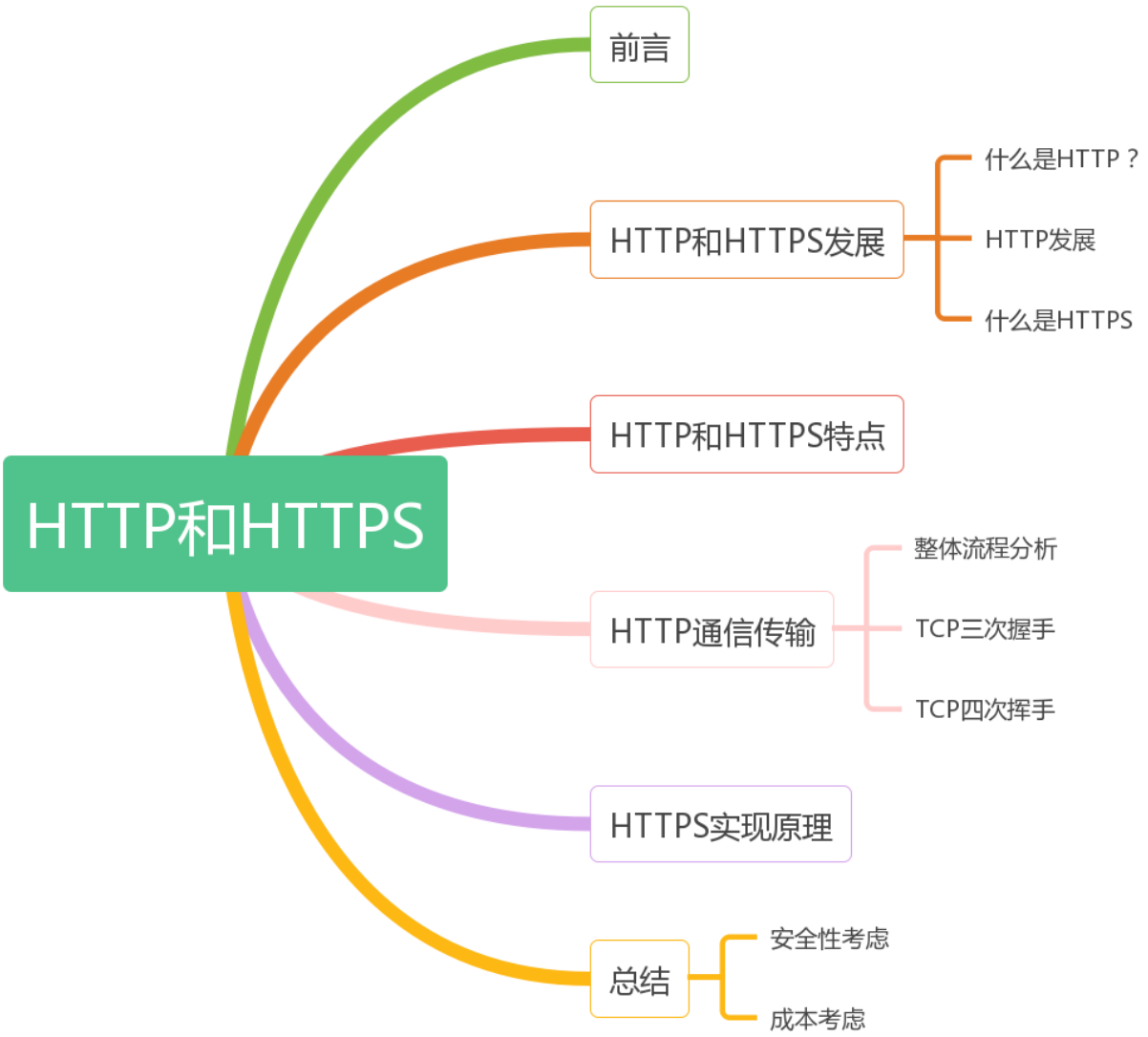
HTTP、HTTPS、WebSocket

1.HTTP



1.HTTP发展史

1.什么是HTTP

超文本传输协议，是一个基于请求与响应，无状态的，应用层的协议，常基于TCP/IP协议传输数据，互联网上应用最为广泛的一种网络协议,所有的WWW文件都必须遵守这个标准。设计HTTP的初衷是为了提供一种发布和接收HTML页面的方法

