实现HTTP服务器

张林伟

小米集团技术委员会
Xiaomi Technical Committee

为什么分享

1. 上层技术不断更迭，而底层原理/协议/思想是不变的（**TCP/IP/HTTP几十年没变**），而这些知识才是一个优秀程序员的**核心竞争力**
2. 互联网的**繁荣离不开TCP/IP/HTTP**等协议，因此有必要深入理解这些底层协议
3. 个人理解有限，希望抛砖引玉，和大家**一起讨论**，加强对这些底层协议的理解

你能学到什么

1. 如何实现一个简单的**静态资源HTTP服务器**、**动态技术原理**、如何**加密HTTP**通信（HTTPS）
2. 如何实现**文本协议**和**二进制协议**
3. 这些技术的**常见应用**

1.静态资源服务器

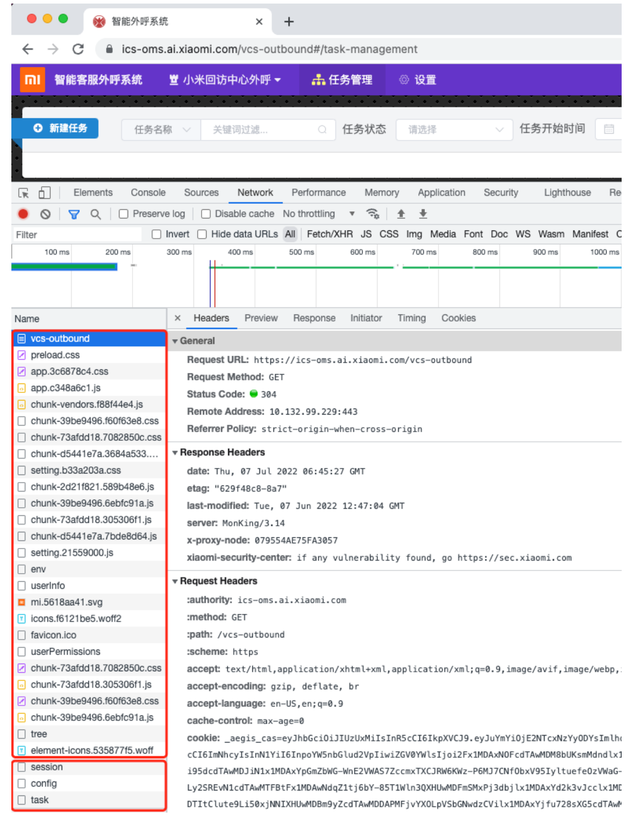
2.服务器动态技术

3.服务器加密通信

1.静态资源服务器

1. 静态资源服务器应用
2. 理解HTTP协议
3. 实现静态资源服务器
4. 总结和启发

静态资源服务器应用



前后端分离

现代web普遍采取前后端分离，将静态资源放到如Nginx服务器上，然后通过请求后端接口获取动态数据。催生出前端工程师和后端工程师工种，各自的工作更加内聚，提高效率。

CDN加速技术

将一个站点上的各种静态资源，分发到分布在全国各地的节点上，使用户可以就近访问所需资源，从而缩短用户下载静态资源的延时，提高用户访问网站的响应速度以及网站的可用性。

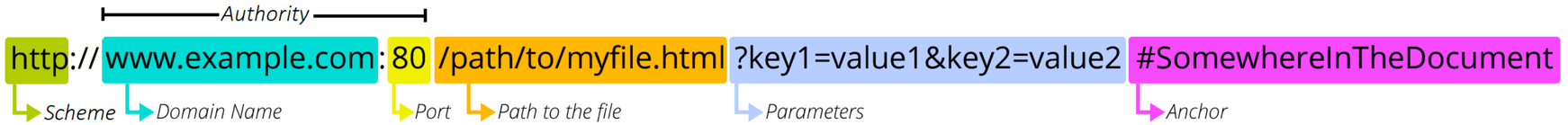
搭建个人博客站点

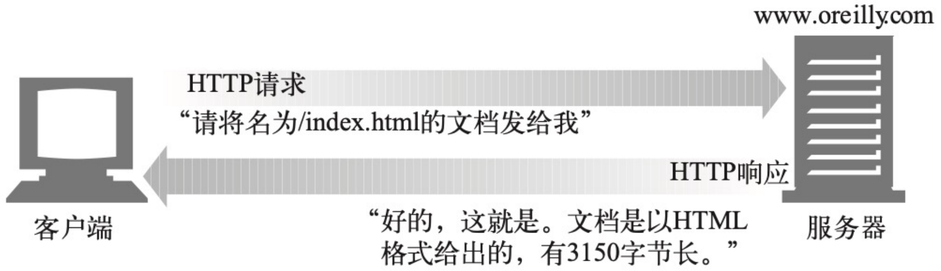
个人博客站点通常是将markdown文本转为静态HTML文档进行展示。

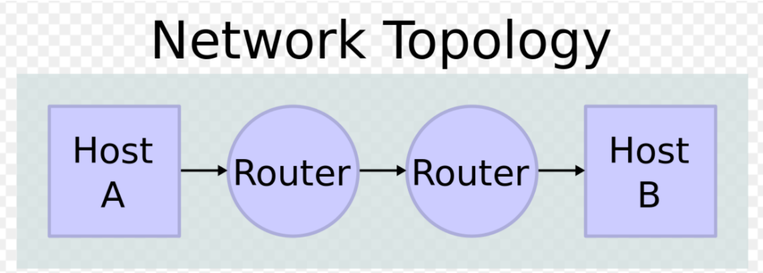
理解HTTP协议

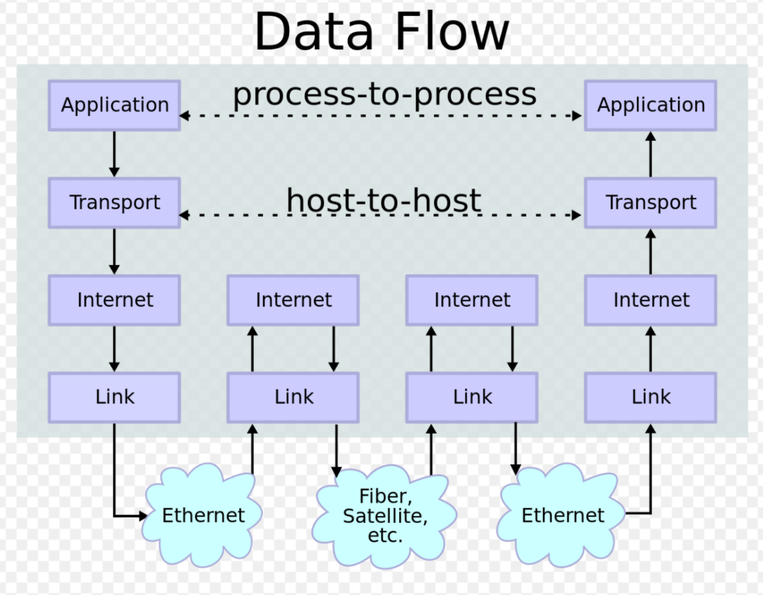
特点

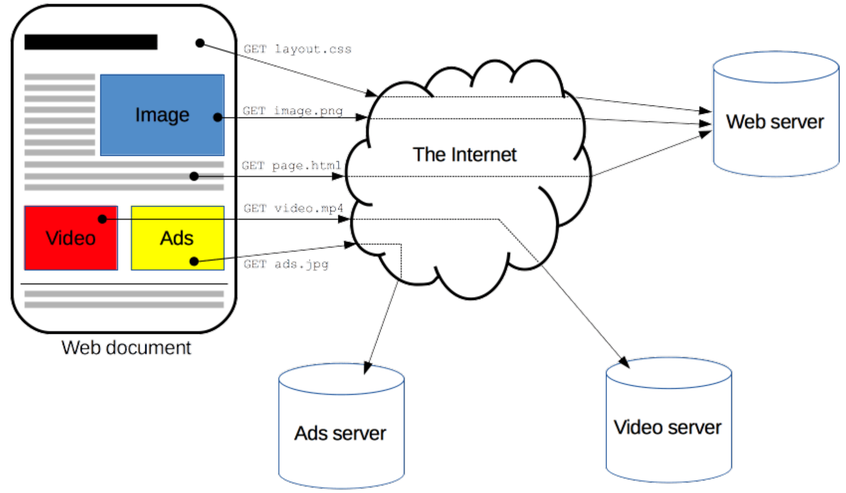
1. 基于**TCP**的**应用层**协议
2. **无状态**协议
3. C/S模型
4. **文本**协议











请求和响应报文

结构

1. 起始行
2. 首部Headers
3. 空行
3. 主体Body（可选）

特点

1. 起始行和首部均为**ASCII编码**
2. 起始行和首部均以**回车符和换行符结尾**
3. 首部以**空行结束**
3. 主体可以包含**文本数据**（编码格式不限）、**二进制数据**（如图片、音视频等），也可以为空
4. 主体若存在，首部必须包含**Content-Type和Content-Length**

```
GET /index.html HTTP/1.1\r\n
Host: localhost:8888\r\n
User-Agent: curl/7.77.0\r\n
Accept: */*\r\n
\r\n
```

```
POST /greeting HTTP/1.1\r\n
Host: foo.com\r\n
Content-Type: application/x-www-form-urlencoded\r\n
Content-Length: 13\r\n
\r\n
say=Hi&to=Mom
```

```
HTTP/1.1 200 OK\r\n
Content-Type: text/html\r\n
Content-Length: 142\r\n
\r\n
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Hello</title>
</head>
<body>
This is index page.
</body>
</html>
```

理解HTTP协议

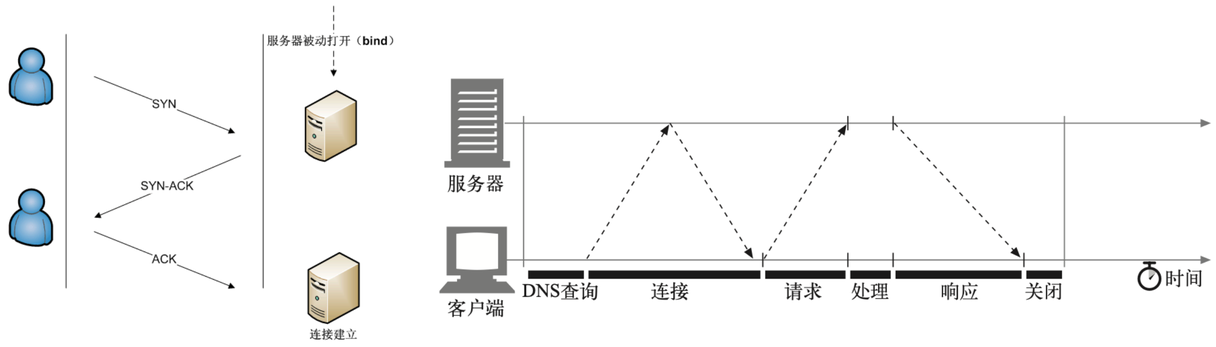
实际案例分析

1. 在线机器人服务间通过HTTP通信，效率低的原因?

