

---

# Table of Contents

Introduction	1.1
Introduction / 前言	1.2
HTTP/2 Protocol Overview / HTTP/2概述	1.3
Document Organization / 文档结构	1.3.1
Conventions and Terminology / 约定和术语	1.3.2
Starting HTTP/2 / 开始HTTP/2	1.4
HTTP/2 Version Identification / HTTP/2版本标识	1.4.1
Starting HTTP/2 for "http" URIs / 为"http" URIs启用HTTP/2协议	1.4.2
HTTP2-Settings Header Field / HTTP2-Settings首部字段	1.4.2.1
Starting HTTP/2 for "https" URIs / 为"https" URIs启用HTTP/2协议	1.4.3
Starting HTTP/2 with Prior Knowledge / 先验下启用HTTP/2	1.4.4
HTTP/2 Connection Preface / HTTP/2连接前奏	1.4.5
HTTP Frames / HTTP帧	1.5
Frame Format / 帧格式	1.5.1
Frame Size / 帧大小	1.5.2
Header Compression and Decompression / 首部压缩与解压缩	1.5.3
Streams and Multiplexing	1.6
Stream States	1.6.1
Stream Identifiers	1.6.1.1
Stream Concurrency	1.6.1.2
Flow Control	1.6.2
Flow-Control Principles	1.6.2.1
Appropriate Use of Flow Control	1.6.2.2
Stream Priority	1.6.3
Stream Dependencies	1.6.3.1
Dependency Weighting	1.6.3.2
Reprioritization	1.6.3.3
Prioritization State Management	1.6.3.4
Default Priorities	1.6.3.5
Error Handling	1.6.4

---

Connection Error Handling	1.6.4.1
Stream Error Handling	1.6.4.2
Connection Termination	1.6.4.3
Extending HTTP/2	1.6.5
Frame / 帧定义	1.7
DATA / DATA 帧	1.7.1
HEADERS / HEADERS 帧	1.7.2
PRIORITY / PRIORITY 帧	1.7.3
RST_STREAM / RST_STREAM 帧	1.7.4
SETTINGS / SETTINGS 帧	1.7.5
SETTINGS Format /	1.7.5.1
Defined SETTINGS Parameters /	1.7.5.2
Settings Synchronization /	1.7.5.3
PUSH_PROMISE / PUSH_PROMISE 帧	1.7.6
PING / PING 帧	1.7.7
GOAWAY / GOAWAY 帧	1.7.8
WINDOW_UPDATE / WINDOW_UPDATE 帧	1.7.9
The Flow-Control Window /	1.7.9.1
Initial Flow-Control Window Size /	1.7.9.2
Reducing the Stream Window Size /	1.7.9.3
CONTINUATION / CONTINUATION 帧	1.7.10
Error Codes / 错误码	1.8
HTTP Request/Response Exchange	1.9
HTTP Request/Response Exchange	1.9.1
Upgrading from HTTP/2	1.9.1.1
HTTP Header Fields	1.9.1.2
Pseudo-Header Fields	1.9.1.2.1
Connection-Specific Header Fields	1.9.1.2.2
Request Pseudo-Header Fields	1.9.1.2.3
Response Pseudo-Header Fields	1.9.1.2.4
Compressing the Cookie Header Field	1.9.1.2.5
Malformed Requests and Responses	1.9.1.2.6
Examples	1.9.1.3
Request Reliability Mechanisms in HTTP/2	1.9.1.4

---

---

Server Push	1.9.2
Push Requests	1.9.2.1
Push Responses	1.9.2.2
The CONNECT Method	1.9.3
Additional HTTP Requirements/Considerations	1.10
Connection Management	1.10.1
Connection Reuse	1.10.1.1
The 421 (Misdirected Request) Status Code	1.10.1.2
Use of TLS Features	1.10.2
TLS 1.2 Features	1.10.2.1
TLS 1.2 Cipher Suites	1.10.2.2
Security Considerations	1.11
Server Authority	1.11.1
Cross-Protocol Attacks	1.11.2
Intermediary Encapsulation Attacks	1.11.3
Cacheability of Pushed Responses	1.11.4
Denial-of-Service Considerations	1.11.5
Limits on Header Block Size	1.11.5.1
CONNECT Issues	1.11.5.2
Use of Compression	1.11.6
Use of Padding	1.11.7
Privacy Considerations	1.11.8
IANA Considerations	1.12
Registration of HTTP/2 Identification Strings	1.12.1
Frame Type Registry	1.12.2
Settings Registry	1.12.3
Error Code Registry	1.12.4
HTTP2-Settings Header Field Registration	1.12.5
PRI Method Registration	1.12.6
The 421 (Misdirected Request) HTTP Status Code	1.12.7
The h2c Upgrade Token	1.12.8
References	1.13
Normative References	1.13.1

---

---

Informative References	1.13.2
A. TLS 1.2 Cipher Suite Black List	1.14
Acknowledgements	1.15
Authors' Addresses	1.16

译者：quafoo

授权：[署名-非商用许可证](#)

本文档尝试将HTTP/2协议[RFC 7540](#)翻译为中文。为了方便研究原文，遂采用中英文对照的形式展示。

## 版权许可

本书采用“保持署名—非商用”[创意共享4.0](#)许可证。

只要保持原作者署名和非商用，您可以自由地阅读、分享、修改本书。

详细的法律条文请参见[创意共享](#)网站。

## 编辑流程

- 首先请安装 [Gitbook](#) 并参考其文档进行编辑。
- 添加或修改内容之后，可以进入 `source` 目录并执行 `gitbook serve`，然后在浏览器中即时预览结果。
- 随时欢迎 `issue` 和 `PR`。

# 1. Introduction / 前言

The Hypertext Transfer Protocol (HTTP) is a wildly successful protocol. However, the way HTTP/1.1 uses the underlying transport ([RFC7230], Section 6) has several characteristics that have a negative overall effect on application performance today.

超文本传输协议(HTTP)是一个非常成功的协议。但是，HTTP/1.1使用下层传输层([\[RFC7230\]](#)，[第6章](#))的方式所具有的一些特性，对如今的应用层性能产生了一个全面的消极的影响。

In particular, HTTP/1.0 allowed only one request to be outstanding at a time on a given TCP connection. HTTP/1.1 added request pipelining, but this only partially addressed request concurrency and still suffers from head-of-line blocking. Therefore, HTTP/1.0 and HTTP/1.1 clients that need to make many requests use multiple connections to a server in order to achieve concurrency and thereby reduce latency.

尤其是，在一个给定的TCP连接上，HTTP/1.0只允许发送一次，每次一个请求。HTTP/1.1管线了管道技术，但这只是部分地实现了并发请求，并且仍然存在着线头阻塞问题。因此，HTTP/1.0和HTTP/1.1的用户要发送大量请求时，就使用多个连接来达到并发和减少延迟的目的。

Furthermore, HTTP header fields are often repetitive and verbose, causing unnecessary network traffic as well as causing the initial TCP [TCP] congestion window to quickly fill. This can result in excessive latency when multiple requests are made on a new TCP connection.

此外，HTTP首部字段常常是重复和冗余的，从而产生了不必要的网络流量，也导致初始TCP拥塞窗口被快速填满。当在一个新建立的TCP连接上发送多个请求时，这会产生极大的时延。

HTTP/2 addresses these issues by defining an optimized mapping of HTTP's semantics to an underlying connection. Specifically, it allows interleaving of request and response messages on the same connection and uses an efficient coding for HTTP header fields. It also allows prioritization of requests, letting more important requests complete more quickly, further improving performance.

通过定义一个优化的HTTP语义到底层连接的映射，HTTP/2解决了这些问题。具体来说，它允许在同一个连接上请求和响应消息交错往来，并且使用了高效的HTTP首部字段编码。它还允许请求具有优先级，让比较重要的请求可以更快地完成，从而进一步改善了性能。

The resulting protocol is more friendly to the network because fewer TCP connections can be used in comparison to HTTP/1.x. This means less competition with other flows and longer-lived connections, which in turn lead to better utilization of available network capacity.

相比HTTP/1.x，HTTP/2使用的TCP连接更少，对网络更友好。这意味着，网络流之间的竞争减少了，连接的生存期延长了，从而相应地提高了可用的网络资源的利用率。

Finally, HTTP/2 also enables more efficient processing of messages through use of binary message framing.

最后，通过使用二进制消息帧，HTTP/2也提高了消息的处理效率。

## 2. HTTP/2 Protocol Overview / HTTP/2概述

HTTP/2 provides an optimized transport for HTTP semantics. HTTP/2 supports all of the core features of HTTP/1.1 but aims to be more efficient in several ways.

针对HTTP语义，HTTP/2提供了一种优化的传输机制。HTTP/2支持HTTP/1.1所有的核心特性，但在一下几个方面更有效率。

The basic protocol unit in HTTP/2 is a frame ([Section 4.1](#)). Each frame type serves a different purpose. For example, [HEADERS](#) and [DATA](#) frames form the basis of HTTP requests and responses ([Section 8.1](#)); other frame types like [SETTINGS](#), [WINDOW\\_UPDATE](#), and [PUSH\\_PROMISE](#) are used in support of other HTTP/2 features.

帧([4.1节](#))是HTTP/2里基本的协议单元。每个类型的帧都服务于不同的目的。例如：[HEADERS](#) 帧和 [DATA](#) 帧形成了HTTP请求和响应([8.1节](#))的主体；其他的诸如 [SETTINGS](#) 、 [WINDOW\\_UPDATE](#) 和 [PUSH\\_PROMISE](#) 等帧类型，用来支持HTTP/2的其他特性。

Multiplexing of requests is achieved by having each HTTP request/response exchange associated with its own stream ([Section 5](#)). Streams are largely independent of each other, so a blocked or stalled request or response does not prevent progress on other streams.

每对HTTP请求/响应的交换都和自己的流([第5章](#))相关联，从而实现了请求的多路复用。流之间是相互独立的，所以阻塞或停止请求或响应不会中断其他的流。

Flow control and prioritization ensure that it is possible to efficiently use multiplexed streams. Flow control ([Section 5.2](#)) helps to ensure that only data that can be used by a receiver is transmitted. Prioritization ([Section 5.3](#)) ensures that limited resources can be directed to the most important streams first.

流量控制和优先级机制确保能够高效地利用多路复用的流。流量控制([5.2节](#))有助于确保只传输接收方能够使用的数据。优先级机制([5.3节](#))确保有限的资源首先被用于最重要的流。

HTTP/2 adds a new interaction mode whereby a server can push responses to a client ([Section 8.2](#)). Server push allows a server to speculatively send data to a client that the server anticipates the client will need, trading off some network usage against a potential latency gain. The server does this by synthesizing a request, which it sends as a [PUSH\\_PROMISE](#) frame. The server is then able to send a response to the synthetic request on a separate stream.



HTTP/2增加了一种新的交互模式，即服务器能向客户端推送响应( 8.2节 )。服务器推送功能允许服务器自主地向客户端发送它预期客户端会需要的数据，从而消耗一些网络带宽来取代潜在的网络延迟。服务器先综合一个PUSH\_PROMISE帧携带的请求，然后在一个独立的流上发送响应。

Because HTTP header fields used in a connection can contain large amounts of redundant data, frames that contain them are compressed (Section 4.3). This has especially advantageous impact upon request sizes in the common case, allowing many requests to be compressed into one packet.

因为连接中使用的HTTP首部字段包含了大量的冗余数据，所以包含这些首部的帧都被压缩了( 4.3节 )。一般情况下，请求能显著地被压缩，从而允许多个请求被压缩到一个包里。

## 2.1 Document Organization / 文档结构

The HTTP/2 specification is split into four parts:

- Starting HTTP/2 ([Section 3](#)) covers how an HTTP/2 connection is initiated.
- The frame ([Section 4](#)) and stream ([Section 5](#)) layers describe the way HTTP/2 frames are structured and formed into multiplexed streams.
- Frame ([Section 6](#)) and error ([Section 7](#)) definitions include details of the frame and error types used in HTTP/2.
- HTTP mappings ([Section 8](#)) and additional requirements ([Section 9](#)) describe how HTTP semantics are expressed using frames and streams.

HTTP/2规范分为四个部分：

- 开始HTTP/2一节( [第3章](#) )涵盖了怎样初始化一个HTTP/2连接。
- 帧层( [第4章](#) )和流层( [第5章](#) )两章描述了HTTP/2帧怎样被构造并形成多路复用的流。
- 帧定义( [第6章](#) )和错误定义( [第7章](#) )包含了HTTP/2的帧和错误类型的细节。
- HTTP映射( [第8章](#) )与附加要求( [第9章](#) )描述了HTTP语义是怎样用帧和流来表达的。

While some of the frame and stream layer concepts are isolated from HTTP, this specification does not define a completely generic frame layer. The frame and stream layers are tailored to the needs of the HTTP protocol and server push.

尽管一些帧和流层的概念与HTTP是相脱离的，但本规范没有定义一个完全通用的帧层。这些帧和流层是根据HTTP协议和服务端推送的需求量身定制的。

## 2.2 Conventions and Terminology / 约定和术语

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [\[RFC2119\]](#).

本档中出现的关键词"MUST","MUST NOT","REQUIRED","SHALL","SHALL NOT","SHOULD","SHOULD NOT","RECOMMENDED","MAY",和"OPTIONAL"在RFC 2119 [\[RFC2119\]](#) 中有解释。

All numeric values are in network byte order. Values are unsigned unless otherwise indicated. Literal values are provided in decimal or hexadecimal as appropriate. Hexadecimal literals are prefixed with 0x to distinguish them from decimal literals.

所有数值都以网络字节序表示。除非另有说明，否则数值都是无符号的。视情况以十进制或十六进制表示字面值。十六进制值带有0x前缀，以和十进制值区分开。

The following terms are used:

- **client**: The endpoint that initiates an HTTP/2 connection. Clients send HTTP requests and receive HTTP responses.
- **connection**: A transport-layer connection between two endpoints.
- **connection error**: An error that affects the entire HTTP/2 connection.
- **endpoint**: Either the client or server of the connection.
- **frame**: The smallest unit of communication within an HTTP/2 connection, consisting of a header and a variable-length sequence of octets structured according to the frame type.
- **peer**: An endpoint. When discussing a particular endpoint, "peer" refers to the endpoint that is remote to the primary subject of discussion.
- **receiver**: An endpoint that is receiving frames.
- **sender**: An endpoint that is transmitting frames.
- **server**: The endpoint that accepts an HTTP/2 connection. Servers receive HTTP requests and send HTTP responses.
- **stream**: A bidirectional flow of frames within the HTTP/2 connection.
- **stream error**: An error on the individual HTTP/2 stream.

文中使用的术语包括：

- **客户端**：发起HTTP/2连接的端点。客户端发送HTTP请求，并接收HTTP响应。

- 连接：两个端点之间的传输层连接。
- 连接错误：影响整个HTTP/2连接的错误。
- 端点：连接中的客户端或服务器。
- 帧：HTTP/2连接中的最小通信单元，由首部和可变长度的字节序列组成，其结构取决于帧类型。
- 对端：一个端点。当讨论一个特定的端点时，『对端』指的是其远程端点。
- 接收端：接收帧的端点。
- 发送端：发送帧的端点。
- 服务端：接受HTTP/2连接请求的端点。服务端接收HTTP请求，并发送HTTP响应。
- 流：HTTP/2连接中的双向帧传输流。
- 流错误：发生在单独的HTTP/2流上的错误。

Finally, the terms "gateway", "intermediary", "proxy", and "tunnel" are defined in [Section 2.3 of \[RFC 7230\]](#). Intermediaries act as both client and server at different times.

最后，『网关』、『中介』、『代理』和『隧道』等术语都在 [\[RFC 7230\]](#) 的 2.3节 中有定义。在不同的时候，中介既可以是客户端，又可以是服务端。

The term "payload body" is defined in [Section 3.3 of \[RFC 7230\]](#).

术语『有效载荷体』在 [\[RFC 7230\]](#) 的 3.3节 中有定义。

# Starting HTTP/2 / 开始HTTP/2

An HTTP/2 connection is an application-layer protocol running on top of a TCP connection ([TCP](#)). The client is the TCP connection initiator.

HTTP/2是运行于TCP连接([TCP](#))之上的应用层协议。客户端是TCP连接的发起者。

HTTP/2 uses the same "http" and "https" URI schemes used by HTTP/1.1. HTTP/2 shares the same default port numbers: 80 for "http" URIs and 443 for "https" URIs. As a result, implementations processing requests for target resource URIs like <http://example.org/foo> or <https://example.com/bar> are required to first discover whether the upstream server (the immediate peer to which the client wishes to establish a connection) supports HTTP/2.

HTTP/2使用与HTTP/1.1相同的URI方案"http"和"https"，共享同样的默认端口号："http"的80端口和"https"的443端口。因此，在处理对例如<http://example.org/foo> 或 <https://example.com/bar> 目标资源URIs的请求前，需要首先确定上游服务端(当前客户端希望直接与之建立连接的对端)是否支持HTTP/2。

The means by which support for HTTP/2 is determined is different for "http" and "https" URIs. Discovery for "http" URIs is described in Section 3.2. Discovery for "https" URIs is described in Section 3.3.

检测"http"和"https"的URIs是否支持HTTP/2的方法是不一样的。检测"http"的URIs在3.2节中描述。检测"https"的URIs在3.3节中描述。

# HTTP/2 Version Identification / HTTP/2版本标识

The protocol defined in this document has two identifiers.

- The string "h2" identifies the protocol where HTTP/2 uses [Transport Layer Security \(TLS\)](#) [TLS12]. This identifier is used in the [TLS application-layer protocol negotiation \(ALPN\) extension](#) [TLS-ALPN] field and in any place where HTTP/2 over TLS is identified.

The "h2" string is serialized into an ALPN protocol identifier as the two-octet sequence: 0x68, 0x32.

- The string "h2c" identifies the protocol where HTTP/2 is run over cleartext TCP. This identifier is used in the HTTP/1.1 Upgrade header field and in any place where HTTP/2 over TCP is identified.

The "h2c" string is reserved from the ALPN identifier space but describes a protocol that does not use TLS.

本文档定义的协议有两个标识符：

- 字符串"h2"表示HTTP/2协议使用了 [安全传输层协议\(TLS\)](#) [TLS12]。该标识符用在 [TLS应用层协议协商\(ALPN\)扩展](#) [TLS-ALPN]字段，以及任何在TLS之上运行HTTP/2的场合。

"h2"字符串被序列化成一个ALPN协议标识符，其形式是两个字节的序列：0x68，0x32。

- 字符串"h2c"表示HTTP/2协议运行在明文TCP上。该标识符用在HTTP/1.1的Upgrade首部字段，以及任何在TCP之上运行HTTP/2的场合。

"h2c"字符串是ALPN标识符空间预留的，但是用来表示不使用TLS的协议。

Negotiating "h2" or "h2c" implies the use of the transport, security, framing, and message semantics described in this document.

"h2"或"h2c"协商需要用到本文档里描述的传输层、安全、帧和消息语义等概念。

# Starting HTTP/2 for "http" URIs / 为"http" URIs启用HTTP/2协议

A client that makes a request for an "http" URI without prior knowledge about support for HTTP/2 on the next hop uses the HTTP Upgrade mechanism ([Section 6.7 of \[RFC7230\]](#)). The client does so by making an HTTP/1.1 request that includes an Upgrade header field with the "h2c" token. Such an HTTP/1.1 request MUST include exactly one HTTP2-Settings ([Section 3.2.1](#)) header field.

For example:

```
GET / HTTP/1.1
Host: server.example.com
Connection: Upgrade, HTTP2-Settings
Upgrade: h2c
HTTP2-Settings: <base64url encoding of HTTP/2 SETTINGS payload>
```

事先不知道下一跳是否支持HTTP/2的客户端，使用HTTP的Upgrade机制( [\[RFC7230\]](#) 的 6.7 节 )发起"http"URI请求。其做法是：客户端先发起HTTP/1.1请求，该请求包含值为"h2c"的Upgrade首部字段，还必须包含一个HTTP2-Settings( [3.2.1节](#) )首部字段。

例如：

```
GET / HTTP/1.1
Host: server.example.com
Connection: Upgrade, HTTP2-Settings
Upgrade: h2c
HTTP2-Settings: <base64url encoding of HTTP/2 SETTINGS payload>
```

Requests that contain a payload body MUST be sent in their entirety before the client can send HTTP/2 frames. This means that a large request can block the use of the connection until it is completely sent.

在客户端能发送HTTP/2帧之前，包含负载的请求必须全部被发送。这意味着一个大的请求能阻塞连接的使用，直到其全部被发送完毕。

If concurrency of an initial request with subsequent requests is important, an OPTIONS request can be used to perform the upgrade to HTTP/2, at the cost of an additional round trip.

如果一个初始请求的后续并发请求很重要，那么可以使用OPTIONS请求来执行升级到HTTP/2的操作，代价是一个额外的往返。

A server that does not support HTTP/2 can respond to the request as though the Upgrade header field were absent:

```
HTTP/1.1 200 OK
Content-Length: 243
Content-Type: text/html
...
```

不支持HTTP/2的服务端对请求进行响应时可以忽略Upgrade首部字段：

```
HTTP/1.1 200 OK
Content-Length: 243
Content-Type: text/html
...
```

A server MUST ignore an "h2" token in an Upgrade header field. Presence of a token with "h2" implies HTTP/2 over TLS, which is instead negotiated as described in [Section 3.3](#).

服务端必须忽略值为"h2"的Upgrade首部字段。"h2"字段表示HTTP/2使用了TLS，其协商方法在 [3.3节](#) 中描述。

A server that supports HTTP/2 accepts the upgrade with a 101 (Switching Protocols) response. After the empty line that terminates the 101 response, the server can begin sending HTTP/2 frames. These frames MUST include a response to the request that initiated the upgrade.

For example:

```
HTTP/1.1 101 Switching Protocols
Connection: Upgrade
Upgrade: h2c

[ HTTP/2 connection ...
```

支持HTTP/2的服务端返回101(Switching Protocols)响应，表示接受升级协议的请求。在结束101响应的空行之后，服务端可以开始发送HTTP/2帧。这些帧必须包含一个对升级请求的响应。

例如：



```
HTTP/1.1 101 Switching Protocols
Connection: Upgrade
Upgrade: h2c

[ HTTP/2 connection ...
```

The first HTTP/2 frame sent by the server MUST be a server connection preface (Section 3.5) consisting of a **SETTINGS** frame (Section 6.5). Upon receiving the 101 response, the client MUST send a connection preface (Section 3.5), which includes a **SETTINGS** frame.

服务端发送的第一个HTTP/2帧必须是由一个 **SETTINGS** 帧( 6.5节 )组成的服务端连接前奏( 3.5节 )。客户端一收到101响应，也必须发送一个包含 **SETTINGS** 帧的连接前奏( 3.5节 )。

The HTTP/1.1 request that is sent prior to upgrade is assigned a stream identifier of 1 (see Section 5.1.1) with default priority values (Section 5.3.5). Stream 1 is implicitly "half-closed" from the client toward the server (see Section 5.1), since the request is completed as an HTTP/1.1 request. After commencing the HTTP/2 connection, stream 1 is used for the response.

升级之前发送的HTTP/1.1请求被分配一个流标识符1 (参见 5.1.1节)，并被赋予默认优先级值( 5.3.5节 )。流1暗示从客户端到服务端(参见 5.1节)是半关闭的，因为作为HTTP/1.1请求它已经完成了。HTTP/2连接开始后，流1用于响应。

# HTTP2-Settings Header Field / HTTP2-Settings首部字段

A request that upgrades from HTTP/1.1 to HTTP/2 MUST include exactly one HTTP2-Settings header field. The HTTP2-Settings header field is a connection-specific header field that includes parameters that govern the HTTP/2 connection, provided in anticipation of the server accepting the request to upgrade.

```
HTTP2-Settings    = token68
```

从HTTP/1.1升级到HTTP/2的请求必须确切地包含一个HTTP2-Settings首部字段。HTTP2-Settings首部字段是一个专用于连接的首部字段，它包含管理HTTP/2连接的参数，其前提是假设服务端会接受升级请求。

```
HTTP2-Settings    = token68
```

A server MUST NOT upgrade the connection to HTTP/2 if this header field is not present or if more than one is present. A server MUST NOT send this header field.

