**[Fh's Blog](http://fhstack.github.io/)**

已识乾坤大，犹怜草木青

[首页](http://fhstack.github.io/) [工具](http://fhstack.github.io/tools/" \o "工具) [归档](http://fhstack.github.io/archives/" \o "归档) [关于](http://fhstack.github.io/about/" \o "关于)

## 文章目录

* [2. 队头阻塞问题](https://fhanddx.top/post/2020-06-02-quic/" \l "2-%E9%98%9F%E5%A4%B4%E9%98%BB%E5%A1%9E%E9%97%AE%E9%A2%98)
* [TCP改造困难](https://fhanddx.top/post/2020-06-02-quic/" \l "tcp%E6%94%B9%E9%80%A0%E5%9B%B0%E9%9A%BE)
* [建立连接握手](https://fhanddx.top/post/2020-06-02-quic/" \l "%E5%BB%BA%E7%AB%8B%E8%BF%9E%E6%8E%A5%E6%8F%A1%E6%89%8B)
* [版本协商](https://fhanddx.top/post/2020-06-02-quic/" \l "%E7%89%88%E6%9C%AC%E5%8D%8F%E5%95%86)
* [流式多路复用](https://fhanddx.top/post/2020-06-02-quic/" \l "%E6%B5%81%E5%BC%8F%E5%A4%9A%E8%B7%AF%E5%A4%8D%E7%94%A8)
* [丢包重传](https://fhanddx.top/post/2020-06-02-quic/" \l "%E4%B8%A2%E5%8C%85%E9%87%8D%E4%BC%A0)
* [流量控制](https://fhanddx.top/post/2020-06-02-quic/" \l "%E6%B5%81%E9%87%8F%E6%8E%A7%E5%88%B6)
* [拥塞控制](https://fhanddx.top/post/2020-06-02-quic/" \l "%E6%8B%A5%E5%A1%9E%E6%8E%A7%E5%88%B6)

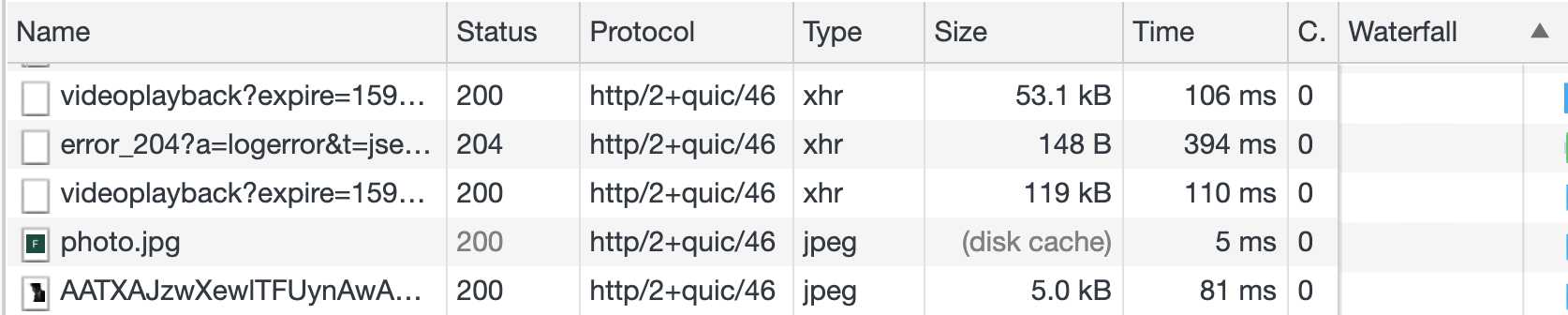
# **QUIC**

2020年6月2日

| [网络](https://fhanddx.top/categories/%E7%BD%91%E7%BB%9C)