2.发展历史

版本	产生时间	内容	发展现状
HTTP/0.9	1991年	不涉及数据包传输，规定客户端和服务器之间通信格式，只能GET请求	没有作为正式的标准
HTTP/1.0	1996年	传输内容格式不限制，增加PUT、PATCH、HEAD、 OPTIONS、DELETE命令	正式作为标准
HTTP/1.1	1997年	持久连接(长连接)、节约带宽、HOST域、管道机制、分块传输编码	2015年前使用最广泛
HTTP/2	2015年	多路复用、服务器推送、头信息压缩、二进制协议等	逐渐覆盖市场

2.HTTP特点：

- 无状态：协议对客户端没有状态存储，对事物处理没有“记忆”能力，比如访问一个网站需要反复进行登录操作
- 无连接：HTTP/1.1之前，由于无状态特点，每次请求需要通过TCP三次握手四次挥手，和服务器重新建立连接。比如某个客户机在短时间多次请求同一个资源，服务器并不能区别是否已经响应过用户的请求，所以每次需要重新响应请求，需要耗费不必要的时间和流量。
- 基于请求和响应：基本的特性，由客户端发起请求，服务端响应
- 简单快速、灵活
- 通信使用明文、请求和响应不会对通信方进行确认、无法保护数据的完整性

3. HTTP消息结构

一个HTTP请求报文由请求行（request line）、请求头（header）、空行和请求数据4个部分组成，下图给出了请求报文的一般格式

1.请求行

请求行由请求方法字段、URL字段和HTTP协议版本字段3个字段组成，它们用空格分隔。例如，**GET /index.html HTTP/1.1**

2.请求头

请求头部由关键字/值对组成，每行一对，关键字和值用英文冒号“:”分隔。请求头部通知服务器有关于客户端请求的信息，典型的请求头有

常见的请求头属性

- **Accept :**指定客户端能够接收的内容类型 | Accept: text/plain, text/html
- **Cache-Control:** 指定请求和响应遵循的缓存机制 | Cache-Control: no-cache

- **Cookie :**HTTP请求发送时，会把保存在该请求域名下的所有cookie值一起发送给web服务器 | Cookie: $Version=1; Skin=new;
- **Content-Type:**请求的与实体对应的MIME信息 | Content-Type: application/x-www-form-urlencoded
- **Host:** 指定请求的服务器的域名和端口号
- **Referer:** 先前网页的地址，当前请求网页紧随其后,即来路
- **User-Agent :** 发起请求的用户的身份标识

3.空行

最后一个请求头之后是一个空行，发送回车符和换行符，通知服务器以下不再有请求头。

4.HTTP响应

HTTP状态码的英文为HTTP Status Code。状态代码由三位数字组成，第一个数字定义了响应的类别，且有五种可能取值。

- **1xx：**指示信息–表示请求已接收，继续处理。
- **2xx：**成功–表示请求已被成功接收、理解、接受。
- **3xx**：重定向–要完成请求必须进行更进一步的操作。
- **4xx：**客户端错误–请求有语法错误或请求无法实现。
- **5xx**：服务器端错误–服务器未能实现合法的请求。

常见的状态码

- **200 OK：**客户端请求成功,一般用于GET和POST请求
- **400 Bad Request：**客户端请求有语法错误，不能被服务器所理解。
- **301 Moved Permanently:**永久移动,请求的资源已被永久移动到新url,返回信息会包含新的url,浏览器会自动定向到新url
- **401 Unauthorized：**请求未经授权，这个状态代码必须和WWW-Authenticate报头域一起使用。
- **403 Forbidden：**服务器收到请求，但是拒绝提供服务。
- **404 Not Found：**请求资源不存在，举个例子：输入了错误的URL。
- **500 Internal Server Error：**服务器发生不可预期的错误。
- **502 Bad Gateway:** 充当网关或代理的服务器,从远端接收到一个无效的请求
- **503 Server Unavailable：**服务器当前不能处理客户端的请求，一段时间后可能恢复正常，举个例子：HTTP/1.1 200 OK（CRLF）

5.关于HTTP请求GET和POST的区别

1.两种提交方式

- **GET提交:**请求的数据会附在URL之后（就是把数据放置在HTTP协议头 < request-line > 中），以?分割URL和传输数据，多个参数用&连接;例如：login.action?name=hyddd&password=idontknow&verify=%E4%BD%A0%E5%A5%BD。如果数据是英文字母/数字，原样发送，如果是空格，转换为+，如果是中文/其他字符，则直接把字符串用BASE64加密，得出如： %E4%BD%A0%E5%A5%BD，其中%XX中的XX为该符号以16进制表示的ASCII。
- **POST提交：** 把提交的数据放置在是HTTP包的包体 < request-body > 中。

