

什么是quic协议？

QUIC（快速UDP互联网连接协议）是Google推出的创新性网络传输协议，旨在突破TCP协议的性能限制。该协议利用UDP实现高效、安全且低延迟的网络传输，现已被IETF确立为标准协议，并作为HTTP/3的基础传输层协议。

QUIC协议设计原理：

- 1) 采用 UDP 替代 TCP：通过绕过操作系统内核的 TCP 协议栈，有效规避队头阻塞问题，同时实现自主可控的可靠传输机制。
- 2) 零往返时连接（0-RTT）：通过首次连接缓存密钥信息，后续连接可跳过握手环节（类似TCP中TLS 1.3的0-RTT机制），显著降低30%-50%的延迟。

QUIC协议优势

特性	说明
多路复用无阻塞	数据流独立传输，单条数据流丢包不会影响其他流（避免了TCP的固有缺陷）。
原生加密	默认启用 TLS 1.3，实现报文头部及载荷的全链路强制加密。
智能拥塞控制	动态调整数据包发送速率（如采用 BBR 算法），以适配不同网络环。
连接迁移	设备在Wi-Fi和5G网络间切换时，能保持稳定连接，无需重新建立握手协议。

QUIC协议与TCP+TLS协议对比

对比项	TCP + TLS	QUIC
握手延迟	1-RTT (TLS 1.2+)	0-RTT (非首次连接)
队头阻塞	存在（单流阻塞）	无（多流独立）
加密范围	仅加密载荷	头部+载荷全加密
协议升级	需操作系统支持	用户态实现，快速迭代
抗丢包能力	依赖重传	前向纠错（FEC） 选项

quic-go介绍

quic-go 是一个基于 Go 语言的 QUIC 协议实现库，全面兼容以下核心标准：

- 1) RFC 9000（QUIC 基础协议）
- 2) RFC 9001（TLS 集成）
- 3) RFC 9002（拥塞控制）

该库采用 UDP 传输实现低延迟通信，并深度整合了 HTTP/3（RFC 9114）及其关键技术：

- 1) QPACK 头部压缩（RFC 9204）
- 2) HTTP 数据报（RFC 9297）

quic-go 在核心规范之外还提供以下增强功能：

- 1) 不可靠数据报扩展（RFC 9221）：支持非可靠传输模式，适用于实时音视频等场景。
- 2) 路径 MTU 发现（RFC 8899）：动态探测网络最大传输单元，优化数据分片。
- 3) QUIC 版本 2（RFC 9369）：兼容新版协议特性。
- 4) qlog 事件日志：遵循 IETF 草案标准记录连接事件，便于调试分析。
- 5) WebTransport 支持：通过 webtransport-go 子库实现基于 HTTP/3 的实时双向通信（草案 ietf-webtransport3）。

实现HTTP3服务端

首先创建QUIC配置对象，接着初始化TLS配置（将InsecureSkipVerify设置为true，系统将跳过证书验证环节）。然后构建HTTP/3服务对象，最后调用其ListenAndServeTLS方法启动服务监听，该方法需指定证书文件路径。代码示例如下：

```
1
2 func main() {
3     quicConfig := &quic.Config{}
4
5     tlsConfig := &tls.Config{
6         InsecureSkipVerify: true,
7     }
8
9     fmt.Println("监听地址：127.0.0.1, 端口号：8181")
10
11    server := &http3.Server{
12        Addr:      "127.0.0.1:8181",
13        Port:      8181,
14        TLSConfig: tlsConfig,
15        QUICConfig: quicConfig,
16        Handler:    http3Handler(),
17    }
18
19    err := server.ListenAndServeTLS("D://ca/server.crt", "D://ca/server.key")
20    if err != nil {
21        logrus.Errorf("ListenAndServeTLS failed , error: %v", err)
22        return
23    }
24 }
```

