# 超文本传输协议（ HTTP/1.1 ）： 报文语构和路由

## 摘要

超文本传输协议是一种为分布式的、协作化的超文本信息系统而设计的无状态应用层协议。本篇文档提供了 `HTTP` 结构和其相关术语的概述，定义了 `http` 和 `https` 统一资源标识符 `URI` 规范及 HTTP/1.1 信息语构和解析要求，并且描述了在实现上的相关安全注意事项。

## 目录

# 1. 引言

超文本传输协议（ `HTTP` ）是一种基于请求/响应模型的无状态应用层协议，它利用其极具拓展性的语义和具有自我描述性的消息内容来与基于网络的超文本信息系统进行灵活的交互。

本篇文档是系统性定义 HTTP/1.1 规范的系列文档中的第一份：

- RFC-7230 报文语构和路由
- RFC-7231 语义和内容
- RFC-7232 条件请求
- RFC-7233 范围请求
- RFC-7234 缓存
- RFC-7235 认证

这份 HTTP/1.1 规范使得 RFC-2616 及 RFC-2145 中关于 HTTP/1.1 的部分被废弃。同时本规范还更新了之前在 RFC-2817 中被定义的"用 `CONNECT` 方法建立网络隧道"，而且定义了原先在 RFC-2818 中被非正式性描述的 `HTTPS URI` 方案。

`HTTP` 是一种用于信息系统的通用接口协议。它被设计出以暴露出独立于所提供资源类型的统一接口给客户端的方式来隐藏服务的实现细节。同样地，服务器也不需要知道客户端的任何目的：一次 `HTTP` 请求可以被视作独立的而非与特定的客户端类型或一连串预定的请求步骤相关。这些特性造就了一个可以高效应用于许多场景并且允许客户端和服务端的实现随着时间推移独立发展的协议。

`HTTP` 也同样被设计来作为与其它非 `HTTP` 信息系统交流的中间协议。`HTTP` **代理（Proxy）**和**网关（Gateway）**以把其它各种协议的信息服务内容转化为可供 `HTTP` 客户端获取并操作的超文本格式的方法来让我们可以使用这些服务。

这种灵活性导致的一个后果是，我们不能用术语定义 `HTTP` 协议里接口后面具体发生的事。我们同样很难去定义通讯的语法、获取资源的目的及接受者的预期行为。如果一次通讯被认为是独立的，那么成功的行为应当反映在服务器所提供观察接口的相应改变上。然而，考虑到可能有多个客户端正在并行请求且相互之间可能存在干扰，我们不能指望这样的改变能在单次响应之后被观察到。

> 原文是"we cannot require that
> such changes be observable beyond the scope of a single response."，猜

> 测是指，可能存在多个客户端请求同一个接口，所以在请求观察接口的时候，并不能确定刚才自己的请求是否为最后的一次请求。

本篇文档描述了在 `HTTP` 中被使用或者涉及的结构化的元素，定义了 `HTTP URI` 和 `HTTPS URI` 规范，描述了总体的网络运转和连接管理，还定义了 `HTTP` 消息的构成及发送需求。我们的目的是定义所有独立于消息语义的关于 `HTTP` 消息处理的必要机制，从而为消息解析和消息转发中介定义完整的需求集。

## 1.1 要求表示规范

本文档中的关键词**MUST**、**MUST NOT**、**REQUIRED**、**SHALL**、**SHALL NOT**、**SHOULD**、**SHOULD NOT**、**RECOMMENDED**、**MAY**和**OPTIONAL**应依照 RFC2119 中的描述，中文对照表如下。

| 原词 | 译词 |
| --- | --- |
| **MUST** | **必须** |
| **MUST NOT** | **绝不能** |
| **REQUORED** | **需要** |
| **SHALL/SHOULD** | **应该** |
| **SHALL NOT/SHOULD NOT** | **不应该** |
| **RECOMMENDED** | **建议** |
| **MAY** | **可以** |
| **OPTIONAL** | **可选地** |

有关错误处理的一致性标准和注意事项会在 Section 2.5 中定义。

## 1.2 语法表示规范

这份规范使用扩充巴科斯范式（ `ABNF` ）作为主要的语法表示规范。另外，在 Section 7 中定义了一个 `ABNF` 的列表扩展符号 `#` 操作符（与 `*` 操作符表示重复的形式类似），它允许紧凑定义的、由逗号分隔的列表。Appendix B 中展示了本规范中收录的 `ABNF` 规则，其中所有的列表定义都使用了这个拓展自标准 `ABNF` 符号的列表操作符。（WTF？真的用到了吗？我怎么看不出来？）

下列核心规则都可供参考，如同 RFC5234, Appendix B.1 中的定

义：`ALPHA`（字母），`CR`（回车），`CRLF（CR LF）`，`CTL`，`DIGIT`（数字 0-9），`DQUOTE`（双引号），`HEXDIG`（十六进制数 0-9/A-F/a-f），`HTAB`（水平 Tab），`LF`（换行），`OCTET`（字节），`SP`（空格）和 `VCHAR`（任何可见的 USASCII 字符）。

按照惯例，以 `obs-` 开头的 `ABNF` 规则表示其已经过时但因为历史遗留问题而继续保留。

# 2. 结构

`HTTP` 是为万维网（`WWW`）的建设而生的，为了支持世界范围的超文本信息系统的可拓展性需要，它也在不断地发展。`HTTP` 的大部分结构都反映在它的符号和语构上。

## 2.1 客户端 / 服务器端消息传递

`HTTP` 是一种无状态的请求/响应协议，它通过一个可靠的传输层或会话层"连接"交换信息。一个 `HTTP` **客户端（Client）**是一个为了发送一个或多个 `HTTP` **请求（Request）**而与服务器建立连接的应用程序。一个 `HTTP` **服务器（Server）**是一个接收 `HTTP` 请求并发送 `HTTP` **响应（Response）**来提供服务的应用程序。

术语**客户端**和**服务器**仅仅指代应用程序的某个特定连接。同一个程序可能同时在一些连接中扮演客户端而在另一些连接中扮演服务器。术语**用户代理（User Agent）**指代任何可以发起 `HTTP` 请求的的应用程序，包含但不仅限于：浏览器、爬虫、命令行工具、定制应用和手机 APP。术语**源服务器（Origin Server）**指代可以正确响应并返回目标资源的应用程序。术语**发信人（Sender）**和**收信人（Recipient）**分别指代任何发送或接收某条指定**报文（Message）**的具体实现。

