Hertz

RESTful

- 设计一个RESTful 规则

gin

- gin 源码阅读(1) - gin 与 net/http 的关系
- gin 源码阅读(2) - http请求是如何流入gin的?
- gin 源码阅读(3) - gin 路由的实现剖析
- gin 源码阅读(4) - 友好的请求参数处理
- gin 源码阅读(5) - 灵活的返回值处理

fasthttp

- fasthttp：比net/http快十倍的Go框架(server 篇)
- Go的fasthttp快的秘诀：简单事情做到极致

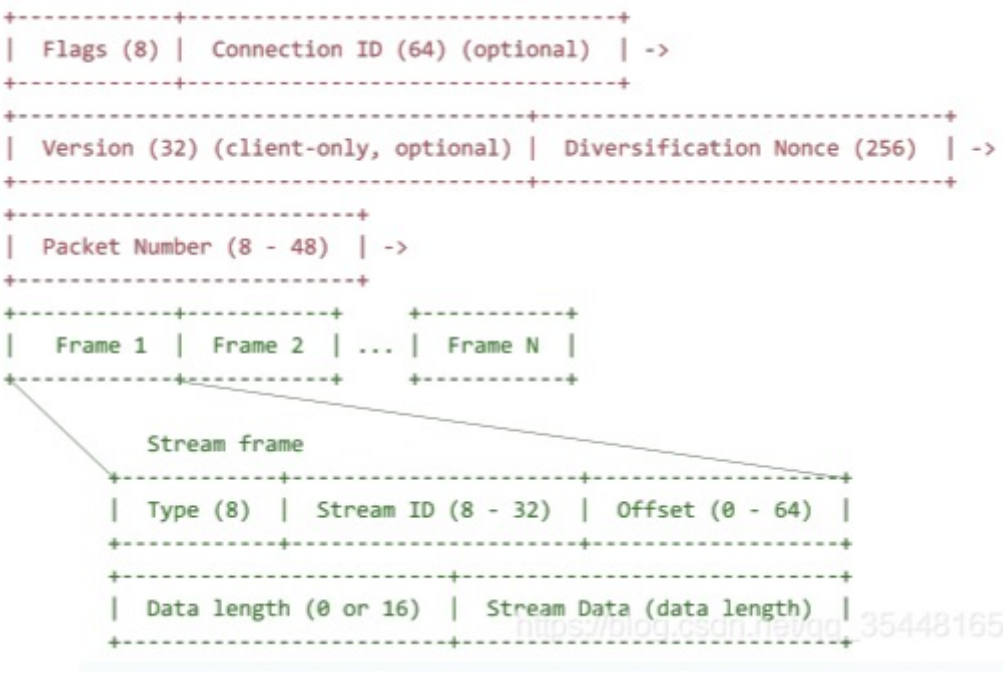
Hertz

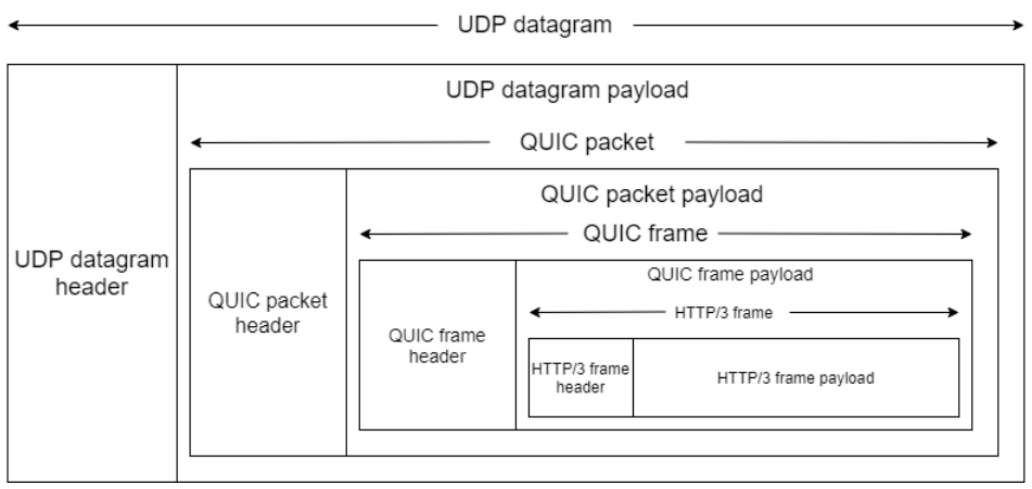
1. 为什么要从net/http到gin?
    - net/http的路由匹配规则过于简单，匹配到对应路由就返回对应handler，不匹配到时候就返回/，不符合RESTful规则
    - gin重写了route
2. 为什么从net/http到fasthttp?
    - net/http：goroutine-per-connection
