超文本传输协议 HTTP1.1（RFC2616）

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，合作式，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，错误代码和报头[47]来实现。HTTP 的一个特点是数据表现形式是可输入的和可协商性的，这就允许系统能被建立而独立于数据传输。

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

1 引论

1.1 目的

超文本传输协议（HTTP）是一种为分布式，合作式，多媒体信息系统服务，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。它允许消息以类似MIME的格式传送，包括有关数据传输的维护信息和关于请求/响应的句法修正。但是，HTTP/1.0没有充分考虑到分层代理，缓存的作用以及对稳定连接和虚拟主机的需求。并且随着不完善的应用程序的激增，HTTP/1.0迫切需要一个新的版本,以便使两个通信应用程序能够确定彼此的真实性能。

这里规定的协议叫做"HTTP/1.1".这个协议与HTTP/1.0相比，要求更为严格，以确保各项功能得到可靠实现。

实际的信息系统除了简单的检索外，要求更多的功能性（functionality），包括查找（search），前端更新（front-end update）和注解（annotation）。HTTP允许可扩充的方法集和报头集以指示请求的目的[47]。它是建立在统一资源标识符（URI）[3]提供的地址（URL）[4]和名字（URN）上[20]，以指出方法应用于哪个资源的。消息以类似于一种叫做多用途网络邮件扩展（MIME）[7] 的互联网邮件的格式传送。

HTTP也是用于用户代理之间及代理/网关到其他网络系统的通用通信协议，这样的网络系统可能由SMTP[16],NNTP[13],FTP[18],Gopher[2]和WAIS[10]协议支持。这样，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"MUST", "MUST NOT", "REQUIRED", "SHALL","SHALL NOT","SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", 和 "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MUST或REQUIRED等级以及所有SHOULD等级的要求，则被称为非条件遵循（unconditionally compliant）的；若满足所有MUST等级的要求但不能满足所有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 节详述。

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（包括出错响应）.

变量（variant）

在任何给定时刻，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。使用变量这个术语并不是意味着资源（resource）是必须由内容协商决定的.

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（网络穿越机器人）,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，它既担当客户端的角色也担当服务器的角色。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）需要代理授权和代理识别，但不修改请求或响应。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明确指出，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、取回和删除。缓存里存放可缓存响应（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，但高速缓存不能被一个充当隧道（tunnel）的服务器使用。

可缓存（cacheable）

我们可以说响应（response）是可缓存的，如果一个缓存（cache）为了响应后继请求而被允许存储响应消息（response message）的副本。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缓存副本的约束。

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）。

如果响应被源服务器（origin server）验证是有效性（validity）的，那么这个响应也同样是第一手的。

明确过期时间（explicit expiration time）

是源服务器希望实体（entity）如果没有被进一步验证（validation）就不要再被缓存（cache）返回的时间。

启发式过期时间（heuristic expiration time）

当没有明确终止时间（explicit expiration time）可利用时，由缓存所指定的终止时间.

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确认的时间点到现在的时

间。

保鲜寿命（freshness lifetime）

一个响应产生的时间点到过期时间点之间的长度。

保鲜（Fresh）

如果一个响应的年龄还没有超过保鲜寿命（freshness lifetime），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），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）具有语意透明性时，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：实体头（entity tag）或最后更改时间（last-modified time）等），这些协议元素被用于识别缓存里保存的数据（即缓存项）是否是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:所有消息都从上游流到下游.

向内/向外（inbound/outbound）

向内和向外指的是消息的请求和响应路径:"向内"即"移向源服务器","向外"即"移向用户代理（user agent）".

1.4 总体操作

HTTP 协议是一种请求/响应协议。 与服务器建立连接后，客户端以请求方法，URI 和协议版本号，然后紧接着跟随一个类 MIME（MIME-like）消息，这个类 MIME 消息包括请求修饰符，客户信息和可能的消息主体。服务器以一个状态行并跟随一个类 MIME（MIME-like）消息响应，状态行包含消息的协议版本和成功出错的状态码，类 MIME 消息包含服务器信息,实体元信息,和可能的实体。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）开始的，由应用到一个需要源服务器资源的请求构成。最简单的情形，可以经用户代理（UA）和源服务器（O）之间的单一连接（v）完成。

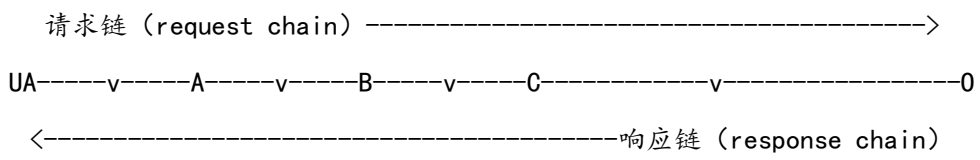
请求链（Request chain）---------->

用户代理（UA）----------单一连接（v）----------源服务器（O）

<----------响应链（response chain）

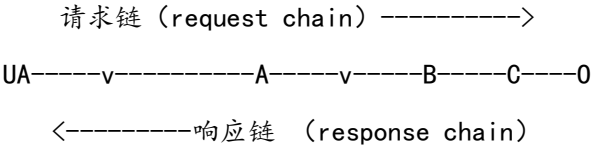
当一个或多个中间者在请求/响应链中出现的时候，会出现更复杂的情形。常见的中间者有三种：代理（proxy），网关（gateway）和隧道（tunnel）。代理（proxy）是一种

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），它充当一个层（layer），这个层在服务器之上，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。



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单独的连接。这个特性很重要，因为某些HTTP通信选项可能只能应用于与最近的非隧道邻接点的连接，只能应用于链的端点（end-point）的连接，或者能应用于此链的所有连接。图表尽管是线性的，每个参与者可能忙于多路同时通信。例如，B可以接收来自不同于A的许多客户的请求，并且/或者可以把请求转到不同于C的服务器，与此同时，它还在处理A的请求。

任何非隧道的通信成员都可能会采用一个内部缓存（cache）来处理请求。如果沿着链的通信成员对请求采用了缓存响应，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，这是在假定B拥有一个来自O（通过C）的以前请求的响应副本，但此响应尚未被UA或A缓存。



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，这些修饰符对缓存动作有特殊的要求。缓存动作和缓存响应的HTTP行为要求将在第13节定义。

实际上，目前万维网上有多种结构和配置的缓存（cache）和代理（proxy）正在被使用。这些系统包括节省带宽的缓存代理（proxy cache），可以广播或多点传送缓存数据的系统，通过CD-ROM分配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，并且可以通过PDA进行低耗无线的，断续连接的访问。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，让它满足那些需要较高可靠性，即使不能达到较高可靠性的要求，也要也让它至少可以指示故障的网络应用的要求。

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。这并不妨碍HTTP的实现被应用于互联网（internt）或其它网的协议之上。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）；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说明书的范围。

大部分HTTP/1.0的实现都是针对每个请求/响应交换产生一个新的连接。而http/1.1中，一个连接可以被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（译注：如：CRLF，DIGIT等等都是规则名），规则名里不能包含"<"和">"，通过等于号把规则名和规则定义（definiation）分离开。空白（white space）是有意义的，因为可以用缩进（indentation，译注：缩进就是空白，后面会讲到LWS） 把规则定义显示成多行。某些基本规则（basic rule，译注：2.2节说明基本规则的语法）使用大写字母包含在规则定义里， 你如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识别规则名（译注：LWS，HT等都是规则名）。

"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）"的符合的字符串是"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"至少需要一次；"1*2element"允许一次或两次。

[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）正好出现了<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（implied） *LWS

本说明所描述的语法是基于字的。除非特别注明，线性空白可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，这并没有改变一个域的解释。任何两个标记（token）之间必须有至少一个分割符，否则将会被理解为单一标记。

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，但这除了实体主体（endtity-body）外（要求比较松的应用见附录19.3）。实体主体（entity-body）的行尾标志是由它的关联媒体类型定义的，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。接收者在解析域值或将消息转送到下游(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 规则仅仅适用于域内容和域值的描述，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          = 1*<除 CTLs 与分割符以外的任意 CHAR >
separators     = " (" | ") " | "<" | ">" | "@"
                 | "," | ";" | ":" | "\" | <">
                 | "/" | "[" | "]" | "?" | "="
                 | "{" | "}" | 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。注释只可以作为域定义的一部分。在其他域里，圆括号被视作域值的一部分。

```
comment        = " (" *( ctext | quoted-pair | comment )  ") "
ctext          = <any TEXT excluding "(" and ")">
```

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（"\"）可以被作为单个字符的引用机制，但是必须要在引号和注释区之内。

```
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。协议的版本号是为了让发送端指明消息的格式和它的能力，这是为了进一步的 HTTP 通信，而不仅仅是获得

通信的特征。协议版本是不需要修改的，当消息组件的增加不会影响通信行为或着只增加了扩展的域值。<minor>数字是递增的，当协议会因为添加一些特征而做了修改的时候。但这些变化不会影响通常的消息解析算法，但是它会给消息添加语意（semantic）并且会暗示发送者具有额外的能力。<major>数字也是不断递增的，当协议的消息格式每次发生变化时。

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。

一个应用程序发送请求或响应消息，如果请求或响应消息里的 HTTP-Version 是"HTTP/1.1"，那么此应用程序必须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"包含在他们的消息里，并且对任何不兼容 HTTP/1.1 的消息必须这么做。关于何时发送特定的 HTTP-Version 值的细节，参见 RFC2145。

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。

代理（proxy）和网关（gateway）应用程序需要被仔细对待，当转发（forwarding）消息的协议版本不同于代理或网关应用程序的协议版本。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，它也必须要降低请求的版本，要么以一个错误响应，要么切换到隧道行为（tunnel behavior）。

由于自从 RFC 2068[33]发布后，产生了与 HTTP/1.0 代理（proxy）的互操作问题，所以缓存代理（caching proxy）必须能改变请求（request），使请求能到达他们能支持的最高版本，但网关（gateway）可以这么做也可以不这么做，而 tunnel 不能这么做。代理（Proxy）/网关（gateway）的响应（Response）必须和请求（request）的 HTTP 版本的 major 数字相同。

> 注：在 HTTP 版本间的转换可能包含头域（header field）的改变，而这些改变会可能会根据 HTTP 版本而被要求或被拒绝。

3.2 统一资源标识符（URI）

URIs 的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，统一资源定位器只是格式化的字符串，它通过名称，地址，或任何别的特征识别资源。

3.2.1 一般语法

根据使用的背景，HTTP 里的 URI 可以表示成绝对（absoulute）形式或相对形式（相对于已知的 URL）。两种形式的区别是根据这样的事实：绝对 URI 总是以一个方案（scheme）名作为开头，其后是一个冒号。关于 URL 更详尽的信息请参看"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它们资源的 URI，并且应该能够处理无限长度的 URI，这种无效长度的 URL 可能会在客户端以 GET 形式的请求产生。服务器应该返回 414 状态码（此状态码代表 Request-URI 太长），如果服务器不能

处理太长的URI 的时候。

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

通过HTTP 协议, http 方案（http scheme）被用于定位网络资源（resourse）的位置。本节定义了这种方案的语法和语义。

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

如果端口为空或未给出，就假定为80。语义即：已识别的资源放在服务器上，在那台主机的那个端口上监听TCP 连接。这时资源的请求的URI 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的域名(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）接收了一个全称的域名，代理不必改变主机。

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）名的比较必须不必分大小写;
- 方案（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]），其它字符都等效于它们的"%HEXHEX"编码.

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,它是一个固定长度的 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]，并且没有年份。如果 HTTP/1.1 客户端和服务器解析日期，他们必须能接收所有三种格式（为了兼容 HTTP/1.0），但是它们只能产生 RFC 1123 里定义的日期格式来填充头域（header field）用到日期的地方。

注：日期值的接收者被鼓励能健壮的接收可能由非 HTTP 应用发来的日期值，例如有时可以通过代理（proxy）/网关（gateway）向 SMTP 或 NNTP 获得或转发消息。

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，不能包含一个额外的 LWS，除非此 LWS 作为在下面的 Http-date 语法中指定的 SP。

```
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：HTTP 对日期/时间格式的要求仅仅应用在协议的消息（译注:原文是 protocol stream，便于理解这里译作消息）里。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

一些 HTTP 头域（header field）允许用整数秒表示时间值，整数秒用十进制表示，此整数秒表示消息被接收后时间。

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

3.4 字符集

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,此方法是用一个或多个表将一个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）。注意反方向的无条件转换（译注:从一个字符序列到一个字节序列的转换）是不需要的，因为并不是所有的字符都能在一个给定的字符集里得到,一个字符集可能提供多个字节序列表征一个特定的字符。这个定义是为了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如 ISO-2022 技术用到的。然而，与 MIME 字符集名字相关的定义必须要充分说明从字节到字符的映射。特别的，使用外部轮廓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 字符集是用不区分大小写的标记（token）表示。所有的标记集由 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，但任何标记值如果它已经在 IANA 字符集注册机构注册了则必须表示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 丢失的字符集（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，译注：这里翻译成"实体主体"因为 Content-Type 头是实体头，消息头可以分为实体头，常用头，请求头，响应头，在译文中多次用到"头"和"头域"，如消息头，消息头域，其实是同一个意思，HTTP1.1 协议有时候概念并不是完全统一的）的字符集是什么。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 也应该这样做，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，可认为是接收端）开始显示一个文档时，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）

内容编码值（content coding value）表示一种已经或可以应用于实体的编码转换（encoding transformation)。内容编码主要用于文档的压缩或其它有效的变换，但这种变换需要不能丢失文档的媒体类型（media type，译注：文档一般会有媒体类型，这通过在 content-type 里指定）的特性，也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)。实体经常被编码的储存，然后直接传送，接收端只能解码。

```
content-coding   = token
```

所有内容编码值（content-coding value）是不区分大小写的。HTTP/1.1 在接收译码（14.3 节）和内容译码（Content-Encoding）（14.11 节）头域里使用内容编码值（content-coding value)。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）充当内容编码值标记（token）的注册机构。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收译码（Accept-Encoding）头域，但不能被用在内容译码（Content-Encoding）头域。.