- 短连接，每次传输需要**重新TCP三次握手**，比较耗时（主流RPC通信框架都是基于长连接）
- 编解码，HTTP协议属于文本协议，需要对所有数据**编解码**，更加耗时
- 传输效率，文本协议较二进制协议，一般来说**相同信息量需要传递更多二进制数据**，效率低

```
> Frame 195: 571 bytes on wire (4568 bits), 571 bytes captured (4568 bits) on interface en0, id 0
> Ethernet II, Src: Apple_03:c0:96 (3c:06:30:03:c0:96), Dst: NewH3CTe_66:68:01 (34:6b:5b:66:68:01)
> Internet Protocol Version 4, Src: 10.224.223.58, Dst: 120.92.65.45
> Transmission Control Protocol, Src Port: 60470, Dst Port: 443, Seq: 1, Ack: 1, Len: 517
v Transport Layer Security
  v TLSv1.2 Record Layer: Handshake Protocol: Client Hello
      Content Type: Handshake (22)
      Version: TLS 1.0 (0x0301)
      Length: 512
    v Handshake Protocol: Client Hello
        Handshake Type: Client Hello (1)
        Length: 508
        Version: TLS 1.2 (0x0303)
      > Random: 397683423b228651e588e6a41383afbe14c9df2b1a1f0e517ec1f95c0c44820d
        Session ID Length: 32
        Session ID: 943d42f6ff470cd10014995e813e1350ea28b2fca9df14b5753ee7ae661c1974
        Cipher Suites Length: 36
      > Cipher Suites (18 suites)
        Compression Methods Length: 1
      > Compression Methods (1 method)
        Extensions Length: 399
      > Extension: Reserved (GREASE) (len=0)
      > Extension: server_name (len=22)
      > Extension: extended_master_secret (len=0)
```

```
0030  10 00 24 11 00 00 16 03  01 02 00 01 00 01 fc 03   ··$·····  ········
0040  03 39 76 83 42 3b 22 86  51 e5 88 e6 a4 13 83 af   ·9v·B;"·  Q·······
0050  be 14 c9 df 2b 1a 1f 0e  51 7e c1 f9 5c 0c 44 82   ····+···  Q~··\·D·
0060  0d 20 94 3d 42 f6 ff 47  0c d1 00 14 99 5e 81 3e   · ·=B··G  ·····^·>
0070  13 50 ea 28 b2 fc a9 df  14 b5 75 3e e7 ae 66 1c   ·P·(····  ··u>··f·
0080  19 74 00 24 5a 5a 13 01  13 02 13 03 c0 2c c0 2b   ·t·$ZZ··  ·····,·+
0090  cc a9 c0 30 c0 2f cc a8  c0 24 c0 23 c0 0a c0 09   ···0·/··  ·$·#····
00a0  c0 28 c0 27 c0 14 c0 13  01 00 01 8f 1a 1a 00 00   ·(·'····  ········
00b0  00 00 00 16 00 14 00 00  11 61 70 6d 2e 66 2e 6d   ·········  ·apm.f.m
00c0  69 6f 66 66 69 63 65 2e  63 6e 00 17 00 00 ff 01   ioffice.  cn······
00d0  00 01 00 00 0a 00 0c 00  0a ca ca 00 1d 00 17 00   ········  ········
00e0  18 00 19 00 0b 00 02 01  00 00 10 00 0b 00 09 08   ········  ········
00f0  68 74 74 70 2f 31 2e 31  00 05 00 05 01 00 00 00   http/1.1  ········
0100  00 00 0d 00 18 00 16 04  03 08 04 04 01 05 03 02   ········  ········
0110  03 08 05 08 05 05 01 08  06 06 01 02 01 00 12 00   ········  ········
0120  00 00 33 00 2b 00 29 ca  ca 00 01 00 00 1d 00 20   ··3·+·)·  ·····
```



xxxxxxxx xxxxxxxx xxxxxxxx -> “512” -> 512（可表示0-999）

xxxxxxxx xxxxxxxx xxxxxxxx -> 512（可表示0-2的24次方）

理解HTTP协议

实际案例分析

2. 外呼系统批量暂停任务，循环发送请求处理 vs 合并操作到一个请求?

由于HTTP优点是可理解性、可扩展性，而不是性能，因此尽量将多个请求合并成一个发送，这样只需要处理一个TCP连接、一次编解码、减少传输内容（多次传输会**重复传输请求头**）。

3. 为了省事，将所有内容（包括无关的）都放到请求或响应body里传递，导致HTTP请求或响应body很大，包含“万物”，有什么弊端?

一个是**传输内容过多**耗时长，在**解析**报文的时候也会更加耗时。

请求和响应报文

请求方法

1. GET 获取资源
2. POST 创建资源
3. PUT/PATCH 修改资源
4. DELETE 删除资源

响应状态码

1. 2xx 成功
2. 3xx 重定向
3. 4xx 客户端错误
4. 5xx 服务端错误

理解HTTP协议

实际案例分析

1. 后端服务提供的接口统一都是POST方法，最佳实践应该是什么？
   - RESTful风格API，通常**URL定位资源**，HTTP**方法代表操作**（**可读性好，便于协作**），如，
     - GET /products　　代表查询产品列表
     - GET /products/3　　代表查询id为3的产品
     - POST /products　　代表创建产品
     - PUT /products/3　　代表**全量修改**id为3的产品
     - PATCH /products/3　　代表**局部修改**id为3的产品
     - DELETE /products/3　　代表删除id为3的产品
   - GET请求通常是不带副作用（修改数据），浏览器可能会**缓存**返回的内容

理解HTTP协议

实际案例分析

2. 后端服务响应码只用200和500，最佳实践应该是什么?
   - RESTful风格API，状态码是有不同含义的，表示此次请求的结果，如
     - 200 OK 请求成功
     - 400 Bad Request 请求语法/格式/参数错误，服务器无法理解
     - 401 Unauthorized 未认证
   - 浏览器对某些状态码会有**特殊行为**（最典型如301永久重定向、307临时重定向）

总结：**RESTful风格API，看URL就知道要什么，看HTTP方法就知道干什么，看HTTP状态码就知道结果如何**。

请求和响应报文

首部 Content-Type

1. HTTP基于MIMIE来描述并标记多媒体内容
2. text/html html格式文本
3. text/plain 普通ascii文本
4. image/jpeg jpeg格式图片
5. image/gif gif格式图片
6. application/vnd.ms-powerpoint 微软ppt格式

首部 Content-Length

1. Content-Length代表主体**字节数**
2. 计算主体字节数时，是计算**主体编码和压缩后**的值

```
print("HTTP/1.1 200 OK", end="\r\n")
print("Content-Type: text/html; charset=utf-8", end="\r\n")
content = "<html><body>" + \
          "<p>Got the student: %s</p>" % get_student_from_db(name) + \
          "</body></html>"
print("Content-Length: %d" % len(bytes(content, encoding="utf-8")), end="\r\n")
print("", end="\r\n")  # 空行
print(content, end="")
```

实现静态资源服务器

实现步骤

1. Socket创建、绑定与监听
2. 多线程处理客户端连接
3. 读取客户端发送内容
4. 从原始字节内容解析请求
5. 获取服务器上指定静态资源并构建响应
6. 将响应转换成字节内容并发送客户端
7. 关闭连接

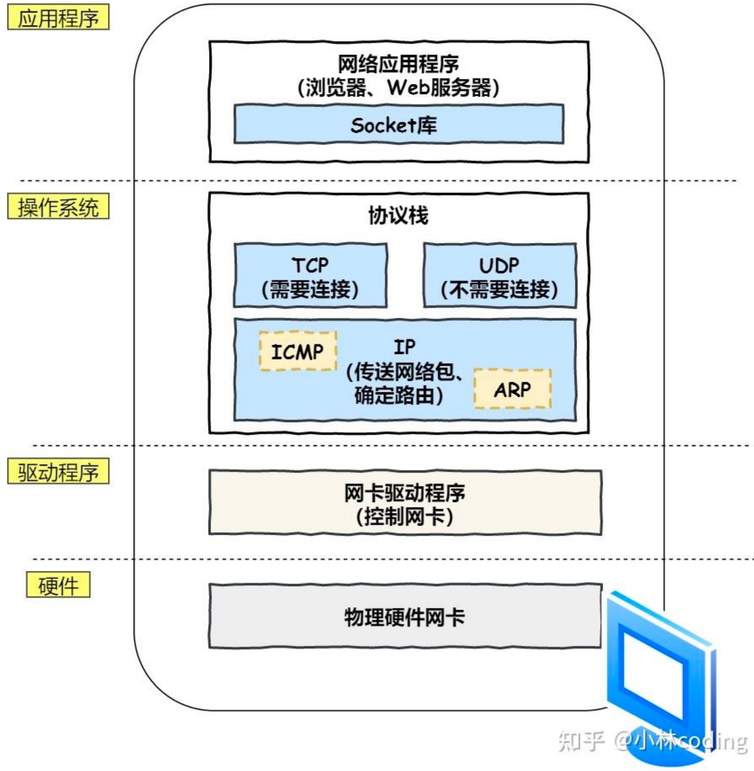
实现静态资源服务器

1.Socket创建、绑定与监听

Socket是操作系统提供的一组API，封装了传输层（TCP/UDP）通信细节

1. **socket**，初始化 socket
2. **bind**，绑定 IP 地址和端口
3. **listen**，开始监听
4. **accept**，接收连接
5. **send**，发送数据
6. **recv**，接收数据

```c
int server_start(int port) {
    printf("Server starting on port %d ...\n", port);

    // 创建套接字
    int sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0) {
        perror("create socket failed.\n");
        exit(1);
    }

    // 绑定端口
    struct sockaddr_in server_addr;
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(port);
    server_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    if (bind(sockfd, (struct sockaddr *) &server_addr, sizeof(server_addr)) < 0) {
        perror("server bind failed.\n");
        exit(1);
    }

    // 监听端口
    if (listen(sockfd, 20) < 0) {
        perror("server listen failed.\n");
        exit(1);
    }
    return sockfd;
}
```



知乎 @小林coding

实现静态资源服务器

2.多线程处理客户端连接

1. 循环接受（**accept**）客户端连接
2. 新建线程执行请求

```c
int accept_conn(int server_sockfd) {
    // 接受客户端连接
    struct sockaddr_in client_addr;
    socklen_t client_len = sizeof(client_addr);
    int client_sockfd = accept(server_sockfd, (struct sockaddr *) &client_addr, &client_len);
    if (client_sockfd < 0) {
        perror("accept conn failed.\n");
        exit(1);
    }
    return client_sockfd;
}
```

```c
int server_sockfd = server_start( port: 8888);
while (1) {
    // 接受客户端连接
    int client_sockfd = accept_conn(server_sockfd);
    // 多线程处理客户端连接
    pthread_t new_thread;
    if (pthread_create(&new_thread, NULL, (void *)handle_conn, (void *) (intptr_t) client_sockfd) != 0) {
        perror("pthread_create failed");
        exit(1);
    }
}
```

实现静态资源服务器

3.读取客户端发送内容

通过**recv**读取客户端发送字节内容

1. 如果首部有Content-Length，需要读取**指定长度的字节内容**
2. 如果首部没有Content-Length或者方法为GET，则**读到空行则结束**

```c
char *read_request(int client_sockfd) {
    char raw_request[1024];
    ssize_t recv_len;
    // 读取客户端请求
    // TODO 循环读取完整请求
    if ((recv_len = recv(client_sockfd, raw_request, sizeof(raw_request), 0)) < 0) {
        perror("recv raw_request failed.\n");
        exit(1);
    }
    return strdup(raw_request);
}
```

实现静态资源服务器

4.从原始字节内容解析请求

1. 需要对uri内容进行**解码**（如浏览器会对中文字符进行编码）
2. 首部存在如Content-Encoding: gzip，需要对主体进行**解压**

```
POST /greeting HTTP/1.1\r\n
Host: foo.com\r\n
Content-Type: application/x-www-form-urlencoded\r\n
Content-Length: 13\r\n
\r\n
say=Hi&to=Mom
```

```c
struct http_request_t *parse_request(char *raw_request) {
    struct http_request_t *request = (struct http_request_t *) malloc(sizeof(struct http_request_t));
    memset(request, 0, sizeof(struct http_request_t));

    // 解析http方法
    size_t method_len = strcspn(raw_request, charset: " ");
    request->method = (char *) malloc( size: method_len + 1);
    memcpy(request->method, raw_request, method_len);
    request->method[method_len] = '\0';
    raw_request += method_len + 1; // 跳过空格

    // 解析url
    size_t url_len = strcspn(raw_request, charset: " ");
    char *url = (char *) malloc( size: url_len + 1);
    memcpy(url, raw_request, url_len);
    url[url_len] = '\0';
    raw_request += url_len + 1; // 跳过空格
    // 解析url中的path
    size_t path_len = strcspn(url, charset: "?");
    char *path = (char *) malloc( size: path_len + 1);
    memcpy(path, url, path_len);
    path[path_len] = '\0';
    request->path = url_decode(path);  // 对url解码（浏览器会对中文字符进行编码）
    url += path_len + 1; // 跳过空格
    // 解析url中的query_string
    size_t query_string_len = strlen(url);
    char *query_string = (char *) malloc( size: query_string_len + 1);
    memcpy(query_string, url, query_string_len);
    query_string[query_string_len] = '\0';
    request->query_string = url_decode(query_string);  // 对url解码（浏览器会对中文字符进行编码）
```

