HokoFly's blog

QUIC传输格式规范(中文)

2018-04-08 | 协议 |

介绍(Introduction)

QUIC (Quick UDP Internet Connection) is a new multiplexed and secure transport atop UDP, designed from the ground up and optimized for HTTP/2 semantics. While built with HTTP/2 as the primary application protocol, QUIC builds on decades of transport and security experience, and implements mechanisms that make it attractive as a modern general-purpose transport. QUIC provides multiplexing and flow control equivalent to HTTP/2, security equivalent to TLS, and connection semantics, reliability, and congestion control equivalent to TCP.

QUIC (Quick UDP Internet Connection，快速UDP互联网连接) 是一个新的基于UDP的多路复用且安全的传输协议，它从头开始设计，且为 HTTP/2 语义做了优化。尽管以 HTTP/2 作为主要的应用协议而构建，然而 QUIC 的构建是基于传输和安全领域数十年的经验的，且实现了使它成为有吸引力的现代通用传输协议的机制。QUIC提供了等价于HTTP/2 的多路复用和流控，等价于 TLS 的安全机制，及等价于 TCP 的连接语义、可靠性和拥塞控制。

QUIC operates entirely in userspace, and is currently shipped to users as a part of the Chromium browser, enabling rapid deployment and experimentation. As a userspace transport atop UDP, QUIC allows innovations which have proven difficult to deploy with existing protocols as they are hampered by legacy clients and middleboxes, or by prolonged Operating System development and deployment cycles.

QUIC完全运行于用户空间，它当前作为 Chromium 浏览器的一部分发布给用户，以便于快速的部署和实验。作为基于 UDP 的用户空间传输协议，QUIC 可以做一些由于遗留的客户端和中间设备，或旷日持久的操作系统开发和部署周期的阻碍，而被证明很难在现有的协议中部署的创新。

An important goal for QUIC is to inform better transport design through rapid experimentation. As a result, we hope to inform and where possible migrate distilled changes into TCP and TLS, which tend to have much longer iteration cycles.

QUIC 的一个重要目标是通过快速的实验获得更好的传输设计相关的知识。作为结果，我们希望将其中的一些精华的改动迁移进 TCP 和 TLS，后者通常有着长得多的迭代周期。

This document describes the conceptual design and the wire specification of the QUIC protocol prior to standardization. Accompanying documents describe the combined crypto and transport handshake [QUIC-CRYPTO], and loss recovery and congestion control [draft-iyengar-quic-loss-recovery]. Additional resources, including a more detailed rationale document, are available on the Chromium QUIC webpage.

这份文档描述标准化前 QUIC 协议的概念设计和协议规范。补充资料描述了加密和传输握手 [QUIC-CRYPTO]，及丢失恢复和拥塞控制 [draft-iyengar-quic-loss-recovery]。其它资源，包括一份更详细的相关文档，可以在 Chromium 的 QUIC 主页 找到。

Proposals for standardization of QUIC based on this early deployment are [draft-hamilton-quic-transport-protocol], [draft-shade-quic-http2-mapping], [draft-iyengar-quic-loss-recovery], and [draft-thomson-quic-tls].

基于早期的部署的 QUIC 标准化建议为 [draft-hamilton-quic-transport-protocol]，[draft-shade-quic-http2-mapping]，[draft-iyengar-quic-loss-recovery]，和 [draft-thomson-quic-tls]。

术语和定义(Conventions and Definitions)

All integer values used in QUIC, including length, version, and type, are in little-endian byte order, and not in network byte order. QUIC does not enforce alignment of types in dynamically sized frames.

A few terms that are used throughout this document are defined below.

- “Client”: The endpoint initiating a QUIC connection.
- “Server”: The endpoint accepting incoming QUIC connections.
- “Endpoint”: The client or server end of a connection.
- “Stream”: A bi-directional flow of bytes across a logical channel within a QUIC connection.
- “Connection”: A conversation between two QUIC endpoints with a single encryption context that multiplexes streams within it.
- “Connection ID”: The identifier for a QUIC connection.
- “QUIC Packet”: A well-formed UDP payload that can be parsed by a QUIC receiver. QUIC packet size in this document refers to the UDP payload size.

QUIC中使用的所有整型值，包括长度、版本号和类型，都是小尾端字节序，而不是网络字节序。QUIC不强制动态大小的帧中的类型对齐。

贯穿本文档使用的一些术语定义如下。

- “客户端”：初始化 QUIC 连接的端点。
- “服务器”：接受进入的 QUIC 连接的端点。
- “端点”：连接的客户端或服务器端。
- “流”：QUIC 连接中穿过一个逻辑通道的双向字节流。
- “连接”：两个 QUIC 端点之间的会话，它具有一个单独的加密上下文且包含多路复用流。
- “连接ID”：QUIC 连接的标识符。
- “QUIC包”：经过良好格式化的 UDP 载荷，可由 QUIC 接收者解析。本文档中的 QUIC 包大小指 UDP 载荷大小。

QUIC概述(A QUIC Overview)

We now briefly describe QUIC’s key mechanisms and benefits. QUIC is functionally equivalent to TCP+TLS+HTTP/2, but implemented on top of UDP. Key advantages of QUIC over TCP+TLS+HTTP/2 include:

- Connection establishment latency
- Flexible congestion control
- Multiplexing without head-of-line blocking
- Authenticated and encrypted header and payload
- Stream and connection flow control
- Connection migration

我们现在简要介绍 QUIC 的关键机制和优势。QUIC 在功能上等价于 TCP + TLS + HTTP/2，但基于 UDP 实现。QUIC 相对于 TCP + TLS + HTTP/2 的主要优势包括：

- 连接建立延迟
- 灵活的拥塞控制
- 多路复用而不存在队首阻塞
- 认证和加密的首部和载荷
- 流和连接的流量控制
- 连接迁移

连接建立延迟(Connection Establishment Latency)

QUIC combines the crypto and transport handshakes, reducing the number of roundtrips required for setting up a secure connection. QUIC connections are commonly 0-RTT, meaning that on most QUIC connections, data can be sent immediately without waiting for a reply from the server, as compared to the 1-3 roundtrips required for TCP+TLS before application data can be sent.

QUIC provides a dedicated stream (Stream ID 1) to be used for performing the handshake, but the details of this handshake protocol are out of this document's scope. For a complete description of the current handshake protocol, please see the QUIC Crypto Handshake document. QUIC current handshake will be replaced by TLS 1.3 in the future.

QUIC将加密和传输握手结合在一起，减少了建立一条安全连接所需的往返。QUIC 连接通常是 0-RTT，意味着相比于 TCP + TLS 中发送应用数据前需要 1-3 个往返的情况，在大多数 QUIC 连接中，数据可以被立即发送而无需等待服务器的响应。

QUIC 提供了一个专门的流（流 ID 为1）用于执行握手，但握手协议的详细内容超出了本文档的范围。要查看当前握手协议的完整描述，请参考 QUIC Crypto Handshake 文档。QUIC 当前的握手协议将在未来被 TLS 1.3 替代。

灵活的拥塞控制(Flexible Congestion Control)

QUIC has pluggable congestion control and richer signaling than TCP, which enables QUIC to provide richer information to congestion control algorithms than TCP. Currently, the default congestion control is a reimplementation of TCP Cubic; we are currently experimenting with alternative approaches.

One example of richer information is that each packet, both original and retransmitted, carries a new packet sequence number. This allows a QUIC sender to distinguish ACKs for retransmissions from ACKs for original transmissions, thus avoiding TCP's retransmission ambiguity problem. QUIC ACKs also explicitly carry the delay between the receipt of a packet and its

acknowledgment being sent, and together with the monotonically-increasing packet numbers, this allows for precise roundtrip-time (RTT) calculation.

Finally, QUIC's ACK frames support up to 256 ack blocks, so QUIC is more resilient to reordering than TCP (with SACK), as well as able to keep more bytes on the wire when there is reordering or loss. Both client and server have a more accurate picture of which packets the peer has received.

QUIC 具有可插入的拥塞控制，且有着比 TCP 更丰富的信令，这使得 QUIC 相对于 TCP 可以为拥塞控制算法提供更丰富的信息。当前，默认的拥塞控制是
TCP Cubic 的重实现；我们目前在实验替代的方法。

更丰富的信息的一个例子是，每个包，包括原始的和重传的，都携带一个新的包序列号。这使得 QUIC 发送者可以将重传包的 ACKs 与原始传输包的 ACKs 区分开来，这样可以避免 TCP 的重传模糊问题。QUIC ACKs 也显式地携带数据包的接收与其确认被发送之间的延迟，与单调递增的包序列号一起，这样可以精确地计算往返时间（RTT）。

最后，QUIC 的 ACK 帧最多支持 256 个 ack 块，因此在重排序时，QUIC 相对于 TCP（使用SACK）更有弹性，这也使得在重排序或丢失出现时，QUIC 可以在线上保留更多在途字节。客户端和服务器都可以更精确地了解到哪些包对端已经接收。

流和连接的流量控制(Stream and Connection Flow Control)

QUIC implements stream- and connection-level flow control, closely following HTTP/2's flow control. QUIC's stream-level flow control works as follows. A QUIC receiver advertises the absolute byte offset within each stream upto which the receiver is willing to receive data. As data is sent, received, and delivered on a particular stream, the receiver sends WINDOW_UPDATE frames that increase the advertised offset limit for that stream, allowing the peer to send more data on that stream.

In addition to per-stream flow control, QUIC implements connection-level flow control to limit the aggregate buffer that a QUIC receiver is willing to allocate to a connection. Connection flow control works in the same way as stream flow control, but the bytes delivered and highest received offset are all aggregates across all streams.

Similar to TCP's receive-window autotuning, QUIC implements autotuning of flow control credits for both stream and connection flow controllers. QUIC's autotuning increases the size of the credits sent per WINDOW_UPDATE frame if it appears to be limiting the sender's rate, and throttles the sender when the receiving application is slow.

QUIC 实现了流级和连接级的流量控制，紧跟 HTTP/2 的流量控制。QUIC 的流级流控工作如下。QUIC 接收者通告每个流中接收者最多想要接收的数据的绝对字节偏移。随着数据在特定流中的发送，接收和传送，接收者发送WINDOW_UPDATE 帧，帧增加该流的通告偏移量限制，允许对端在该流上发送更多的数据。

除了每个流的流控制外，QUIC 还实现连接级的流控制，以限制 QUIC 接收者愿意为连接分配的总缓冲区。连接的流控制工作方式与流的流控制一样，但传送的字节和最大的接收偏移是所有流的总和。

与 TCP 的接收窗口自动调整类似，QUIC 实现流和连接流控制器的流控制信用的自动调整。如果 QUIC 的自动调整似乎限制了发送方的速率，并且在接收应用程序缓慢的时候抑制发送方，则 QUIC 的自动调整会增加每个 WINDOW_UPDATE 帧发送的信用额。

多路复用(Multiplexing)

HTTP/2 on TCP suffers from head-of-line blocking in TCP. Since HTTP/2 multiplexes many streams atop TCP's single-bytestream abstraction, a loss of a TCP segment results in blocking of all subsequent segments until a retransmission arrives, irrespective of the HTTP/2 stream that is encapsulated in subsequent segments.

Because QUIC is designed from the ground up for multiplexed operation, lost packets carrying data for an individual stream generally only impact that specific stream. Each stream frame can be immediately dispatched to that stream on arrival, so streams without loss can continue to be reassembled and make forward progress in the application.