新的内容编码值标记（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）

传输编码值（transfer-coding value，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，见下面红字的规则）被用来表示一个已经，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

所有传输编码值（transfer-coding value，译注：上面红体字等号右边规则表达式所表达的值）是大小写不敏感。传输编码值在 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中断消息。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，译注：transfer 应该是转移，迁移的意思，又例如 HTTP 协议，应该翻译成"超文本转移协议"，但是历史上都翻译成"超文本传输协议"，所以这里翻译成"超文本传输协议"）值有相类似型，它被定义能够实现在 7 位传输服务上保证二进制数据的传输安全。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关注点。在 HTTP 中，消息主体存在不安全的隐患，因为有时候很难确定消息主体的长度，在共享的传输上加密数据也会带来安全性问题（7.2.2 节）。

网络分配数字权威（IANA）担任注册传输编码值标（token）记的角色。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）。

新的传输编码值标记应该注册，这同新的内容编码值标记也需要注册一样。

接收端接收到一个带有传输编码（transfer-coding）（译注：通过消息头域 transfer-encoding 指明此实体主体的传输编码）的实体主体（entity body），如果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），并且要切断联系。服务器不能向 HTTP/1.0 客户发送传输编码。

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，译注：消息主体与实体主体是有区别的，后面章节将会介绍）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)。这允许发送端能动态生成内容，并能携带有用的信息，这些信息能让接收者判断消息是否接收完整。

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）以任一大小为 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附加的HTTP头域（header field）Trailer 头域（Trailer header field，译注：Trailer 头是常用消息头，在 14.40 节说明）被应用来指明哪些头域被包含在块编码的尾部（trailer） （见 14.40 节）

如果服务器要用块传输编码进行响应，它不能包含尾部（trailer），除非以下至少一条为真：

a) 如果此响应的请求包括一个 TE 头域，并且它指明了传输编码中的"trailers"是可接受的，当响应的传输编码（transfer-coding）是块编码时。这在 14.39 节中描述；或者

b) 如果服务器是响应的源服务器，并且接收端接收块传输编码响应但不会去理会响应的尾部（trailer，译注：尾部包含头域，头域就是消息的元数据（metadata））并且这种方式源服务器是可以接受的，这时服务器是不需要把尾部（trailer）包含进消息的块传输编码中去的。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。

此要求防止了一种互操作性的失败，当消息被一个 HTTP/1.1（或更迟的）代理（proxy）接收并且转到一个 HTTP/1.0 的接收端的时候。

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正文（chunked-body）进行解码。

所有 HTTP/1.1 应用程序必须能接收和解码块（chunked）传输译码，并且必须忽略它们不能理解的块扩展（chunk-extentsion，译注：见上面的规则表达式）。

3.7 媒体类型（Media Type）

HTTP 在 Content-Type（14.17 节）实体头域和 Accept 请求头域里利用网络媒体[17]类型，这是为了提供公开的，可扩展的数据打印和类型协商。

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

type = token

subtype = token

参数（parameter）以一种属性/值对的形式跟随 type/subtype（如 3.6 节定义）。

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)。当向一个旧的 HTTP 应用程序发送数据时，发送端只有在被 type/subtype 定义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）

网络媒体类型以规范化的格式被注册。一个实体主体（entity-body）通过 HTTP 消息传输，它必须在传输前以一种合适的规范化的格式表征除了文本类型（text type），文本类型将会在下一段阐述。

当消息以一种规范化的格式表现时，文本类型的子类型（subtype）运用 GRLF 作为文本里的换行符。HTTP 放松了这个要求，并且允许文本媒体以一个 CR 或 LF 代表一个换行符传输，并且这样做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，如果文本字符集（character set）不能用字节 13 和 10 来分别地表征 CR 和 LF 因为存在一些多字节字符，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 消息结构里，一个单独的 CR 或 LF 都不能代替 CRLF（如头域和多边界体（multipart boundaries）结构里）。

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（译注：被编码前的数据）在被编码前必须是一种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）。当发送者没有在媒体类型（media-type）里指明 charset 参数（parameter）时，文本类型的子媒体类型（subtype）被认为是缺省的 ISO-8859-1 字符集当被接收者接收后。数据必须被合适的字符集标识。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"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）和任何其他媒体类型的消息主体相同对待：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 用户代理一样的或者相似的行为，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（bady-part）里的 MIME 头域对于 HTTP 并没有太大意义除了 MIME 语意。

通常， 一个 HTTP 用户代理应该遵循与一个 MINE 用户代理相同或相似。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"类型已经被规范的定义为传送窗体数据（译注：一般用窗体上传数据时,上传的数据类型就是为 multipart/form-data 类型),当用 POST 请求方法处理数据时。这在 RFC 1867[15]里定义。

3.8 产品标记 （product Tokens）

产品标记用产品名和版本号识别通讯应用软件。很多头域都会利用产品标记，它允许构成应用程序重要部分的子产品被以空白分隔列举。通常，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
```

产品标示应言简意赅。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）来表针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不被客户端接受。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，因为这些值仅仅表现一中相对的降级。

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部分或多部分构成：一个主语言标签和可能为空的子标签系列。

```
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，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）而使用和比较在 13.3.3 节被定义。一个实体标签由一个给定的晦涩的引用字符串（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，并且能彼此之间能互相替代，并且也不会在语义上有太大改变。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于不同的 URI 请求用来获得的实体。相同实体标签值运用于不同 URI 请求获得的实体，并不意味着这些实体是等价的。

3.12 范围单位（Range Units）

HTTP/1.1 允许一个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）。任何实体根据不同的结构化单元都能被分解是子范围

```
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 实现时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靠有关对范围的了解。

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和从服务器到客户的响应组成.

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）。两种类型的消息由开始行（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，一个以 CRLF 为前缀的什么也没有的行），一个可有可无的消息主体（message-body）。

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。

一般一个有问题的 HTTP/1.0 客户端会在 POST 请求消息之后产生额外的 CRLF。为了重述什么是 BNF 明确禁止的，一个 HTTP/1.1 客户端不能在请求前和请求后加一些不必要的 CRLF。

4.2 消息头 （Message Headers）

HTTP 头域包括常用头（4.5 节），请求头（5.3 节），响应头（6.2 节）和实体头（7.1 节）域。它们遵循 RFC822[0]3.1 节中给出的同一个常规的格式。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。头域能被延升多行，通过在这些行前面加一些 SP 或 HT。应用程本应该遵循"常用格式"当产生 HTTP 消息时，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 可能会被移除而不会改变域值的语意。任何出现在 filed-content 之间的 LWS 可能会被一个 SP 代替在解析域值之前或把这个消息往下流传递时。

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。

多个消息头域使用同一个域名（filed-name）可能会出现在一些消息中，如果一个头域的域值被定义成一个以逗号隔开的列表。把相同名的多个头域结合成一个"域名：域值"对的形式而不改变消息的语意，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。同名的头域的接收顺序对合并的域值的解释有重要意义，所以代理（proxy）不能改变域值的顺序，当它把此消息再次转发时。

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)。这些消息主体（message-body）仅仅当传输编码（transfer-coding）应用于传输译码（Transfer-Encoding）头域时才和实体主体（entity-body）不同，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，它是为了保证安全和适合的消息传输。传输译码（Transfer-Encoding）头域是消息的属性，而不是实体的属性，并且沿着请求/响应链能被添加或删除。（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）头域来暗示的。一个消息主体（message-body）不能被包含在请求里如果请求方法（见 5.1.1 节）不允许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)，那么消息主体应该被忽略当服务器处理这个请求时。

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，虽然它可能长度为零.

4.4 消息的长度（Message Length）

一条消息的传输长度（transfer-length）是消息主体（message-body）的长度，当消息主体出现在消息中时；那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 请求的响应）总是被头域后的第一个空行（译注:CRLF）终止，不管消息里是否有实体头域（entity-header fields）。

2. 如果 Transfer-Encoding 头域（见 14.41 节）出现，并且它的域值不是"identity"，那么传输长度（transfer-length）被"块"传输编码定义，除非消息因为关闭连接而被终结了。

3. 如果 Content-Length 头域（属于实体头域）（见 14.13 节）出现，那么它的十进制值（以字节表示）就代表实体主体长度（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）也代表传输长度（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，消息里应用了传输译码（Transfer-Encoding）头域）。如果一个消息即有传输译码（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，服务器必须能定界此消息利用这节的 1,3 或 5 项里定义的方法。

5. 通过服务器关闭连接能确定消息的传输长度。（关闭连接并不能用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）的头域，除非服务器是和 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），那么服务器应该以 400 响应（错误的请求）如果他不能判断消息长度的话，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），因此可以允许这种机制来处理消息当消息的长度不能被提前决定时。

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 用户代理必须通知用户当一个无效的长度接收了。

4.5 常用头域（General Header Fields）

有一些头域即适用于请求也适用于响应消息，但是这些头域并不适合被传输的实体。这些头域只能应用与被传输的消息。

```
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
```

常用头域的名能被扩展，但这要和协议版本的变化相结合。然而，新的或实验性的头域可能被赋予常用头域的语意，如果通信里的所有参与者都认为他们是常用头域。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，最后以 CRLF 结束。元素是以 SP 字符分隔。CR 或 LF 是不被允许的除了最后的 CRLF。

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

5.1.1 方法（Method）

方法标记指示了在被 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"                ; Section 9.2
               | "GET"                    ; Section 9.3
               | "HEAD"                   ; Section 9.4
               | "POST"                   ; Section 9.5
               | "PUT"                    ; Section 9.6
               | "DELETE"                 ; Section 9.7
               | "TRACE"                  ; Section 9.8
```

| "CONNECT" ; Section 9.9

| extension-method

extension-method = token

资源允许的方法由 Allow 头域指定（14.7 节）。响应的返回码总是通知客户是否一个方法对一个资源在当前是被允许的，因为被允许的方法集合能被动态的改变。一个源服务器应该返回 405 状态码（方法不能允许）如果此方法对于一个请求资源在服务器那里不被允许，并且返回 501 状态码（没有实现）如果此方法不被源服务器识别或实现。方法 GET 和 HEAD 必须被所有常规目的的服务器支持。所有其它的方法是可选的；然而，如果上面的所有方法都被实现，这些方法必须遵循的语意和第 9 节指定的相同。

5.1.2 请求 URL（Request-URI）

Request-URI 是一种全球统一资源标识符（３.2 节），并且它指定请求的资源。

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

Request-URI 的 OPTIONS 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，但是能应用于服务器，并且只能被允许当使用的方法不能应用于资源的时候。举例如下

OPTIONS * HTTP/1.1

当向代理（proxy）提交请求时，绝对 URI（absoluteRUI）是不可缺少的。代理（proxy）可能会被要求再次转寄此请求。注意，代理可能再次提交此请求到另一个代理或直接给源服务器。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，本地变量，数字 IP 地址。一个请求行（Request-Line）的例子如下：

GET 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绝对 URI（absoluteURI）地址，所有基于 HTTP/1.1 的服务器必须接受绝对 URL 地址，即使 HTTP/1.1 的客户端只向代理产生给绝对 URI（absoluteURI）。

authority 部分只被用于 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)必须被用作 Request-URI,并且此 URI(authority)的网络位置必须在 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，这是会发送下面行：

GET /pub/WWW/TheProject.html HTTP/1.1

Host: www.w3.org

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

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解码它。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"当它转寄此请求到下一个服务器时，除了根据上面声明的把一个空的 abs_path 用"/"代替。

注：不重写的规则防止了代理（proxy）改变请求的意思，当源服务器不能正确的利用非保留（non-reserved）的 URI 字符时 。实现者应该知道某些 pre-HTTP/1.1 代理（proxy）能重写 Request-URI。

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

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

如果源服务器资源不依赖于请求的 Host 头域指定的主机(译注:可能会存在虚拟主机)，那么它会忽略请求的头域当它决定其资源的时候。(对于一个支持 Host 的 HTTP/1.1 服务器，在 19.6.1.1 节描述了其他的需求)。

源服务器如果根据请求里的主机区别资源（这是因为存在虚拟主机和虚拟主机名），它必须遵循下面的规则去决定请求的资源当出现一个 HTTP/1.1 请求时：

1. 如果 Request-URI 是绝对地址（absoluteURI），这时请求里的主机存在于 Request-URI 里。任何出现在请求里 Host 头域值应当被忽略。
2. 假如 Request-URI 不是绝对地址（absoluteURI），并且请求包括一个 Host 头域，则主机由该 Host 头域值决定.
3. 假如由规则 1 或规则２定义的主机是一个无效的主机，则应当以一个 400（错误请求）错误消息返回。

缺少 Host 头域的 HTTP/1.0 请求的接收者可能会推测决定什么样的资源被请求（例如：检查 URI 的路径对于某个特定的主机是唯一的）。

5.3 请求报头域（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符，这和程序语言方法调用的参数语义是一样的。

```
request-header = Accept                   ; Section 14.1
               | Accept-Charset           ; Section 14.2
               | Accept-Encoding          ; Section 14.3
               | Accept-Language          ; Section 14.4
               | Authorization            ; Section 14.8
               | Expect                   ; Section 14.20
               | From                     ; Section 14.22
               | Host                     ; Section 14.23
               | If-Match                 ; Section 14.24
               | If-Modified-Since        ; Section 14.25
               | If-None-Match            ; Section 14.26
               | If-Range                 ; Section 14.27
               | If-Unmodified-Since      ; Section 14.28
```

```
                | Max-Forwards              ; Section 14.31
                | Proxy-Authorization      ; Section 14.34
                | Range                    ; Section 14.35
                | Referer                  ; Section 14.36
                | TE                       ; Section 14.39
                | User-Agent               ; Section 14.43
```

请求头域的名字是能结合于协议版本而能被扩展的。然而新的或实验性的头域应该被给出请求头域的语义如果通信的所有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）

接收和翻译一个请求消息后，服务器发出一个 HTTP 响应消息。

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

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，除了最后的回车或换行外，中间不允许有回车换行.

```
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）

状态码是试图理解和满足请求的三位数字的整数码，这些码的完整定义在第十章。原因短语（Reason-Phrase）是为了给出的关于状态码的文本描述。状态码用于控制条件，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型。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　　- 接收到请求，继续进程.
- 2xx：成功　　- 步骤成功接收，被理解，并被接受
- 3xx：重发　　- 为了完成请求,必须采取进一步措施.
- 4xx：客户端出错　　- 请求包括错的顺序或不能完成.
- 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：   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：局部内容
       |"300"   ;   10.3.1 节：多样选择
       |"301"   ;   10.3.2 节：永久移动
       |"302"   ;   10.3.3 节：创建
       |"303"   ;   10.3.4 节：观察别的部分
       |"304"   ;   10.3.5 节：只读
       |"305"   ;   10.3.6 节：用户代理
       |"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：没找到
       |"405"   ;   10.4.6 节：不允许的方式
       |"406"   ;   10.4.7 节：不接受
       |"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：  停止
       |"411"   ;   10.4.12 节：  需要的长度
       |"412"   ;  10.4.13 节；  预处理失败
```

|"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: 服务不能实现

|"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，尽管那样的理解显而易见是很合算的。但是，应用程序必须了解由第一位数字指定的状态码的类型，任何未被识别的响应应被看作是该类型的 x00 状态，有一个例外就是未被识别的响应不能缓存。例如，如果客户端收到一个未被识别的状态码 431，则可以安全的假定请求有错,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）应当把实体和响应一起提交给用户，因为实体很可能包括人可读的关于解释不正常状态的信息。

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）指定的资源的更进一步的访问。

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

响应头域的名字能依赖于协议版本的变化而扩展。然而，新的或者实验性的头域可能会给于响应头域的语义如果通信的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）

如果不被请求的方法或响应的状态码所限制，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 实体报文域（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）从消息主体通过传输译码头域（Transfer-Encoding）译码得到，传输

译码用于确保消息的安全和合适的传输。

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 Content-Type 和 Content-Encoding 决定。这些头域定义了两层的顺序的编码模型：

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。没有缺省的译码。

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。如果并且只有媒体类型没有通过 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 持续连接（Persistent Connection）

8.1.1 目的

在持续连接之前，为获取每一个 URL 指定的资源都必须建立了独立的 TCP 连接， 这就加重了 HTTP 服务器的负担，易引起互联网的阻塞。嵌入的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提交多个请求。目前已有针对这些性能问题的分析以及原型实现的结果[26][30]。 对其他方法也有了初步探究，如 T/TCP [27]。

持续 HTTP 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，代理服务器，网关，隧道或缓存）的 CPU 时间，还节省了主机用于 TCP 协议控制块的内存。
- 能在连接上进行流水线请求方式。 流水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），并且此时只进行了一个 TCP 连接，从而效率更高，减少了时间。
- 网络阻塞会被减少，这是由于 TCP 连接后减少了包的数量，并且由于允许 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 改进的越来越好，因为在不需要关闭 TCP 连接的代价下可以报告错误。将来的 HTTP 版本客户端可以乐观的尝试一个新的特性，但是如果和老的服务器通信时，