2.传输数据的大小：

首先声明,HTTP协议没有对传输的数据大小进行限制，HTTP协议规范也没有对URL长度进行限制。 而在实际开发中存在的限制主要有：

- **GET:**特定浏览器和服务器对URL长度有限制，例如IE对URL长度的限制是2083字节(2K+35)。对于其他浏览器，如Netscape、FireFox等，理论上没有长度限制，其限制取决于操作系统的支持。因此对于GET提交时，传输数据就会受到URL长度的限制
- **POST:** 由于不是通过URL传值，理论上数据不受限。但实际各个WEB服务器会规定对post提交数据大小进行限制，Apache、IIS6都有各自的配置。

3.安全性:

POST的安全性要比GET的安全性高。

注意：这里所说的安全性和上面GET提到的“安全”不是同个概念。上面“安全”的含义仅仅是不作数据修改，而这里安全的含义是真正的Security的含义，比如：通过GET提交数据，用户名和密码将明文出现在URL上，因为(1)登录页面有可能被浏览器缓存， (2)其他人查看浏览器的历史纪录，那么别人就可以拿到你的账号和密码了

2.HTTPS实现原理

> HTTPS是一种通过计算机网络进行安全通信的传输协议，经由HTTP进行通信，利用SSL/TLS建立全信道，加密数据包。
> HTTPS使用的主要目的是提供对网站服务器的身份认证，同时保护交换数据的隐私与完整性。
> PS:TLS是传输层加密协议，前身是SSL协议，由网景公司1995年发布，有时候两者不区分

1.对称秘钥加密

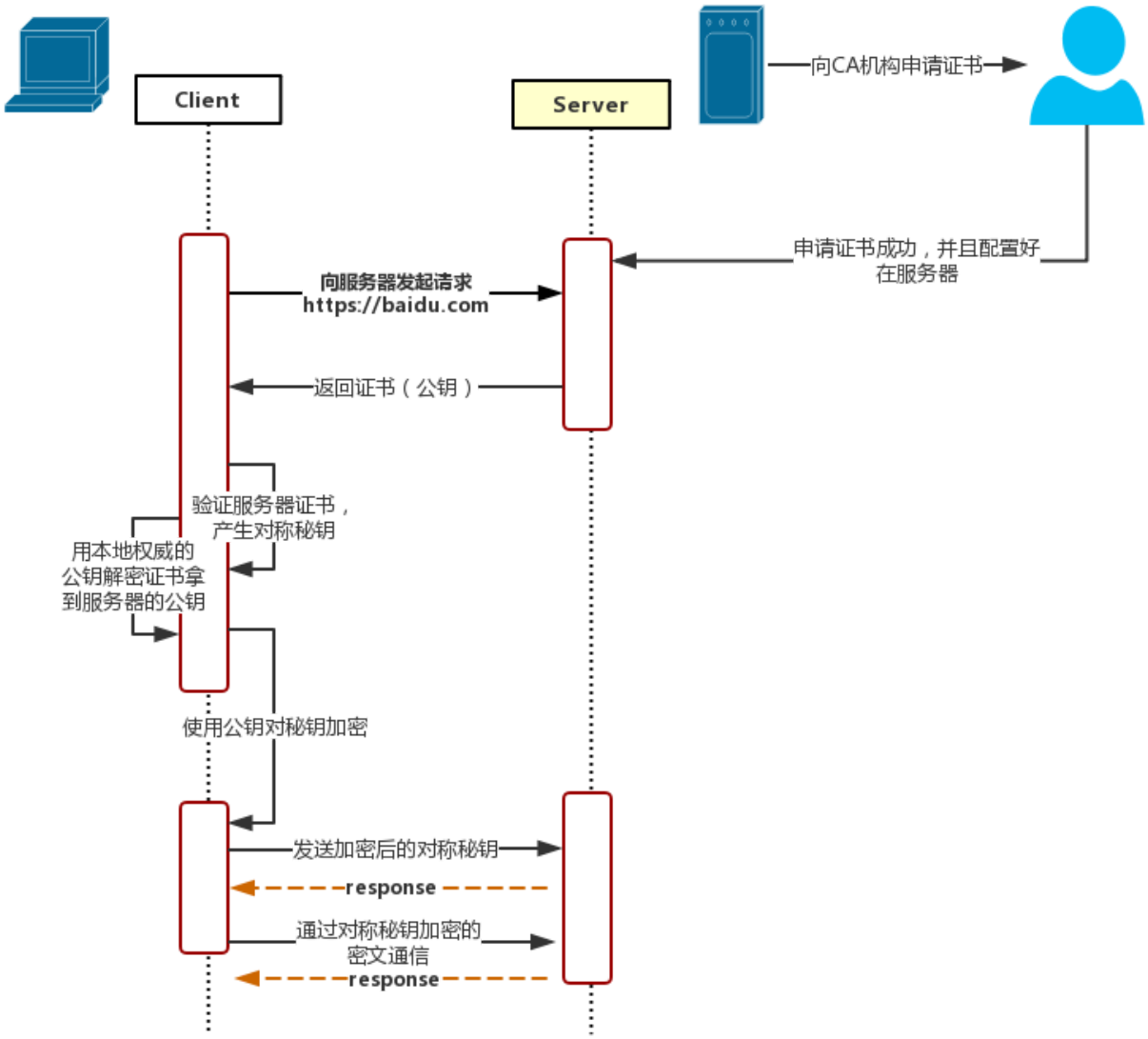
- 客户端自己封装一种加密算法，将给服务端发送的数据进行加密，并且将数据加密的方式即秘钥发送给密文，服务端收到秘钥和数据，用秘钥进行解密