|16 阅读

# 背景与QUIC现状

最近因为毕设需要，详细研究了一下Google的QUIC。  
我主要看的是Google官方团队的一篇2017年的[论文](https://static.googleusercontent.com/media/research.google.com/zh-CN//pubs/archive/46403.pdf" \o "论文)与初版的QUIC规范[文档](https://docs.google.com/document/d/1WJvyZflAO2pq77yOLbp9NsGjC1CHetAXV8I0fQe-B_U/edit" \l "heading=h.o9jvitkc5d2g" \o "文档)。根绝Google自己的数据，Google有30%的对外流量已经在使用QUIC，因此根据Google的用户量，可以估计互联网上有至少7%的流量在走QUIC协议，显然是不小的比例了。像油管这样的流媒体应用客户端很早就改造成使用QUIC和服务端通信了，随便打开一个油管的视频，可以看到都是的QUIC协议：  
[](http://fhanddx.top/wp-content/uploads/2020/05/wp_editor_md_65e6cbf57e7a2b68f8bd1d4fafcf70a3.jpg)  
根据Google自己的说法，改用QUIC后，油管桌面端和移动端的播放失败率至少分别降低了18.0%和15.3%，可见QUIC的性能是值得肯定的。  
接下来的部分对QUIC协议进行一个整体上的介绍，核心在于围绕两个问题

1. QUIC解决了什么问题？
2. QUIC怎么解决的？

# WHY QUIC？

近年来随着用户量剧增，无论是网页Web应用还是移动端APP，对延迟的要求显然都越来越高。  
网页和移动端APP基本上与服务端都走的HTTPS协议。HTTPS协议栈基于TLS/TCP，因此无论HTTP协议怎么变，如HTTP1.1、HTTP2这些再怎么优化或者调整，都无法摆脱任何TLS/TCP本身存在的一些问题。从协议栈整体来看，目前基于TLS/TCP的HTTPS协议栈问题主要以下方面：

在不考虑TCP的[Fast Open](https://tools.ietf.org/html/rfc7413" \o "Fast Open){.wp-editor-md-post-content-link}的情况下，客户端与服务端开始通信至少需要3\_RTT的延迟，具体来说：三次握手客户端等待至少1.5\_RTT、TLS等待2\*RTT。在一些短连接业务场景下，例如交易，这个开销很划不来。