在错误被报告后就要用旧的语义而进行重新尝试连接。

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续连接是任何HTTP/1.1 连接的缺省方式。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续连接，即便在接到服务器的出错响应时也应如此。

持续连接提供了一种可以由客户端或服务器发信号来终止 TCP 连接的机制。利用 Connect 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）

除非请求的 Connect 头域中包含了"close"标签,HTTP/1.1 服务器总可以假定 HTTP/1.1 客户端想要维持持续连接（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 Connect 头域。

一个 HTTP/1.1 客户端可能期望保持连接一直开着，但这必须是基于服务器响应里是否包含一个 Connect 头域并且此头域里是否包含"close"。如果客户端不想为更多的请求维持连接，它应该发送一个值为"close"的 Connect 头域。

如果客户端和服务器之一在发送的 Connect 头域里包含"close"，那么客户端的请求将会变为此连接的最后一个请求。。

客户端和服务器不应该认为持续连接是低于 1.1 的 HTTP 版本所拥有的，除非它被显示地指明了。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 流水线（pilelining）

支持持续连接（persistent conncetio）的客户端可以以流水线的方式发送请求（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。

假定持续连接并且在连接建立后进行流水线方式请求的客户端应该准备去重新尝试他们的连接如果第一个流水线请求方式尝试失败。如果客户端重新尝试连接，在客户端知道连接是持续之前客户端不能进行流水线发送请求。客户端必须准备去重新发送请求如果服务器在响应所有对应的请求之前关闭连接

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）进行流水线方式的请求。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。一个可能希望发送一个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 代理服务器 （Proxy Servers）

它是非常重要的因为代理服务器正确地实现了 Connect 头域的属性,这在 14.10 节指出了。

代理服务器必须分别向和它相连的客户端或者源服务器（或其他的代理服务器）指明持续连接。每一个持续连接只能应用于一个传输层连接。

代理服务器不能和一个 HTTP/1.0 客户端建立一个 HTTP/1.1 持续连接（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 实际的考虑 (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。代理服务器会选一个较高的值,因为客户端很可能会与同一服务器建立多个连接。持续连接方式的采用对于客户端与服务器的时限均未提出任何要求。

当客户端或服务器希望超时终止时, 它应该按规范终止传输连接。客户端与服务器端应始终注意对方是否终止了连接,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,或代理服务器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,服务器与代理服务器必须有能力从异步的连接终止事件中恢复。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新传输遗弃的请求序列而不需要用户进行交互来实现。对非等幂方法的请求不能自动进行重试请求,尽管客户代理(user agent)可能能提供一个人工操作去重试这些请求。应该用用户代理(user-agent)软件对应用程序语义识别的确认来替代用户的确认。如果再次重试请求失败,那么就不能再进行重复请求了。

服务器尽可能应该在每次连接中至少响应一个请求。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。

使用持续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服务器保持两个以上的连接。代理服务器与其它服务器或代理服务器之间应维护 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 消息传送要求(Message Transmission Requirements)

8.2.1 持续连接与流量控制(Persistent Connections and Flow Control)

HTTP/1.1 服务器应当保持持续连接并使用 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)客户端应在发送消息主体的时候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。若正文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来提前标记报文结束。若消息主体前有 Content-Length 头域,则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判定服务器是否愿意接受客户端发来的消息主体(基于请求头域)在客户端发送此请求消息主体前。 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合适的或会降低效率。

HTTP/1.1 客户端的要求:

- 若客户端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。

由于存在旧的实现方法，协议允许二义性的情形存在，这在客户端在发送"Expect:100-continue"后而没有接收一个 417（期望失败）状态码或着 100（继续）状态码的情况下发生。因此，当一个客户端发送此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）状态码响应并且继续从输入流里接收数据，或者以 final 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 final 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 final 状态码的响应，它就不能再去那个执行请求的方法（如：POST 方法，PUT 方法）。
- 如请求消息不含值为"100-continue"的 Expect 请求头域， 源服务器不应发送 100（继续）响应。当请求来自 HTTP/1.0（或更早）的客户端时也不得发送 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。
- 发送 100（继续）响应的源服务器必须最终能发送一个 final 状态响应，一旦请求消息主体被它接收到并且已经被它处理了，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，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 final 的状态响应，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用来防止服务器抵抗服务攻击,或者防止服务器被客户端实现攻击。

对 HTTP/1.1 代理服务器的要求：

- 若代理服务器接到一个请求,此请求包含值为"100-continue"的 Expect 请求头域，并且代理服务器可能知道下一站点的服务器遵循 HTTP/1.1 或更高版协议，或者不知道下一站点服务器的 HTTP 版本，那么它必须包含此 Expect 头域来转发此请求。
- 若代理服务器知道下一站点服务器版本是 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。
- 代理服务器应当维护一个缓存，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，那么代理服务器不能转发 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，并且客户端直接与 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，但新加的方法不能认为能分享与扩展的客户端和服务器同样的语义。

Hst 请求头域（见 13.23 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。

特别的，GET 和 HEAD 方法仅仅应该获取资源而不是执行动作（action）。这些方法应该被考虑是"安全"的。可以让用户代理用其他的方法，如：POST，PUT，DELETE，这样用户代理就能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，因此不需要对这些副作用负责。

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，有可能几个请求的序列是不等幂的，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。

这种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，那么这个生成的数据应该被作为实体在响应中返回，但这并不是过程的资源文本，除非资源文本恰好是过程的输出（译注：URI 指示的资源是动态生成的）。

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，这通过利用缓存的实体的更新，从而不用多次请求或传输客户已经拥有的数据。

如果请求方法包含一个 Range 头域,那么 GET 方法就变成"部分 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，这通过允许获取部分实体，从而不需要传输客户端已经拥有的数据。

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用于更新以前的那个资源的缓存实体.。如果出现一个新的域值指明了缓存实体和当前源服务器上实体的不同（可能因为 Content-Length,Content-MD5,ETag 或 Last-Modified 值的改变),那么缓存(cache)必须认为此缓存项是过时的（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此实体引用了一个新资源和一个 Location 头域（见 14.30 节）。

POST 方法的响应是可缓存的。除非响应里有 Cache-Control 或者 Expires 头域指示其响应不可缓存。然而，303（见其他）响应能被利用去指导用户代理（agent）去获得可缓存的响应。

POST 请求必须遵循 8.2 节里指明的消息传输需求。

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此 URI 所指定资源实体的修改版本。如果请求 URI（Request-URI）指定的资源不存在，并且此 URI 被用户代理（user agent，译注：用户代理可认为是客户浏览器）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 方法的响应不应该被缓存。

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：网关和代理服务器的区别是：网关可以进行协议转换，而代理服务器不能，只是起代理的作用，比如缓存服务器其实就是一个代理服务器），或者一个单独接收注释的实体。而 PUT 方法请求里有一个实体——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他 URI 指定的资源。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（永久移动了）响应；用户代理可以自己决定是否重定向请求。

一个独立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，此 URI 区别于其文章其他特殊版本的 URI。这种情况下，一个通用 URI 的 PUT 请求可能会导致其资源的其他 URI 被源服务器定义。

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 节中的消息传输要求。

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 动作没有通过，应该以 202（已接受）响应；如果 DELETE 方法请求已经通过了，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,）。最后的接收者或者是接收请求里 Max-Forwards 头域值为 0 源服务器或者是代理服务器或者是网关。TRACE 请求不能包含一个实体。

TRACE 方法允许客户端知道请求链的另一端接收什么，并且利用那些数据去测试或诊断。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去测试一串代理服务器是否在无限回路里转发消息。

如果请求是有效的，响应应该在响应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。TRACE 方法的响应不能不缓存。

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服务器（proxy，译注：可以为代理，也可以是代理服务器）。

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备用的响应，包含一个 Status-Line 和可选的头域，并且一一个空行结束（译注：空行就是 CRLF）。没有必须的头域对这类状态码。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。

客户端必须能准备去接受一个或多个 1xx 状态响应优先于一个常规响应，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。

代理服务器必须能转发 1xx 响应，除非代理服务器和它的客户端的连接关闭了，或者除非代理服务器自己响应请求并产生 1xx 响应。（例如：如果代理服务器添加了"Expect:100-continue"头域当转发请求时，那么它不必转发相应的 100（继续）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。服务器必须发送一个 final 响应在请求接收后。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，然后紧接着跟随一个结束此 101 响应的空行。

只有当协议切换时能受益，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，同步的协议带来好处时，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了，理解了，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，但是此资源最确定的 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，此头域为请求的变量（译注：变量的含义见第 1.3 节"变量"的解释）指明当前的实体标签（entity tag）值，当资源被创建时，见 14.19 节。

10.2.3 202 接受（Accepted）

请求已经被接受了，但是还没有对此请求处理完。请求可能处理完也可能没有最终被处理完，因为当处理发生的时候服务器可能会发现此请求不能被处理。

202 响应是非委托的。这是为了允许服务器为其他处理（如：每天执行一次的过程）而接受一个请求从而不需要用户代理和服务器之间在处理完成之前必须进行持久连接。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。

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

此状态码响应指明响应里的实体头域里的元信息对源服务器来说是没有意义的，这些元信息是从局部的或第三方的响应副本里得到的。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。203 响应状态码的使用并不是必须的并且只有在响应是非 200（Ok）响应时才是合适的。

10.2.5 204 无内容 （No Content）

服务器已经满足了请求但不需要返回一个实体，并且可能只想返回更新了的元信息。204 状态响应可能包含一个新的或更新了的，和请求变量（译注：变量是资源的一种表现形式，这和 rest 架构里的定义是一样的，所以这里我们可以理解请求变量是请求资源的一种表现形式）相联系的元信息（元信息以实体头域的形式表式）。

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，虽然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此文档。

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 Content-Range 头域，此头域指明了响应里的范围；或者一个值为 "multipart/byteranges" 的 Content-Type 头域和为每部分指明范围的 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，如果这些头域已经在以前相同请求的 200 响应里返回过。
- Expire，Cache-Control，和/或者 Vary 头域，如果域值与同一变量以前响应中的域值不一样。

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，那么响应不能包含其他的实体头域;这能防止出现在缓存的实体主体和更新的头域之间的不一致性。其他的情况下，响应必须包含所有的实体头域，这些头域可能已经在以前的相同请求的 200 响应里返回过。

缓存不能把 206 响应和以前的缓存内容结合如果 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回路，因为此回路能产生网络拥挤。

> 注：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）

资源对应有多个表现形式，客户端请求资源的表现形式对应于其中的一个，每个表现形式都有一个指向自己的位置(location)，并且代理驱动协商（agent-driven negotiation）能选择一个更适的表现形式并重定向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。依赖于此格式和用户代理的能力，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。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，否则此响应的头域应该包含对此新 URI 超文本链接的一个超文本提示。

如果客户端接收了一个 301 状态响应，此响应相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：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 用户代理将错误的把此请求变成一个 GET 请求。

10.3.3 302 发现(Found)

请求的资源放在一个不同的暂时的 URI 下。因为重定向可能会偶尔被改变，客户端应该继续利用请求 URI(Request-URI)为将来的请求。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，否则响应的实体应该包含指向此新 URI 的超文本链接的短超超文本提示。

如果 302 状态代码在一个不是 GET 和 HEAD 方法请求的响应中，用户代理不能自动重定

向请求除非用户确认可以自动重定向，因为这能改变请求提交的的条件。

注：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把 302 响应看成是 303 响应，根据 Location 头域值的 URI 执行 GET 请求，不管原始的请求方法。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允许 POST 执行脚本的输出重定向到用户选择的资源。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，303 响应的实体应该包含一个短的指向新的 URI 链接的超文本提示。

注：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 没有被改变(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,并且代理服务器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），缓存将会正确地操作。

- ETag 和/或 Content-Location,如果这些头域已经在相请求的 200 响应里出现过。

- Expires，Cache-Control，和/或 Vary，如果这些头域的域值可能与以前相同请求的响应的域值不一致。

如果条件 GET 用强缓存验证（见 13.3.3）节，那么响应不应该包含其他的实体头域。当条件 GET 用于弱缓存验证时，那么响应不能包含其他的实体头域；这能防止缓存的实体主体和更新了的头域之间的不一致性。

如果 304 响应指明实体不能被缓存，那么此缓存必须遗弃此响应，并且以无条件请求重复请求。

如果缓存利用一个 304 响应去更新一个缓存项,那么此缓存必须更新缓存项里关于响应里新的域值的那些域值。

10.3.6 305 使用代理服务器 (User Proxy)

请求资源必须能通过代理服务器访问，代理服务器的地址在响应的 Location 头域里指定。Location 头域指定了代理服务器的 URI。接收者被期望通过代理服务器重复此请求，

305 响应必须能被源服务器产生。

> 注：RFC 2068 并没有说明 305 响应的目的是重定向一个独立请求，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。由于重新向可能会偶尔改变，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。除非请求方法是 HEAD，否则响应的实体应该包含一个指向新 URI 的超文本链接提示，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重复原始请求到新的 URI 的必需的信息。

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它可以重定向，因为这可能改变请求提交的条件。

10.4 客户错误 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确保客户端回复包含在响应里的接收包，这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，

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。客户端不应该没有改变请求而重复请求。

10.4.2 401 未授权的（Unauthorized）

请求需要用户授权。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发。客户端会以一个 Authorization 头域重复此请求。如果请求包含了一个授权证书,如果服务器以 401 响应,它指明这些证书的授权被拒绝。如果 401 响应包含一个同样的授权激发和以前的响应一样,并且用户代理已经尝试至少授权了一次，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。授权是没有作用的并且请求不能被重复。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。条件是长期的还是暂时的也没有被服务器指明。410（Gone）状态响应应该被使用，如果服务器知道一个旧的资源不能长期的使用并且没有转发地址时。 此状态码通常别用于当服务器不希望指出为什么请求被拒绝，或者没有任何其他响应是适合的。

10.4.6 405 不被允许的方法（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，响应应该包含一个实体，此实体包含一系列实体的特性和位置，通过他们用户或用户代理能选择最合适的资源的表现形式。实体格式被媒体类型指定。依赖于此格式和用户代理的能力，选择最合适的会被自动执行。然而，此规范并没有定义自动执行选择的标准。

> 注：HTTP/1.1服务器被允许返回这样的响应，此响应根据接受头域（译注：如：Accept, Accept-Charset, Accept-Encoding, or Accept-Language）是不可接受的。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响应来决定是否此响应是可接受的。

如果响应是不可接受的,用户代理应该暂时停止更多的数据的接收并且询问用户去决定进一步的动作。

10.4.8 407 代理服务器授权所需（Proxy Authentication Required）

此状态码和401（Unauthorized）相似，但是指明客户端必须首先利用代理服务器对自己授权。代理服务器必须返回一个Proxy-Authenticate头域（见14.33节），此头域包含一个适用于代理服务器的授权激发。客户端可能会重复此请求利用一个合适的Proxy-Autorization头域（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。客户端端可能在以后会重复此请求。

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可能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，这是不可能的并且也是不需要的。

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提交的实体包含资源的改变，这些改变和以前的请求的改变想冲突，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，此响应实体应该包含这两个版本的不同，响应的格式在响应的 Content-Type 头域里指定。

10.4.11 410 不存在（gone）

请求资源不再可以在服务器上得到并且也不知道转发地址（译注：转向此资源的地址）。此条件是长期条件。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能方便判定条件是否是长期的，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 响应主要的目的是通知接收者资源不能被得到并且告诉接收者指向那个资源的连接已经被移动了。这样一个事件对时间要求比较严格的服务来说是比较普遍的，并且对属于个人但已经不在服务器站点工作的人的资源来说，这个事件来也是比较普遍的。把所有长期不能得到的资源标记成"gone"或保持这些标记任意长时间，这是没有必要的。