Caveat: QUIC currently compresses HTTP headers via HTTP/2 HPACK header compression on a dedicated header stream(3), which imposes head-of-line blocking for header frames only.

基于 TCP 的 HTTP/2 深受 TCP 的队首阻塞问题困扰。由于 HTTP/2 在 TCP 的单个字节流抽象之上多路复用许多流，一个 TCP 片段的丢失将导致所有后续片段的阻塞直到重传到达，而封装在后续片段中的 HTTP/2 流可能和丢失的片段毫无关系。

由于 QUIC 是为多路复用操作从头设计的，携带个别流的的数据的包丢失时，通常只影响该流。每个流的帧可以在到达时立即发送给该流，因此，没有丢失数据的流可以继续重新汇集，并在应用程序中继续进行。

附加说明：当前 QUIC 在一个专门的首部流 (3) 中，通过 HTTP/2 HPACK 首部压缩压缩 HTTP 首部，则只有首部帧会出现队首阻塞问题。

认证和加密的首部和载荷(Authenticated and Encrypted Header and Payload)

TCP headers appear in plaintext on the wire and not authenticated, causing a plethora of injection and header manipulation issues for TCP, such as receive-window manipulation and sequence-number overwriting. While some of these are active attacks, others are mechanisms used by middleboxes in the network sometimes in an attempt to transparently improve TCP performance. However, even "performance-enhancing" middleboxes still effectively limit the evolvability of the transport protocol, as has been observed in the design of MPTCP and in its subsequent deployability issues.

QUIC packets are always authenticated and typically the payload is fully encrypted. The parts of the packet header which are not encrypted are still authenticated by the receiver, so as to thwart any packet injection or manipulation by third parties. QUIC protects connections from witting or unwitting middlebox manipulation of end-to-end communication.

Caveat: PUBLIC_RESET packets that reset a connection are currently not authenticated.

TCP 首部在网络中以明文出现，它没有经过认证，这导致了大量的 TCP 注入和首部管理问题，比如接收窗口管理和序列号覆写。尽管这些问题中的一些是主动攻击，有时其它则是一些网络中的中间盒子用来尝试透明地提升 TCP 性能的机制。然而，甚至 "性能增强" 中间设备依然有效地限制着传输协议的发展，这已经在 MPTCP 的设计及其后续的部署问题中观察到。

QUIC 数据包总是经过认证的，而且典型情况下载荷是全加密的。数据包头部不加密的部分依然会被接收者认证，以阻止任何第三方的数据包注入或操纵。QUIC 保护连接的端到端通信免遭智能或不知情的中间设备操纵。

警告：复位连接的 PUBLIC _RESET 包当前未经认证。

连接迁移(Connection Migration)

TCP connections are identified by a 4-tuple of source address, source port, destination address and destination port. A well-known problem with TCP is that connections do not survive IP address changes (for example, by switching from WiFi to cellular) or port number changes (when a client's NAT binding expires causing a change in the port number seen at the server). While MPTCP addresses the connection migration problem for TCP, it is still plagued by lack of middlebox support and lack of OS deployment.

QUIC connections are identified by a 64-bit Connection ID, randomly generated by the client. QUIC can survive IP address changes and NAT re-bindings since the Connection ID remains the same across these migrations. QUIC also provides automatic cryptographic verification of a migrating client, since a migrating client continues to use the same session key for encrypting and decrypting packets.

In cases when the connection is unambiguously identified by the 4-tuple, such as when a server sends packets to a client using an ephemeral port, there is an option to not send the connection ID to save bytes on the wire.

TCP 连接由源地址，源端口，目标地址和目标端口的4元组标识。TCP 一个广为人知的问题是，IP 地址改变（比如，由 WiFi 网络切换到移动网络）或端口号改变（当客户端的NAT绑定超时导致服务器看到的端口号改变）时连接会断掉。尽管 MPTCP 解决了 TCP 的连接迁移问题，但它依然为缺少中间设备和OS部署支持所困扰。

QUIC连接由一个 64-bit 连接 ID 标识，它由客户端随机地产生。在IP地址改变和 NAT 重绑定时，QUIC 连接可以继续存活，因为连接 ID 在这些迁移过程中保持不变。由于迁移客户端继续使用相同的会话密钥来加密和解密数据包，QUIC还提供了迁移客户端的自动加密验证。

当连接明确地用4元组标识时，比如服务器使用短暂的端口给客户端发送数据包时，有一个选项可用来不发送连接 ID 以节省线上传输的字节。

包类型和格式(Packet Types and Formats)

QUIC has Special Packets and Regular Packets. There are two types of Special Packets: Version Negotiation Packets and Public Reset Packets, and regular packets containing frames.

All QUIC packets should be sized to fit within the path's MTU to avoid IP fragmentation. Path MTU discovery is a work in progress, and the current QUIC implementation uses a 1350-byte maximum QUIC packet size for IPv6, 1370 for IPv4. Both sizes are without IP and UDP overhead.

QUIC 具有特殊包和普通包。有两种类型特殊包：版本协商包 (Version Negotiation Packets) 和 公共复位包 (Public Reset Packets)，普通包包含帧。

所有 QUIC 包的大小应该适配在路径的 MTU 以避免IP分片。路径 MTU 发现是正在进行中的工作，而当前 QUIC 实现为 IPv6 使用 1350 字节的最大QUIC包大小，IPv4 使用1370字节。两个大小都没有 IP 和 UDP 过载。

QUIC公共包头(QUIC Public Packet Header)

All QUIC packets on the wire begin with a public header sized between 1 and 51 bytes. The wire format for the public header is as follows:

传输的所有 QUIC 包以大小介于1至51字节的公共包头开始。公共包头的格式如下：

```
--- src
     0        1        2        3        4           8
+--------+--------+--------+--------+--------+---    ---+
| Public |    Connection ID (64)    ...                | ->
|Flags(8)|      (optional)                             |
+--------+--------+--------+--------+--------+---    ---+

     9       10       11       12
+--------+--------+--------+--------+
|      QUIC Version (32)            | ->
|         (optional)                |
+--------+--------+--------+--------+


    13       14       15       16      17       18       19       20
+--------+--------+--------+--------+--------+--------+--------+--------+
|                       Diversification Nonce                           | ->
|                             (optional)                                |
+--------+--------+--------+--------+--------+--------+--------+--------+

    21       22       23       24      25       26       27       28
+--------+--------+--------+--------+--------+--------+--------+--------+
|                   Diversification Nonce Continued                     | ->
|                             (optional)                                |
+--------+--------+--------+--------+--------+--------+--------+--------+

    29       30       31       32      33       34       35       36
+--------+--------+--------+--------+--------+--------+--------+--------+
|                   Diversification Nonce Continued                     | ->
|                             (optional)                                |
+--------+--------+--------+--------+--------+--------+--------+--------+

    37       38       39       40      41       42       43       44
+--------+--------+--------+--------+--------+--------+--------+--------+
|                   Diversification Nonce Continued                     | ->
|                             (optional)                                |
+--------+--------+--------+--------+--------+--------+--------+--------+


    45      46       47       48       49       50
+--------+--------+--------+--------+--------+--------+
|           Packet Number (8, 16, 32, or 48)          |
|                  (variable length)                  |
+--------+--------+--------+--------+--------+--------+
```

```
45
46  ---
```

The payload may include various type-dependent header bytes as described below.
The fields in the public header are the following:

- **Public Flags**:
    - 0x01 = PUBLIC_FLAG_VERSION. Interpretation of this flag depends on whether the packet is sent by the server or the client. When sent by the client, setting it indicates that the header contains a QUIC Version (see below). This bit must be set by a client in all packets until confirmation from the server arrives agreeing to the proposed version is received by the client. A server indicates agreement on a version by sending packets without setting this bit. When this bit is set by the server, the packet is a Version Negotiation Packet. Version Negotiation is described in more detail later.
    - 0x02 = PUBLIC_FLAG_RESET. Set to indicate that the packet is a Public Reset packet.
    - 0x04 = Indicates the presence of a 32 byte diversification nonce in the header.
    - 0x08 = Indicates the full 8 byte Connection ID is present in the packet. This must be set in all packets until negotiated to a different value for a given direction (e.g., client may request fewer bytes of the Connection ID be presented).
    - Two bits at 0x30 indicate the number of low-order-bytes of the packet number that are present in each packet. The bits are only used for Frame Packets. For Public Reset and Version Negotiation Packets (sent by the server) which don't have a packet number, these bits are not used and must be set to 0. Within this 2 bit mask:
        - 0x30 indicates that 6 bytes of the packet number is present
        - 0x20 indicates that 4 bytes of the packet number is present
        - 0x10 indicates that 2 bytes of the packet number is present
        - 0x00 indicates that 1 byte of the packet number is present
        - 0x40 is reserved for multipath use.
        - 0x80 is currently unused, and must be set to 0.
- **Connection ID**: This is an unsigned 64 bit statistically random number selected by the client that is the identifier of the connection. Because QUIC connections are designed to remain established even if the client roams, the IP 4-tuple (source IP, source port, destination IP, destination port) may be insufficient to identify the connection. For each transmission direction, when the 4-tuple is sufficient to identify the connection, the connection ID may be omitted.

- **QUIC Version**: A 32 bit opaque tag that represents the version of the QUIC protocol. Only present if the public flags contain FLAG_VERSION (i.e public_flags & FLAG_VERSION !=0). A client may set this flag, and include EXACTLY one proposed version, as well as including arbitrary data (conforming to that version). A server may set this flag when the client-proposed version was unsupported, and may then provide a list (0 or more) of acceptable versions, but MUST not include any data after the version(s). Examples of version values in recent experimental versions include "Q025" which corresponds to byte 9 containing 'Q', byte 10 containing '0', etc. [See list of changes in various versions listed at the end of this document.]

- **Packet Number**: The lower 8, 16, 32, or 48 bits of the packet number, based on which FLAG_?BYTE_SEQUENCE_NUMBER flag is set in the public flags. Each Regular Packet (as opposed to the Special public reset and version negotiation packets) is

assigned a packet number by the sender. The first packet sent by an endpoint shall have a packet number of 1, and each subsequent packet shall have a packet number one larger than that of the previous packet.

The lower 64 bits of the packet number is used as part of a cryptographic nonce; therefore, a QUIC endpoint must not send a packet with a packet number that cannot be represented in 64 bits. If a QUIC endpoint transmits a packet with a packet number of (2^64-1), that packet must include a CONNECTION_CLOSE frame with an error code of QUIC_SEQUENCE_NUMBER_LIMIT_REACHED, and the endpoint must not transmit any additional packets.

At most the lower 48 bits of a packet number are transmitted. To enable unambiguous reconstruction of the packet number by the receiver, a QUIC endpoint must not transmit a packet whose packet number is larger by (2^(bitlength-2)) than the largest packet number for which an acknowledgement is known to have been transmitted by the receiver. Therefore, there must never be more than (2^46) packets in flight.

Any truncated packet number shall be inferred to have the value closest to the one more than the largest known packet number of the endpoint which transmitted the packet that originally contained the truncated packet number. The transmitted portion of the packet number matches the lowest bits of the inferred value.