    - fasthttp：单Reactor多goroutine，复用，使用byte[]代替string
        - 链接复用：
        - 内存复用：大量使用了sync.Pool
3. 为什么从Gin到Hertz?
    - Gin提供了良好的协议扩展能力：Engine中绕回进行协议选择
4. Hertz梳理：
    - Engine串通整条链路
    - 传输层：
        - connection接口：连接、读、写、Buffer(减少磁盘I/O读写)
        - transport接口：ListenAndServe()
    - 协议层：
5. 和Hertz的两个问题
    - 传输层接口，quic没法实现，也不符合预期
    - quic协议conn和stream概念和tcp的conn概念不统一
6. 解决方案

- onData选择不同回调

报文格式

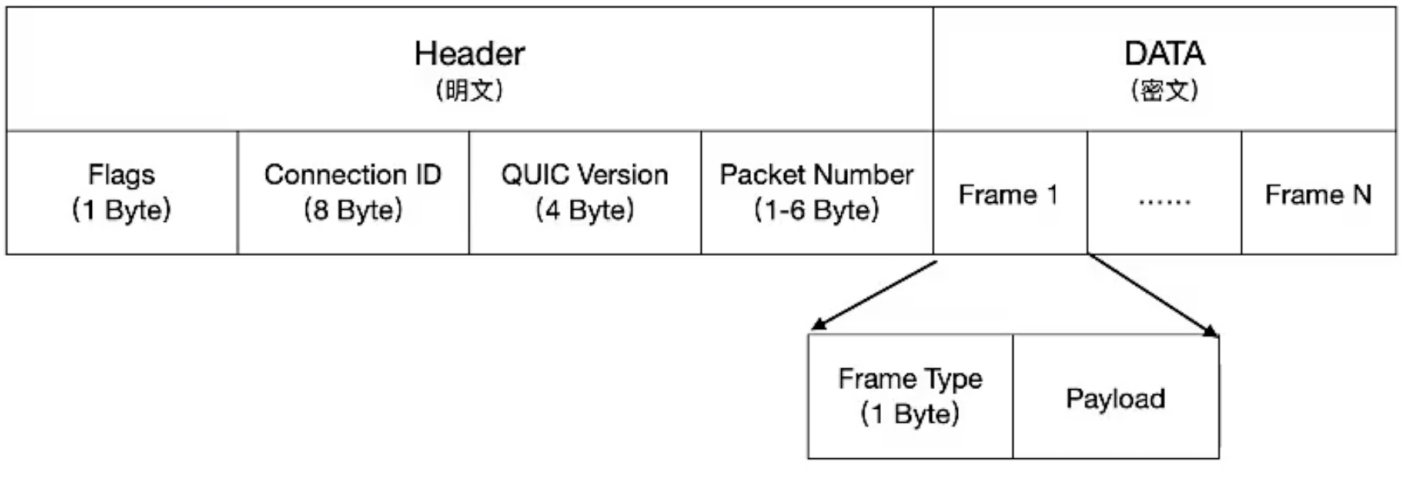
- 总览：





QUIC packet

- QUIC packet = Header + Data
- Long Packet Header 用于首次建立连接。
- Short Packet Header 用于日常传输数据。



