quic_doc_zh Documentation

发布 *1.0.0*

ZhanPw

2019 年 **02** 月 **23** 日

Contents:

CHAPTER 1

贡献者

这个协议不是本文档作者一人的成果，而是是许多工程师共同努力的成果。 QUIC背后的设计和理论基础来自Jim Roskind的工作[1]。 按字母顺序排列，项目的贡献者为：Britt Cyr, Ryan Hamilton, Jana Iyengar, Fedor Kouranov, Charles Krasic, Jo Kulik, Adam Langley, Jim Roskind, Robbie Shade, Satyam Shekhar, Cherie Shi, Ian Swett, Raman Tenneti, Antonio Vicente, Patrik Westin, Alyssa Wilk, Dale Worley, Dan Zhang, Daniel Ziegler.

[1] https://goo.gl/dMVtFi

CHAPTER 2

致谢

特别感谢以下的各位帮助塑造QUIC及其部署： Chris Bentzel，Misha Efimov，Roberto Peon，Alistair Riddoch，Siddharth Vijayakrishnan和Assar Westerlund。 在proto-quic@chromium.org邮件列表中，QUIC也从人们在私人对话和公共对话中的讨论中获益匪浅。

CHAPTER 3

介绍

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。

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。

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。

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]。

CHAPTER 4

术语和定义

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。

CHAPTER 5

QUIC 概述

5.1 连接建立延迟

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。

5.2 灵活的拥塞控制

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。

5.3 流和连接的流量控制

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。

5.4 多路复用

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。

5.5 认证和加密的首部和载荷

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。

5.6 连接迁移

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。

CHAPTER 6

包类型和格式

6.1 QUIC公共包头

传输的所有 QUIC 包以大小介于1至19字节的公共包头开始。公共包头的格式如下：：

```
0         1       2        3        4            8
+---------+--------+--------+--------+--------+---   ---+
| Public  | Connection ID (0, 8, 32, or 64)       ...   | ->
| Flags(8)| (variable length)                           |
+---------+--------+--------+--------+--------+---   ---+


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


  13 14 15 16 17 18
  +--------+--------+--------+--------+--------+--------+
  | Packet Number (8, 16, 32, or 48)                     |
  | (variable length)                                    |
  +--------+--------+--------+--------+--------+--------+
```

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。
- 0x0C处的两个比特指示分组中存在的连接ID的大小。 在所有分组中必须将这些比特设置为0x0C， 直到给定方向协商更换为一个不同的值（例如，客户端可以请求呈现更少的连接ID字节）。
  - 0x0C 表示连接ID占了8个字节
  - 0x08 表示连接ID占了4个字节
  - 0x04 表示连接ID占了1个字节
  - 0x00 表示连接ID被删除了
- 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 暂时未使用，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)。

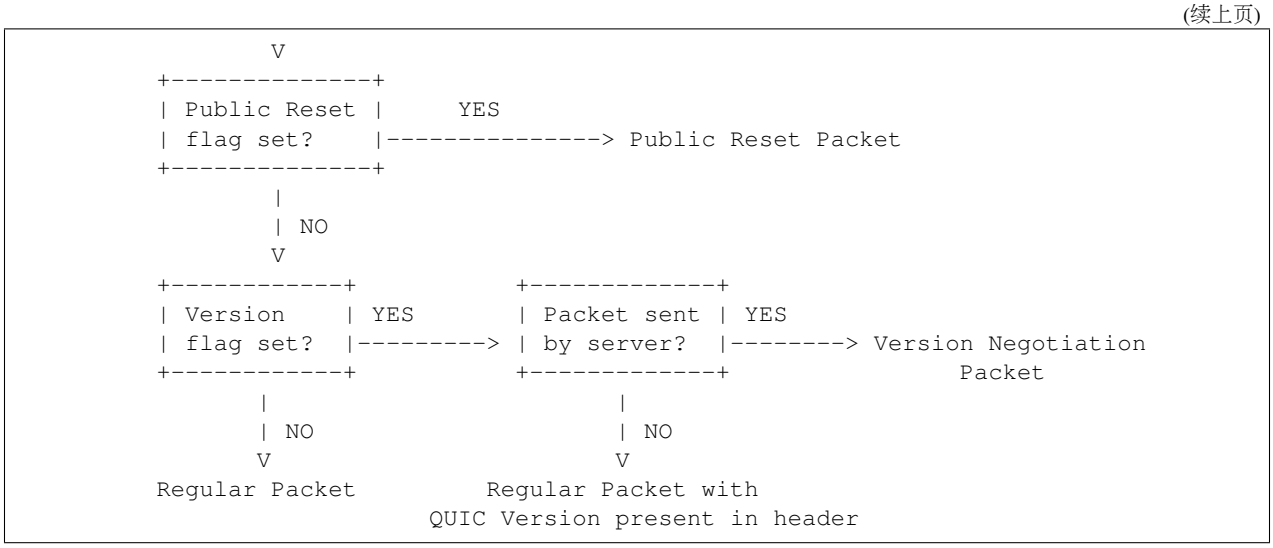
  任何截断的包号应该被推断为具有最接近但大于传输最初包含了截断包号的包的对端的最大已知包号的值。包号的发送部分匹配推断值的最低位。

