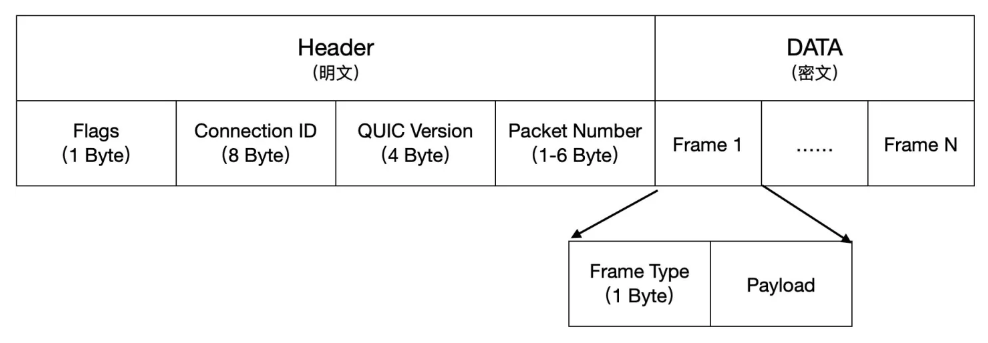
# Quic协议设计思想与迁移

1. **设计背景与核心目标**

QUIC（Quick UDP Internet Connections）协议由Google于2012年提出，旨在解决传统TCP+TLS协议在现代互联网环境中暴露的**延迟高、部署僵化、队头阻塞**等核心问题。其设计目标聚焦于：

1. **降低连接延迟**：减少握手往返次数，实现0-RTT数据传输
2. **提升传输效率**：通过多路复用与独立流设计避免队头阻塞
3. **增强灵活性**：在用户空间实现协议，支持动态拥塞控制算法更新
4. **保障安全性**：默认集成TLS 1.3，实现端到端加密
5. **支持网络适应性**：通过连接ID实现无缝网络切换
6. **核心设计思想**
7. 数据格式

