTP 协议在这几年已经有了相当的发展。它受益于大量积极的开发人员的社区--许多人已经通过 www-talk 邮件列表参与进来--并且通常就是那个社区对 HTTP 和万维网的成功作了重大贡献。 Marc Andreessen, Robert Cailliau, Daniel W. Connolly, Bob Denny, John Franks, Jean-Francois Groff , Phillip M. Hallam-Baker, Hakon W. Lie, Ari Luotonen, Rob McCool, Lou Montulli, Dave Raggett, Tony Sanders, 和 Marc VanHeyningen 因为他们在定义协议早期方面的贡献应该得到特别的赞誉。

这篇文档从所有那些参加 HTTP-WG 的人员的注释中获得了很大的益处。除了已经提到的那些人以外，下列人士对这个规范做出了贡献：

Gary Adams
Harald Tveit Alvestrand
Keith Ball
Brian Behlendorf
Paul Burchard
Maurizio Codogno
Mike Cowlishaw
Roman Czyborra
Michael A. Dolan
David J. Fiander
Ross Patterson
Albert Lunde
John C. Mallery
Jean-Philippe Martin-Flatin
Mitra
David Morris
Gavin Nicol
Bill Perry
Jeffrey Perry
Scott Powers

Alan Freier Owen Rees
Marc Hedlund Luigi Rizzo
Greg Herlihy David Robinson
Koen Holtman Marc Salomon
Alex Hopmann Rich Salz
Bob Jernigan Allan M. Schiffman
Shel Kaphan Jim Seidman
Rohit Khare Chuck Shotton
John Klensin Eric W. Sink
Martijn Koster Simon E. Spero
Alexei Kosut Richard N. Taylor
David M. Kristol Robert S. Thau
Daniel LaLiberte Bill （BearHeart） Weinman
Ben Laurie Francois Yergeau
Paul J. Leach Mary Ellen Zurko
Daniel DuBois Josh Cohen

缓存设计的许多内容和介绍应归于以下人士的建议和注释：Shel Kaphan, Paul Leach, Koen Holtman, David Morris, 和 Larry Masinter。

大部分规范的范围是基于 Ari Luotonen 和 John Franks 早期做的工作，以及从 Steve Zilles 另外加入的内容。

感谢 Palo Alto 的"cave men"。你们知道你们是谁。

Jim Gettys（这篇文档现在的编者）特别希望感谢 Roy Fielding，这篇文档以前的编者，连同 John Klensin, Jeff Mogul, Paul Leach, Dave Kristol, Koen Holtman, John Franks, Josh Cohen, Alex Hopmann, Scott Lawrence, 和 Larry Masinter 一起感谢他们的帮助。还要特别感谢 Jeff Mogul 和 Scott Lawrence 对“MUST/MAY/ SHOULD”使用的检查。

Apache 组，Anselm Baird-Smith，Jigsaw 的作者，和 Henrik Frystyk 在早期实现了 RFC 2068，我们希望感谢他们发现了许多这篇文档正尝试纠正的问题。

19 附录

19.1 互联网媒体类型 message/http 和 application/http

这篇文档除了定义 HTTP/1.1 协议外，同时还被作为互联网媒体类型“message/http”和“application/http”的规范。此类型用于封装一个 HTTP 请求消息或响应消息，这假设此类型遵循 MIME 对 所有“消息”类型关于行长度和编码的限制。application/http 类型可以用来封装一个或者更多 HTTP 请求或响应信息（非混合的）的管线（pipeline）。下列是在 IANA[17]注册的。