实现静态资源服务器

5.获取服务器上指定静态资源并构建响应

1. 判断是否为静态资源请求（可以通过请求path**后缀**，如/xx.jpeg；也可以协商 **/html**/xx 为静态资源请求；path为 / 一般默认为 /index.html）
2. 通过**拼接静态资源根目录和请求path**，得到静态资源绝对路径
3. 如果静态资源存在，则读取静态资源字节内容并构建**200响应**；如果静态资源不存在则构建**404响应**

```
GET /index.html HTTP/1.1\r\n
Host: localhost:8888\r\n
User-Agent: curl/7.77.0\r\n
Accept: */*\r\n
\r\n
```

```c
struct http_response_t *process_request(struct http_request_t *request) {
    printf("Thread %d process request: %s %s\n", (int) pthread_self(), request->method, request->path);

    if (is_static_request(request)) {
        return execute_file(request);
    }
```

```c
struct http_response_t *execute_file(struct http_request_t *request) {
    printf("execute_file for request %s %s\n", request->method, request->path);
    if (strcmp(request->path, "/") == 0) {
        request->path = "/index.html";
    }
    // 读取文件内容
    char *filename = str_concat(HTML_ROOT, request->path);
    char *content = read_file_as_str(filename);
    if (content == NULL) {
        // 文件不存在，返回404
        return build_response_404();
    }
    // 构建响应
    struct http_response_t *response = (struct http_response_t *)malloc(sizeof(struct http_response_t));
    response->version = HTTP_VERSION_11;
    response->status_code = 200;
    response->status_text = "OK";
    response->content_type = "text/html";
    response->content_length = strlen(content);
    response->body = content;
    return response;
}
```

实现静态资源服务器

6.将响应转换成字节内容并发送客户端

将响应转换成字节内容并通过**send**发送给客户端

```c
void send_response(int client_sockfd, struct http_response_t *response) {
    // 发送响应
    char *raw_response = response->raw_response ? response->raw_response : generate_raw_response(response);
    if (send(client_sockfd, raw_response, strlen(raw_response), 0) < 0) {  // 需要用strlen，sizeof返回的是指针大小，即char *
        perror("send response failed.\n");
        exit(1);
    }
}
```

7.关闭连接

如果首部有Connection: keep-alive，则保持开启

```
HTTP/1.1 200 OK\r\n
Content-Type: text/html\r\n
Content-Length: 142\r\n
\r\n
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Hello</title>
</head>
<body>
This is index page.
</body>
</html>
```

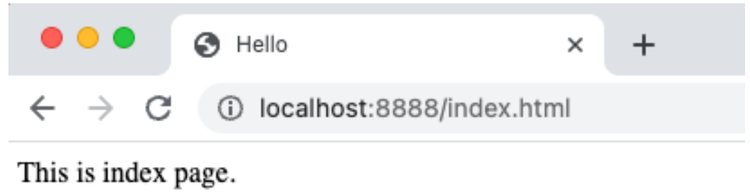
实现静态资源服务器

测试

1. 通过curl进行测试
2. 通过telnet进行测试
3. 通过chrome进行测试

```
$ curl "http://localhost:8888/index.html" --verbose
*   Trying ::1:8888...
* connect to ::1 port 8888 failed: Connection refused
*   Trying 127.0.0.1:8888...
* Connected to localhost (127.0.0.1) port 8888 (#0)
> GET /index.html HTTP/1.1
> Host: localhost:8888
> User-Agent: curl/7.77.0
> Accept: */*
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 200 OK
< Content-Type: text/html
< Content-Length: 142
<
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Hello</title>
</head>
```

```
$ telnet localhost 8888
Trying ::1...
telnet: connect to address ::1: Connection refused
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
GET /index.html HTTP/1.1
Host: localhost:8888
User-Agent: curl/7.77.0
Accept: */*


HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 142

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Hello</title>
</head>
```



总结和启发

1. 应用层协议**基于Socket** API实现，本质就是按照协议**发送和接收指定格式数据**
2. 文本协议都会**带有编码信息**，要么默认，要么显示指定，由于需要编解码，会比二进制协议**慢**（因此尽量不要携带无关信息），**相同信息量需要更多传输比特**
3. TCP连接建立需要三次握手（**多次网络来回**），尽量**复用**连接，**合并**请求

1.静态资源服务器

2.服务器动态技术

3.服务器加密通信

2.服务器动态技术

1. 动态技术应用
2. 动态技术 - CGI
3. 动态技术 - Java Servlet
4. 总结和启发

动态技术应用

前后端分离

现代web普遍采取前后端分离，后端服务可以专注于接口实现逻辑，符合依赖倒置原则（模块间依赖于抽象接口，而不是具体实现）。催生出前端工程师和后端工程师工种，各自的工作更加内聚，提高效率。

通过接口隐藏后端复杂性

大型网站，一个简单的接口背后可能存在复杂的处理逻辑，技术可能涉及到Java web、大数据处理、分布式系统、数据库、缓存、微服务通信等等，通过接口隐藏内部复杂性，外部无感知。

动态技术

静态资源服务器的不足

早期的Web服务器，只能响应浏览器发来的HTTP静态资源的请求，并将存储在服务器中的静态资源返回给浏览器。 随着Web技术的发展，传统的**静态页面无法满足用户需求**，用户希望页面**灵活具有交互性、因人而异**。

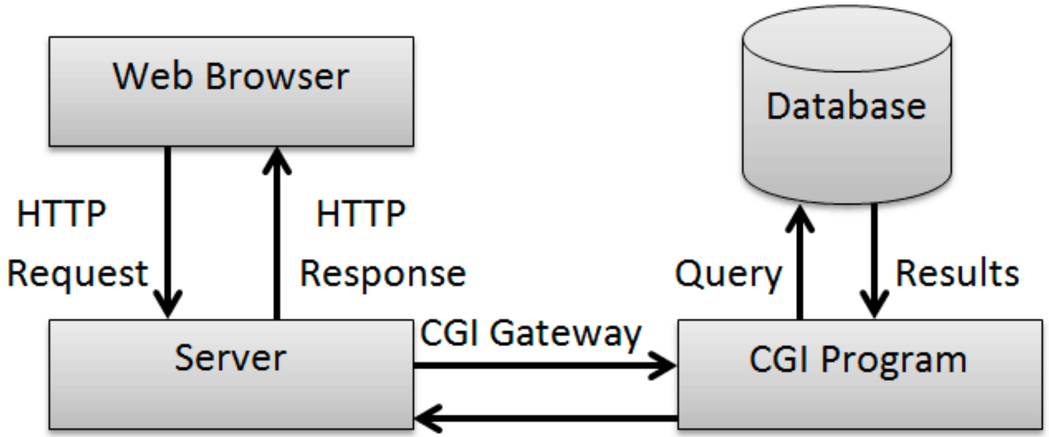
因此衍生出了多种动态技术

1. **CGI（Common Gateway Interface）**
2. **Java Servlet**
3. 微软ASP（Active Server Pages）

动态技术 – CGI

理解

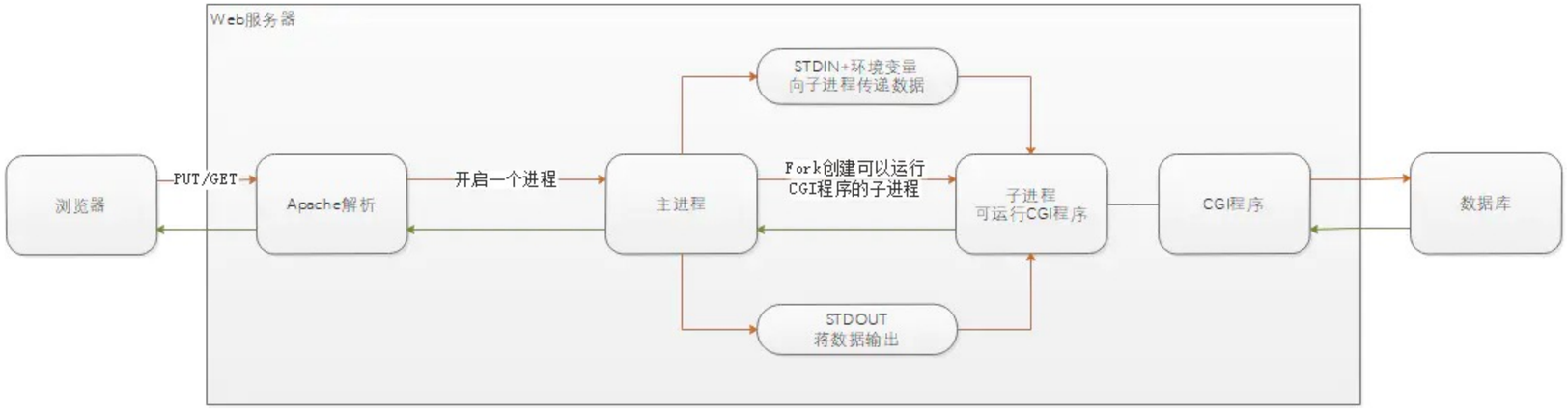
Web服务器不能够直接运行动态脚本，为了解决Web服务器与外部应用程序（称CGI脚本）之间数据互通，于是出现了CGI通用网关接口。 简单理解，可以认为CGI是Web服务器和运行其上的应用程序进行**“交流”的一种约定**。



动态技术 – CGI

协议规定

1. 采用 **fork-and-exec 模式**，即来一个请求，fork一个子进程来exec CGI脚本
2. 请求的Header头设置成子进程的**环境变量**
3. 请求的Body正文设置成子进程的**标准输入STDIN**
4. 子进程的**标准输出STDOUT设置为响应**，包含Header头和Body正文

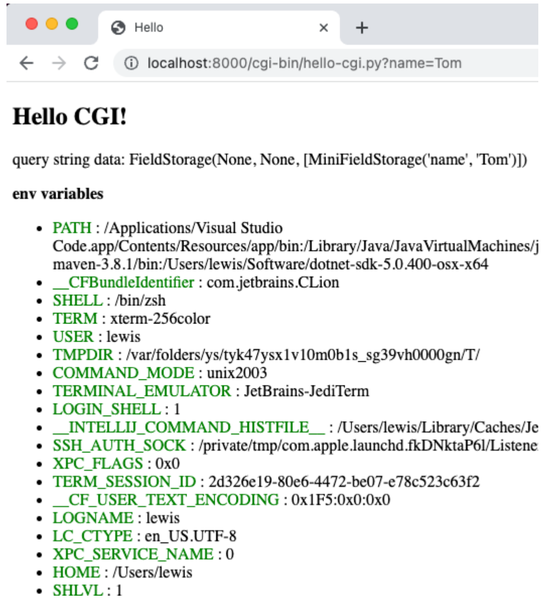


动态技术 - CGI

利用Python标准库快速体验CGI动态技术

1. 通过 python3 -m http.server --bind localhost --cgi 8000 命令启动HTTP服务器
2. 在当前目录创建 cgi-bin 目录，并在 cgi-bin目录下编写一个CGI脚本 hello-cgi.py
3. 浏览器访问 http://localhost:8000/cgi-bin/hello-cgi.py?name=Tom

```python
#!/usr/bin/env python3
# -*- coding: UTF-8 -*-

import cgi, cgitb, os

cgitb.enable()  # 开启调试模式

print("Content-Type: text/html\n")
print("<!doctype html>"
      "<title>Hello</title>"
      "<h2>Hello CGI!</h2>"
      "<p>query string data: %s</p>" % cgi.FieldStorage())  # 打印query string

# 打印环境变量
print("<b>env variables</b><br>")
print("<ul>")
for key in os.environ.keys():
    print("<li><span style='color:green'>%30s </span> : %s </li>" % (key, os.environ[key]))
print("</ul>")
```



动态技术 – CGI

实现步骤

1. Socket创建、绑定与监听
2. 多线程处理客户端连接
3. 读取客户端发送内容
4. 从原始字节内容解析请求
5. **fork进程执行服务器上指定CGI脚本，并将请求通过环境变量和标准输入传递给子进程，接收子进程标准输出作为响应**
6. 将响应转换成字节内容并发送客户端
7. 关闭连接

动态技术 – CGI

fork进程执行CGI脚本

1. 进程间通信采用**管道**
2. 子进程将标准输入和标准输出**重定向**到管道
3. 父进程**通过管道**进行读写

```
// fork子进程
if ( (pid = fork()) < 0) {  // 创建子进程
    puts("Error forking...\n");
    return build_response_500();
}

if (pid == 0) {
    // 子进程
    close(parent_child_pipe[1]);  // 关闭父进程到子进程的管道写端
    close(child_parent_pipe[0]);  // 关闭子进程到父进程的管道读端

    // 将父进程到子进程的管道读端重定向到标准输入
    dup2(parent_child_pipe[0], STDIN_FILENO);
    // 将子进程到父进程的管道写端重定向到标准输出
    dup2(child_parent_pipe[1], STDOUT_FILENO);

    // 设置环境变量
    char method_env[255];
    char query_string_env[255];
    char content_type_env[255];
    char content_length_env[255];
    sprintf(method_env, "REQUEST_METHOD=%s", request->method);
    sprintf(query_string_env, "QUERY_STRING=%s", request->query_string);
    sprintf(content_type_env, "CONTENT_TYPE=%s", request->content_type);
    sprintf(content_length_env, "CONTENT_LENGTH=%d", request->content_length);
    putenv(method_env);
    putenv(query_string_env);
    putenv(content_type_env);
    putenv(content_length_env);

    // 执行cgi程序
    char command[255];
    // 拼接成 ./cgi-bin/student.py (cgi脚本会被编译进build目录)
    sprintf(command, ".%s", request->path);
    if (execl(command, command, (char *) 0) < 0) {
        puts(generate_raw_response(build_response_500()));
    }
```

```
// 父进程
close(parent_child_pipe[0]);  // 关闭父进程到子进程的管道读端
close(child_parent_pipe[1]);  // 关闭子进程到父进程的管道写端

// 将body数据写入管道
if (request->content_length > 0) {
    write( fd: parent_child_pipe[1], request->body, request->content_length);
}

// 关闭管道会发生EOF https://stackoverflow.com/questions/22032120/closing-pipe-file-descriptor-in-c
close(parent_child_pipe[1]);  // 关闭父进程到子进程的管道写端

// TODO 可以等待超时，超时则kill子进程并返回500
wait(&rv);  // 等待子进程结束，返回子进程的退出状态

// 读取CGI脚本的输出
if ( (n = read(child_parent_pipe[0], raw_response, MAXLINE)) < 0) {
    printf("Error reading from CGI script.\n");
    return build_response_500();
}
if (n == 0) {
    printf("Reading nothing from CGI script.\n");
    return build_response_500();
}
raw_response[n] = '\0';

return build_raw_response(raw_response);
```