公共标记处理流程图如下：：

```
Check the public flags in public header
                    |
                    |
```

(continues on next page)

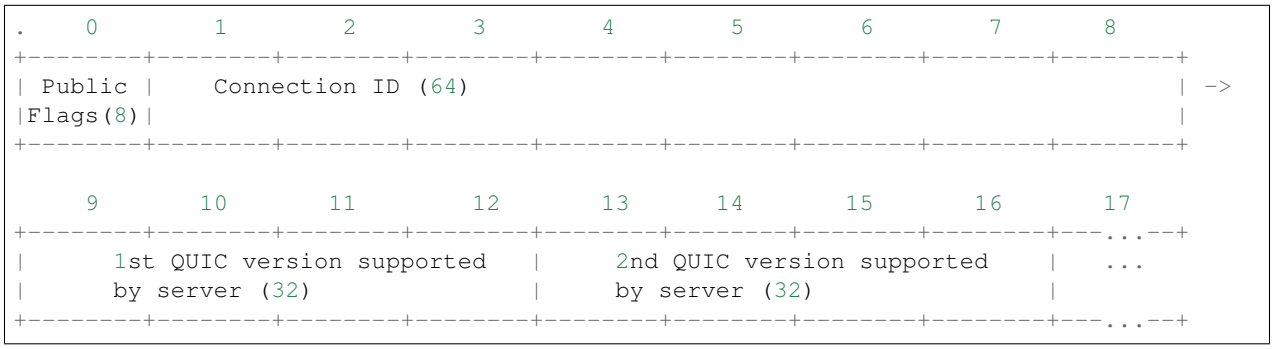
(续上页)



6.2 特殊包

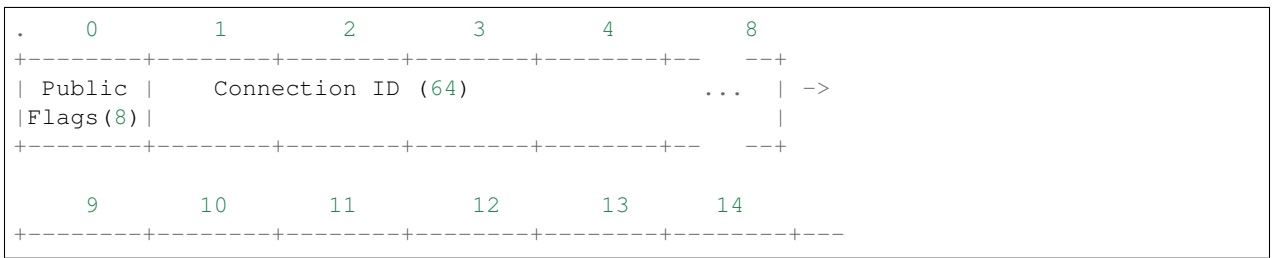
6.2.1 版本协商包

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：：



6.2.2 公共复位包

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]）：：



(continues on next page)

(续上页)

```
|      Quic Tag (32)                |  Tag value map       ... ->
|         (PRST)                    |  (variable length)
+--------+--------+--------+--------+--------+--------+---
```

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。)

6.3 普通包

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，明文以私有表头（Private Header）开头。:

```
0        1
+---------+---------+
| Private | FEC (8) |
| Flags(8)| (opt)   |
+---------+---------+
```