如果该首部字段没有出现，或者出现了不止一个，那么服务端一定不要把连接升级到HTTP/2。服务端一定不要发送该首部字段。

The content of the HTTP2-Settings header field is the payload of a [SETTINGS](#) frame ([Section 6.5](#)), encoded as a base64url string (that is, the URL- and filename-safe Base64 encoding described in [Section 5](#) of [\[RFC4648\]](#), with any trailing '=' characters omitted). The [ABNF \[RFC5234\]](#) production for token68 is defined in [Section 2.1](#) of [\[RFC7235\]](#).

HTTP2-Settings首部字段的值是 [SETTINGS](#) 帧 ([6.5节](#)) 的有效载荷被编码成的base64url串 (即，[\[RFC4648\]](#) 的 [第5节](#) 描述的URL-和文件名安全Base64编码，忽略任何拖尾'='字符)。ABNF [\[RFC5234\]](#) 产生token68在 [\[RFC7235\]](#) 的 [2.1节](#) 有定义。

Since the upgrade is only intended to apply to the immediate connection, a client sending the HTTP2-Settings header field MUST also send HTTP2-Settings as a connection option in the Connection header field to prevent it from being forwarded (see [Section 6.1](#) of [\[RFC7230\]](#)).

因为升级操作只适用于相邻端点的直连，发送HTTP2-Settings首部字段的客户端也必须在发送的Connection首部字段值里加上HTTP2-Settings选项，以阻止它被转发(参见 [\[RFC7230\]](#) 的 [6.1节](#))。

A server decodes and interprets these values as it would any other **SETTINGS** frame. Explicit acknowledgement of these settings (Section 6.5.3) is not necessary, since a 101 response serves as implicit acknowledgement. Providing these values in the upgrade request gives a client an opportunity to provide parameters prior to receiving any frames from the server.

就像对其他的 **SETTINGS** 帧那样，服务端对这些值进行解码和解释。没有必要对这些设置(6.5.3节)进行显示的确认，因为101响应就相当于隐式的确认。在收到服务端发送的帧之前，客户端有机会在升级请求的这些值里提供一些参数。

## Starting HTTP/2 for "https" URIs / 为"https" URIs启用HTTP/2协议

A client that makes a request to an "https" URI uses [TLS \[TLS12\]](#) with the [application-layer protocol negotiation \(ALPN\) extension \[TLS-ALPN\]](#).

客户端对"https" URI发起请求时使用带有 [应用层协议协商\(ALPN\)扩展](#) 的 [TLS \[TLS12\]](#)。

HTTP/2 over TLS uses the "h2" protocol identifier. The "h2c" protocol identifier MUST NOT be sent by a client or selected by a server; the "h2c" protocol identifier describes a protocol that does not use TLS.

运行在TLS之上的HTTP/2使用"h2"协议标识符。此时，客户端不能发送"h2c"协议标识符，服务端也不能选择"h2c"协议标识符；"h2c"协议标识符表示HTTP/2不使用TLS。

Once TLS negotiation is complete, both the client and the server MUST send a connection preface ([Section 3.5](#)).

一旦TLS协商完成，客户端和服务端都必须发送一个连接前奏([3.5节](#))。

## Starting HTTP/2 with Prior Knowledge / 先知情况下启用HTTP/2

A client can learn that a particular server supports HTTP/2 by other means. For example, [\[ALT-SVC\]](#) describes a mechanism for advertising this capability.

客户端可以通过其他方式了解服务端是否支持HTTP/2。例如，[\[ALT-SVC\]](#)描述了一种通知支持HTTP/2的机制。

A client **MUST** send the connection preface ([Section 3.5](#)) and then **MAY** immediately send HTTP/2 frames to such a server; servers can identify these connections by the presence of the connection preface. This only affects the establishment of HTTP/2 connections over cleartext TCP; implementations that support HTTP/2 over TLS **MUST** use protocol negotiation in TLS [\[TLS-ALPN\]](#).

客户端必须先向这种服务端发送连接前奏([3.5节](#))，然后可以立即发送HTTP/2帧。服务端能通过连接前奏识别出这种连接。这会影响基于明文TCP建立的HTTP/2连接。基于TLS的HTTP/2实现必须使用TLS中的协议协商[\[TLS-ALPN\]](#)。

Likewise, the server **MUST** send a connection preface ([Section 3.5](#)).

同样，服务端也必须发送一个连接前奏([3.5节](#))。

Without additional information, prior support for HTTP/2 is not a strong signal that a given server will support HTTP/2 for future connections. For example, it is possible for server configurations to change, for configurations to differ between instances in clustered servers, or for network conditions to change.

没有额外的参考信息，某个服务端先前支持HTTP/2并不能表明它在以后的连接中仍会支持HTTP/2。例如，可能服务端配置改变了，或者集群中不同服务器的配置有差异，或者网络状况改变了。

# HTTP/2 Connection Preface / HTTP/2连接前奏

In HTTP/2, each endpoint is required to send a connection preface as a final confirmation of the protocol in use and to establish the initial settings for the HTTP/2 connection. The client and server each send a different connection preface.

在HTTP/2中，要求两端都要发送一个连接前奏，作为对所使用协议的最终确认，并确定HTTP/2连接的初始设置。客户端和服务端各自发送不同的连接前奏。

The client connection preface starts with a sequence of 24 octets, which in hex notation is:

```
0x505249202a20485454502f322e300d0a0d0a534d0d0a0d0a
```

That is, the connection preface starts with the string `PRI * HTTP/2.0\r\n\r\nSM\r\n\r\n`. This sequence **MUST** be followed by a [SETTINGS](#) frame ([Section 6.5](#)), which **MAY** be empty. The client sends the client connection preface immediately upon receipt of a 101 (Switching Protocols) response (indicating a successful upgrade) or as the first application data octets of a TLS connection. If starting an HTTP/2 connection with prior knowledge of server support for the protocol, the client connection preface is sent upon connection establishment.

客户端连接前奏以一个24字节的序列开始，用十六进制表示为：

```
0x505249202a20485454502f322e300d0a0d0a534d0d0a0d0a
```

即，连接前奏以字符串"`PRI * HTTP/2.0\r\n\r\nSM\r\n\r\n`"开始。这个序列后面必须跟一个可以为空的 [SETTINGS](#) 帧( [6.5节](#) )。客户端一收到101(Switching Protocols)响应(表示成功升级)后，就立即发送客户端连接前奏，或者将其作为TLS连接的第一批应用程序数据字节。如果在预先知道服务端支持HTTP/2的情况下启用HTTP/2连接，客户端连接前奏在连接建立时发送。

Note: The client connection preface is selected so that a large proportion of HTTP/1.1 or HTTP/1.0 servers and intermediaries do not attempt to process further frames. Note that this does not address the concerns raised in [\[TALKING\]](#).

注意：客户端连接前奏是专门挑选的，目的是为了让大部分HTTP/1.1或HTTP/1.0服务器和中介不会试图处理后面的帧，但这并没有解决在 [\[TALKING\]](#) 中提出的问题。

The server connection preface consists of a potentially empty **SETTINGS** frame (Section 6.5) that MUST be the first frame the server sends in the HTTP/2 connection.

服务端连接前奏包含一个可能为空的 **SETTINGS** 帧( 6.5 节 )，它必须由服务端在HTTP/2连接中首先发送。

The **SETTINGS** frames received from a peer as part of the connection preface MUST be acknowledged (see Section 6.5.3) after sending the connection preface.

在发送完本端的连接前奏之后，必须对收到的作为对端连接前奏一部分的 **SETTINGS** 帧进行确认(参见 6.5.3 节)。

To avoid unnecessary latency, clients are permitted to send additional frames to the server immediately after sending the client connection preface, without waiting to receive the server connection preface. It is important to note, however, that the server connection preface **SETTINGS** frame might include parameters that necessarily alter how a client is expected to communicate with the server. Upon receiving the **SETTINGS** frame, the client is expected to honor any parameters established. In some configurations, it is possible for the server to transmit **SETTINGS** before the client sends additional frames, providing an opportunity to avoid this issue.

为了避免不必要的延迟，允许客户端发送完连接前奏后就立即向服务端发送其他的帧，而不必等待服务端的连接前奏。不过需要注意的是，服务端连接前奏的 **SETTINGS** 帧中可能包含一些期望客户端如何与服务端进行通信所必须修改的参数。在收到这些 **SETTINGS** 帧以后，客户端应当遵守所有设置的参数。在某些配置中，服务端是可以在客户端发送额外的帧之前传送 **SETTINGS** 帧的，这样就避免前边所说的问题。

Clients and servers MUST treat an invalid connection preface as a connection error (Section 5.4.1) of type **PROTOCOL\_ERROR**. A **GOAWAY** frame (Section 6.8) MAY be omitted in this case, since an invalid preface indicates that the peer is not using HTTP/2.

客户端和服务端都必须将无效的连接前奏处理为连接错误( 5.4.1 节 )，错误类型为 **PROTOCOL\_ERROR**。在这种情况下，可以忽略 **GOAWAY** 帧( 6.8 节 )，因为无效的连接前奏表示对端并没有使用HTTP/2。

# HTTP Frames / HTTP 帧

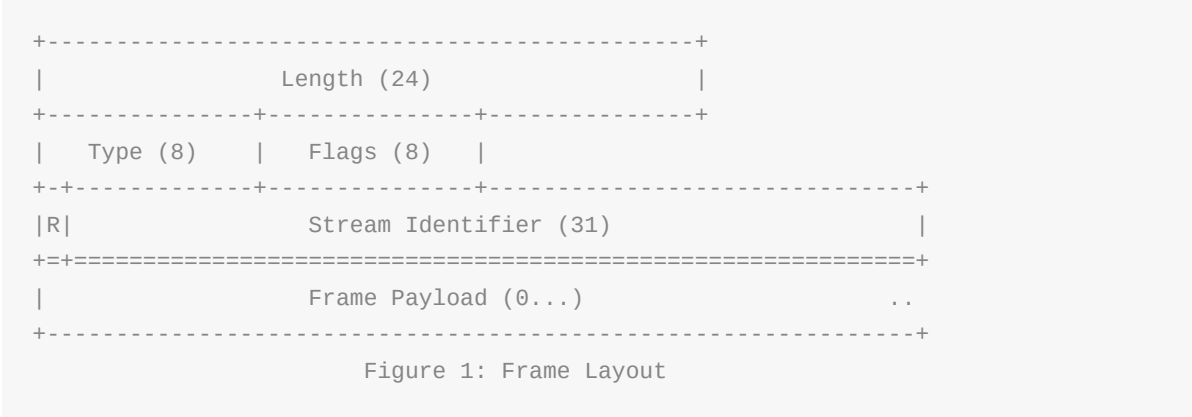
Once the HTTP/2 connection is established, endpoints can begin exchanging frames.

一旦建立了HTTP/2连接，端点之间就能开始交换帧了。

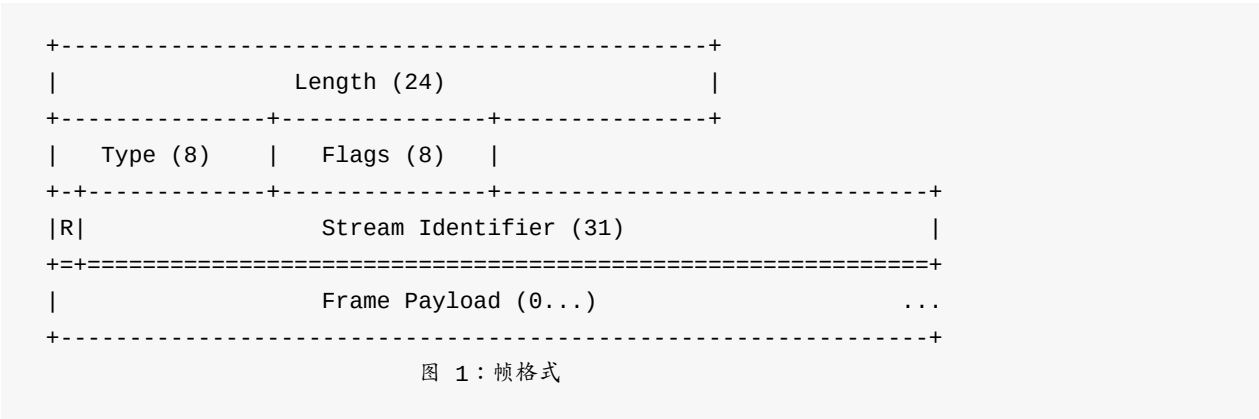


# Frame Format / 帧格式

All frames begin with a fixed 9-octet header followed by a variable-length payload.



所有的帧都以固定的9字节的首部开始，后面接可变长度的有效载荷。



The fields of the frame header are defined as:

- **Length:** The length of the frame payload expressed as an unsigned 24-bit integer. Values greater than  $2^{14}$  (16,384) MUST NOT be sent unless the receiver has set a larger value for SETTINGS\_MAX\_FRAME\_SIZE.

The 9 octets of the frame header are not included in this value.

- **Type:** The 8-bit type of the frame. The frame type determines the format and semantics of the frame. Implementations MUST ignore and discard any frame that has a type that is unknown.

- **Flags:** An 8-bit field reserved for boolean flags specific to the frame type.

Flags are assigned semantics specific to the indicated frame type. Flags that have no defined semantics for a particular frame type MUST be ignored and MUST be left unset (0x0) when sending.

- **R:** A reserved 1-bit field. The semantics of this bit are undefined, and the bit MUST remain unset (0x0) when sending and MUST be ignored when receiving.
- **Stream Identifier:** A stream identifier (see [Section 5.1.1](#)) expressed as an unsigned 31-bit integer. The value 0x0 is reserved for frames that are associated with the connection as a whole as opposed to an individual stream.

The structure and content of the frame payload is dependent entirely on the frame type.

帧首部字段定义如下：

- **Length:** 帧有效载荷的长度，以24位无符号整数表示。除非接收端通过 SETTINGS\_MAX\_FRAME\_SIZE 设置了更大的值，否则不能发送Length值大于 $2^{14}$  (16,384)的帧。

帧首部的9字节长度不计入该值。

- **Type:** 8bit的帧类型。帧类型决定了帧的格式和语义。必须忽略和丢弃任何未知的帧类型。

- **Flags:** 为帧类型保留的8bit布尔标识字段。

针对确定的帧类型赋予标识特定的语义。与确定的帧类型语义不相符的标识必须被忽略，并且在发送时必须未设置的(0x0)。

- **R:** 1bit的保留字段。未定义该bit的语义。当发送时，该bit必须是未设置的(0x0)；当接收时，必须忽略该bit。
- **Stream Identifier:** 流标识符(参见 [5.1.1节](#))是一个31bit的无符号整数。值0x0是保留的，表明帧是与整体的连接相关的，而不是和单独的流相关。

帧有效载荷结构和内容完全取决于帧类型。

## Frame Size / 帧大小

The size of a frame payload is limited by the maximum size that a receiver advertises in the [SETTINGS\\_MAX\\_FRAME\\_SIZE](#) setting. This setting can have any value between  $2^{14}$  (16,384) and  $2^{24}-1$  (16,777,215) octets, inclusive.

帧有效载荷的大小不能超过接收端通过 [SETTINGS\\_MAX\\_FRAME\\_SIZE](#) 所告知的值。该值可以取 $2^{14}$  (16,384)和 $2^{24}-1$  (16,777,215)之间的任何值，包括 $2^{24}-1$  (16,777,215)。

All implementations MUST be capable of receiving and minimally processing frames up to  $2^{14}$  octets in length, plus the 9-octet frame header ([Section 4.1](#)). The size of the frame header is not included when describing frame sizes.

所有的实现都必须能接收，并且处理最小长度为 $2^{14}$  字节、外加9字节帧首部([4.1节](#))的帧。当描述帧大小时，不包括帧首部的大小。

Note: Certain frame types, such as PING ([Section 6.7](#)), impose additional limits on the amount of payload data allowed.

注意：某些帧类型，如PING([6.7节](#))帧，对允许的有效载荷数据量强加了额外的限制。

An endpoint MUST send an error code of [FRAME\\_SIZE\\_ERROR](#) if a frame exceeds the size defined in [SETTINGS\\_MAX\\_FRAME\\_SIZE](#), exceeds any limit defined for the frame type, or is too small to contain mandatory frame data. A frame size error in a frame that could alter the state of the entire connection MUST be treated as a connection error ([Section 5.4.1](#)); this includes any frame carrying a header block ([Section 4.3](#)) (that is, [HEADERS](#), [PUSH\\_PROMISE](#), and [CONTINUATION](#)), [SETTINGS](#), and any frame with a stream identifier of 0.

如果一个帧超出了 [SETTINGS\\_MAX\\_FRAME\\_SIZE](#) 设置的大小，或者超出了任何为该帧类型设定的限制，或者帧太小而无法包含强制性的帧数据，端点必须发送一个 [FRAME\\_SIZE\\_ERROR](#) 错误码。必须将可以改变整个连接状态的帧大小错误处理为连接错误([5.4.1节](#))，这包括所有携带首部块([4.3节](#))的帧(即，[HEADERS](#)，[PUSH\\_PROMISE](#) 和 [CONTINUATION](#))，[SETTINGS](#) 帧，和所有流标识符为0的帧。

Endpoints are not obligated to use all available space in a frame. Responsiveness can be improved by using frames that are smaller than the permitted maximum size. Sending large frames can result in delays in sending time-sensitive frames (such as [RST\\_STREAM](#), [WINDOW\\_UPDATE](#), or [PRIORITY](#)), which, if blocked by the transmission of a large frame, could affect performance.

端点不必用掉帧的所有可用空间。如果帧大小小于允许的最大值，可以改善响应性能。发送大的帧会导致时间敏感的帧(如，[RST\\_STREAM](#)，[WINDOW\\_UPDATE](#)，或 [PRIORITY](#))发送延迟，这些帧如果被大帧阻塞住了，会影响性能。

# Header Compression and Decompression / 首部压缩与解压缩

Just as in HTTP/1, a header field in HTTP/2 is a name with one or more associated values. Header fields are used within HTTP request and response messages as well as in server push operations (see [Section 8.2](#)).

正如在HTTP/1里那样，HTTP/2里的首部字段也是一个键具有一个或多个值。这些首部字段用于HTTP请求和响应消息，也用于服务端推送操作([8.2节](#))。

Header lists are collections of zero or more header fields. When transmitted over a connection, a header list is serialized into a header block using [HTTP header compression](#) [COMPRESSION]. The serialized header block is then divided into one or more octet sequences, called header block fragments, and transmitted within the payload of HEADERS ([Section 6.2](#)), PUSH\_PROMISE ([Section 6.6](#)), or CONTINUATION ([Section 6.10](#)) frames.

首部列表是零个或多个首部字段的集合。当通过连接传送时，首部列表被 [HTTP首部压缩](#) [COMPRESSION]序列化成首部块。然后，序列化的首部块又被划分成一个或多个叫做首部块片段的字节序列，并通过HEADERS([6.2节](#))、PUSH\_PROMISE([6.6节](#))，或者CONTINUATION([6.10节](#))帧的有效负载传送。

The [Cookie header field](#) [COOKIE] is treated specially by the HTTP mapping (see [Section 8.1.2.5](#)).

[Cookie首部字段](#) [COOKIE] 需要HTTP映射特殊对待(参见 [8.1.2.5节](#))。

A receiving endpoint reassembles the header block by concatenating its fragments and then decompresses the block to reconstruct the header list.

接收端连接片段重组首部块，然后解压首部块重建首部列表。

A complete header block consists of either:

- a single [HEADERS](#) or [PUSH\\_PROMISE](#) frame, with the END\_HEADERS flag set, or
- a [HEADERS](#) or [PUSH\\_PROMISE](#) frame with the END\_HEADERS flag cleared and one or more [CONTINUATION](#) frames, where the last [CONTINUATION](#) frame has the END\_HEADERS flag set.

一个完整的首部块包括以下二者之一：

- 一个单独的、设置了END\_HEADERS标识的 [HEADERS](#) 或 [PUSH\\_PROMISE](#) 帧，或者

- 一个单独的、清除了END\_HEADERS标识的 HEADERS 或 PUSH\_PROMISE 帧，并有一个或多个 CONTINUATION 帧，且最后一个 CONTINUATION 帧设置了 END\_HEADERS标识。

Header compression is stateful. One compression context and one decompression context are used for the entire connection. A decoding error in a header block MUST be treated as a connection error (Section 5.4.1) of type COMPRESSION\_ERROR.

首部压缩是有状态的。一个连接具有一个压缩上下文和一个解压缩上下文。必须将首部块解码错误当做类型为 COMPRESSION\_ERROR 的连接错误( 5.4.1节 )处理。

Each header block is processed as a discrete unit. Header blocks MUST be transmitted as a contiguous sequence of frames, with no interleaved frames of any other type or from any other stream. The last frame in a sequence of HEADERS or CONTINUATION frames has the END\_HEADERS flag set. The last frame in a sequence of PUSH\_PROMISE or CONTINUATION frames has the END\_HEADERS flag set. This allows a header block to be logically equivalent to a single frame.

将每个首部块当做离散的单元处理。必须将首部块作为连续的帧序列传送，而没有交错任何其他类型或其他流的帧。HEADERS 或 CONTINUATION 帧序列的最后一个帧设置 END\_HEADERS标识。PUSH\_PROMISE 或 CONTINUATION 帧序列的最后一个帧设置 END\_HEADERS标识。这让首部块在逻辑上等价于一个单独的帧。

Header block fragments can only be sent as the payload of HEADERS, PUSH\_PROMISE, or CONTINUATION frames because these frames carry data that can modify the compression context maintained by a receiver. An endpoint receiving HEADERS, PUSH\_PROMISE, or CONTINUATION frames needs to reassemble header blocks and perform decompression even if the frames are to be discarded. A receiver MUST terminate the connection with a connection error (Section 5.4.1) of type COMPRESSION\_ERROR if it does not decompress a header block.

首部块片段只能作为 HEADERS 、PUSH\_PROMISE ，或 CONTINUATION 帧的有效载荷进行传送，因为这些帧携带的数据能修改接收端维护的压缩上下文。即使收到要被丢弃的 HEADERS 、PUSH\_PROMISE ，或 CONTINUATION 帧，接收端点也需要重组首部块并解压。如果接收端不解压首部块，它必须用类型为 COMPRESSION\_ERROR 的连接错误( 5.4.1节 )终止连接。











































## Frame / 帧定义

This specification defines a number of frame types, each identified by a unique 8-bit type code. Each frame type serves a distinct purpose in the establishment and management either of the connection as a whole or of individual streams.

该规范定义了若干帧类型，每种帧类型由唯一的8bit类型码标识。在建立和管理整个连接或者单独一个流时，每种帧类型都服务于特定的目的。

The transmission of specific frame types can alter the state of a connection. If endpoints fail to maintain a synchronized view of the connection state, successful communication within the connection will no longer be possible. Therefore, it is important that endpoints have a shared comprehension of how the state is affected by the use any given frame.