10.4.12 411 必需的长度（Length Required）

此响应服务器拒绝接受没有包含 Content-Length 头域的请求。客户端可以重复此请求如果它添加了一个有效的 Content-Length 头域，此头域包含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）

先决条件在一个或多个请求头域里指定，412 响应是表明先决条件在服务器上测试等于 false。此响应允许客户端把先决条件放到请求消息的元信息里（头域数据）。

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 头域用来指明此请求是暂时的并且什么时间后客户端应该重试。

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理解。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请求的资源支持。

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 值不和选择资源的当前的 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）。

当此状态码响应在一个 byte-range 请求后返回时，此响应应该包含一个 Content-Range 实体头域用来指定选择资源的当前长度（见 14.16 节）。此响应不能利用 multipart/byteranges 媒体类型。

10.4.18 417 期望失败

Expect 请求头域里指定的希望不能被服务器满足，或者，如果服务器是代理服务器，服务器有有不确定的理由确定请求不能被下一站的服务器满足。

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）

服务器不能支持满足请求的功能需求。这个响应是很合适的当服务器不能识别请求方法时并且不能支持它请求的资源的时候。

10.5.3 502 坏网关（Bad Gateway）

此响应说明：服务器，当作为网关或代理时，它从上行服务器接收了一个无效的响应并尝试满足客户端的请求。

10.5.4 503 难以获得的服务（Service Unavailable）

服务器不能处理请求由于服务器暂时的过载或维护。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：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）

服务器，当作为网关或代理服务器时，不能接收一个从被 URI 指定的上行服务器的响应（例如：HTTP，FTP，LDAP 服务器）或者它为完成请求而需要访问的一些其他的辅助性服务器（例如：DNS 服务器）。

注：一些部署的代理服务器将会返回 400 或 500 响应当 DNS 查找超时。

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的能被服务器支持的协议版本。

11. 入口验证（Access Authentication）

HTTP 提供一些可选的响应授权激发机制，这些机制能被用于服务器激发客户端请求并且使客户端授权。常用访问授权框架，还有"basic"和"digest"授权规范，都在"HTTP Authentication:basic and Digest Access Authentication"[43]规范里指定。HTTP/1.1 规范采用了"激发（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，希望提供给用户相应于请求最容易得到的实体。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容易得到的实体有喜好，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 提供了一些"内容协商"机制 —— 当有多个可得的表现形式的时候，对特定的响应选择最好的表现形式的处理过程。

> 注：没有称做"格式协商"（译注："格式"指的是"媒体类型"）的，因为可替换的表现形式可能会同原来的有相同的媒体类型，只是利用了此媒体类型不同的性质，例如一种不同的语言。

任何包含一个实体主体的响应包括错误响应都可能会受协商的支配。

有两种类型的内容协商在 HTTP 中：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并且能被单独使用或联合使用。一个联合使用方法的协商会被叫做透明协商，当缓存利用代理驱动协商的信息的时候，此代理驱动协商的信息被为后续请求提供服务器驱动协商的源服务器提供。

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关于请求的其他信息（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进行描述是比较困难的时候（译注：代理驱动协商），或者当服务器期望发送"最好的猜测"给客户端而只通过一个响应（以避免后续请求的回路（一个请求会返回一个响应）延迟如果此"最好的猜测"对用户适合的时候）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. 对服务器不可能确切的决定对用户来说什么是最好的，因为那需要对用户代理和用

户对此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. 使用户代理描述请求里的能力是非常无效的(假设只有响应的一小部分有多个表现形式）还有会侵犯用户的隐私。

3. 使源服务器的实现变得复杂，也对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。

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的支配。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，响应的最好表现形式的选择被用户代理执行，这在接收到源服务器一个初始的响应后。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，当响应可能会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。

代理驱动协商也同样存在需要第二次请求而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，所以不能防止任何这样的机制被用于 HTTP/1.1

HTTP/1.1 定义了 300（多个选择）和 406（不接受的）状态响应，当使用代理驱动协商时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的响应的是时候。

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。当一个缓存被提供了构成响应的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）并且维度的差异能完全被缓存理解,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去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。

此规范没有定义透明协商的机制，所以，它不能防止任何这样的机制被用于 HTTP/1.1。

13 HTTP 中的缓存

HTTP 典型应用于能通过采用缓存技术而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，因此有必要单独的来介绍基本的缓存设计。

如果缓存不能改善性能，他将一无用处。HTTP/1.1 中缓存的目的是为了在很多情况下减少发送请求，同时在许多情况下可以不需要发送完整响应。前者减少了网络回路（译注：一个请求会返回一个响应，请求响应这个过程就是一个回路）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。

对行为，可行性，和关闭的操作的要求放松了语义透明性的目的。HTTP/1.1 协议允许服务器，缓存，和客户端能显示地降低透明性当在必要的时候。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），因此此协议透明性在下面情况下才能被放松要求：

- 只有在一个显示的协议层的请求时，透明性才能被客户端或源服务器放松
- 当出现一个对终端用户的显示的警告时，透明性才能被缓存或被客户端放松

因此，HTTP/1.1 协议提供这些重要的元素：

1. 提供完整语义透明的协议特征，当这些特征被通信的所有方需要时
2. 允许源服务器或用户代理显示的请求和控制不透明操作的协议特征
3. 允许缓存给这样的响应绑定警告信息的协议特征,这些响应不能保留请求的语义透明的近似

一个基本的原则是客户端必须能够发现语义透明性的潜在的放松。

> 注：服务器，缓存，或者客户端的实现者可能会面对设计上的判断，而这些判断没有显示地在此规范里讨论。如果一个判断可能会影响语义透明性，那么实现者应该能维持语义透明性，除非一个仔细的完整的分析能说明打破透明性的好处。

13.1.1 缓存正确性（Cache Correctness）

一个正确的缓存必须能用最新的响应来响应请求,此响应当在下面的条件满足时才适合此请求（见 13.2.5，13.2.6，和 13.12）

此缓存响应已被检测与假设通过源服务器重验证后源服务器返回的响应相等价。

此缓存响应是足够保鲜(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，和缓存的最严格的保鲜要求（见 14.9 节）；如果源服务器指定了保鲜寿命，这说明它是源服务器自己的保鲜要求。

由于客户端和源服务器的最严格的保鲜要求，如果一个缓存的响应不是足够保鲜的，那么在仔细考虑的情况下，缓存可能仍然返回此缓存的响应通过合适的 Warning 头域（见 13.1.5 和 14.46 节），除非此响应被阻止（例如：通过"no-store" cache-directive ，或者通过一个"no-cache"cache-request-directive；见 14.9 节）。

此缓存响应是一个 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，那么一个正确的缓存应该用它缓存的响应去响应请求，如果此缓存的响应能正确的服务于请求；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警告指示存在通信失败。

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（没有被

改变）的响应），此响应应该是正常情况下要发送到请求的客户端的，并且此响应并不是新鲜的，那么此缓存应该把此响应转发给请求客户端不需要添加一个新的 Warning 头域（但是没有移去已经存在的 Warning 头域）。缓存并不是简单的因为传输中响应变得陈旧而尝试去重验证响应；这可能会导致一个无限的循环。用户代理接收一个陈旧的没有 Warning 头域的响应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效果。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。

警告可能被用于其他的目的，和缓存相关的目的和其他的目的。警告和错误状态码的区别在于是否是真正的失败。

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

- 1xx 警告描述了响应的保鲜或重验证状态信息，并且这些信息应该在一个成功的重验证之后删除。1xx 警告码可能是由缓存产生的，当缓存验证一个缓存项时。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，并且这些信息不能在一个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。穿过 HTTP/1.0 缓存响应中的警告会携带一个额外的 warning-date 域，这是为了防止将来的 HTTP/1.1 接收端信任一个错误的缓存警告。

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（或者被源服务器或者被一个缓存发送的），这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 版本不能指定一个严格的优先值去决定警告的优先和顺序显示，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 基本的缓存机制（服务器指定过期时间和验证器）对缓存是隐含的指令。某些情况下，服务器或客户端可能需要给 HTTP 缓存提供显示的指令。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，那么最具严格解释的头域值会被应用（也就是说，能保留语义透明性的值）。然而，一些情况下，cache-control 指令被显示地指定用来削弱语义透明性的相似性（例如，"max-stale" 或者 "public"）。

Cache-control 指令在 14.9 节被描述。

13.1.4 显示的用户代理警告（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）从来不要被验证。或者，用户代理可能习惯于给任何请求加上"Cache-Control: max-stale=3600"。用户代理不能缺省的执行不透明行为，或者不能缺省的执行导致非正常的无效率的缓存行为，但是可能被显示地设置去这样做通过一个显示的用户动作。

如果用户已经覆盖基本的缓存机制，那么用户代理应该给用户指示何时显示不能满足服务器透明要求的信息（特别地，如果显示的实体被认为是陈旧的）。因为此协议通常允许用户代理去判定响应是否是陈旧或不是陈旧的，所以此指示只需要当实际发生时显示。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。

如果用户以一种方式已经覆盖了缓存机制，这种方式可能不正常地减少缓存效率，那么用户代理应该继续指示用户的状态（例如，通过一个图片显示）以便用户不能不注意地消费过度的资源或者忍受过度的延迟。

13.1.5 规则和警告的例外情况

在某些情况下，缓存的操作者应该设置缓存返回陈旧的响应，即使此响应没有被客户端请求。这个判定本来不应该轻易决定，但是由于某些原因可能会这样做，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），因为这样能使客户端提示用户响应是陈旧的。

用户代理照样能采取步骤去获得第一手的或保鲜的响应。由于这个原因，如果客户端显示地请求第一手的或保鲜的响应，缓存就不能返回一个陈旧的响应，除非由于技术或策略方面的原因。

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

当源服务器是过期信息得主要来源时，有时候客户端可能需要控制缓存去判定是否返回一个缓存响应而不需要缓存通过服务器验证它。客户端通过利用一些 Cache-Control 头域来达到此目的。

客户端的请求可能会指定自己愿意接受一个没有验证的响应的最大的年龄（age）；指定 0 值会强迫缓存重验证所有的响应。客户端照样会指定在响应过期前的最小保持时间。这些选项增加了对缓存行为的限制，同时这样做并不能进一步地放松缓存语义透明的近似。

客户端可能照样会指定它会接受陈旧响应直到陈旧数达到最大数量。这放松了对缓存的限制，同时这可能违反了源服务器指定对语义透明性的限制，但是这可能支持无连接的操作或者高获得性当连接不好时。

13.2 过期模型（Expiration Model）

13.2.1 服务器指定模型（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。避免对源服务器请求的主要机制是服务器将来会提供一个显示过期时间（explicit expiration time），此显示过期时间用来指示响应可能会满足后续的请求。也就是说，客户端请求响应时，缓存能返回一个保鲜（fresh）的响应而不需要从源服务器那里获

得。

我们希望服务器给响应设置显示过期时间（explicit expiration time），服务器相信在过期时间之前实体不会改变。这能保持语义透明性（译注：语义透明性情况 1.3 节里关于"语义透明"的解释），只要服务器对过期时间仔细选择。

过期机制只能应用于能从缓存获得的响应，不能应用于客户端请求的第一手（first-hand，见 1.3 节术语）的响应。

如果源服务器希望强制一个语义透明缓存去验证每个请求，它会使显示过期时间（explicit expiration time）设为过去。这就是说响应总是陈旧的，所以此缓存应该验证此响应当缓存利用此响应去服务于后续的请求时。见 14.9.4 节，有更多强迫重验证的方法

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，源服务器应该在 Cache-Control 头域里指定"must-revalidate"缓存控制指令（见 14.9 节）。

源服务器利用 Expires 头域或在 Cache-Control 头域里通过 max-age 缓存控制指令，来指定显示过期时间（explicit expiration time）。

过期时间不能被用于强制客户代理去重新刷新它的显示或重载资源；过期的语义只能应用于缓存机制，并且这个机制值只需要检测资源的过期状态当请求那个资源的一个新请求发生时。见 13.13 节，关于缓存和历史机制的区别。

13.2.2 启发式过期

因为源服务器不能总是提供一个显示过期时间（explicit expiration time），HTTP 缓存通常会设置一个启发式过期时间（heuristic expiration time），它采用一种算法，此算法利用其它的值（例如 Last-Modified 时间）去估计一个似乎合理的过期时间。HTTP/1.1 规范没有提供特定的算法，但是的确是加强了对这些算法结果的最坏情况的限制。因为启发式过期时间可能会对语义透明性妥协，他们本应该被谨慎地使用，并且我们鼓励源服务器尽可能提供显示过期时间。

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。

在此讨论中我们用"now"来表示主机进行计算时时钟的"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] 或其他类似协议来将其时钟同步到一个全球性的精确时间标准上。

HTTP1.1 协议要求源服务器尽可能在发送每条响应时都附加一个 Date 头域，要尽可能在每个响应里给出响应产生的时间（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 响应头域来传送响应消息的估计年龄。Age 响应头域值是缓存对响应从产生或被重验证开始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)可以通过两种完全不同的途径来计算::

1. 如果本地时钟与源服务器时钟同步的相当好，则用 now－date_value ，若结果为负，则取零。
2. 如果途径响应路径（response path）的所有缓存均遵循 HTTP1.1 协议，则用 age_value。

如果我们有两种独立的方法计算响应的年龄（译注：注意这里是响应的年龄），我们可以合并二者如下:

corrected_received_age = max（now - date_value，age_value）

并且只要我们或者有同步的时钟或者响应途径的所有缓存遵循 HTTP/1.1，我们就能得到一个可信赖的结果。

由于网络附加延时,一些重要时隙会在服务器产生响应与下一个缓存或客户端接收之间流逝。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（译注：就是存在 Age 头域的请求）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，此算法会导致稳定的结果。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。