媒体类型名称： message
媒体子类型名称： http
必须参数： 无
可选参数： 版本，信息类型

版本：封装信息的 HTTP 版本号（例如，"1.1"）。如果不存在，版本可以从消息的第一行确定。

信息类型：信息类型--"请求"或者"响应"。如果不存在，类型可以从报文的第一行确定。

编码考虑：只有"7bit"，"8bit"，或者"二进制"是允许的。
安全考虑：无

媒体类型名称：　application
媒体子类型名称：　http
必须参数：　无
可选参数：　版本，信息类型
版本：封装信息的 HTTP 版本号（例如，"1.1"）。如果不存在，版本可以从报文的第一行确定。
信息类型：信息类型--"request"或者"response"。如果不存在，类型可以从报文的第一行确定。
编码考虑：用这种类型封装的 HTTP 信息是"二进制"的格式；当通过 E-mail 传递的时候一种合适的内容传输编码是必须的。
安全考虑：无

19.2 互联网媒体类型 multipart/byteranges

当一个 HTTP 206（部分内容）响应信息包含多个范围的内容（请求响应的内容有多个非重叠的范围），这些是作为一个多部分消息主体来被传送的。这种用途的媒体类型被称作"multipart/byteranges"。

multipart/byteranges 媒体类型包括两个或者更多的部分，每一个都有自己 Content-type 和 Content-Range 头域。必需的分界参数（boundary parameter）指定分界字符串，此分界字符串用来隔离每一部分。

媒体类型名称：　multipart
媒体子类型名称：　byteranges
必须参数：　boundary
可选参数：　无
编码考虑：只有"7bit"，"8bit"，或者"二进制"是允许的。
安全考虑：无
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-type:multipart/byteranges;boundary=THIS_STRING_SEPARATES

--THIS_STRING_SEPARATES
Content-type: application/pdf
Content-range: bytes 500-999/8000

...第一部分...
--THIS_STRING_SEPARATES
Content-type: application/pdf
Content-range: bytes 7000-7999/8000