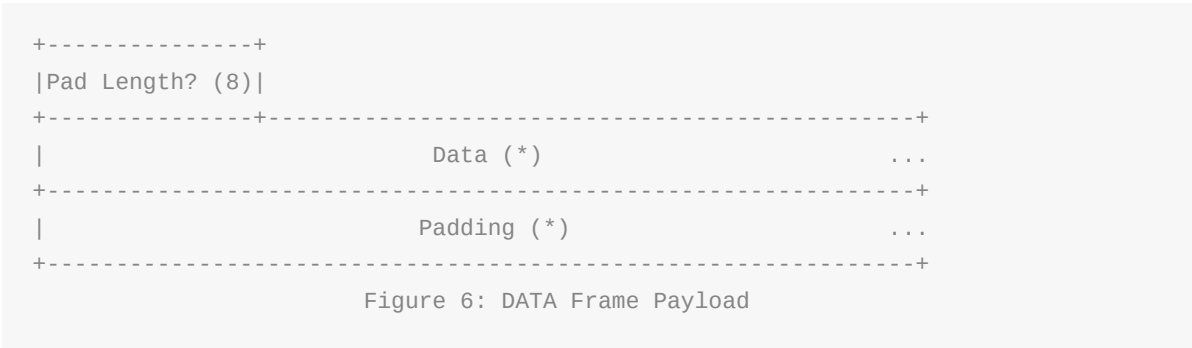
传送特定的帧类型可以改变连接的状态。如果两端不能保持同步的连接状态，就不可能再成功地进行通信。因此，对给定的帧怎样影响连接的状态，两端的理解应该保持一致。

# DATA / DATA帧

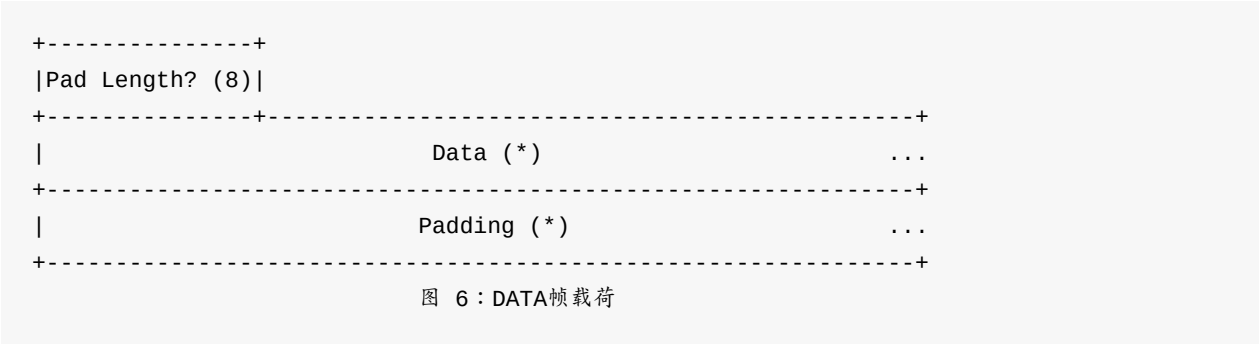
DATA frames (type=0x0) convey arbitrary, variable-length sequences of octets associated with a stream. One or more DATA frames are used, for instance, to carry HTTP request or response payloads.

DATA帧(type=0x0)用于传送某一个流的任意的、可变长度的字节序列。比如：用一个或多个DATA帧来携带HTTP请求或响应的载荷。

DATA frames MAY also contain padding. Padding can be added to DATA frames to obscure the size of messages. Padding is a security feature; see [Section 10.7](#).



DATA帧也可以包含填充数。为了模糊消息的大小，可以在DATA帧里加入填充数。填充数是一种安全特性，参见 [10.7节](#)。



The DATA frame contains the following fields:

- **Pad Length:** An 8-bit field containing the length of the frame padding in units of octets. This field is conditional (as signified by a "?" in the diagram) and is only present if the PADDED flag is set.
- **Data:** Application data. The amount of data is the remainder of the frame payload after subtracting the length of the other fields that are present.
- **Padding:** Padding octets that contain no application semantic value. Padding octets MUST be set to zero when sending. A receiver is not obligated to verify padding but MAY treat non-zero padding as a connection error ([Section 5.4.1](#)) of type [PROTOCOL\\_ERROR](#).

DATA 帧包含如下域：

- **填充长度(Pad Length)：**一个8bit的域，包含帧填充数据的字节长度。该域是可选的(正如框图中的"?"所示)，只有当设置了PADDED标识时，才会有该域。
- **数据(Data)：**应用数据。数据量等于帧载荷减去其它域的长度。
- **填充数据(Padding)：**不包含应用语义值的填充字节。当发送的时候，必须将填充数设置为0。接收方不必非得校验填充数据，但是可能会把非零的填充数当做 [PROTOCOL\\_ERROR](#) 类型的连接错误( [5.4.1 节](#) )。

The DATA frame defines the following flags:

- **END\_STREAM (0x1):** When set, bit 0 indicates that this frame is the last that the endpoint will send for the identified stream. Setting this flag causes the stream to enter one of the "half-closed" states or the "closed" state ([Section 5.1](#)).
- **PADDED (0x8):** When set, bit 3 indicates that the Pad Length field and any padding that it describes are present.

DATA 帧定义了如下标识：

- **END\_STREAM(0x1)：**当设置了该标识，第0位就指明了该帧是端点在指定流上发送的最后一帧。设置该标识会使流进入半关闭状态之一或者关闭状态( [5.1 节](#) )。
- **PADDED(0x8)：**当设置了该标识，第3位表示存在填充长度域和相应的填充数据。

DATA frames MUST be associated with a stream. If a DATA frame is received whose stream identifier field is 0x0, the recipient MUST respond with a connection error ([Section 5.4.1](#)) of type [PROTOCOL\\_ERROR](#).

DATA 帧必须和某一个流相关联。如果收到一个流标识符域为0x0的DATA帧，接收方必须响应一个 [PROTOCOL\\_ERROR](#) 类型的连接错误( [5.4.1 节](#) )。

DATA frames are subject to flow control and can only be sent when a stream is in the "open" or "half-closed (remote)" state. The entire DATA frame payload is included in flow control, including the Pad Length and Padding fields if present. If a DATA frame is received whose stream is not in "open" or "half-closed (local)" state, the recipient MUST respond with a stream error ([Section 5.4.2](#)) of type [STREAM\\_CLOSED](#).

DATA 帧受流量控制限制，并且只能在流处于 打开(open) 或者 半关闭(远端)(half-closed (remote)) 状态时发送。整个DATA帧载荷都受流量控制限制，如果有的话，也包括 填充长度 (Pad Length) 域和 填充数据(Padding) 域。如果流不是 打开(open) 或者 半关闭(本地)(half-closed (local)) 状态时收到了DATA帧，接收方必须响应一个 [STREAM\\_CLOSED](#) 类型的流错误( [5.4.2 节](#) )。

The total number of padding octets is determined by the value of the Pad Length field. If the length of the padding is the length of the frame payload or greater, the recipient MUST treat this as a connection error ([Section 5.4.1](#)) of type [PROTOCOL\\_ERROR](#).

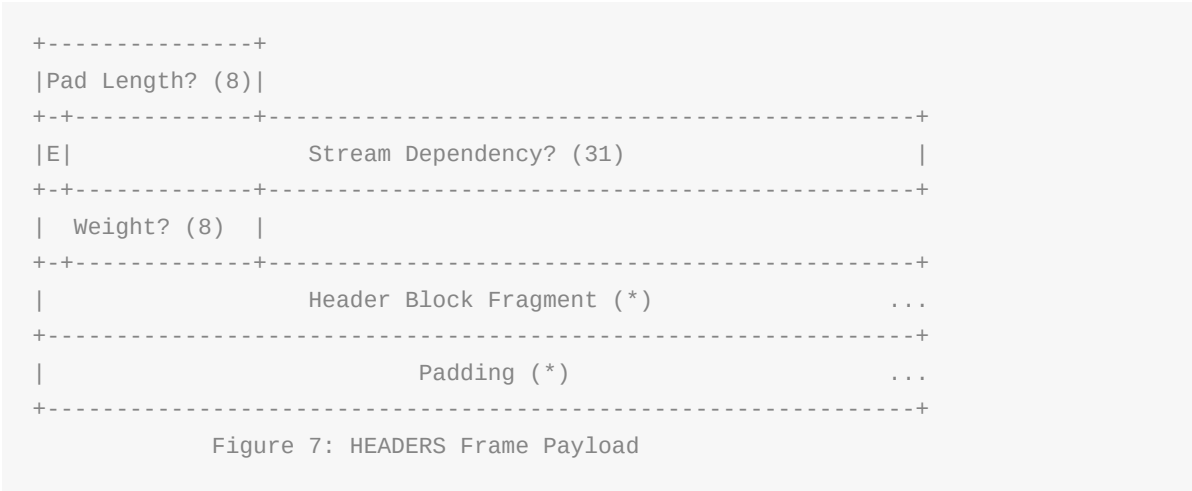
填充长度(Pad Length) 域的值决定了填充数据的字节总数。如果填充数据的长度大于等于帧载荷的长度，接收方必须把这种情况当做 [PROTOCOL\\_ERROR](#) 类型的连接错误( [5.4.1 节](#) )。

Note: A frame can be increased in size by one octet by including a Pad Length field with a value of zero.

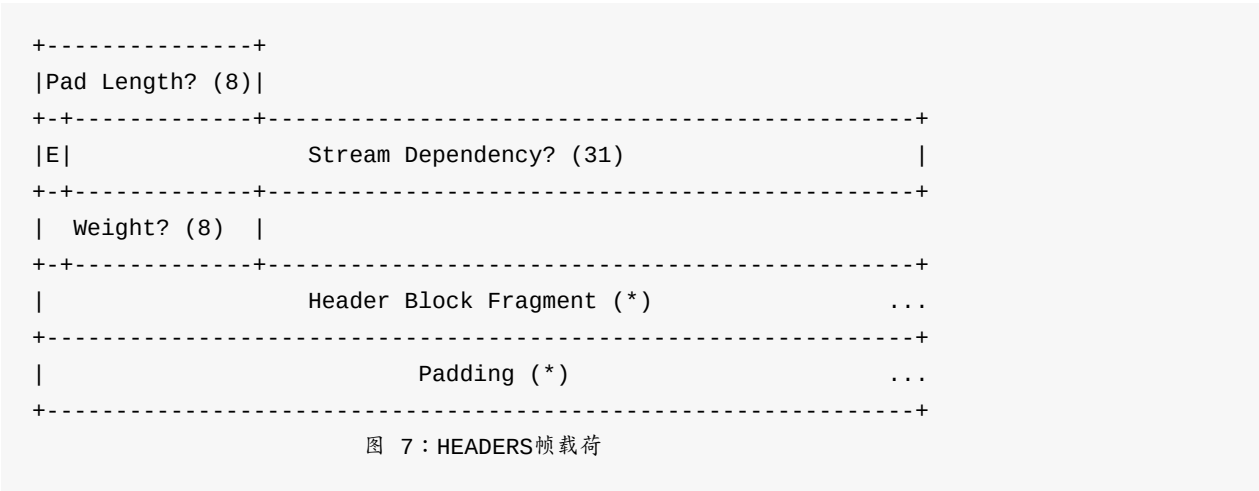
注意：通过包含一个值为零的 填充长度(Pad Length) 域，帧大小可以增加一个字节。

# HEADERS / HEADERS帧

The HEADERS frame (type=0x1) is used to open a stream ([Section 5.1](#)), and additionally carries a header block fragment. HEADERS frames can be sent on a stream in the "idle", "reserved (local)", "open", or "half-closed (remote)" state.



HEADERS帧(type=0x1)用来打开一个流([5.1节](#))，再额外地携带一个首部块片段(Header Block Fragment)。HEADERS帧可以在一个流处于 空闲(idle)、保留(本地)(reserved (local))、打开(open)、或者 半关闭(远端)(half-closed (remote)) 状态时被发送。



The HEADERS frame payload has the following fields:

- **Pad Length:** An 8-bit field containing the length of the frame padding in units of octets. This field is only present if the PADDED flag is set.
- **E:** A single-bit flag indicating that the stream dependency is exclusive (see [Section 5.3](#)). This field is only present if the PRIORITY flag is set.
- **Stream Dependency:** A 31-bit stream identifier for the stream that this stream depends on (see [Section 5.3](#)). This field is only present if the PRIORITY flag is set.
- **Weight:** An unsigned 8-bit integer representing a priority weight for the stream (see [Section 5.3](#)). Add one to the value to obtain a weight between 1 and 256. This field is only present if the PRIORITY flag is set.
- **Header Block Fragment:** A header block fragment ([Section 4.3](#)).
- **Padding:** Padding octets.

HEADERS 帧载荷包含如下域：

- **填充长度(Pad Length)：**一个8bit的域，包含帧填充数据的字节长度。只有当设置了PADDED标识时，才会有该域。
- **E标识：**1bit标识，表示流依赖是否是专用的(参见 [5.3节](#))。只有当设置了PRIORITY标识，才会有该域。
- **流依赖(Stream Dependency)：**该流所依赖的流(参见 [5.3节](#))的31bit标识符。只有当设置了PRIORITY标识，才会有该域。
- **权重(Weight)：**一个8bit的无符号整数，表示该流的优先级权重(参见 [5.3节](#))。范围是1到255。只有当设置了PRIORITY标识，才会有该域。
- **首部块片段(Header Block Fragment)：**一个首部块片段(参见 [4.3节](#))。
- **填充数据(Padding)：**填充字节。

The HEADERS frame defines the following flags:

- **END\_STREAM (0x1):** When set, bit 0 indicates that the header block ([Section 4.3](#)) is the last that the endpoint will send for the identified stream.

A HEADERS frame carries the END\_STREAM flag that signals the end of a stream. However, a HEADERS frame with the END\_STREAM flag set can be followed by [CONTINUATION](#) frames on the same stream. Logically, the [CONTINUATION](#) frames are part of the HEADERS frame.

- **END\_HEADERS (0x4):** When set, bit 2 indicates that this frame contains an entire header block ([Section 4.3](#)) and is not followed by any [CONTINUATION](#) frames.

A HEADERS frame without the END\_HEADERS flag set **MUST** be followed by a [CONTINUATION](#) frame for the same stream. A receiver **MUST** treat the receipt of any other type of frame or a frame on a different stream as a connection error ([Section 5.4.1](#)) of type [PROTOCOL\\_ERROR](#).

- **PADDED (0x8):** When set, bit 3 indicates that the Pad Length field and any padding that it describes are present.
- **PRIORITY (0x20):** When set, bit 5 indicates that the Exclusive Flag (E), Stream Dependency, and Weight fields are present; see [Section 5.3](#).

HEADERS 帧定义了如下标识：

- **END\_STREAM(0x1)：**当设置了该标识，第0 bit位就指明了该首部块是端点在指定流上发送的最后一帧( [4.3节](#) )。

HEADERS 帧携带的END\_STREAM标识指明流要结束了。但是，在同一个流上，设置了END\_STREAM标识的HEADERS帧后面还可以有 [CONTINUATION](#) 帧。从逻辑上来说，[CONTINUATION](#) 帧也是HEADERS帧的一部分。

- **END\_HEADERS(0x4)：**当设置了该标识，第2 bit位表示该帧包含整个首部块( [4.3节](#) )，后面不会有任何 [CONTINUATION](#) 帧。

对于同一个流，没有设置END\_HEADERS标识的HEADERS帧后面必须跟一个 [CONTINUATION](#) 帧。如果接收到任何其他类型的帧，或者其他流上的帧，接收方必须将其看做 [PROTOCOL\\_ERROR](#) 类型的连接错误( [5.4.1节](#) )。

- **PADDED(0x8)：**当设置了该标识，第3 bit位表示存在 填充长度(Pad Length) 域和 填充数据(padding)。
- **PRIORITY(0x20)：**当设置了该标识，第5 bit位表示存在 独占标识(E)(Exclusive Flag (E))、流依赖(Stream Dependency) 和 权重(Weight) 域。参加 [5.3节](#)



The payload of a HEADERS frame contains a header block fragment ([Section 4.3](#)). A header block that does not fit within a HEADERS frame is continued in a CONTINUATION frame ([Section 6.10](#)).

HEADERS 帧的载荷包含一个首部块片段([4.3 节](#))。同一个首部块在一个 HEADERS 帧里装不下就继续装入 CONTINUATION 帧([6.10 节](#))。

HEADERS frames MUST be associated with a stream. If a HEADERS frame is received whose stream identifier field is 0x0, the recipient MUST respond with a connection error ([Section 5.4.1](#)) of type `PROTOCOL_ERROR`.

HEADERS 帧必须与某一个流相关联。如果收到一个流标识符域为 0x0 的 HEADERS 帧，接收方必须响应一个 `PROTOCOL_ERROR` 类型的连接错误([5.4.1 节](#))。

The HEADERS frame changes the connection state as described in [Section 4.3](#).

HEADERS 帧改变连接状态在 [4.3 节](#) 中有表述。

The HEADERS frame can include padding. Padding fields and flags are identical to those defined for DATA frames ([Section 6.1](#)). Padding that exceeds the size remaining for the header block fragment MUST be treated as a `PROTOCOL_ERROR`.

HEADERS 帧可以包含填充数据。填充数据域和填充标识与 DATA 帧([6.1 节](#))中定义的一样。如果填充数据量超出了为首部块片段预留的大小，必须将其处理为 `PROTOCOL_ERROR`。

Prioritization information in a HEADERS frame is logically equivalent to a separate `PRIORITY` frame, but inclusion in HEADERS avoids the potential for churn in stream prioritization when new streams are created. Prioritization fields in HEADERS frames subsequent to the first on a stream reprioritize the stream ([Section 5.3.3](#)).

HEADERS 帧里的优先级信息逻辑上等价于一个单独的 `PRIORITY` 帧，但是包含在 HEADERS 帧里可以避免创建新流时对流优先级潜在的扰动。一个流上第一个 HEADERS 帧之后的 HEADERS 帧里的优先级域会变更该流的优先级顺序([5.3.3 节](#))。

# PRIORITY / PRIORITY帧

The PRIORITY frame (type=0x2) specifies the sender-advised priority of a stream (Section 5.3). It can be sent in any stream state, including idle or closed streams.

```
+-----+
|E|          Stream Dependency (31)          |
+-----+
|  Weight (8)  |
+-----+
```

Figure 8: PRIORITY Frame Payload

PRIORITY帧(type=0x2)指定了发送者建议的流优先级( 5.3节 )。可以在任何流状态下发送PRIORITY帧，包括空闲(idle)的和 关闭(closed) 的流。



- The payload of a PRIORITY frame contains the following fields:
- **E:** A single-bit flag indicating that the stream dependency is exclusive (see Section 5.3).
  - **Stream Dependency:** A 31-bit stream identifier for the stream that this stream depends on (see Section 5.3).
  - **Weight:** An unsigned 8-bit integer representing a priority weight for the stream (see Section 5.3). Add one to the value to obtain a weight between 1 and 256.

PRIORITY帧的载荷包含以下域：

- **E标识(E):** 1bit标识，指明流依赖(Stream Dependency)是否是专用的(参见 5.3节 )。
- **流依赖(Stream Dependency):** 该流所依赖的流(参见 5.3节 )的31bit标识符。
- **权重(Weight):** 一个8bit的无符号整数，表示该流的优先级权重(参见 5.3节 )。范围是1到255。

The PRIORITY frame does not define any flags.

PRIORITY帧没有定义任何标识。

The PRIORITY frame always identifies a stream. If a PRIORITY frame is received with a stream identifier of 0x0, the recipient MUST respond with a connection error (Section 5.4.1) of type `PROTOCOL_ERROR`.

PRIORITY 帧总是标识出了某一个流。如果收到一个流标识符为 0x0 的 PRIORITY 帧，接收方必须响应一个 `PROTOCOL_ERROR` 类型的连接错误( 5.4.1 节 )。

The PRIORITY frame can be sent on a stream in any state, though it cannot be sent between consecutive frames that comprise a single header block (Section 4.3). Note that this frame could arrive after processing or frame sending has completed, which would cause it to have no effect on the identified stream. For a stream that is in the "half-closed (remote)" or "closed" state, this frame can only affect processing of the identified stream and its dependent streams; it does not affect frame transmission on that stream.

可以在处于任意状态的流上发送 PRIORITY 帧，但却不能在包含同一个首部块 4.3 节的连续帧之间发送。需要注意的是，PRIORITY 帧可能会在处理或发送帧完成后到达，这会导致 PRIORITY 帧在特定的流上不起作用。对处于 半关闭(远端)(half-closed (remote)) 或者 关闭(closed) 状态的流来说，PRIORITY 帧只能影响对该特定流及其从属流的处理，而不会影响在该流上的帧传送。

The PRIORITY frame can be sent for a stream in the "idle" or "closed" state. This allows for the reprioritization of a group of dependent streams by altering the priority of an unused or closed parent stream.

可以在处于 空闲(idle) 或者 关闭(closed) 状态的流上发送 PRIORITY 帧，从而可以通过改变一个未使用的或者关闭的母流的优先级，来改变其从属流的优先级顺序。

A PRIORITY frame with a length other than 5 octets MUST be treated as a stream error (Section 5.4.2) of type `FRAME_SIZE_ERROR`.

必须将不是 5 字节长度的 PRIORITY 帧当做一个类型为 `FRAME_SIZE_ERROR` 的流错误 5.4.2 节。

## RST\_STREAM / RST\_STREAM帧

The RST\_STREAM frame (type=0x3) allows for immediate termination of a stream. RST\_STREAM is sent to request cancellation of a stream or to indicate that an error condition has occurred.

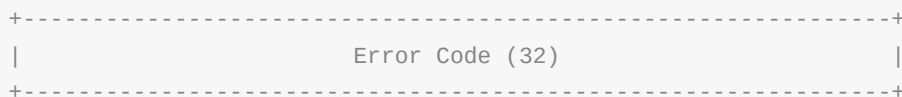


Figure 9: RST\_STREAM Frame Payload

RST\_STREAM帧(type=0x3)可以立即终结一个流。RST\_STREAM用来请求取消一个流，或者表示发生了一个错误。

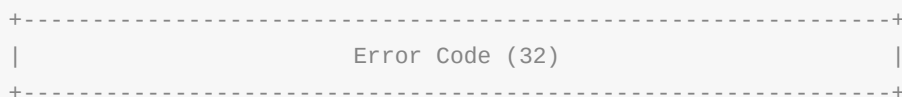


图 9：RST\_STREAM帧负载

The RST\_STREAM frame contains a single unsigned, 32-bit integer identifying the error code ( [Section 7](#) ). The error code indicates why the stream is being terminated.

RST\_STREAM帧包含一个32bit的无符号整数表示的错误码( [第7章](#) )。错误码指明了流为什么被终结。

The RST\_STREAM frame does not define any flags.

RST\_STREAM帧没有定义任何特征位标记。

The RST\_STREAM frame fully terminates the referenced stream and causes it to enter the "closed" state. After receiving a RST\_STREAM on a stream, the receiver MUST NOT send additional frames for that stream, with the exception of [PRIORITY](#). However, after sending the RST\_STREAM, the sending endpoint MUST be prepared to receive and process additional frames sent on the stream that might have been sent by the peer prior to the arrival of the RST\_STREAM.

RST\_STREAM帧完全终结了其所在的流，并且促使该流进入 关闭(closed) 状态。当在某一个流上收到一个RST\_STREAM帧以后，除了 [PRIORITY](#) 帧，接收端不能在该流上发送任何其他帧。但是，在发送完RST\_STREAM帧以后，发送端必须准备好接收并处理该流上的其他帧，这些帧可能是在RST\_STREAM帧到达前由对端所发送。

RST\_STREAM frames MUST be associated with a stream. If a RST\_STREAM frame is received with a stream identifier of 0x0, the recipient MUST treat this as a connection error ([Section 5.4.1](#) of type [PROTOCOL\\_ERROR](#)).

RST\_STREAM帧必须和一个流相关联。如果收到一个流标识符为0x0的RST\_STREAM帧，接收端必须将其处理为类型为 [PROTOCOL\\_ERROR](#) 的连接错误( [5.4.1节](#) )。

RST\_STREAM frames MUST NOT be sent for a stream in the "idle" state. If a RST\_STREAM frame identifying an idle stream is received, the recipient MUST treat this as a connection error ([Section 5.4.1](#)) of type [PROTOCOL\\_ERROR](#).

不能在处于 空闲(idle) 状态的流上发送RST\_STREAM帧。如果收到一个空闲流上的RST\_STREAM帧，接收方必须将其处理为类型为 [PROTOCOL\\_ERROR](#) 的连接错误( [5.4.1节](#) )。

A RST\_STREAM frame with a length other than 4 octets MUST be treated as a connection error ([Section 5.4.1](#)) of type [FRAME\\_SIZE\\_ERROR](#).

必须将不是4字节长度的RST\_STREAM帧当做一个类型为 [FRAME\\_SIZE\\_ERROR](#) 的连接错误 [5.4.1节](#) 。

## SETTINGS / SETTINGS 帧

The SETTINGS frame (type=0x4) conveys configuration parameters that affect how endpoints communicate, such as preferences and constraints on peer behavior. The SETTINGS frame is also used to acknowledge the receipt of those parameters. Individually, a SETTINGS parameter can also be referred to as a "setting".