缓存收到响应时，它计算响应年龄的算法如下:

```
/*
 * 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);

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;

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开始到现在的时间间隔（以秒记）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)。缓存器必须对年龄大于24小时的响应附加113警告，如果此响应不带这种警告。

而且，如果响应有最后修改时间，启发式过期值应不大于从那个时间开始间隔的某个分数。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）

由于过期值很容易被设置，有可能两个缓存会包含同一资源的不同保鲜值。

如果客户端执行请求接收到一个非第一手的响应,此响应在此客户端拥有的缓存里仍然是保鲜的,并且缓存里的缓存项里的 Date 头域的值比此新响应的 Date 头域值要新,那么客户端应该忽略此响应。如果这样,它可能会重新以"Cache-Control: max-age=0"指令(见 14.9 节)请求去强制任何中间缓存通过源服务器对其进行检查。

如果缓存对有不同验证器(validator)的同一个表现形式(representation)有两个保鲜响应,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从其他缓存得到响应,或者由于客户端请求对一个保鲜缓存项的重载或重验证(revalidation)。

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,所以有些响应会经过一些缓存,而其他的响应会经过其他的缓存,客户端收到响应的顺序可能与源服务器发响应的顺序不同。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)和过期值都不能影响响应的顺序,因为可能会出现晚一点的响应可能会故意携带过早的过期时间。日期值的精度被规定只有一秒。

当客户端试着重新验证一个缓存项的时候(译注:这里的客户端应该包含一个本地缓存),而且接收到的响应的 Date 头域晚于已存在的缓存项,那么客户端应该重新进行无条件请求,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)这些中间缓存的副本,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)

当缓存有一个旧缓存项并且缓存想利用此缓存项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)对其进行检验看看此缓存项是否可用。我们称做"验证(validating)"此缓存项。因为我们不想对完整响应的传输付出太大代价,而且如果缓存项无效时也不会产生额外的回路(译注:回路的意思,如:从请求到响应就一条回路),HTTP1.1 协议支持使用条件方法。

协议支持条件方法的关键特征是围绕"缓存验证器(cache validator)"展开的。当源服务器产成一个完整响应时,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(用户代理或缓存服务器)对对应有缓存项的资源执行条件请求时,客户端包含一个相关的验证器(validator)在请求里。

服务器(译注:服务器可能是缓存服务器)则核对请求里的验证器和当前本地的验证器是否匹配,如果他们匹配(见 13.3.3),则返回一个特定状态码(通常为 304(没有改变))的响应并且此响应不包含实体主体(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,我们就避免了传输完整响应如果验证器匹配,同时我们也避免了额外的回路如果验证器不匹配。

在 HTTP1.1 协议中,一个条件请求和普通的请求差不多,除了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。

协议中包括缓存验证条件的正和负。也就是说请求方法将会执行如果验证器的匹配或不会执行如果验证器不匹配。

> 注：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的副本过期，除非显示地用缓存控制指令来禁止缓存这样做。然而，缓存不能执行条件方法获取资源如果它没有此资源实体的验证器，那意味着缓存副本将会不可更新在它过期后。

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，缓存项被认为是有效的如果实体自从 Last-Modifed 值之后没有改变。

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"不透明（opaque）的缓存验证器。这能得到更可靠的验证当在不方便存放修改日期的情况下，当在 HTTP 日期值的一秒精度不能满足需要的情况下，或当在源服务器希望避免使用修改日期产生的冲突的情况下。

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（entity，实体主体或实体头域）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，能被当作弱验证器，因为在一秒内资源可能改变两次。是否支持弱验证器是选择的。然而，弱验证器可以能高效的去缓存等效对象。

客户端产生请求并把验证器包含在一个验证头域里的时候或在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，因为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 请求里既可以包含弱验证器也可以包含强验证器。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。有两种验证器的比较方法，这依赖于是否能利用弱验证器进行比较：

- 强比较方法：在考虑相等的情况下，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：在考虑相等的情况下，两验证器必须完全一致，但至少有一个验证器可能在不影响结果的情况下被标明为"weak"。

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

最后修改时间（译注：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：

- 验证器被源服务器用来和当前实体的验证器进行比较，并且
- 源服务器知道相关的实体不会在当前验证器函盖的秒数内改变两次。

或者

- 验证器可能被客户端用于 If-Modified-Since 或者 If-Unmodified-Since 头域里，因为客户端有一个关于此实体的缓存项，并且
- 缓存项包含一个日期值，此日期值给出了源服务器发送源响应（译注：源服务器产生的响应）的时间，并且
- Last-Modifed 头域值至少提前于日期值（Date 头域值）60 秒。

或者

- 验证器已经被中间缓存同此实体的缓存项里的验证器比较过 ，并且
- 缓存项包含日期值（Date 头域值），此日期值指明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 头域值）60 秒。

此种方法依赖于以下事实，如果两个响应被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（Last-Modified）时间，那么至少有一个响应的日期值（译注：Date 头域的值）和最后修改时间值（Last-Modifed 头域的值）是相等的。60 秒的限制能保证 Date 和 Last-Modifed 头域的值在不同时刻产生。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)时间和而没有透明验证器的值时，它可能会认为只有最后修改（Last-Modified）时间是强的。

若缓存或源服务器收到一个条件请求，而不是得到完整响应的 GET 请求时，他必须使用强比较方法去计算此条件。

此规定允许 HTTP1.1 的,缓存和客户端安全地执行子范围（sub-range）请求来请求从 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应该被使用，出于何种目的被使用。

HTTP/1.1 源服务器：

- 应该发送一个实体标签验证器除非源服务器产生这样一个实体标签不可行。

- 可能会发送弱实体标签而不是强实体标签,如果使用弱实体标签能提高性能的话或者如果发送一个强实体标签不可行的情况下。
- 应该发送一个 Last-Modifed 值如果可行的话，除非打破语义透明性（这可能由利用 If-Modified-Since 头域里的日期产生）可能会导致严重的后果。

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：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复利用一个特定的强实体标签值。缓存项应该能保持任意长的时间，而不管过期时间（expiraton time），所以缓存可能会再去尝试去利用在过去某一时刻获得的验证器去验证缓存项。

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（利用了 If-Match 或 If-None-Match 的请求）里利用实体标签。
- 仅当 Last-Modified 值被源服务器提供时，HTTP/1.1 客户端应该在非子范围缓存条件请求（利用 If-Modified-Since）里利用此值。
- 仅当 Last-Modified 值被 HTTP/1.0 源服务器提供，HTTP/1.1 客户端可能会在子范围缓存条件请求（利用了 If-Unmodified-Since）里利用此值。
- 如果一个实体标签和 Last-Modified 值被源服务器提高，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/1.1 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且此请求同时包含 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（Not Modified）除非这样做能当当前实体的验证器和请求里所有的条件头域里的验证器一致。

HTTP/1.1 缓存服务器，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的副本给客户端除非那个副本的验证器和所有请求里条件头域里的验证器一致。

> 注：这些规则背的常用的原则是 HTTP/1.1 服务器和客户端应该在请求和响应里尽可能传输非冗余的信息。接收这些非冗余信息的 HTTP/1.1 系统将会假设它接收了验证器。
>
> HTTP/1.0 客户端和缓存将会忽略实体标签。通常，Last-Modified 值被这些系统接收后将会保证透明和高效的缓存行为，所以 HTTP/1.1 源服务器这时将会提供 Last-Modified 值。在这些情况下，当 HTTP/1.0 系统利用一个 Last-Modified 值作为一个验证器可能会带来严重的后果时，HTTP/1.1 服务器将不会提供 Last-Modified 值。

13.3.5 非验证条件（Non-validating ConditionIs）

实体标签背后的原则是只有服务的作者才知道资源的语义而去选择一个合适的缓存验

证机制，并且任何验证器比较方法的标准都可能会带来风险。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的响应作为缓存项，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。如果既没有和响应相关的缓存验证器也没有和响应相关的显示过期时间（explicit expiration time，译注：见 13.3.1 节里关于什么是显示过期时间的说明），我们不会认为它是可缓存的，但是某些缓存可能会违反这个期望（例如，当不能进行网络连接时）。客户端能经常发现从缓存里获得的响应，只需要通过把 Date 头域值同当前时间作比较。

注：某些 HTTP1.0 缓存可能违反这一期望而不提示警告。

还有,某些情况下可能不便保留一实体,或将其返回给后续请求.

然而，在一些情况下，缓存不适合保存一个实体，或者不适合把它放于后续请求的响应里。这可能因为服务作者认为完全语义透明性是有必要的,或着因为安全和隐私的考虑。某些缓存控制指令是为了让服务器能指明某些资源实体或其中的一部分不能被缓存。

注意在 14.8 节里描述了防止一个共享缓存去保存和返回一个以前请求的响应，如果那个请求包含一个 Authorization 头域。

除非缓存控制指令防止此响应被缓存，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，但这必须受限于过期机制（expiration mechanism）。然而，缓存如果不支持 Range 和 Content-Range 头域，那么它不能缓存 206 响应（部分内容）响应。

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明确地允许它能这样做。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

缓存的目的是存储请求的响应信息，为了响应将来的请求。在很多情况下，缓存能返回响应的合适部分给请求者。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服务器的行为，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，并且必须在从缓存项形成的响应里传输。
- hop-by-hop 头域，他们被只对传输层上的连接有意义，并且不能被缓存保存或被代理转发。

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

所有其他被 HTTP/1.1 定义的头域均为 end-to-end 头域。

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，基于某一些 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它这样做。

一个透明代理（译注：代理一般是缓存，缓存可以叫缓存代理，缓存服务器，代理服务器）不能改变请求或响应里的下面的头域，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下面的的头域：

- Expires

但它可以添加这些头域如果响应里没有这些头域的时候。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（译注：缓存就是一个代理，我们一般称做缓存代理，或缓存服务器，或缓存代理服务器，都是一个意思）不能在消息中改变或添加下面的头域如果此消息包含 no-transform 缓存控制指令，或在任何请求里也不能添加或改变这些头域。

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

一个非透明代理可能会改变或添加这些头域给一个消息如果此消息不包含 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）。

> 警告：end-to-end 头域的不必要的改变可能会导致认证机的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出现的头域值。

请求或响应里的 Content-Length 头域被添加或被删除会根据 4.4 节的规则。一个透明代理必须保留实体主体的 entity-length（见 7.2.2），尽管它可以可能改变

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，那么缓存可能把存入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）。

存储于缓存项里端 end-to-end 头域被用于构造响应，除了：

- 任何存储的警告码是 1xx（见 14.46）Warning 头域必须从缓存项和转发的响应里删除。
- 任何存储的警告码是 2xx 的 Warning 头域必须要在缓存项和转发的响应里保留。
- 任何在 304 或 206 响应里的 end-to-end 头域必须替换缓存项里相应的头域

除非缓存决定去删除缓存项，否则它必须照样能用接收的响应里的相应的 end-to-end 头域去替换存储在缓存项里的头域，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。

从另一方面说，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：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）响应去更新任何同一实体或实体的子范围的以前响应的头域，虽然它可能没有意义或这样做不正确。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）携带了另一个子范围，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两者遵循下面的规则：

- 输入响应和缓存项都有缓存验证器。
- 当利用强比较方法的时候，两个缓存验证器完全匹配（见 13.3.3）。

如果任何一个要求不能满足，缓存必须利用最近的部分响应（这基于任何响应的 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），而且必须丢弃其他的部分信息。

13.6 缓存已经协商过的响应（Caching Negotiated Responses）

在 Vary 头域出现在响应里的时候，服务器驱动内容协商（12.1 节）的使用会改变缓存

利用响应去响应后续请求利用的条件和过程。见 14.44 节关于服务器利用 Vary 头域的描述。

服务器应该利用 Vary 头域去通知一个缓存什么样的请求头域应该被使用从而从一个可缓存的并受限于服务器驱动协商的响应的多个表现形式中选择合适的表现形式。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的请求，此请求的 URI 对应一个或多个包含了 Vary 头域的缓存项时，此缓存不能利用这样一个缓存项去构造出一个响应去服务于新来的请求，除非所有出现在新请求里的选择请求头域匹配存储在源请求里被存储的请求头域。

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，并且后续那个资源的请求只能合适地被源服务器解析。

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一个条件请求把此新请求接力到源服务器并且此源服务器以一个 304（没有改变）的状态码进行响应并包含一个实体标签或者 Content-Location 头域指明将要被使用的实体。

如果一个实体标签被赋予一个缓存的表现形式，那么此转发的请求将是条件的并且所有那个资源的缓存项的实体标签将会被包含于 If-None-Match 头域里。这向服务器表达当前缓存拥有的实体集，以至于如果这些实体里的任何实体匹配请求的实体，服务器会利用 Etag 头域在 304（没有改变）响应里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经存在的缓存项，那么新响应应该被利用去更新已经存在的缓存项的头域，而且此结果必须返回给客户端。

如果任何已经存在的缓存项包含只有相关实体的部分内容，那么此实体的实体标签不应该被包含在 If-None-Match 头域里除非这个请求是为了请求实体的一个范围，此范围完全可以被缓存项满足。

如果缓存接收到一个成功的响应，此响应的 Content-Location 头域匹配于已经存在的缓存项的 Content-Location 头域对同一请求 URI 来说，并且此响应的实体标签不同于已经存在的缓存项的实体标签，而且此响应的 Date 头域值比已经存在的缓存项更近，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使用的缓存，此种情况下，可用性由适当的安全机制控制。所有其它的缓存均被认为是共享缓存。此协议的其它部分给一些对共享缓存的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）也可以存储它，但是必须把它看作部分响应。部分相应可以合并（见 13.5.4）；合并结果可能是完整的响应或可能仍是部分的响应。.缓存不能把部分响应返回给客户端除非有明确要求可以这样做例如利用 206（部分内容）状态码响应。缓存不能使用一个 200（OK）状态码返回一个部分响应。

如果缓存在试图重验证一个缓存项而收到一个5xx响应时,它既可以将此响应转发给请求的客户端,或者做的就像服务器不能响应似的。在后面的情况下,它可能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）

除非源服务器明确地禁止缓存保存它们的响应，对任何资源应用的GET和HEAD方法不应该有副作用,这些副作用会导致错误的行为如果这些响应将从缓存产生。他们可能会仍然有副作用,但缓存在决定缓存时不必考虑这些副作用。缓存总是期望去观察一个源服务器对缓存的明确限制。

一个例外：有些应用习惯于在在GETs和HEADs方法里使用查询URLs(在rel_path_part里包含一个"?")去执行一个操作而带来很大的副作用,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示过期时间（explicit expiration time)。这意味着此URIs的以前从HTTP/1.0服务器产生的响应不能来自于缓存。见9.1.1节相关的信息。

13.10 在更新或删除后的无效性

对源服务器上的某资源执行方法的副作用可能会使一个或多个已经存在的缓存项的不透明性无效。那就是说，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将会对这一个新的请求返回什么。

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，一些规则帮助减少错误行为的可能。

在此节里，短语"使实体无效"意味着缓存会将所有那个实体的实例从它的存储里移除,或者把这些实体的实例标记为"无效的"并且在他们可能作为后续请求的响应时会被进行重验证。

一些HTTP方法必须让缓存去使一个实体无效 。这些实体被请求URI指定，或在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的无效性处理必须只有在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给源服务器（译注：直接穿过缓存在服务器上修改)。这通常包括所有方法除了GET和HEAD方法。缓存在将此种请求转发给服务器并获得相应的响应前不能对请求客户端做出响应。这个不能防止代理缓存(译注：也可以叫缓存服务器，缓存）在服务器已经发送最终回复之前发送100（继续）响应。

相反情况（通常叫"写回"或"拷贝回"缓存）在HTTP1.1中是不允许的，这是由于提供一致更新非常困难，并且服务器，缓存和网络的故障会出现在比写回早。

13.12 缓存替换 （Cache Replacement）

如果一个新的可缓存的响应被缓存接收，同时对这一资源的响应已经在缓存里存在，那么缓存应该利用最新的响应去回复当前的请求。缓存可能会把此新响应放进存储里，并且如果它满足所有其他的要求，缓存将会利用此响应来响应任何将来的请求。如果缓存想把此新的响应加进缓存的存储，13.5.3 的规则必须应用。

> 注：一个新响应如果它的 Date 头域值比已经存在的缓存响应的 Date 头域值老，那么它是不能缓存的。

13.13 历史列表（History Lists）

用户代理经常使用历史机制，如"Back"按钮和历史列表，来重新展示在一个会话里接收的一个稍早的实体。

历史机制和缓存机制是不同。历史机制不应该尝试展示一个当前资源状态的语义透明视图。其历史机制只是为了展示在获得资源时看到了什么。

默认情况，一个过期时间不会应用于一个历史机制。如果实体仍然在存储里，历史机制应该显示它即使实体过期了，除非用户叫用户代理去刷新过期的文档。

这不能被解释去防止历史机制告诉用户视图可能过期了。

> 注：如果历史机制没必要地防止了用户看陈旧的资源，那么这会强制服务作者去避免利用 HTTP 过期控制和缓存控制。服务作者可能会认为这是非常重要的当用户没有被呈现错误消息或警告消息当他们利用导向按钮（如 BACK 按钮）去看以前获得的资源时。即使有时这些资源本不能被缓存保存或应该可能会很快过期，用户界面可能会强制服务作者去求助于其他防止缓存的方法（例如，一次性 URLs）为了避免历史机制功能的不合适的作用。

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，发送者和接收者指的是客户端和服务器，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服务器返回给客户端可接受的响应媒体类型。Accept 头域能被用于指明请求是期望服务器返回某些期望的媒体类型的响应，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/*"指明了 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。第一个"q"参数(如果有的话)把 accept-params 和 media-range 参数分离了。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：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。虽然这防止里任何媒体类型参数以"q"命名并使用于 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 头域里利用任何媒体类型参数也是很少的。将来的媒体类型不鼓励任何以"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最容易得的 audio 类型，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的 Accept 头域值判定它不能发送可接受的（acceptable）的响应，那么服务器应该发送 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 能被更具特指的 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，那么最特指的引用应该优先。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：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是被用户可设置的。

14.2 Accept-Charset

Accept-Charset 请求头域可以用来指出请求客户端能接受什么样的字符集响应。这个头域允许客户端能通知服务器指定何种此客户端更能理解的或更具特殊目的的字符集的响应。