载荷可以包含多个如下所述类型相关的头部字节。

公共头部中的字段如下：

- **公共标记（Public Flags）**：
    - 0x01 = PUBLIC_FLAG_VERSION。这个标记的含义与包是由服务器还是客户端发送的有关。当由客户端发送时，设置它表示头部包含 QUIC 版本 (参考下面的说明)。客户端必须在所有的包中设置这个位，直到客户端收到来自服务器的确认，同意所提议的版本。服务器通过发送不设置该位的包来表示同意版本。当这个位由服务器设置时，包是版本协商包。版本协商在后面更详细地描述。
    - 0x02 = PUBLIC_FLAG_RESET。设置来表示包是公共复位包。
    - 0x04 = 表明在头部中存在 32字节的多样化随机数。
    - 0x08 = 表明包中存在完整的8字节连接ID。必须为所有包设置该位，直到为给定方向协商出不同的值 (比如，客户端可以请求包含更少字节的连接ID)。
    - 0x30 处的两位表示每个包中存在的数据包编号的低位字节数。这些位只用于帧包。没有包号的公共复位和版本协商包 (由服务器发送)，不使用这些位，且必须被设置为0。这2位的掩码：
        - 0x30 表示包号占用6个字节。
        - 0x20 表示包号占用4个字节。
        - 0x10 表示包号占用2个字节。
        - 0x00 表示包号占用1个字节。
        - 0x40 为多路径使用保留。
        - 0x80 当前未使用，且必须被设置为0。
- **连接ID**：这是客户端选择的无符号64位统计随机数，该数字是连接的标识符。由于 QUIC 的连接被设计为，即使客户端漫游，连接依然保持建立状态，因而 IP 4元组（源IP，源端口，目标IP，目标端口）可能不足以标识连接。对每个传输方向，当4元组足以标识连接时，连接ID可以省略。

- **QUIC版本**：表示 QUIC 协议版本的32位不透明标记。只有在公共标记包含 FLAG_VERSION（比如 public_flags & FLAG_VERSION !=0）时才存在。客户端可以设置这个标记，并 准确 包含一个提议版本，同时包含任意的数据（与该版本一致）。当客户端提议的版本不支持时，服务器可以设置这个标记，并可以提供一个可接受版本的列表（0或多个），但 一定不能(MUST not) 在版本信息之后包含任何数据。最近的实验版本的版本值示例包括 “Q025”，它对应于 byte 9 包含 ‘Q”，byte 10 包含 ‘0”，等等。[参考本文末尾的不同版本变化列表。]
- **包号**：包号的低 8，16，32，或 48 位，基于公共标记的 FLAG_?BYTE_SEQUENCE_NUMBER 标记被设置为什么。每个普通包（与特别的公共复位和版本协商包相反）由发送者分配包号。由某一端发送的首包包号应该为1，后续每个包的包号应该比前一个的大1。

  包号的低64位被用作加密随机数的一部分；然而，QUIC 端点一定不能发送其包号无法以 64 位表示的包。如果 QUIC 端点传输了包号为 (2^64-1) 的包，则该包必须包含错误码为 QUIC_SEQUENCE_NUMBER_LIMIT_REACHED 的 CONNECTION_CLOSE 帧，且对端一定不能再传输任何其它包。

  最多传输包号的低48位。要使接收者可以明确的重建包号，QUIC端点一定不能传输一个确认包已知已经由接收者发送的最大包号大 (2^(bitlength-2)) 的包。然而，在途包的数目不能超过 (2^46)。

  任何截断的包号应该被推断为具有 最接近但大于 传输最初包含了截断包号的包的对端 的最大已知包号的值。包号的发送部分匹配推断值的最低位。

A Public Flags processing flowchart follows:

公共标记处理流程图如下：

```
--- src
Check the public flags in public header
                 |
                 |
                 V
           +--------------+
           | Public Reset |    YES
           | flag set?    |----------------> Public Reset Packet
           +--------------+
                 |
                 | NO
                 V
           +-------------+          +--------------+
           | Version     |   YES    | Packet sent |  YES
           | flag set?   |--------->| by server?  |--------> Version Negotiation
           +-------------+          +--------------+               Packet
                 |                         |
                 | NO                      | NO
                 V                         V
           Regular Packet          Regular Packet with
                                  QUIC Version present in header
---
```

特殊包(Special Packets)

版本协商包(Version Negotiation Packet)

A version negotiation packet is only sent by the server. Version Negotiation packets begin with an 8-bit public flags and 64-bit Connection ID. The public flags must set PUBLIC_FLAG_VERSION and indicate the 64-bit Connection ID. The rest of the Version Negotiation packet is a list of 4-byte versions which the server supports:

只有服务器会发送版本协商包。版本协商包以8位的公共标记和64位的连接ID开始。公共标记必须设置PUBLIC_FLAG_VERSION，并指明64位的连接ID。版本协商包的其余部分是服务器支持的版本的4字节列表：

```
--- src
     0        1        2        3        4        5        6        7       8
+--------+--------+--------+--------+--------+--------+--------+--------+--------
| Public |    Connection ID (64)
|Flags(8)|
+--------+--------+--------+--------+--------+--------+--------+--------+--------

     9       10       11       12       13      14       15       16       17
+--------+--------+--------+--------+--------+--------+--------+--------+---...--
|      1st QUIC version supported   |     2nd QUIC version supported    |   ...
|      by server (32)               |     by server (32)                |
+--------+--------+--------+--------+--------+--------+--------+--------+---...--

---
```

公共复位包(Public Reset Packet)

A Public Reset packet begins with an 8-bit public flags and 64-bit Connection ID. The public flags must set PUBLIC_FLAG_RESET and indicate the 64-bit Connection ID. The rest of the Public Reset packet is encoded as if it were a crypto handshake message of the tag PRST (see [QUIC-CRYPTO]):

公共复位包以8位的公共标记和64位的连接ID开始。公共标记必须设置 PUBLIC_FLAG_RESET，并表明64位的连接ID。公共复位包的其余部分像标记 PRST 的加密握手消息那样编码（参考[QUIC-CRYPTO]）：

```
--- src
     0        1        2        3        4         8
+--------+--------+--------+--------+--------+--   --+
| Public |    Connection ID (64)               ...  | ->
|Flags(8)|                                          |
+--------+--------+--------+--------+--------+--   --+

     9       10       11       12       13      14
+--------+--------+--------+--------+--------+--------+---
|      Quic Tag (32)                |  Tag value map      ... ->
|         (PRST)                    |  (variable length)
```

```
+--------+--------+--------+--------+--------+--------+---
---
```

Tag value map: The tag value map contains the following tag-values:

- RNON (public reset nonce proof) - a 64-bit unsigned integer. Mandatory.
- RSEQ (rejected packet number) - a 64-bit packet number. Mandatory.
- CADR (client address) - the observed client IP address and port number. This is currently for debugging purposes only and hence is optional.

(TODO: Public Reset packet should include authenticated (destination) server IP/port.)

标记值映射：标记值映射包含如下的标记值：

- RNON (public reset nonce proof) - 一个64位的无符号整数。必须。
- RSEQ (rejected packet number) - 一个64位的包号。必须。
- CADR (client address) - 观察到的客户端IP地址和端口号。它当前只被用于调试，因而是可选的。
  (TODO：公共复位包应该包含认证的（目标）服务器 IP/端口。)

普通包(Regular Packets)

Regular Packets are authenticated and encrypted. The Public Header is authenticated but not encrypted, and the rest of the packet starting with the first frame is encrypted. Immediately following the Public Header, Regular Packets contain AEAD (authenticated encryption and associated data) data. This data must be decrypted in order for the contents to be interpreted. After decryption, the plaintext consists of a sequence of frames.

(TODO: Document the inputs to encryption and decryption and describe trial decryption.)

普通包已经过认证和加密。公共头部已认证但未加密，从第一帧开始的包的其余部分已加密。紧随公共头部之后，普通包包含AEAD（authenticated encryption and associated data）数据。要解释内容，这些数据必须先解密。解密之后，明文由一系列帧组成。

(TODO: 文档化加密和解密的输入，并描述试用解密)

帧包(Frame Packet)

Frame Packets have a payload that is a series of type-prefixed frames. The format of frame types is defined later in this document, but the general format of a Frame Packet is as follows:

帧包具有一个载荷，它是一系列的类型前缀帧。帧类型的格式将在本文档的后面定义，但帧包的通用格式如下：

```
--- src
+--------+---...---+--------+---...---+
| Type   | Payload | Type   | Payload |
```

```
4   +--------+---...---+--------+---...---+
5   ---
```

QUIC 连接的生命周期(Life of a QUIC Connection)

连接建立(Connection Establishment)

A QUIC client is the endpoint that initiates a connection. QUIC's connection establishment intertwines version negotiation with the crypto and transport handshakes to reduce connection establishment latency. We first describe version negotiation below.

QUIC客户端初始化一个连接。QUIC的连接建立将版本协商与加密和传输握手交织在一起以减少连接建立延迟。我们将在下面首先描述版本协商。

Each of the initial packets sent from the client to the server must set the version flag, and must specify the version of the protocol being used. Every packet sent by the client must have the version flag on, until it receives a packet from the server with the version flag off. After the server receives the first packet from the client with the version flag off, it must ignore any (possibly delayed) packets with the version flag on.

最初由客户端发向服务器的每个包必须设置版本标记，而且必须指定使用的协议版本。客户端发送的每个包必须开启版本标记，直到它从服务器收到了版本标记关闭的包。在服务器从客户端收到了第一个版本标记关闭的包之后，它必须忽略任何版本标记打开的包（可能由于延迟）。

When the server receives a packet with a Connection ID for a new connection, it will compare the client's version to the versions it supports. If the client's version is acceptable to the server, the server will use this protocol version for the lifetime of the connection. In this case, all packets sent by the server will have the version flag off.

当服务器收到一个含有新连接的连接ID的包，它将对比客户端的版本和它支持的版本。如果服务器可以接受客户端的版本，服务器将为连接的整个生命周期使用这个协议版本。在这种情况下，服务器发送的所有包的版本标记都是关闭的。

If the client's version is not acceptable to the server, a 1-RTT delay will be incurred. The server will send a Version Negotiation Packet to the client. This packet will have the version flag set and will include the server's set of supported versions.

如果客户端的版本不被服务器接受，则将导致1-RTT的延迟。服务器将发送一个版本协商包给客户端。这个包将设置版本标记，并将包含服务器支持的版本的集合。

When the client receives a Version Negotiation Packet from the server, it will select an acceptable protocol version and resend all packets using this version. These packet must continue to have the version flag set and must include the new negotiated protocol version. Eventually, the client receives the first Regular Packet (i.e. not a Version Negotiation Packet) from the server indicating the end of version negotiation, and the client now sends all subsequent packets with the version flag off.

当客户端从服务器收到一个版本协商包，它将选择一个可接受的协议版本并使用这个版本重发所有包。这些包必须持续设置版本标记，而且必须包含新协商的协议版本。最后，客户端从服务器收到第一个普通包（比如，一个非版本协商包）表明版本协商的结束，此后客户端发送的所有后续包版本标记关闭。

In order to avoid downgrade attacks, the version of the protocol that the client specified in the first packet and the set of versions supported by the server must be included in the crypto handshake data. The client needs to verify that the server's version list from the handshake matches the list of versions in the Version Negotiation Packet. The server needs to verify that the client's version from the handshake represents a version of the protocol that it does not actually support.