SETTINGS 帧(type=0x4)用来传送影响两端通信的配置参数，比如：对对端行为的偏好与约束等。SETTINGS 帧也用于通知对端自己收到了这些参数。特别地，SETTIGNS 参数也可以被称做 设置参数(setting)。

SETTINGS parameters are not negotiated; they describe characteristics of the sending peer, which are used by the receiving peer. Different values for the same parameter can be advertised by each peer. For example, a client might set a high initial flow-control window, whereas a server might set a lower value to conserve resources.

SETTINGS 参数不是靠协商得来的。这些参数描述了发送端的特性，并被接收端所使用。对于相同的参数，两端可以使用不同的值。例如，客户端可以设置一个较大的流量控制窗口(flow-control window) 值，而服务端为了保存资源，可以设置一个较小的值。

A SETTINGS frame MUST be sent by both endpoints at the start of a connection and MAY be sent at any other time by either endpoint over the lifetime of the connection. Implementations MUST support all of the parameters defined by this specification.

在连接建立的时候两端必须发送SETTIGNS 帧，也可以在连接的整个生命周期内由任何一端在任意时间发送SETTIGNS 帧。实现必须支持本规范文档定义的所有参数。

Each parameter in a SETTINGS frame replaces any existing value for that parameter. Parameters are processed in the order in which they appear, and a receiver of a SETTINGS frame does not need to maintain any state other than the current value of its parameters. Therefore, the value of a SETTINGS parameter is the last value that is seen by a receiver.

SETTIGNS 帧里的参数会各自替换该参数已存在的值。以参数出现的顺序来处理参数，除了参数的当前值，SETTIGNS 帧的接收方不需要维护任何状态。因此，接收方看到的最后的值就是SETTIGNS 帧的参数值。

SETTINGS parameters are acknowledged by the receiving peer. To enable this, the SETTINGS frame defines the following flag:

- **ACK (0x1):** When set, bit 0 indicates that this frame acknowledges receipt and application of the peer's SETTINGS frame. When this bit is set, the payload of the SETTINGS frame MUST be empty. Receipt of a SETTINGS frame with the ACK flag set and a length field value other than 0 MUST be treated as a connection error (Section 5.4.1) of type `FRAME_SIZE_ERROR`. For more information, see Section 6.5.3 ("Settings Synchronization").

SETTINGS 帧携带的参数会被接收方确认。为了做到这个，SETTINGS 帧定义了如下标识：

- **ACK (0x1):** 当设置了该标识，bit 0 表示该帧确认收到并应用了对端的 SETTINGS 帧。当设置了该 bit，SETTINGS 帧的负载必须为空。如果收到了设置了 ACK 标识，并且长度域的值不为 0 的 SETTINGS 帧，必须将其当做类型为 `FRAME_SIZE_ERROR` 的连接错误(5.4.1 节)。更多信息，参见 6.5.3 节 ("Settings Synchronization")。

SETTINGS frames always apply to a connection, never a single stream. The stream identifier for a SETTINGS frame MUST be zero (0x0). If an endpoint receives a SETTINGS frame whose stream identifier field is anything other than 0x0, the endpoint MUST respond with a connection error (Section 5.4.1) of type `PROTOCOL_ERROR`.

SETTINGS 帧总是作用于连接，而不是一个流。SETTINGS 帧的流标识符必须为 0(0x0)。如果一端收到了流标识符不是 0x0 的 SETTINGS 帧，该端点必须响应一个类型为 `PROTOCOL_ERROR` 的连接错误(Section 5.4.1)。

The SETTINGS frame affects connection state. A badly formed or incomplete SETTINGS frame MUST be treated as a connection error (Section 5.4.1) of type `PROTOCOL_ERROR`.

SETTINGS 帧影响连接状态。必须将损坏的或者不完整的 SETTINGS 帧当做类型为 `PROTOCOL_ERROR` 的连接错误(5.4.1 节)。

A SETTINGS frame with a length other than a multiple of 6 octets MUST be treated as a connection error (Section 5.4.1) of type `FRAME_SIZE_ERROR`.

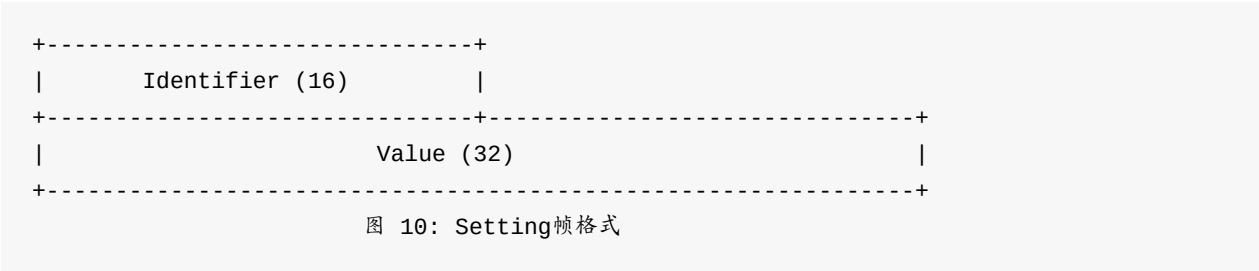
必须将长度不是 6 的倍数的 SETTINGS 帧当做类型为 `FRAME_SIZE_ERROR` 的连接错误(5.4.1 节)。

# SETTINGS Format / SETTINGS帧格式

The payload of a SETTINGS frame consists of zero or more parameters, each consisting of an unsigned 16-bit setting identifier and an unsigned 32-bit value.



SETTIGNS帧的负载包含零个或者多个参数，每个参数包含一个16bit的无符号设置标识符(setting identifier) 和一个32bit的无符号值。





## Defined SETTINGS Parameters / 已定义的 SETTINGS 帧参数

The following parameters are defined:

- **SETTINGS\_HEADER\_TABLE\_SIZE (0x1):** Allows the sender to inform the remote endpoint of the maximum size of the header compression table used to decode header blocks, in octets. The encoder can select any size equal to or less than this value by using signaling specific to the header compression format inside a header block (see [\[COMPRESSION\]](#)). The initial value is 4,096 octets.
- **SETTINGS\_ENABLE\_PUSH (0x2):** This setting can be used to disable server push ([Section 8.2](#)). An endpoint MUST NOT send a [PUSH\\_PROMISE](#) frame if it receives this parameter set to a value of 0. An endpoint that has both set this parameter to 0 and had it acknowledged MUST treat the receipt of a [PUSH\\_PROMISE](#) frame as a connection error ([Section 5.4.1](#)) of type [PROTOCOL\\_ERROR](#).

The initial value is 1, which indicates that server push is permitted. Any value other than 0 or 1 MUST be treated as a connection error ([Section 5.4.1](#)) of type [PROTOCOL\\_ERROR](#).

- **SETTINGS\_MAX\_CONCURRENT\_STREAMS (0x3):** Indicates the maximum number of concurrent streams that the sender will allow. This limit is directional: it applies to the number of streams that the sender permits the receiver to create. Initially, there is no limit to this value. It is recommended that this value be no smaller than 100, so as to not unnecessarily limit parallelism.

A value of 0 for `SETTINGS_MAX_CONCURRENT_STREAMS` SHOULD NOT be treated as special by endpoints. A zero value does prevent the creation of new streams; however, this can also happen for any limit that is exhausted with active streams. Servers SHOULD only set a zero value for short durations; if a server does not wish to accept requests, closing the connection is more appropriate.

- **SETTINGS\_INITIAL\_WINDOW\_SIZE (0x4):** Indicates the sender's initial window size (in octets) for stream-level flow control. The initial value is 2<sup>16</sup>-1 (65,535) octets.

This setting affects the window size of all streams (see [Section 6.9.2](#)).

Values above the maximum flow-control window size of  $2^{31}-1$  MUST be treated as a connection error (Section 5.4.1) of type `FLOW_CONTROL_ERROR`.

- **SETTINGS\_MAX\_FRAME\_SIZE (0x5):** Indicates the size of the largest frame payload that the sender is willing to receive, in octets.

The initial value is  $2^{14}$  (16,384) octets. The value advertised by an endpoint MUST be between this initial value and the maximum allowed frame size ( $2^{24} - 1$  or 16,777,215 octets), inclusive. Values outside this range MUST be treated as a connection error (Section 5.4.1) of type `PROTOCOL_ERROR`.

- **SETTINGS\_MAX\_HEADER\_LIST\_SIZE (0x6):** This advisory setting informs a peer of the maximum size of header list that the sender is prepared to accept, in octets. The value is based on the uncompressed size of header fields, including the length of the name and value in octets plus an overhead of 32 octets for each header field.

For any given request, a lower limit than what is advertised MAY be enforced. The initial value of this setting is unlimited.

定义了如下参数：

- **SETTINGS\_HEADER\_TABLE\_SIZE (0x1):** 允许发送方通知远端用于解码首部块的首部压缩表的最大字节值。通过使用特定于首部块(参见 [\[COMPRESSION\]](#))内部的首部压缩格式的信令，编码器可以选择任何小于等于该值的大小。其初始值是4096字节。
- **SETTINGS\_ENABLE\_PUSH (0x2):** 该设置用于关闭服务端推送(8.2节)。如果一端收到了该参数值为0，该端点不能发送 `PUSH_PROMISE` 帧。如果一端把该参数设置为0，并且收到了确认，当它收到了 `PUSH_PROMISE` 帧时，它必须将其当做类型 `PROTOCOL_ERROR` 的连接错误(5.4.1节)。

该值初始为1，表示允许服务端推送功能。如果该值不是0或者1，必须将其当做类型为 `PROTOCOL_ERROR` 的连接错误(5.4.1节)。

- **SETTINGS\_MAX\_CONCURRENT\_STREAMS (0x3):** 指明发送端允许的最大并发流数。该值是有方向性的：它适用于发送端允许接收端创建的流数目。最初，对该值是没有限制的。为了不非必要地限制并发，推荐该值不小于100。

端点不应该特殊对待 `SETTINGS_MAX_CONCURRENT_STREAMS` 为0的值。该值为0的确会阻止创建新流，但是，对任何耗尽限制数内活跃流的情况，也会发生阻止新流的创建。服务端应该只能在短期内设置该值为0。如果服务端不想接收请求，关闭连接更合适。

- **SETTINGS\_INITIAL\_WINDOW\_SIZE (0x4):** 指明发送端流级别的流量控制窗口的初始字节大小。该初始值是  $2^{16} - 1$  (65,535)字节。

该设置会影响所有流的窗口大小(参见 6.9.2 节)。

如果该值大于流量控制窗口的最大值 $2^{31} - 1$ ，必须将其当做类型为 `FLOW_CONTROL_ERROR` 的连接错误( 5.4.1 节)。

- **SETTINGS\_MAX\_FRAME\_SIZE (0x5):** 指明发送端希望接收的最大帧负载的字节值。

初始值是 $2^{14}$  (16,384)字节。一端告知的该值必须介于初始值和允许的最大帧大小( $2^{24} - 1$  或者 16,777,215字节)之间，包括该最大值。必须将超出该范围的值当做类型为 `PROTOCOL_ERROR` 的连接错误( 5.4.1 节)。

- **SETTINGS\_MAX\_HEADER\_LIST\_SIZE (0x6):** 该建议设置通知对端发送端准备接收的首部列表大小的最大字节值。该值是基于未压缩的首部域大小，包括名称和值的字节长度，外加每个首部域的32字节的开销。

对于任何给定的请求，其大小必须低于告知的值。该设置的初始值大小是没有限制的。

An endpoint that receives a SETTINGS frame with any unknown or unsupported identifier MUST ignore that setting.

如果一端收到了带有任何未知的或不支持的标识符的SETTINGS帧，必须忽略该设置参数。

# Settings Synchronization / 设置同步

Most values in SETTINGS benefit from or require an understanding of when the peer has received and applied the changed parameter values. In order to provide such synchronization timepoints, the recipient of a SETTINGS frame in which the ACK flag is not set **MUST** apply the updated parameters as soon as possible upon receipt.

SETTINGS帧里的大部分值都受益于或者要求理解对端什么时候收到并且应用了已改变的参数值。为了提供这样的同步时间点，接收方必须一收到没有设置ACK标识的SETTINGS帧就尽快应用更新后的参数值。

The values in the SETTINGS frame **MUST** be processed in the order they appear, with no other frame processing between values. Unsupported parameters **MUST** be ignored. Once all values have been processed, the recipient **MUST** immediately emit a SETTINGS frame with the ACK flag set. Upon receiving a SETTINGS frame with the ACK flag set, the sender of the altered parameters can rely on the setting having been applied.

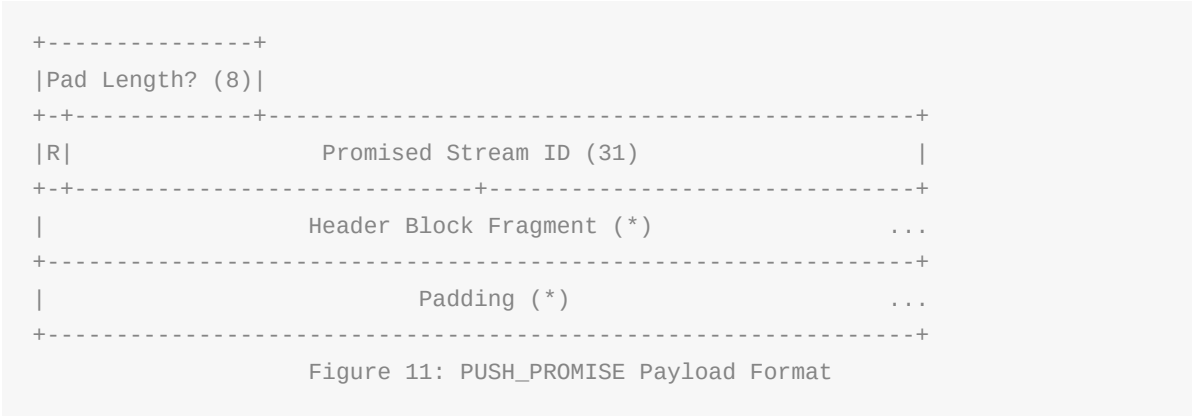
SETTINGS帧里的值必须以其出现的顺序进行处理，而不能在值之间加入对其它帧的处理。不支持的参数必须被忽略。一旦所有的值都被处理完毕，接收方必须立即发送一个设置了ACK标识的SETTIGNS帧。一收到设置了ACK标识的SETTIGNS帧，已改变的参数的发送方就可以信赖已被应用的这些设置。

If the sender of a SETTINGS frame does not receive an acknowledgement within a reasonable amount of time, it **MAY** issue a connection error ([Section 5.4.1](#)) of type **SETTINGS\_TIMEOUT**.

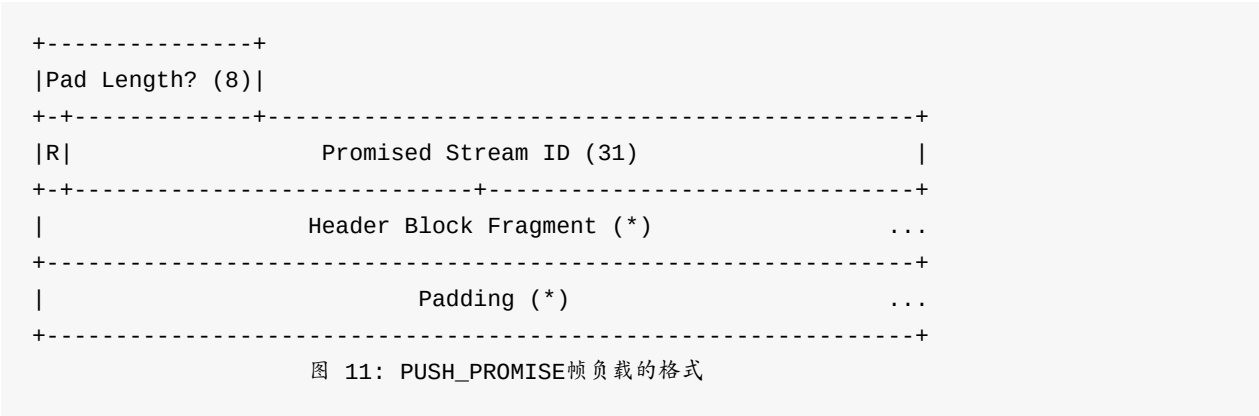
如果SETTINGS帧的发送方在一个合理的时间没有收到确认，它可以发送一个类型为**SETTINGS\_TIMEOUT**的连接错误([5.4.1节](#))。

# PUSH\_PROMISE / PUSH\_PROMISE 帧

The PUSH\_PROMISE frame (type=0x5) is used to notify the peer endpoint in advance of streams the sender intends to initiate. The PUSH\_PROMISE frame includes the unsigned 31-bit identifier of the stream the endpoint plans to create along with a set of headers that provide additional context for the stream. [Section 8.2](#) contains a thorough description of the use of PUSH\_PROMISE frames.



在发送端准备初始化流之前，要发送PUSH\_PROMISE(type=0x5)帧来通知对端。  
PUSH\_PROMISE帧包含31位的无符号流标识符，该流标识符和为流提供额外的上下文的首部集一起由端点创建的。[8.2节](#) 详细描述了PUSH\_PROMISE帧的使用。



The PUSH\_PROMISE frame payload has the following fields:

- **Pad Length:** An 8-bit field containing the length of the frame padding in units of octets. This field is only present if the PADDED flag is set.
- **R:** A single reserved bit.
- **Promised Stream ID:** An unsigned 31-bit integer that identifies the stream that is reserved by the PUSH\_PROMISE. The promised stream identifier MUST be a valid choice for the next stream sent by the sender (see "new stream identifier" in Section 5.1.1).
- **Header Block Fragment:** A header block fragment (Section 4.3) containing request header fields.
- **Padding:** Padding octets.

PUSH\_PROMISE 帧负载具有以下域：

- **Pad Length:** 一个8bit的域，包含帧填充数据的字节长度。只有当设置了PADDED标识时，该域才会出现。
- **R:** 保留的1bit位。
- **Promised Stream ID:** 31bit的无符号整数，标记PUSH\_PROMISE帧保留的流。对于发送端来说，该标识符必须是可用于下一个流的有效值(参见 5.1.1 节的创建流标识符)。
- **Header Block Fragment:** 包含请求首部域的头部块片段( 4.3 节 )。
- **Padding:** 填充字节。

The PUSH\_PROMISE frame defines the following flags:

- **END\_HEADERS (0x4):** When set, bit 2 indicates that this frame contains an entire header block (Section 4.3) and is not followed by any CONTINUATION frames.

A PUSH\_PROMISE frame without the END\_HEADERS flag set MUST be followed by a CONTINUATION frame for the same stream. A receiver MUST treat the receipt of any other type of frame or a frame on a different stream as a connection error (Section 5.4.1) of type `PROTOCOL_ERROR`.

- **PADDED (0x8):** When set, bit 3 indicates that the Pad Length field and any padding that it describes are present.

PUSH\_PROMISE 帧定义了如下标识符：

- **END\_HEADERS (0x4):** 当设置了该标识符，第二个bit位表示这个帧包含整个首部块( 4.3 节 )，并且后面没有跟随任何 CONTINUATION 帧。

对于同一个流，没有设置END\_HEADERS标识的PUSH\_PROMISE帧后面必须跟随有CONTINUATION帧。如果接收端收到了其他类型的帧，或者是其他流上的帧，接收端必须将其当做类型为 `PROTOCOL_ERROR` 的流错误( 5.4.1 节 )。

- **PADDED (0x8):** 当设置了该标识符，第3个bit位表示出现了填充长度(Pad Length)域和它所描述的填充字节。

PUSH\_PROMISE frames MUST only be sent on a peer-initiated stream that is in either the "open" or "half-closed (remote)" state. The stream identifier of a PUSH\_PROMISE frame indicates the stream it is associated with. If the stream identifier field specifies the value 0x0, a recipient MUST respond with a connection error (Section 5.4.1) of type **PROTOCOL\_ERROR**.

只能在处于 打开(open) 或者 半关闭(远端)(half-closed (remote)) 状态的对等初始化的流上发送PUSH\_PROMISE帧。PUSH\_PROMISE帧的流标识符指明其所关联的流。如果流标识符域的值是0x0，接收方必须响应一个类型为 **PROTOCOL\_ERROR** 的连接错误(5.4.1节)。

Promised streams are not required to be used in the order they are promised. The PUSH\_PROMISE only reserves stream identifiers for later use.

不要求被允诺的流以他们被允诺的顺序来被使用。PUSH\_PROMISE帧只保留用于以后的流标识符。

PUSH\_PROMISE MUST NOT be sent if the SETTINGS\_ENABLE\_PUSH setting of the peer endpoint is set to 0. An endpoint that has set this setting and has received acknowledgement MUST treat the receipt of a PUSH\_PROMISE frame as a connection error (Section 5.4.1) of type **PROTOCOL\_ERROR**.

如果对端的SETTINGS\_ENABLE\_PUSH被设置为0，不能发送PUSH\_PROMISE帧。如果一端已经设置了该设置项，并且收到了确认，当它收到了PUSH\_PROMISE帧时，必须将其当做类型为 **PROTOCOL\_ERROR** 的连接错误(5.4.1节)。

Recipients of PUSH\_PROMISE frames can choose to reject promised streams by returning a **RST\_STREAM** referencing the promised stream identifier back to the sender of the PUSH\_PROMISE.

PUSH\_PROMISE帧的接收方可以通过给PUSH\_PROMISE帧的发送方返回一个 **RST\_STREAM** 帧来选择拒绝被允诺的流，该RST\_STREAM帧包含被允诺的流的标识符。

A PUSH\_PROMISE frame modifies the connection state in two ways. First, the inclusion of a header block (Section 4.3) potentially modifies the state maintained for header compression. Second, PUSH\_PROMISE also reserves a stream for later use, causing the promised stream to enter the "reserved" state. A sender MUST NOT send a PUSH\_PROMISE on a stream unless that stream is either "open" or "half-closed (remote)"; the sender MUST ensure that the promised stream is a valid choice for a new stream identifier (Section 5.1.1) (that is, the promised stream MUST be in the "idle" state).



PUSH\_PROMISE 帧有两种方式来修改连接状态。首先，包含一个首部块(4.3节)会潜在地修改为首部压缩维护的状态。其次，PUSH\_PROMISE 帧也会为以后的使用保留一个流，从而使被允诺的流进入保留(reserved)状态。只有当流处于打开(open)或者半关闭(远端)(half-closed (remote))状态时，发送方才能在该流上发送PUSH\_PROMISE 帧；发送方必须确保被允诺的流对一个新的流标识符(5.1.1节)是有效的(即，被允诺的流必须处于空闲(idle)状态)。

Since PUSH\_PROMISE reserves a stream, ignoring a PUSH\_PROMISE frame causes the stream state to become indeterminate. A receiver MUST treat the receipt of a PUSH\_PROMISE on a stream that is neither "open" nor "half-closed (local)" as a connection error (Section 5.4.1) of type **PROTOCOL\_ERROR**. However, an endpoint that has sent **RST\_STREAM** on the associated stream MUST handle PUSH\_PROMISE frames that might have been created before the **RST\_STREAM** frame is received and processed.

因为PUSH\_PROMISE 帧会保留一个流，所以忽略PUSH\_PROMISE 帧会导致该流的状态变成不确定的。如果在既不是打开(open)也不是半关闭(half-closed(local))状态的流上收到了PUSH\_PROMISE 帧，接收方必须将其当做类型为 **PROTOCOL\_ERROR** 的连接错误(5.4.1节)。但是，在关联流上发送了 **RST\_STREAM** 帧的端点必须能处理PUSH\_PROMISE 帧，该PUSH\_PROMISE 帧可能在收到并处理 **RST\_STREAM** 帧之前就已经被创建了。

A receiver MUST treat the receipt of a PUSH\_PROMISE that promises an illegal stream identifier (Section 5.1.1) as a connection error (Section 5.4.1) of type **PROTOCOL\_ERROR**. Note that an illegal stream identifier is an identifier for a stream that is not currently in the "idle" state.

如果接收端收到允诺了一个非法的流标识符(5.1.1节)的PUSH\_PROMISE 帧，必须将其当做类型为 **PROTOCOL\_ERROR** 的连接错误(5.4.1节)。注意，非法的流标识符是当前状态不是空闲(idle)状态的流的标识符。

The PUSH\_PROMISE frame can include padding. Padding fields and flags are identical to those defined for DATA frames (Section 6.1).