动态技术 – CGI

编写CGI脚本

1. 从**环境变量**读取http请求头部信息
2. 从**标准输入**流中读取请求body
3. 将响应写到**标准输出**流中

```python
# 从环境变量读取http请求头部信息
query_string = os.getenv("QUERY_STRING")
method = os.getenv("REQUEST_METHOD")
content_type = os.getenv("CONTENT_TYPE")

if method == "GET":
    query_string_pairs = query_string.split("&")
    for pair in query_string_pairs:
        key_value = pair.split("=")
        if key_value[0] == "name":
            name = key_value[1]
            # 将响应内容写入到标准输出流中
            print("HTTP/1.1 200 OK", end="\r\n")
            print("Content-Type: text/html; charset=utf-8", end="\r\n")
            content = "<html><body>" + \
                      "<p>Got the student: %s</p>" % get_student_from_db(name) + \
                      "</body></html>"
            print("Content-Length: %d" % len(bytes(content, encoding="utf-8")), end="\r\n")
            print("", end="\r\n")  # 空行
            print(content, end="")

elif method == "POST":
    # 从标准输入流中读取请求body
    body = input()
    name = ""
    age = -1
    grade = "Unknown"
    if "application/json" not in content_type:
        body_dict = json.loads(body)
        name = body_dict["name"]
        age = body_dict["age"]
        grade = body_dict["grade"]

    print("HTTP/1.1 200 OK", end="\r\n")
    print("Content-Type: text/html; charset=utf-8", end="\r\n")
    content = "<html><body>" + "<p>Created student: %s</p>" % create_student_in_db(name, age, grade) + "</body></html>"
    # len统计str类型的结果是字符数（一个汉字为一个字符），非字节数（一个汉字utf-8编码是3个字节）
    print("Content-Length: %d" % len(bytes(content, encoding="utf-8")), end="\r\n")
    print("", end="\r\n")  # 空行
    print(content, end="")
exit(0)
```

动态技术 – CGI

测试

通过curl进行测试

```
$ curl "http://localhost:8888/cgi-bin/student.py?name=Tom"
<html><body><p>Got the student: {'name': 'Tom', 'age': '20', 'grade': 'A'}</p></body></html>%
```

```
$ curl -H "Content-Type: application/json" -X POST -d '{"name":"John","age":18,"grade":"B"}' "http://localhost:8888/cgi-bin/student.py"
<html><body><p>Created student: {'name': 'John', 'age': 18, 'grade': 'B'}</p></body></html>%
```

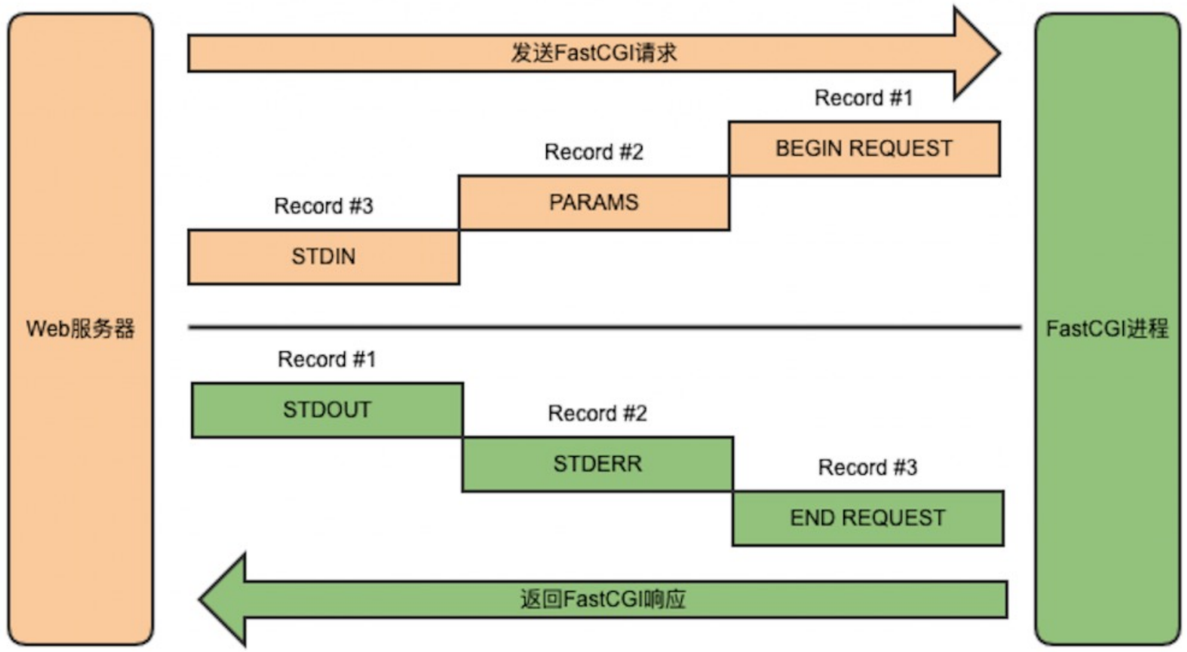
动态技术 – CGI

CGI的缺陷

1. fork-and-exec 模式**低效**，进程不断创建和销毁，**开销很大**
2. CGI脚本**无法重用**数据库连接等资源，**不支持本地缓存**等

FastCGI的出现

1. FastCGI是CGI的**改良版本**
2. FastCGI通过**常驻**一个管理进程和多个CGI解释器进程，进程间**通信通过Socket**进行



动态技术 – Java Servlet

理解

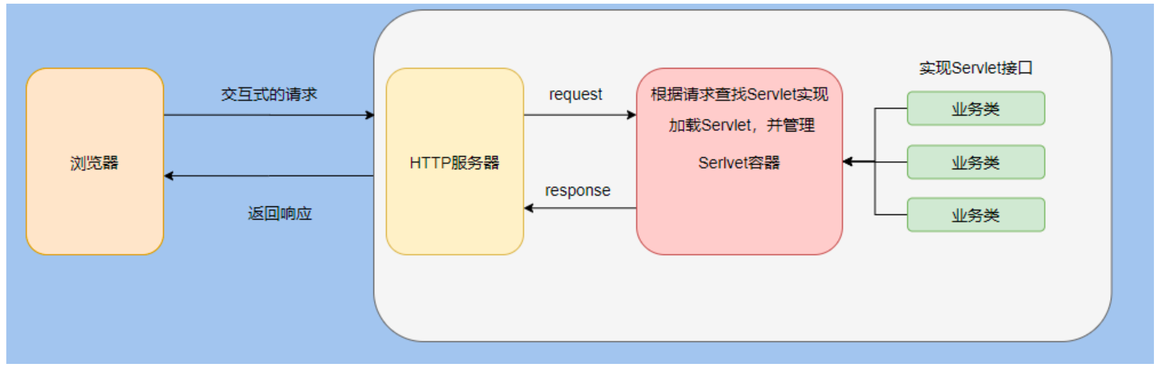
Servlet是基于Java语言的Web服务器端编程技术，按照Java EE规范定义，Servlet是运行在Servlet容器中的**Java类**，它能处理Web客户的HTTP请求，并产生HTTP响应。

Servlet容器
负责**加载和管理**Servlet接口实现类整个生命周期；根据请求**找到**对应的Servlet实现类并执行

Web容器
HTTP 服务器 + Servlet 容器

```java
public interface Servlet {
    public void init(ServletConfig config) throws ServletException;
    public void service(ServletRequest req, ServletResponse res)
        throws ServletException, IOException;
    public void destroy();
    ...
}
```

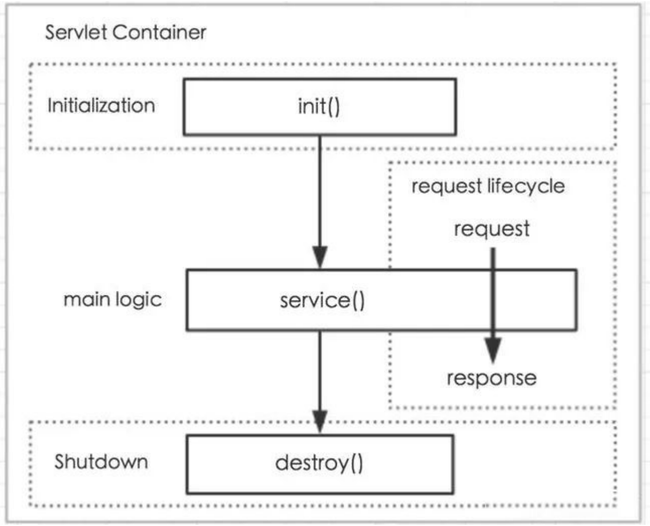
```java
public abstract class HttpServlet extends GenericServlet {
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) ...
    protected void doPost(HttpServletRequest req, HttpServletResponse resp) ...
    protected void doPut(HttpServletRequest req, HttpServletResponse resp) ...
    public void service(ServletRequest req, ServletResponse res) ...
    ...
}
```



动态技术 – Java Servlet

Servlet生命周期

1. 当调用某个servlet时，**载入**该servlet类，并调用其**init方法**
2. 创建HttpServletRequest和HttpServletResponse对象
3. 调用**service方法**，并将request和response对象传入
4. 当关闭servlet时，调用其**destroy方法**，并**卸载**该类



动态技术 – Java Servlet

Servlet与JSP关系

Java服务器页面（JSP）是HttpServlet的扩展。由于HttpServlet大多是用来响应HTTP请求，并返回Web页面，所以不可避免地，在编写Servlet时会涉及大量的HTML内容，这给Servlet的书写效率和可读性带来很大障碍，JSP便是在这个基础上产生的。其功能是**使用HTML的书写格式，在适当的地方加入Java代码片段**，将程序员从复杂的HTML中解放出来，更专注于Servlet本身的内容。

JSP在首次被访问的时候被应用服务器**转换为Servlet**，在以后的运行中，容器直接调用这个Servlet，而不再访问JSP页面。**JSP的实质仍然是Servlet**。

Hello.jsp

```
<html>
<head>
    <title>Hello World - JSP</title>
</head>
<body>
    <%-- JSP Comment --%>
    <h1>Hello World!</h1>
    <p>
    <%
         out.println("Your IP address is ");
    %>
    <span style="color:red">
        <%= request.getRemoteAddr() %>
    </span>
    </p>
</body>
</html>
```

动态技术 – Java Servlet

实现步骤

1. Socket创建、绑定与监听
2. 多线程处理客户端连接
3. 读取客户端发送内容
4. 从原始字节内容解析请求
5. **将请求交由Servlet容器处理，Servlet容器根据请求path找到对应Servlet类执行并返回响应**
6. 将响应转换成字节内容并发送客户端
7. 关闭连接

动态技术 – Java Servlet

Servlet容器处理

1. 初始化类加载器
2. **加载**Servlet类
3. **实例化**Servlet对象并执行**init方法**
4. 调用Servlet对象**service**方法处理请求

```
if (request.getUri().startsWith("/servlet")) {
  // 走servlet容器处理
  new ServletProcessor().process(request, response);
} else {
  new StaticResourceProcessor().process(request, response);
}

...

// 初始化类加载器
String uri = request.getUri();
String servletName = uri.substring(uri.lastIndexOf("/") + 1);
URLClassLoader loader = null;
try {
  URL[] urls = new URL[1];
  URLStreamHandler streamHandler = null;
  File classpath = new File(Constants.WEB_ROOT);
  String repository = (new URL("file", null, classPath.getCanonicalPath() + File.separator)
  urls[0] = new URL(null, repository, streamHandler);
  loader = new URLClassLoader(urls);
} catch (IOExcetion e) {
  ...
}
// 加载servlet类
Class servletClass = null;
try {
  servletClass = loader.loadClass(servletName);
} catch (ClassNotFoundException, e) {
  ...
}
// 实例化servlet对象并执行
Servlet servlet = null;
try {
  servlet = servletClass.newInstance();
  servlet.init();
  servlet.service(request, response);
} catch (Throwable e) {
  ...
}
```