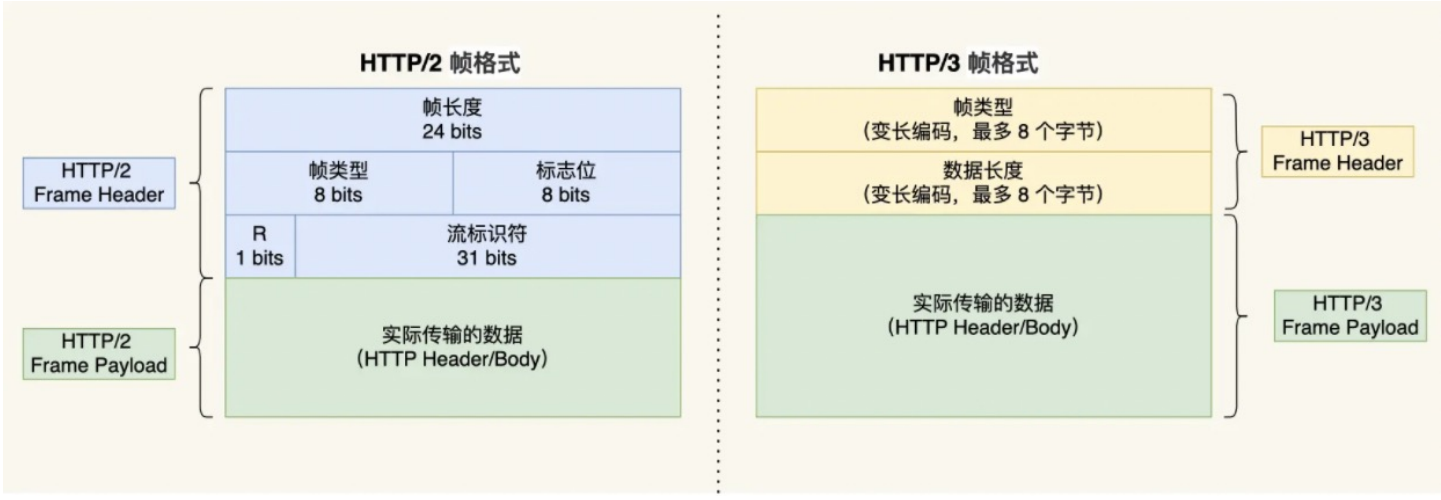
Stream Frame

- Data里面很多Frame：Stream、ACK、Padding、Blocked
- 主要看Stream Frame（传输应用数据）：

<table>
<tr><td colspan="5">Stream Frame
（数据流帧）</td></tr>
<tr><td rowspan="2">Frame Type
(1 Byte)</td><td colspan="4">Payload</td></tr>
<tr><td>Stream ID
(1-4 Byte)</td><td>offset
(0-8 Byte)</td><td>Data Length
(2 Byte)</td><td>Data
（应用数据）</td></tr>
</table>