PUSH\_PROMISE 帧可以包含填充数据。填充数据域和标识符跟为DATA帧(6.1节)定义的相同。



# PING / PING 帧

The PING frame (type=0x6) is a mechanism for measuring a minimal round-trip time from the sender, as well as determining whether an idle connection is still functional. PING frames can be sent from any endpoint.




Figure 12: PING Payload Format

除了判断一个空闲的连接是否仍然可用之外，PING帧(type=0x6)还是发送端测量最小往返时间的一种机制。任何端点都可以发送PING帧。



图 12: PING帧的负载格式

In addition to the frame header, PING frames MUST contain 8 octets of opaque data in the payload. A sender can include any value it chooses and use those octets in any fashion.

除了帧首部，PING帧还必须在负载中包含8字节的不透明数据。发送端可以选择任意值，而且可以以任意形式使用这些值。

Receivers of a PING frame that does not include an ACK flag MUST send a PING frame with the ACK flag set in response, with an identical payload. PING responses SHOULD be given higher priority than any other frame.

如果接收端收到了不包含ACK标识的PING帧，必须响应一个设置了ACK标识的PING帧，其负载与收到的PING帧的负载相同。PING帧的响应应该被赋予最高优先级。

The PING frame defines the following flags:

- **ACK (0x1):** When set, bit 0 indicates that this PING frame is a PING response. An endpoint MUST set this flag in PING responses. An endpoint MUST NOT respond to PING frames containing this flag.

PING帧定义了如下标识符：

- **ACK (0x1):** 当设置了该标识符，第0位表示该PING帧是一个PING帧的响应。端点必须在PING帧的响应里设置该标识符。端点不能响应包含该标识符的PING帧。

PING frames are not associated with any individual stream. If a PING frame is received with a stream identifier field value other than 0x0, the recipient MUST respond with a connection error (Section 5.4.1) of type **PROTOCOL\_ERROR**.

PING帧不能与任何流相关联。如果收到了流标识符域的值不是0x0的PING帧，接收端必须响应一个类型为 **PROTOCOL\_ERROR** 的连接错误( 5.4.1节 )。

Receipt of a PING frame with a length field value other than 8 MUST be treated as a connection error (Section 5.4.1) of type **FRAME\_SIZE\_ERROR**.

如果收到了长度域的值不是8的PING帧，必须将其当做类型为 **FRAME\_SIZE\_ERROR** 的连接错误( 5.4.1节 )。

## GOAWAY / GOAWAY 帧

The GOAWAY frame (type=0x7) is used to initiate shutdown of a connection or to signal serious error conditions. GOAWAY allows an endpoint to gracefully stop accepting new streams while still finishing processing of previously established streams. This enables administrative actions, like server maintenance.

GOAWAY 帧(type=0x7)用于发起关闭连接，或者警示严重错误。GOAWAY 帧能让端点优雅地停止接受新流，同时仍然能处理完先前建立的流。这使类似于服务端维护的管理行为成为可能。

There is an inherent race condition between an endpoint starting new streams and the remote sending a GOAWAY frame. To deal with this case, the GOAWAY contains the stream identifier of the last peer-initiated stream that was or might be processed on the sending endpoint in this connection. For instance, if the server sends a GOAWAY frame, the identified stream is the highest-numbered stream initiated by the client.

在发起新流的端点和发送GOAWAY帧的远端之间，内在地存在着竞争条件。为了解决这个问题，GOAWAY帧包含对端最后初始创建的流的标识符，该流被或者可能被连接的发送端处理。例如，如果服务端发送一个GOAWAY帧，被标识的流就是由客户端初始创建的最大标号的流。

Once sent, the sender will ignore frames sent on streams initiated by the receiver if the stream has an identifier higher than the included last stream identifier. Receivers of a GOAWAY frame MUST NOT open additional streams on the connection, although a new connection can be established for new streams.

一旦GOAWAY帧被发送，如果接收端初始创建的流的标识符大于GOAWAY帧携带的最后的流标识符，发送端将会忽略在这些流上发送的帧。尽管可以在新的连接上创建新的流，但是GOAWAY帧的接收端不能在原有的连接上再打开其他的流。

If the receiver of the GOAWAY has sent data on streams with a higher stream identifier than what is indicated in the GOAWAY frame, those streams are not or will not be processed. The receiver of the GOAWAY frame can treat the streams as though they had never been created at all, thereby allowing those streams to be retried later on a new connection.

如果GOAWAY帧的接收端在流上发送数据，这些流的流标识符比GOAWAY帧里携带的流标识符大，那么这些流将不会被处理。GOAWAY帧的接收端可以将这些流看做从来没有创建过一样，因此，这些流随后可以在一个新的连接上重新被创建。

Endpoints SHOULD always send a GOAWAY frame before closing a connection so that the remote peer can know whether a stream has been partially processed or not. For example, if an HTTP client sends a POST at the same time that a server closes a connection, the client cannot know if the server started to process that POST request if the server does not send a GOAWAY frame to indicate what streams it might have acted on.

在关闭连接之前，端点应该总是发送一个GOAWAY帧，从而可以让远端知道流是否被部分地处理了。例如，假设在服务端关闭连接的同时，HTTP客户端发送了一个POST请求，如果服务端没有发送GOAWAY帧来指明关闭连接影响到了哪些流，客户端不可能知道服务端是否开始处理该POST请求。

An endpoint might choose to close a connection without sending a GOAWAY for misbehaving peers.

如果对端出错了，一端可以不用发送一个GOAWAY帧而关闭连接。

A GOAWAY frame might not immediately precede closing of the connection; a receiver of a GOAWAY that has no more use for the connection SHOULD still send a GOAWAY frame before terminating the connection.

```

+---+-----+
|R|                Last-Stream-ID (31)                |
+---+-----+
|                Error Code (32)                        |
+---+-----+
|                Additional Debug Data (*)              |
+---+-----+

```

Figure 13: GOAWAY Payload Format

GOAWAY帧之后可能不会立即关闭连接。GOAWAY帧的接收端应该在关闭连接之前仍然发送一个GOAWAY帧。

```

+---+-----+
|R|                Last-Stream-ID (31)                |
+---+-----+
|                Error Code (32)                        |
+---+-----+
|                Additional Debug Data (*)              |
+---+-----+

```

图 13: GOAWAY帧负载格式

The GOAWAY frame does not define any flags.

GOAWAY帧没有定义任何标志。

The GOAWAY frame applies to the connection, not a specific stream. An endpoint MUST treat a GOAWAY frame with a stream identifier other than 0x0 as a connection error (Section 5.4.1) of type `PROTOCOL_ERROR`.

GOAWAY帧作用于连接，而不是流。如果GOAWAY帧的流标识符不是0x0，一端必须将其当做类型为 `PROTOCOL_ERROR` 的连接错误( 5.4.1 节 )。

The last stream identifier in the GOAWAY frame contains the highest-numbered stream identifier for which the sender of the GOAWAY frame might have taken some action on or might yet take action on. All streams up to and including the identified stream might have been processed in some way. The last stream identifier can be set to 0 if no streams were processed.

GOAWAY帧里最后的流标识符包含最大数字的流标识符，GOAWAY帧的发送者可能已经、或者仍然在作用于该流。小于等于该标识符的流以某种方式被处理。如果没有流被处理，最后的流标识符可以被设置为0。

Note: In this context, "processed" means that some data from the stream was passed to some higher layer of software that might have taken some action as a result.

注意：在这个上下文里，『被处理』表示该流传送的一些数据被传递给软件的更高层，结果导致了采取了一些行动。

If a connection terminates without a GOAWAY frame, the last stream identifier is effectively the highest possible stream identifier.

如果没有GOAWAY帧就关闭连接，最后的流标识符实际上就是最大可能的流标识符。

On streams with lower- or equal-numbered identifiers that were not closed completely prior to the connection being closed, reattempting requests, transactions, or any protocol activity is not possible, with the exception of idempotent actions like HTTP GET, PUT, or DELETE. Any protocol activity that uses higher-numbered streams can be safely retried using a new connection.

连接关闭之前，在没有完全关闭的、流标识符小于等于最大值的流上，重新尝试请求、事务，或者任何协议行为是不可能的，但诸如HTTP GET、PUT、或者DELETE等幂等动作除外。在流标识符大于最大值的流上的任何协议动作都可以在一个新连接上安全地重试。

Activity on streams numbered lower or equal to the last stream identifier might still complete successfully. The sender of a GOAWAY frame might gracefully shut down a connection by sending a GOAWAY frame, maintaining the connection in an "open" state until all in-progress streams complete.

在小于等于最后的流标识符的流上的行为可能仍然能成功地完成。GOAWAY 帧的发送端可能会通过发送一个GOAWAY帧来优雅地关闭连接，同时保持连接在 打开(open) 状态，直到所有的流都完成。

An endpoint MAY send multiple GOAWAY frames if circumstances change. For instance, an endpoint that sends GOAWAY with **NO\_ERROR** during graceful shutdown could subsequently encounter a condition that requires immediate termination of the connection. The last stream identifier from the last GOAWAY frame received indicates which streams could have been acted upon. Endpoints MUST NOT increase the value they send in the last stream identifier, since the peers might already have retried unprocessed requests on another connection.

如果环境变化了，一端可能会发送多个GOAWAY帧。例如，在优雅地关闭期间，发送带有 **NO\_ERROR** 的GOAWAY帧的端点可能随后就会遭遇需要立即关闭连接的情况。收到的最后的GOAWAY帧上最后的流标识符指明哪些流受影响了。端点不能增加它们发送的最后的流标识符的值，因为对端可能已经在另外一个连接上重试未处理的请求了。

A client that is unable to retry requests loses all requests that are in flight when the server closes the connection. This is especially true for intermediaries that might not be serving clients using HTTP/2. A server that is attempting to gracefully shut down a connection SHOULD send an initial GOAWAY frame with the last stream identifier set to  $2^{31} - 1$  and a **NO\_ERROR** code. This signals to the client that a shutdown is imminent and that initiating further requests is prohibited. After allowing time for any in-flight stream creation (at least one round-trip time), the server can send another GOAWAY frame with an updated last stream identifier. This ensures that a connection can be cleanly shut down without losing requests.

当服务端关闭连接时，不能重试请求的客户端会丢失所有传输途中的请求。这对不向客户端提供HTTP/2服务的中介来说尤其正确。试图优雅地关闭连接的服务端应该发送一个初始GOAWAY帧，其 最后的流标识符(last stream identifier) 设置为 $2^{31} - 1$ ，并且带有一个 **NO\_ERROR** 码。这会通知客户端，连接即将关闭，禁止再发起新的请求。一段可以让传输途中的流创建完成的时间(至少一个往返时间)过后，服务端可以发送其它的GOAWAY帧，携带更新的 最后流标识符(last time identifier)。这样可以确保干净地关闭连接，不会丢失请求。

After sending a GOAWAY frame, the sender can discard frames for streams initiated by the receiver with identifiers higher than the identified last stream. However, any frames that alter connection state cannot be completely ignored. For instance, **HEADERS**, **PUSH\_PROMISE**, and **CONTINUATION** frames MUST be minimally processed to ensure the state maintained for header compression is consistent (see [Section 4.3](#)); similarly, DATA frames MUST be counted toward the connection flow-control window. Failure to process these frames can cause flow control or header compression state to become unsynchronized.

发送了GOAWAY帧以后，发送端可以丢弃接收端初始创建的、其标识符大于最后的流标识符的流上的帧。但是，改变连接状态的帧不能被完全忽略。例如，HEADERS帧，PUSH\_PROMISE帧，和CONTINUATION帧，必须至少被处理，以确保为首部压缩维护的状态是一致(参见4.3节)。类似地，必须将DATA帧计入连接的流量控制窗口。对这些帧的处理失败会导致流量控制或者首部压缩状态变得不同步。

The GOAWAY frame also contains a 32-bit error code (Section 7) that contains the reason for closing the connection.

GOAWAY帧还包含一个32bit的错误码(第7章)，该错误码包含了关闭连接的原因。

Endpoints MAY append opaque data to the payload of any GOAWAY frame. Additional debug data is intended for diagnostic purposes only and carries no semantic value. Debug information could contain security- or privacy-sensitive data. Logged or otherwise persistently stored debug data MUST have adequate safeguards to prevent unauthorized access.

端点可以在任何GOAWAY帧的负载上附加不透明的数据。额外的调试数据只用于诊断目的，并且不携带语义值。调试信息可以包含安全相关的、或者隐私敏感的数据。登陆相关的、或者其他持久存储的调试数据必须具备足够的安全措施，以阻止未授权访问。

# WINDOW\_UPDATE / WINDOW\_UPDATE 帧

The WINDOW\_UPDATE frame (type=0x8) is used to implement flow control; see [Section 5.2](#) for an overview.

WINDOW\_UPDATE 帧(type=0x8)用于执行流量控制功能；参见 [5.2节](#) 的概述。

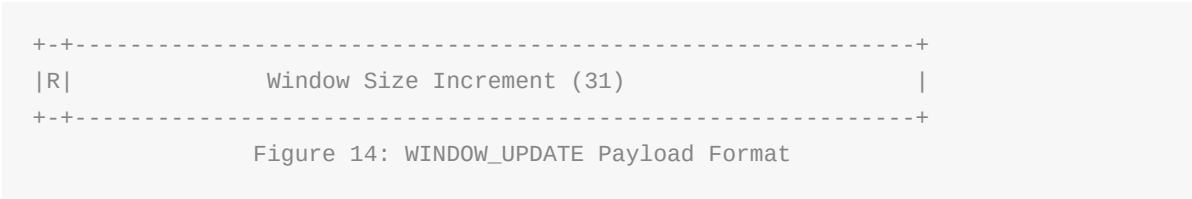
Flow control operates at two levels: on each individual stream and on the entire connection.

流量控制操作有两个层次：在每一个单独的流上和在整个连接上。

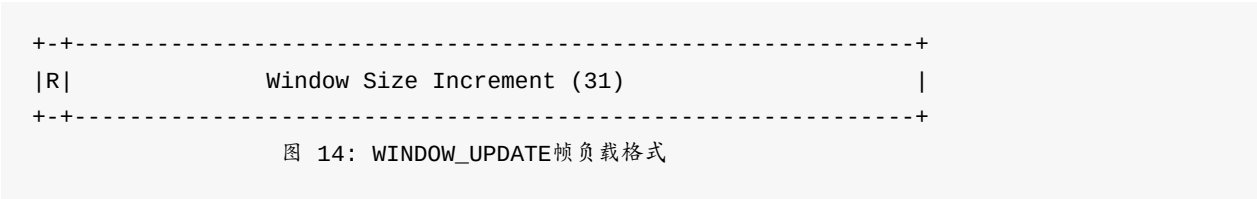
Both types of flow control are hop by hop, that is, only between the two endpoints. Intermediaries do not forward WINDOW\_UPDATE frames between dependent connections. However, throttling of data transfer by any receiver can indirectly cause the propagation of flow-control information toward the original sender.

这两种流量控制都是逐跳的，即，只在两个端点之间。中介不会两个独立的连接之间转发 WINDOW\_UPDATE 帧。但是，任何接收端对数据传输的遏制都会间接地导致流量控制信息向源发送端传播。

Flow control only applies to frames that are identified as being subject to flow control. Of the frame types defined in this document, this includes only [DATA](#) frames. Frames that are exempt from flow control MUST be accepted and processed, unless the receiver is unable to assign resources to handling the frame. A receiver MAY respond with a stream error ([Section 5.4.2](#)) or connection error ([Section 5.4.1](#)) of type [FLOW\\_CONTROL\\_ERROR](#) if it is unable to accept a frame.



流量控制功能只适用于被标识的、受流量控制影响的帧。本文档定义的帧类型里，只有 [DATA](#) 帧 受流量控制影响。除非接收端不能再分配资源去处理这些帧，否则不受流量控制影响的帧 必须被接收并处理。如果接收端不能再接收帧了，它可以响应一个 [FLOW\\_CONTROL\\_ERROR](#) 类型的 流错误( [5.4.2节](#) ) 或者 连接错误( [5.4.1节](#) )。





The payload of a WINDOW\_UPDATE frame is one reserved bit plus an unsigned 31-bit integer indicating the number of octets that the sender can transmit in addition to the existing flow-control window. The legal range for the increment to the flow-control window is 1 to  $2^{31} - 1$  (2,147,483,647) octets.

WINDOW\_UPDATE 帧的负载由保留的1bit位，加上一个31bit的无符号整数组成，该整数表示除了现有的流量控制窗口之外，发送端还可以传送的字节数。流量控制窗口增长的合法范围是1到 $2^{31} - 1$ (2,147,483,647)字节。

The WINDOW\_UPDATE frame does not define any flags.

WINDOW\_UPDATE 帧没有定义任何标志。

The WINDOW\_UPDATE frame can be specific to a stream or to the entire connection. In the former case, the frame's stream identifier indicates the affected stream; in the latter, the value "0" indicates that the entire connection is the subject of the frame.

WINDOW\_UPDATE 帧可以是针对一个流的，或者是针对整个连接的。如果是前者，WINDOW\_UPDATE 帧的流标识符指明了受影响的流；如果是后者，值为0表示整个连接都受 WINDOW\_UPDATE 帧的影响。

A receiver MUST treat the receipt of a WINDOW\_UPDATE frame with an flow-control window increment of 0 as a stream error (Section 5.4.2) of type **PROTOCOL\_ERROR**; errors on the connection flow-control window MUST be treated as a connection error (Section 5.4.1).

如果收到了 流量控制窗口增量(flow-control window increment) 为0的WINDOW\_UPDATE 帧，接收端必须将其当做类型为 **PROTOCOL\_ERROR** 的流错误( 5.4.2 节 )；发生在连接上的流量控制窗口错误必须被当做连接错误( 5.4.1 节 )。

WINDOW\_UPDATE can be sent by a peer that has sent a frame bearing the END\_STREAM flag. This means that a receiver could receive a WINDOW\_UPDATE frame on a "half-closed (remote)" or "closed" stream. A receiver MUST NOT treat this as an error (see Section 5.1).

WINDOW\_UPDATE 可以由发送过携带有END\_STREAM标志的帧的对端发送。这意味着接收端可能会在 半关闭(远端)(half-closed (remote)) 或者 关闭(closed) 状态的流上收到 WINDOW\_UPDATE 帧。接收端不能将其当做错误(参加 5.1 节 )。

A receiver that receives a flow-controlled frame MUST always account for its contribution against the connection flow-control window, unless the receiver treats this as a connection error (Section 5.4.1). This is necessary even if the frame is in error. The sender counts the frame toward the flow-control window, but if the receiver does not, the flow-control window at the sender and receiver can become different.

除非将其当做连接错误( [5.4.1节](#) ), 否则当接收端收到被流量控制影响的帧时, 必须总是将其从流量控制窗口中减掉。即使帧有错误, 这也是有必要的。发送端将该帧计入流量控制窗口, 但是如果接收端没有这样做, 发送端和接收端的流量控制窗口就会变得不同。

A WINDOW\_UPDATE frame with a length other than 4 octets MUST be treated as a connection error ([Section 5.4.1](#)) of type [FRAME\\_SIZE\\_ERROR](#).

如果WINDOW\_UPDATE帧的长度不是4字节, 必须将其当做类型为 [FRAME\\_SIZE\\_ERROR](#) 的连接错误( [5.4.1节](#) )。

## The Flow-Control Window / 流量控制窗口

Flow control in HTTP/2 is implemented using a window kept by each sender on every stream. The flow-control window is a simple integer value that indicates how many octets of data the sender is permitted to transmit; as such, its size is a measure of the buffering capacity of the receiver.

HTTP/2中的流量控制功能通过每个流上的发送端各自维持的窗口来实现。流量控制窗口是一个简单的整数值，指出了准许发送端传送的数据的字节数。这样，窗口值衡量了接收端的缓存能力。

Two flow-control windows are applicable: the stream flow-control window and the connection flow-control window. The sender MUST NOT send a flow-controlled frame with a length that exceeds the space available in either of the flow-control windows advertised by the receiver. Frames with zero length with the END\_STREAM flag set (that is, an empty DATA frame) MAY be sent if there is no available space in either flow-control window.

有两种流量控制窗口：流的流量控制窗口和连接的流量控制窗口。发送端不能发送受流量控制影响的、其长度超出接收端告知的这两种流量控制窗口可用空间的帧。即使这两种流量控制窗口都没有可用空间了，也可以发送长度为0、设置了END\_STREAM标志的帧(即，空的DATA帧)。

For flow-control calculations, the 9-octet frame header is not counted.

9字节的帧首部不计入流量控制的计算。

After sending a flow-controlled frame, the sender reduces the space available in both windows by the length of the transmitted frame.

发送了一个受流量控制影响的帧以后，发送端减少两个窗口的可用空间，减少量为已经发送的帧的长度大小。

The receiver of a frame sends a WINDOW\_UPDATE frame as it consumes data and frees up space in flow-control windows. Separate WINDOW\_UPDATE frames are sent for the stream- and connection-level flow-control windows.

当帧的接收端消耗了数据并释放了流量控制窗口的空间时，它发送一个WINDOW\_UPDATE帧。对于流级别和连接级别的流量控制窗口，需要分别发送WINDOW\_UPDATE帧。

A sender that receives a WINDOW\_UPDATE frame updates the corresponding window by the amount specified in the frame.

收到WINDOW\_UPDATE帧的发送端要更新相应的窗口，更新量已在WINDOW\_UPDATE帧里指定。

A sender MUST NOT allow a flow-control window to exceed  $2^{31} - 1$  octets. If a sender receives a WINDOW\_UPDATE that causes a flow-control window to exceed this maximum, it MUST terminate either the stream or the connection, as appropriate. For streams, the sender sends a RST\_STREAM with an error code of FLOW\_CONTROL\_ERROR; for the connection, a GOAWAY frame with an error code of FLOW\_CONTROL\_ERROR is sent.

发送端不能让流量控制窗口超出  $2^{31} - 1$  字节。如果发送端收到一个WINDOW\_UPDATE帧使流量控制窗口超出该最大值，它必须终止相应的流或连接。对于流，发送端发送一个RST\_STREAM帧，错误码为 FLOW\_CONTROL\_ERROR；对于连接，发送一个GOAWAY帧，错误码为 FLOW\_CONTROL\_ERROR。

Flow-controlled frames from the sender and WINDOW\_UPDATE frames from the receiver are completely asynchronous with respect to each other. This property allows a receiver to aggressively update the window size kept by the sender to prevent streams from stalling.

发送端发送的受流量控制影响的帧和来自于接收端的WINDOW\_UPDATE帧之间彼此完全异步。这个属性允许接收端剧烈地更新发送端维持的窗口大小，以防止流中止。

# Initial Flow-Control Window Size / 初始化流量控制窗口大小

When an HTTP/2 connection is first established, new streams are created with an initial flow-control window size of 65,535 octets. The connection flow-control window is also 65,535 octets. Both endpoints can adjust the initial window size for new streams by including a value for `SETTINGS_INITIAL_WINDOW_SIZE` in the `SETTINGS` frame that forms part of the connection preface. The connection flow-control window can only be changed using `WINDOW_UPDATE` frames.

当HTTP/2连接首次被建立时，新创建流的初始流量控制窗口大小为65,535字节。连接的流量控制窗口也是65,535字节。两端都可以通过组成连接序幕的 `SETTING` 帧里的 `SETTINGS_INITIAL_WINDOW_SIZE` 来调整新流的初始流量控制窗口的大小。连接的流量控制窗口只能通过`WINDOW_UPDATE`帧来改变。

Prior to receiving a `SETTINGS` frame that sets a value for `SETTINGS_INITIAL_WINDOW_SIZE`, an endpoint can only use the default initial window size when sending flow-controlled frames. Similarly, the connection flow-control window is set to the default initial window size until a `WINDOW_UPDATE` frame is received.