`HTTP` 依赖统一资源标识符（`URI`）标准 RFC 3986 来标识目标资源 Section 5.1 和资源之间的关系。消息以一种跟互联网邮件 RFC 5322 及多用途互联网邮件扩展（`MIME`）RFC 2045 相似的格式被发送（Appendix A of RFC 7231 查看 `HTTP` 与 `MIME` 消息的不同点）。

大多数的 `HTTP` 通讯都从一个为了展示某些被一个 `URI` 标识的资源而发起的取回请求（`GET`）开始。在最简单的情况下，这个通讯可能只通过用户代理（`UA`）和源服务器（`O`）之间的单个连接（===）就能完成。

```
          请求 >
用户代理 ====================================== 源服务器
                                          < 响应
```

一个**客户端**会发送一个请求**报文**格式的 `HTTP` 请求给服务器，一条请求**报文**以包含 `Method` （ `HTTP` 方法）、 `URI` 和 `Protocol Version` （协议版本）的请求行开始，请求行之后是由请求修饰符、客户端信息和表示元数据组成的头字段，头字段由一个空行标志结束，最后就是包含有效载荷的报文主体（如果有的话）。

一个服务器响应一个客户端请求通过返回一个或多个 `HTTP` 响应**报文**，每个**报文**都以一个包含 `Protocol Version` （协议版本），一个成功或失败的状态码和一个可读的状态语句的状态行开始，之后可能会跟着由服务器信息、资源元数据和表示元数据组成的头字段，头字段由一个空行标志结束，最后就是包含有效载荷的报文主体（如果有的话）。

一个连接可能可以被用于多次请求/响应通讯，如 Section 6.3 中定义。

下面的例子演示了一个典型的 `GET` 请求的**报文**交换， `URI` 是 "http://www.example.com/hello.txt"：

客户端请求：

```
GET /hello.txt HTTP/1.1
User-Agent: curl/7.16.3 libcurl/7.16.3 OpenSSL/0.9.7l zlib/1.2.
3
Host: www.example.com
Accept-Language: en, mi
```

服务器响应

```
HTTP/1.1 200 OK
Date: Mon, 27 Jul 2009 12:28:53 GMT
Server: Apache
Last-Modified: Wed, 22 Jul 2009 19:15:56 GMT
ETag: "34aa387-d-1568eb00"
Accept-Ranges: bytes
Content-Length: 51
Vary: Accept-Encoding
Content-Type: text/plain
```

```
Hello World! My payload includes a trailing CRLF.
```

## 2.2 实现的多样性

当考虑 `HTTP` 的设计时，很容易陷入一个思维陷阱——认为所有的用户代理都是通用浏览器且所有的源服务器都是大型公共网站。现实并非如此，常见的用户代理包括家用设施、音响、电子秤、固件更新脚本、命令行工具、手机 APP 和各种形状大小的通讯工具。同样，常见的 `HTTP` 源服务器包括家用自动化单元、可配置的网络组件、办公机器、自动化机器人、新闻提要、交通摄像头、广告选择器以及视频交流平台。

术语**用户代理**不仅仅指可以在请求时直接与代理软件进行交互的人类用户。在很多情况下，一个用户代理被安装或者配置在后台运行然后储存结果用于之后的查看（或者仅仅储存有意义的子集或者错误信息）。比如说爬虫，就是典型的，给一个起始 `URI` 和配置就可以在遍历爬取整个互联网时遵循指定的行为。

`HTTP` 实现的多样性意味着不是所有的用户代理在考虑一些安全或隐私问题时都可以提供给用户交互式的建议或者充足的警告。在少数需要报告错误给用户的情况下，把报告写入错误控制台或者日志文件也是可接受的。同样地，需要用户确认的自动化行为可能会在用户处理之前被提前选择的配置、运行时选项或者简单的不安全行为黑名单所处理；确认并不意味着会弹出一个用户界面或者打断当前进程，如果用户早就做出了选择的话。

## 2.3 中介

`HTTP` 允许使用**中介（Intermediaries）**来发起链式请求。**中介**一共有三种形式：**代理（Proxy）**，**网关（Gateway）**和**隧道（Tunnel）**。在某些情况下，一个单独的**中介**可能会同时扮演**源服务器**、**代理**、**网关**或**隧道**，并基于每一条原始请求来切换行为。

```
        >             >             >             >
用户代理 ========== A ========== B ========== C ========== 源服务器
                  <             <             <             <
```

上面这幅图演示了在**用户代理**和**源服务器**之间的三个**中介**（A、B 和C）。一个请求或响应报文将会穿过四个单独的连接。一些 `HTTP` 通信的选项可能只会在某些连接上生效，比如最近的连接、没有隧道的临近节点、链的终端节点、或者所有节点间的连接。虽然这幅图是线性的，但每一个参与节点都可能同时在其它的

链里面扮演其它的角色。比如 B 可能从 A 之外的很多客户端接收请求，同时又向 C 之外的其它服务器转发请求，且同时在处理 A 的请求。同样地，之后的请求可能会通过连接中的不同路径，常见于基于动态配置的负载均衡。

术语**上游（upstream）**和 **下游（downstream）**用于描述报文的流向：所有的报文都从**上游**流向**下游**。术语 **入站（inbound）**和**出站（outbound）**用于描述请求的路由方向：**入站**表示向着**源服务器**方向，**出站**表示向着**用户代理**方向。

**代理**是一种由客户端自己选择的消息转发代理，通常遵循本地配置文件的规则来接收关于一些 `absolute URI` 的请求并尝试以转换到 `HTTP` 接口的方式满足这些请求。一些转换是小菜一碟，比如仅仅改变 `HTTP` 请求的 `URI`，但在另一情况下，可能需要将流量转换到另一个完全不同的应用层协议。**代理**经常用同一个**中介**代理一组 `HTTP` 请求，出于安全、注解服务或共享缓存的目的。也有一些**代理**被设计来对特定的消息进行处理，如同 Section 5.7.2 中所描述的那样。

**网关**也称作**反向代理（Reverse Proxy）**对出站连接来说可看做**源服务器**，但它会把接收的请求转发到其他的**服务器**。**网关**通常用于封装历史遗留系统或者不被信任的信息服务、通过”加速器“缓存来提高服务器性能、或者给跑在多台机器上的 `HTTP` 服务分区或负载均衡。

所有适用于**源服务器**的 `HTTP` 需求可以同样被用于**网关**的出站连接。而**网关**到后端**服务器**的入站通讯则可以用任何协议，包括超出本文档规范的 `HTTP` 扩展。当然，想要与第三方 `HTTP` **服务器**进行互操作的 `HTTP-to-HTTP` **网关**应当符合用户代理的需求（即**网关**请求其它**服务器**的时候应该被视为**用户代理**）。