一个 QUIC 数据包的格式如下：

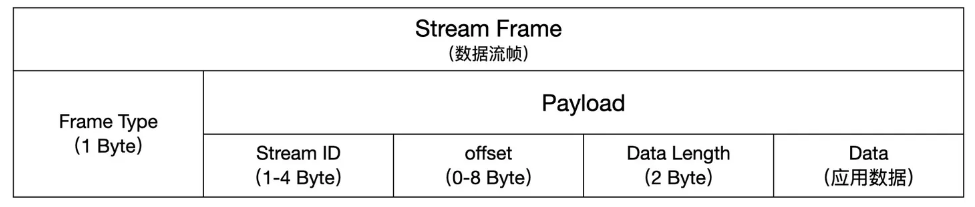


由 header 和 data 两部分组成。

header 是明文的，包含 4 个字段：Flags、Connection ID、QUIC Version、PacketNumber；

data 是加密的，可以包含 1 个或多个 frame，每个 frame 又分为 type 和payload，其中 payload 就是应用数据；

数据帧有很多类型：Stream、ACK、Padding、Window\_Update、Blocked 等，这里重点介绍下用于传输应用数据的 Stream 帧。



****Frame Type：**** 帧类型，占用 1 个字节



（1）Bit7：必须设置为 1，表示 Stream 帧

（2）Bit6：如果设置为 1，表示发送端在这个 stream 上已经结束发送数据，流将处于半关闭状态

（3）Bit5：如果设置为 1，表示 Stream 头中包含 Data length 字段

（4）Bit432：表示 offset 的长度。000 表示 0 字节，001 表示 2 字节，010 表示 3 字节，以此类推

（5）Bit10：表示 Stream ID 的长度。00 表示 1 字节，01 表示 2 字节，10 表示 3 字节，11 表示 4 字节

****Stream ID：****流 ID，用于标识数据包所属的流。后面的流量控制和多路复用会涉及到

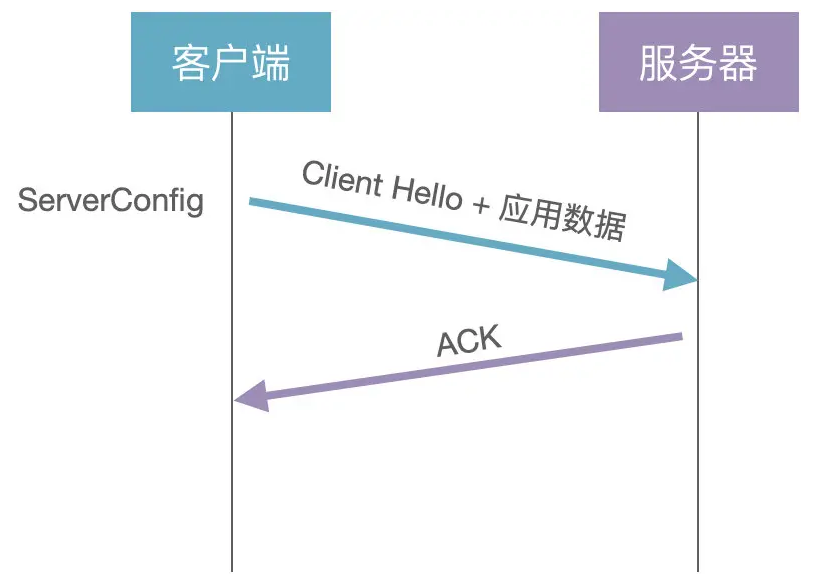
**Offset**：偏移量，表示该数据包在整个数据中的偏移量，用于数据排序。

****Data Length：****数据长度，占用 2 个字节，表示实际应用数据的长度

****Data：****实际的应用数据

1. 建立连接
   1. 0-RTT 握手

客户端缓存了 ServerConfig（B=b\*G%P），下次建连直接使用缓存数据计算通信密钥：

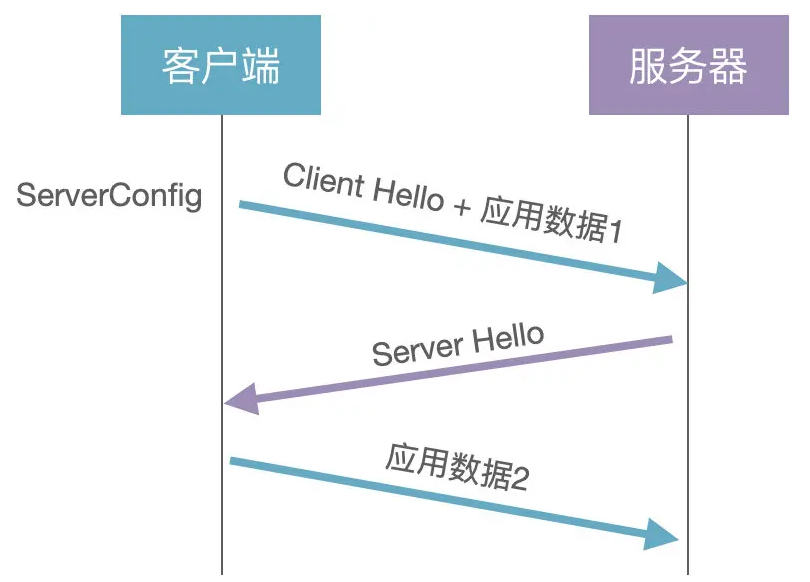


1. 客户端：生成随机数 c，选择公开的大数 G 和 P，计算 A=c\*G%P，将 A 和 G 发送给服务器，也就是 Client Hello 消息
2. 客户端：客户端直接使用缓存的 ServerConfig 计算通信密钥 KEY = c*B = c*b\*G%P，加密发送应用数据
3. 服务器：根据 Client Hello 消息计算通信密钥 KEY = b*A = b*c\*G%P