私有表头中的字段如下：

- 私有标志
  - 0x01 = FLAG_ENTROPY - 对于数据分组，表示该分组包含1比特的熵，对于fec分组，包含受保护分组的熵的异或。
  - 0x02 = FLAG_FEC_GROUP - 表示是否存在fec字节。
  - 0x04 = FLAG_FEC - 表示该数据包表示FEC数据包。
- FEC (FEC组编号偏移): FEC组编号是FEC组中第一个数据包的包编号。 FEC组编号偏移是一个8位无符号值，应从当前数据包的包编号中减去该编号，以产生该包的FEC组编号。 仅当私有标志包含FLAG_FEC_GROUP时才会出现此情况。 单个FEC组内的所有数据包必须具有编码为相同字节数的数据包编号（即，数据包编号编码在组中不得更改）

(TODO: 文档化加密和解密的输入，并描述试用解密)

6.3.1 帧包

在私人表头的格式之外，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：:

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

6.3.2 FEC包

FEC分组（具有FLAG_FEC集的那些分组）具有载荷：其仅包含FEC组中的每个数据分组的空填充有效载荷的XOR。 FEC数据包还必须设置FLAG_FEC_GROUP。:

```
+-----...----+
| Redundancy |
+-----...----+
```

CHAPTER 7

QUIC 连接的生命周期

7.1 连接建立

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。

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）。

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。

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。

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。

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。

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。

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。

7.2 数据传输

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。

7.2.1 QUIC 流的生命周期

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。）

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。

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。

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。

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。

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. 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。

7.3 连接终止

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。

CHAPTER 8

帧类型和格式

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。

8.1 帧类型

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：：

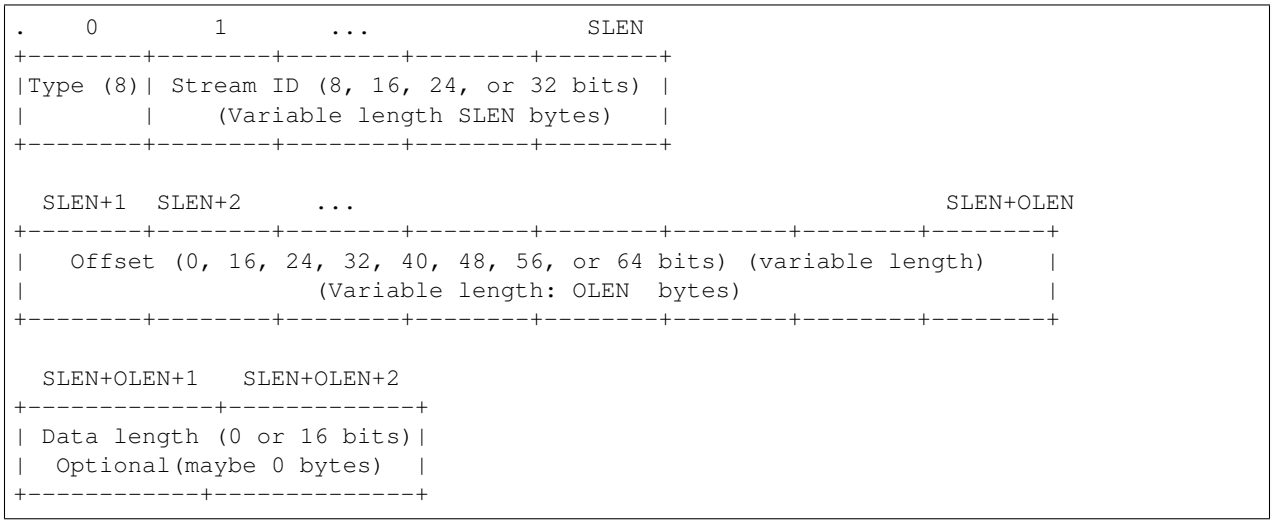
Type-field value	Control Frame-type
1fdooossB	STREAM
01ntllmmB	ACK
001xxxxxB	CONGESTION_FEEDBACK

当前定义的普通帧类型如下：：

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

8.2 STREAM帧

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：：



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。

8.3 ACK帧

发送ACK帧以通知端已经接收了哪些分组，以及接收器仍然认为丢失了哪些分组（可能需要重新发送丢失分组的内容）。

QUIC的ACK帧的设计不同于TCP和SCTP的SACK表示，因为QUIC ACK指示到目前为止观察到的最大分组数， 其后是丢失分组列表，或NACK，范围指示在该分组号下面接收的分组中的间隙。

为了将NACK范围限制为尚未传送给对等体的NACK范围，端周期性地发送STOP_WAITING帧， 该信号通知接收器停止等待低于指定序列号的分组，从而提高接收端“最小未ack（least unacked）”分组号。