**隧道**在两个连接之间充当中继的作用，它只需要原封不动地转发流量。一旦被启用，**隧道**就不被视作 `HTTP` 通讯中的一部分，即使它可能发起了一个新的 `HTTP` 请求。只有在两端的连接都断开时，**隧道**才会停止工作并退出。**隧道**通常通过一个中介来续接虚拟连接，比如通过一个共享防火墙代理用传输层安全协议（`TLS`）建立可靠的通讯。

上文中提到的几种**中介**仅仅考虑了其本身参与 `HTTP` 通讯的行为。还存在**中介**能参与网络协议栈更底层的行为，过滤或重定向 `HTTP` 流量而不会让发送者察觉，更不用发送者的许可。来自更底层网络中介的中间人攻击是无法在应用层上分辨的，这往往造成 `HTTP` 语义的误读并导致安全漏洞或交互性问题。

比如，一个 **拦截代理（interception proxy）**（也称为 `transparent proxy` 或 `captive portal`）与 `HTTP` **代理**的不同之处在于它不是由**客户端**自己选择的，但它会过滤或者重定向 `TCP` 端口 80 的出口流量（也可能是其他常见端口的流

量）。**拦截代理**在公共网络接入点是非常常见的，作为一种强制账号订阅以使用非本地网络服务的手段，或者用于企业内部防火墙强制施行网络使用政策。

`HTTP` 被定义为一种无状态协议，意味着每一个请求报文可以被独立地解析。很多实现都依赖 `HTTP` 的无状态特性以重用代理服务器的连接或者在多台服务器之间动态负载均衡。所以，任何**服务器绝不能**认为来自同一个连接的两个请求是来自同一个**用户代理**的，除非这个连接是安全且是某个**用户代理**专用的。众所周知的是一些非标准的 `HTTP` 扩展（比如 RFC-4559）违背了这项规定，从而导致了安全和交互性问题。

## 2.4 缓存

**缓存仓库（Cache）**指代储存之前的响应信息的本地仓库，以及控制、存取、删除储备信息的子系统。**缓存仓库**会缓存允许储存的响应以降低之后的响应时间和网络带宽消费，并得到与真实请求一致的结果。任何**客户端**或者**服务器可以**采用一些缓存方案，但是当**服务器**作为**隧道**的时候不能使用缓存。

缓存的作用是缩短了请求/响应链，如果链上的某个节点返回被缓存的响应。下图描述了实际的通讯链，在 B 缓存了之前 O 返回的（经由 C）的响应，而 UA 和 A 没有缓存的情况下：

```
        >             >
用户代理 ========== A ========== B ---------- C ---------- 源服务器
                  <             <
```

一份**响应**是 "cacheable(可缓存的)" 意味着缓存系统可以拷贝并储存它的响应报文以用于接下来的请求。即使一份响应是可缓存的，依然可能存在来自**客户端**或**源服务器**的关于缓存使用场景的额外约束。`HTTP` 的缓存行为要求细则和可缓存响应定义在 Section 2 of RFC7234。

关于缓存方案实施的结构和配置广泛存在于万维网和各大组织中。包括国际级的代理服务器使用缓存来节省跨洋带宽、通用的缓存内容广播或组播系统、打包缓存资源用于离线或高延迟环境等等。

## 2.5 一致性和错误处理

这份规范要规定 `HTTP` 通讯中各参与角色的一致性标准。`HTTP` 的要求细节存在于**发信人**、**收信人**、**客户端**、**服务器**、**用户代理**、**中介**、**源服务器**、**代理**、**网**

**关**、或者**缓存仓库**中，取决于要求中约束了哪些行为。当他们并非只应用于单次通讯中时，其具体实现、资源拥有者和协议元素注册则会产生额外的（社会）要求。

动词 "产生（generate）" 和 "发送（send）" 在要求中的区别在于一个创建了协议元素而另一个仅仅转发了它收到的元素给下游。

如果一个具体实现遵循了它在 `HTTP` 通讯中所扮演角色的全部要求，那么可以认为它是符合规范的。

一致性包括协议元素的语构一致性和语义一致性。任何**发信人绝不能**创建它认为是错误的协议元素。任何**发信人绝不能**创建不符合相应 `ABNF` 规则语法的元素。在一个给定的报文内，任何**发信人绝不能**创建只能由其它角色（非**发信人**）创建的协议元素或语构。

当一个已经收到的协议元素被解析时，任何**收信人必须**能够解析出任何适用于**收信人**长度合理的值并匹配所有在 `ABNF` 规则中定义的语法。值得注意的是，即使是这样，一些被接收的协议元素依然可能无法解析。比如，一个**中介**转发一个报文时可能会把一个头字段解析为通用的字段名和字段值，但在它转发这个头字段之前并不会对字段值进行进一步的解析。

`HTTP` 对它的大部分协议元素都没有特别的长度限制，因为长度存在大范围的波动是合理的，这取决于使用场景和实现目的。于是，怎样的长度是合理的取决于**发信人**和**收信人**的协商。此外，一些协议元素的通常意义上的合理长度在 `HTTP` 投入使用的二十年来也在不断变化，并且可以认定在未来还会继续发生变化。

但至少，**收信人必须**能够处理和它在其它报文中创建的同一协议元素一样长的内容。比如，一个**源服务器**给它的某个资源创建了一个非常长的 `URI reference`，那当它收到以此为目标的请求时，也应当可以处理它。

任何**收信人必须**根据本规范来解释其所接收协议元素的语义，包括这篇规范的扩展，除非**收信人**认为（凭借经验或者配置）**发信人**误解并错误地实现了这些语义。比如，一个**源服务器**可能会无视所接收的 `Accept-Encoding` 头字段，如果对其 `User-Agent` 头字段的检查发现特定的内容编码会在其特定版本的**用户代理**上失败。

除非有什么特别的规定，否则，任何**收信人可以**尝试从一个非法的结构中恢复可用的协议元素。`HTTP` 并没有定义特别的错误处理机制，除非这个错误对服务的安全性有着直接的冲击，因为不同的应用程序可能需要不同的错误处理策略。比如，一个 Web 浏览器可能想要正大光明地恢复一个定位头字段没有通过 `ABNF`

解析的**响应**，但一个系统控制客户端可能认为任何形式的错误恢复都是危险的。

## 2.6 协议版本

`HTTP` 使用一种 `<major>.<minor>` 的版本号形式去表明协议版本。这份规范定义了版本 `1.1` 。这个协议版本仅作为与**发信人**保持一致性（本规范中规定的完整要求集合）的标识。