为了避免降级攻击，客户端在第一个包中指定的协议版本，以及服务器支持的版本集合必须被包含在加密的握手数据中。客户端需要验证握手中的服务器版本列表与版本协商包中的版本列表匹配。服务器需要验证握手中的客户端版本表示一个它实际上不支持的协议版本。

The rest of the connection establishment is described in the handshake document [QUIC-CRYPTO]. The crypto handshake is performed over the dedicated crypto stream (Stream ID 1).

连接建立的其余部分在握手文档中描述 [QUIC-CRYPTO]。加密握手在专门的加密流（流 ID 1）中执行。

During connection establishment, the handshake must negotiate various transport parameters. The currently defined transport parameters are described later in the document.

在连接握手期间，握手必须协商多种传输参数。当前已定义的传输参数在本文档的后面描述。

数据传输(Data Transfer)

QUIC implements connection reliability, congestion control, and flow control. QUIC flow control closely follows HTTP/2's flow control. QUIC reliability and congestion control are described in an accompanying document. A QUIC connection uses a single packet sequence number space for shared congestion control and loss recovery across the connection.

All data transferred in a QUIC connection, including the crypto handshake, is sent as data inside streams, but the ACKs acknowledge QUIC Packets.

This section conceptually describes the use of streams for data transfer within a QUIC connection. The various frames that are mentioned in this section are described in the section on Frame Types and Formats.

QUIC实现了连接可靠性，拥塞控制，和流量控制。QUIC流量控制与HTTP/2的流量控制很接近。QUIC可靠性和拥塞控制在一份附带文档中描述。QUIC连接为跨连接的共享拥塞控制和丢失恢复，而使用一个单独的包序列号空间。

QUIC连接中传输的所有数据，包括加密握手，被作为流内的数据发送，但ACKs确认QUIC包。

这个部分概念性地描述一个QUIC连接内数据传输的流的使用。本节提到的各种各样的帧在 帧类型和格式 一节中描述。

QUIC流的生命(Life of a QUIC Stream)

Streams are independent sequences of bi-directional data cut into stream frames. Streams can be created either by the client or the server, can concurrently send data interleaved with other streams, and can be cancelled. QUIC's stream lifetime is modeled

closely after HTTP/2's [RFC7540].
(HTTP/2's usage of QUIC streams is described in more detail later in the document.)

流是独立的双向数据序列，且被切割为流帧。流可以由客户端创建，也可以由服务器创建，可以与其它流并行交错地发送数据，且可以取消。QUIC流的生命周期模型与HTTP/2 [RFC 7540] 的很接近。（QUIC流的HTTP/2使用在本文档的后面部分有更详细的描述。）

Stream creation is done implicitly, by sending a STREAM frame for a given stream. To avoid stream ID collision, the Stream-ID must be even if the server initiates the stream, and odd if the client initiates the stream. 0 is not a valid Stream-ID. Stream 1 is reserved for the crypto handshake, which should be the first client-initiated stream. When using HTTP/2 over QUIC, Stream 3 is reserved for transmitting compressed headers for all other streams, ensuring reliable in-order delivery and processing of headers.

通过为一个给定的流发送一个STREAM帧，流创建显式地完成。为了避免流ID冲突，如果流是由服务器初始化的话，流 ID 必须是偶数，如果流由客户端初始化，则必须为奇数。0不是一个有效的流 ID。流 1 被保留用来加密握手，它应该是第一个客户端初始化的流。当基于QUIC使用HTTP/2时，流 3 被保留来为其它流传输压缩的首部，以确保首部的处理和传送可靠且有序。

Stream-IDs from each side of the connection must increase monotonically as new streams are created. E.g. Stream 2 may be created after stream 3, but stream 7 must not be created after stream 9. The peer may receive streams out of order. For example, if a server receives packet 10 including frames for stream 9 before it receives packet 9 including frames for stream 7, it should handle this gracefully.

随着新流的创建，连接的每一边的 流 ID 必须单调地递增。比如 流 2 可能在 流 3 之后创建，但 流 7 一定不能在 流 9 之后创建。对端可以接收乱序的流。比如，如果服务器收到了包 10，其中包含 流 9 的帧，在它收到包含 流 7 的帧的 包 9 之前，它应该优雅地处理这种情况。

If the endpoint receiving a STREAM frame does not want to accept the stream, it can immediately respond with a RST_STREAM frame (described below). Note, however, that the initiating endpoint may have already sent data on the stream as well; this data must be ignored.

如果端点收到一个STREAM帧，但它不想接受流，它可以立即以一个RST_STREAM帧（稍后描述）响应。注意，然而，初始化流的端点可能也已经在那个流上发送了数据；这些数据必须被忽略。

Once a stream is created, it can be used to send and receive data. This means that a series of stream frames can be sent by a QUIC endpoint on a stream until the stream is terminated in that direction.

一旦流创建好，它可被用于发送和接收数据。这意味着一系列的流帧可被QUIC端点在那个流上发送，直到流在那个方向上被终止。

Either QUIC endpoint can terminate a stream normally. There are three ways that streams can be terminated:

QUIC连接的任何一端都可以正常地终止一个流。有三种方式可以终止流：

1. **Normal termination**: Since streams are bidirectional, streams can be “half-closed” or “closed”. When one side of the stream sends a frame with the FIN bit set to true, the stream is considered to be “half-closed” in that direction. A FIN indicates that no further data will be sent from the sender of the FIN on this stream. When a QUIC endpoint has both sent and received a FIN, the endpoint considers the stream to be “closed”. While the FIN should be sent with the last user data for a stream, the FIN bit can be sent on an empty stream frame following the last data on the stream.

2. **Abrupt termination**: Either the client or server can send a RST_STREAM frame for a stream at any time. A RST_STREAM frame contains an error code to indicate the reason for failure (error codes are listed later in the document.) When a RST_STREAM frame is sent from the stream originator, it indicates a failure to complete the stream and that no further data will be sent on the stream. When a RST_STREAM frame is sent from the stream receiver, the sender, upon receipt, should stop sending any data on the stream. The stream receiver should be aware that there is a race between data already in transit from the sender and the time the RST_STREAM frame is received. In order to ensure that the connection-level flow control is correctly accounted, even if a RST_STREAM frame is received, a sender needs to ensure that either: the FIN and all bytes in the stream are received by the peer or a RST_STREAM frame is received by the peer. This also means that the sender of a RST_STREAM frame needs to continue responding to incoming STREAM_FRAMEs on this stream with the appropriate WINDOW_UPDATEs to ensure that the sender does not get flow control blocked attempting to deliver the FIN.

3. Streams are also terminated when the connection is terminated, as described in the next section.

1. **正常终止**：由于流是双向的，流可以是 “half-closed（半关闭）”或”closed（关闭）”状态。当流的一边发送一个FIN位被设为ture的帧，流被认为在那个方向上是”half-closed（半关闭）”的。FIN指明这个流上打开了FIN的发送者将不会在这个流上发送更多数据了。当QUIC的两个端点都发送并接收到了FIN，则端点认为流是”closed（关闭）”状态的。尽管FIN应该随着流的最后的用户数据一起发送，但FIN位可以被流的最后的数据帧后面的空流帧发送。
2. **异常终止**：客户端或服务器可以在任何时候为一个流发送RST_STREAM帧。RST_STREAM帧包含一个错误码用以指示失败原因（本文当的后面部分会列出错误码）。当流的发起者发送了一个RST_STREAM帧，它表示完成流失败了，而且不会有更多的数据在那个流上发送了。当RST_STREAM帧是由流的接收者发送的时，发送者，一旦接收，应该停止在那个流上发送任何数据。流接收者应该意识到发送者已经传输的数据和RST_STREAM帧接收的时间之间存在着竞态。为了确保连接级的流量控制可以被正确的实现，即使收到了一个RST_STREAM帧，发送者依然需要确保两者之一：对端收到流的FIN和所有字节或者对端收到一个RST_STREAM帧。这还意味着RST_STREAM帧的发送者需要持续以适当的WINDOW_UPDATE响应进入的那个流的STREAM_FREAME以确保发送者不让流量控制被阻塞而试图传送FIN。
3. 当连接终止时流也会被终止，如在下一节描述的那样。

连接终止(Connection Termination)

Connections should remain open until they become idle for a pre-negotiated period of time. When a server decides to terminate an idle connection, it should not notify the client to avoid waking up the radio on mobile devices. A QUIC connection, once established, can be terminated in one of two ways:

1. **Explicit Shutdown**: An endpoint sends a CONNECTION_CLOSE frame to the peer initiating a connection termination. An endpoint may send a GOAWAY frame to the peer prior to a CONNECTION_CLOSE to indicate that the connection will soon

be terminated. A GOAWAY frame when sent signals to the peer that any active streams will continue to be processed, but the sender of the GOAWAY will not initiate any additional streams and will not accept any new incoming streams. On termination of the active streams, a CONNECTION_CLOSE may be sent. If an endpoint sends a CONNECTION_CLOSE frame while unterminated streams are active (no FIN bit or RST_STREAM frames have been sent or received for one or more streams), then the peer must assume that the streams were incomplete and were abnormally terminated.

2. **Implicit Shutdown**: The default idle timeout for a QUIC connection is 30 seconds, and is a required parameter("ICSL") in connection negotiation. The maximum is 10 minutes. If there is no network activity for the duration of the idle timeout, the connection is closed. By default a CONNECTION_CLOSE frame will be sent. A silent close option can be enabled when it is expensive to send an explicit close, such as mobile networks that must wake up the radio.

An endpoint may also send a PUBLIC_RESET packet at any time during the connection to abruptly terminate an active connection. A PUBLIC_RESET is the QUIC equivalent of a TCP RST.

连接应该保持打开状态，直到他们在预协商周期的时间后变为空闲。当服务器决定终止一个空闲的连接时，它不应该通知客户端来避免唤醒移动设备的无线电模块。QUIC连接，一旦建立，可由两种方式中的一种终止：

1. **显式关闭**：一个端点发送一个CONNECTION_CLOSE帧给对端来初始化一个连接终止。一个端点可以在一个CONNECTION_CLOSE之前发送一个GOAWAY帧给对端来表明连接将在不久后终止。当发送GOAWAY帧时，通知对端任何活跃的流将继续被处理，但GOAWAY的发送者将不再初始化任何额外的流，且不接受任何新进入的流。在任何活跃的流的终止中，可以发送CONNECTION_CLOSE。如果一个端点在未终止的流活跃时发送了一个CONNECTION_CLOSE帧（一个或多个流还没有FIN位或RST_STREAM帧被发送或接收），则对端必须假设流是不完整的且被异常地终止。
2. **隐式关闭**：QUIC连接默认的空闲超时时间是30秒，且是连接协商中的一个必须参数("ICSL")。最大值是10分钟。如果在空闲超时期间没有网络活动，连接将关闭。默认情况下将发送一个CONNECTION_CLSOE帧。当发送一个显式的关闭比较昂贵时可以启用安静关闭选项，比如移动网络必须唤醒无线电模块。

一个端点还可以在连接期间的任何时间发送一个PUBLIC_RESET包来突然地终止活跃的连接。QUIC中的PUBLIC_RESET等价于TCP的RST。

帧类型和格式(Frame Types and Formats)

QUIC Frame Packets are populated by frames. which have a Frame Type byte, which itself has a type-dependent interpretation, followed by type-dependent frame header fields. All frames are contained within single QUIC Packets and no frame can span across a QUIC Packet boundary.