总结和启发

1. 背后是通过**执行程序**来**动态生成响应**
2. 服务器和程序之间需要**按照指定协议通信**（CGI、内存对象传递）
3. 通过**进程隔离**运行用户代码：用户进程崩溃可通过主进程拉起，防止用户代码导致主进程崩溃无法恢复

总结

Web服务器本质

1.**监听**端口接收客户端连接（socket）
2.**接收并解析**HTTP请求
3.使用通用或专用协议对请求进行**分发**
    3.1. 静态资源（可理解为分发到磁盘读写）
    3.2. 动态页面（分发到应用程序）
    3.3. 反向代理（可理解为用HTTP协议进行HTTP请求的分发）
4.接收分发的请求产生的运行结果
5.将结果格式化成HTTP Response并写到socket里面
6.关闭连接或者Keep-Alive

进程内分发 vs 进程外分发

1. 进程内分发一般需要**通过内存传递一些对象**，这种分发方式**通常也是绑定语言**的，所以一般必须用相应的语言实现（比如Servlet）；某些语言有特殊的FFI（如Python有C API），也可以**通过FFI**的方式调用
2. 进程外分发只要**序列化格式匹配**就可以在不同语言之间通用，如CGI、FastCGI、HTTP等，这些分发可以用统一的方法

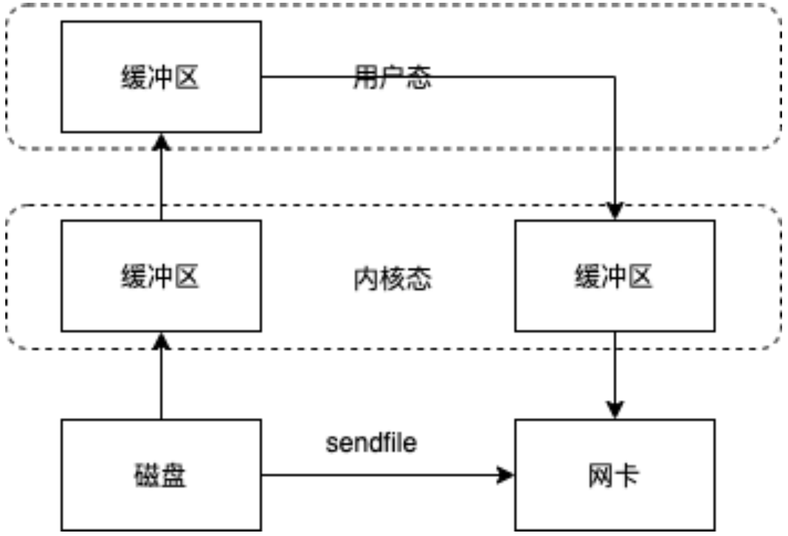
业界知名HTTP服务器介绍和其优化思路

知名服务器

1. Nginx
2. Tomcat

优化思路

1. 采用IO多路复用与多进程结合
2. sendfile零拷贝机制
3. 利用缓存（本地缓存、HTTP缓存）加速
4. 内容压缩



1.静态资源服务器

2.服务器动态技术

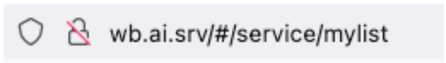
3.服务器加密通信

3.服务器加密通信

1. HTTPS应用
2. 密码学基础
3. 数字签名与证书
4. SSL/TLS协议
5. 总结和启发

HTTPS应用

网站安全标志和网页搜索排名



现代浏览器会给HTTPS网站添加**安全标志**，而HTTP网站则会添加**不安全标志**。而针对HTTPS网站内容，搜索引擎会提高网站在同类行业中的**排名优先级**，提高站点的**可信度**、**品牌形象**。

杜绝中间人流量劫持

在浏览HTTP网站时，可能会被中间人**劫持流量**（如运营商），导致网页被插入很多**广告**。

苹果App Store应用强制HTTPS

苹果宣布 2017 年 1 月 1 日起，所有提交到 App Store 的 App 必须强制开启 ATS 安全标准(App Transport Security)，所有连接**必须使用 HTTPS** 加密。

密码学基础

Hash算法

公式

摘要/散列/指纹 = hash (消息)

特点

1. **相同**消息得到**相同**摘要值
2. 不管消息长度，摘要值**长度固定**
3. 不管消息长度，**运算快速**
4. **不可逆**，很难逆向通过摘要计算原始消息
5. 原始消息**一旦修改**，即时微小修改，摘要值也会**变化**

常见算法

MD5、SHA系列

作用

消息**防篡改**

密码学基础

MAC(Message Authentication Code) 算法

公式

MAC值 = mac (消息，密钥)

特点

相对于hash算法，需要额外提供一个**会话密钥**，来表示**消息是特定人发送**的

常见算法

HMAC（Hash-based MAC）

作用

消息**防篡改**、**消息验证**（防止中间人攻击）

密码学基础

对称加密算法

公式

密文 = Encrypt (明文，算法，密钥)

明文 = Decrypt (密文，算法，密钥)

特点

1. 加密和解密使用**同一个密钥**
2. 加密和解密为**互逆过程**

常见算法

AES、DES

作用

消息**机密性**

密码学基础

公开密钥/非对称加密算法

公式

密文1 = Encrypt (明文，算法，公钥)　密文2 = Encrypt (明文，算法，私钥)
明文 = Decrypt (密文1，算法，私钥)　明文 = Decrypt (密文2，算法，公钥)

特点

1. 密钥为**一对**，分别为公钥和私钥
2. 如果使用**公钥加密**，则需要**私钥解密**；如果使用**私钥加密**，则需要**公钥解密**
3. 运算**速度慢**

常见算法

RSA

作用

消息**机密性**（常用于密钥协商、数字签名）

数字签名和证书

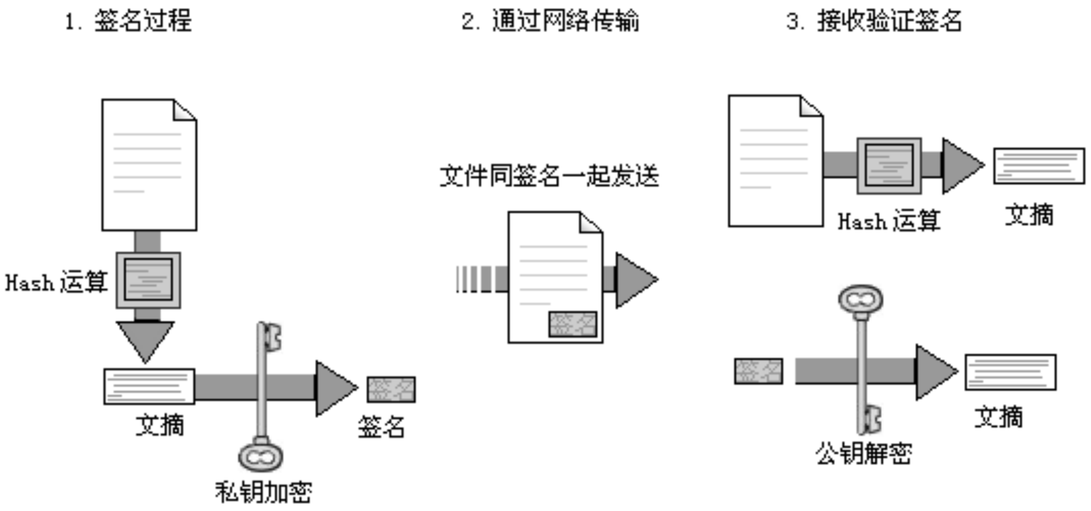
数字签名

公式

签名值 = Encrypt (hash (消息)，算法，私钥)

特点

1. 基于非对称加密算法
2. 用**私钥加密**的消息称为**签名**，只有拥有私钥的用户可以生成签名
3. 用**公钥解密**签名这一步称为**验证签名**，所有用户都可以验证签名
4. 一般不直接对消息进行签名，而是对消息的**哈希值进行签名**



验证签名

1. 用**公钥对签名进行解密**，得到hash值1
2. 对消息中的**正文进行hash计算**，得到hash值2
3. **比较**hash值1和hash值2，如果相同，则验证成功

作用

消息**防篡改**、**身份验证**

数字签名和证书

证书

为解决**安全分发公钥**问题，对公钥**合法性**提供证明的技术

特点

1. 基于**数字签名**
2. 对**公钥**的数字签名
3. 由**权威机构CA**签发
4. 证书包含**服务器信息**、**公钥**、**签名**

作用

消息**防篡改**、**身份验证**（防止中间人替换公钥）

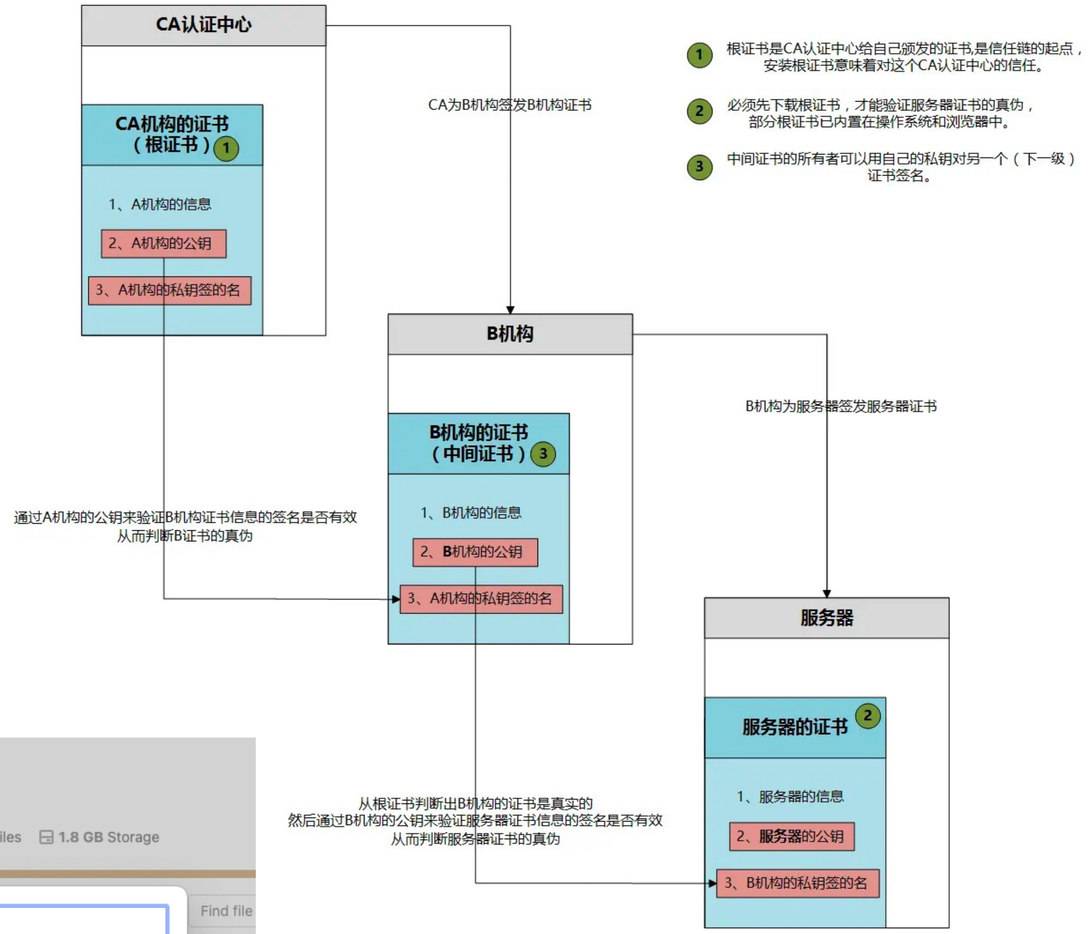
数字签名和证书

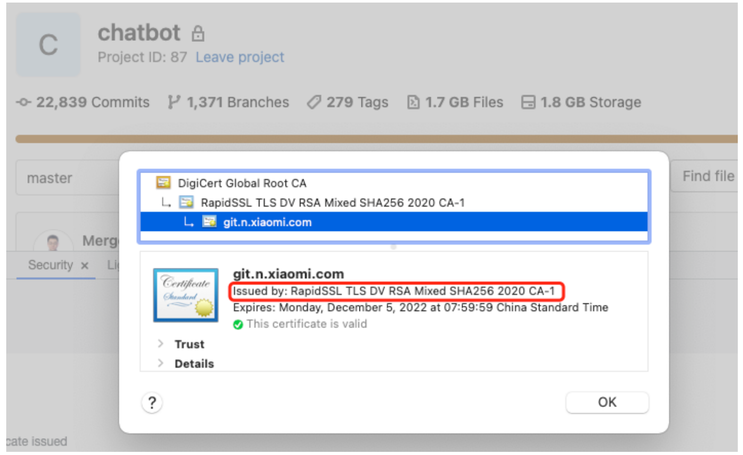
证书链

根证书 -> 中间证书 -> 服务器实体证书

特点
1. 基于**信任CA机构**前提
2. 根证书是**自签名**（issuer和subject相同）证书
3. 非根证书由**上一级证书签发**
4. 根证书**内置**于操作系统和浏览器
5. 证书链构成**信任链**