定义HTTP3服务的URL路由处理器http3Handler，并配置NotFoundHandler和MethodNotAllowedHandler分别处理无效URL和不支持的方法请求。代码示例如下：

```
1
2 func http3Handler() http.Handler {
3     router := mux.NewRouter()
4
5     // 获取用户信息
6     router.HandleFunc("/user/info", getUserInfo).Methods(http.MethodGet)
7
8     // 删除用户信息
9     router.HandleFunc("/user/info/{name}", delUserInfo).Methods(http.MethodDelete)
10
11    // 添加用户信息
12    router.HandleFunc("/user/info", addUserInfo).Methods(http.MethodPost)
13
14    // 未找到匹配到URL处理
15    router.NotFoundHandler = http.HandlerFunc(notFoundHandler)
16
17    // 未找到匹配到方法处理
18    router.MethodNotAllowedHandler = http.HandlerFunc(notMethodHandler)
19
20    return router
21 }
```

20
21

如上代码所示，实现了用户信息查询、删除及新增三个接口功能。具体代码示例如下：

```
1 // 获取用户信息
2 func getUserInfo(resp http.ResponseWriter, req *http.Request) {
3
4     fmt.Printf("获取用户信息接口调用, URL: %s, 客户端地址: %s\n", req.URL.Path, req.RemoteAddr)
5
6     sendResponseWithBody(resp, "", "获取用户信息")
7 }
8 // 添加用户信息
9 func addUserInfo(resp http.ResponseWriter, req *http.Request) {
10
11     fmt.Printf("添加用户信息接口调用, URL: %s, 客户端地址: %s\n", req.URL.Path, req.RemoteAddr)
12
13     body, err := io.ReadAll(req.Body)
14     if err != nil {
15         fmt.Printf("读取消息体失败, %v", err)
16         sendResponse(resp, http.StatusBadRequest, err.Error())
17         return
18     }
19
20     fmt.Printf("添加用户信息: %s\n", string(body))
21
22     sendResponse(resp, http.StatusOK, "")
23 }
24 // 删除用户信息
25 func delUserInfo(resp http.ResponseWriter, req *http.Request) {
26
27     fmt.Printf("删除用户信息接口调用, URL: %s, 客户端地址: %s\n", req.URL.Path, req.RemoteAddr)
28
29     vars := mux.Vars(req)
30
31     fmt.Printf("删除[%s]用户信息\n", vars["name"])
32
33     sendResponse(resp, http.StatusOK, "")
34 }
35
```

提供两种消息发送方式：支持附带消息体的完整发送和不带消息体的简化发送。，具体代码示例如下：

```
1 // 响应信息
2 func sendResponse(resp http.ResponseWriter, statusCode int, statusDesc string) {
3
4     resp.WriteHeader(statusCode)
5
6     if statusDesc == "" {
7         _, _ = io.WriteString(resp, statusDesc)
8     } else {
9         _, _ = io.WriteString(resp, http.StatusText(statusCode))
10    }
11 }
12 // 发送响应消息，携带消息体
13 func sendResponseWithBody(resp http.ResponseWriter, contentType, body string) {
```

```

12
13     if contentType != "" {
14         resp.Header().Set("Content-Type", contentType)
15     }
16
17     resp.Header().Set("Content-Length", strconv.Itoa(len(body)))
18
19     _, err := io.WriteString(resp, body)
20     if err != nil {
21         sendResponse(resp, http.StatusInternalServerError, err.Error())
22     }
23 }
24
25
26