客户端不需要经过握手就可以发送应用数据，这就是 0-RTT 握手

* 1. 前向安全

前向安全是指用来产生会话密钥的长期密钥泄露出去，不会泄漏以前的通讯内容。

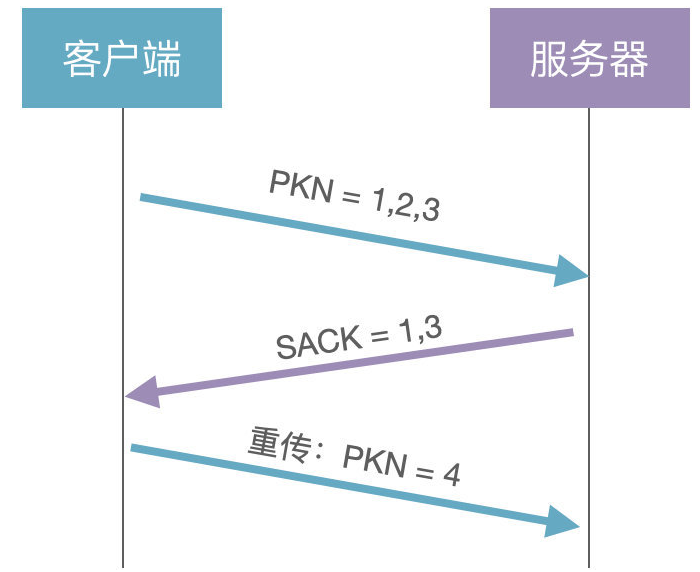


1. 客户端：生成随机数 a，选择公开的大数 G 和 P，计算 A=a\*G%P，将 A 和 G 发送给服务器，也就是 Client Hello 消息
2. 客户端：客户端直接使用缓存的 ServerConfig 计算初始密钥 initKey = a*B = a*b\*G%P，加密发送应用数据 1
3. 服务器：根据 Client Hello 消息计算初始密钥 initKey = b*A = b*a\*G%P
4. 服务器：生成随机数 c，计算 C=c\*G%P，使用 initKey 加密 C，发送给客户端，也就是 Server Hello 消息
5. 客户端：使用 initKey 解码获取 C，计算会话密钥 sessionKey = a*C = a*c\*G%P，加密发送应用数据 2
6. 服务器：计算会话密钥 sessionKey = c*A = c*a\*G%P，解密获取应用数据 2

客户端缓存的 ServerConfig 是服务器静态配置的，是可以长期使用的。客户端通过 ServerConfig 实现 0-RTT 握手，使用会话密钥 sessionKey 保证通信数据的前向安全。