2.非对称秘钥加密

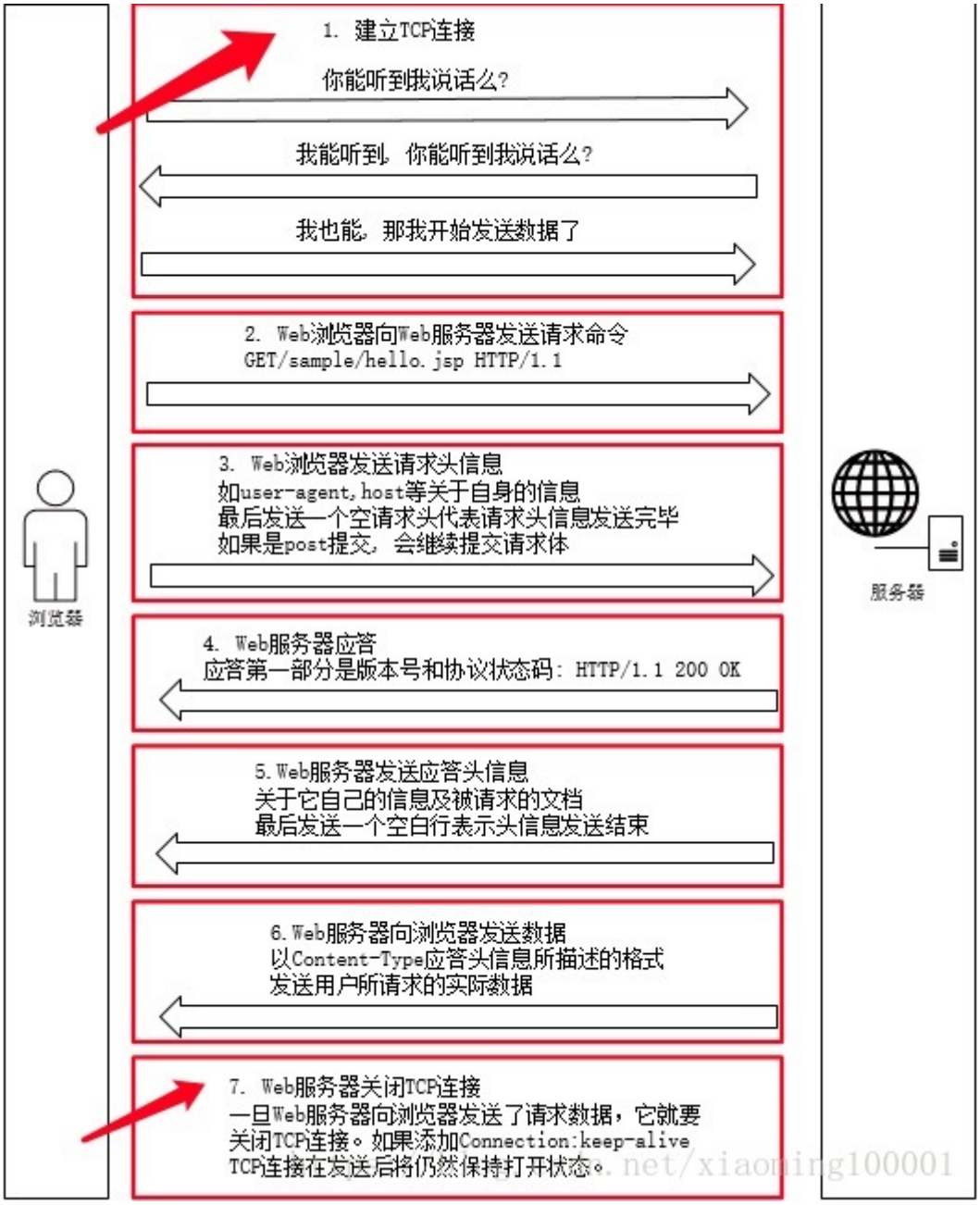
- 服务器端为客户端统一创建一个加密方式，由服务器端指定创建，称为公钥，服务器端所创建的加密方式统一的分发给即将要进行服务器连接的客户端，客户端只需要将密文发送给服务端，服务端通过公钥加密，但是的建立两次连接，先传送公钥，也存在隐患，模拟某些服务器的公钥，发送的数据被第三方截取。