QUIC帧包由帧填充。它具有一个帧类型字节，它本身具有一个依赖类型的解释，后面是依赖类型的帧首部字段。所有的帧被包含在单独的QUIC包中，且没有帧可以跨越QUIC包边界。

帧类型(Frame Types)

There are two interpretations for the Frame Type byte, resulting in two frame types: Special Frame Types, and Regular Frame Types. Special Frame Types encode both a Frame Type and corresponding flags all in the Frame Type byte, while Regular Frame Types use the Frame Type byte simply.

Currently defined Special Frame Types are:

帧类型字节有两种解释，导致两种帧类型：特殊帧类型，和普通帧类型。特殊帧类型在帧类型字节中同时编码帧类型和对应的标记，而普通帧类型简单地使用帧类型字节。

当前定义的特殊帧类型如下：

```
--- src
   +------------------+----------------------------+
   | Type-field value |     Control Frame-type     |
   +------------------+----------------------------+
   |     1fdooossB    |  STREAM                    |
   |     01ntllmmB    |  ACK                       |
   |     001xxxxxB    |  CONGESTION_FEEDBACK       |
   +------------------+----------------------------+
---
```

Currently defined Regular Frame Types are:
当前定义的普通帧类型如下：

```
--- src
   +------------------+----------------------------+
   | Type-field value |     Control Frame-type     |
   +------------------+----------------------------+
   | 00000000B (0x00) |  PADDING                   |
   | 00000001B (0x01) |  RST_STREAM                |
   | 00000010B (0x02) |  CONNECTION_CLOSE          |
   | 00000011B (0x03) |  GOAWAY                    |
   | 00000100B (0x04) |  WINDOW_UPDATE             |
   | 00000101B (0x05) |  BLOCKED                   |
   | 00000110B (0x06) |  STOP_WAITING              |
   | 00000111B (0x07) |  PING                      |
   +------------------+----------------------------+
---
```

STREAM帧(STREAM Frame)

The STREAM frame is used to both implicitly create a stream and to send data on it, and is as follows:

STREAM帧同时被用于隐式地创建流和在流上发送数据，它的格式如下：

```
--- src
      0        1       …               SLEN
+--------+--------+--------+--------+--------+
|Type (8)| Stream ID (8, 16, 24, or 32 bits) |
|        |    (Variable length SLEN bytes)   |
+--------+--------+--------+--------+--------+
```

```
 7
 8     SLEN+1  SLEN+2     …                                        SLEN+OLEN
 9   +--------+--------+--------+--------+--------+--------+--------+--------+
10   |   Offset (0, 16, 24, 32, 40, 48, 56, or 64 bits) (variable length)    |
11   |                    (Variable length: OLEN  bytes)                     |
12   +--------+--------+--------+--------+--------+--------+--------+--------+
13
14     SLEN+OLEN+1   SLEN+OLEN+2
15   +-------------+-------------+
16   | Data length (0 or 16 bits)|
17   |  Optional(maybe 0 bytes)  |
18   +-------------+-------------+
19   ---
```

The fields in the STREAM frame header are as follows:

- **Frame Type**: The Frame Type byte is an 8-bit value containing various flags (1fdooossB):
  - The leftmost bit must be set to 1 indicating that this is a STREAM frame.
  - The 'f' bit is the FIN bit. When set to 1, this bit indicates the sender is done sending on this stream and wishes to "half-close" (described in more detail later.)
  - which is described in more detail later in this document.
  - The 'd' bit indicates whether a Data Length is present in the STREAM header. When set to 0, this field indicates that the STREAM frame extends to the end of the Packet.
  - The next three 'ooo' bits encode the length of the Offset header field as 0, 16, 24, 32, 40, 48, 56, or 64 bits long.
  - The next two 'ss' bits encode the length of the Stream ID header field as 8, 16, 24, or 32 bits long.
- **Stream ID**: A variable-sized unsigned ID unique to this stream.
- **Offset**: A variable-sized unsigned number specifying the byte offset in the stream for this block of data.
- **Data length**: An optional 16-bit unsigned number specifying the length of the data in this stream frame. The option to omit the length should only be used when the packet is a "full-sized" Packet, to avoid the risk of corruption via padding.

A stream frame must always have either non-zero data length or the FIN bit set.

STREAM帧首部中的字段如下：

- **帧类型**：帧类型字节是一个包含多种标记 (1fdooossB) 的8位值：
  - 最左边的位必须被设为 1 以指明这是一个STREAM帧。
  - 'f' 位是FIN位。当被设置为 1 时，这个位表明发送者已经完成在流上的发送并希望 "half-close（半关闭）"（稍后将详细描述）。本文档的后面将更详细地描述。
  - 'd' 位表明STREAM头部中是否包含数据长度。当设为0时，这个字段表明STREAM帧扩展至包的结尾。
  - 接下来的三个'ooo'位编码Offset头部字段的长度为0，16，24，32，40，48，56，或64位长。
  - 接下来的两个 'ss' 位编码流 ID头部字段的长度为 8，16，24，或32位长。
- **流 ID**：一个大小可变的流唯一的无符号ID。
- **偏移**：一个大小可变的无符号数字指定流中这块数据的字节偏移。
- **数据长度**：一个可选的16位无符号数字指定这个流帧中数据的长度。只有当包是 "全大小(full-sized)" 包时，才应该省略长度，来避免填充破坏的风险。

一个流帧必须总是要么具有非零的数据长度，要么设置了FIN位。

ACK帧(ACK Frame)

The ACK frame is sent to inform the peer which packets have been received, as well as which packets are still considered missing by the receiver (the contents of missing packets may need to be resent). The ack frame contains between 1 and 256 ack blocks. Ack blocks are ranges of acknowledged packets, similar to TCP's SACK blocks, but QUIC has no equivalent of TCP's cumulative ack point, because packets are retransmitted with new sequence numbers.

To limit the ACK blocks to the ones that haven't yet been received by the peer, the peer periodically sends STOP_WAITING frames that signal the receiver to stop acking packets below a specified sequence number, raising the "least unacked" packet number at the receiver. A sender of an ACK frame thus reports only those ACK blocks between the received least unacked and the reported largest observed packet numbers. It is recommended for the sender to send the most recent largest acked packet it has received in an ack as the stop waiting frame's least unacked value.

Unlike TCP SACKs, QUIC ACK blocks are irrevocable, so once a packet is acked, even if it does not appear in a future ack frame, it is assumed to be acked.

As a replacement for QUIC's deprecated entropy, the sender can intentionally skip packet numbers to introduce entropy into the connection. The sender must always close the connection if an unsent packet number is acked, so this mechanism automatically defeats any potential attackers. The ack format is efficient at expressing blocks of missing packets, so this has a low cost to the receiver and sender and efficiently provides up to 8 bits of entropy on demand, rather than incurring the constant overhead and achieving 8 bits of entropy. The 8 bits is the longest gap between ack ranges the ack format can efficiently express.

Section Offsets

0: Start of the ack frame.

T: Byte offset of the start of the timestamp section.

A: Byte offset of the start of the ack block section.

N: Length in bytes of the largest acked.

ACK帧被发送用以通知对端哪些包已经收到，还有哪些包依然被接收者认为丢失了（丢失包的内容可能需要被重发）。ACK帧包含1 到 256 个ack块。Ack块是确认的包的范围，与TCP的SACK块类似，但QUIC没有等价的TCP的累积ack点，由于包将以新的序列号重传。

要限制ACK块为还没有被对端接收的，对端周期性地发送STOP_WAITING帧，用来通知接收者停止确认小于特定序列号的包，再接收者产生"最小未确认"包数字。ACK帧的发送者这样只报告那些在接收到的最小未确认和报告的最大已发现包号之间的ACK块。建议发送者在ack中发送它已经接收到的最近最大确认包，作为stop waiting帧的最小未确认值。

不像TCP SACK，QUIC ACK块是不可变的，因此一旦一个包被确认了，即使它没有出现在未来的ack帧中，它也被假设已经确认。

作为QUIC的已废弃的熵的替代，发送者可以有意地跳过包号来为连接引入熵。如果一个未发送的包号被确认了，则发送者必须总是关闭连接，因此这种机制自动地防御了任何潜在的攻击。ack格式在表达丢失包的块上是比较高效的，因此对于接收者和发送者这还是成本比较低的，且可以根据需要有效地提供至多8 位的熵，而不是通过恒定的开销来实现8位的熵。8位是ack范围和ack格式之间可高效地表达的最大gap。

段偏移

0：Ack帧的起始位置。

T：时间戳段起始位置的字节偏移量。

A：Ack块段起始位置的字节偏移量。

N：最大已确认的字节长度。

```
--- src
     0                            1  => N                       N+1 => A(aka N + 3)
+---------+-----------------------------------------------------+--------+--------+
|   Type  |                    Largest Acked                    |  Largest Acked  |
|   (8)   |    (8, 16, 32, or 48 bits, determined by ll)        | Delta Time (16) |
|01nullmm |                                                     |                 |
+---------+-----------------------------------------------------+--------+--------+


     A             A + 1  ==>  A + N
+--------+-----------------------------------------+
| Number |             First Ack                   |
|Blocks-1|           Block Length                  |
| (opt)  |(8, 16, 32 or 48 bits, determined by mm)|
+--------+-----------------------------------------+

  A + N + 1               A + N + 2  ==>  T(aka A + 2N + 1)
+------------+-------------------------------------------------+
| Gap to next|              Ack Block Length                   |
| Block (8)  |   (8, 16, 32, or 48 bits, determined by mm)     |
| (Repeats)  |       (repeats Number Ranges times)             |
+------------+-------------------------------------------------+
     T        T+1             T+2                  (Repeated Num Timestamps)
+----------+--------+--------------------+ ...  --------+------------------+
|   Num    | Delta  |     Time Since     |     | Delta  |       Time       |
|Timestamps|Largest |    Largest Acked   |     |Largest |  Since Previous  |
|   (8)    | Acked  |      (32 bits)     |     | Acked  |Timestamp(16 bits)|
+----------+--------+--------------------+     +--------+------------------+
---
```

The fields in the ACK frame are as follows:

- **Frame Type**: The Frame Type byte is an 8-bit value containing various flags (01nullmmB).
  - The first two bits must be set to 01 indicating that this is an ACK frame.
  - The 'n' bit indicates whether the frame has more than 1 ack range.
  - The 'u' bit is unused.
  - The two 'll' bits encode the length of the Largest Observed field as 1, 2, 4, or 6 bytes long.
  - The two 'mm' bits encode the length of the Missing Packet Sequence Number Delta field as 1, 2, 4, or 6 bytes long.
- **Largest Acked**: A variable-sized unsigned value representing the largest packet number the peer has observed.
- **Largest Acked Delta Time**: A 16-bit unsigned float with 11 explicit bits of mantissa and 5 bits of explicit exponent, specifying the time elapsed in microseconds from when largest acked was received until this Ack frame was sent. The bit format is loosely modeled after IEEE 754. For example, 1 microsecond is represented as 0x1, which has an exponent of zero, presented in the 5 high order bits, and mantissa of 1, presented in the 11 low order bits. When the explicit exponent is greater than zero, an implicit high-order 12th bit of 1 is assumed in the mantissa. For example, a floating value of 0x800 has an explicit exponent of 1, as well as an explicit mantissa of 0, but then has an effective mantissa of 4096 (12th bit is assumed to be 1). Additionally, the actual exponent is one-less than the explicit exponent, and the value represents 4096 microseconds. Any values larger than the representable range are clamped to 0xFFFF.
- **Ack Block Section**:
  - **Num Blocks**: An optional 8-bit unsigned value specifying one less than the number of ack blocks. Only present if the 'n' flag bit is 1.
  - **Ack block length**: A variable-sized packet number delta. For the first missing packet range, the ack block starts at largest acked. For the first ack block, the length of the ack block is 1 + this value. For subsequent ack blocks, it is the length of the ack block. For non-first blocks, a value of 0 indicates more than 256 packets in a row were lost.
  - **Gap to next block**: An 8-bit unsigned value specifying the number of packets between ack blocks.
- Timestamp Section:
  - **Num Timestamp**: An 8-bit unsigned value specifying the number of timestamps that are included in this ack frame. There will be this many pairs of following in the timestamps.
  - **Delta Largest Observed**: An 8-bit unsigned value specifying the packet number delta from the first timestamp to the largest observed. Therefore, the packet number is the largest observed minus the delta largest observed.
  - **First Timestamp**: A 32-bit unsigned value specifying the time delta in microseconds, from the beginning of the connection of the arrival of the packet specified by Largest Observed minus Delta Largest Observed.
  - **Delta Largest Observed (Repeated)**: (Same as above.)
  - **Time Since Previous Timestamp (Repeated)**: A 16-bit unsigned value specifying delta from the previous timestamp. It is encoded in the same format as the Ack Delay Time.