```
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想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
```

特殊的"*"值如果出现在 Accept-Charset 头域里，那么将匹配任何字符集（character set）（包含 ISO-8859-1）。如果没有"*"出现在 Accept-Charset 头域里，那么所有出现的字符集的质量值为 0，除了 ISO-8859-1，它的质量值为 1 如果没有出现在 Accept-Charset 头域里。

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服务器不能发送客户端想要的 Accept-Charset 里指定的字符集的响应，那么服务器将发送一个 406（不能接受的）错误响应，然而发送一个不能不能让客户端接受的字符集的响应也是允许的。

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

服务器测试一个内容编码(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:

- 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")
- 如果"*"字符出现在 Accept-Encoding 头域里,那么它匹配任何没有出现在 Accept-Encoding 头域的可得的内容编码。
- 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。
- "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,那么服务器将利用"identity"内容编码,除非服务器有另外的信息指明其他的内容编码对客户端是有意义的。

> 注:如果请求不能包含一个 Accept-Encoding 头域,并且如果"identity"内容编码是不能接受的,那么通常不能被 HTTP/1.0 客户端理解的内容编码(也就是说,"gzip"和"compress")通常是最喜爱的;一些老的客户端有时候会不合适的显示以其他内容编码的消息。服务器应该照样能基于特定的用户代理或服务器的信息做除决定用何种内容编码响应。

> 注: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 将不能工作并且在 x-gzip 或 x-compress 里不能被允许。

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(译注:如上面例子的 da,en-gb,en）如果它能精确和此语言标签相等，或者它能精确匹配此标签的前缀（在"-"以前的部分)。特殊的"*"，如果出现在 Accept-Language 头域里，表明能匹配任何不在此头域里的任何标签。

注：前缀匹配规则意味着用户能理解匹配此前缀的所有语言。

一个语言标签的质量因子是 Accept-Language 头域里最能匹配此语言标签的 language-range 的质量值。如果 Accept-Language 头域里没有 language-range 匹配此语言标签，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可接受的。如果一个 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。

由于个别的用户的理解程度不一样,建议客户端应用程序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：当让用户作出选择时，我们要实现者一个事实那就是客户不熟悉语言的细节，并且不会提供一个合适的指导。例如，用户可能会认为当选择了"en-gb"会提供任何类型的英语文档即使 British English 是不可得的。一个用户代理应该能建议去增加一个"en"去得到最合适的匹配行为。

14.5 Accept-Range

Accept-Range 响应头域允许服务器指明它对客户的范围请求（range request，译注：当在请求消息里出现 Range 头域时表明此请求是范围请求）的接受程度。

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

但是没有必要这样做。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

去告诉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（译注：一般是缓存）对响应产生（或重验证）时刻后经过时间的估计。一个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 HTTP/1.1 服务器如果包含一个缓存，那么它必须包含一个 Age 头域在它拥有缓存产生的任何响应里。缓存应该利用一个至少 31 位的运算类型。

14.7 Allow

Allow 实体头域中列出了请求 URI（Request-URI）指定资源所支持的几种方法。此头域的目的是严格地让接收端知道资源所适合的方法。Allow 头域必须出现在 405（方法不被允许）响应中。

```
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。然而只有在 Allow 头域域中给出的方法才应该被执行。Allow 头域里指定的方法被源服务器定义，并且在每次请求的响应里都应该出现这些方法。

Allow 头域里可以被提供在一个 PUT 请求里，这是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，而且应该包含一个 Allow 头域在响应里并且给出实际支持的方法。

代理服务器不能改变 Allow 头域即便不理解此头域里指明的所有方法，因为用户代理可能和源服务器通信有其他的目的。

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 响应后就没有必要让服务器给自己授权了。Authorization 头域由包含用户代理对那个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，那么这个证书应该对此域里所有的其他请求是有效的（假设在此授权模式下，证书不会根据一个激发值或利用一个同步的时钟而变化）。

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，那么它不能返回此请求相应的响应来回复任何其他的请求，除非下面指定的异常发生：

如果此响应包含"s-maxage"缓存控制指令，那么此缓存可能会利用那个响应来响应一个后续的请求。但是（如果指定的 age 过期了）一个代理缓存必须首先通过源服务器来重

验证此响应,利用新请求里的 Authorization 请求头域去让源服务器授权此新请求。(这是为"s-maxage=0"定义的行为。) 如果响应包含"s-maxmax=0", 那么代理必须总是重验证此响应在重利用它之前。

如果此响应包含"must-revalidate"缓存控制指令, 那么缓存可能会利用那个响应来响应一个后续的请求。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,利用新请求里的 Authorization 请求头域去让源服务器去授权此新请求。

如果响应包含共有缓存指令, 那么它可能会响应任何后续的请求。

14.9 Cache-Control

Cache-Control 常用头域被用于指定指令, 此指令必须被在请求/响应链上的所有缓存机制遵守。这些指令指定防止缓存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响应中也会有同样的指令。

> 请注意 HTTP/1.0 缓存可能并不实现 Cache-Control,而是只实现 Pragma: no-cache (参见 14.31 节)。

代理或网关应用程序必须让缓存指令通过不管这些指令对自己的影响,因为这些指令可能对请求/响应链上的所有接收者都适用。不可能为一个特定的缓存指定一个缓存指令。

```
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。当这样一个指令伴有一个 1#field-name 参数时，它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了响应为可缓存的，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

指明响应响应可以被任何缓存保存,即便该响应通常是不可缓存的或该响应只被一个非共享缓存保存的情况下。(参见 14.8 节,关于 Authorization 头域的更多详述。)

private

表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。这使源服务器可以申明响应的特定部分是针对一个用户,并且对其他用户的请求而言不是有效的响应。一个私有（非共享）缓存可以缓存此响应。

注: 单词 private 的使用仅用来控制响应在何处可被缓存,而不能保证消息内容的私有性。

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的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,即使此缓存已经被设置能返回给客户端请求一个陈旧的响应。

若 no-cache 缓存控制指令指定一个或多个 field-name，那么一个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任何其它限制。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的情况下。这允许源服务器能防止缓存重利用响应里的某些头域，但仍然可以允许缓存响应的剩余部分。

注：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无意的泄露或保留了敏感信息（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，一个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，一个缓存不能保存此响应或引起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"在这个上下文里（context，译注：也可以叫背景）的意思是指缓存不能有意的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,用户也可能会显示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。历史缓存（译注：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需求，还有就是给那些对偶然信息发布（这通过未预料地对缓存数据组织的访问）比较在意的作者也提供方便。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是能充分地保护隐私。特别是，恶毒得或者有损害的缓存可能不能识别到或遵循此指令，并且网络通信随时也容易受到窃听。

14.9.3 对基本过期机制的改进

实体的过期时间（译注：这里的过期时间也就是"显示过期时间"，见 13.3.1 里关于"显示过期时间"的说明）可由源服务器利用"Expires"头域（参见 14.21 节）指定。 或者，它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缓存的响应里的时候,如果此缓存的响应当前年龄（current age，译注:见 13.2.3 节关于 current age 的定义）比假如一个新的请求此时去请求那个资源而得到响应里的 age 值（以秒为单位）要大，那么此缓存的响应就是陈旧的（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"）除非更多其他的具有限制性的缓存控制指令出现在响应里。

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对一个给定的响应提供一个更长的过期时间给 HTTP/1.1 缓存（或以后的版本），这对比 HTTP/1.0 缓存来说。这个规则可能会很有用，如果某个 HTTP/1.0 缓存不合适地计算了年龄或过期时间由于不同步的时钟。

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"缓存响应控制指令（Cache-Control response directive）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，它应该把此响应看成一个不能缓存的，这是为了保持和 HTTP/1.0 服务器兼容。

> 注：一个源服务器可能希望把一个相对新的 HTTP 缓存控制特性，例如"private"缓存控制指令，用在一个存有旧版本缓存的网络上，而且此旧版本缓存不能理解此特性。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，这样以便防止旧版本的缓存不合适的保存此响应。

s-maxage

> 如果一个响应包含一个 s-maxage 缓存控制指令，那么对于一个共享缓存（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，此共享缓存不能利用此缓存项在它变旧后，在没有通过源服务器首先重验证它的情况下，去响应后续的请求。s-maxage 缓存控制指令总是被私有缓存忽略。

大多数不遵循此规范的老版本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个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

> 表明客户端愿接受一个这样一个响应，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）（译注：如果大于的话，表明此响应是陈旧的）。除非 max-stale 缓存控制指令也包含在响应里，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，其保鲜寿命不小于其响应当前年龄(译注：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。 若客户端请求里为 max-age 指定了一个值，则表明客户端愿接受过期时间不超过它在 max-stale 里指定秒数的响应。若 max-age 未指定一个值，则客户端愿接受任意年龄的陈旧响应。

如果缓存返回了一个陈旧响应，这是由于 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,那么此缓存必须把一个 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"必须"等级关于缓存验证（例如，一个"must-revalidate"缓存控制指令）的要求冲突。

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去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，坚持想让一个缓存通过源服务器去重验证它的缓存项，或者从源服务器那里重新加载它的缓存项。end-to-end 重验证可能会有必要的，如果缓存或源服务器已经估计了缓存响应（cached response）的过期时间。end-to-end 重载可能是有必要的，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，在这种情况下，我们称它为"没指定的 end-to-end 重验证（unspecified end-to-end revalidation）"；当客户端有本地缓存副本,在这种情况下,我们称它为"指定的 end-to-end 重验证(specific end-to-end revalidation）"。

客户端能指定下面三种动作，利用缓存控制请求指令（Cache-Control request directives）：

end-to-end reload

> 请求包括"no-cache"缓存控制指令，或为了与 HTTP/1.0 客户端兼容的"Pragma: no-cache"缓存控制指令。头域名不能被包含在 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

> 这样的请求包含一个"max-age=0"缓存控制指令，它强制每个途径源服务器的缓存去通过下一个缓存或服务器来重验证它所拥有的缓存项（如果有的话）。此初始请求包含一个带有客户端当前验证器的缓存验证条件。

unspecified end-to-end revalidation

> 这样的请求包含一个"max-age=0"缓存控制指令，它强制每个途径源服务器的缓存去通过下一个缓存或服务器来重验证它所拥有的缓存项（如果有的话）。此初始请求不包含一个缓存验证条件；沿着路径上的第一个缓存（如果有的话），如果它拥有那个资源的一个缓存项,那么此缓存会包含一个带有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的缓存项,并且客户端已经在请求里包含了它拥有的验证器,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两个验证器中的一个去执行请求。
>
> 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 303（没有改变）回复，那么此缓存能返回一个它自己的当前已经验证了的副本给客户端同时以一个 200（OK）状态码。如果服务器以一个新实体和一个新的缓存验证器来回复请求,那么此中间缓存能把返回的验证器同客户端请求里的验证器作比较，这通过利用一个强比较方法。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，那么它不应该 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，一个客户端可能希望一个缓存只返回它当前

保存的响应，并且不需要通过源服务器对其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 缓存控制指令在请求里。如果缓存接收了这样的指令，那么它应该利用一个与请求所要求的缓存项去响应，或者以 504（网关超时）状态码响应。然而，如果一组缓存作为一个统一的系统来操作，并且有非常好的内部连接，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（它只有一个小影响），所以此协议照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响应里并被一个缓存接收后，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，如果（基于源服务器的 Expires 或 max-age 值）缓存响应是陈旧的，那么，此缓存每次必须进行 end-to-end 重验证）

must-revalidate 缓存控制指令对某个协议特性的可信赖性操作要有必要支持。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。

服务器应该发送 must-revalidate 缓存控制指令,只有在客户端或缓存对那个实体的重验证请求的操作失败的时候，例如一个不动声息的没有执行的金融事务。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（译注：验证失败说明缓存里保存的副本通过源服务器验证是无效的;验证成功说明缓存里保存的副本通过源服务器验证是有效的，它可以用来响应后续的请求）。

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，它必须显示地去警告用户这是一个没有验证的响应，而且用户户代理还应该需要用户的确认信息。

proxy-revalidate

proxy-revalidate 缓存控制指令和 must-revalidate 缓存控制指令有相同的语义，除了它并不能应用于非共享的用户代理的缓存。它能被用于一个已经被授权请求的响应，去允许用户的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（这是为了保证每个用户已经被授权）。注意这样的授权响应照样需要 public 缓存控制指令，这是为了允许他们完全能被缓存。

14.9.5 No-Transform 缓存控制指令

no-ransform

中间缓存(代理)的实现者们已经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，会引发严重的运作方面的问题。比如，医学图象应用，科学数据分析和端到端认证都依赖于接收到的实体主体与原实体主体一比一的关系。

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就不应

该改变 13.5.2 节中列出的头域。这意味着缓存或代理不得改变由这些报头定义的实体主体的任何方面，包括实体主体本身的值。

14.9.6 缓存控制扩展（Cache control Extendions）

Cache-Control 头域可通过一个或多个 cache-extension 标记的使用来实现其扩展。其中每一标记都赋予一可选的值。 信息扩展（那些无须改变缓存机行为的）可以不经改变其它缓存控制指令的语义而添加。行为扩展是通过修改现有缓存控制指令的基本行为来实现的。 新缓存控制指令与标准缓存控制指令都被提供，以至于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，而那些理解新指令的应用程序则将其看做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。

这一扩展机制依赖于这样一个 HTTP 缓存，此缓存遵从所有缓存控制指令的缓存，遵从某些扩展，而且会忽略所有它不能理解的缓存控制指令。

例如，考虑一个假想的名为"community"的新的缓存响应控制指令，此指令被看成是对 private 缓存控制指令的修饰。我们定义这个新的缓存控制指令以表明：除了非共享缓存能保存此响应之外，还有在 community 里以它值命名的社区，只有在此社区里的成员所共享的任何缓存才能保存此响应。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个私有响应，那么此源服务器应该包含：

```
Cache-Control: private, community="UCI"
```

一个见到此头域的缓存将会正确的操作，即使此缓存不能理解这个 community 缓存扩展，因为缓存照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。

14.10 Connection

Connection 常用头域允许发送者指定某一特定连接中的选项，这些选项不得由代理（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 代理必须在转发报文之前解析 Connection 头域，然后针对此头域中每一个 connection-token，从报文中移开所有与 connection-token 里同名的头域。 连接选项是由 Connect 头域中的 connection-token 指明的，而非任何对应的附加的头域，因为这些附加头域在缺少与连接选项相关的参数时可能无法被传送。

Connect 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，这是为了让发送者表明在完成响应后连接将被关闭。

例如