3.证书秘钥加密

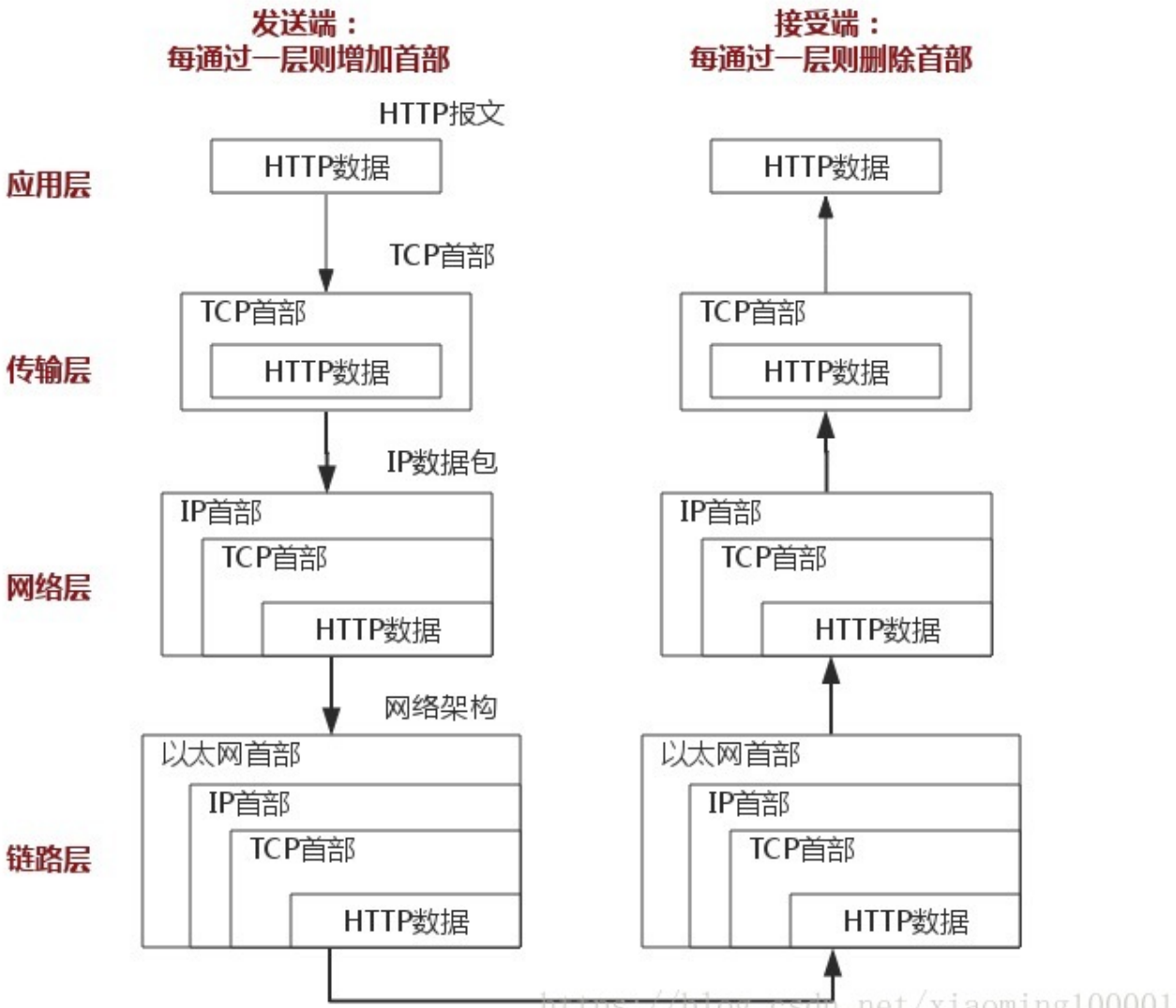
- 用到三方机构，数字证书，服务器端和客户端足以信任的机构，服务器端将公钥发送给三方机构，在三方机构做一个防伪标识，数字签名，公钥携带三方机构的证书发送给客户端，客户端使用公钥对数据进行加密。