这个版本的 `HTTP` 版本信息存在于报文第一行的 `HTTP-version` 字段。`HTTP-version` 是区分大小写的。

```
HTTP-version  = HTTP-name "/" DIGIT "." DIGIT
HTTP-name     = %x48.54.54.50 ; "HTTP", case-sensitive
```

`HTTP` 版本号由两个以 "." 隔开的数字组成。第一个数字（`major version`）决定了 `HTTP` 报文的语构，而第二个数字（ `minor version` ）表示在这个主版本中最高位的副版本号，为了之后更进一步的通讯，副版本号往往用来说明**发信人**的通讯标准。副版本号通知**发信人**的通讯兼容性，即使**发信人**只能使用这个协议版本向后兼容的子集，从而让**收信人**知道哪一些新特性能够在响应或者之后的请求中使用。

当一个 HTTP/1.1 的报文发给一个 HTTP/1.0 或协议版本未知的**收信人**，在这种所有新特性都会被忽略的情况下，HTTP/1.1 的报文会以兼容 HTTP/1.0 的形式被构建。这份规范列举了一些新特性的接受者版本要求，所以**发信人**只会使用**收信人**兼容的特性，直到它通过配置或返回消息发现**收信人**已经支持 HTTP/1.1 了。

头字段的解析在同一主版本号的不同副版本号之间是没有区别的，即使**收信人**对缺省值的默认行为不同。除非特别规定，定义在 HTTP/1.1 中的头字段就是定义在 HTTP/1.x 中的头字段。尤其是 `Host` 和 `Connection` 头字段应当在所有的 HTTP/1.x 的实现中得到相应的实现，无论它们是否兼容 HTTP/1.1。

新的头字段能在不改变版本号的情况下被引入，只要他们定义的语义能允许他们被无法识别他们的**收信人**安全地忽略。在 Section 3.2.1 中 我们讨论了头字段的可拓展性。

处理 `HTTP` 报文的**中介**（即除 `tunnels` 之外的所有**中介**）**必须**在转发报文时附带它们自己的 `HTTP-version` 。换句话来说，无论在接收消息时还说发送消息时，他们都必须确认报文的 `HTTP-version` 与自身的兼容性，而不应盲目地转

发报文的 `start-line`。转发 `HTTP` 报文而不重写 `HTTP-version` 可能会导致通讯错误，当下游的**收信人**根据**发信人**的协议版本去判断哪些特性是安全的。

任何**客户端应该**发送不超过**服务器**能支持的最高主版本，同时与自身能兼容的最高版本相同版本的请求，如果这些都是已知的话。同时，任何**客户端绝不能**发送与自身版本不兼容的请求。

任何**服务器可以**返回一个 HTTP/1.0 的响应，当它知道或怀疑该**客户端**错误地实现了 `HTTP` 规范或无法正确地处理更高版本的响应，比如**客户端**无法正确地解析版本号或者发现有一个**中介**盲目地转发了 `HTTP-version`，即使它并不兼容该报文的协议版本。这样的协议降级**不应该**被实施，除非被**客户端**某些特定的属性触发，比如某些请求头字段（比如 `User-Agent`）的值正好与某些已知错误的值相匹配。

`HTTP` 版本规范设计的目的是，只有在引入了不兼容的报文语构时才会升主版本号，而只有在引入了影响报文语义的变化或者为**发信人**增加了额外的功能时，才会升副版本号。当然，RFC-2068 到 RFC-2616 之间的改变不足以升级副版本号，这份规范的版本重定义也特别地避免了影响版本号的的改变。

当一个**收信人**收到了与自身所实现协议的主版本号一致，但副版本号更高的 `HTTP` 报文，它**应该**把改报文当做自身可兼容的最高版本来处理。**收信人**可以假定拥有更高副版本号的报文对任何主版本号相同的实现都是充分向后兼任并可以被安全处理的，只要该**收信人**自身并没有表明过它支持该更高版本的协议。

## 2.7 统一资源标识符

统一资源标识符（`URIs`）作为一种标识资源的手段被用于 `HTTP` 中。`URI references` 被用于表明请求目的、表示重定向、以及定义关系。

有关 `URI-reference`，`absolute-URI`，`relative-part`，`scheme`，`authority`，`port`，`path-abempty`，`segment`，`query`，和 `fragment` 的定义均源于 `URI` 通用语构。"absolute-path" 规则是为了可以包含一个非空路径的协议元素定义的（这个规则与允许使用空路径的 `path-abempty` 以及不允许以 `//` 开头路径的 `path-absolute` 规则略有不同）。`partial-URI` 规则是为了能包含无 `fragment` 的 `relative-URI` 的协议元素而定义的。

`URI-reference = <URI-reference, see` RFC3986, Section 4.1>

`absolute-URI = <absolute-URI, see` RFC3986, Section 4.3>

`relative-part = <relative-part, see` RFC3986, Section 4.2>

`scheme = <scheme, see` RFC3986, Section 3.1>

`authority = <authority, see` RFC3986, Section 3.2>

`uri-host = <host, see` RFC3986, Section 3.2.2>

`port = <port, see` RFC3986, Section 3.2.3>

`path-abempty = <path-abempty, see` RFC3986, Section 3.3>

`segment = <segment, see` RFC3986, Section 3.3>

`query = <query, see` RFC3986, Section 3.4>

`fragment = <fragment, see` RFC3986, Section 3.5>

```
absolute-path = 1*( "/" segment )
```

```
partial-URI = relative-part [ "?" query ]
```

任何一个允许 `URI reference` 的 `HTTP` 协议元素都会以 `ABNF` 的形式表明它被允许的 reference 形式，比如只能以绝对形式（`absolute-URI`）、只有 `path` 和可能的 `query` 成分，或者一些上述规则的组合。除非特别指明，`URI references` 会以对应的 effective request URI 的形式被解析。

### 2.7.1 `HTTP URI` 格式

`http URI` 格式被定义的目的是根据监听指定端口 TCP 连接的潜在 `HTTP` **源服务器**来建立具有层级命名空间的标识符。

```
http-URI = "http:" "//" authority path-abempty [ "?" query ]
                [ "#" fragment ]
```

`http URI` 中由包含 host 并可能附带 `TCP` 端口的 `authority` 来标识**源服务器**。分级的 `path` 和可选的 `query` 则作为在**源服务器**的名字空间中定位潜在目标资源的标识。可选的 `fragment` 可作为定位二级资源的间接标识，它是独立于 `URI scheme` 的，如同 Section 3.5 of RFC-3986 中所定义的那样。