在收到设置了 `SETTINGS_INITIAL_WINDOW_SIZE` 的 `SETTIGNS` 帧之前，当一端发送受流量控制影响的帧时，只能使用默认的初始窗口大小。相似地，直到收到了 `WINDOW_UPDATE` 帧之前，连接的流量控制窗口都设置为默认的初始窗口大小。

In addition to changing the flow-control window for streams that are not yet active, a `SETTINGS` frame can alter the initial flow-control window size for streams with active flow-control windows (that is, streams in the "open" or "half-closed (remote)" state). When the value of `SETTINGS_INITIAL_WINDOW_SIZE` changes, a receiver MUST adjust the size of all stream flow-control windows that it maintains by the difference between the new value and the old value.

除了改变还未激活的流的流量控制窗口，`SETTIGNS` 帧还可以用激活的流量控制窗口改变流（即，处于 打开(open) 或者 半关闭(远端)(half-closed (remote)) 状态的流)的初始流量控制窗口的大小。当 `SETTINGS_INITIAL_WINDOW_SIZE` 的值变化了，接收端必须调整它所维护的所有流的流量控制窗口的值。

A change to `SETTINGS_INITIAL_WINDOW_SIZE` can cause the available space in a flow-control window to become negative. A sender MUST track the negative flow-control window and MUST NOT send new flow-controlled frames until it receives `WINDOW_UPDATE` frames that cause the flow-control window to become positive.

改变 `SETTINGS_INITIAL_WINDOW_SIZE` 可以引发流量控制窗口的可用空间变成负值。发送端必须追踪负的流量控制窗口，并且直到它收到了使流量控制窗口变成正值的 `WINDOW_UPDATE` 帧，才能发送新的受流量控制影响的帧。

For example, if the client sends 60 KB immediately on connection establishment and the server sets the initial window size to be 16 KB, the client will recalculate the available flow-control window to be -44 KB on receipt of the `SETTINGS` frame. The client retains a negative flow-control window until `WINDOW_UPDATE` frames restore the window to being positive, after which the client can resume sending.

例如，如果连接一建立客户端就立即发送60KB的数据，而服务端却将初始窗口大小设置为16KB，那么客户端一收到 `SETTINGS` 帧，就会将可用的流量控制窗口重新计算为-44KB。客户端保持负的流量控制窗口，直到`WINDOW_UPDATE`帧将窗口值恢复为正值，这之后，客户端才可以继续发送数据。

A `SETTINGS` frame cannot alter the connection flow-control window.

`SETTINGS` 帧不能改变连接的流量控制窗口值。

An endpoint MUST treat a change to `SETTINGS_INITIAL_WINDOW_SIZE` that causes any flow-control window to exceed the maximum size as a connection error (Section 5.4.1) of type `FLOW_CONTROL_ERROR`.

如果改变 `SETTINGS_INITIAL_WINDOW_SIZE` 导致流量控制窗口超出了最大值，一端必须将其当做类型为 `FLOW_CONTROL_ERROR` 的连接错误( 5.4.1 节 )。

## Reducing the Stream Window Size / 减小流的窗口大小

A receiver that wishes to use a smaller flow-control window than the current size can send a new **SETTINGS** frame. However, the receiver **MUST** be prepared to receive data that exceeds this window size, since the sender might send data that exceeds the lower limit prior to processing the **SETTINGS** frame.

如果接收端希望使用比当前值小的流量控制窗口，可以发送一个新的 **SETTINGS** 帧。但是，接收端必须准备好接收超出该窗口值的数据，因为可能在处理 **SETTINGS** 帧之前，发送端已经发送了超出该较小窗口值的数据。

After sending a **SETTINGS** frame that reduces the initial flow-control window size, a receiver **MAY** continue to process streams that exceed flow-control limits. Allowing streams to continue does not allow the receiver to immediately reduce the space it reserves for flow-control windows. Progress on these streams can also stall, since **WINDOW\_UPDATE** frames are needed to allow the sender to resume sending. The receiver **MAY** instead send a **RST\_STREAM** with an error code of **FLOW\_CONTROL\_ERROR** for the affected streams.

发送了可以减小初始流量控制窗口大小的 **SETTINGS** 帧以后，接收端可以继续处理超出流量控制限制的流。允许流继续意味着不允许接收端立即减小它为流量控制窗口保留的空间。因为需要 **WINDOW\_UPDATE** 帧允许发送端重新开始发送数据，对这些流的处理也可以中止。对于受影响的流，接收端可以代替发送一个 **RST\_STREAM** 帧，携带错误码 **FLOW\_CONTROL\_ERROR**。

# CONTINUATION / CONTINUATION 帧

The CONTINUATION frame (type=0x9) is used to continue a sequence of header block fragments (Section 4.3). Any number of CONTINUATION frames can be sent, as long as the preceding frame is on the same stream and is a HEADERS, PUSH\_PROMISE, or CONTINUATION frame without the END\_HEADERS flag set.

```
+-----+
|                                     ...
|      Header Block Fragment (*)
+-----+
```

Figure 15: CONTINUATION Frame Payload

CONTINUATION 帧(type=0x9)用于继续传送首部块片段序列(4.3节)。只要前面的帧在同一个流上，而且是一个没有设置END\_HEADERS标志的 HEADERS 帧，PUSH\_PROMISE 帧，或者CONTINUATION帧，就可以发送任意数量的CONTINUATION帧。

```
+-----+
|                                     ...
|      Header Block Fragment (*)
+-----+
```

图 15: CONTINUATION 帧负载

The CONTINUATION frame payload contains a header block fragment (Section 4.3).

CONTINUATION 帧的负载包含一个首部块片段(4.3节)。

The CONTINUATION frame defines the following flag:

- **END\_HEADERS (0x4):** When set, bit 2 indicates that this frame ends a header block (Section 4.3).

If the END\_HEADERS bit is not set, this frame MUST be followed by another CONTINUATION frame. A receiver MUST treat the receipt of any other type of frame or a frame on a different stream as a connection error (Section 5.4.1) of type **PROTOCOL\_ERROR**.

CONTINUATION 帧定义了如下标志：

- **END\_HEADERS (0x4):** 当设置了该标志，第2个bit位表示该帧是一个首部块的结束(4.3节)。

如果没有设置 END\_HEADERS bit位，该帧的后面必须跟有其他的CONTINUATION帧。如果接收端收到了任何其他类型的帧，或者另外一条流上的帧，必须将其当做类型为 **PROTOCOL\_ERROR** 的连接错误(5.4.1节)。



The CONTINUATION frame changes the connection state as defined in [Section 4.3](#).

CONTINUATION 帧改变了 [4.3 节](#) 定义的连接状态。

CONTINUATION frames MUST be associated with a stream. If a CONTINUATION frame is received whose stream identifier field is 0x0, the recipient MUST respond with a connection error ([Section 5.4.1](#)) of type `PROTOCOL_ERROR`.

CONTINUATION 帧必须和一个流相关联。如果收到了流标识符域的值 0x0 的 CONTINUATION 帧，接收端必须响应一个类型为 `PROTOCOL_ERROR` 的连接错误( [5.4.1 节](#) )。

A CONTINUATION frame MUST be preceded by a `HEADERS`, `PUSH_PROMISE` or CONTINUATION frame without the `END_HEADERS` flag set. A recipient that observes violation of this rule MUST respond with a connection error ([Section 5.4.1](#)) of type `PROTOCOL_ERROR`.

CONTINUATION 帧之前必须是一个没有设置 `END_HEADERS` 标志的 `HEADERS` 帧，`PUSH_PROMISE` 帧，或者 CONTINUATION 帧。如果接收端发现违反该规则了，必须响应一个类型为 `PROTOCOL_ERROR` 的连接错误( [5.4.1 节](#) )。

## Error codes / 错误码

Error codes are 32-bit fields that are used in [RST\\_STREAM](#) and [GOAWAY](#) frames to convey the reasons for the stream or connection error.

错误码是 [RST\\_STREAM](#) 帧和 [GOAWAY](#) 帧中32-bit位的域，用于表达流或连接错误的原因。

Error codes share a common code space. Some error codes apply only to either streams or the entire connection and have no defined semantics in the other context.

错误码共享一个公共的码空间。一些错误码要么只适用于流，要么只适用于整个连接，并且在其他的上下文中没有语义定义。

The following error codes are defined:

- **NO\_ERROR (0x0):** The associated condition is not a result of an error. For example, a [GOAWAY](#) might include this code to indicate graceful shutdown of a connection.
- **PROTOCOL\_ERROR (0x1):** The endpoint detected an unspecified protocol error. This error is for use when a more specific error code is not available.
- **INTERNAL\_ERROR (0x2):** The endpoint encountered an unexpected internal error.
- **FLOW\_CONTROL\_ERROR (0x3):** The endpoint detected that its peer violated the flow-control protocol.
- **SETTINGS\_TIMEOUT (0x4):** The endpoint sent a [SETTINGS](#) frame but did not receive a response in a timely manner. See [Section 6.5.3](#) ("Settings Synchronization").
- **STREAM\_CLOSED (0x5):** The endpoint received a frame after a stream was half-closed.
- **FRAME\_SIZE\_ERROR (0x6):** The endpoint received a frame with an invalid size.
- **REFUSED\_STREAM (0x7):** The endpoint refused the stream prior to performing any application processing (see [Section 8.1.4](#) for details).
- **CANCEL (0x8):** Used by the endpoint to indicate that the stream is no longer needed.
- **COMPRESSION\_ERROR (0x9):** The endpoint is unable to maintain the header compression context for the connection.
- **CONNECT\_ERROR (0xa):** The connection established in response to a [CONNECT](#) request ([Section 8.3](#)) was reset or abnormally closed.
- **ENHANCE\_YOUR\_CALM (0xb):** The endpoint detected that its peer is exhibiting a behavior that might be generating excessive load.
- **INADEQUATE\_SECURITY (0xc):** The underlying transport has properties that do not meet minimum security requirements (see [Section 9.2](#)).
- **HTTP\_1\_1\_REQUIRED (0xd):** The endpoint requires that HTTP/1.1 be used instead of HTTP/2.

定义了以下错误码：

- **NO\_ERROR (0x0):** 相关情况不是错误。例如，[GOAWAY](#) 帧可以包含该码，表示优雅地关闭连接。
- **PROTOCOL\_ERROR (0x1):** 端点检测到一个不确定的协议错误。当没有更加具体的错误码可用时，可以使用该错误码。
- **INTERNAL\_ERROR (0x2):** 端点遭遇到未知的内部错误。
- **FLOW\_CONTROL\_ERROR (0x3):** 端点检测到其对端违反了流量控制协议。
- **SETTINGS\_TIMEOUT (0x4):** 端点发送了一个 [SETTIGNS](#) 帧，但是没有及时收到响应。参见 [6.5.3节](#) ("同步设置")。

- **STREAM\_CLOSED (0x5):** 流半关闭以后，端点收到一个帧。
- **FRAME\_SIZE\_ERROR (0x6):** 端点收到了一个大小无效的帧。
- **REFUSED\_STREAM (0x7):** 在执行任何处理之前，端点拒绝了流(细节参见 [8.4.1节](#))。
- **CANCEL (0x8):** 被端点用于表示不再需要该流了。
- **COMPRESSION\_ERROR (0x9):** 端点不能为连接维持首部压缩上下文。
- **CONNECT\_ERROR (0xa):** 响应CONNECT请求( [8.3节](#) )而建立的连接被重置，或者被非正常关闭。
- **ENHANCE\_YOUR\_CALM (0xb):** 端点检测到对端正在展现出可能会产生极大负荷的行为。
- **INADEQUATE\_SECURITY (0xc):** 下层的传输层具有不满足最低安全要求( [9.2节](#) )的属性。
- **HTTP\_1\_1\_REQUIRED (0xd):** 端点要求使用HTTP/1.1代替HTTP/2。

Unknown or unsupported error codes MUST NOT trigger any special behavior. These MAY be treated by an implementation as being equivalent to [INTERNAL\\_ERROR](#).

未知的或者不支持的错误码不能触发任何特别的行为。实现可以将这些错误看做等价于 [INTERNAL\\_ERROR](#)。

## HTTP Message Exchanges / HTTP消息交换

HTTP/2 is intended to be as compatible as possible with current uses of HTTP. This means that, from the application perspective, the features of the protocol are largely unchanged. To achieve this, all request and response semantics are preserved, although the syntax of conveying those semantics has changed.

HTTP/2需要和当前使用的HTTP尽可能地兼容。这意味着，从应用程序的角度来看，大部分的协议特性不变。尽管表达这些语义的语法已经改变，为了达到这个目的，所有请求和响应的语义都被保留。

Thus, the specification and requirements of HTTP/1.1 Semantics and Content [\[RFC7231\]](#), Conditional Requests [\[RFC7232\]](#), Range Requests [\[RFC7233\]](#), Caching [\[RFC7234\]](#), and Authentication [\[RFC7235\]](#) are applicable to HTTP/2. Selected portions of HTTP/1.1 Message Syntax and Routing [\[RFC7230\]](#), such as the HTTP and HTTPS URI schemes, are also applicable in HTTP/2, but the expression of those semantics for this protocol are defined in the sections below.

这样，HTTP/1.1的语义和内容 [\[RFC7231\]](#)、条件请求 [\[RFC7232\]](#)、范围请求 [\[RFC7233\]](#)、缓存 [\[RFC7234\]](#) 和 认证 [\[RFC7235\]](#) 的协议规范和要求也适用于HTTP/2。HTTP/1.1消息语法和路由 [\[RFC7230\]](#) 选择的部分，如HTTP和HTTPS URI方案，也适用于HTTP/2，但是那些语义在HTTP/2协议中的表达在下面的章节有定义。

# HTTP Request/Response Exchange

A client sends an HTTP request on a new stream, using a previously unused stream identifier ([Section 5.1.1](#)). A server sends an HTTP response on the same stream as the request.

客户端在一个新流上发送一个HTTP请求，该新流使用了先前未使用的流标识符([5.1.1节](#))。服务端在同样的流上发送一个HTTP响应。

An HTTP message (request or response) consists of:

1. for a response only, zero or more **HEADERS** frames (each followed by zero or more **CONTINUATION** frames) containing the message headers of informational (1xx) HTTP responses (see [\[RFC7230\]](#), [Section 3.2](#) and [\[RFC7231\]](#), [Section 6.2](#)),
2. one **HEADERS** frame (followed by zero or more **CONTINUATION** frames) containing the message headers (see [\[RFC7230\]](#), [Section 3.2](#)),
3. zero or more **DATA** frames containing the payload body (see [\[RFC7230\]](#), [Section 3.3](#)), and
4. optionally, one **HEADERS** frame, followed by zero or more **CONTINUATION** frames containing the trailer-part, if present (see [\[RFC7230\]](#), [Section 4.1.2](#)).

一个HTTP消息(请求或响应)包括：

1. 仅仅对响应而言，零个或多个 **HEADERS** 帧(每个后面跟随零个或多个 **CONTINUATION** 帧)，这些帧包含信息性(1xx)HTTP响应的消息首部(参见 [\[RFC7230\]](#)，[3.2节](#)，和 [\[RFC7231\]](#)，[6.2节](#))，
2. 一个包含消息首部(参见 [\[RFC7230\]](#)，[3.2节](#))的 **HEADERS** 帧(后面跟随零个或多个 **CONTINUATION** 帧)，
3. 包含负载体(参见 [\[RFC7230\]](#)，[3.3节](#))的零个或多个 **DATA** 帧，和
4. 可选地，一个 **HEADERS** 帧，后面跟随零个或多个 **CONTINUATION** 帧，如果有(参见 [\[RFC7230\]](#)，[4.1.2节](#))，也包括拖挂部分。

The last frame in the sequence bears an **END\_STREAM** flag, noting that a **HEADERS** frame bearing the **END\_STREAM** flag can be followed by **CONTINUATION** frames that carry any remaining portions of the header block.

序列的最后一帧带有**END\_STREAM**标志，注意带有**END\_STREAM**标志的 **HEADERS** 帧后面可以跟装载首部块剩余部分的 **CONTINUATION** 帧。

Other frames (from any stream) MUST NOT occur between the **HEADERS** frame and any **CONTINUATION** frames that might follow.

无论是来自哪个流的其它的帧都不能出现在 **HEADERS** 帧和后面可能跟随的 **CONTINUATION** 帧之间。

HTTP/2 uses DATA frames to carry message payloads. The chunked transfer encoding defined in [Section 4.1](#) of [\[RFC7230\]](#) MUST NOT be used in HTTP/2.

HTTP/2使用DATA帧来传送消息负载。[\[RFC7230\] 4.1节](#) 定义的分块传输编码不能在HTTP/2中使用。

Trailing header fields are carried in a header block that also terminates the stream. Such a header block is a sequence starting with a **HEADERS** frame, followed by zero or more **CONTINUATION** frames, where the **HEADERS** frame bears an END\_STREAM flag. Header blocks after the first that do not terminate the stream are not part of an HTTP request or response.

拖尾首部字段在终结流的首部块里传送。这样的首部块是一个序列，从一个 **HEADERS** 帧开始，后跟零个或多个 **CONTINUATION** 帧，其中 **HEADERS** 帧携带一个END\_STREAM标志。没有终结流的首部块之后的首部块不属于HTTP请求或响应的一部分。

A **HEADERS** frame (and associated **CONTINUATION** frames) can only appear at the start or end of a stream. An endpoint that receives a **HEADERS** frame without the END\_STREAM flag set after receiving a final (non-informational) status code MUST treat the corresponding request or response as malformed ([Section 8.1.2.6](#)).

**HEADERS** 帧(和相关联的 **CONTINUATION** 帧)只能出现在流的开始或结束。如果端点在收到一个最终的(非信息性的)状态码以后又收到一个没有设置END\_STREAM标志的 **HEADERS** 帧，必须将相应的请求或响应当做是有缺陷的([8.1.2.6节](#))。

An HTTP request/response exchange fully consumes a single stream. A request starts with the **HEADERS** frame that puts the stream into an "open" state. The request ends with a frame bearing END\_STREAM, which causes the stream to become "half-closed (local)" for the client and "half-closed (remote)" for the server. A response starts with a **HEADERS** frame and ends with a frame bearing END\_STREAM, which places the stream in the "closed" state.

一个完整的HTTP请求/响应交换流程需要消耗单独的一个流。请求从 **HEADERS** 帧开始，将流置于 打开(open) 状态。请求以携带END\_STREAM标志的帧结束，从而对客户端来说，流进入 半关闭(本地)(half-closed (local)) 状态，对服务端来说，流进入 半关闭(远端)(half-closed (remote)) 状态。响应从一个 **HEADERS** 帧开始，以一个携带END\_STREAM标志的帧结束，然后流进入 关闭(closed) 状态。

An HTTP response is complete after the server sends — or the client receives — a frame with the `END_STREAM` flag set (including any `CONTINUATION` frames needed to complete a header block). A server can send a complete response prior to the client sending an entire request if the response does not depend on any portion of the request that has not been sent and received. When this is true, a server MAY request that the client abort transmission of a request without error by sending a `RST_STREAM` with an error code of `NO_ERROR` after sending a complete response (i.e., a frame with the `END_STREAM` flag). Clients MUST NOT discard responses as a result of receiving such a `RST_STREAM`, though clients can always discard responses at their discretion for other reasons.

服务端发送或者客户端接收一个设置了`END_STREAM`标志的帧(包括任何传送完成首部块的`CONTINUATION` 帧)以后，HTTP响应结束。如果响应完全不依赖于还未发送和接收的请求，那么在客户端发送整个请求之前，服务端可以发送一个完整的响应。这种情况下，发送完一个完整的响应(即，携带`END_STREAM`标志的帧)以后，通过发送一个错误码为 `NO_ERROR` 的 `RST_STREAM` 帧，服务端可以要求客户端取消传送请求。尽管客户端总是可以因为其它原因而按自己的意愿丢弃响应，但客户端不能因为收到了这样一个 `RST_STREAM` 帧就丢弃响应。



## Upgrading from HTTP/2 / 从HTTP/2升级

HTTP/2 removes support for the 101 (Switching Protocols) informational status code ([\[RFC7231\]](#), [Section 6.2.2](#)).

HTTP/2移除了对101(Switching Protocols)信息性状态码( [\[RFC7231\]](#) , [6.2.2节](#) )的支持。

The semantics of 101 (Switching Protocols) aren't applicable to a multiplexed protocol. Alternative protocols are able to use the same mechanisms that HTTP/2 uses to negotiate their use (see [Section 3](#)).

101(Switching Protocols)的语义不适用于多路复用的协议。替代协议可以使用跟HTTP/2相同的协商机制(参见 [第3章](#))。

# HTTP Header Fields / HTTP首部字段

HTTP header fields carry information as a series of key-value pairs. For a listing of registered HTTP headers, see the "Message Header Field" registry maintained at <https://www.iana.org/assignments/message-headers>.

HTTP首部字段以一系列键值对的形式携带信息。登记的HTTP首部列表，可以参考在<https://www.iana.org/assignments/message-headers> 处维护的"Message Header Field"记录。

Just as in HTTP/1.x, header field names are strings of ASCII characters that are compared in a case-insensitive fashion. However, header field names **MUST** be converted to lowercase prior to their encoding in HTTP/2. A request or response containing uppercase header field names **MUST** be treated as malformed ([Section 8.1.2.6](#)).

像在HTTP/1.x中一样，首部字段名称是ASCII字符串，不区分大小写。但是，在被HTTP/2编码之前，必须将首部字段名称转换成小写的。必须将包含大写首部字段名称的请求或响应看做是畸形的([8.1.2.6节](#))。

## Pseudo-Header Fields / 伪首部字段

While HTTP/1.x used the message start-line (see [\[RFC7230\]](#), [Section 3.1](#)) to convey the target URI, the method of the request, and the status code for the response, HTTP/2 uses special pseudo-header fields beginning with ':' character (ASCII 0x3a) for this purpose.

HTTP/1.x使用消息起始行([\[RFC7230\]](#)，[3.1节](#))表达目标URI。对于同样的目的，HTTP/2使用以':'字符(ASCII 0x3a)开始的特殊的伪首部字段来表示请求的方法和响应的状态码。

Pseudo-header fields are not HTTP header fields. Endpoints MUST NOT generate pseudo-header fields other than those defined in this document.

伪首部字段不是HTTP首部字段。端点不能生成本文档定义之外的伪首部字段。

Pseudo-header fields are only valid in the context in which they are defined. Pseudo-header fields defined for requests MUST NOT appear in responses; pseudo-header fields defined for responses MUST NOT appear in requests. Pseudo-header fields MUST NOT appear in trailers. Endpoints MUST treat a request or response that contains undefined or invalid pseudo-header fields as malformed ([Section 8.1.2.6](#)).

伪首部字段只在它们被定义的上下文中才有效。为请求定义的伪首部字段不能出现在响应中；为响应定义的伪首部字段不能出现在请求中。伪首部字段不能出现在拖尾中。如果请求或响应包含未定义的或无效的伪首部字段，端点必须将该请求或响应看做是有缺陷的([8.1.2.6节](#))。

All pseudo-header fields MUST appear in the header block before regular header fields. Any request or response that contains a pseudo-header field that appears in a header block after a regular header field MUST be treated as malformed ([Section 8.1.2.6](#)).

在首部块中，所有的伪首部字段都必须出现在正常的首部字段之前。如果请求或响应包含的伪首部字段在首部块中位于正常的首部字段之后，必须将该请求或响应看作是有缺陷的([8.1.2.6节](#))。

## Connection-Specific Header Fields / 连接专用的首部字段

HTTP/2 does not use the Connection header field to indicate connection-specific header fields; in this protocol, connection-specific metadata is conveyed by other means. An endpoint MUST NOT generate an HTTP/2 message containing connection-specific header fields; any message containing connection-specific header fields MUST be treated as malformed ([Section 8.1.2.6](#)).

HTTP/2不使用Connection首部字段来表示连接专用的首部字段；在本协议中，连接专用的元数据通过其他方式来表达。端点不能生成一个包含连接专用首部字段的HTTP/2消息；必须将任何包含连接专用首部字段的消息看做是有缺陷的([8.1.2.6节](#))。

The only exception to this is the TE header field, which MAY be present in an HTTP/2 request; when it is, it MUST NOT contain any value other than "trailers".

唯一的例外是TE首部字段，它可以出现在HTTP/2请求里，但它的值只能是"trailers"。

This means that an intermediary transforming an HTTP/1.x message to HTTP/2 will need to remove any header fields nominated by the Connection header field, along with the Connection header field itself. Such intermediaries SHOULD also remove other connection-specific header fields, such as Keep-Alive, Proxy-Connection, Transfer-Encoding, and Upgrade, even if they are not nominated by the Connection header field.