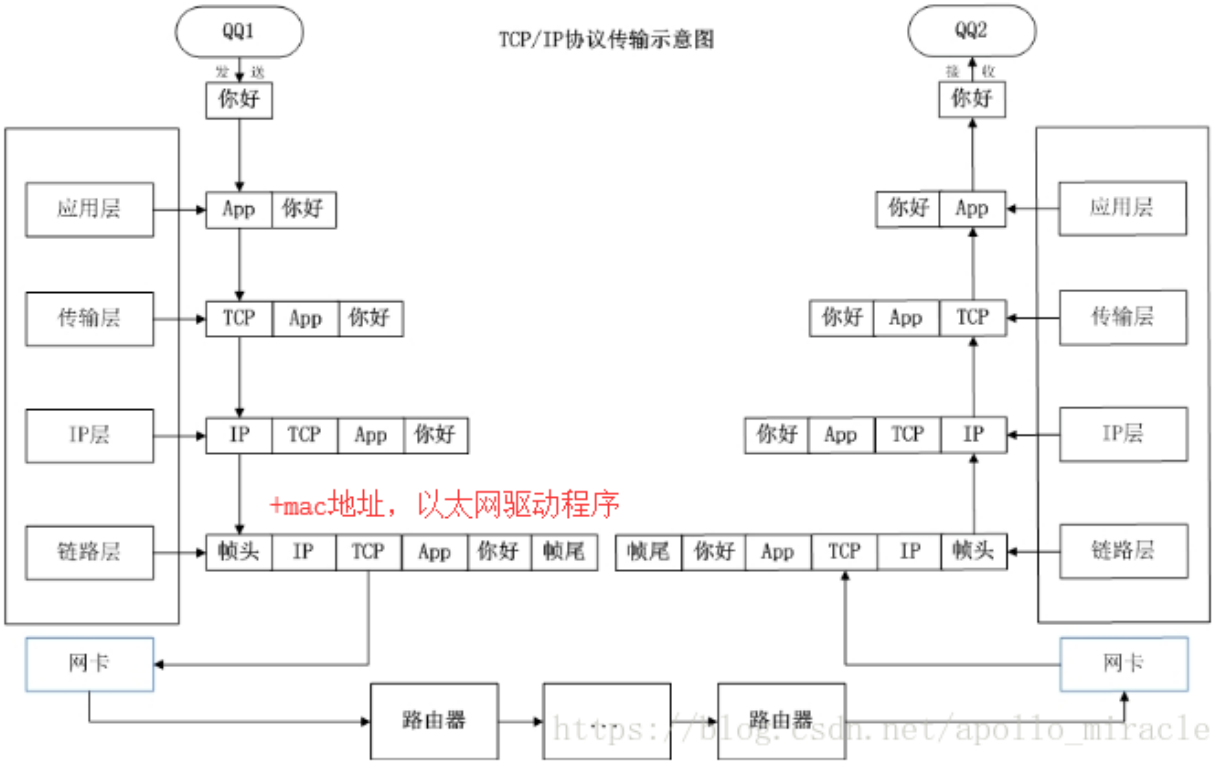

Client
Server
向CA机构申请证书
申请证书成功，并且配置好
在服务器
向服务器发起请求
https://baidu.com
返回证书（公钥）
验证服务器证书，
产生对称秘钥
用本地权威的
公钥解密证书拿
到服务器的公钥
使用公钥对秘钥加密
发送加密后的对称秘钥
response
通过对称秘钥加密的
密文通信
response

3.http通信原理



客户端输入URL回车，DNS解析域名得到服务器的IP地址，服务器在80端口监听客户端请求，端口通过TCP/IP协议（可以通过Socket实现）建立连接。HTTP属于TCP/IP模型中的运用层协议，所以通信的过程其实是对应数据的入栈和出栈