证书验证
1. 根证书公钥验证中间证书
2. 中间证书公钥验证服务器实体证书





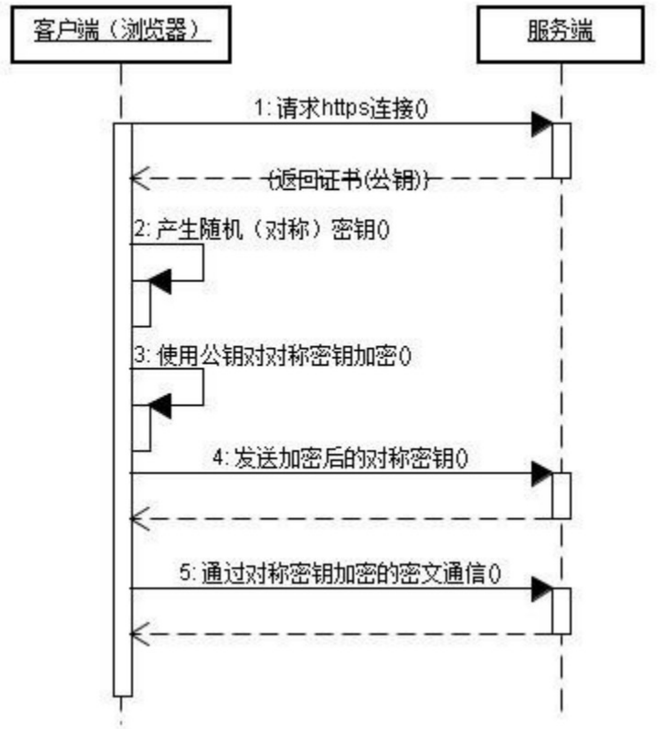
SSL/TLS协议

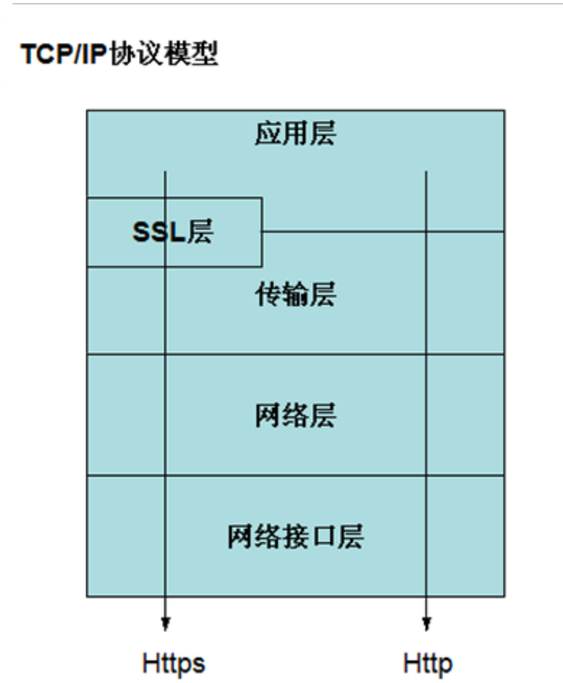
SSL/TLS协议历史

SSL（Secure Sockets Layer）协议是由网景浏览器1994年推出，作为浏览器扩展用于解决HTTP加密问题。之后IETF组织将SSL进行标准化，并更名为TLS（Transport Layer Security）协议。因此，**TLS可以理解为SSL的后续版本**。

TLS协议在TCP/IP协议栈中的定位

1. TLS协议介于应用层和传输层**之间**
2. HTTPS基于TLS协议





HTTPS宏观理解

1. **验证证书**
2. 通过非对称加密**协商出对称密钥**
3. 基于**对称密钥进行加密**通信

SSL/TLS协议

TLS协议整体理解

1. **二进制**协议
2. 下层为记录协议，负责对消息进行**加密解密**和**传输**
3. 上层协议基于记录协议进行传输
4. 握手协议，负责连接初期**协商出对称密钥**
5. 密码规格变更协议，负责**通知**记录协议是否开启加密传输
6. 警告协议，负责传达警告，如证书不合法
7. 应用数据协议，负责**传输应用数据**，如HTTP报文

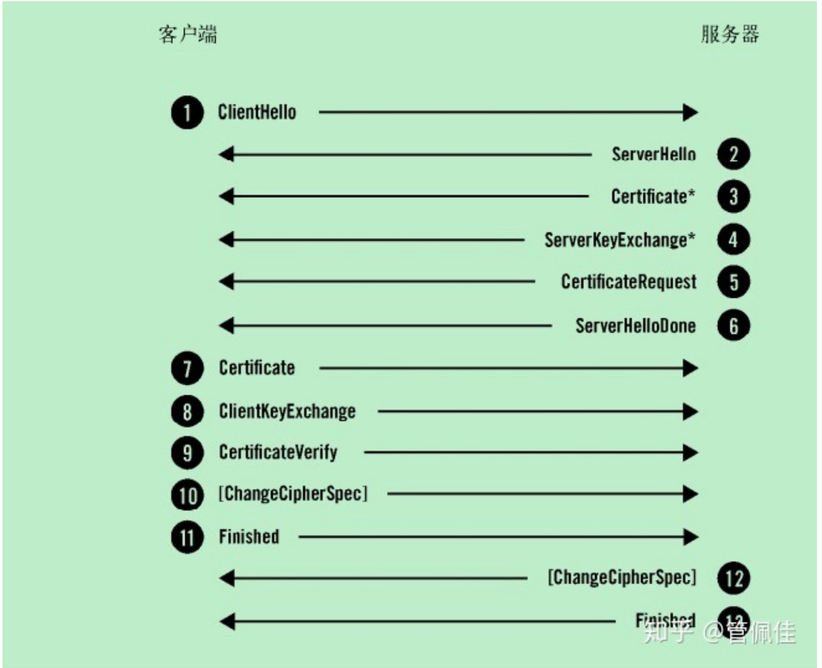
SSL/TLS协议

握手协议

1. **交换证书**进行身份验证
2. **交换随机数**（客户端和服务端各随机生成一个）
3. 生成**pre-master密钥**（随机数）并用**公钥加密**传输
4. 通过客户端随机数、服务端随机数和pre-master密钥三个随机数**推出master密钥**
5. 通过客户端随机数、服务端随机数和master密钥三个随机数**推出key block密钥**（最终对称加密密钥）

实现

通过对Socket API accept进行**封装**，将握手细节隐藏在内部




```c
/*****************************************************************************
 * socket相关函数包装
 *****************************************************************************/
int tls_accept(int server_sockfd) {
    // 接收客户端连接
    struct sockaddr_in client_addr;
    socklen_t client_len = sizeof(client_addr);
    int client_sockfd = accept(server_sockfd, (struct sockaddr *) &client_addr, &client_len);
    if (client_sockfd < 0) {
        perror("accept conn failed.\n");
        exit(1);
    }

    // 创建并初始化tls context
    tls_context_t *context = tls_context_init();
    context->client_sockfd = client_sockfd;

    // 执行 tls 握手
    if (tls_handshake(context)) {
        perror("tls handshake failed.\n");
        exit(1);
    }
    printf("Accept a new connection from %s:%d\n", inet_ntoa(client_addr.sin_addr), ntohs( x: client_addr.sin_port));
    return client_sockfd;
}
```

SSL/TLS协议

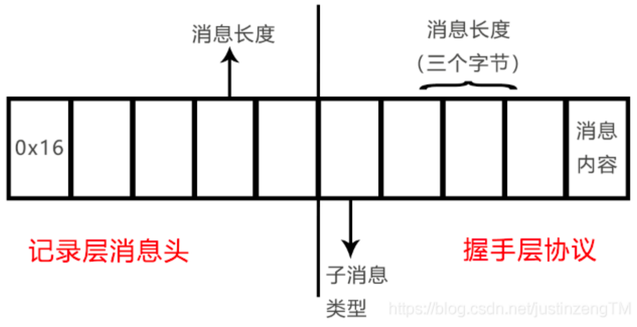
握手步骤1 - ClientHello

客户端发送

1. 客户端**随机数**
2. 客户端支持的**密钥套件列表**

服务端接收

1. 保存客户端随机数
2. 从密钥套件列表**选择一个密钥套件**



https://blog.csdn.net/justinzengTM

```
Frame 195: 571 bytes on wire (4568 bits), 571 bytes captured (4568 bits) on interface en0, id 0
Ethernet II, Src: Apple_03:c0:96 (3c:06:30:03:c0:96), Dst: NewH3CTe_66:68:01 (34:6b:5b:66:68:01)
Internet Protocol Version 4, Src: 10.224.223.58, Dst: 120.92.65.45
Transmission Control Protocol, Src Port: 60470, Dst Port: 443, Seq: 1, Ack: 1, Len: 517
Transport Layer Security
  TLSv1.2 Record Layer: Handshake Protocol: Client Hello
    Content Type: Handshake (22)
    Version: TLS 1.0 (0x0301)
    Length: 512
    Handshake Protocol: Client Hello
      Handshake Type: Client Hello (1)
      Length: 508
      Version: TLS 1.2 (0x0303)
      Random: 397683423b228651e588e6a41383afbe14c9df2b1a1f0e517ec1f95c0c44820d
      Session ID Length: 32
      Session ID: 943d42f6ff470cd10014995e813e1350ea28b2fca9df14b5753ee7ae661c1974
      Cipher Suites Length: 36
      Cipher Suites (18 suites)
      Compression Methods Length: 1
      Compression Methods (1 method)
      Extensions Length: 399
      Extension: Reserved (GREASE) (len=0)
      Extension: server_name (len=22)
      Extension: extended_master_secret (len=0)
```

```
0030  10 00 24 11 00 00 16 03  01 02 00 01 00 01 fc 03   ··$·····  ········
0040  03 39 76 83 42 3b 22 86  51 e5 88 e6 a4 13 83 af   ·9v·B;"·  Q·······
0050  be 14 c9 df 2b 1a 1f 0e  51 7e c1 f9 5c 0c 44 82   ····+···  Q~··\·D·
0060  0d 20 94 3d 42 f6 ff 47  0c d1 00 14 99 5e 81 3e   · ·=B··G  ·····^·>
0070  13 50 ea 28 b2 fc a9 df  14 b5 75 3e e7 ae 66 1c   ·P·(····  ··u>··f·
0080  19 74 00 24 5a 5a 13 01  13 02 13 03 c0 2c c0 2b   ·t·$ZZ··  ·····,·+
0090  cc a9 c0 30 c0 2f cc a8  c0 24 c0 23 c0 0a c0 09   ···0·/··  ·$·#····
00a0  c0 28 c0 27 c0 14 c0 13  01 00 01 8f 1a 1a 00 00   ·(·'····  ········
00b0  00 00 00 16 00 14 00 00  11 61 70 6d 2e 66 2e 6d   ·········  ·apm.f.m
00c0  69 6f 66 66 69 63 65 2e  63 6e 00 17 00 00 ff 01   ioffice.  cn······
00d0  00 01 00 00 0a 00 0c 00  0a ca ca 00 1d 00 17 00   ········  ········
00e0  18 00 19 00 0b 00 02 01  00 00 10 00 0b 00 09 08   ········  ········
00f0  68 74 74 70 2f 31 2e 31  00 05 00 05 01 00 00 00   http/1.1  ········
0100  00 00 0d 00 18 00 16 04  03 08 04 04 01 05 03 02   ········  ········
0110  03 08 05 08 05 05 01 08  06 06 01 02 01 00 12 00   ········  ········
0120  00 00 33 00 2b 00 29 ca  ca 00 01 00 00 1d 00 20   ··3·+·)·  ·······
```