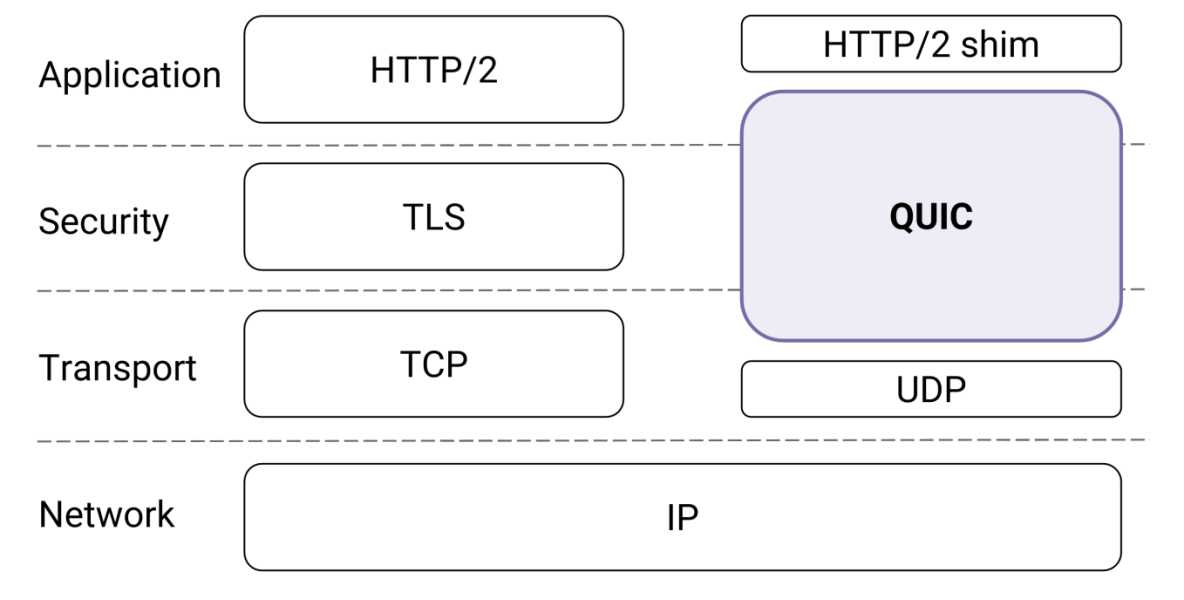
## 2. 队头阻塞问题

先说说关于队头阻塞问题的背景。  
早期为了提高HTTP在一条TCP连接上的信道利用率，在HTTP1.1加入了pipeline机制，多个request可以连续发送而不用等待response返回，但是对应的response必须按request的发送顺序返回  
  
原因也很简单，因为无序返回的话就不知道每个response对应的是哪个request。因此当某个处于中间顺序的request所对应的那次请求在服务端处理时阻塞住或者发生丢包时，后面的response就都受影响被阻塞了。  
故为了解决这种队头阻塞（Head-of-line Blocking），SPDY与之后的HTTP2都通过二进制分帧给各个request和对应的response增加标识使response不需要按顺序返回，客户端可以通过标识将返回的response重新组装，这在一定程度上解决了对头阻塞。然而，只要底层使用的是TCP协议，即使HTTP2已经抽象出多条HTTP层面可以并行的stream来传输多个不同的request与response，在TCP的这条全双工字节流通信连路上就必定还是可能会存在队头阻塞。  
  
如stream3、4仍然会因为stream2的丢包而阻塞，从本质上来说就是因为只要TCP底层不是并行的流，这种情况仍然是无奈的。

## TCP改造困难

为什么要新开发一个QUIC，而不改造TCP？  
根本原因在于改造TCP成本太大了。RFC793在1981年被制定出来后，可以说无一例外的说，操作系统都将TCP协议栈实现在了内核里。因此对TCP的任何改造想要得到部署，都需要跟随操作系统的版本升级，这种系统级别的漫长迭代周期与部署成本使得所有TCP进行长期改造与升级变得十分困难，的确还不如重新开发一个新协议来的方便。

# 为什么QUIC要基于UDP

[](http://fhanddx.top/wp-content/uploads/2020/06/wp_editor_md_3dd4d3d5399625186a98dd7de3a5215c.jpg)一个新的协议被提出，要做两端部署是很容易的事情，因为我们的客户端和服务端是自己能控制的，但是两端路径上的中间盒（[middlebox](https://tools.ietf.org/html/rfc3234" \o "middlebox))是不可能被我们所控制的。就互联网发展到很成熟的今天来说，我们已经不太可能要求所有ISP使用一个新提出的标准来实现路由器或增加对新协议支持。因此，假如QUIC从IP协议上建立一个拥有自己运输层新协议号的的协议，一个最直接的问题是，很多NAT是不认识QUIC协议的…  
不得不说使用UDP来开发新的协议是很不错的事情，因为UDP足够轻量，简单的8字节报头，使得基于开发新协议变得非常可行。其次正如上面说的，用UDP包装一层就不受中间盒的限制了，毕竟任何设备都支持UDP。

# QUIC核心原理

关于QUIC的实现，目前Google已经交给IETF组织专门的一个工作组在进行相关完善与标准化工作，本文的QUIC是Google的QUIC35版本中的设计与实现，与现在IETF正在制定的可能有一些细节上的不同，不过核心是一直的，关于QUIC的最新工作有兴趣的话参考[IETF QUIC working group](https://datatracker.ietf.org/wg/quic/" \o "IETF QUIC working group)。  
我们会避免在一些原理上过多陷入细节，更具体的还是应该参考相关文档。

## 建立连接握手

QUIC在建立连接过程中，将握手和加密协商组合在了一起，因此QUIC的建立连接比TCP会复杂很多，其中加密会涉及Diffie-Hellman算法。  
根据个人需要，如果想把QUIC的握手弄得非常清楚的话，建议参考[这篇](https://blog.csdn.net/chuanglan/article/details/85106706" \o "这篇)了解明白Diffie-Hellman的基本原理，以及QUIC在握手过程中如何实现密钥的生成。  
关于QUIC的建立连接，对照上图，我们简化后大致过程如下：