ACK帧中的字段如下：

- **帧类型**：帧类型字节是一个8位的值，其中包含了多种标记（01nullmmB）。
  - 开始的两位必须被设置为 01，以表明这是一个ACK帧。
  - 'n' 位表明帧是否有多于 1 个ack帧。
  - 'u' 位未使用。
  - 两个'll'位编码最大已观察字段的长度为1，2，4，或者 6字节长。
  - 两个'mm'位编码丢失包序列号差值字段的长度为 1，2，4，或者 6字节长。
- **最大已确认(Largest Acked)**：一个大小可变的无符号值，表示对端已观察到的最大的包号。

- **最大已确认差值时间(Largest Acked Delta Time)**：一个16 位的无符号浮点数，其中11个显式的位为底数，5位的显式指数，描述了从最大已确认包收到到这个Ack帧发送之间经过的微秒数。位格式近似于以IEEE 754建模。比如，1 微秒用 0x1 表示，它的指数为0，在高5位中表示，底数为1，在低 11 位中表示。当显式的指数大于0的时候，则假设底数包含一个隐式的高阶12位的1。比如，浮点值 0x800 有一个显式的指数1，同时有一个显式的底数0，但之后有一个有效的底数 4096（假设12位为1）。此外，实际的指数比显式的指数小1，值表示4096微秒。任何大于可表示范围的值限定为 0xFFFF。
- **Ack 块段 (Ack Block Section)**：
  - **块个数 (Num Blocks)**：一个可选的8位无符号值描述了ack块的个数减一。只有在 'n' 标记位为 1 时才有。
  - **Ack块长度 (Ack block length)**：一个大小可变包号差值。对于第一个丢失包范围，ack块以最大已确认包开始。对于首个ack块，ack块的长度为 1 + 该值。对于后续的ack块，它是ack块的长度。对于非首个块，0值表示多于256个包丢失了。
  - **到下一块的间隙 (Gap to next block)**：一个8位的无符号值，描述了ack块之间包的个数。
- **时间戳段 (Timestamp Section)**：
  - **时间戳个数 (Num Timestamp)**：一个8位无符号值描述了包含在这个ack帧中的时间戳的个数。在后面的timestamps中将由许多的对。
  - **已观察最大差值 (Delta Largest Observed)**：一个8位无符号值描述了首个时间戳和最大已观察包之间包号的差值。然而，包号为最大已观察包号 减去 已观察最大差值 (delta largest observed)。
  - **首个时间戳 (First Timestamp)**：一个32位无符号值描述自由最大已观察包号描述的包的到达的连接的开始，减去已观察最大差值，所得到的时间差值的微秒数。
  - **已观察最大差值（重复）(Delta Largest Observed(Repeated))**：（同上。）
  - **自前一个时间戳的时间（重复） (Time Since Previous Timestamp (Repeated))**：一个16位的无符号值描述了与前一个时间戳的差值。它的编码格式与 Ack Delay Time相同。

STOP_WAITING 帧(STOP_WAITING Frame)

The STOP_WAITING frame is sent to inform the peer that it should not continue to wait for packets with packet numbers lower than a specified value. The packet number is encoded in 1, 2, 4 or 6 bytes, using the same coding length as is specified for the packet number for the enclosing packet's header (specified in the QUIC Frame Packet's Public Flags field.) The frame is as follows:

STOP_WAITING 帧用于通知对端，它不应该继续等待包号小于特定值的包。包号以1，2，4或6字节编码，using the same coding length as is specified for the packet number for the enclosing packet's header (specified in the QUIC Frame Packet's Public Flags field.) 这个帧如下:

```
--- src
     0        1        2        3         4       5       6
+--------+--------+--------+--------+--------+-------+-------+
|Type (8)|   Least unacked delta (8, 16, 32, or 48 bits)     |
|        |                       (variable length)           |
+--------+--------+--------+--------+--------+--------+------+
---
```

The fields in the STOP_WAITING frame are as follows:

- **Frame Type**: The Frame Type byte is an 8-bit value that must be set to 0x06 indicating that this is a STOP_WAITING frame.

- **Least Unacked Delta**: A variable length packet number delta with the same length as the packet header's packet number. Subtract it from the header's packet number to determine the least unacked. The resulting least unacked is the smallest packet number of any packet for which the sender is still awaiting an ack. If the receiver is missing any packets smaller than this value, the receiver should consider those packets to be irrecoverably lost.

STOP_WAITING帧中的字段如下：

- **帧类型**：帧类型是一个8位的值，它必须被设置为0x06以表明这是一个STOP_WAITING帧。

- **最小未确认差值**：一个可变长度的包号差值，与包首部的包号长度相同。将它从头部的包号减去以确定最小的未确认包。结果的最小未确认包是发送者依然在等待确认的包号最小的包。如果接收者丢失了任何比这个值小的包，接收者应该将那些包认做无可挽回的丢失。

WINDOW_UPDATE 帧(WINDOW_UPDATE Frame)

The WINDOW_UPDATE frame is used to inform the peer of an increase in an endpoint's flow control receive window. The stream ID can be 0, indicating this WINDOW_UPDATE applies to the connection level flow control window, or > 0 indicating that the specified stream should increase its flow control window. The frame is as follows:

An absolute byte offset is specified, and the receiver of a WINDOW_UPDATE frame may only send up to that number of bytes on the specified stream. Violating flow control by sending further bytes will result in the receiving endpoint closing the connection.

On receipt of multiple WINDOW_UPDATE frames for a specific stream ID, it is only necessary to keep track of the maximum byte offset.

Both stream and session windows start with a default value of 16 KB, but this is typically increased during the handshake. To do this, an endpoint should negotiate the SFCW (Stream Flow Control Window) and CFCW (Connection/Session Flow Control Window) parameters in the handshake. The value associated with each tag should be the number of bytes for initial stream window and initial connection window respectively.

The frame is as follows:

WINDOW_UPDATE 帧用于通知对端一个端点的流量控制接收窗口的增长。流ID可以是0，表示这个WINDOW_UPDATE应用于连接级的流量控制窗口，或者 > 0 表示指定的流应该增长它的流量控制窗口。帧如下：

指定一个完全的字节偏移量，WINDOW_UPDATE帧的接收者可以只在那个流上至多发送那个字节数。发送更多字节而违背流量控制将导致接收端关闭连接。

为特定流ID收到多个WINDOW_UPDATE帧时，只需要追踪最大的字节偏移即可。

流和会话窗口都以一个默认值16KB开始，但是这个值典型地在握手期间增长。为了做到这一点，端点应该在握手中协商 SFCW (Stream Flow Control Window) 和 CFCW (Connection/Session Flow Control Window) 参数。与每个标记关联的值应该分别是初始流窗口和初始连接窗口的字节数。

帧如下：

```
--- src
    0         1                 4        5                 12
+--------+--------+-- ... --+--------+--------+-- ... --+--------+
|Type(8) |    Stream ID (32 bits)    |  Byte offset (64 bits)    |
+--------+--------+-- ... --+--------+--------+-- ... --+--------+
---
```

The fields in the WINDOW_UPDATE frame are as follows:

- **Frame Type**: The Frame Type byte is an 8-bit value that must be set to 0x04 indicating that this is a WINDOW_UPDATE frame.
- **Stream ID**: ID of the stream whose flow control windows is being updated, or 0 to specify the connection-level flow control window.
- **Byte offset**: A 64-bit unsigned integer indicating the absolute byte offset of data which can be sent on the given stream. In the case of connection level flow control, the cumulative number of bytes which can be sent on all currently open streams.

WINDOW_UPDATE帧中的字段如下：

- **帧类型**：帧类型是一个8位值，它必须被设置为0x04以表示这是一个WINDOW_UPDATE帧。
- **流 ID**：要更新流控制窗口的流的ID，或者为0来描述连接级的流控制窗口。
- **字节偏移**：一个64位无符号整型值，表示在给定的流上可以发送的数据的完整字节偏移量。在连接级流量控制的情况下，是在当前所有打开的流上可以发送的字节的总和。

BLOCKED 帧(BLOCKED Frame)

The BLOCKED frame is used to indicate to the remote endpoint that this endpoint is ready to send data (and has data to send), but is currently flow control blocked. This is a purely informational frame, which is extremely useful for debugging purposes. A receiver of a BLOCKED frame should simply discard it (after possibly printing a helpful log message). The frame is as follows:

BLOCKED帧用于向远端指明本端点已经准备好发送数据了（且有数据要发送），但是当前被流量控制阻塞了。这是一个纯粹的信息帧，它对于调试极其有用。BLOCKED帧的接收者应该简单的丢弃它（可能在打印了一条有帮助的log消息之后）。帧如下：

```
--- src
     0         1         2         3         4
+--------+--------+--------+--------+--------+
|Type(8) |          Stream ID (32 bits)       |
+--------+--------+--------+--------+--------+
---
```

The fields in the BLOCKED frame are as follows:

- **Frame Type**: The Frame Type byte is an 8-bit value that must be set to 0x05 indicating that this is a BLOCKED frame.
- **Stream ID**: A 32-bit unsigned number indicating the stream which is flow control blocked. A non-zero Stream ID field specifies the stream that is flow control blocked. When zero, the Stream ID field indicates that the connection is flow control blocked at the connection level.

BLOCKED帧中的字段如下：

- **帧类型**：帧类型是一个8位值，它必须被设置为0x05以表示这是一个BLOCKED帧。
- **流 ID**：一个32位的无符号数，表示流量控制阻塞的流。非零 流 ID 字段描述了被流量控制阻塞的流。当这个值为0时，流ID字段在连接级指明连接被流量控制阻塞了。

CONGESTION_FEEDBACK 帧(CONGESTION_FEEDBACK Frame)

The CONGESTION_FEEDBACK frame is an experimental frame currently not used. It is intended to provide extra congestion feedback information outside the scope of the standard ack frame. A CONGESTION_FEEDBACK frame must have the first three bits of the Frame Type set to 001. The last 5 bits of the Frame Type field are reserved for future use.