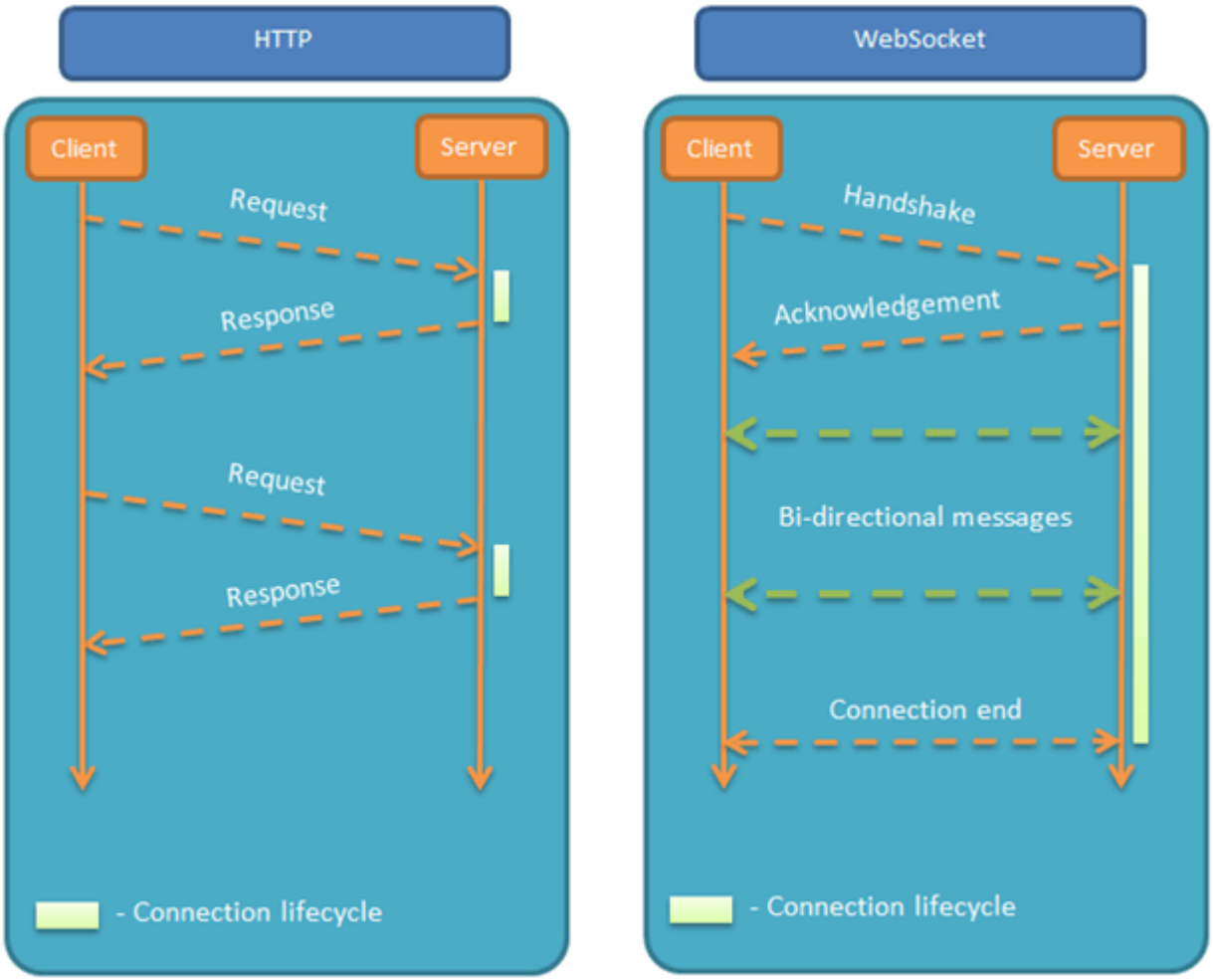


报文从运用层传送到运输层，运输层通过TCP三次握手和服务器建立连接，四次挥手释放连接。

4.WebSocket

1.WebSocket 与 HTTP

- WebSocket 的最大特点就是，服务器可以主动向客户端推送信息，客户端也可以主动向服务器发送信息，是全双工通信。
- TTP 有 1.1 和 1.0 之说，也就是所谓的 keep-alive ，把多个 HTTP 请求合并为一个，但是 Websocket 其实是一个新协议，跟 HTTP 协议基本没有关系，只是为了兼容现有浏览器，所以**在握手阶段使用了 HTTP**



2.WebSocket - 握手

WebSocket 是基于 HTTP 协议的，或者说借用了 HTTP 协议来完成一部分握手

```
GET /chatsocket HTTP/1.1
Host: 127.0.0.1:8002
Connection: Upgrade
Pragma: no-cache
Cache-Control: no-cache
Upgrade: websocket
Origin: http://localhost:63342
Sec-WebSocket-Version: 13                #版本
Sec-WebSocket-Key: mnwFxiOlctXFN/DeMt1Amg==
Sec-WebSocket-Extensions: permessage-deflate; client_max_window_bits
```