因此，ACK帧的发送方仅报告所接收的最小未被ack和所报告的最大观察分组号之间的那些NACK范围。 框架如下：：

```
.    0         1                                   N
+--------+----------+-------------------------------------------------------+
| Type   | Received |                   Largest Observed                    |
| (8)    | Entropy  |                (8, 16, 32, or 48 bits)                |
+--------+----------+-------------------------------------------------------+


    N+1      N+2 N+3 N+4 N+8
+--------+--------+-------------+----------+---------------------------------+
|    Ack Delay    |     Num     | Delta    |         First Timestamp         |
|     Time (16)   | Timestamp   | Largest  |            (32 bits)           |
|                 |     (8)     | Observed |                                 |
+--------+--------+-------------+----------+---------------------------------+


     N+9          N+11 - X
+----------+--------------------+
| Delta    | Time Since         |
| Largest  | Previous Timestamp|  <-- Repeat (NumTimestamp - 1) times
| Observed |    (16 bits)       |
+----------+--------------------+


    X                        X+1 - Y                              Y+1
+--------+-----------------------------------------------------+---------+
| Number |       Missing Packet Sequence Number Delta          | Range   |
| Ranges |             (8, 16, 32, or 48 bits)                 | Length  |
| (opt)  |          (repeats Number Ranges times)              | (Repeat)|
+--------+-----------------------------------------------------+---------+


    Y+2                         Y+3 - Z
+--------+-------------------------------------------------------+
| Number |               Revived Packet Number                   |
| Revived|  (8, 16, 32, or 48 bits, same as Largest Observed)    |
| (opt)  |            (repeats Number Revived times)             |
+--------+-------------------------------------------------------+
```

ACK帧中的字段如下：

- 帧类型(Frame Type)：帧类型字节是包含各种标志的8位值（01ntllmmB）。
  - 前两位必须设置为01，表示这是一个ACK帧。
  - 'n'位表示帧是否具有任何NACK范围。
  - 't'位指示ACK帧是否已被截断。 当完整的ACK帧不适合单个QUIC数据包， 或者当NACK范围的数量超过可报告的NACK范围的最大数量（255）时，可能发生截断。 截断时，ACK帧将最大观察到的数据包数量限制为可以报告的最大数据包，即使接收方可 能已收到数据包数量大于观察到的最大数据包的数据包。
  - 两个'll'位将最大观察字段(Largest Observed field)的长度编码为1,2,4或6字节长。
  - 两个"mm"位将缺失数据包序列号(Missing Packet Sequence Number) Delta字段的长度编码为1,2,4或6个字节长。
- 接收熵(Received Entropy)：一个8位无符号值，指定所有接收数据包中直到最 大观察数据包的熵的累积散列。 熵累积将在本节后面介绍。

- 最大观察值(Largest Observed)：可变大小的无符号值，表示对等方观察到的最 大数据包编号。 当截断ACK帧时，它指示大于所接收的指定最大观测数的分组数， 但是关于那些附加接收的信息不能填入该帧（通常由于分组大小限制）。
- 确认延迟时间(Ack Delay Time)：16位无符号浮点数，具有11个明确的尾数位和5位显式指数， 指定从接收到最大观测值到发送此Ack帧之间经过的时间（以微秒为单位）。 在IEEE 754之后，比特格式被宽松地建模。例如，1微秒表示为0x1，其指数为零， 以5个高阶位表示，尾数为1，以11个低阶位表示。 当显式指数大于零时，在尾数中 假定隐含的高阶12位为1。 例如，浮动值0x800的显式指数为1，显式尾数为0，但其 有效尾数为4096（假设第12位为1）。 此外，实际指数比显式指数小1，值表示4096微秒。 任何大于可表示范围的值都被钳制为0xFFFF。
- **时间戳段(Timestamp Section):**
  - 时间戳数(Num Timestamp): 一个8位无符号值，指定此ack帧中包含的时间戳数。 在时间戳中会有许多对<packet number，timestamp>。
  - 已观察最大差值(Delta Largest Observed)：一个8位无符号值，指定从第一个时间戳到观察到的最大时间 戳的数据包数量差值。 因此，分组数是观察到的最大值减去观察到的最大值。
  - 第一个时间戳(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(Delta Largest Observed)，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 确认延迟时间(Ack Delay Time)相同。
- **丢失数据包段（Missing Packet Section:）：**
  - Num范围（Num Ranges）：一个可选的8位无符号值，指定最大观察值和最小未 封装之间丢失数据包范围的数量。 仅在'n'标志位为1时才出现。
  - 缺少数据包序列号增量（Missing Packet Sequence Number Delta）：可变大小的数据包数量增量。 对于第一个丢失的数据包范围，它是观察到的最大数据包的增量。 对于后续的nack范围，它是范围之间接 收的数据包数。 在第一个nack范围的情况下，值0指定报告为观察到的最大数据包丢失。 在后来的nack范围的情况下，值0表示丢失的分组范围是连续的（仅当一行中超过256个分组丢失时使用）。
  - 范围长度（Range Length）：一个8位无符号值，指定的值小于该范围内的连续nack数。
- 已恢复数据包部分：
  - Num Revived：一个8位无符号值，指定通过FEC恢复的已恢复数据包的数量。 就像Num Ranges字段一样，只有'n'标志位为1时才会出现此字段。
  - 恢复的分组序列号：可变大小的无符号值，表示对等体通过FEC恢复的分组。 其长度与"最大观察"字段的长度相同。 此列表中的所有数据包编号按升序排序（最小的第一个）， 并且还必须存在于NACK范围列表中。