任何**发信人绝不能**创建一个 `host` 标识为空的 `HTTP URI`。收到这种 `URI` 的**收信人必须**把它作为无效 `URI` 拒绝处理。

如果提供的 host 标识是一个 IP 地址，则**源服务器**监听在此 IP 地址的指定 TCP 端口上。如果 host 是一个注册名（域名？），这个注册名就是一个通过像 DNS 这样的名字解析服务来找**源服务器** IP 地址的间接标识。如果端口号为空或者没有指定，则默认 TCP 端口 80（万维网服务的保留端口）。

要记住，有着指定 authority 的 URI 的存在并不意味着这个 host 和 port 总有一个 HTTP **服务器**在监听。每个人都可以创建一个 URI 。 Authority 决定的是谁有权利、有权威性地响应一个请求并返回目标资源。注册名和 IP 地址被赋予的天然特性，基于对指定 host 和 port 是否存在 HTTP **服务器**的控制，创造了联合的命名空间。在 Section 9.1 中我们可以看到关于建立权威性的安全考量。

当一个 http URI 在某些语境下被用于访问指定的资源，任何客户端**可以**尝试访问此资源，通过把 host 解析为 IP 地址、在指定端口建立 TCP 连接，然后发送含有 URI 标识数据 的请求报文。如果该**服务器**用一个非过渡的 HTTP 响应报文来回复某请求，像 Section 6 of RFC-7231 中描述的那样，则可以认为这是一次对该客户端请求的具有权威性的响应。

虽然 HTTP 独立于传输层协议，但 http scheme 特指基于 TCP 的服务，因为名称代理过程（代理 authority ，即名字空间？）依赖 TCP 来建立权威的链接。基于其它底层连接协议的 HTTP 服务一般会使用其它的 URI scheme 作为标识，就像 https scheme 被用于传递有端对端加密需求的资源。其它的协议可能也会被用于提供对 http 标识资源的访问 —— 它只是一种基于 TCP 的特定的，广为人知的接口。

URI 通用语法中的 authority 也含有一个 deprecated 的 userinfo 子字段 Section 3.2.1 of RFC-3986 以便于在 URI 中包含用户认证信息。某些实现充分地利用了 userinfo 字段来进行认证信息的内部配置，比如命令调用中的选项、配置文件、或者书签列表，即使如此这样的使用可能会暴露用户的标识或者密码。任何**发信人绝不能**创建 useinfo 子字段（以及它的 "@"）当一个 HTTP URI references 在一个报文中作为请求目标或者头字段的值。在利用一个从不被信任来源收到的 HTTP URI references 之前，该**收信人应该**解析其中的 userinfo 字段并把它作为一种错误来处理。 userinfo 字段可能被用于混淆 authority 以发起网络钓鱼攻击。

### 2.7.2 HTTPS URI 格式

https URI 格式被定义的目的是根据监听指定端口被 TLS 加密的 TCP 连接的潜在 HTTP origin server 来建立具有层级命名空间的标识符。

所有在上文中列举到的有关 `HTTP scheme` 的要求也同样适用于 `HTTPS scheme`，唯二的例外就是，`TCP` 端口缺省时的默认值是 443 而非 80，以及**用户代理必须**确保与**源服务器**的连接在发送第一个 `HTTP` 请求之前是端对端强加密的。

```
https-`URI` = "https:" "//" authority path-abempty [ "?" query
]
                    [ "#" fragment ]
```

要划重点的是，`HTTPS URI scheme` 同时基于 `TLS` 和 `TCP` 来建立连接。通过 `HTTPS scheme` 来访问的的资源并不与 `HTTP scheme` 共享，即使他们的资源标识指向同一个 `authority`（同一个 `host` 且监听在同一个 `TCP` 端口）。他们是不同的命名空间并且被认为是不同的**源服务器**。然而，被定义作用于整个 `host` 域的 `HTTP` 扩展，比如 `Cookie Protocol`，可以允许一个服务提供的信息被匹配为同一个主机域组的其它服务使用。

访问一个 `HTTPS` 标识资源的过程定义在 RFC-2818 中。

#### 2.7.3 `HTTP` 和 `HTTPS URI` 的正规化和匹配

因为 `HTTP` 和 `HTTPS schemes` 遵从于 `URI` 通用语构，所以其 `URIs` 的正规化和匹配都可以参照定义在 Section 6 of RFC-3986 中的算法，使用上面提到的每个 `scheme` 的默认描述。

如果 `URI` 中的端口等于该 `scheme` 的默认端口，则其标准形式就是忽略 `port` 子字段。当一个 `OPTIONS` 请求的请求目标没有使用 `absolute form`，一个空的 `path` 字段与 `absolute path` 的 `/` 等同，所以它的正规形式就是使用 `/` 作为 `path`。`scheme` 和 `host` 都是不区分大小写的，但常见为小写；其它的成分在匹配时都是区分大小写的。不在保留字符集中的字符等同于它们的百分号编码的的十六进制数：标准形式就是将他们解码（可参阅 Section 2.1 and 2.2 of RFC-3986）。

比如，下列三个 `URIs` 是等同的

```
http://example.com:80/~smith/home.html
http://EXAMPLE.com/%7Esmith/home.html
http://EXAMPLE.com:/%7esmith/home.html
```

## 3. 报文格式

所有的 HTTP/1.1 报文都由 `start-line`（起始行）开始，然后紧跟着与 `Internet Message Format` 格式类似的字节流：零个或多个头字段（统称为 `headers` 或 `header section`），一个空行标志着 `header section` 的结束，紧接着是可能存在的 `message body`（报文主体）。

```
HTTP-message   = start-line
                 *( header-field CRLF )
                 CRLF
                 [ message-body ]
```

解析 `HTTP` 报文的通常步骤是：读取 `start-line` 作为一个结构体，读出每一个头字段为一张哈希表，之后再利用已经解析到的数据来判断是否存在 `message body`。如果存在，则把它当做字节流读入，直到读入字节数等于 `message body` 的长度或者连接关闭。

任何**收信人必须**把 `HTTP` 报文作为一串以 `US-ASCII` 的超集编码的字节。把 `HTTP` 报文作为 `Unicode` 字符流而无视编码将因为不同的字符串处理库对待含有 LF(%x0A) 的非法多字节字符有不同的处理方法而导致安全漏洞。只有在协议元素已经从报文中解析出来以后，比如在某个头字段的值（该头字段已经被解析，比如存入了一张哈希表），基于字符串的解析器才是安全的。