* 1. 可靠传输
     1. 确认包被接收

通过包号（PKN）和确认应答（SACK）

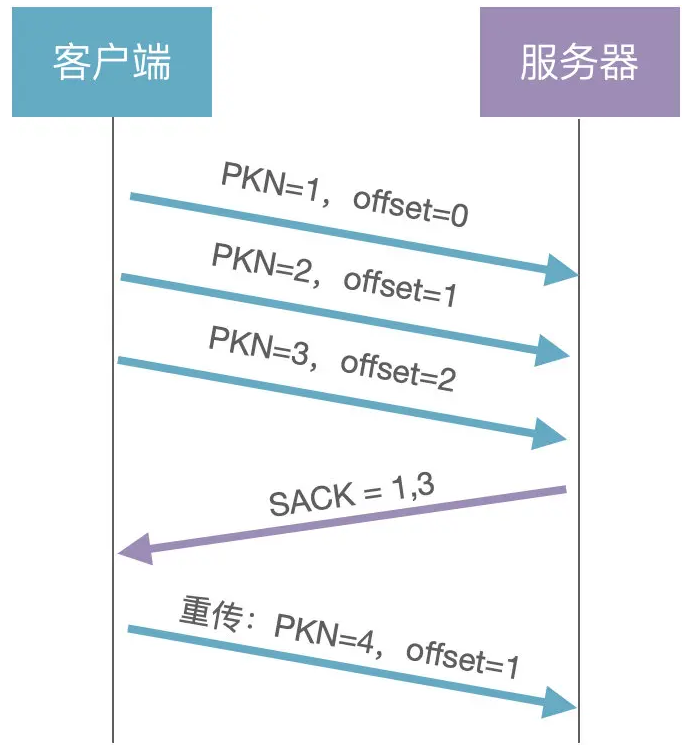


1. 客户端：发送 3 个数据包给服务器（PKN = 1，2，3）
2. 服务器：通过 SACK 告知客户端已经收到了 1 和 3，没有收到 2
3. 客户端：重传第 2 个数据包（PKN=4）

QUIC 的数据包号是单调递增的。也就是说，之前发送的数据包（PKN=2）和重传的数据包（PKN=4），虽然数据一样，但包号不同。

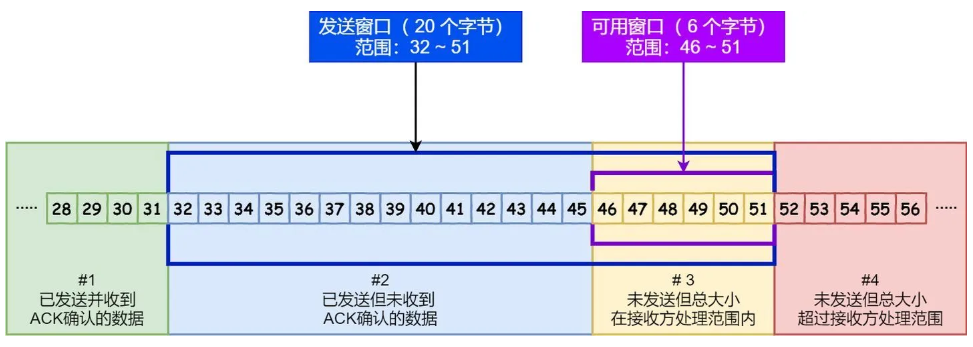
* + 1. 数据有序性的保证

通过数据偏移量 offset每个数据包都有一个 offset 字段，表示在整个数据中的偏移量

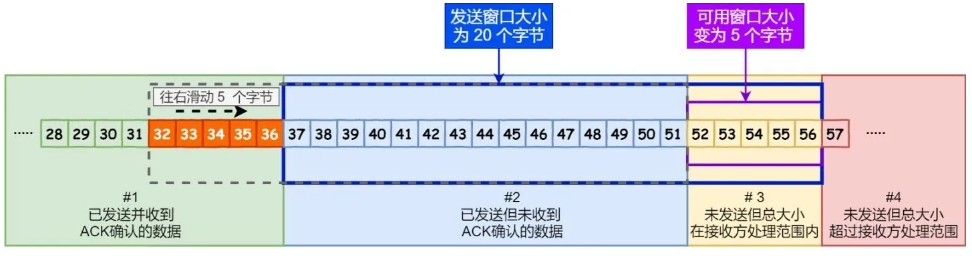


* 1. 流量控制

和 TCP 一样，QUIC 也是利用滑动窗口机制实现流量控制



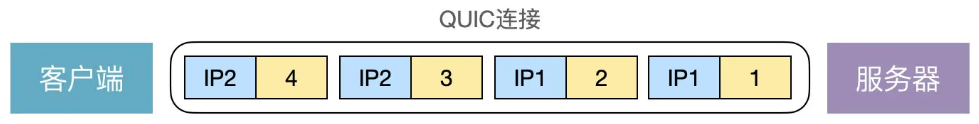
发送端的窗口大小由接收端告知，包括发送窗口和可用窗口，如果发送端收到了接收端的 ACK 确认应答（比如 ACK 36），那整个窗口就会向右滑动，发送新的数据包。