### 一、初次连接

1.客户端发送早期CHLO(client hello)报文，不加密  
2.服务端回应一个REJ(reject)报文，不加密  
3.客户端发送一个完整CHLO报文，不加密  
到目前为止，客户端可以开始和服务端进行加密通信  
4.服务端回应一个SHLO报文告诉客户端连接建立，不加密  
到目前为止，服务端可以开始和客户端进行加密通信  
密钥通过前三步就已经可以计算了，因此可以认为QUIC客户端的握手延迟只有差不多1\_RTT，服务端1.5\_RTT。

### 二、重复连接

1. 客户端发送一个完整CHLO报文，不加密  
   到目前为止，客户端可以开始和服务端进行加密通信
2. 服务端回应一个SHLO报文告诉客户端连接建立，不加密  
   到目前为止，服务端可以开始和客户端进行加密通信  
   重复建立连接时握手延迟是0RTT，因为客户端缓存了之前交换过的密钥值。

关于重复连接还有一点需要强调的是，QUIC连接核心是密钥，每条QUIC连接以Connection ID（参考下文QUIC报头）标识，而不是IP:Port五元组。因此QUIC可在客户端缓存过与特定服务端协商好的密钥值的情况下，实现跨IP地址0RTT建立连接，这一点TCP是不可能做到的，且应用前景非常不错，比如手游从wifi切换到4G无缝衔接不卡顿

### 三、重复连接失败

重复连接必然会有失败的情况，例如服务端密钥相关的证书或者票据等等，这是如上面那张图最右边的情况

1. 客户端发送完CHLO想直接开始通信
2. 客户端开始发送加密消息
3. 服务端回应一个REJ报文表示连接无法建立  
   接下里来客户端和服务端就重复初始连接的握手过程，以重新建立连接。  
   可见QUIC的建立连接不仅直接把加密直接做了，并且延迟的考虑上非常好。

## 版本协商

QUIC在建立连接的时候，会通过版本协商来尽量使用双方都支持的最新版本。客户端可以在CHLO包中将自己选用的最新版本号发过去，若服务端支持则继续走握手流程，若是不支持，则服务端返回自己所有支持的版本，客户端重新选择一个支持的版本再走握手流程。

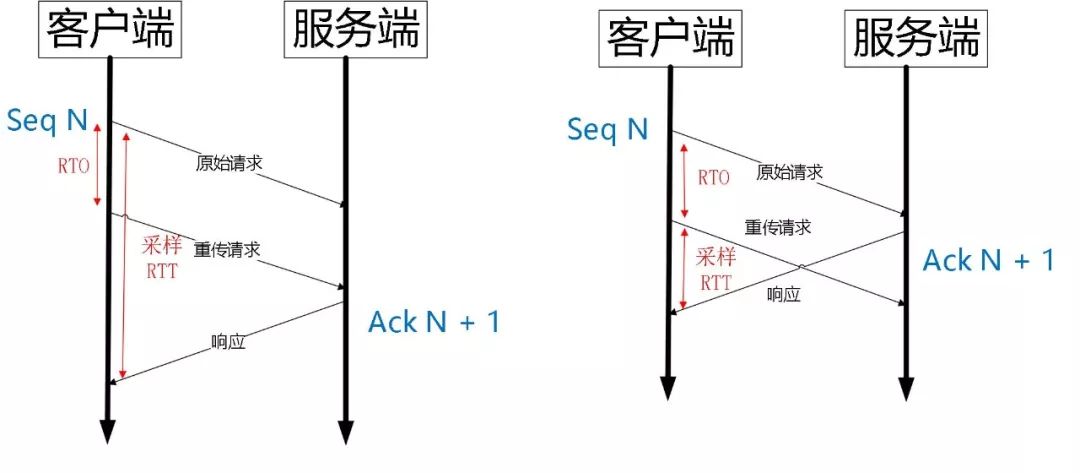
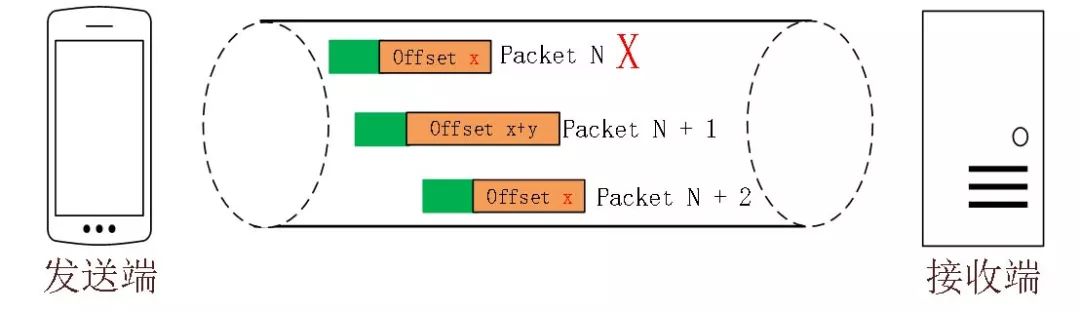
## 流式多路复用