```

当HTTP3服务端遇到无法识别的URL或HTTP请求方法时，会自动触发NotFoundHandler和MethodNotAllowedHandler方法，NotFoundHandler和MethodNotAllowedHandler具体实现代码如下：

```

1
2 // 未找到匹配到URL处理
3 func notFoundHandler(resp http.ResponseWriter, req *http.Request) {
4     fmt.Printf("unkown url: %s, method:%s, relay 404\n", req.URL, req.Method)
5
6     sendResponse(resp, http.StatusNotFound, "URL not found")
7 }
8 // 未找到匹配到方法处理
9 func notMethodHandler(resp http.ResponseWriter, req *http.Request) {
10    fmt.Printf("unkown method:%s, url: %s, relay 405\n", req.Method, req.URL)
11
12    sendResponse(resp, http.StatusMethodNotAllowed, "Method Not Allowed")
13 }
14
15

```

实现HTTP3客户端

首先创建QUIC配置对象，并初始化x509证书池。接着设置TLS配置，启用InsecureSkipVerify选项以跳过证书验证。完成这些基础配置后，创建RoundTripper并构建最终的HTTP/3客户端对象。代码示例如下：

```

1
2 func main() {
3
4     quicConfig := &quic.Config{}
5
6     pool, err := x509.SystemCertPool()
7     if err != nil {
8         return
9     }
10
11     tlsConfig := &tls.Config{
12         RootCAs:      pool,
13         InsecureSkipVerify: true,
14     }
15
16     roundTripper := &http3.RoundTripper{
17         TLSClientConfig: tlsConfig,
18         QuicConfig:      quicConfig,
19     }
20 }

```

```

14     }
15
16     hclient := &http.Client{
17         Transport: roundTripper,
18     }
19
20     getUserInfo(hclient)
21
22     addUserInfo(hclient, "李四")
23
24     deleteUserInfo(hclient, "张三")
25 }
26
27
28
29

```

如需使用证书，客户端需通过调用pool接口进行证书添加，上述示例演示的是无证书验证场景。代码示例如下：

```

1
2 caCertRaw, err := os.ReadFile("D://ca/ca.crt")
3 if err != nil {
4     return
5 }
6
7 if ok := pool.AppendCertsFromPEM(caCertRaw); !ok {
8     return
9 }
10

```

HTTP3实现用户信息的获取、新增与删除接口调用，代码示例如下：

```

1
2 // 获取用户信息
3 func getUserInfo(client *http.Client) {
4
5     fmt.Println("获取用户信息:")
6
7     request, err := http.NewRequest("GET", "https://127.0.0.1:8181/user/info", nil)
8     if err != nil {
9         return
10    }
11
12    resp, err := client.Do(request)
13    if err != nil {
14        return
15    }
16
17    body := &bytes.Buffer{}
18
19    _, err = io.Copy(body, resp.Body)
20
21    fmt.Printf("获取用户信息, 响应码: %d, 消息体: %s\n", resp.StatusCode, body)
22 }
23
24 // 添加用户信息
25 func addUserInfo(client *http.Client, name string) {
26
27     fmt.Printf("添加用户信息: %s", name)
28
29     request, err := http.NewRequest("POST", "https://127.0.0.1:8181/user/info", strings.NewReader(name))
30     if err != nil {
31         return
32     }
33
34 }
35

```

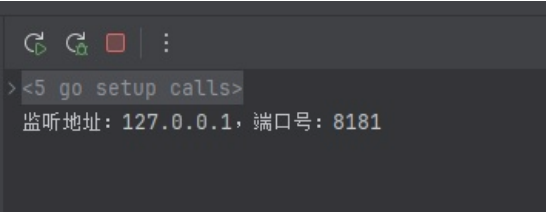
```

25
26     resp, err := client.Do(request)
27     if err != nil {
28         return
29     }
30
31     body := &bytes.Buffer{}
32
33     _, err = io.Copy(body, resp.Body)
34
35     fmt.Printf("添加用户信息, 响应码: %d\n", resp.StatusCode)
36 }
37
38 // 删除用户信息
39 func deleteUserInfo(client *http.Client, name string) {
40
41     fmt.Printf("删除用户: %s", name)
42
43     request, err := http.NewRequest("DELETE",
44         fmt.Sprintf("https://127.0.0.1:8181/user/info/%s", name), nil)
45     if err != nil {
46         return
47     }
48
49     resp, err := client.Do(request)
50     if err != nil {
51         return
52     }
53
54     body := &bytes.Buffer{}
55
56     _, err = io.Copy(body, resp.Body)
57
58     fmt.Printf("删除用户信息, 响应码: %d\n", resp.StatusCode)
59 }
60
61
62
63
64
65
66