一个 `HTTP` 报文可以在进行增量处理或者转发下游时作为 `stream`（流）来解析。当然，**收信人**不能依赖报文分片的增量运输，因为一些 `HTTP` 的具体实现会为了网络效率、安全检查或者有效载荷转换而缓冲或者延迟报文转发。

任何**发信人绝不能**在 `start-line` 和第一个头字段之间发送空格。在 `start-line` 和第一个头字段之间收到空格的**收信人必须**要么拒绝处理此报文，要么忽略以空格开头的行（即忽略第一行，然后把下一行作为第一行，直到收到一个有效的头字段或者整个 `header section` 结束）。

这种空格的存在可能是想尝试欺骗**服务器**，使其忽略这个头字段或者把这一行之后当做一个新的请求来处理，这些都可能会导致安全漏洞，如果请求链上的其它实现对同样的信息作出了不同的解释。同样，这种空格如果出现在响应报文中可能会被一些**客户端**忽略或导致其它的**客户端**停止解析。

## 3.1 起始行

一个 `HTTP` 报文既可以是从**客户端**到**服务器**的请求报文，也可以是**服务器**到**客户端**的响应报文。在语构上，这两种报文唯二的区别之一是 `start-line`：请求报

文是 `request-line`（请求行）而响应报文是 `status-line`（状态行），而另一个区别就是计算 `message body` 的算法不同 Section 3.3。

在理论上，一个**客户端**可以接收请求而一个**服务器**也可以接收响应，辨别他们全凭 `start-line`；但事实上，**服务器**会被实现为只接收请求（收到响应会被认为是未知或者不合法的请求方法），同样**客户端**被实现为只接收响应。

```
start-line     = request-line / status-line
```

### 3.1.1 请求行

一个 `request-line`（请求行）以一个 `method token` 开始，紧随其后的是一个单空格（`SP`）、`request-target`（请求目标）、另一个单空格（`SP`）、协议版本，最后以 `CRLF` 结尾。

```
request-line   = method SP request-target SP HTTP-version CRLF
```

`method token` 表示对目标资源进行操作的请求方法，它是区分大小写的。

```
method         = token
```

规范中定义的请求方法可以在 Section 4 of RFC-7231 中找到，后面还有关于 `HTTP` 方法注册和定义新方法的考量信息。

`request-target` 表示应用此请求的目标资源，如同 Section 5.3 中所定义的那样。

接受者一般把 `request-line` 根据空格分成三部分解析（参考 Section 3.5），因为这三个字段内都不允许出现空格。不幸的是，一些**用户代理**错误地编码或者去除在超文本中找到的空格，导致这些非法字符混入 `request-target` 中而被发送。

接收到非法 `request-line` 的**收信人应该**要么返回一个 `400 (Bad Request)` 错误，要么返回一个 `301 (Moved Permanently)` 以适当编码后的 `request-target` 重定向。任何**收信人不应该**尝试去自动纠正而不返回重定向，因为非法的 `request-line` 可能会被故意用来绕过请求链上的安全过滤器。

`HTTP` 没有在 `request-line` 的长度上施加任何预定义的限制，如同 Section 2.5 中所描述的那样。一个**服务器**收到一个比它实现的任何方法都要长的请求方

法时**应该**返回 `501 (Not Implemented)` 状态码。一个**服务器**收到一个比它所期望的任何 `URI` 都要长的 `request-target` 时**必须**返回 `414 (URI Too Long)` 状态码（可参阅 Section 6.5.12 of RFC-7231）。

在实际操作中，可以发现各种各样的对 `request-line` 的特定限制。我们**建议**所有的 `HTTP senders` 和**收信人**最少支持 8000 个字节长度的 `request-line`。

### 3.1.2 状态行

响应报文的第一行是 `status-line`（状态行），由协议版本、一个单空格（`SP`）、状态码、另一个空格（`SP`）、一个可能为空的状态码文本描述、一个 `CRLF` 依次排列构成。

```
status-line = HTTP-version SP status-code SP reason-phrase CRLF
```

状态码由三位十进制整数构成，用于描述**服务器**尝试理解并满足**客户端**相应的请求而返回结果。剩下的响应报文将根据状态码被定义的语义而被解析。可参阅 Section 6 of RFC 7231 来获取状态码的语义信息，包括状态码的类别（由以一个数字决定）、本规范定义的状态码、有关新状态码定义的考量以及 `IANA` 注册。

```
status-code    = 3DIGIT
```

`reason-phrase` 元素存在的唯一目的就是给状态码提供文本描述，基本上是因为早期的互联网应用协议多采用可交互式文本客户端。任何**客户端应该**忽略 `reason-phrase` 的内容。

```
reason-phrase  = *( HTAB / SP / VCHAR / obs-text )
```

## 3.2 头字段

每个头字段都由一个不区分大小写的字段名、一个冒号（`:`）、一个可选的空格、字段值、另一个可选的空格依次排列构成。

```
header-field   = field-name ":" OWS field-value OWS

field-name     = token
field-value    = *( field-content / obs-fold )
```

```
field-content  = field-vchar [ 1*( SP / HTAB ) field-vchar ]
field-vchar    = VCHAR / obs-text

obs-fold       = CRLF 1*( SP / HTAB )
               ; obsolete line folding
               ; see Section 3.2.4
```

字段名让响应的字段值有了该头字段所定义的语义。比如，`Date` 头字段被定义于 Section 7.1.1.2 of RFC 7231 ，它表示着 `HTTP` 报文的创建时间戳。

### 3.2.1 字段的可扩展性

头字段具有极高的可扩展性：引入新的字段名并为其定义大致的语义、在指定 `HTTP` 报文中被使用的头字段数量都是不受限的。已经存在的的字段被广泛定义于本规范的任何部分以及很多本文档集之外的其它规范中。

只要能得到接受者的理解，新的头字段可以这样被定义：他们可以覆盖或者改进原先已经被定义头字段的解释、定义请求评测的前提、或者重新定义响应的含义。

任何**代理必须**转发无法识别的头字段除非这个字段名被列举在 `Connection header field` （Section 6.1）中或者该**代理**被特意配置过以屏蔽或转换这些字段。其它的**收信人应该**忽略无法识别的头字段。这些要求允许增强 `HTTP` 的功能而无需预先更新已经部署的**中介**。

所有已定义的头字段应当在 `IANA` 的 `Message Headers` 中被注册，如同 Section 8.3 of RFC 7231 中所描述的那样。

### 3.2.2 字段顺序

根据收到字段名的头字段顺序并不是很重要。当然，先发含控制数据的头字段在实践中很有用，像请求 `Host` 头字段和响应的 `Date` 头字段，这样的话具体实现可以尽早决定是否要继续处理该报文。任何**服务器绝不能**在收到完整的 `header section` 之前对目标资源执行请求，因为之后收到的头字段可能会包含条件语句、认证凭证、或者有意误导的、会对请求处理产生冲击的重复头字段。