和 TCP 不同的是，QUIC 的滑动窗口分为 Connection 和 Stream 两种级别。Connection 流量控制：规定了所有数据流的总窗口大小；Stream 流量控制：规定了每个流的窗口大小。

* 1. 连接迁移

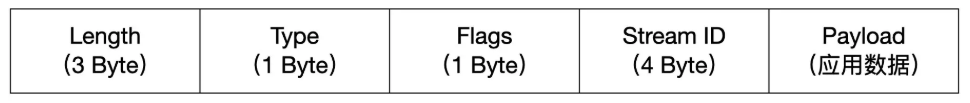
连接迁移：当客户端切换网络时，和服务器的连接并不会断开，仍然可以正常通信，对于 TCP 协议而言，这是不可能做到的。因为 TCP 的连接基于 4 元组：源 IP、源端口、目的 IP、目的端口，只要其中 1 个发生变化，就需要重新建立连接。但 QUIC 的连接是基于 64 位的 Connection ID，网络切换并不会影响 Connection ID 的变化，连接在逻辑上仍然是通的。



假设客户端先使用 IP1 发送了 1 和 2 数据包，之后切换网络，IP 变更为 IP2，发送了 3 和 4 数据包，服务器根据数据包头部的 Connection ID 字段可以判断这 4 个包是来自于同一个客户端。QUIC 能实现连接迁移的根本原因是底层使用 UDP 协议就是面向无连接的。

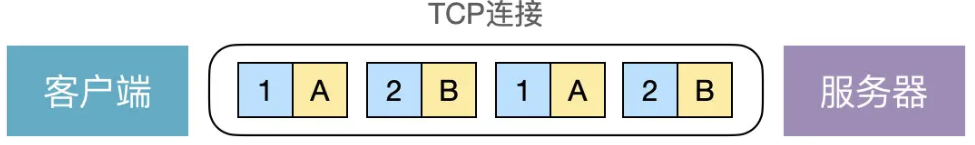
* 1. 多路复用

多路复用是指单条 TCP 连接上可以同时发送多个 HTTP 请求，解决了 HTTP1.1 中单个连接 1 次只能发送 1 个请求的性能瓶颈。HTTP/2 能实现多路复用的根本原因是采用了二进制帧格式的数据结构。

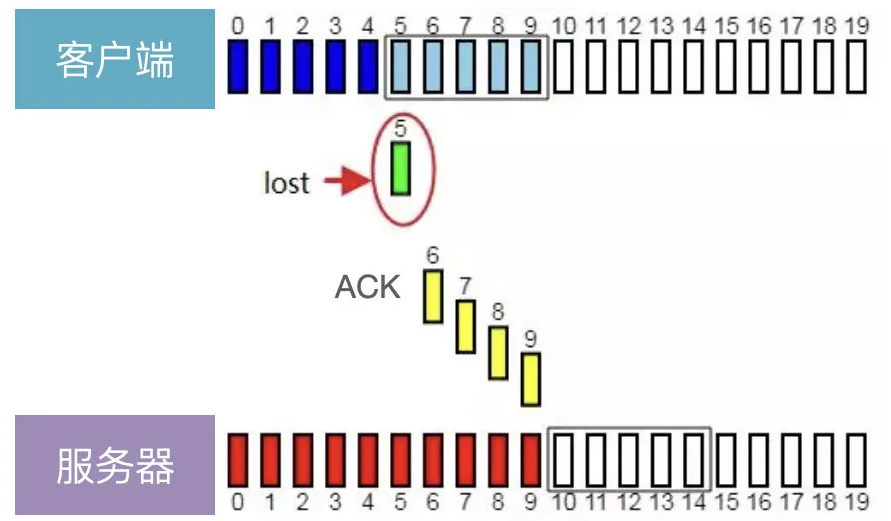


* Length：表示 Payload 的长度
* Type：表示帧类型
* Flags：帧标识
* Stream ID：数据帧所属的流
* Payload：应用数据，长度由 Length 字段指定