8.3.1 熵积累（Entropy Accumulation）

分组子集（对于接收器或发送器已知）的熵比特被累积为8比特无符号值，并且类似地在STOP_WAITING帧和ACK帧中呈现。 如果我们将E（k）定义为分组号k中存在的FLAG_ENTROPY比特，则第k个分组的贡献C（k）被定义为左移k mod 8比特的E（k）。 然后，对于期望子集中的所有分组，累积熵是贡献C（k）的按位XOR和。

8.4 STOP_WAITING 帧

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，使用与封闭数据包表头的数据包编号相同的编码长度 (在QUIC Frame Packet的Public Flags字段中指定。) 这个帧如下：：

```
.    0        1        2        3        4       5       6       7
+--------+--------+--------+--------+--------+--------+-------+-------+
|Type (8)|Sent    |   Least unacked delta (8, 16, 32, or 48 bits)     |
|        |Entropy |                 (variable length)                 |
+--------+--------+--------+--------+--------+--------+--------+------+
```

STOP_WAITING帧中的字段如下：

- 帧类型（Frame Type）：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。
- 发送熵（Sent Entropy）：一个8位无符号值，指定所有已发送数据包中的熵的累积哈希值，该数据包的数据包数量小于最小未ack的数据包。 [有关此计算的详细信息，请参阅“ACK帧”部分中的“熵累积”部分。]
- 最小未确认差值（Least Unacked Delta）：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。

8.5 WINDOW_UPDATE 帧

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：：

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

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。

8.6 BLOCKED 帧

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：：

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

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。

8.7 CONGESTION_FEEDBACK 帧

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位预留将来使用。

8.8 PADDING 帧

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。

8.9 RST_STREAM 帧

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：：

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

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。

8.10 PING 帧

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。

8.11 CONNECTION_CLOSE 帧

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：：

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

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。

8.12 GOAWAY 帧

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：：

```
.    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)
+--------+--------+--------+----- ...
```

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。

CHAPTER 9

QUIC 传输参数

QUIC握手用于为QUIC连接协商一系列传输参数。

9.1 必要参数

- SFCW: 流级别的流量控制窗口。大小为字节级别。
- CFCW: 连接级别的流量控制窗口。大小为字节级别。

9.2 可选参数

- 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。

CHAPTER 10

QUIC 错误码

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: 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。

CHAPTER 11

优先级

(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.

QUIC将使用HTTP / 2优先级排序机制。 粗略地说，流可能依赖于另一个流。 在这种情况下，“父”流应该有效地使“子”流饿死（strave）。 此外，父流具有明确的优先级。 父流不应该饿死（strave）其他父流，但应该取得与其相对优先级成比例的进度。

CHAPTER 12

建立在QUIC之上的HTTP/2

由于QUIC将各种HTTP / 2机制与传输机制集成在一起，因此QUIC实现了许多在HTTP / 2中也指定的功能。 因此，QUIC允许HTTP / 2机制被QUIC的实现取代，从而降低了HTTP / 2协议的复杂性。 本节简要介绍如何通过QUIC提供HTTP / 2语义实现。

12.1 流量管理

当通过QUIC发送HTTP / 2报头和数据时，QUIC层处理大部分流管理。 HTTP / 2流ID由QUIC流ID替换。 使用QUIC时，HTTP / 2不需要进行 任何显式的流帧 - 通过QUIC流发送的数据只包含HTTP / 2报头或正文。 当QUIC流在相应方向上关闭时，认为请求和响应完成。