任何**发信人MUST MOT**创建重复字段名的头字段，除非该头字段所有的字段值被定义为逗号分隔的列表 [即 `#(values)` ] 或者该头字段是一个众所周知的例外。

任何**收信人可以**结合多个同名头字段为一个 `field-name: field-value` 对，通过将每个随后的字段值按照顺序并入合并的，用逗号分隔的字段值而不改变报文的语义。因此同名多字段的头字段的接收顺序对于合并值的解析是很重要的；任

何**代理**在转发报文时**绝不能**改变这些字段值的顺序。

> 在实践中，`Set-Cookie` 头字段 [RFC-6265] 经常在一个请求报文中出现多次而非使用列表语法，违背了上面所说的同名多字段的要求。但考虑到它不能被结合为一个单独的字段值，**收信人**在处理头字段时应当把 `Set-Cookie` 当做特例来处理。（参阅 Appendix A.2.3 of Kri2001 了解更多内容）

### 3.2.3 空白

本规范使用了三条规则来标识线性空格的使用：`OWS (optional whitespace)`，`RWS (required whitespace)`，`BWS ("bad" whitespace)` 。

`OWS` 规则表示可能会存在零个或多个线性空格。出于提高协议元素的可读性的目的，任何**发信人应该**在 `OWS` 的位置使用单空格；否则，任何**发信人不应该**生成 `OWS`，除非出于就绪报文（即待发送的报文）过滤时修正无效或不需要的协议元素的需求。

`RWS` 规则表示需要至少一个线性空格来分隔字段的 `tokens` 。一个**发信人应该**在 `RWS` 的位置生成单空格。

`BWS` 规则只在表示因历史原因而在语法上允许的 `OWS` 而被使用。任何**发信人绝不能**在报文中生成 `BWS` 。任何**收信人必须**在解读该协议元素之前解析并移除这样的 `BWS` 。

```
OWS            = *( SP / HTAB )
               ; optional whitespace
RWS            = 1*( SP / HTAB )
               ; required whitespace
BWS            = OWS
               ; "bad" whitespace
```

### 3.2.4 字段解析

报文解析使用一种通用算法，与特定的头字段名无关。特定字段值的内容在报文解析的下一阶段之前都不会被解析（通常在整个 `header section` 都已经被处理了）。所以，本规范不会像前几个版本一样使用 `ABNF` 规则来定义每一对 `Field-Name: Field Value` 。取而代之的是，我们会根据每一个注册的字段名的 `ABNF` 规则来定义响应字段值的有效语法（即在字段值被通用字段解析器从 `header section` 提取出来之后）。

头字段名和冒号之间不允许出现空格。在以前，以不同方法的处理此类空格导致

请求路由和处理响应时存在安全漏洞。任何**服务器必须**拒绝任何收到的，在头字段和冒号之间存在空格的请求报文并返回 `400（Bad Request）`状态码。任何**代理必须**在转发响应报文到下游时移除此类空格。

一个字段值的前后都有 `OWS`；字段值之前有一个单空格可以增加人的可读性。字段值的开头结尾都不能存在空格：第一个非空格字节之前和最后一个非空格字节之后的的空格在从头字段中解析字段值的时候应当被解析器移除。

由于历史原因， `HTTP` 头字段值可以被延长为多行，通过在每一个额外行前面加上至少一个空格或水平 `tab` （`obs-fold`）。本规范弃用了不在 `message/HTTP media type` （Section 8.3.1）中的 `line folding`。任何**发信人绝不能**生成一个包含 `line folding` 的报文（即，不存在能匹配 obs-fold 规则的头字段值）除非该报文中包含 `message/HTTP media type` 的头字段。

一个**服务器**如果收到了非 `message/HTTP media type`，但匹配了 `obs-fold` 规则的头字段值，它**必须**要么拒绝该报文并返回 `400（Bad Request）`，最好附带一份说明来解释这个已经被废弃的 `line folding` 无法被接受；要么在解析该字段值或转发下游之前，使用一个或多个空格来替换每个收到的 `obs-fold`。

一个**代理**或**网关**如果收到含有非 `message/HTTP media type`，但匹配了 `obs-fold` 规则的头字段值的响应报文，它**必须**要么丢弃此报文并返回 `502（Bad Gateway）`，最好附带一份说明来解释这个已经被废弃的 `line folding` 无法被接受；要么在解析该字段值或转发下游之前，使用一个或多个空格来替换每个收到的 `obs-fold`。

一个**用户代理**如果收到含有非 `message/HTTP media type`，但匹配了 `obs-fold` 规则的头字段值的响应报文，它**必须**在解析该字段值之前使用一个或多个空格来替换每个收到的 `obs-fold`。

由于历史原因， `HTTP` 曾允许字段文本内容使用 `ISO-8859-1` 字符集，其它字符集必须使用 RFC-2047 编码。但在实践中，大多数的 `HTTP` 头字段值只使用 `US-ASCII` 字符集 的子集。新定义的头字段**应该**限制它们的字段值只使用 `US-ASCII`。一个**收信人应该**把字段内容中使用其它字符集的字节（`obs-text`）当做 `opaque data`（透明数据）来对待。

### 3.2.5 字段限制

`HTTP` 并没有对每个头字段或者整个 `header section` 预定义任何长度限制，如同 Section2.5 中所描述的那样。但在实践中我们可以发现各种对一些特定头字段长度的特别限制，常常由该字段的语义而定。

收到一个或多个过长（超出预期长度）的请求头字段的**服务器必须**返回一个合适的 `4xx (Client Error)` 状态码。忽略此类头字段可能会造成服务器端漏洞并增加 `request smuggling attacks` （Section 9.5）发生的风险。

任何**客户端可以**丢弃或者截断过长的响应头字段，如果该字段的语义允许我们可以在不改变报文结构和响应语义的情况下安全地忽略被丢弃的值。

### 3.2.6 字段值的构成

大部分 `HTTP` 头字段值使用被空格或其他分隔符隔开的常见语法结构定义（`token`、`quoted-string`、和 `comment`）。分隔符集合是 `US-ASCII` 可见字符集中不能出现在 `token` 中的子集（`DQUOTE` 和 `" ( ) , / : ; \ < \ = \ > ? @ [ \ \ ] { } "`）。

```
token          = 1*tchar

tchar          = "!" / "#" / "$" / "%" / "&" / "'" / "*"
               / "+" / "-" / "." / "^" / "_" / "`" / "|" / "~"
               / DIGIT / ALPHA
               ; any VCHAR, except delimiters
```