这意味着中介将一个HTTP/1.x消息转换成HTTP/2消息时，需要将所有的连接首部字段随同Connection首部字段一起移除。这样的中介也应该移除其他的连接专用首部字段，例如 Keep-Alive 、 Proxy-Connection 、 Transfer-Encoding 和 Upgrade ，即使这些首部字段不是以Connection命名的字段。

Note: HTTP/2 purposefully does not support upgrade to another protocol. The handshake methods described in [Section 3](#) are believed sufficient to negotiate the use of alternative protocols.

注意：HTTP/2有意地不支持通过upgrade首部字段转换到其它协议。[第3章](#)描述的握手方法协商使用其它替代协议时足够用了。

# Request Pseudo-Header Fields / 请求的伪首部字段

The following pseudo-header fields are defined for HTTP/2 requests:

- The `:method` pseudo-header field includes the HTTP method ([\[RFC7231\]](#), [Section 4](#)).
- The `:scheme` pseudo-header field includes the scheme portion of the target URI ([\[RFC3986\]](#), [Section 3.1](#)).

`:scheme` is not restricted to `http` and `https` schemes URIs. A proxy or gateway can translate requests for non-HTTP schemes, enabling the use of HTTP to interact with non-HTTP services.

- The `:authority` pseudo-header field includes the authority portion of the target URI ([\[RFC3986\]](#), [Section 3.2](#)). The authority MUST NOT include the deprecated `userinfo` subcomponent for `http` or `https` schemes URIs.

To ensure that the HTTP/1.1 request line can be reproduced accurately, this pseudo-header field MUST be omitted when translating from an HTTP/1.1 request that has a request target in origin or asterisk form (see [\[RFC7230\]](#), [Section 5.3](#)).

Clients that generate HTTP/2 requests directly SHOULD use the `:authority` pseudo-header field instead of the `Host` header field. An intermediary that converts an HTTP/2 request to HTTP/1.1 MUST create a `Host` header field if one is not present in a request by copying the value of the `:authority` pseudo-header field.

- The `:path` pseudo-header field includes the path and query parts of the target URI (the `path-absolute` production and optionally a `'?'` character followed by the query production (see [Sections 3.3](#) and [3.4](#) of [\[RFC3986\]](#)). A request in asterisk form includes the value `''` for the `:path` pseudo-header field.

This pseudo-header field MUST NOT be empty for `http` or `https` URIs; `http` or `https` URIs that do not contain a path component MUST include a value of `'/'`. The exception to this rule is an `OPTIONS` request for an `http` or `https` URI that does not include a path component; these MUST include a `:path` pseudo-header field with a value of `''` (see [\[RFC7230\]](#), [Section 5.3.4](#)).

HTTP/2请求定义了如下伪首部字段：

- `:method` 伪首部字段包含HTTP方法( [\[RFC7231\]](#)，第4章 )。
- `:scheme` 伪首部字段包含目标URI( [\[RFC3986\]](#)，3.1节 )的方案部分。

`:scheme` 并不局限于http和https URIs方案。代理或网关可以转化非HTTP方案的请求，从而可以使用HTTP和非HTTP的服务进行交互。

- `:authority` 伪首部字段包含目标URI( [\[RFC3986\]](#)，[3.2节](#) )的authority部分。authority不能包含http或https URIs方案废弃的用户信息的子组件。

为了确保HTTP/1.1请求队列可以被精确地复制，当转换具有源或星号形式(参见 [\[RFC7230\]](#)，[5.3节](#) )请求目标的HTTP/1.1请求时，该伪首部字段必须被忽略。直接生成HTTP/2请求的客户端应该使用 `:authority` 伪首部字段代替Host首部字段。中介将HTTP/2请求转换为HTTP/1.1请求时，如果通过拷贝 `:authority` 伪首部字段的值，请求里没有出现Host首部字段，就必须创建一个Host首部字段。

- `:path` 伪首部字段包含目标URI的path和query部分(绝对路径部分和一个后跟query部分的可选的'?'字符(参见 [\[RFC3986\]](#) 的 [3.3节](#) 和 [3.4节](#) ))。星号形式的请求包含`:path`伪首部字段的'\*'值。

对于http或者https URIs，该伪首部字段不能为空。不包含path组件的http或https URIs必须包含一个'/'值。该规则的例外情况是OPTIONS请求，其http或https URI不包含path组件，此时必须包含一个值为'\*'的 `:path` 伪首部字段(参见 [\[RFC7230\]](#)，[5.3.4节](#) )。

All HTTP/2 requests MUST include exactly one valid value for the `:method`, `:scheme`, and `:path` pseudo-header fields, unless it is a CONNECT request ([Section 8.3](#)). An HTTP request that omits mandatory pseudo-header fields is malformed ([Section 8.1.2.6](#)).

除非是CONNECT请求( [8.3节](#) )，否则对于 `:method` 、 `:scheme` 和 `:path` 伪首部字段，所有的HTTP/2请求都必须只包含一个有效值。忽略了强制的伪首部字段的HTTP请求是有缺陷的( [8.1.2.6节](#) )。

HTTP/2 does not define a way to carry the version identifier that is included in the HTTP/1.1 request line.

HTTP/2没有定义携带HTTP/1.1请求行里包含的版本号的方法。

## Response Pseudo-Header Fields / 响应的伪首部字段

For HTTP/2 responses, a single `:status` pseudo-header field is defined that carries the HTTP status code field (see [\[RFC7231\]](#), [Section 6](#)). This pseudo-header field **MUST** be included in all responses; otherwise, the response is malformed ([Section 8.1.2.6](#)).

对于HTTP/2响应，只定义了一个 `:status` 伪首部字段，其携带了HTTP状态码(参见[\[RFC7231\]](#)，[第6章](#))。所有的响应都必须包含该伪首部字段，否则，响应就是有缺陷的([8.1.2.6节](#))。

HTTP/2 does not define a way to carry the version or reason phrase that is included in an HTTP/1.1 status line.

HTTP/2没有定义携带HTTP/1.1状态行里包含的版本或者原因短语的方法。

# Compressing the Cookie Header Field / 压缩Cookie首部字段

The [Cookie header field](#) [COOKIE] uses a semi-colon (";") to delimit cookie-pairs (or "crumbs"). This header field doesn't follow the list construction rules in HTTP (see [\[RFC7230\]](#), [Section 3.2.2](#)), which prevents cookie-pairs from being separated into different name-value pairs. This can significantly reduce compression efficiency as individual cookie-pairs are updated.

[Cookie首部字段](#) [COOKIE] 使用分号来分隔cookie对(或『面包屑』)。该首部字段没有遵循HTTP里的列表创建规则(参见 [\[RFC7230\]](#)，[3.2.2节](#))，这阻止了将cookie对分隔为不同的名-值对。这会极大地降低cookie对更新时的压缩效率。

To allow for better compression efficiency, the Cookie header field MAY be split into separate header fields, each with one or more cookie-pairs. If there are multiple Cookie header fields after decompression, these MUST be concatenated into a single octet string using the two-octet delimiter of 0x3B, 0x20 (the ASCII string "; ") before being passed into a non-HTTP/2 context, such as an HTTP/1.1 connection, or a generic HTTP server application.

为了更好的压缩效率，可以将Cookie首部字段拆分成单独的首部字段，每一个都有一个或多个cookie对。如果解压缩后有多个Cookie首部字段，在将其传入一个非HTTP/2的上下文(比如：HTTP/1.1连接，或者通用的HTTP服务器应用)之前，必须使用两个字节的分隔符0x3B，0x20(即ASCII字符串"; ")将这些Cookie首部字段连接成一个字符串。

Therefore, the following two lists of Cookie header fields are semantically equivalent.

```
cookie: a=b; c=d; e=f

cookie: a=b
cookie: c=d
cookie: e=f
```

因此，以下两个Cookie首部字段列表在语义上是等价的：

```
cookie: a=b; c=d; e=f

cookie: a=b
cookie: c=d
cookie: e=f
```





# Malformed Requests and Responses / 有缺陷的请求和响应

A malformed request or response is one that is an otherwise valid sequence of HTTP/2 frames but is invalid due to the presence of extraneous frames, prohibited header fields, the absence of mandatory header fields, or the inclusion of uppercase header field names.

请求或响应本来是有效的HTTP/2帧序列，但由于出现了无关的帧，禁止的首部字段，缺少了强制的首部字段，或者包含了大写的首部字段名，有缺陷的请求或响应就变成了无效的HTTP/2帧序列。

A request or response that includes a payload body can include a content-length header field. A request or response is also malformed if the value of a content-length header field does not equal the sum of the [DATA](#) frame payload lengths that form the body. A response that is defined to have no payload, as described in [\[RFC7230\]](#), [Section 3.3.2](#), can have a non-zero content-length header field, even though no content is included in [DATA](#) frames.

包含负载体的请求或响应可以包含一个 content-length 首部字段。如果 content-length 首部字段的值不等于组成负载体的 [DATA](#) 帧负载的长度之和，请求或响应也是有缺陷的。被定义为没有负载的响应(就像 [\[RFC7230\]](#)，[3.3.2节](#) 里描述的那样)，即使 [DATA](#) 帧里没有内容，也可以有一个值为非零的 content-length 首部字段。

Intermediaries that process HTTP requests or responses (i.e., any intermediary not acting as a tunnel) MUST NOT forward a malformed request or response. Malformed requests or responses that are detected MUST be treated as a stream error ([Section 5.4.2](#)) of type [PROTOCOL\\_ERROR](#).

处理HTTP请求或响应的中介(即，任何不充当隧道的中介)不能转发有缺陷的请求或响应。必须把探测到的有缺陷的请求或响应当做类型为 [PROTOCOL\\_ERROR](#) 的流错误([5.4.2节](#))。

For malformed requests, a server MAY send an HTTP response prior to closing or resetting the stream. Clients MUST NOT accept a malformed response. Note that these requirements are intended to protect against several types of common attacks against HTTP; they are deliberately strict because being permissive can expose implementations to these vulnerabilities.

对于有缺陷的请求，服务端可以在关闭或重置流之前发送一个HTTP响应。客户端不能接受有缺陷的响应。注意，这些要求的目的是为了免受针对HTTP的几种常见的攻击；这些规则有意地严格，是因为可以避免暴露这些脆弱点的实现细节。



## Examples / 示例

This section shows HTTP/1.1 requests and responses, with illustrations of equivalent HTTP/2 requests and responses.

本节展示了HTTP/1.1请求和响应，及其等价的HTTP/2请求和响应实例。

An HTTP GET request includes request header fields and no payload body and is therefore transmitted as a single **HEADERS** frame, followed by zero or more **CONTINUATION** frames containing the serialized block of request header fields. The **HEADERS** frame in the following has both the **END\_HEADERS** and **END\_STREAM** flags set; no **CONTINUATION** frames are sent.

```
GET /resource HTTP/1.1      HEADERS
Host: example.org           ==>  + END_STREAM
Accept: image/jpeg          + END_HEADERS
                             :method = GET
                             :scheme = https
                             :path = /resource
                             host = example.org
                             accept = image/jpeg
```

HTTP GET请求包含请求首部字段，没有负载体，因此可以通过一个 **HEADERS** 帧来传输，后跟零个或多个包含序列化请求首部块的 **CONTINUATION** 帧。下面的 **HEADERS** 帧包含 **END\_HEADERS**和**END\_STREAM**标志，没有发送 **CONTINUATION** 帧。

```
GET /resource HTTP/1.1      HEADERS
Host: example.org           ==>  + END_STREAM
Accept: image/jpeg          + END_HEADERS
                             :method = GET
                             :scheme = https
                             :path = /resource
                             host = example.org
                             accept = image/jpeg
```

Similarly, a response that includes only response header fields is transmitted as a **HEADERS** frame (again, followed by zero or more **CONTINUATION** frames) containing the serialized block of response header fields.

```

HTTP/1.1 304 Not Modified      HEADERS
ETag: "xyzzzy"                ==>  + END_STREAM
Expires: Thu, 23 Jan ...      + END_HEADERS
                               :status = 304
                               etag = "xyzzzy"
                               expires = Thu, 23 Jan ...

```

类似地，只包含响应首部字段的响应通过一个 **HEADERS** 帧(同样，后跟零个或多个 **CONTINUATION** 帧)来传输，其包含序列化的响应首部块。

```

HTTP/1.1 304 Not Modified      HEADERS
ETag: "xyzzzy"                ==>  + END_STREAM
Expires: Thu, 23 Jan ...      + END_HEADERS
                               :status = 304
                               etag = "xyzzzy"
                               expires = Thu, 23 Jan ...

```

An HTTP POST request that includes request header fields and payload data is transmitted as one **HEADERS** frame, followed by zero or more **CONTINUATION** frames containing the request header fields, followed by one or more **DATA** frames, with the last **CONTINUATION** (or **HEADERS**) frame having the **END\_HEADERS** flag set and the final **DATA** frame having the **END\_STREAM** flag set:

```

POST /resource HTTP/1.1      HEADERS
Host: example.org            ==>  - END_STREAM
Content-Type: image/jpeg     - END_HEADERS
Content-Length: 123          :method = POST
                               :path = /resource
                               :scheme = https
{binary data}

CONTINUATION
+ END_HEADERS
content-type = image/jpeg
host = example.org
content-length = 123

DATA
+ END_STREAM
{binary data}

```

包含请求首部字段和负载数据的HTTP POST请求通过一个 **HEADERS** 帧传输，后跟零个或多个包含请求首部字段的 **CONTINUATION** 帧，后跟一个或多个 **DATA** 帧，最后的 **CONTINUATION** 帧(或者 **HEADERS** 帧)设置了END\_HEADERS标志，最后的 **DATA** 帧设置了END\_STREAM标志：

```

POST /resource HTTP/1.1
Host: example.org
Content-Type: image/jpeg
Content-Length: 123
{binary data}

==> HEADERS
      - END_STREAM
      - END_HEADERS
      :method = POST
      :path = /resource
      :scheme = https

      CONTINUATION
      + END_HEADERS
      content-type = image/jpeg
      host = example.org
      content-length = 123

      DATA
      + END_STREAM
      {binary data}

```

Note that data contributing to any given header field could be spread between header block fragments. The allocation of header fields to frames in this example is illustrative only.

注意，组成任何给定首部字段的数据可以在首部块片段之间散布。本例帧中首部字段的分配仅仅是示例性的。

A response that includes header fields and payload data is transmitted as a **HEADERS** frame, followed by zero or more **CONTINUATION** frames, followed by one or more **DATA** frames, with the last **DATA** frame in the sequence having the END\_STREAM flag set:

```

HTTP/1.1 200 OK
Content-Type: image/jpeg
Content-Length: 123
{binary data}

==> HEADERS
      - END_STREAM
      + END_HEADERS
      :status = 200
      content-type = image/jpeg
      content-length = 123

      DATA
      + END_STREAM
      {binary data}

```

包含首部字段和负载数据的响应通过一个 **HEADERS** 帧传输，后跟零个或多个 **CONTINUATION** 帧，后跟一个或多个 **DATA** 帧，序列的最后一个 **DATA** 帧设置了 **END\_STREAM** 标志：

```
HTTP/1.1 200 OK           HEADERS
Content-Type: image/jpeg  ==> - END_STREAM
Content-Length: 123       + END_HEADERS
                           :status = 200
                           content-type = image/jpeg
                           content-length = 123
{binary data}

                           DATA
                           + END_STREAM
                           {binary data}
```

An informational response using a 1xx status code other than 101 is transmitted as a **HEADERS** frame, followed by zero or more **CONTINUATION** frames.

一个使用除101之外的1xx状态码的信息性响应通过一个 **HEADERS** 帧传输，后跟零个或多个 **CONTINUATION** 帧。

Trailing header fields are sent as a header block after both the request or response header block and all the **DATA** frames have been sent. The **HEADERS** frame starting the trailers header block has the **END\_STREAM** flag set.

请求或响应和所有的 **DATA** 帧被发送完之后，拖尾的首部字段被当做一个首部块发送。开始拖尾首部块的 **HEADERS** 帧设置了 **END\_STREAM** 标志。

The following example includes both a 100 (Continue) status code, which is sent in response to a request containing a "100-continue" token in the Expect header field, and trailing header fields:

```
HTTP/1.1 100 Continue           HEADERS
Extension-Field: bar           ==>  - END_STREAM
                                + END_HEADERS
                                :status = 100
                                extension-field = bar

HTTP/1.1 200 OK                HEADERS
Content-Type: image/jpeg       ==>  - END_STREAM
Transfer-Encoding: chunked      + END_HEADERS
Trailer: Foo                   :status = 200
                                content-length = 123
                                content-type = image/jpeg
                                trailer = Foo

123
{binary data}
0
Foo: bar                       DATA
                                - END_STREAM
                                {binary data}

                                HEADERS
                                + END_STREAM
                                + END_HEADERS
                                foo = bar
```

如果请求的Expect首部字段里包含一个"100-continue"标识，那么应在其响应里发送一个100(Continue)状态码，下面的示例就包括了一个这样的状态码，还包括了拖尾的首部字段：



```
HTTP/1.1 100 Continue
Extension-Field: bar    ==>  HEADERS
                             - END_STREAM
                             + END_HEADERS
                             :status = 100
                             extension-field = bar

HTTP/1.1 200 OK
Content-Type: image/jpeg  ==>  HEADERS
Transfer-Encoding: chunked    - END_STREAM
Trailer: Foo                  + END_HEADERS
                               :status = 200
                               content-length = 123
                               content-type = image/jpeg
                               trailer = Foo

123
{binary data}
0
Foo: bar

DATA
- END_STREAM
{binary data}

HEADERS
+ END_STREAM
+ END_HEADERS
foo = bar
```

# Request Reliability Mechanisms in HTTP/2

## / HTTP/2里的请求可靠机制

In HTTP/1.1, an HTTP client is unable to retry a non-idempotent request when an error occurs because there is no means to determine the nature of the error. It is possible that some server processing occurred prior to the error, which could result in undesirable effects if the request were reattempted.

在HTTP/1.1里，发生错误时，HTTP客户端不能重试一个非幂等的请求，因为没有办法判定错误的性质。相比错误，一些服务端可能优先处理已经发生的请求，如果重试发生错误的请求，可能会导致不可预料的影响。

HTTP/2 provides two mechanisms for providing a guarantee to a client that a request has not been processed:

- The **GOAWAY** frame indicates the highest stream number that might have been processed. Requests on streams with higher numbers are therefore guaranteed to be safe to retry.
- The **REFUSED\_STREAM** error code can be included in a **RST\_STREAM** frame to indicate that the stream is being closed prior to any processing having occurred. Any request that was sent on the reset stream can be safely retried.

对于请求尚未被处理的客户端，HTTP/2提供了两种保障机制：

- **GOAWAY** 帧指出了可能已被处理的最大流号。因此，在流号更大的流上发送请求时，可以保证重试安全。
- **RST\_STREAM** 帧可以包含 **REFUSED\_STREAM** 错误码，表明未处理任何请求，流被关闭。在已被重置的流上发送的任何请求都可以安全地重试。

Requests that have not been processed have not failed; clients MAY automatically retry them, even those with non-idempotent methods.

未经处理的请求不算失败，客户端可以自动重试这些请求，即使其是非幂等的。

A server MUST NOT indicate that a stream has not been processed unless it can guarantee that fact. If frames that are on a stream are passed to the application layer for any stream, then **REFUSED\_STREAM** MUST NOT be used for that stream, and a **GOAWAY** frame MUST include a stream identifier that is greater than or equal to the given stream identifier.

服务端不能说明某个流未被处理，除非它能保证这一事实。如果某个流上的帧被传给应用层的任意一个流，那么 **REFUSED\_STREAM** 错误码不能用于该流，且 **GOAWAY** 帧包含的流标识符必须大于等于该给定流的标识符。

In addition to these mechanisms, the **PING** frame provides a way for a client to easily test a connection. Connections that remain idle can become broken as some middleboxes (for instance, network address translators or load balancers) silently discard connection bindings. The **PING** frame allows a client to safely test whether a connection is still active without sending a request.

除了这些机制之外，**PING** 帧还提供了让客户端方便地测试连接的方法。空闲连接会由于一些中间设备(例如，网络地址转换器或者负载均衡器)默默地丢弃连接绑定而断掉。**PING** 帧让客户端可以不用发送请求就能安全地测试连接是否仍然有效。

## Server Push / 服务端推送

HTTP/2 allows a server to pre-emptively send (or "push") responses (along with corresponding "promised" requests) to a client in association with a previous client-initiated request. This can be useful when the server knows the client will need to have those responses available in order to fully process the response to the original request.

HTTP/2允许服务端抢先向客户端发送(或『推送』)响应(以及相应的『被允诺的』请求), 这些响应跟先前客户端发起的请求有关。为了完整地处理对最初请求的响应, 客户端将需要服务端推送的响应, 当服务端了解到这一点时, 服务端推送功能就会是有用的。

A client can request that server push be disabled, though this is negotiated for each hop independently. The `SETTINGS_ENABLE_PUSH` setting can be set to 0 to indicate that server push is disabled.

客户端可以要求关闭服务端推送功能, 但这是每一跳独立协商地。`SETTINGS_ENABLE_PUSH` 设置为0, 表示关闭服务端推送功能。

Promised requests MUST be cacheable (see [\[RFC7231\]](#), Section 4.2.3), MUST be safe (see [\[RFC7231\]](#), Section 4.2.1), and MUST NOT include a request body. Clients that receive a promised request that is not cacheable, that is not known to be safe, or that indicates the presence of a request body MUST reset the promised stream with a stream error (Section 5.4.2) of type `PROTOCOL_ERROR`. Note this could result in the promised stream being reset if the client does not recognize a newly defined method as being safe.

被允诺的请求必须是可缓存的(参见 [\[RFC7231\]](#), 4.2.3节), 必须是安全的(参见 [\[RFC7231\]](#), 4.2.1节), 而且不能包含请求体。如果客户端收到的被允诺的请求是不可缓存的, 不安全的, 或者含有请求体, 那么客户端必须使用类型为 `PROTOCOL_ERROR` 的流错误(5.4.2节)来重置该被允诺的流。注意, 如果客户端认为一个新定义的方法是不安全的, 这将会导致被允诺的流被重置。

Pushed responses that are cacheable (see [\[RFC7234\]](#), Section 3) can be stored by the client, if it implements an HTTP cache. Pushed responses are considered successfully validated on the origin server (e.g., if the "no-cache" cache response directive is present ([\[RFC7234\]](#), Section 5.2.2)) while the stream identified by the promised stream ID is still open.

如果客户端实现了HTTP缓存功能, 就可以存储被推送的、可缓存的(参见 [\[RFC7234\]](#), 第3章)响应。被推送的响应被认为在源服务器上已被成功地验证过(例如, 如果存在"no-cache"缓存响应指令([\[RFC7234\]](#), 5.2.2节)), 而被允诺的流ID标识的流仍处于打开状态。

Pushed responses that are not cacheable MUST NOT be stored by any HTTP cache. They MAY be made available to the application separately.

任何HTTP缓存都不能存储不可缓存的被推送响应。这些响应可以单独提供给应用程序。

The server MUST include a value in the :authority pseudo-header field for which the server is authoritative (see [Section 10.1](#)). A client MUST treat a `PUSH_PROMISE` for which the server is not authoritative as a stream error ([Section 5.4.2](#)) of type `PROTOCOL_ERROR`.

服务端必须在 `:authority` 伪首部字段中包含一个值，以表明服务端是权威可信的(参见 [10.1 节](#))。客户端必须将不可信的服务端的 `PUSH_PROMISE` 帧当做类型为 `PROTOCOL_ERROR` 的流错误([5.4.2 节](#))。

An intermediary can receive pushes from the server and choose not to forward them on to the client. In other words, how to make use of the pushed information is up to that intermediary. Equally, the intermediary might choose to make additional pushes to the client, without any action taken by the server.