HTTP/3 Frame

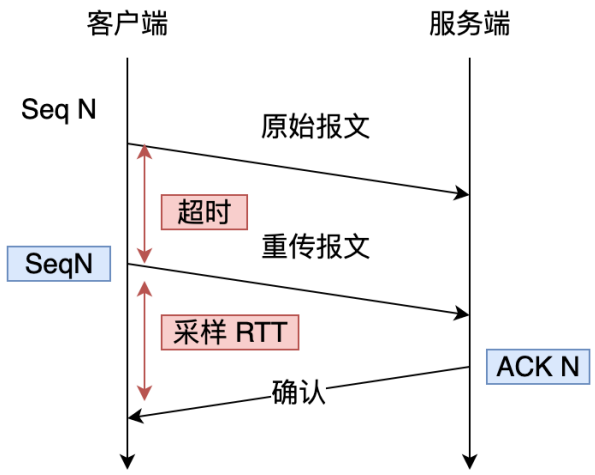
- Stream Data里面是HTTP3 Frame

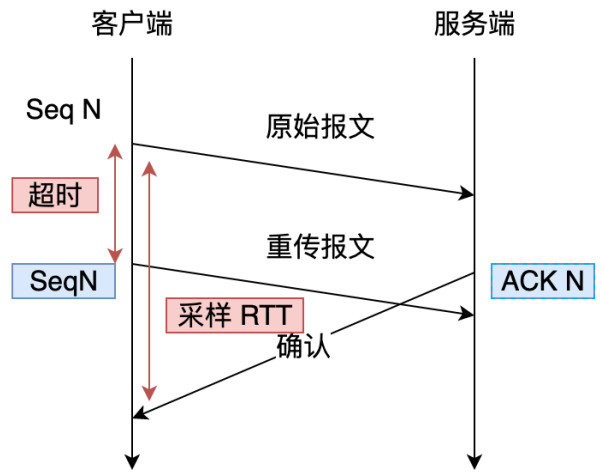


quic协议

- https://blog.csdn.net/wolfGuiDao/article/details/108729560
- https://www.cnblogs.com/lfri/p/quic.html
- TCP缺陷
  - 建立连接延迟
  - TCP层面的队头阻塞问题
  - 网络迁移需要重新建立连接

1. 可靠传输：
   a. Packet Number，配合 Stream ID 与 Offsetm，保证可靠
   b. Packet Number
      - QUIC packet里面的Packet Number
      - **严格单调递增**
      - 为什么单调递增：
        - TCP中有ACK歧义问题；下图2中我收到的是第一次还是第二次的ACK?
        - RTO（超时时间）根据RTT进行估算，上述歧义问题会导致RTT计算不准确





- 因此使用单调递增的Packet Number

c. Stream Frame

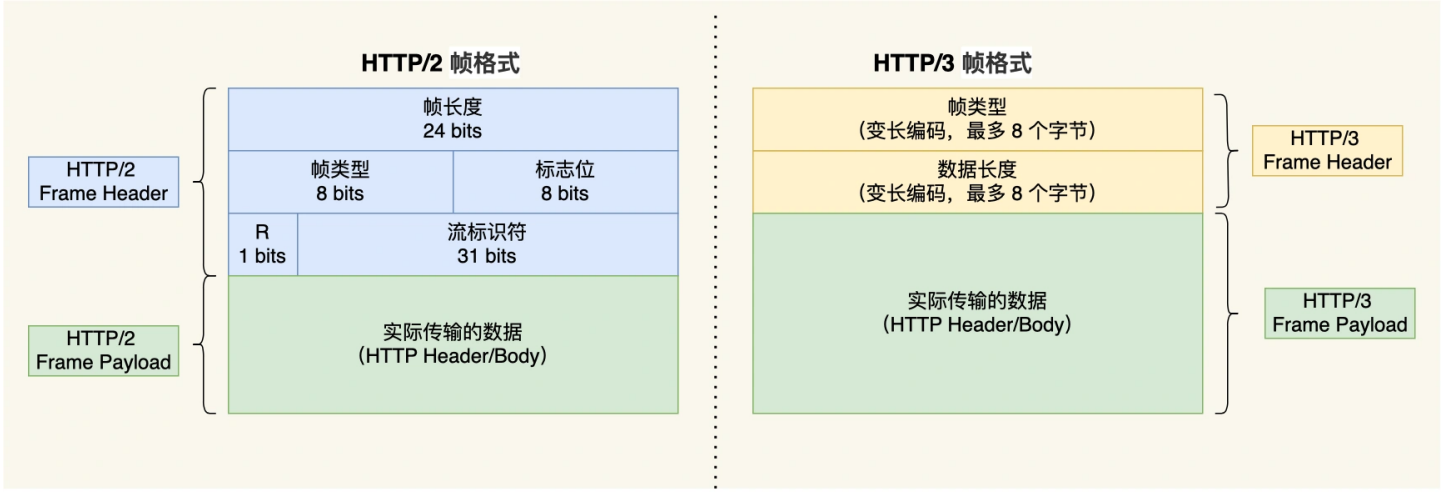
- Stream ID标识属于哪个Stream
- Offset表示数据迁移
- 丢失的数据包和重传的数据包 Stream ID 与 Offset 都一致，说明这两个数据包的内容一致。