```

代码运行

HTTP3服务端启动结果如下图所示：



```

><5 go setup calls>
监听地址: 127.0.0.1, 端口号: 8181

```

客户端运行结果如下图所示：

```
><5 go setup calls>
获取用户信息：
获取用户信息，响应码：200， 消息体：获取用户信息
添加用户信息:李四
添加用户信息，响应码：200
删除用户:张三
删除用户信息，响应码：200

Process finished with the exit code 0
```

服务端接收到客户端请求后输出如下图所示：

```
><5 go setup calls>
监听地址：127.0.0.1，端口号：8181
获取用户信息接口调用，URL： /user/info, 客户端地址： 127.0.0.1:62992
添加用户信息接口调用，URL： /user/info, 客户端地址： 127.0.0.1:62992
添加用户信息： 李四
删除用户信息接口调用，URL： /user/info/张三, 客户端地址： 127.0.0.1:62992
删除[张三]用户信息
```

经抓包分析，服务器与客户端之间的通信协议采用QUIC协议，如下图所示：

quic						
No.	Time	Source	Destination	Protocol	Length	Info
1134	2025-0...	127.0.0.1	127.0.0.1	QUIC	1284	Initial, DCID=e2b34bb8a517d64f974130a0b49ede65
1135	2025-0...	127.0.0.1	127.0.0.1	QUIC	1312	Handshake, DCID=4a2a7b5d, SCID=f2253bcf
1136	2025-0...	127.0.0.1	127.0.0.1	QUIC	344	Protected Payload (KP0), DCID=4a2a7b5d
1137	2025-0...	127.0.0.1	127.0.0.1	QUIC	62	Protected Payload (KP0), DCID=4a2a7b5d
1138	2025-0...	127.0.0.1	127.0.0.1	QUIC	1284	Initial, DCID=f2253bcf, SCID=4a2a7b5d, PKN
1139	2025-0...	127.0.0.1	127.0.0.1	QUIC	208	Protected Payload (KP0)
1140	2025-0...	127.0.0.1	127.0.0.1	QUIC	61	Protected Payload (KP0)
1141	2025-0...	127.0.0.1	127.0.0.1	QUIC	104	Protected Payload (KP0)
1142	2025-0...	127.0.0.1	127.0.0.1	QUIC	60	Protected Payload (KP0)
1143	2025-0...	127.0.0.1	127.0.0.1	QUIC	319	Protected Payload (KP0)
1144	2025-0...	127.0.0.1	127.0.0.1	QUIC	60	Protected Payload (KP0)
1145	2025-0...	127.0.0.1	127.0.0.1	QUIC	134	Protected Payload (KP0)

> Frame 1134: 1284 bytes on wire (10272 bits), 1284 bytes captured (10272 bits) on interface \Device\NPF{...}

> Null/Loopback

> Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1

> User Datagram Protocol, Src Port: 62993, Dst Port: 8181

> QUIC IETF

> QUIC Connection information

[Packet Length: 1252]1... .. = Header Form: Long Header (1).1.. .. = Fixed Bit: True..00 .. = Packet Type: Initial (0)[.... 00.. = Reserved: 0][.... ..01 = Packet Number Length: 2 bytes (1)]Version: 1 (0x00000001)Destination Connection ID Length: 16Destination Connection ID: e2b34bb8a517d64f974130a0b49ede65Source Connection ID Length: 4Source Connection ID: 4a2a7b5dToken Length: 0Length: 1222[Packet Number: 0]