流控制由QUIC处理，不需要在HTTP / 2中重新实现。 QUIC的流量控制器取代了当前HTTP / 2部署中两个 级别不匹配的流量控制器 - 一个在HTTP / 2级别，另一个在TCP级别。

12.2 HTTP/2 头部压缩

QUIC为HTTP / 2实现了HPACK头压缩[4]，遗憾的是，它引入了一些HOL（Head-of-Line）阻塞， 因为HTTP / 2头块必须按它们被压缩的顺序解压缩。

由于可以在接收器处以任意顺序处理流，因此通过使用流ID 3在专用报头流上发送所有报 头来强制执行对报头的严格排序。因此，使用QUIC的HTTP/2接收器仅在接收到报头流 上的相应报头之后才处理来自流的数据。

未来的工作将调整QUIC中的压缩器和解压缩器，以便压缩输出不依赖于未ack的先前压缩状态。 这可以通过创建HPACK状态的“检查点”来完成，这些检查点在标题被ack时更新。 压缩标头时，QUIC只会相对于先前的“检查点”进行压缩。

12.3 解析 HTTP/2 头部

在专用标头流上发送的字节只是HTTP / 2 HEADERS帧。 RFC 7540 [5]中描述了这些帧的确切布局。

12.4 持久连接

与使用TCP时不同，QUIC的基础连接保证是持久的。 因此，HTTP“Connection”表头不适用。 为了获得最佳性能，预计客户端不会关闭QUIC连接，直到用户使用该连接加载完成了所有网页， 或者直到服务器关闭连接。

12.5 HTTP中的QUIC协商

’Alternate-Protocol’表头用于协商在未来的HTTP请求中使用QUIC。 要将QUIC指定为端口123上可用的备用协议，服务器将使用：：

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

当客户端收到公告QUIC的备用协议表头’Alternate-Protocol’时，它可以尝试将QUIC用于该域上的未来安全连接。 由于中间盒和/或防火墙可以阻止QUIC和/或UDP通信，因此当QUIC可达性被破坏时，客户端应该优雅回退到使用传统的TCP。

注意，服务器可以使用多个字段值或’Alternate-Protocol’的逗号分隔字段值进行回复，以指示它支持的各种传输方式。

服务器还可以发送标头以通知不应在此域上使用QUIC。 如果它发送备用协议所需的头，客户端应该记住以后不在该域上使用QUIC， 并且不进行任何UDP探测以查看QUIC是否可用。

CHAPTER 13

握手协议条件

QUIC提供了一个专用流（流ID 1），用于执行组合连接和安全握手， 但此握手协议的详细信息超出了本文档的范围。 但是，QUIC确实对任何此类握手协议施加了许多要求。 以下需求列表记录了当前原型握手的属性，这些属性应由任何未来的握手协议提供。

13.1 以 0-RTT 建立连接

QUIC握手协议可以成功实现大多数连接的0-RTT，并且对于QUIC的延迟改进至关重要。

13.2 源地址欺骗防御

TCP通过在SYN，SYN_ACK交换的过程中进行往返来验证客户端的地址。 QUIC使用服务器在先前连接中提供的源地址令牌。

13.3 不透明的源地址令牌

QUIC服务器在源地址令牌中存储多个数据，以用于来自同一客户端的后续连接。 这包括最近使用的源地址，测量的客户端带宽和服务器指定的连接ID（对于无状态REJ）。

替代握手协议的源地址令牌的模拟需要（i）在客户端不透明， 以及（ii）足够大以允许存储这些信息位。 或者，握手协议应该具有不同的方法来在客户端存储该信息。

13.4 传输参数协商

除了协商加密参数之外，QUIC握手还协商QUIC和HTTP/2级别的参数，包括最大开放QUIC流和其他QUIC连接选项。

13.5 证书压缩

QUIC握手压缩证书，以便REJ（包括常见的Google证书链）能够容纳两个1350字节的数据包。 这有助于在不降低0-RTT速率的情况下减少QUIC的放大攻击足迹（footprint）。

13.6 服务器配置更新

QUIC使用服务器配置更新（SCUP）消息刷新源地址令牌（STK）和服务器配置中间连接， 从而延长客户端可以建立0-RTT连接的时间段。

CHAPTER 14

Indices and tables

- genindex
- modindex
- search