...第二部分
--THIS_STRING_SEPARATES--
```

注意：
1）在实体（entity）中，在第一个分界字符串之前可以有多余的 CRLFs。

2） 虽然 RFC 2046 [40]允许分界字符串加引号，但是一些现有的实现会不正确的处理分界字符串
3） 许多浏览器和服务器是按照字节范围标准的早期草案关于使用 multipart/x-byteranges 媒体类型来进行编码的的，这个草案不总是完全和 HTTP/1.1 中描述的版本兼容。

19.3 放松的应用程序 （Tolerent Applications）

虽然这篇文档列出了 HTTP/1.1 消息所必须的元素，但是并不是所有的应用程序都能正确地实现。因此我们建议运行程序可以容忍偏差只要这些偏差能被无歧义的理解。

客户端应该放松地解析 Status-Line（状态行）；服务器也应该放松地解析 Request-Line（请求行）。特别的，他们应该可以接受头域之间任何数量的 SP 或 HT 字符，即使协议规定只有一个 SP。

消息头域的行终结符是 CRLF。然而，当解析这样的头域时，我们建议应用程序能识别单一 LF 作为行终结符并能忽略 CR。

实体主体（entity-body）的字符集应该被标记为应用于实体主体字符编码的最小公分母，并且期望不对实体进行标记要优于对实体标记为 US-ASCII 或 ISO-8859-1。见 3.7.1 和 3.4.1。

对关于日期分析和编码的要求的额外规则以及其它对日期编码的潜在问题包含：

- HTTP/1.1 客户端和缓存应该假定一个似乎是 50 多年以后的 RFC-850 日期实际上是过去的（这有助于解决"2000 年"问题）。

- 一个 HTTP/1.1 的实现可以内部地表示一个比正确日期值更早的已解析后的 Expires 日期，但是一定不要（MUST NOT）内部地表示一个比正确日期值更迟的已解析过的 Expires 日期。

- 所有过期日期相关的计算必须用 GMT 时间。本地时区一定不能（MUST NOT）影响年龄或过期时间的计算。

- 如果一个 HTTP 头域不正确的携带了一个非 GMT 时间区的日期值，那么必须利用最保守的可能转换把此日期值转换成 GMT 时间值。

19.4 HTTP 实体和 RFC 2045 实体之间的区别

HTTP/1.1 使用了许多 Internet Mail （RFC 822 [9]）和 Multipurpose Internet Mail Extensions（MIME [7]）里定义的结构，去允许实体以多种表现形式和扩展机制去传输。然而，RFC2045 讨论邮件，并且 HTTP 有许多不同于 RFC2045 里描述的特征。这些不同被小心地挑选出来优化二进制连接的性能，为了允许使用新的媒体类型有更大的灵活性，为了使时间比较变得容易，和为了承认一些早期 HTTP 服务器和客户端的实效。

本附录描述了 HTTP 协议不同于 RFC 2045 的特殊区域。在严格的 MIME 环境中的代理和网关应该意识到这些不同并且在必要的时候要提供合适地转换。从 MIME 环境到 HTTP 的代理和网关也需要意识到这些不同因为一些转换可能是需要的。

19.4.1 MIME 版本（MIME-Version）

HTTP 不是一个遵守 MIME 的协议。然而 HTTP/1.1 消息可以包含一个单独的 MIME-Version 常用头域用来指出什么样的 MIME 协议版本被用于去构造消息。利用 MIME-Version 头域指明完全

遵循 MIME 协议的消息（在 RFC2045[7]）。代理/网关要保证完全遵守 MIME 协议当把 HTTP 消息输出到严格 MIME 环境的时候。

MIME-Version = "MIME-Version" ":" 1*DIGIT "." 1*DIGIT

在 HTTP/1.1 用的缺省值是 MIME1.0 版本。然而，HTTP/1.1 消息的解析和语义是由本文档而不是 MIME 规范定义的。

19.4.2 转换到规范化形式 （Conversion to Canoical Form）

RFC 2045 [7]要求一互联网邮件实体在被传输之前要被转换成一个规范化的形式，这在 RFC2049[48]里第四章里描述的。本文档的 3.7.1 节描述了当用 HTTP 协议传输时允许使用的“text”子媒体类型的形式。RFC2046 要求类型为“text”的内容要用 CRLF 表示为换行符，以及在换行符外禁止使用 CR 或 LF。

RFC 2046 需要像在“text”类型的内容里一样，用 CRLF 表示行间隔符并禁止在行间隔符序列以外使用 CR 或者 LF。HTTP 允许 CRLF，单个 CR，和单个 LF 来表示一个换行符在一个文本内容消息中。

在可能的地方，从 HTTP 到 MIME 严格环境的代理或网关应该把 RFC2049 里描述的 text 媒体类型里所有换行符转换成 RFC2049 里 CRLF 的规范形式。然而，注意这可能在 Content-Encoding 出现的时候，以及 HTTP 允许利用一些没有利用 13 和 10 代表 CR 和 LF 的字符集时候都会变得复杂。

实现者应该注意转换将会破坏任何应用于源内容（original content）的密码校验和，除非源内容已经是规范化形式。因此，对任何在 HTTP 中使用校验和的内容被建议要表示为规范化形式。

19.4.3 日期格式的转换 （Conversion of Date Formate）

为了简化日期比较的过程，HTTP/1.1 使用了一个限制的日期格式（3.3.1 节）。其它协议的代理和网关应该保证任何消息里出现的 Date 头域应该遵循 HTTP/1.1 规定的格式，如果有必要需要重写此日期。

19.4.4 Content-Encoding 头域介绍 （Introduction of Content-Encoding）

RFC 2045 不包含任何等价于 HTTP/1.1 里 Content-Encoding 头域的概念。因为这个头域作为媒体类型（media type）的修饰，从 HTTP 协议到 MIME 遵守协议的代理和网关在转发消息之前必须既能改变 Content-Type 头域的值，也能解码实体主体（entity-body）.。（一些为互联网邮件类型的 Content-Type 的实验性应用有使用一个媒体类型参数“; conversions=<content-coding>”去执行等价于 Content-Encoding 的功能。然而，此参数并不是 RFC2045 的部分）

19.4.5 没有 Content-Transfer-Encoding 头域

HTTP 不使用 RFC 2045 里的 Content-Transfer-Encoding（CTE）头域。从使用 MIME 协议到 HTTP 的代理和网关在把响应消息发送给 HTTP 客户端之前必须删除任何非等价（non-identity，译注：identity 编码，表示没有进行任何编码）CTE（"quoted-printable"或"base64"）编码。

从 HTTP 到 MIME 协议遵循的代理和网关要确保消息在那个协议安全传输上是用正确的格式和正确的编码，“安全传输”是通过使用的协议的限制而被定义的。这样一个代理或网关应该用

合适的 Content-Transfer-Encoding 头域来标记数据如果这样做将提高安全传输的可能性。

19.4.6 Transfer-Encoding 头域的介绍

HTTP/1.1 介绍了 Transfer-Encoding 头域（14.41 节）。代理/网关在转发经由 MIME 协议的消息之前必须移除任何传输编码。

一个解码"chunked"传输代码（3.6 节）的程序可以用代码表示如下：