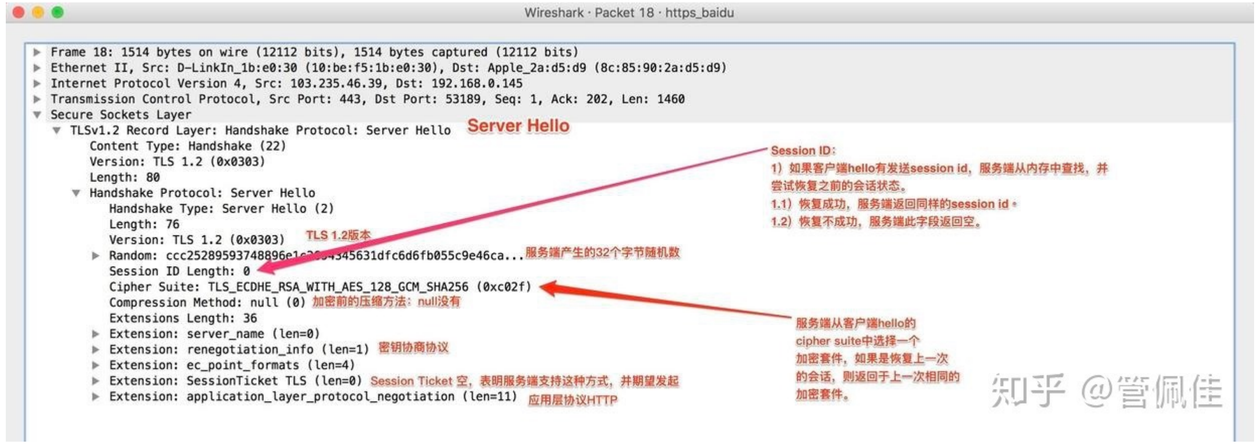
```
Frame 16: 255 bytes on wire (2040 bits), 255 bytes captured (2040 bits)
Ethernet II, Src: Apple_2a:d5:d9 (8c:85:90:2a:d5:d9), Dst: D-LinkIn_1b:e0:30
Internet Protocol Version 4, Src: 192.168.0.145, Dst: 103.235.46.39
Transmission Control Protocol, Src Port: 53189, Dst Port: 443, Seq: 1, Ack: 1, Len: 201
Secure Sockets Layer                                  安全套接字层数据包
  TLSv1.2 Record Layer: Handshake Protocol: Client Hello
    Content Type: Handshake (22)
    Version: TLS 1.0 (0x0301)
    Length: 196
    Handshake Protocol: Client Hello                  握手协议：Client Hello
      Handshake Type: Client Hello (1)
      Length: 192
      Version: TLS 1.2 (0x0303)                       TLS 1.2版本
      Random: ea100ce266a198303aab77fda15c4a6f46158a4ef558728e...  由客户端生成的32个字节的安全随机数
      Session ID Length: 0
      Cipher Suites Length: 28
      Cipher Suites (14 suites) 客户端也就是浏览器支持的加密算法
      Compression Methods Length: 1
      Compression Methods (1 method)  用于加密前的压缩方法
      Extensions Length: 123
      Extension: Reserved (GREASE) (len=0)  保留字段
      Extension: renegotiation_info (len=1) 密钥协商协议
      Extension: server_name (len=18)  请求的域名信息
      Extension: extended_master_secret (len=0)
      Extension: SessionTicket TLS (len=0)  服务端支持session ticket机制的话，将会把session ticket返回，此数据存储在客户端
      Extension: signature_algorithms (len=20)  客户端支持的签名hash算法
      Extension: status_request (len=5)
      Extension: signed_certificate_timestamp (len=0) 签名证书时间戳
      Extension: application_layer_protocol_negotiation (len=14) 应用层协议HTTP
      Extension: channel_id (len=0)
      Extension: ec_point_formats (len=2)
      Extension: supported_groups (len=10)
      Extension: Reserved (GREASE) (len=1)

session id：是被服务端定义的，如果客户端hello消息中的session id不为空，
说明两端曾经握手成功过。服务端将以前协商好的信息存储起来，快速进行握手过程。
session id 为空，说明没有记录的会话，需要完整的进行握手。

每个加密组件(Cipher Suite)都包括了下面5类算法：
1、authentication (认证算法)
2、encryption (加密算法 )
3、message authentication code (消息认证码算法 简称MAC)
4、key exchange (密钥交换算法)
5、key derivation function （密钥衍生算法）

0030  20 00 31 7d 00 00 16 03  01 00 c4 01 00 00 c0 03   .1}..... ........
0040  03 ea 10 0c e2 66 a1 98  30 3a ab 77 fd a1 5c 4a   .....f.. 0:.w..\J
```

SSL/TLS协议

握手步骤2 - ServerHello

服务端发送
1. 服务端**随机数**
2. 选择的**密钥套件**

客户端接收
1. 保存服务端随机数
2. 通过密钥套件，知道本次非对称加密和对称加密等**算法类型**

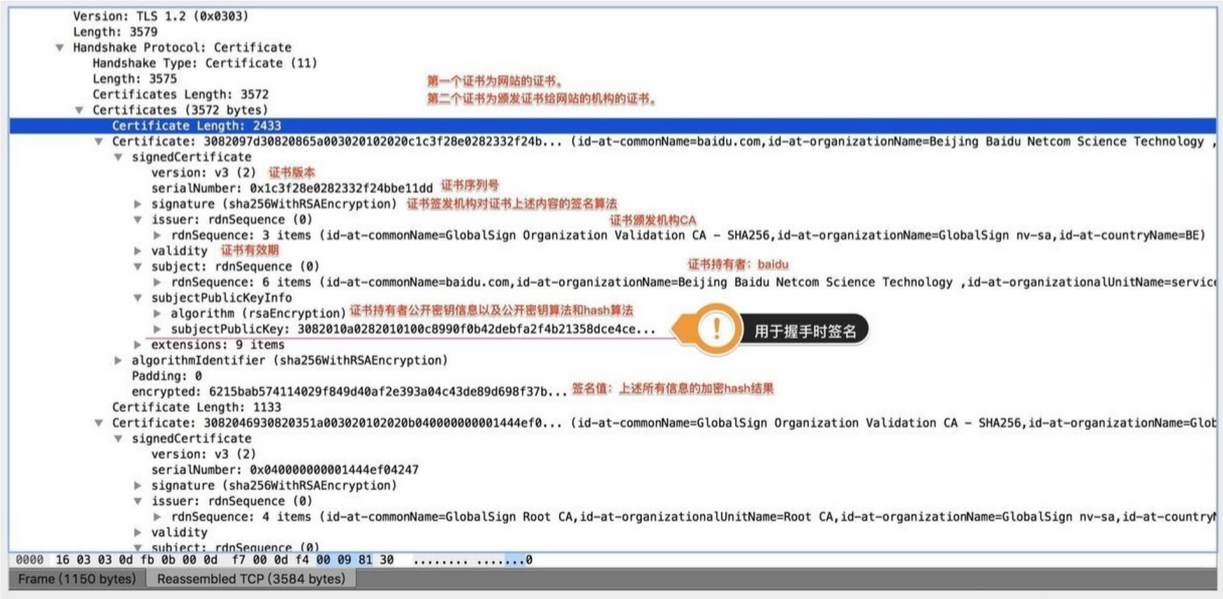


SSL/TLS协议

握手步骤3 - ServerCertificate

服务端发送
证书链（一般只用发中间证书和服务器实体证书，不用发根证书）

客户端接收
1. **验证证书**链
2. 从服务器实体证书中获取服务器**公钥**



SSL/TLS协议

握手步骤4 - ServerHelloDone

服务端发送
空消息，表明服务端发送完毕

客户端接收
得知服务器发送完毕，客户端开始发送消息

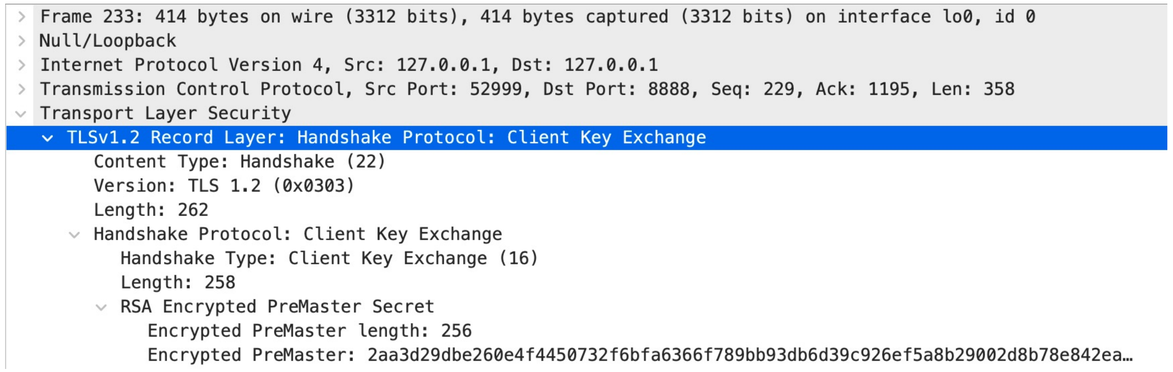
```
> Frame 201: 94 bytes on wire (752 bits), 94 bytes captured (752 bits) on interface en0, id 0
> Ethernet II, Src: NewH3CTe_66:68:01 (34:6b:5b:66:68:01), Dst: Apple_03:c0:96 (3c:06:30:03:c0:96)
> Internet Protocol Version 4, Src: 120.92.65.45, Dst: 10.224.223.58
> Transmission Control Protocol, Src Port: 443, Dst Port: 60470, Seq: 5553, Ack: 518, Len: 40
> [2 Reassembled TCP Segments (338 bytes): #200(307), #201(31)]
> Transport Layer Security
v Transport Layer Security
  v TLSv1.2 Record Layer: Handshake Protocol: Server Hello Done
      Content Type: Handshake (22)
      Version: TLS 1.2 (0x0303)
      Length: 4
    > Handshake Protocol: Server Hello Done
```

SSL/TLS协议

握手步骤5 - ClientKeyExchange

客户端发送
随机生成**pre-master密钥**，并通过服务端**公钥进行加密**

服务端接收
1. 通过**私钥解密**得到pre-master密钥
2. 通过客户端随机数、服务端随机数和pre-master密钥**推出master密钥**



密码规格变更协议 - ClientChangeCipherSpec

客户端发送
通知服务端，后续数据开始加密传输

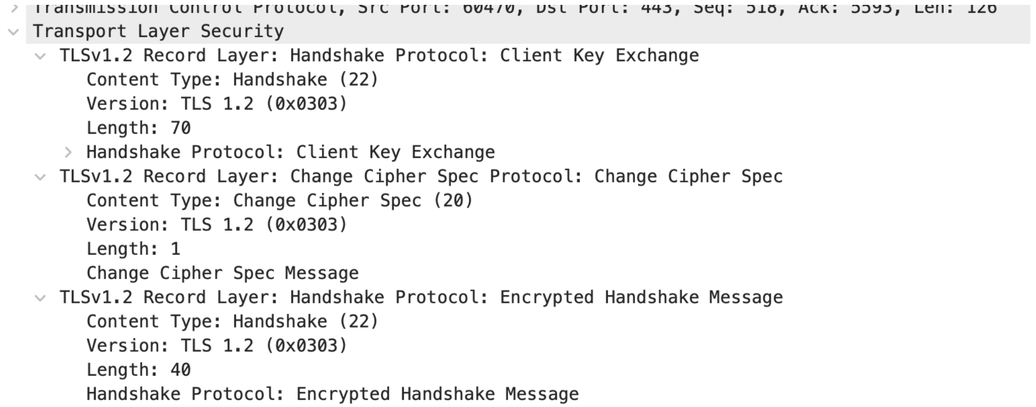
服务端接收
后续客户端传输内容需要进行解密

SSL/TLS协议

握手步骤6 - ClientFinished

客户端发送
1. 对**前面所有握手消息**进行hash运算得到hash值
2. 通过master-secret密钥对hash值加密，得到verify_data
3. 加密传输verify_data