中介可以接收来自服务端的推送，并选择不向客户端转发这些推送。换句话说，怎样利用被推送来的信息取决于该中介。同样，中介可以选择向客户端做额外的推送，不需要服务端采取任何行动。

A client cannot push. Thus, servers MUST treat the receipt of a `PUSH_PROMISE` frame as a connection error ([Section 5.4.1](#)) of type `PROTOCOL_ERROR`. Clients MUST reject any attempt to change the `SETTINGS_ENABLE_PUSH` setting to a value other than 0 by treating the message as a connection error ([Section 5.4.1](#)) of type `PROTOCOL_ERROR`.

客户端不能推送。因此，如果服务端收到了 `PUSH_PROMISE` 帧，必须将其当做类型为 `PROTOCOL_ERROR` 的连接错误([5.4.1 节](#))。客户端必须拒绝任何将 `SETTING_ENABLE_PUSH` 改为非0值的行为，否则就将其当做类型为 `PROTOCOL_ERROR` 的连接错误([5.4.1 节](#))。

# Push Requests / 推送请求

Server push is semantically equivalent to a server responding to a request; however, in this case, that request is also sent by the server, as a [PUSH\\_PROMISE](#) frame.

服务端推送在语义上等价于服务端对请求的响应。但在本小节，请求也被服务端以 [PUSH\\_PROMISE](#) 帧的形式发送。

The [PUSH\\_PROMISE](#) frame includes a header block that contains a complete set of request header fields that the server attributes to the request. It is not possible to push a response to a request that includes a request body.

[PUSH\\_PROMISE](#) 帧包含一个首部块，该首部块包含完整的一套请求首部字段，服务端将这些字段归因于请求。不可能向包含请求体的请求推送响应。

Pushed responses are always associated with an explicit request from the client. The [PUSH\\_PROMISE](#) frames sent by the server are sent on that explicit request's stream. The [PUSH\\_PROMISE](#) frame also includes a promised stream identifier, chosen from the stream identifiers available to the server (see [Section 5.1.1](#)).

被推送的响应总是和明显来自于客户端的请求有关联。服务端在该请求所在的流上发送 [PUSH\\_PROMISE](#) 帧。[PUSH\\_PROMISE](#) 帧也包含一个被允诺的流标识符，该流标识符是从服务端可用的流标识符里选出来的([5.1.1节](#))。

The header fields in [PUSH\\_PROMISE](#) and any subsequent [CONTINUATION](#) frames MUST be a valid and complete set of request header fields ([Section 8.1.2.3](#)). The server MUST include a method in the :method pseudo-header field that is safe and cacheable. If a client receives a [PUSH\\_PROMISE](#) that does not include a complete and valid set of header fields or the :method pseudo-header field identifies a method that is not safe, it MUST respond with a stream error ([Section 5.4.2](#)) of type [PROTOCOL\\_ERROR](#).

[PUSH\\_PROMISE](#) 帧和任何随后的 [CONTINUATION](#) 帧里的首部字段必须是完整有效的一套请求首部字段([8.1.2.3节](#))。服务端必须在 :method 伪首部字段里包含一个安全的可缓存的方法。如果客户端收到了一个 [PUSH\\_PROMISE](#) 帧，其没有包含完整有效的一套首部字段，或者 :method 伪首部字段标识了一个不安全的方法，就必须响应一个类型为 [PROTOCOL\\_ERROR](#) 的流错误([5.4.2节](#))。

The server SHOULD send [PUSH\\_PROMISE](#) ([Section 6.6](#)) frames prior to sending any frames that reference the promised responses. This avoids a race where clients issue requests prior to receiving any [PUSH\\_PROMISE](#) frames.

在发送任何跟被允诺的响应有关联的帧之前，服务端应该优先发送 **PUSH\_PROMISE** 帧( 6.6 节)。这就避免了一种竞态条件，即客户端在收到任何 **PUSH\_PROMISE** 帧之前就发送请求。

For example, if the server receives a request for a document containing embedded links to multiple image files and the server chooses to push those additional images to the client, sending **PUSH\_PROMISE** frames before the **DATA** frames that contain the image links ensures that the client is able to see that a resource will be pushed before discovering embedded links. Similarly, if the server pushes responses referenced by the header block (for instance, in Link header fields), sending a **PUSH\_PROMISE** before sending the header block ensures that clients do not request those resources.

例如，如果服务端收到了一个对文档的请求，该文档包含内嵌的指向多个图片文件的链接，且服务端选择向客户端推送那些额外的图片，那么在发送包含图片链接的 **DATA** 帧之前发送 **PUSH\_PROMISE** 帧可以确保客户端在发现内嵌的链接之前，能够知道有一个资源将要被推送过来。同样地，如果服务端推送被首部块引用的响应(比如，在链接的首部字段里)，在发送首部块之前发送一个 **PUSH\_PROMISE** 帧，可以确保客户端不再请求那些资源。

**PUSH\_PROMISE** frames MUST NOT be sent by the client.

客户端不能发送 **PUSH\_PROMISE** 帧。

**PUSH\_PROMISE** frames can be sent by the server in response to any client-initiated stream, but the stream MUST be in either the "open" or "half-closed (remote)" state with respect to the server. **PUSH\_PROMISE** frames are interspersed with the frames that comprise a response, though they cannot be interspersed with **HEADERS** and **CONTINUATION** frames that comprise a single header block.

服务端可以发送 **PUSH\_PROMISE** 帧，以响应任何客户端发起的流，但是相对于服务端该流必须处于 打开(open) 或者 半关闭(远端)(half-closed(remote)) 状态。**PUSH\_PROMISE** 帧之间夹杂着包含响应的帧，但不能夹杂着包含一个单独的首部块的 **HEADERS** 帧 和 **CONTINUATION** 帧。

Sending a **PUSH\_PROMISE** frame creates a new stream and puts the stream into the "reserved (local)" state for the server and the "reserved (remote)" state for the client.

发送一个 **PUSH\_PROMISE** 帧会创建一个新流。在服务端，该流会进入 被保留的(本地)(reserved (local)) 状态；在客户端，该流会进入 被保留的(远端)(reserved (remote)) 状态。



## Push Responses / 推送响应

After sending the `PUSH_PROMISE` frame, the server can begin delivering the pushed response as a response (Section 8.1.2.4) on a server-initiated stream that uses the promised stream identifier. The server uses this stream to transmit an HTTP response, using the same sequence of frames as defined in Section 8.1. This stream becomes "half-closed" to the client (Section 5.1) after the initial `HEADERS` frame is sent.

在发送 `PUSH_PROMISE` 帧以后，在一个服务端发起的、使用被允诺的流标识符的流上，服务端可以开始将被推送的响应(8.1.2.4节)当做响应传送。服务端使用该流传送HTTP响应，帧顺序跟8.1节定义的相同。在初始 `HEADERS` 帧被发送以后，对客户端(5.1节)来说，该流变为半关闭(half-closed)状态。

Once a client receives a `PUSH_PROMISE` frame and chooses to accept the pushed response, the client SHOULD NOT issue any requests for the promised response until after the promised stream has closed.

一旦客户端收到了 `PUSH_PROMISE` 帧，并选择接收被推送的响应，客户端就不应该为被允诺的响应发送任何请求，直到被允诺的流被关闭以后。

If the client determines, for any reason, that it does not wish to receive the pushed response from the server or if the server takes too long to begin sending the promised response, the client can send a `RST_STREAM` frame, using either the `CANCEL` or `REFUSED_STREAM` code and referencing the pushed stream's identifier.

不管出于什么原因，如果客户端决定不再从服务端接收被推送的响应，或者如果服务端花费了太长时间准备发送被允诺的响应，客户端可以发送一个 `RST_STREAM` 帧，该帧可以使用 `CANCEL` 或者 `REFUSED_STREAM` 码，并引用被推送的流标识符。

A client can use the `SETTINGS_MAX_CONCURRENT_STREAMS` setting to limit the number of responses that can be concurrently pushed by a server. Advertising a `SETTINGS_MAX_CONCURRENT_STREAMS` value of zero disables server push by preventing the server from creating the necessary streams. This does not prohibit a server from sending `PUSH_PROMISE` frames; clients need to reset any promised streams that are not wanted.

客户端可以使用 `SETTINGS_MAX_CONCURRENT_STREAMS` 设置项来限制服务端并行推送响应的数量。通告 `SETTINGS_MAX_CONCURRENT_STREAMS` 的值为零，会通过阻止服务端创建必要的流的方式来关闭服务端推送功能。这不会阻止服务端发送 `PUSH_PROMISE` 帧；客户端需要重置任何不想要的被允诺的流。



Clients receiving a pushed response MUST validate that either the server is authoritative (see [Section 10.1](#)) or the proxy that provided the pushed response is configured for the corresponding request. For example, a server that offers a certificate for only the example.com DNS-ID or Common Name is not permitted to push a response for <https://www.example.org/doc>.

收到被推送响应的客户端必须确认，要么服务端是可信的(参见 [10.1 节](#))，要么提供被推送响应的代理为相应的请求做了配置。例如，只为 example.com 的DNS-ID或者域名提供证书的服务端不准向 <https://www.example.org/doc> 推送响应。

The response for a **PUSH\_PROMISE** stream begins with a **HEADERS** frame, which immediately puts the stream into the "half-closed (remote)" state for the server and "half-closed (local)" state for the client, and ends with a frame bearing **END\_STREAM**, which places the stream in the "closed" state.

**PUSH\_PROMISE** 流的响应以 **HEADERS** 帧开始，这会立即将流在服务端置于 半关闭(远端)(half-closed(remote)) 状态，在客户端置于 半关闭(本地)(half-closed(local)) 状态，最后以携带 **END\_STREAM** 的帧结束，这会将流置于 关闭(closed) 状态。

Note: The client never sends a frame with the **END\_STREAM** flag for a server push.

注意：客户端从来不会为服务端推送发送带有 **END\_STREAM** 标志的帧。

# The CONNECT Method / CONNECT 方法

In HTTP/1.x, the pseudo-method CONNECT ([\[RFC7231\]](#), [Section 4.3.6](#)) is used to convert an HTTP connection into a tunnel to a remote host. CONNECT is primarily used with HTTP proxies to establish a TLS session with an origin server for the purposes of interacting with https resources.

在HTTP/1.x里，CONNECT([\[RFC7231\]](#), [4.3.6节](#)) 伪方法用于将和远端主机的HTTP连接转换为隧道。CONNECT 主要用于HTTP代理和源服务器建立TLS会话，其目的是和https资源交互。

In HTTP/2, the CONNECT method is used to establish a tunnel over a single HTTP/2 stream to a remote host for similar purposes. The HTTP header field mapping works as defined in [Section 8.1.2.3](#) ("Request Pseudo-Header Fields"), with a few differences. Specifically:

- The `:method` pseudo-header field is set to CONNECT.
- The `:scheme` and `:path` pseudo-header fields MUST be omitted.
- The `:authority` pseudo-header field contains the host and port to connect to (equivalent to the authority-form of the request-target of CONNECT requests (see [\[RFC7230\]](#), [Section 5.3](#))).

在HTTP/2中，CONNECT 方法用于在一个到远端主机的单独的HTTP/2流之上建立隧道。HTTP首部字段映射以像在 [8.1.2.3节](#)([请求的伪首部字段](#)) 定义的那样起作用，但有一些不同，即：

- `:method` 伪首部字段设置为 CONNECT。
- 必须忽略 `:scheme` 和 `:path` 伪首部字段。
- `:authority` 伪首部字段包含要连接的主机和端口(等价于 CONNECT 请求目标的 `authority`形式([\[RFC7230\]](#), [5.3节](#)))。

A CONNECT request that does not conform to these restrictions is malformed ([Section 8.1.2.6](#)).

不符合这些限制的 CONNECT 请求是有缺陷的([8.1.2.6节](#))。

A proxy that supports CONNECT establishes a [TCP connection](#) [*TCP*] to the server identified in the `:authority` pseudo-header field. Once this connection is successfully established, the proxy sends a [HEADERS](#) frame containing a 2xx series status code to the client, as defined in [\[RFC7231\]](#), [Section 4.3.6](#).

支持 CONNECT 的代理建立到服务端的 [TCP连接 \[TCP\]](#)，连接靠 `:authority` 伪首部字段进行区分。一旦连接成功建立，代理就向客户端发送一个包含2xx系列状态码的 [HEADERS](#) 帧，正如 [\[RFC7231\]](#) 的 4.3.6节 定义的那样。

After the initial [HEADERS](#) frame sent by each peer, all subsequent [DATA](#) frames correspond to data sent on the TCP connection. The payload of any [DATA](#) frames sent by the client is transmitted by the proxy to the TCP server; data received from the TCP server is assembled into [DATA](#) frames by the proxy. Frame types other than [DATA](#) or stream management frames ([RST\\_STREAM](#), [WINDOW\\_UPDATE](#), and [PRIORITY](#)) MUST NOT be sent on a connected stream and MUST be treated as a stream error ([Section 5.4.2](#)) if received.

两端发送完初始的 [HEADERS](#) 帧以后，所有随后的 [DATA](#) 帧就对应于在TCP连接上发送的数据。客户端发送的任意 [DATA](#) 帧的负载通过代理传送给TCP服务器；从TCP服务器收到的数据被代理组装成 [DATA](#) 帧。不能在处于连接状态的流上发送除 [DATA](#) 帧或流管理帧 ([RST\\_STREAM](#), [WINDOW\\_UPDATE](#), 和 [PRIORITY](#))之外的帧类型，如果收到了这样的帧，必须将其当做流错误( [5.4.2节](#) )。

The TCP connection can be closed by either peer. The [END\\_STREAM](#) flag on a [DATA](#) frame is treated as being equivalent to the TCP FIN bit. A client is expected to send a [DATA](#) frame with the [END\\_STREAM](#) flag set after receiving a frame bearing the [END\\_STREAM](#) flag. A proxy that receives a [DATA](#) frame with the [END\\_STREAM](#) flag set sends the attached data with the FIN bit set on the last TCP segment. A proxy that receives a TCP segment with the FIN bit set sends a [DATA](#) frame with the [END\\_STREAM](#) flag set. Note that the final TCP segment or [DATA](#) frame could be empty.

TCP连接可以由任意一端关闭。[DATA](#) 帧上的 [END\\_STREAM](#) 标志被当做等价于TCP的FIN比特。收到一个带有 [END\\_STREAM](#) 标志的帧以后，客户端应该发送一个设置了 [END\\_STREAM](#) 标志的 [DATA](#) 帧。如果代理收到了设置有 [END\\_STREAM](#) 标志的 [DATA](#) 帧，就发送附加的数据，并在最后一个TCP片段上设置了 FIN 标志位。如果代理收到了一个设置了 FIN 标志位的TCP片段，就发送一个设置了 [END\\_STREAM](#) 标志的 [DATA](#) 帧。注意，最后的TCP片段或者 [DATA](#) 帧可以为空。

A TCP connection error is signaled with [RST\\_STREAM](#). A proxy treats any error in the TCP connection, which includes receiving a TCP segment with the RST bit set, as a stream error ([Section 5.4.2](#)) of type [CONNECT\\_ERROR](#). Correspondingly, a proxy MUST send a TCP segment with the RST bit set if it detects an error with the stream or the HTTP/2 connection.

TCP连接错误用 [RST\\_STREAM](#) 帧表示。代理将TCP连接中的任意错误，包括收到设置了 RST标志位的TCP片段，都当做类型为 [CONNECT\\_ERROR](#) 的流错误( [5.4.2节](#) )。相应地，如果代理探测到了流错误或者HTTP/2连接错误，就必须发送一个设置了RST标志位的TCP片

段。

## Additional HTTP Requirements/Considerations / 额外的 HTTP 要求或考虑

This section outlines attributes of the HTTP protocol that improve interoperability, reduce exposure to known security vulnerabilities, or reduce the potential for implementation variation.

该部分概述的HTTP协议特性，有助于改善互操作性，减少曝光已知的安全薄弱点，或者减少潜在的实施变量。

# Connection Management / 连接管理

HTTP/2 connections are persistent. For best performance, it is expected that clients will not close connections until it is determined that no further communication with a server is necessary (for example, when a user navigates away from a particular web page) or until the server closes the connection.

HTTP/2连接是持久连接。为了获得最佳性能，人们期望客户端不要关闭连接，直到确定不需要再继续和服务端进行通信(例如，当用户离开一个特定的web页面时)，或者直到服务端关闭连接。

Clients SHOULD NOT open more than one HTTP/2 connection to a given host and port pair, where the host is derived from a URI, a selected [alternative service \[ALT-SVC\]](#), or a configured proxy.

对于给定的一对主机和端口，客户端应该只打开一个 HTTP/2 连接，其中主机或者来自于一个 URI，一种选定的 [可替换的服务 \[ALT-SVC\]](#)，或者来自于一个已配置的代理。

A client can create additional connections as replacements, either to replace connections that are near to exhausting the available stream identifier space ([Section 5.1.1](#)), to refresh the keying material for a TLS connection, or to replace connections that have encountered errors ([Section 5.4.1](#)).

客户端可以创建额外的连接作为替补，或者用来替换将要耗尽可用流标识符空间([5.1.1节](#))的连接，为一个 TLS 连接刷新关键资料，或者用来替换遇到错误([5.4.1节](#))的连接。

A client MAY open multiple connections to the same IP address and TCP port using different [Server Name Indication \[TLS-EXT\]](#) values or to provide different TLS client certificates but SHOULD avoid creating multiple connections with the same configuration.

客户端可以和相同的 IP 地址和 TCP 端口建立多个连接，这些地址和端口使用不同的[服务器名称指示 \[TLS-EXT\]](#) 值，或者提供不同的 TLS 客户端证书。但是应该避免使用相同的配置创建多个连接。

Servers are encouraged to maintain open connections for as long as possible but are permitted to terminate idle connections if necessary. When either endpoint chooses to close the transport-layer TCP connection, the terminating endpoint SHOULD first send a [GOAWAY \(Section 6.8\)](#) frame so that both endpoints can reliably determine whether previously sent frames have been processed and gracefully complete or terminate any necessary remaining tasks.

鼓励服务端尽可能长地维持打开的连接，但是准许服务端在必要时可以关闭空闲的连接。当任何一端选择关闭传输层 TCP 连接时，主动关闭的一端应该先发送一个 **GOAWAY** 帧(6.8节)，这样两端都可以确定地知道先前发送的帧是否已被处理，并且优雅地完成或者终结任何剩下的必要任务。

# Connection Reuse

Connections that are made to an origin server, either directly or through a tunnel created using the CONNECT method (Section 8.3), MAY be reused for requests with multiple different URI authority components. A connection can be reused as long as the origin server is authoritative (Section 10.1). For TCP connections without TLS, this depends on the host having resolved to the same IP address.

For https resources, connection reuse additionally depends on having a certificate that is valid for the host in the URI. The certificate presented by the server MUST satisfy any checks that the client would perform when forming a new TLS connection for the host in the URI.

An origin server might offer a certificate with multiple subjectAltName attributes or names with wildcards, one of which is valid for the authority in the URI. For example, a certificate with a subjectAltName of \*.example.com might permit the use of the same connection for requests to URIs starting with <https://a.example.com/> and <https://b.example.com/>.

In some deployments, reusing a connection for multiple origins can result in requests being directed to the wrong origin server. For example, TLS termination might be performed by a middlebox that uses the TLS Server Name Indication (SNI) [TLS-EXT] extension to select an origin server. This means that it is possible for clients to send confidential information to servers that might not be the intended target for the request, even though the server is otherwise authoritative.

A server that does not wish clients to reuse connections can indicate that it is not authoritative for a request by sending a 421 (Misdirected Request) status code in response to the request (see Section 9.1.2).

A client that is configured to use a proxy over HTTP/2 directs requests to that proxy through a single connection. That is, all requests sent via a proxy reuse the connection to the proxy.



# The 421 (Misdirected Request) Status Code

The 421 (Misdirected Request) status code indicates that the request was directed at a server that is not able to produce a response. This can be sent by a server that is not configured to produce responses for the combination of scheme and authority that are included in the request URI.

Clients receiving a 421 (Misdirected Request) response from a server MAY retry the request — whether the request method is idempotent or not — over a different connection. This is possible if a connection is reused (Section 9.1.1) or if an alternative service is selected [ALT-SVC].

This status code MUST NOT be generated by proxies.

A 421 response is cacheable by default, i.e., unless otherwise indicated by the method definition or explicit cache controls (see Section 4.2.2 of [RFC7234]).

## Use of TLS Features

Implementations of HTTP/2 MUST use TLS version 1.2 [TLS12] or higher for HTTP/2 over TLS. The general TLS usage guidance in [TLSBCP] SHOULD be followed, with some additional restrictions that are specific to HTTP/2.

The TLS implementation MUST support the Server Name Indication (SNI) [TLS-EXT] extension to TLS. HTTP/2 clients MUST indicate the target domain name when negotiating TLS.

Deployments of HTTP/2 that negotiate TLS 1.3 or higher need only support and use the SNI extension; deployments of TLS 1.2 are subject to the requirements in the following sections. Implementations are encouraged to provide defaults that comply, but it is recognized that deployments are ultimately responsible for compliance.

# TLS 1.2 Features

This section describes restrictions on the TLS 1.2 feature set that can be used with HTTP/2. Due to deployment limitations, it might not be possible to fail TLS negotiation when these restrictions are not met. An endpoint MAY immediately terminate an HTTP/2 connection that does not meet these TLS requirements with a connection error (Section 5.4.1) of type `INADEQUATE_SECURITY`.

A deployment of HTTP/2 over TLS 1.2 MUST disable compression. TLS compression can lead to the exposure of information that would not otherwise be revealed [RFC3749]. Generic compression is unnecessary since HTTP/2 provides compression features that are more aware of context and therefore likely to be more appropriate for use for performance, security, or other reasons.

A deployment of HTTP/2 over TLS 1.2 MUST disable renegotiation. An endpoint MUST treat a TLS renegotiation as a connection error (Section 5.4.1) of type `PROTOCOL_ERROR`. Note that disabling renegotiation can result in long-lived connections becoming unusable due to limits on the number of messages the underlying cipher suite can encipher.

An endpoint MAY use renegotiation to provide confidentiality protection for client credentials offered in the handshake, but any renegotiation MUST occur prior to sending the connection preface. A server SHOULD request a client certificate if it sees a renegotiation request immediately after establishing a connection.

This effectively prevents the use of renegotiation in response to a request for a specific protected resource. A future specification might provide a way to support this use case. Alternatively, a server might use an error (Section 5.4) of type `HTTP_1_1_REQUIRED` to request the client use a protocol that supports renegotiation.

Implementations MUST support ephemeral key exchange sizes of at least 2048 bits for cipher suites that use ephemeral finite field Diffie-Hellman (DHE) [TLS12] and 224 bits for cipher suites that use ephemeral elliptic curve Diffie-Hellman (ECDHE) [RFC4492]. Clients MUST accept DHE sizes of up to 4096 bits. Endpoints MAY treat negotiation of key sizes smaller than the lower limits as a connection error (Section 5.4.1) of type `INADEQUATE_SECURITY`.

# TLS 1.2 Cipher Suites

A deployment of HTTP/2 over TLS 1.2 SHOULD NOT use any of the cipher suites that are listed in the cipher suite black list (Appendix A).

Endpoints MAY choose to generate a connection error (Section 5.4.1) of type INADEQUATE\_SECURITY if one of the cipher suites from the black list is negotiated. A deployment that chooses to use a black-listed cipher suite risks triggering a connection error unless the set of potential peers is known to accept that cipher suite.

Implementations MUST NOT generate this error in reaction to the negotiation of a cipher suite that is not on the black list. Consequently, when clients offer a cipher suite that is not on the black list, they have to be prepared to use that cipher suite with HTTP/2.

The black list includes the cipher suite that TLS 1.2 makes mandatory, which means that TLS 1.2 deployments could have non-intersecting sets of permitted cipher suites. To avoid this problem causing TLS handshake failures, deployments of HTTP/2 that use TLS 1.2 MUST support TLS\_ECDHE\_RSA\_WITH\_AES\_128\_GCM\_SHA256 [TLS-ECDHE] with the P-256 elliptic curve [FIPS186].

Note that clients might advertise support of cipher suites that are on the black list in order to allow for connection to servers that do not support HTTP/2. This allows servers to select HTTP/1.1 with a cipher suite that is on the HTTP/2 black list. However, this can result in HTTP/2 being negotiated with a black-listed cipher suite if the application protocol and cipher suite are independently selected.

























