CONGESTION_FEEDBACK帧是一个实验性的帧，当前未使用。这个帧的本意是在ACK帧之外提供额外的拥塞反馈信息。CONGESTION_FEEDBACK帧必须将表示帧类型的前三位设置为001，后5位预留。

PADDING 帧(PADDING Frame)

The PADDING frame pads a packet with 0x00 bytes. When this frame is encountered, the rest of the packet is expected to be padding bytes. The frame contains 0x00 bytes and extends to the end of the QUIC packet. A PADDING frame only has a Frame Type field, and must have the 8-bit Frame Type field set to 0x00.

PADDING帧使用0x00字节填充一个包。当遇到该帧时，包的剩余部分需要被填充字节。该帧包含0x00字节并扩展至QUIC包的末端。PADDING帧只有一个帧类型字段，且必须将8位的帧类型字段设为0x00。

RST_STREAM 帧(RST_STREAM Frame)

The RST_STREAM frame allows for abnormal termination of a stream. When sent by the creator of a stream, it indicates the creator wishes to cancel the stream. When sent by the receiver of a stream, it indicates an error or that the receiver did not want to accept the stream, so the stream should be closed. The frame is as follows:

RST_STREAM帧允许异常终止一条流。当这个帧是流的创建者发出的，表示创建者希望取消这条流。当接收端发送这个帧，表示有错误或者当前接收端不希望接收这个流，因此这个流应该被关闭。帧结构如下：

```
--- src
     0        1           4      5             12     8            16
+-------+--------+-- ... ----+--------+-- ... ------+-------+-- ... ------+
|Type(8)| StreamID (32 bits) | Byte offset (64 bits)| Error code (32 bits)|
+-------+--------+-- ... ----+--------+-- ... ------+-------+-- ... ------+
---
```

The fields in a RST_STREAM frame are as follows:

- **Frame type**: The Frame Type is an 8-bit value that must be set to 0x01 specifying that this is a RST_STREAM frame.
- **Stream ID**: The 32-bit Stream ID of the stream being terminated.
- **Byte offset**: A 64-bit unsigned integer indicating the absolute byte offset of the end of data for this stream.
- **Error code**: A 32-bit QuicErrorCode which indicates why the stream is being closed. QuicErrorCodes are listed later in this document.

RST_STREAM帧的字段如下:

- **帧类型**: 帧类型是一个8位的值，必须设置为0x01表示这是一个RST_STREAM帧。
- **流标识符**: 32位流标识符，表示将被终止的流。
- **字节偏移**: 64位无符号整型表示流数据的绝对字节偏移。
- **错误码**: 32位的QUIC错误码表示流被关闭的原因，错误码在文档后续列出。

PING 帧(PING frame)

The PING frame can be used by an endpoint to verify that a peer is still alive. The PING frame contains no payload. The receiver of a PING frame simply needs to ACK the packet containing this frame. The PING frame should be used to keep a connection alive when a stream is open. The default is to do this after 15 seconds of quiescence, which is much shorter than most NATs time out. A PING frame only has a Frame Type field, and must have the 8-bit Frame Type field set to 0x07.

PING帧用来验证对端是否仍然存活。PING帧不包含载荷。PING帧的接收方只需要应答（ACK）包含该帧的包。PING帧应该被用于当一条流被打开时，保持连接存活。默认是在15秒静默后发出PING帧，这比大多数NAT超时要短得多。PING帧只有帧类型字段，且必须将8位的帧类型字段设为0x07。

CONNECTION_CLOSE 帧(CONNECTION_CLOSE frame)

The CONNECTION_CLOSE frame allows for notification that the connection is being closed. If there are streams in flight, those streams are all implicitly closed when the connection is closed. (Ideally, a GOAWAY frame would be sent with enough time that all streams are torn down.) The frame is as follows:

CONNECTION_CLOSE帧用来通知连接将被关闭。如果流仍然有数据在发送，那么在连接关闭时，这些流将被隐式关闭。（理论上一个GOAWAY帧应该已经被发送了足够的时间使所有流都关闭。）帧结构如下：

```
--- src
     0        1             4        5        6       7
+--------+--------+-- ... -----+--------+--------+--------+----- ...
|Type(8) | Error code (32 bits)| Reason phrase   |  Reason phrase
|        |                     | length (16 bits)|(variable length)
+--------+--------+-- ... -----+--------+--------+--------+----- ...
---
```

The fields of a CONNECTION_CLOSE frame are as follows:

- **Frame Type**: An 8-bit value that must be set to 0x02 specifying that this is a CONNECTION_CLOSE frame.
- **Error Code**: A 32-bit field containing the QuicErrorCode which indicates the reason for closing this connection.
- **Reason Phrase Length**: A 16-bit unsigned number specifying the length of the reason phrase. This may be zero if the sender chooses to not give details beyond the QuicErrorCode.
- **Reason Phrase**: An optional human-readable explanation for why the connection was closed.

CONNECTION_CLOSE帧的字段如下:

- **帧类型**: 8位的值必须设置为0x02，表示这个帧是一个CONNECTION_CLOSE帧。
- **错误码**: 32位字段包含了QUIC错误码，表明连接关闭原因。
- **原因描述长度**: 16位无符号数，表示reason phrase的长度。如果发送方除了错误码之外，并不打算给出详细情况，那么该字段可能为0。
- **原因描述**: 可选的可读的连接关闭的原因。

GOAWAY 帧(GOAWAY Frame)

The GOAWAY frame allows for notification that the connection should stop being used, and will likely be aborted in the future. Any active streams will continue to be processed, but the sender of the GOAWAY will not initiate any additional streams, and will not accept any new streams. The frame is as follows:

GOAWAY帧用来通知该连接将停止使用，将来将要被终止。任何激活的流在收到GOAWAY帧后将继续被处理，但是GOAWAY帧的发送端将不再初始化任何额外的流，也不会接受任何新的流。帧结构如下：

```
--- src
     0        1             4      5       6       7      8
+--------+--------+-- ... -----+-------+-------+-------+------+
|Type(8) | Error code (32 bits)| Last Good Stream ID (32 bits)| ->
+--------+--------+-- ... -----+-------+-------+-------+------+

      9        10       11
+--------+--------+--------+----- ...
| Reason phrase   |  Reason phrase
| length (16 bits)|(variable length)
+--------+--------+--------+----- ...
---
```

The fields of a GOAWAY frame are as follows:

- **Frame type**: An 8-bit value that must be set to 0x03 specifying that this is a GOAWAY frame.
- **Error Code**: A 32-bit field containing the QuicErrorCode which indicates the reason for closing this connection.
- **Last Good Stream ID**: The last Stream ID which was accepted by the sender of the GOAWAY message. If no streams were replied to, this value must be set to 0.

- **Reason Phrase Length**: A 16-bit unsigned number specifying the length of the reason phrase. This may be zero if the sender chooses to not give details beyond the error code.
- **Reason Phrase**: An optional human-readable explanation for why the connection was closed.

GOAWAY帧字段如下:

- **分片类型**: 8位的值必须设置为0x03表示这个帧是一个GOAWAY帧。
- **错误码**: 32位字段包含QUIC错误码，表示关闭连接的原因。
- **上一个好的流标识符**: 上一个被GOAWAY发送端接收的流标识符。如果没有流可以用来回复，则这个值为0。
- **原因描述长度**: 16位无符号数，表示reason phrase的长度。如果发送方除了错误码之外，并不打算给出详细情况，那么该字段可能为0。
- **原因描述**: 可选的可读的连接关闭的原因。

QUIC 传输参数(QUIC Transport Parameters)

The handshake is responsible for negotiating a variety of transport parameters for a QUIC connection.

QUIC握手用于为QUIC建链协商一系列传输参数。

必要参数(Required Parameters)

- **SFCW** - Stream Flow Control Window. The size in bytes of the stream level flow control window.

- **CFCW** - Connection Flow Control Window. The size in bytes of the connection level flow control window.

- **SFCW**: 流级别的流量控制窗口。字节级别大小。

- **CFCW**: 连接级别的流量控制窗口。字节级别大小。

可选参数(Optional Parameters)

- **SRBF** - Socket receive buffer size in bytes. The peer may want to limit their max CWND to something similar to the socket receive buffer if they fear the peer may sometimes be delayed in reading packets from kernel's socket buffer. Defaults to 256kbytes and has a minimum value of 16kbytes.

- **TCID** - Connection ID truncation. Indicates support for truncated Connection IDs. If sent by a peer, indicates the connection IDs sent to the peer should be truncated to 0 bytes. Useful for cases when a client ephemeral port is only used for a single connection.

- **COPT** - Connection Options are a repeated tag field. The field contains any connection options being requested by the client or server. These are typically used for experimentation and will evolve over time. Example use cases include changing congestion control algorithms and parameters such as initial window.

- **SRBF**: 套接字接收buffer字节大小。对端可能需要限制他们的最大拥塞窗口，防止数据在内核套接字缓冲区读取包数据时产生延迟。默认为256kbytes，最小为16kbytes。

- **TCID**: 连接标识符截断。表示支持截断的连接标识符。如果由对端发送，标志发送到对端的连接标识符必须被截断到0字节。一种有效的场景为: 当客户端的临时端口被用于单个连接的情况。
- **COPT**: 连接选项是一个重复的标签字段。这些字段包含客户端或者服务端所请求的所有连接选项。主要用于实验，后续将进行演进。例如，使用这个参数来初始化拥塞控制算法和其相关参数，如初始窗口。

错误码(QuicErrorCodes)

The number to code mappings for QuicErrorCodes are currently defined in the Chromium source code in src/net/quic/quic_protocol.h. (TODO: hardcode numbers and add them here)

QUIC错误码从数值到编码的映射关系，现在在Chromium源码的src/net/quic/quic_protocol.h中定义。（TODO: 硬编码的数值在这里添加）

- QUIC_NO_ERROR: There was no error. This is not valid for RST_STREAM frames or CONNECTION_CLOSE frames
- QUIC_STREAM_DATA_AFTER_TERMINATION: There were data frames after the a fin or reset.
- QUIC_SERVER_ERROR_PROCESSING_STREAM: There was some server error which halted stream processing.
- QUIC_MULTIPLE_TERMINATION_OFFSETS: The sender received two mismatching fin or reset offsets for a single stream.
- QUIC_BAD_APPLICATION_PAYLOAD: The sender received bad application data.
- QUIC_INVALID_PACKET_HEADER: The sender received a malformed packet header.
- QUIC_INVALID_FRAME_DATA: The sender received an frame data. The more detailed error codes below are prefered where possible.
- QUIC_INVALID_FEC_DATA: FEC data is malformed.
- QUIC_INVALID_RST_STREAM_DATA: Stream rst data is malformed
- QUIC_INVALID_CONNECTION_CLOSE_DATA: Connection close data is malformed.
- QUIC_INVALID_ACK_DATA: Ack data is malformed.
- QUIC_DECRYPTION_FAILURE: There was an error decrypting.
- QUIC_ENCRYPTION_FAILURE: There was an error encrypting.
- QUIC_PACKET_TOO_LARGE: The packet exceeded MaxPacketSize.
- QUIC_PACKET_FOR_NONEXISTENT_STREAM: Data was sent for a stream which did not exist.
- QUIC_CLIENT_GOING_AWAY: The client is going away (browser close, etc.)
- QUIC_SERVER_GOING_AWAY: The server is going away (restart etc.)
- QUIC_INVALID_STREAM_ID: A stream ID was invalid.
- QUIC_TOO_MANY_OPEN_STREAMS: Too many streams already open.
- QUIC_CONNECTION_TIMED_OUT: We hit our pre-negotiated (or default) timeout
- QUIC_CRYPTO_TAGS_OUT_OF_ORDER: Handshake message contained out of order tags.
- QUIC_CRYPTO_TOO_MANY_ENTRIES: Handshake message contained too many entries.
- QUIC_CRYPTO_INVALID_VALUE_LENGTH: Handshake message contained an invalid value length.
- QUIC_CRYPTO_MESSAGE_AFTER_HANDSHAKE_COMPLETE: A crypto message was received after the handshake was complete.
- QUIC_INVALID_CRYPTO_MESSAGE_TYPE: A crypto message was received with an illegal message tag.