QUIC在传输层面实现了真正的流式多路复用，QUIC的信道上不是TCP那种一条全双工通信管道，而是多条轻量的流(stream)，流是一个轻量级的抽象，在流上可以进行双向的可靠字节流传输。  
  
每条QUIC连接以Connection ID标识，单个QUIC报文可以包含多个不同的流帧(Frame)，流用Stream ID标识。  
在QUIC连接建立后，上方都可以主动建立新的流，因此为了避免冲突，客户端建立的流的Stream ID都使用奇数，服务端使用偶数。  
由于每条流直接的传输是隔离的，这样的一个显著好处是，即使发生UDP丢包，也只会影响相应QUIC报文中包含的流帧所在的流，不会阻塞任何其他不相关的流，因此这就算是彻底解决了基于TCP的HTTP2 stream的队头阻塞问题。

## 丢包重传

QUIC在丢包重传上相比起TCP主要有两大进步

### 一、无重传歧义

TCP用序号来标识每一个包，事实上TCP的序号也标识着TCP报文中的数据在字节流中的顺序，因此发送端认为发生了丢包，对报文进行重传时，可能报文没有丢失导致厚后续收到两个ACK报文，这个在TCP中导致重传歧义问题，因为不知道这两个ACK报文的ACK号是相等的，导致无法确定收到的这两个ACK包与发送的包是如何对应的的。重传歧义的最直接影响是导致无法对信道的RTT进行准确估计，从而影响拥塞控制。  
[](https://pic2.zhimg.com/80/v2-8db4c3c378edaac0060b4238e3554091_1440w.jpg)  
QUIC就完全没有这个问题，因为QUIC把Packet Number与标识数据在字节流中顺序的Offset分为了两个字段，从QUIC报文中可以看到，每个QUIC报文具有一个包序号，这个包序号是严格递增的，这个包序号只代表时序，与数据无关，每条流帧自己有Offset字段用于标识别数据的顺序。  
[](https://pic2.zhimg.com/80/v2-60985053d9de4e8e74042c33587ec35d_1440w.jpg)  
如发送端认为N号包丢失时，在N+2号保重传对应流帧的数据，接收端仍然可以使用Offset来正确拼接数据。

### 二、更高的ACK效率

TCP即使有SACK的情况下，由于TCP报文中扩展字段最大只有40字节，时间戳占用了10个字节，一个SACK块占用8字节，因此单个报文中也最多SACK三个块。  
QUIC中有专门的ACK类型的流帧，一个帧中最大可以携带256个数据块。因此QUIC的ACK效率是远远超过TCP，这一点可以节省很多信道与带宽资源。