2. 无队头阻塞的多路复用
   - HTTP/1：请求响应模型
   - HTTP/2：多个Stream之间依赖一个TCP连接，TCP需要之前的全部确认才能移动
   - QUIC：给每个Stream分配了一个独立的滑动窗口，多个Stream之间没有互相依赖
3. 低延迟建立连接
   - QUIC内部包含TLS1.3
     - client提供自身支持的client拖选曲线类型，计算出相应公钥，发给server
     - server选择相应的椭圆曲线参数，计算自身公钥，并且取出client公钥作为计算参数，计算出主密钥，发给client
     - client取出server公钥，计算出主密钥
   - 1RTT建立连接
   - 0RTT传输数据
     - 在client hello的时候写到数据（server自行相信或者拒绝数据）
4. 连接迁移connection id
   - 取消四元组
   - 使用connection id
5. 流量控制
   - Stream 级别的流量控制
   - Connection 流量控制：各个 Stream 接收窗口大小之和
6. 改进的拥塞控制
   - QUIC处于应用层，QUIC 可以随浏览器更新，QUIC 的拥塞控制算法就可以有较快的迭代速度，可以针对不同的应用设置不同的拥塞控制算法
   - TCP处于内核层，拥塞控制算法迭代速度的很慢

HTTP/3协议

- QPACK

- QPACK也是一种霍夫曼编码，按照内容出现次数进行编码，次数越多的内容，编码后，占用字节数就越少
- 采用了静态表、动态表及 Huffman 编码



quic-go

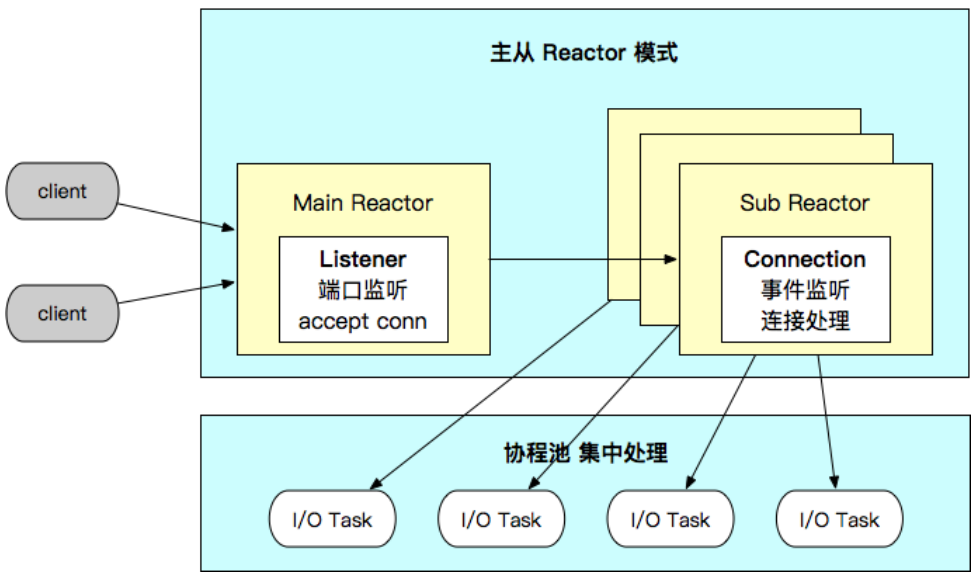
- https://github.com/lucas-clemente/quic-go/tree/master/http3