服务端接收
验证verify_data，确保前面握手消息**没被篡改**



SSL/TLS协议

密码规格变更协议 - ServerChangeCipherSpec

服务端发送
通知客户端，后续数据开始加密传输

客户端接收
后续服务端传输内容需要进行解密

```
> Frame 205: 105 bytes on wire (840 bits), 105 bytes captured (840 bits) on interface en0, id 0
> Ethernet II, Src: NewH3CTe_66:68:01 (34:6b:5b:66:68:01), Dst: Apple_03:c0:96 (3c:06:30:03:c0:96)
> Internet Protocol Version 4, Src: 120.92.65.45, Dst: 10.224.223.58
> Transmission Control Protocol, Src Port: 443, Dst Port: 60470, Seq: 5593, Ack: 644, Len: 51
v Transport Layer Security
  v TLSv1.2 Record Layer: Change Cipher Spec Protocol: Change Cipher Spec
      Content Type: Change Cipher Spec (20)
      Version: TLS 1.2 (0x0303)
      Length: 1
      Change Cipher Spec Message
  v TLSv1.2 Record Layer: Handshake Protocol: Encrypted Handshake Message
      Content Type: Handshake (22)
      Version: TLS 1.2 (0x0303)
      Length: 40
      Handshake Protocol: Encrypted Handshake Message
```

握手步骤7 - ServerFinished

服务端发送
1. 对**前面所有握手消息**进行hash运算得到hash值
2. 通过master-secret密钥对hash值加密，得到verify_data
3. 加密传输verify_data

客户端接收
验证verify_data，确保前面握手消息**没被篡改**

SSL/TLS协议

应用数据协议

1. 应用数据通过记录协议传输，由记录层加密解密（握手阶段协商出对称密钥），上层无感知
2. 通过对Socket API **recv和send封装**，将加密解密细节隐藏在内部

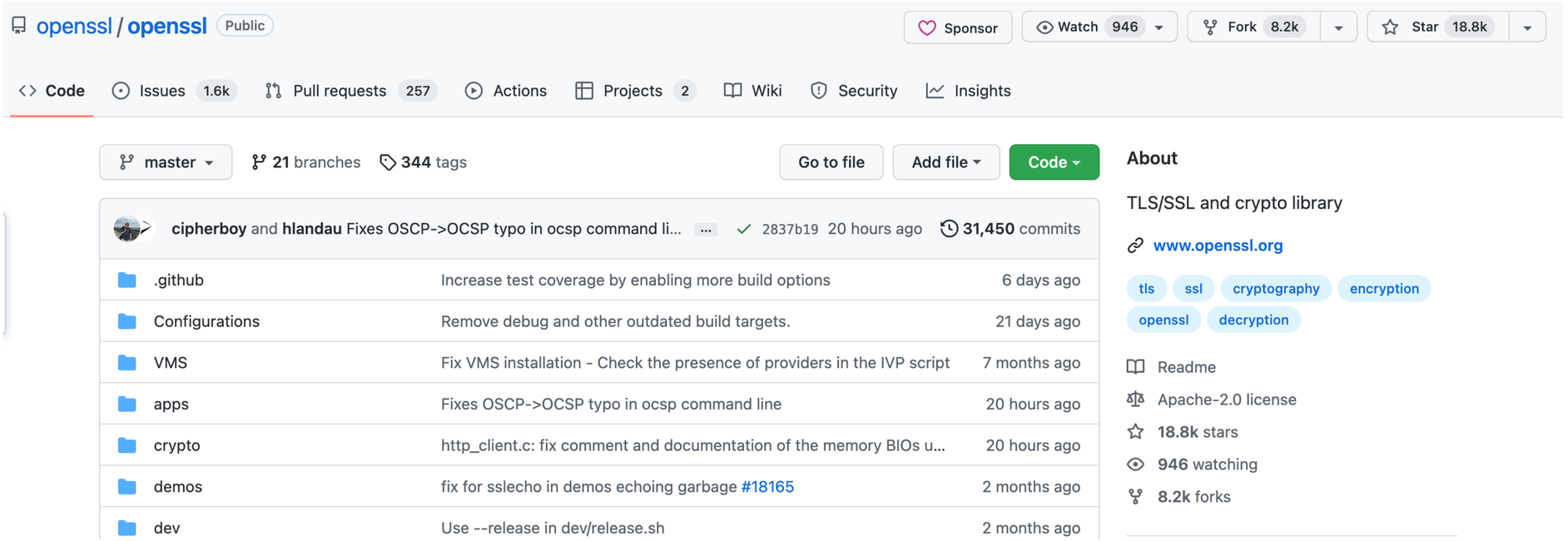
```
> Frame 207: 568 bytes on wire (4544 bits), 568 bytes captured (4544 bits) on interface en0, id 0
> Ethernet II, Src: Apple_03:c0:96 (3c:06:30:03:c0:96), Dst: NewH3CTe_66:68:01 (34:6b:5b:66:68:01)
> Internet Protocol Version 4, Src: 10.224.223.58, Dst: 120.92.65.45
> Transmission Control Protocol, Src Port: 60470, Dst Port: 443, Seq: 644, Ack: 5644, Len: 514
v Transport Layer Security
  v TLSv1.2 Record Layer: Application Data Protocol: http-over-tls
      Content Type: Application Data (23)
      Version: TLS 1.2 (0x0303)
      Length: 509
      Encrypted Application Data: 00000000000000001187b8eab9a6c03d28f6b02a4dec70ae0cd3e52c3deb464af34c5e2e4…
      [Application Data Protocol: http-over-tls]
```

```
// socket相关包装函数
int tls_accept(int server_sockfd);
char *tls_recv(int client_sockfd);
int tls_send(int client_sockfd, char *msg);
```

SSL/TLS协议

OpenSSL

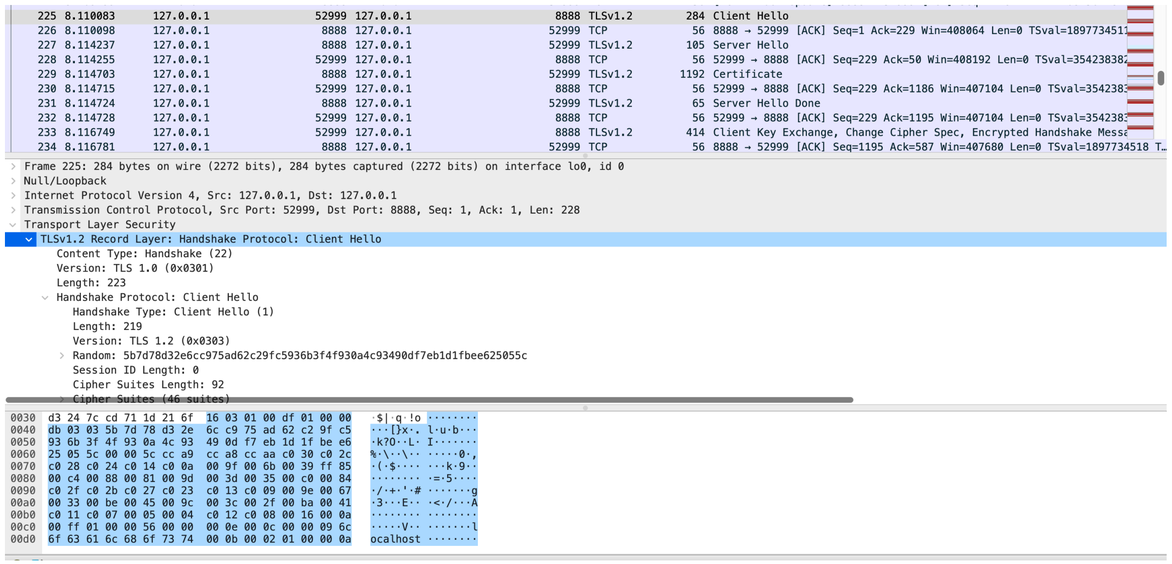
1. 是大家共同努力开发出的代码可靠、功能齐全、商业级别的开源工具集
2. 主要包含两部分：**密码学库**和**TLS协议实现**
3. 内置于Linux等操作系统



SSL/TLS协议

测试

1. 通过OpenSSL生成自签名证书（直接作为服务器证书）
2. 通过curl调试
3. 通过wireshark调试




```
$ curl --cacert cert/certificate.pem "https://localhost:8888/index.html" --verbose
*   Trying ::1:8888...
* connect to ::1 port 8888 failed: Connection refused
*   Trying 127.0.0.1:8888...
* Connected to localhost (127.0.0.1) port 8888 (#0)
* ALPN, offering h2
* ALPN, offering http/1.1
* successfully set certificate verify locations:
*  CAfile: cert/certificate.pem
*  CApath: none
* TLSv1.2 (OUT), TLS handshake, Client hello (1):
* TLSv1.2 (IN), TLS handshake, Server hello (2):
* TLSv1.2 (IN), TLS handshake, Certificate (11):
* TLSv1.2 (IN), TLS handshake, Server finished (14):
* TLSv1.2 (OUT), TLS handshake, Client key exchange (16):
* TLSv1.2 (OUT), TLS change cipher, Change cipher spec (1):
* TLSv1.2 (OUT), TLS handshake, Finished (20):
```

```
$ curl --cacert cert/certificate.pem "https://localhost:8888/index.html" --trace -
== Info:   Trying ::1:8888...
== Info: connect to ::1 port 8888 failed: Connection refused
== Info:   Trying 127.0.0.1:8888...
== Info: Connected to localhost (127.0.0.1) port 8888 (#0)
== Info: ALPN, offering h2
== Info: ALPN, offering http/1.1
== Info: successfully set certificate verify locations:
== Info:  CAfile: cert/certificate.pem
== Info:  CApath: none
== Info: TLSv1.2 (OUT), TLS handshake, Client hello (1):
=> Send SSL data, 223 bytes (0xdf)
0000: 01 00 00 db 03 03 06 9e 3e 7a 2c bb 8b e7 71 cd ........>z,...q.
0010: fe fc 18 30 ba 01 97 de d2 c5 39 bb 06 9b de 85 ...0......9.....
0020: 1a c8 b8 0b 68 a5 00 00 5c cc a9 cc a8 cc aa c0 ....h...\.......
0030: 30 c0 2c c0 28 c0 24 c0 14 c0 0a 00 9f 00 6b 00 0.,.(.$.......k.
0040: 39 ff 85 00 c4 00 88 00 81 00 9d 00 3d 00 35 00 9...........=.5.
0050: c0 00 84 c0 2f c0 2b c0 27 c0 23 c0 13 c0 09 00 ..../.+.'.#.....
0060: 9e 00 67 00 33 00 be 00 45 00 9c 00 3c 00 2f 00 ..g.3...E...<./.
0070: ba 00 41 c0 11 c0 07 00 05 00 04 c0 12 c0 08 00 ..A.............
0080: 16 00 0a 00 ff 01 00 00 56 00 00 00 0e 00 0c 00 ........V.......
0090: 00 09 6c 6f 63 61 6c 68 6f 73 74 00 0b 00 02 01 ..localhost.....
00a0: 00 00 0a 00 08 00 06 00 1d 00 17 00 18 00 0d 00 ................
00b0: 1c 00 1a 06 01 06 03 ef ef 05 01 05 03 04 01 04 ................
00c0: 03 ee ee ed ed 03 01 03 03 02 01 02 03 00 10 00 ................
00d0: 0e 00 0c 02 68 32 08 68 74 74 70 2f 31 2e 31    ....h2.http/1.1
== Info: TLSv1.2 (IN), TLS handshake, Server hello (2):
<= Recv SSL data, 44 bytes (0x2c)
0000: 02 00 00 28 03 03 62 c6 53 43 31 f5 bc 24 96 71 ...(..b.SC1..$.q
0010: 2d 12 13 65 1c a6 fc b1 e3 18 e8 28 52 23 85 96 -..e.......(R#..
0020: d4 76 d9 e9 e3 f8 00 00 3d 00 00 00             .v......=...
== Info: TLSv1.2 (IN), TLS handshake, Certificate (11):
<= Recv SSL data, 1131 bytes (0x46b)
0000: 0b 00 04 67 00 04 64 00 04 61 30 82 04 5d 30 82 ...g..d..a0..]0.
0010: 03 45 a0 03 02 01 02 02 09 00 90 25 c5 9b 59 53 .E.........%..YS
0020: 6a 34 30 0d 06 09 2a 86 48 86 f7 0d 01 01 0b 05 j40...*.H.......
0030: 00 30 7b 31 0b 30 09 06 03 55 04 06 13 02 43 4e .0{1.0...U....CN
```

总结和启发

1. **安全大于性能**（握手需要产生三个随机数来推导出对称密钥）
2. 二进制协议解析需要**按字节解析**，**效率更高**

参考资料

1.《HTTP权威指南》
2.《深入浅出HTTPS：从原理到实战》
3.《深入剖析Tomcat》
4. RFC 2616 HTTP/1.1协议
5. RFC 3875 CGI 1.1 协议
6. RFC 5246 TLS 1.2 协议

谢谢！


小米集团技术委员会
MITC
XIAOMI TECHNICAL COMMITTEE