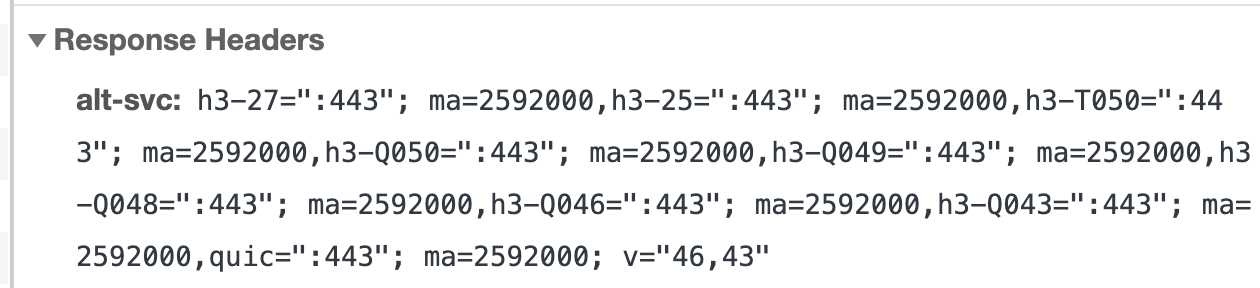
## 流量控制

QUIC的流量从连接级别与流级别同时进行，限制单条流的接收缓冲区与连接级别的缓冲区，连接级别的缓冲区就是连接的所有流接收缓冲区的总和。

## 拥塞控制

QUIC在拥塞控制方面的进步是其拥塞控制算法是可插拔的，拥塞控制算法可插拔的好处非常多。  
1.任何拥塞控制算法的切换和部署成本都非常小，用户态不受内核版本限制  
2.可以做到单机不同连接使用不同的拥塞控制算法，例如某些服务使用BBR、某些服务使用Cubic，这在TCP中是不可想象的事情

# QUIC与HTTP2/3

QUIC最初就是为了HTTP2协议所设计，然而考虑到QUIC尚未全面普及，因此并未所有HTTP2都可以走QUIC来实现，因此目前规定首先还是先通过TCP/TLS来进行首次HTTP通信，服务端在response中通过alt-svc字段来告知客户端可以升级为使用QUIC  
[](http://fhanddx.top/wp-content/uploads/2020/06/wp_editor_md_6bd4f028bea77c05ccba27ba1b7ec8c4.jpg)  
Google目前对QUIC的的应用主要都是用于作为HTTP2的底层实现，IETF工作组正在将QUIC划分为HTTP实现部分与传输两个部分，让QUIC直接应用于其他应用数据的传输成为标准的一部分，目前IETF组织已经确定处于酝酿中的HTTP/3完全通过QUIC来进行实现，即HTTP-over-QUIC。

# 开源实现

总的来说目前QUIC开源实现里基于go的比较活跃

1. Chrome  
   Google自家的chrome浏览器源码中当然有完整的QUIC的C++实现，不过应该是是有客户端（没有具体看过？）
2. [caddy](https://github.com/caddyserver/caddy" \o "caddy)（非常活跃）  
   caddy是一个go实现的server，常用作web server来用，也可以用于任何应用的server，目前完整的实现了QUIC协议栈
3. [quic-go](https://github.com/lucas-clemente/quic-go" \o "quic-go)（非常活跃）  
   也是go实现，比较粗糙的完整实现了目前IETF的QUIC草案  
   4.[libquic](https://github.com/devsisters/libquic)（将近四年为更新过了）  
   C++实现，感觉现在没有维护了…

# 总结

综合来看QUIC假如作为未来的新可靠传输协议，其优秀无可置疑。不过网络发展至今，任何新协议的推广都是一件很不容易的事情，不知道十年后QUIC能否真正的取代TCP。

* ****原文作者：****[Fh](http://fhstack.github.io/)
* ****原文链接：****<http://fhstack.github.io/post/2020-06-02-quic/>
* ****版权声明：****本作品采用[知识共享署名-非商业性使用-禁止演绎 4.0 国际许可协议](https://creativecommons.org/licenses/by-nc-nd/4.0/)进行许可，非商业转载请注明出处（作者，原文链接），商业转载请联系作者获得授权。