- **Sec-WebSocket-Key** 是一个 Base64 encode 的值，这个是浏览器随机生成的，发送给服务器
- 服务端从请求(HTTP的请求头)信息中提取 **Sec-WebSocket-Key**，利用magic_string 和 Sec-WebSocket-Key 进行**hmac1**加密，再进行**base64**加密
- 将加密结果响应给客户端，服务器会返回下列东西，表示已经接受到请求， 成功建立 WebSocket

```
headers = get_headers(data)  # 提取请求头信息
# 对请求头中的sec-websocket-key进行加密
response_tpl = "HTTP/1.1 101 Switching Protocols\r\n" \
```

```
                "Upgrade:websocket\r\n" \
                "Connection: Upgrade\r\n" \
                "Sec-WebSocket-Accept: %s\r\n" \
                "WebSocket-Location: ws://127.0.0.1:9527\r\n\r\n"

value = headers['Sec-WebSocket-Key'] + magic_string
print(value)
ac = base64.b64encode(hashlib.sha1(value.encode('utf-8')).digest())
response_str = response_tpl % (ac.decode('utf-8'))
# 响应【握手】信息
conn.send(response_str.encode("utf8"))
```

- 依然是固定的，通过**Upgrade**告诉客户端即将升级的是 WebSocket 协议，而不是 mozillasocket，lurnarsocket 或者 shitsocket。
- 然后， **Sec-WebSocket-Accept** 这个则是经过服务器确认，并且加密过后的 **Sec-WebSocket-Key** 。 服务器：好啦好啦，知道啦，给你看我的 ID CARD 来证明行了吧。
- **Sec-WebSocket-Protocol** 则是表示最终使用的协议。

3.WebSocket - 解密

```
hashstr = b'\x81\x83\xceH\xb6\x85\xffz\x85'
# b'\x81    \x83    \xceH\xb6\x85\xffz\x85'

# 将第二个字节也就是 \x83 第9-16位 进行与127进行位运算
payload = hashstr[1] & 127
print(payload)
if payload == 127:
    extend_payload_len = hashstr[2:10]
    mask = hashstr[10:14]
    decoded = hashstr[14:]
# 当位运算结果等于127时,则第3-10个字节为数据长度
# 第11-14字节为mask 解密所需字符串
# 则数据为第15字节至结尾

if payload == 126:
    extend_payload_len = hashstr[2:4]
    mask = hashstr[4:8]
    decoded = hashstr[8:]
# 当位运算结果等于126时,则第3-4个字节为数据长度
# 第5-8字节为mask 解密所需字符串
# 则数据为第9字节至结尾


if payload <= 125:
    extend_payload_len = None
    mask = hashstr[2:6]
    decoded = hashstr[6:]

# 当位运算结果小于等于125时,则这个数字就是数据的长度
# 第3-6字节为mask 解密所需字符串
# 则数据为第7字节至结尾

str_byte = bytearray()
```

```
for i in range(len(decoded)):
    byte = decoded[i] ^ mask[i % 4]
    str_byte.append(byte)

print(str_byte.decode("utf8"))
```

4.WebSocket - 加密

```
import struct
msg_bytes = "hello".encode("utf8")
token = b"\x81"
length = len(msg_bytes)

if length < 126:
    token += struct.pack("B", length)
elif length == 126:
    token += struct.pack("!BH", 126, length)
else:
    token += struct.pack("!BQ", 127, length)
msg = token + msg_bytes
print(msg)
```

5.WebSocket - 作用

1.ajax轮询 和 long poll 的原理。

- ajax轮询的原理非常简单，让浏览器隔个几秒就发送一次请求，询问服务器是否有新信息
- long poll 其实原理跟 ajax轮询 差不多，都是采用轮询的方式，不过采取的是阻塞模型（一直打电话，没收到就不挂电话），也就是说，客户端发起请求后，如果没消息，就一直不返回 Response 给客户端。直到有消息才返回，返回完之后，客户端再次建立连接，周而复始。

2.WebSocket

- 这两种方式都不是最好的方式，需要很多资源，一种需要更快的速度，一种需要更多的'电话。这两种都会导致'电话'的需求越来越高。
- 忘记说了 HTTP 还是一个无状态协议。通俗的说就是，服务器因为每天要接待太多客户了，是个健忘鬼，你一挂电话，他就把你的东西全忘光了，把你的东西全丢掉了。你第二次还得再告诉服务器一遍。
- 只需要经过一次 HTTP 请求，就可以做到源源不断的信息传送了,当服务器完成协议升级后（HTTP->Websocket），服务端就可以主动推送信息给客户端啦。