```
Connection: close
```

表明无论是出现在请求或响应的头域中，都表明连接不应被视为在完成现有请求/响应后是"持续的（persistent）"（参见 8.1 节）。

不支持持续连接的 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 Connect 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。这样做避免了早于 HTTP/1.1 版本的代理错误地转发这些头域。

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的内容编码，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身份下对文档进行压缩。

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），此头域必须列出不是 identity 的内容编码。

若实体的内容编码（content-coding）是一个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能会在其它的实体头域里提供，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，那么缺省是内容是支持所有不同语言的用户。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,也可能发送者不知此内容该面向何种语言。

要面向多种听众，在 Content-Language 头域里可列出多种语言。例如，同时用毛里土语和英语发行"Treaty of Waitangi"就可以用下面表示：

Content-Language: mi, en

然而，在实体中有多种语言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 -- 它不限于文本式的文档。

14.13 Content-Length

Content-Length 实体头域按十进制或八位字节数指明了发给给接收者的实体主体的大小，或是在使用 HEAD 方法的情况下，指明若请求为 GET 方法时应该发送的实体主体的大小。

```
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。 4.4 节描述了如何判断消息主体的长度，如果一个 Content-Length 没有在消息里给定。

请注意此头域的含义与 MIME 中的关于此头域的定义有很大的不同，MIME 中，它在 content-type 类型为"message/external-body"的消息里是可选的。在 HTTP 中，除非被 4.4 节里定义的规则被禁用，否则一旦消息的长度要在传送前被确定，就应发送此头域。

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和请求 URI 不是同一位置时。一个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，并且这些实体会有各自的位置，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 的替代物；它只能是陈述了请求时对应的特定实体资源的位置。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资源的话。

一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 URI，那么缓存不会利用此实体去响应使用了 Content-Locaton 里 URI 的后续请求。然而，Content-Location 能被用于区分同一个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。

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

14.15 Content-MD5

如 RFC 1864[23]中所定义的，Content-MD5 实体头域含有的是实体主体的 MD5 摘要，这是为了给一个 end-to-end 消息的实体主体的提供完整性检测。(注：MIC 有利于检测实体主体传送中的偶发性的改动，但不一定能防范恶意袭击。)

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生成，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；不得由代理服务器和网关生成，否则会有悖于其作为端到端完整性检验的价值。任何实体正主体的接收者，包括代理和网关，都可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检验 Content-MD5 值与接收到的实体之前被解除。

这样造成的后果是：摘要完全按照实体主体（entity-body）若未经传输编码（transfer encoding）编码而被以发出的顺序进行逐字节计算得到的。

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。

由此产生了一系列影响。复合媒体类型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 的内容已经应用了此编码,并且认为此 body-part 被包含在 Content-MD5 的摘要里是在应用了此编码之后。Transfer-Encoding 头域不需要出现在 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 和与 Content-Encoding 头域。其次，HTTP 比 MIME 更多地使用二进制内容类型，在此种情况下，用于计算摘要的字节顺序也即由类型定义的传输字节的顺序。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那一部分被采用。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。

与 byte-ranges-specifier 值（参见 14.35.1 节）不同的是，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）的服务器应该包含一个 Content-Range 头域，且里面的 byte-range-resp-spec 的值为"*"。instance-length 指明了选定资源的长度。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 消息里包含单一范围时，(比如，对单一范围请求的响应，或对一组能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 头域来表明实际需要被传输字节的长度。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，对多个未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。为此目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。

对单一范围请求的响应不得使用 multipart/byteranges 媒体类型。若对多个范围请求的响应结果为一个单一范围，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）。一个客户端无法对 multipart/byteranges 消息解码，那么它不能在一个请求中请求多个字节范围。

当客户端在一个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的顺序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字 byte-range-spec，它应把请求里的无效范围看成不存在。（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个请求，此请求包含一个无法满足的 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 08:12:31 GMT

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，如果这条消息要被这个接收者缓存或者这条消息需要穿过一个需要日期的协议网关。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，应该使用一种机制使，例如 NTP[28]，去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。一个恰当的相当精确的日期和时间。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 赋予一个过去值，此值为服务器设置时间或此设置时间之前的值。(这允许响应的"pre-expiration"不需要为每个资源保存分离的 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由 14.24，14.26 和 14.44。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 expection，服务器必须以 417（期望失败）状态码响应，或者如果服务器对请求遇到其它问题，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）给出了在何日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验证（或者被一个拥有实体的保鲜副本的中间缓存）。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源资源（译注：存放于源服务器的资源）在 Expire 指定时间时、之前或之后将会改变或将会不存在。

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他无效的日期格式，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。

对于缺省不可被缓存的响应而言，除非被 Cache-Control 头域（见 14.9 节）指明，否则如果 Expires 头域里日期值为响应将来的时间，那么就表明此响应是可缓存的。

14.22 From

From 请求报头域，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"定义同时也在 RFC 1123 [8]里修订了:

```
From   = "From" ":" mailbox
```

例如:

```
From: webmaster@w3.org
```

头域可以被用于记录日志和作为识别无效或多余请求的资源。他不应该用作不安全形式的访问保护。这个头域的解释是：请求是代表所指定的人执行的，此人应该承担这个方法执行的责任。特别的，机器人代理程序应该包含这个头域，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）分离的。例如，当一个请求通过一个代理（proxy）时，初始请求发送者的地址应该被使用。

客户端在没有用户的允许是不应该发出 From 头域的，因为它可能和用户的个人利益或者他们站点的安全政策相冲突。强烈建议在任何一次请求之前用户能取消，授权，和修改这个头域的值。

14.23 Host

Host 请求头域说明了正在请求资源的网络主机和端口号，这可以从源 URI 或引用资源（通常是一个 HTTP URL，这在 3.3.3 节里描述）。Host 头域值必须代表源服务器或网关（被源 URL 指定）的命名授权（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 地址的服务器,它的根"/"URL 对应有多个主机名。

```
Host = "Host" ":" host [ ":" port ] ; Section 3.2.2
```

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合适的 Host 头域，此头域指定了代理请求的服务地址。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里获得一个或多个实体(entity),那么他可以通过在 If-Match 头域里包含相应的实体标签（entity tag）来验证这些实体的一个或多个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。它照样被用于防止通过更新请求对一个资源其它版本的不经意修改。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相似的 GET 请求（没有 If-Match 头域）返回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个更新方法（updating method)（例如 PUT 方法）去修改一个客户端上次请求的但现在已经改变了的资源时，

如果请求在假设在没有 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 头域但没有 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后才能被服务器返回。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: Range 请求头域修改了 If-Modified-Since 的含义；详细信息见 14.35。

注: 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: 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，而不是稍差的比较方法，来决定是否发送响应 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，客户端被建议去利用精确的日期字符串,此字符串是以前的 Last-Modified 头域里被接收的。

注: 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是由服务器通过对时间的理解来解释的。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刻与后来请求里头域 If-Modified-Since 指定的日期之间，文档已经改变，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。一个客户端如果有一个或多个从某资源获得的实体，那么他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方法（如，PUT）不经意的改变一个已经存在的资源，而客户端还相信那个资源并不存在时。

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

资源的修改日期不能匹配此请求里 If-Modified-Since 头域（假设有的话）里提供的日期 。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。那就是说，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

如果客户端在其所指的缓存中有一个实体的部分副本,并希望其缓存中有此实体的完整副本，那么此客户端应该在一个条件 GET（conditional GET）（在 GET 请求消息里利用了 If-Unmodified-Since 和 If-Match 头域，或利用了其中一个）请求里利用 Range 请求头域（request-header）。然而，如果由于实体被改变而使条件失败，那么此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 头域允许客户端短路（short-circuit）第二次请求。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（译注：发送 range 指明的实体范围）；如果实体改变了，那就把整个新的实体发过来。

```
If-Range = "if-Range" ":"( entity-tag | HTTP-date)
```

若客户端没有一个实体的实体标签（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 头域应该只能与一个 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）指明了变量（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。

源服务器不得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，此服务器应该用消息产生的时间替换那个日期。

源服务器获得实体的 Last-Modified 值应该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 服务器应该发生 Last-Modified 头域无能何时。

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 指示的 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 应该指明了服务器自动重定向资

源喜爱的 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 请求头域提供一种机制, 那就是利用 TRACE (9.8 节) 和 OPTIONS (9.2 节)方法去限制代理或网关的数量,这些代理或网关能传递请求到下一个入流(inbound)服务器。这是非常有帮助的, 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。

Max-Forwards = "Max - Forwards" ":" 1*DIGIT

Max-Forwards 值是十进制的整数, 它指定了请求消息剩余重定向的次数。

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) 用相同的语义, 并且它为了同 HTTP/1.0 向后兼容而定义的。当一个 no-cache 请求发送给一个不遵循 HTTP/1.1 的服务器时, 客户端应该即包含 pragma 指令, 也应该包含 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 指明了授权方案，而 parameters 应用于此请求 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行（downstrem）客户端。然而，一个中间代理可能需要从请求下行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个需要授权的代理能给自己（或客户端的用户）授权。Proxy-Authorization 头域值由包含用户代理授权信息的证书组成，此授权信息是关于对代理和/或请求资源域来说的。

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利用 Proxy-Authenticate 头域进行授权的输出代理。

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 方法可以请求一个或多个实体的字节范围，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本应该尽可能支持字节范围，因为 Range 高效地支持从部分传输失败的恢复，并且支持高效地从大的实体中获取部分内容。

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，那么这将会改变返回结果（译注：本来

是返回整个实体，但出现 Range 头域后，就返回了一部分）如果 GET 请求假设在没有 Range 头域时被服务器成功处理。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 中任意一个或他们两个，或者利用了 If-Unmodified-Since 和 If-Match 中的任意一个或他们两个）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成功并且条件为真。但它不会影响 304（没有改变）响应的返回如果条件为假。

某些情形下，除了使用 Range 头域外，使用 If-Range 头域（见 14.27 节）可能更合适。

如果支持范围请求的代理接收了一个范围请求，它会转发（forward）此请求到入流（inbound）服务器，并且接收整个返回实体，但它只是返回给客户的请求的范围。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，为了对服务器有用，允许客户指定某资源的 URI，客户端从此资源获得的请求 URI 的地址（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产生返回到资源的 URI 链接的列表。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 从一个本身没有 URI 的资源获得，例如用户从键盘输入。

获得请求 URI 的资源地址（URI）-为了服务器的利益.Referer 请求头允许服务器生成关于到资源的反向连接（back-link）的列表,为了兴趣,记录,优化的高速缓存等等.它也允许追踪过时的或错误类型的连接（link）以便维护.如果请求 URI 是从一个没有自己的 URI 的源获得的,如从使用者键盘的输入,那么一定不要发送 Referer 域.

```
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。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:

```
Server: CERN/3.0 libwww/2.17
```

若响应是通过代理服务器转发的，则代理程序不得修改服务器响应头域。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，和是否愿意接收块(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。

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:

块(chunked)传输编码总是可以接受的(译注:所以不需要在 TE 头域里指定块(chunked)传输编码，因为请求端总是可以接收块传输编码)。如果在 TE 头域里出现关键字"trailers",那么客户端指明它愿意接受块(chunked)传输编码响应里的尾部(trailer)。这意味着客户端或者正在声明所有的下游客户端愿意接收块（chunked）传输编码响应里的尾部（trailer），或者声明它愿意代表下游接收端去尝试缓存响应。

> 注：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。

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））

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 常用头域值指明了在以块（chunked）传输编码消息里的尾部（trailer）里用到的头域。

Trailer = "Trailer" ":" 1#field-name

一个 http/1.1 消息会包含一个 Trailer 头域，如果它利用了块（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，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

传输编码（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编码（Transfer-Encoding）头域。

14.42 Upgrade

Upgrade 常用头域允许客户端指定它支持什么样的附加传输协议，如果服务器会切换到 Upgrade 指定的协议如果它觉得合适的话。服务器必须利用 Upgrade 头域于一个 101(切

换协议）响应里，用来指明将哪个协议被切换了。

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。这样做是通过允许客户端通知自己期望使用另一种协议来实现的，例如更新版本的 HTTP 协议，即使当前请求仍然使用 HTTP/1.1。使不兼容协议的迁移变得简单，这只需要客户端去发起一个都被支持的协议的请求,并且向服务器指明自己想要使用更好的协议如果可行的话。

Upgrade 头域只能应用于切换应用程序层（application -layer）协议，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；服务器可以接受并且可以选择。在协议改变后应用程序层（apllication-layer）通信的能力和本质，完全依赖于新协议的选择，尽管在改变协议后的第一次动作必须是对初始 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用在同一协议里使用此名字才有用。

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，产品标记按重要程度的顺序排列从而去指定应用程序。

User-Agent = "User-Agent" ":" 1*(product | comment)

例子:

User-Agent: CERN-LineMode/2.15 libwww/2.17b3

14.44 Vary

Vary 头域值指定了一些请求头域，这些请求头域用来决定当缓存中存在一个响应是保鲜时缓存是否被允许去利用此响应去回复后续请求而不需要重验证（revalidation）。对于一个不能被缓存或陈旧的响应，Vary 头域值用于告诉用户代理选择表现形式（reprentation）的标准。一个 Vary 头域值是"*"意味着缓存不能从后续请求的请求头域来决定合适表现形式的响应。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。这样做是允许缓存合适地解析关于那个资源的将来请求，并通知用户代理那个资源导向地址的出现。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并且响应据此而变化的维度信息。

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"*"意味着不受限于请求头域的非特定参数在选择响应表现形式中起作用 。"*"值不能被代理服务器产生；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和代理使用，用来指明用户代理和服务器之间关于请求的中间协议和接收者,和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的状态或变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来自于缓存操作或来自于应用于消息实体主体转换的关于语义透明性（semantic transparency）的缺少。

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，并且能被应用于响应消息。新的 Warning 头域应该加在任何已存在 Warning 头域的后面。缓存不能删除任何它接收的消息里的 Warning 头域。然而，如果一个缓存要去成功验证了一个缓存项，那么它应该移除任何附加在那个缓存项以前的 Warning 头域除了特定于此 Warning code 的 Warning 头域。它然后必须添加任何接收的 Warning 头域于验证响应

里。换句话说，附加的必须是最近的相关于响应的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）的该变，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 指明授权方案 (译注：有的地方翻译成模式) 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。讨论并不包含对披露问题的明确的解决办法，然而，确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 不能调整数据传输的内容，也没有任何经验方法去决定给定请求背景里特定信息的敏感性。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。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。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关于指定防火墙后的主机 (host) 头域信息的传输。特别地，代理应该移除或用杀毒后的版本替换任何产生于防火墙之后的 Via 头域。

Referer 头域允许学习的读模型和反向链接牵引。虽然它非常有用，但也会被滥用如果

用户详情没有从包含在Referer头域里信息分离开来。即使当个人信息已经被移除了，那么这个Referer头域也可能指定私有的文档URI，此文档不适合作为共有的。

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，因此这些信息在用户使此头域内容无效、有效和改变的情况下不能被传输。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触发器来使发送From和Referer头域信息有效或失效。

User-Agent（14.4节）或Server（14.38节）头域有时候能被用来去确定一个特定的客户端或服务器存在安全漏洞。不幸的是，同样的信息经常被用于其它的有价值的目的因为HTTP现在没有更好的机制。

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资源,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/匿名方式会有一个触发开关，此开关可能使Referer头域和From头域信息的发送有效/无效。

客户端不应该包含一个Referer头域在一个非安全HTTP请求里，如果参考页面在一个安全的协议上传输。

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，和用户代理将记录请求URI于某些对第三方可见的地方。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。提供选择设置Accept-Language头域内容于每次请求里的用户代理被强烈鼓励让设置过程包含一个让用户知道隐私丢失的消息。

限制私有信息的方法可能是在缺省情况下让用户代理不发送Accept-Language头域,并且让用户代理询问用户是否发送Accept-Language头域给服务器如果用户代理通过查看任何由服务器产生的Vary响应头域时发现这次发送能提高服务的质量。

请求里的用户配置性的接受头域如果这些接受头域（accept header fileds）包含质量值,那么他们应该被服务器用作相对信赖和长久的用户标识符。这样的用户标识符将会允许内容提供者进行click-trail跟踪以及允许合作内容提供者匹配跨服务器click-trail或者形成单个用户窗体提交。注意对于许多并不在代理服务器后面的用户，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。在代理服务器被用作增强隐私的环境里，用户代理在提供给终端用户接受（accept）头域配置选项上应该是保守的。提供高度头域配置能力的用户代理应该警告用户隐私可能的丢失。

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小心地限制HTTP请求返回的文档给那些由管理员授权的人。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。对于这一个系统，如果HTTP服务器允许访问一个资源，但这些资源通过HTTP服务器不能访问,那么一个HTTP服务器必须不允许任何这样的构造

存在于请求URI。同样的，用作服务器内部文件的引用的文件（如访问控制文件，配置文件，脚本代码）必须受到保护不让不合适的获取，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。

特别地,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，而不是缓存以前主机（host）名称查找（host name lookups）结果。许多平台已经能本地的去缓存主机名称查找（host name lookups）当在合适的时候，并且他们应该被配置成能这样做。然而，只有当被名称服务器报告的TTL（Time To Live）信息使被缓存的信息仍然有用时，缓存查找（lookups）才是合适的。

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,那么他们就会被欺骗当以前访问的IP地址改变时。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。

此要求照样能提供客户端负载平衡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,那么它必须检查自称的某个组织控制下产生响应里Location和Content-Location头域值,以确认这些组织没有企图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确定地保留授权信息。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。在某些情况下，证书的缓存能干涉应用程序的安全模型，此情况包含但不局限于:

- 这样的客户端。此客户端已经空闲时间超长并且服务器可能希望再次让客户端出示证书。
- 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"logout"或者"commit"的按钮），依据此指令应用程序的服务器端"知道"没有更多的理由为客户端保留证书。

这是作为当前单独研究的。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。特别地，能缓存证书的用户代理被鼓励去提供一个容易地访问控制机制让在用户的控制下去丢弃缓存的证书。

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。代理运行在其上系统的折中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。一个妥协的代理，或一个没有考虑安全性和隐私性的代理可能会被用做进行攻击的代理。

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。特别的，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己不比运行代理的操作员更值得信赖；HTTP 协议自身不能解决这个问题。

当合适的时候，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　　Ross Patterson

Harald Tveit Alvestrand

Keith Ball

Brian Behlendorf

Paul Burchard

Maurizio Codogno

Mike Cowlishaw

Roman Czyborra

Michael A. Dolan

David J. Fiander

Alan Freier

Marc Hedlund

Greg Herlihy

Koen Holtman

Alex Hopmann

Bob Jernigan

Shel Kaphan

Albert Lunde

John C. Mallery

Jean-Philippe Martin-Flatin

Mitra

David Morris

Gavin Nicol

Bill Perry

Jeffrey Perry

Scott Powers

Owen Rees

Luigi Rizzo

David Robinson

Marc Salomon

Rich Salz

Allan M. Schiffman

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。

17 参考资料 （Reference）

[1] Alvestrand, H., "Tags for the Identification of Languages", RFC 1766, March 1995.

[2] Anklesaria, F., McCahill, M., Lindner, P., Johnson, D., Torrey, D. and B. Alberti, "The Internet Gopher Protocol （a distributed document search and retrieval protocol）", RFC 1436, March 1993.

[3] Berners-Lee, T., "Universal Resource Identifiers in WWW", RFC 1630, June 1994.

[4] Berners-Lee, T., Masinter, L. and M. McCahill, "Uniform Resource Locators （URL）", RFC 1738, December 1994.

[5] Berners-Lee, T. and D. Connolly, "Hypertext Markup Language - 2.0", RFC 1866, November 1995.

[6] Berners-Lee, T., Fielding, R. and H. Frystyk, "Hypertext Transfer Protocol -- HTTP/1.0", RFC 1945, May 1996.

[7] Freed, N. and N. Borenstein, "Multipurpose Internet Mail Extensions （MIME） Part One: Format of Internet Message Bodies",

RFC 2045, November 1996.

[8] Braden, R., "Requirements for Internet Hosts -- Communication Layers", STD 3, RFC 1123, October 1989.

[9] Crocker, D., "Standard for The Format of ARPA Internet Text Messages", STD 11, RFC 822, August 1982.

[10] Davis, F., Kahle, B., Morris, H., Salem, J., Shen, T., Wang, R., Sui, J., and M. Grinbaum, "WAIS Interface Protocol Prototype Functional Specification," (v1.5), Thinking Machines Corporation, April 1990.

[11] Fielding, R., "Relative Uniform Resource Locators", RFC 1808, June 1995.

[12] Horton, M. and R. Adams, "Standard for Interchange of USENET Messages", RFC 1036, December 1987.

[13] Kantor, B. and P. Lapsley, "Network News Transfer Protocol", RFC 977, February 1986.

[14] Moore, K., "MIME (Multipurpose Internet Mail Extensions) Part

Three: Message Header Extensions for Non-ASCII Text", RFC 2047, November 1996.

[15] Nebel, E. and L. Masinter, "Form-based File Upload in HTML", RFC 1867, November 1995.

[16] Postel, J., "Simple Mail Transfer Protocol", STD 10, RFC 821, August 1982.

[17] Postel, J., "Media Type Registration Procedure", RFC 1590, November 1996.

[18] Postel, J. and J. Reynolds, "File Transfer Protocol", STD 9, RFC 959, October 1985.

[19] Reynolds, J. and J. Postel, "Assigned Numbers", STD 2, RFC 1700, October 1994.

[20] Sollins, K. and L. Masinter, "Functional Requirements for Uniform Resource Names", RFC 1737, December 1994.

[21] US-ASCII. Coded Character Set - 7-Bit American Standard Code for Information Interchange. Standard ANSI X3.4-1986, ANSI, 1986.

[22] ISO-8859. International Standard -- Information Processing --

8-bit Single-Byte Coded Graphic Character Sets --

Part 1: Latin alphabet No. 1, ISO-8859-1:1987.

Part 2: Latin alphabet No. 2, ISO-8859-2, 1987.

Part 3: Latin alphabet No. 3, ISO-8859-3, 1988.

Part 4: Latin alphabet No. 4, ISO-8859-4, 1988.

Part 5: Latin/Cyrillic alphabet, ISO-8859-5, 1988.

Part 6: Latin/Arabic alphabet, ISO-8859-6, 1987.

Part 7: Latin/Greek alphabet, ISO-8859-7, 1987.

Part 8: Latin/Hebrew alphabet, ISO-8859-8, 1988.

Part 9: Latin alphabet No. 5, ISO-8859-9, 1990.

[23] Meyers, J. and M. Rose, "The Content-MD5 Header Field", RFC

1864, October 1995.

[24] Carpenter, B. and Y. Rekhter, "Renumbering Needs Work", RFC

1900, February 1996.

[25] Deutsch, P., "GZIP file format specification version 4.3", RFC

1952, May 1996.

[26] Venkata N. Padmanabhan, and Jeffrey C. Mogul. "Improving HTTP Latency", Computer Networks and ISDN Systems, v. 28, pp. 25-35, Dec. 1995. Slightly revised version of paper in Proc. 2nd International WWW Conference '94: Mosaic and the Web, Oct. 1994, which is available at http://www.ncsa.uiuc.edu/SDG/IT94/Proceedings/DDay/mogul/HTTPLatency.html.

[27] Joe Touch, John Heidemann, and Katia Obraczka. "Analysis of HTTP Performance", <URL: http://www.isi.edu/touch/pubs/http-perf96/>, ISI Research Report ISI/RR-98-463, (original report dated Aug. 1996), USC/Information Sciences Institute, August 1998.

[28] Mills, D., "Network Time Protocol (Version 3) Specification, Implementation and Analysis", RFC 1305, March 1992.

[29] Deutsch, P., "DEFLATE Compressed Data Format Specification version 1.3", RFC 1951, May 1996.

[30] S. Spero, "Analysis of HTTP Performance Problems,"

http://sunsite.unc.edu/mdma-release/http-prob.html.

[31] Deutsch, P. and J. Gailly, "ZLIB Compressed Data Format

Specification version 3.3", RFC 1950, May 1996.

[32] Franks, J., Hallam-Baker, P., Hostetler, J., Leach, P.,

Luotonen, A., Sink, E. and L. Stewart, "An Extension to HTTP:

Digest Access Authentication", RFC 2069, January 1997.

[33] Fielding, R., Gettys, J., Mogul, J., Frystyk, H. and T.

Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1", RFC

2068, January 1997.

[34] Bradner, S., "Key words for use in RFCs to Indicate Requirement

Levels", BCP 14, RFC 2119, March 1997.

[35] Troost, R. and Dorner, S., "Communicating Presentation

Information in Internet Messages: The Content-Disposition

Header", RFC 1806, June 1995.

[36] Mogul, J., Fielding, R., Gettys, J. and H. Frystyk, "Use and

Interpretation of HTTP Version Numbers", RFC 2145, May 1997. [jg639]

[37] Palme, J., "Common Internet Message Headers", RFC 2076, February 1997. [jg640]

[38] Yergeau, F., "UTF-8, a transformation format of Unicode and ISO-10646", RFC 2279, January 1998. [jg641]

[39] Nielsen, H.F., Gettys, J., Baird-Smith, A., Prud'hommeaux, E., Lie, H., and C. Lilley. "Network Performance Effects of HTTP/1.1, CSS1, and PNG," Proceedings of ACM SIGCOMM '97, Cannes France, September 1997.[jg642]

[40] Freed, N. and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types", RFC 2046, November 1996. [jg643]

[41] Alvestrand, H., "IETF Policy on Character Sets and Languages", BCP 18, RFC 2277, January 1998. [jg644]

[42] Berners-Lee, T., Fielding, R. and L. Masinter, "Uniform Resource

Identifiers (URI): Generic Syntax and Semantics", RFC 2396, August 1998. [jg645]

[43] Franks, J., Hallam-Baker, P., Hostetler, J., Lawrence, S., Leach, P., Luotonen, A., Sink, E. and L. Stewart, "HTTP Authentication: Basic and Digest Access Authentication", RFC 2617, June 1999. [jg646]

[44] Luotonen, A., "Tunneling TCP based protocols through Web proxy servers," Work in Progress. [jg647]

[45] Palme, J. and A. Hopmann, "MIME E-mail Encapsulation of Aggregate Documents, such as HTML (MHTML)", RFC 2110, March 1997.

[46] Bradner, S., "The Internet Standards Process -- Revision 3", BCP 9, RFC 2026, October 1996.

[47] Masinter, L., "Hyper Text Coffee Pot Control Protocol (HTCPCP/1.0)", RFC 2324, 1 April 1998.

[48] Freed, N. and N. Borenstein, "Multipurpose Internet Mail

Extensions （MIME） Part Five: Conformance Criteria and Examples", RFC 2049, November 1996.

[49] Troost, R., Dorner, S. and K. Moore, "Communicating Presentation Information in Internet Messages: The Content-Disposition Header Field", RFC 2183, August 1997.

18 作者地址

Roy T. Fielding

Information and Computer Science

University of California, Irvine

Irvine, CA 92697-3425, USA

Fax: +1 （949） 824-1715

EMail: fielding@ics.uci.edu

James Gettys

World Wide Web Consortium

MIT Laboratory for Computer Science

545 Technology Square

Cambridge, MA 02139, USA

Fax: +1 (617) 258 8682

EMail: jg@w3.org

Jeffrey C. Mogul

Western Research Laboratory

Compaq Computer Corporation

250 University Avenue

Palo Alto, California, 94305, USA

EMail: mogul@wrl.dec.com

Henrik Frystyk Nielsen

World Wide Web Consortium

MIT Laboratory for Computer Science

545 Technology Square

Cambridge, MA 02139, USA

Fax: +1 (617) 258 8682

EMail: frystyk@w3.org

Larry Masinter

Xerox Corporation

3333 Coyote Hill Road

Palo Alto, CA 94034, USA

EMail: masinter@parc.xerox.com

Paul J. Leach

Microsoft Corporation

1 Microsoft Way

Redmond, WA 98052, USA

EMail: paulle@microsoft.com

Tim Berners-Lee

Director, World Wide Web Consortium

MIT Laboratory for Computer Science

545 Technology Square

Cambridge, MA 02139, USA

Fax: +1 (617) 258 8682

EMail: timbl@w3.org

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，还用作互联网媒介类型"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)的传输路径(pipeline)。下列是在 IANA[17]注册的。

媒介类型名称:　　message

媒介次类型名称:　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

媒介类型名称:　　application

媒介次类型名称:　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HTTP206（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。

媒介类型名称：　multipart

媒介次类型名称：　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:

在实体（entity）中，在第一个分界字符串之前可以有多余的CRLFs。

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

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 宽松的应用程序 （Tolerent Applications）

虽然这篇文档列出了HTTP/1.1消息所必须的元素，但是并不是所有的程序在实现的时候都是正确的。因此我们建议当偏差可以明确解释的时候，运算程序对这种偏差应该是宽容的。

客户端应该宽松的解析Status-Line(状态行);服务器也应该宽松的解析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 的代理服务器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（MAY）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]需要一个 Internet mail 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。这篇文档的 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 要求以类型为"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 到严格 MIME 环境的代理或网关应该（SHOULD）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 遵守的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 Internet mail 类型的 Content-Type 的实验性的应用已经使用了一个媒体类型参数"; 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

客户端和服务器必须支持 Host 请求头域，并且如果 Host 请求头域在 HTTP/1.1 请求里缺少，那么服务器应该报告一个错误，并且服务器能接受一个绝对 URIs（5.1.2 节），对于这个要求是在此规范里最重要的改变。上面的改变将允许互联网，一旦旧的客户端不再常用时，去支持一个 IP 地址的多个 web 站点，这大大简化了大型运算的 web 服务器，在那里分配多个 IP 地址给一个主机（host）会产生很严重的问题。此互联网照样

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 HTTPURLs 的域名。给定 web 的增长速度，服务器的部署数量部，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（MUST）支持 Host 请求头域。

--发送 HTTP/1.1 请求的客户端必须（MUST）发送 Host 头域。

--如果 HTTP/1.1 请求不包括 Host 请求头域，服务器就必须（MUST）报告错误 400（Bad Request）。

--服务器必须（MUST）接受绝对 URIs（absolute URIs）。

19.6.2 和 HTTP/1.0 持续连接的兼容

一些客户端和服务器可能希望和一些对以前持续连接实现的 HTTP/1.0 客户端和服务器兼容。HTTP/1.0 持久连接需要显示地协商，因为它们不是缺省的行为。持久连接的 HTTP/1.0 实验性的实现有错误,并且 HTTP/1.1 里的新的功能被设计成去矫正这些问题。此问题是一些已经存在的 1.0 客户端可能会发送 Keep-Alive 头域给不理解连接的代理服务器，然后代理服务器会把此 Keep-Alive 转发给下一个入流（inbound）服务器。所以 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,并且它必须能安全的使用甚至当和忽略连接的老代理。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（the Connection: Keep-Alive and 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……