## **See Also**

* [Redis源码剖析——dict（二）](https://fhanddx.top/post/2019-06-17-redisyuanmapouxidicter/)
* [Redis源码剖析——dict（一）](https://fhanddx.top/post/2019-06-14-redisyuanmapouxidict/)
* [Redis源码剖析——intset](https://fhanddx.top/post/2019-06-11-redisyuanmapouxiintset/)
* [Redis源码剖析——HyperLogLog](https://fhanddx.top/post/2019-05-12-redisyuanmapouxihyperloglog/)
* [Redis源码剖析——跳表（三）](https://fhanddx.top/post/2019-05-08-redisyuanmapouxitiaobiaosan/)

没有标签

© 2021 [Fh's Blog By Fh](http://fhstack.github.io/)

[Powered byHugo](https://gohugo.io/" \t "/Users/yangwenshuo/Documents\\x/_black)

[Design by飞雪无情](https://www.flysnow.org/" \t "/Users/yangwenshuo/Documents\\x/_black)

[ThemeMaupassant](https://github.com/flysnow-org/maupassant-hugo" \t "/Users/yangwenshuo/Documents\\x/_black)

窗体顶端

窗体底端

### 最近文章

* [sync.Map源码阅读笔记](http://fhstack.github.io/post/2020-06-27-sync-mapyuanmayuedubiji/" \o "sync.Map源码阅读笔记)
* [QUIC](http://fhstack.github.io/post/2020-06-02-quic/" \o "QUIC)
* [Redis源码剖析——dict（二）](http://fhstack.github.io/post/2019-06-17-redisyuanmapouxidicter/" \o "Redis源码剖析——dict（二）)
* [Redis源码剖析——dict（一）](http://fhstack.github.io/post/2019-06-14-redisyuanmapouxidict/" \o "Redis源码剖析——dict（一）)
* [Redis源码剖析——intset](http://fhstack.github.io/post/2019-06-11-redisyuanmapouxiintset/" \o "Redis源码剖析——intset)
* [Redis源码剖析——HyperLogLog](http://fhstack.github.io/post/2019-05-12-redisyuanmapouxihyperloglog/" \o "Redis源码剖析——HyperLogLog)
* [Redis源码剖析——跳表（三）](http://fhstack.github.io/post/2019-05-08-redisyuanmapouxitiaobiaosan/" \o "Redis源码剖析——跳表（三）)
* [Redis源码剖析——跳表（二）](http://fhstack.github.io/post/2019-05-05-redisyuanmapouxitiaobiaoer/" \o "Redis源码剖析——跳表（二）)
* [Redis源码剖析——跳表（一）](http://fhstack.github.io/post/2019-05-03-redisyuanmapouxitiaobiaoyi/" \o "Redis源码剖析——跳表（一）)
* [你还在这样写快排？](http://fhstack.github.io/post/2019-04-20-nihuanzaizheyangxiekuaipai/" \o "你还在这样写快排？)

### [分类](https://fhanddx.top/categories/)

* [C/C++ (1)](http://fhstack.github.io/categories/C/C++/)
* [Golang (6)](http://fhstack.github.io/categories/Golang/)
* [linux (3)](http://fhstack.github.io/categories/linux/)
* [杂记 (1)](http://fhstack.github.io/categories/%E6%9D%82%E8%AE%B0/)
* [源码阅读 (17)](http://fhstack.github.io/categories/%E6%BA%90%E7%A0%81%E9%98%85%E8%AF%BB/)
* [算法与数据结构 (2)](http://fhstack.github.io/categories/%E7%AE%97%E6%B3%95%E4%B8%8E%E6%95%B0%E6%8D%AE%E7%BB%93%E6%9E%84/)
* [网络 (1)](http://fhstack.github.io/categories/%E7%BD%91%E7%BB%9C/)

### [标签](https://fhanddx.top/tags/)

### 友情链接

* [我的Githhub](https://www.github.com/fhstack" \o "我的Github" \t "/Users/yangwenshuo/Documents\\x/_blank)

### 其它

* [文章 RSS](http://fhstack.github.io/index.xml)