```
length := 0
read chunk-size, chunk-extension （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
```

19.4.7 MHTML 和行长度限制

和 MHTML 实现共享代码的 HTTP 实现需要了解 MIME 行长度限制。因为 HTTP 没有这个限制，HTTP 并不折叠长行。用 HTTP 传输的 MHTML 消息遵守所有 MHTML 的规定，包括行长度的限制和折叠，规范化等，因为 HTTP 传输所有消息主体（见 3.7.2）并且不解析消息的内容和消息中包含任何 MIME 头域。

19.5 其它特征

RFC 1945 和 RFC 2068 里一些协议元素被一些已经存在的 HTTP 实现使用，但是这些协议元素对于大多数 HTTP/1.1 应用程序既不兼容也不正确。实现者被建议去了解这些特征，但是不能依赖于它们的出现或不依赖于与其它 HTTP/1.1 应用程序的互操作性。这些特征中的一些特征描述了实验性的特征，以及还有一些特征描述了没有在基本 HTTP/1.1 规范里被描述的实验性部署特征。

一些其它头域，如 Content-Dispositon 和 Title 头域，他们来自于 SMTP 和 MIME 协议，他们同样经常被实现（见 2076[37]）.

19.5.1 Content-Disposition

Content-Disposition 响应头域被建议作为一个这样的用途，那就是如果用户请求要使内容被保存为一个文件，那么此头域被源服务器使用去建议的一个缺省的文件名。此用途来自于 RFC1806[35]关于对 Content-Disposition 的定义。

```
content-disposition = "Content-Disposition" ":"
                              disposition-type *（ ";" disposition-parm ）
```

```
disposition-type = "attachment" | disp-extension-token
disposition-parm = filename-parm | disp-extension-parm
filename-parm = "filename" "=" quoted-string
disp-extension-token = token
disp-extension-parm = token "=" （ token | quoted-string ）
```

一个例子是：