- QUIC_SEQUENCE_NUMBER_LIMIT_REACHED: Transmitting an additional packet would cause a packet number to be reused.

- QUIC_NO_ERROR: 无错误。这个值对于RST_STREAM帧和CONNECTION_CLOSE帧无效。
- QUIC_STREAM_DATA_AFTER_TERMINATION: 在FIN或者RESET状态之后仍然有数据到达。
- QUIC_SERVER_ERROR_PROCESSING_STREAM: 服务端的错误终止了流数据处理。
- QUIC_MULTIPLE_TERMINATION_OFFSETS: 发送端的单个流收到了两个不匹配的FIN或者RESET偏移。
- QUIC_BAD_APPLICATION_PAYLOAD: 发送端收到了错误的应用层数据。
- QUIC_INVALID_PACKET_HEADER: 发送端收到了异常的包头。
- QUIC_INVALID_FRAME_DATA: 发送端收到一个帧数据，更多的细节的错误码会优先选择。
- QUIC_INVALID_FEC_DATA: 异常的FEC数据。
- QUIC_INVALID_RST_STREAM_DATA: 流RST数据异常。
- QUIC_INVALID_CONNECTION_CLOSE_DATA: 连接关闭数据异常。
- QUIC_INVALID_ACK_DATA: Ack数据异常。
- QUIC_DECRYPTION_FAILURE: 解密错误。
- QUIC_ENCRYPTION_FAILURE: 加密错误。
- QUIC_PACKET_TOO_LARGE: 包大小超过最大值。
- QUIC_PACKET_FOR_NONEXISTENT_STREAM: 数据发送到一个不存在的流。
- QUIC_CLIENT_GOING_AWAY: 客户端关闭(浏览器关闭等)。
- QUIC_SERVER_GOING_AWAY: 服务端关闭(重启等)。
- QUIC_INVALID_STREAM_ID: 无效的流标识符。
- QUIC_TOO_MANY_OPEN_STREAMS: 打开的流过多。
- QUIC_CONNECTION_TIMED_OUT: 我们达到预协商(或者默认)的超时时间。
- QUIC_CRYPTO_TAGS_OUT_OF_ORDER: 握手信息中包含了乱序的标签。
- QUIC_CRYPTO_TOO_MANY_ENTRIES: 握手信息中包含过多的实例。
- QUIC_CRYPTO_INVALID_VALUE_LENGTH: 握手信息中包含无效的长度值。
- QUIC_CRYPTO_MESSAGE_AFTER_HANDSHAKE_COMPLETE: 握手完成后收到一个加密信息。
- QUIC_INVALID_CRYPTO_MESSAGE_TYPE: 接收到一个非法标签的加密信息。
- QUIC_SEQUENCE_NUMBER_LIMIT_REACHED: 传输一个额外包，可能导致包号重用。

优先级(Priority)

(TODO: implement)

QUIC will use the HTTP/2 prioritization mechanism. Roughly, a stream may be dependent on another stream. In this situation, the “parent” stream should effectively starve the “child” stream. In addition, parent streams have an explicit priority. Parent streams should not starve other parent streams, but should make progress proportional to their relative priority.

HTTP/2 Layering over QUIC

Since QUIC integrates various HTTP/2 mechanisms with transport mechanisms, QUIC implements a number of features that are also specified in HTTP/2. As a result, QUIC allows HTTP/2 mechanisms to be replaced by QUIC’s implementation, reducing complexity in the HTTP/2 protocol. This section briefly describes how HTTP/2 semantics can be offered over a QUIC implementation.

###Stream Management

When HTTP/2 headers and data are sent over QUIC, the QUIC layer handles most of the stream management. HTTP/2 Stream IDs are replaced by QUIC Stream IDs. HTTP/2 does not need to do any explicit stream framing when using QUIC—data sent over a QUIC stream simply consists of HTTP/2 headers or body. Requests and responses are considered complete when the QUIC stream is closed in the corresponding direction.

Stream flow control is handled by QUIC, and does not need to be re-implemented in HTTP/2. QUIC's flow controller replaces the two levels of poorly matched flow controllers in current HTTP/2 deployments—one at the HTTP/2 level, and the other at the TCP level.

HTTP/2 Header Compression

QUIC implements HPACK header compression for HTTP/2 [RFC7541], which unfortunately introduces some Head-of-Line blocking since HTTP/2 header blocks must be decompressed in the order they were compressed.

Since streams may be processed in arbitrary order at a receiver, strict ordering across headers is enforced by sending all headers on a dedicated headers stream, with Stream ID 3. An HTTP/2 receiver using QUIC would thus process data from a stream only after receiving the corresponding header on the headers stream.

Future work will tweak the compressor and decompressor in QUIC so that the compressed output does not depend on unacked previous compressed state. This could be done, perhaps, by creating "checkpoints" of HPACK state which are updated when headers have been acked. When compressing headers QUIC would only compress relative to the previous "checkpoint".

Parsing HTTP/2 Headers

Bytes sent on the dedicated headers stream are simply HTTP/2 HEADERS frames. The exact layout of these frames is described in Section 6.2 of [RFC7540].

QUIC Negotiation in HTTP

The Alternate-Protocol header is used to negotiate use of QUIC on future HTTP requests. To specify QUIC as an alternate protocol available on port 123, a server uses:

```
--- src
"Alternate-Protocol: 123:quic"
---
```

When a client receives a Alternate-Protocol header advertising QUIC, it can then attempt to use QUIC for future secure connections on that domain. Since middleboxes and/or firewalls can block QUIC and/or UDP communication, a client should implement a graceful fallback to TCP when QUIC reachability is broken.

Note that the server may reply with multiple field values or a comma-separated field value for Alternate-Protocol to indicate the various transports it supports.

A server can also send a header to notify that QUIC should not be used on this domain. If it sends the alternate-protocol-required header, the client should remember to not use QUIC on that domain in future, and not do any UDP probing to see if QUIC is available.

Handshake Protocol Requirements

QUIC provides a dedicated stream (Stream ID 1) to be used for performing a combined connection and security handshake, but the details of this handshake protocol are out of this document's scope. However, QUIC does impose a number of requirements on any such handshake protocol. The following list of requirements documents properties of the current prototype handshake which should be provided by any future handshake protocol.

Connection Establishment in 0-RTT

The QUIC handshake protocol manages to successfully achieve 0-RTT for most connections, and is critical to QUIC's latency improvements.

Source Address Spoofing Defense

TCP verifies the client's address by burning a round trip on the SYN, SYN_ACK exchange. QUIC uses a source address token delivered by the server in a previous connection.

Opaque Source Address Tokens

QUIC servers store a number of pieces of data in the source address token, for use on a subsequent connection from the same client. This includes recently used source addresses, measured bandwidth to the client, and server-designated connection IDs (for Stateless REJs). An alternative handshake protocol's analog of a source address token needs to be (i) opaque at the client, and (ii) large enough to permit these bits of information to be stored. Alternatively, the handshake protocol should have a different method to store this information at the client.

Transport Parameter Negotiation

In addition to negotiating crypto parameters, the QUIC handshake also negotiates QUIC and HTTP/2 level parameters, including max open QUIC streams and other QUIC connection options.

Certificate Compression

The QUIC handshake compresses certificates so that an REJ, including the common Google certificate chain, is able to fit into two 1350 byte packets. This helps to reduce the amplification attack footprint of QUIC without reducing 0-RTT rate.

Server Config Update

QUIC uses a Server Config Update (SCUP) message to refresh the source-address token (STK) and server config mid-connection, extending the period over which 0-RTT connections can be established by the client.

Recent Changes By Version

- Q009: added priority as the first 4 bytes on spdy streams.
- Q010: renumber the various frame types
- Q011: shrunk the fnv128 hash on NULL encrypted packets from 16 bytes to 12 bytes.
- Q012: optimize the ack frame format to reduce the size and better handle ranges of nacks, which should make truncated acks virtually impossible. Also adding an explicit flag for truncated acks and - moving the ack outside of the connection close frame.
- Q013: Compressed headers for *all* data streams are serialized into a reserved stream. This ensures serialized handling of headers, independent of stream cancellation notification.
- Q014: Added WINDOW_UPDATE and BLOCKED frames, no behavioral change.
- Q015: Removes the accumulated_number_of_lost_packets field from the TCP and inter arrival congestion feedback frames and adds an explicit list of recovered packets to the ack frame.
- Q016: Breaks out the sent_info field from the ACK frame into a new STOP_WAITING frame.
- Changed GUID to Connection ID
- Q017: Adds stream level flow control
- Q018: Added a PING frame
- Q019: Adds session/connection level flow control
- Q020: Allow endpoints to set different stream/session flow control windows
- Q021: Crypto and headers streams are flow controlled (at stream level)
- Q023: Ack frames include packet timestamps
- Q024: HTTP/2-style header compression
- Q025: HTTP/2-style header keys. Removal of error_details from the RST_STREAM frame.
- Q026: Token binding, adds expected leaf cert (XLCT) tag to client hello
- Q027: Adds a nonce to the server hello
- Q029: Server and client honor QUIC_STREAM_NO_ERROR on early response
- Q030: Add server side support of certificate transparency.
- Q031: Adds a SHA256 hash of the serialized client hello messages to crypto proof.
- Q032: FEC related fields are removed from wire format.
- Q033: Adds an optional diversification nonce to packet headers, and eliminates the 2 byte and 4 byte connection ID length public flags.
- Q034: Removes entropy and private flags and changes the ack frame from nack ranges to ack ranges and removes truncated acks.
- Q035: Allows each endpoint to independently set maximum number of supported incoming streams using the MIDS ("Maximum Incoming Dynamic Streams") tag instead of the older MSPC ("Maximum Streams Per - Connection") tag.
- Q036: Adds support for inducing head-of-line blocking between streams via the new FHOL tag in the handshake.

Contributors

This protocol is the outcome of work by many engineers, not just the authors of this document. The design and rationale behind QUIC draw significantly from work by Jim Roskind. In alphabetical order, the contributors to the project are: Britt Cyr, Jeremy Dorfman, Ryan Hamilton, Jana Iyengar, Fedor Kouranov, Charles Krasic, Jo Kulik, Adam Langley, Jim Roskind, Robbie Shade, Satyam Shekhar, Cherie Shi, Ian Swett, Raman Tenneti, Victor Vasiliev, Antonio Vicente, Patrik Westin, Alyssa Wilk, Dale Worley, Fan Yang, Dan Zhang, Daniel Ziegler.

Acknowledgments

Special thanks are due to the following for helping shape QUIC and its deployment: Chris Bentzel, Misha Efimov, Roberto Peon, Alistair Riddoch, Siddharth Vijayakrishnan, and Assar Westerlund. QUIC has also benefited immensely from discussions with folks in private conversations and public ones on the proto-quic@chromium.org mailing list.

QUIC