一个请求就对应一条流，通过 Stream ID 就可以判断该数据帧属于哪个请求，假设有 A 和 B 两个请求，对应的 Stream ID 分别为 1 和 2，那这个 TCP 连接上传输的数据大概如下：

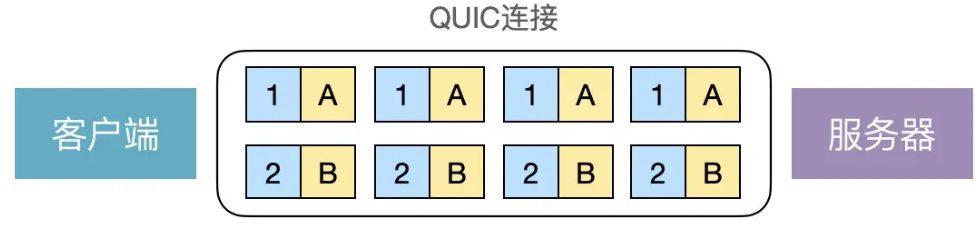


虽然在 HTTP 应用层，可以同时发送多个请求，但是在 TCP 传输层，仍然只有 1 个滑动窗口来发送这些数据包，考虑下面的情形：



客户端发送的 5 个数据包（56789）服务器都收到了，并且回应了 5 个 ACK，但是第 5 个数据包的 ACK 丢失了，导致客户端的发送窗口无法向前移动，也就无法发送新的数据，这就是 TCP 层的队头阻塞问题。

HTTP/2 虽然通过多路复用解决了 HTTP 层的队头阻塞，但仍然存在 TCP 层的队头阻塞。为解决这个问题， QUIC 给每个请求流都分配一个独立的滑动窗口，不再让所有请求流都共享一个滑动窗口。



A 请求流上的丢包不会影响 B 请求流上的数据发送。但是，对于每个请求流而言，也是存在队头阻塞问题的，也就是说，虽然 QUIC 解决了 TCP 层的队头阻塞，但仍然存在单条流上的队头阻塞。这就是 QUIC 声明的无队头阻塞的多路复用。

1. **迁移到muduo库**
2. 模块划分与架构设计
   1. 网络 I/O 模块适配

·**UDP Socket 支持**：在 Muduo 中扩展或新增一个 UDP 版的 Channel 类（例如 UDPSocketChannel），封装 UDP socket 的创建、读写、事件注册等操作。

·**事件注册与分发**：利用 Muduo 的 EventLoop 和 Poller 模块，实现 UDP socket 的事件注册（可读、可写、错误）并在事件触发时调用相应的回调函数。

* 1. 定时器与重传机制

· **TimerQueue 整合**：利用 Muduo 的 TimerQueue 模块，为 QUIC 协议实现超时重传、拥塞控制定时器和其它与时间相关的调度任务。

·**重传逻辑实现**：结合 QUIC 的包序号和 ACK 反馈，设计一个定时器驱动的重传机制，与 Muduo 的事件处理机制无缝集成。

* 1. 线程安全与并发模型

·**One Loop Per Thread 原则**：为每个 UDP 连接或 QUIC 会话分配对应的 EventLoop，实现线程内独占操作，确保并发安全。

·**业务线程分离**：将 QUIC 协议的包处理与业务逻辑分离，利用 Muduo 的线程池处理复杂数据处理任务，确保 I/O 线程不会被阻塞。

* 1. 数据加密与协议状态机

·**加密解密模块调用**：QUIC 协议内置 TLS 1.3 加密，可直接调用现有 QUIC 库的加密模块，不需要在 Muduo 内重新实现；重点是如何将加密后的数据包与 Muduo 的数据处理流程对接。