netpoll

- 字节跳动在 Go 网络库上的实践
- 一些点：

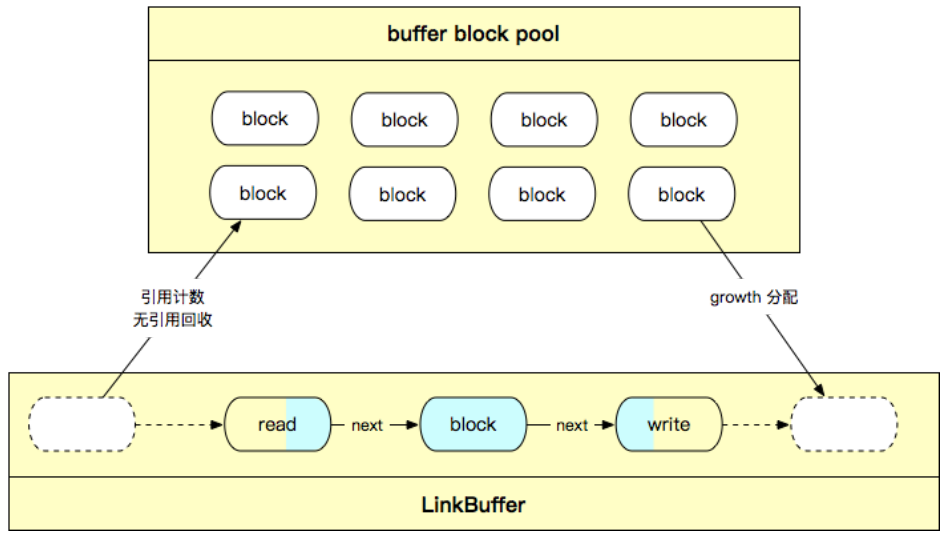
a. Multi-Reactor在muduo和netpoll上的异同

- muduo：subReactor，one loop per thread
- netpoll：subReactor配合一堆协程池
  - 协程池会导致问题：多个Reactor挂到一个P上，成了串行执行；本身goroutine有调度问题，会有延迟；
  - 为什么net库没问题：Go 在 runtime 中对于 net 库有做特殊优化，net库本身做了goroutine-per-connection优势
  - 如何解决协程池的问题：修改 Go runtime 源码



b. Buffer设计

- 问题：I/O 读数据写 buffer 和上层代码读 buffer 存在并发读写，所以搞了一层**copy-buffer**给业务层代码使用
- muduo buffer为什么是并发安全的：因为one loop per thread
- netpoll做法：基于链表数组实现，链表拼接



Kitex

- 字节跳动 Go RPC 框架 KiteX 性能优化实践
- Kitex 框架入门系列（1）
- Kitex 框架入门系列（2）Middleware

既然有 HTTP 协议，为什么还要有 RPC?

- 向系统**外部**暴露采用**HTTP**，向系统**内部**暴露调用采用**RPC**方式
- **前后端之间**（网关之前）用**HTTP**，**各个服**务之间用**RPC**
- HTTP 也是 RPC 的一种实现
- 最主要的原因还是RPC框架包含了重试机制，路由策略，负载均衡策略，高可用策略，流量控制策略等等

- RPC：
  - 传输效率高，定制化协议，性能消耗低
  - 可以使用HTTP/2
  - 基于thrift实现高性能传输
  - 可以基于HTTP协议，**也可以基于TCP协议**
  - 自带负载均衡、服务治理
- HTTP：
  - 超文本协议，头部臃肿
  - 大部分基础HTTP1.1，很多没用的内容
  - 基于json实现
  - 基于HTTP协议（相当于中间又加了一层，RPC直接通过TCP协议）
  - 需要配置Nginx，HAProxy等等
- 为什么要自研RPC框架？HTTP和RPC的区别