```
Content-Disposition: attachment; filename="fname.ext"
```

接收用户的代理不应该（SHOULD NOT）关注任何在 filename-parm 参数中出现的文件路径信息，这个参数被认为在这次仅仅是应用于 HTTP 实现。文件名应该（SHOULD）只被当作一个终端组件。

如果此头域用于一个 Content-Type 为 application/octet-stream 响应里，那么含义就是用户代理不应该展现响应，但是它应该直接进入一个‘保存响应为…’对话框。

见 15.5 节关于 Content-Disposition 的的安全问题。

19.6 和以前版本的兼容

要求和以前的版本的兼容超出了协议规范的范围。然而 HTTP/1.1 有意设计成很容易支持以前的版本。必须值得注意的是，在写这份规范的时候，我们希望商业的 HTTP/1.1 服务器去：

--接受 HTTP/0.9，1.0 和 1.1 请求的请求行（Request-Line）格式；
--理解 HTTP/0.9，1.0 或 1.1 格式中的任何有效请求；
--恰当地用客户端使用的主要版本来响应。

并且我们希望 HTTP/1.1 的客户端：

--接受 HTTP/1.0 和 1.1 响应的状态行（Status-Line）格式；
--懂得 HTTP/0.9，1.0 或 1.1 的格式的任何有效的响应。

对大多数 HTTP/1.0 的实现，每一个连接要在请求之前被客户端建立，并且在发送响应之后要被服务器关闭。一些实现了在 RFC 2068 [33]的 19.7.1 节描述的持久连接的 Keep-Alive 版本。

19.6.1 对 HTTP/1.0 的改变

这一部分总结 HTTP/1.0 和 HTTP/1.1 之间主要的区别。

19.6.1.1 对多主机 web 服务器和保留 IP 地址简化的改变

客户端和服务器都支持 Host 请求头域，并且如果 Host 请求头域在 HTTP/1.1 请求里没有出现必须报告一错误，并且服务器能接受一个绝对 URIs（5.1.2 节），这些要求是此规范里最重要的改变。

旧的 HTTP/1.0 客户端会认为 IP 地址和服务器是一对一关系；没有其它机制去区分 IP 地址和目标服务器。上面的改变将允许互联网，一旦旧客户端不再普遍使用时，可以去支持一个 IP 地址对应多个 web 站点的情况，这大大简化了大型的 web 运行服务器，在那上面上分配多个 IP 地址给一个主机（host）会产生严重问题。互联网照样能恢复这样一个 IP 地址，此 IP 地址作为特殊目的被分配给被用于根级（root-level）HTTP URLs 的域名。给定 web 的增长速度和服务

器的部署数量，那么所有 HTTP 实现（包括对已存 HTTP/1.0 应用程序）应该正确地满足下面这些需求：

--客户端和服务器都必须支持 Host 请求头域。

--发送 HTTP/1.1 请求的客户端必须发送 Host 头域。

--如果 HTTP/1.1 请求不包括 Host 请求头域，服务器必须报告错误 400（Bad Request）。

--服务器必须接受绝对 URIs（absolute URIs）。

19.6.2 和 HTTP/1.0 持久连接的兼容

一些客户端和服务器可能希望和一些以前持久连接实现的 HTTP/1.0 客户端和服务器保持兼容 。HTTP/1.0 持久连接需要显式地协商，因为持久连接不是 HTTP/1.0 的缺省行为。关于持久连接的 HTTP/1.0 实验性实现存在缺陷，并且 HTTP/1.1 里的新的功能被设计成去矫正这些问题。这个缺陷是：一些已经存在的 1.0 客户端可能会发送 Keep-Alive 头域给不理解持久连接的代理，然后代理会把此 Keep-Alive 转发给下一个内向（inbound）服务器。结果是 HTTP/1.0 客户端必须禁止利用 Keep-Alive，当和代理会话的时候。

然而，和代理进行会话最重要是利用持久连接，所以那个禁止很显然不能被接受。因此，我们需要一些其它的机制去指明需要一个持久连接，并且它必须能安全地被使用甚至是当和忽略持久连接的旧代理会话。持久连接缺省是为 HTTP/1.1 消息的；为了声明非持久连接（见 14.10 节），我们介绍一个新的关键字（Connection：close）。

持久连接的源 HTTP/1.0 的形式（Connection: Keep-Alive 和 Keep-Alive 头域）在 RFC2068 里说明

19.6.3 对 RFC 2068 的改变

这篇规范已经被仔细的审查来纠正关键字的用法和消除它们的歧义；RFC 2068 在遵守 RFC 2119 [34] 制定的协定方面有很多问题。

澄清哪个错误代码将会使入流服务器失灵（例如 DNS 失灵）。（10.5.5 节）

CREATE 有一个特点当一个资源第一次被创建的时候必须需要一个 Etag。（10.2.2 节）

Content-Base 头域从此规范里删除了：它无法广泛的实现，并且在没有健壮的扩展机制的情况下，没有简单的，安全的方式去介绍它。并且，它以一种相似的而不是相同的方式在 MHTML[45]里被使用。

略…..

20 索引 （Index）

21 全部版权声明

略…..