双引号围起来的文本字符串会被解析为单个值。

```
quoted-string  = DQUOTE *( qdtext / quoted-pair ) DQUOTE
qdtext         = HTAB / SP /%x21 / %x23-5B / %x5D-7E / obs-text
obs-text       = %x80-FF
```

注释允许以括号括起的形式被包含在头字段中。注释只被允许出现在字段值定义中包含 `comment` 字段里。

```
comment        = "(" *( ctext / quoted-pair / comment ) ")"
ctext          = HTAB / SP / %x21-27 / %x2A-5B / %x5D-7E / obs-
text
```

反斜杠（`\`）可以在 `quoted-string` 或者 `comment` 结构中用作一种 `single-octet quoting mechanism`（单字节转义机制，应该就是指转义？）。处理 `quoted-string` 值的**收信人必须**把 `quoted-pair` 替换为反斜杠后面的字符来处理。

```
quoted-pair    = "\" ( HTAB / SP / VCHAR / obs-text )
```

任何**发信人不应该**在 `quoted-string` 中生成 `quoted-pair` 除非有必要在字符串中转义 `DQUOTE` 和 `backslash` 。任何**发信人不应该**在 `comment` 中生成 `quoted-pair` 除非有必要在 `comment` 中转义括号【`(` 和 `)`】和 `backslash` 。

## 3.3 报文主体

一个 `HTTP` 报文的 `message body` （报文主体，如果有的话）被用来承载该请求或相应的 `payload body` （有效载荷主体）。一般来说报文主体等同于有效载荷，除非该报文使用了 `transfer coding` （传输编码），如同 Section 3.3.1 中描述的那样。

```
message-body = *OCTET
```

这条规则适用于报文中允许报文主体的情况，无论是在请求时还是相应时。

请求中存在消息主体的标识是 `Content-Length` 或 `Transfer-Encoding` 头字段。请求报文的构成独立于请求方法的语义，即使该方面没有定义任何有关消息体使用的内容。

响应中是否含有报文主体取决于它所响应请求的请求方法和该响应的状态码（Section 3.1.1）。对 `HEAD` 请求方法（Section 4.3.2 of RFC-7231）的响应不能包含报文主体，因为与其相关的头字段（比如 `Transfer-Encoding` , `Content-Length` 等）一旦出现也只能表示，当该请求方法为 `GET` （Section 4.3.1 of RFC-7231）时该头字段的值。对 `CONNECT` 请求方法（Section 4.3.6 of RFC-7231）的 `2xx (Successful)` 响应要切换到 `tunnel` mode（隧道模式）而不能含有消息主体。所有的 `1xx (Informational)` , `204 (No Content)` , 和 `304 (Not Modified)` 响应都不能含有消息主体。其它所有的响应都含有消息主体，虽然主体的长度可能为0。

### 3.3.1 Transfer-Encoding

`Transfer-Encoding` 头字段列出了有效载荷主体形成消息主体所使用（或将要使用）传输编码的编码名。传输编码定义在 Section 4

```
Transfer-Encoding = 1#transfer-coding
```

`Transfer-Encoding` 类似于在 `MIME` 中被用于在7位传输服务（RFC2045, Section 6）上安全传输二进制数据的 `Content-Transfer-Encoding` 字段。当然，想要在能正确处理 8 位字符编码的传输协议上保证安全传输则需要注重不同的关键点。对 `HTTP` 来说， `Transfer-Encoding` 主要是用来准确地划分动态生成的有效载荷，以及区分出那些仅用于提升传输效率和安全性而非出于所选资源的特点才使用的有效载荷编码。

任何**收信人必须**能解析 `chunked` 传输编码，因为该编码在有效载荷主体大小无法预知时的报文构成中扮演着极其重要的角色。一个**发信人绝不能**对一个报文主体使用多次 `chunked` （即不允许分块一个已经被分块的报文）。如果一个请求的有效载荷体除了 `chunked` 以外使用了其它的传输编码，那该**发信人必须**使用 `chunked` 作为最后一次编码以确保报文的正确形成。如果一个响应的有效载荷体除了 `chunked` 以外使用了其它的传输编码，那么该**发信人必须**要么使用 `chunked` 作为最后一次编码，要么使用关闭连接的方式来终止该报文。

比如

```
Transfer-Encoding: gzip, chunked
```

表示该有效载荷主体在形成报文主体时先使用了 `gzip` 编码压缩然后再使用 `chunked` 编码进行分片。

与 `Content-Encoding` （Section 3.1.2.1 of RFC7231）不同， `Transfer-Encoding` 是报文的属性而非内容的属性，而且请求/响应链上的任何一个**收信人可以**按照传输编码来解码报文主体或者使用额外的传输编码，当然 `Transfer-Encoding` 的字段值也应发生相应的改变。关于编码参数的额外信息可由本篇规范以外定义的其它头字段提供。

对一个 `HEAD` 请求的响应或者对一个 `GET` 请求的 `304(Not Modified)` 响应都**可以**含有 `Transfer-Encoding` 头字段，虽然它们都不包含报文主体，但该头字段可以表示当该请求是一个非条件 `GET` （`unconditional GET`）时报文主体将会使用的传输编码。当然，由于响应链上的任何 `receipient` （包括**源服务器**）都可以把不需要的传输编码移除，本规范不对这种标识作任何要求。

任何**服务器绝不能**在任何状态码为 `1xx (Informational)` or `204 (No Content)` 的响应中发送 `Transfer-Encoding` 头字段。任何**服务器绝不能**在任何状态码为 `2xx (Successful)` 的对 `CONNECT` 请求（Section 4.3.6 of RFC7231）的响应中发送 `Transfer-Encoding` 头字段。

`Transfer-Encoding` 头字段添加于 `HTTP/1.1`。我们大致可以认为只有仅支持 `HTTP/1.0` 的具体实现会无法理解怎样去处理使用了传输编码的有效载荷。任何**客户端绝不能**发送含有 `Transfer-Encoding` 的请求，除非它知道**服务器**能处理 `HTTP/1.1`（或更高版本）的请求；这样的信息可能来源于用户配置或者先前收到的响应报文。任何**服务器绝不能**发送包含 `Transfer-Encoding` 的响应，除非对应的请求表明了协议版本为 `HTTP/1.1`（或更高版本）。

任何收到传输编码无法理解的的报文的**服务器**都**应该**返回一个 `501 (Not Implement)` 响应。