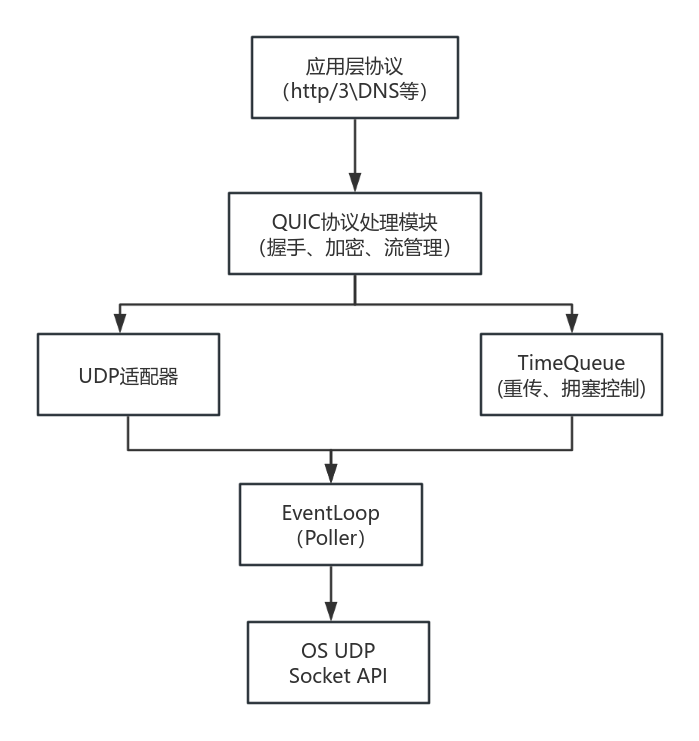
·**连接状态管理**：设计一个状态机模块，将 QUIC 的连接状态（握手、数据传输、关闭）与 Muduo 中的连接管理逻辑结合，实现状态转换的事件驱动处理。

* 1. 接口适配与数据流管理

·**抽象层设计**：定义一组适配接口，将 QUIC 库的内部接口适配到 Muduo 框架中，如数据包的解析、序列号管理、ACK 处理、流管理等。

·**多路复用处理**：在 QUIC 数据流中，需支持多流数据独立处理，可考虑在 Muduo 的事件循环中为每个 QUIC 流创建独立的回调和缓冲区，确保不同流间互不干扰。

1. 移植方案总体架构



1. 具体实现步骤
   1. 扩展 Muduo 的 Socket 抽象

在 Muduo 中新增 UDP 支持，封装 UDP Socket（例如 UDPSocket 类），并基于此实现一个 UDPSocketChannel 类，将 UDP 的读写和事件注册包装为与 TCP Channel 类似的接口。

* 1. 构建 QUIC 数据包处理模块

将现有 QUIC 库（例如 quiche、ngtcp2 等）的数据包解析、加密、状态机逻辑封装为独立模块，通过适配接口与 Muduo 的数据输入输出对接。

* 1. 整合 EventLoop 与 TimerQueue

在 QUIC 模块中添加对 Muduo TimerQueue 的调用，实现 QUIC 包重传、超时检测以及拥塞控制定时器的调度。

* 1. 设计连接管理器

实现一个基于 Muduo 连接模型的 QUIC 连接管理器，将 QUIC 的连接状态机嵌入到每个 UDP 连接中，并利用 Muduo 的线程模型（one loop per thread）来确保线程安全。

* 1. 接口适配与回调机制

定义并实现适配层，使 QUIC 模块的事件（如新数据包到达、ACK 收到、握手完成）触发 Muduo 的事件回调，反之亦然。

* 1. 测试与性能调优

编写单元测试与集成测试，验证 QUIC 模块在 Muduo 框架下的正确性和性能表现，并根据测试结果进行性能优化，如缓冲区管理、事件处理延迟等。