域名查询,通过DNS服务器查询到IP,向web服务器发送请求,web服务器回传页面内容

http基于传输层tcp,基础网络层IP

DNS:域名到IP地址的解析

本地电脑会有一个映射关系,host文件里面会有对应的ip地址,如果找到了就直接使用

CDN和DNS什么关系,解析过程是怎么样的



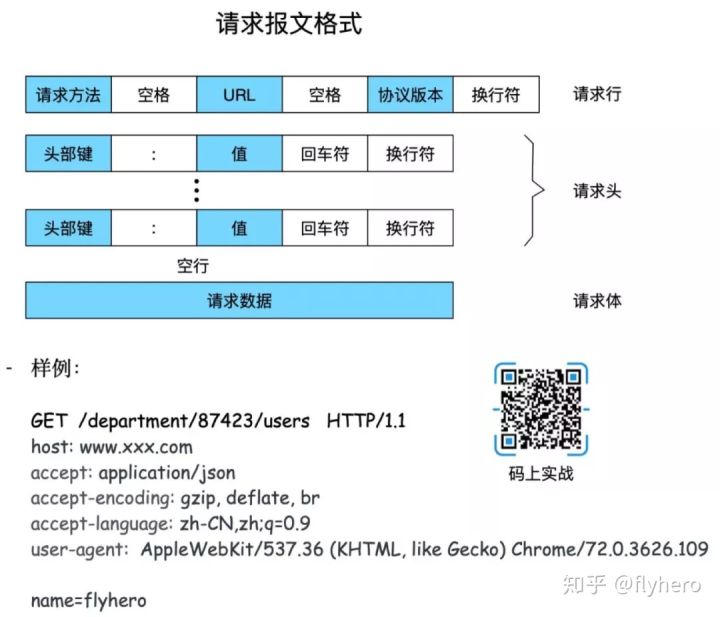
请求（Request）

客户端发送一个HTTP请求到服务端的格式：

请求行

请求头

请求体



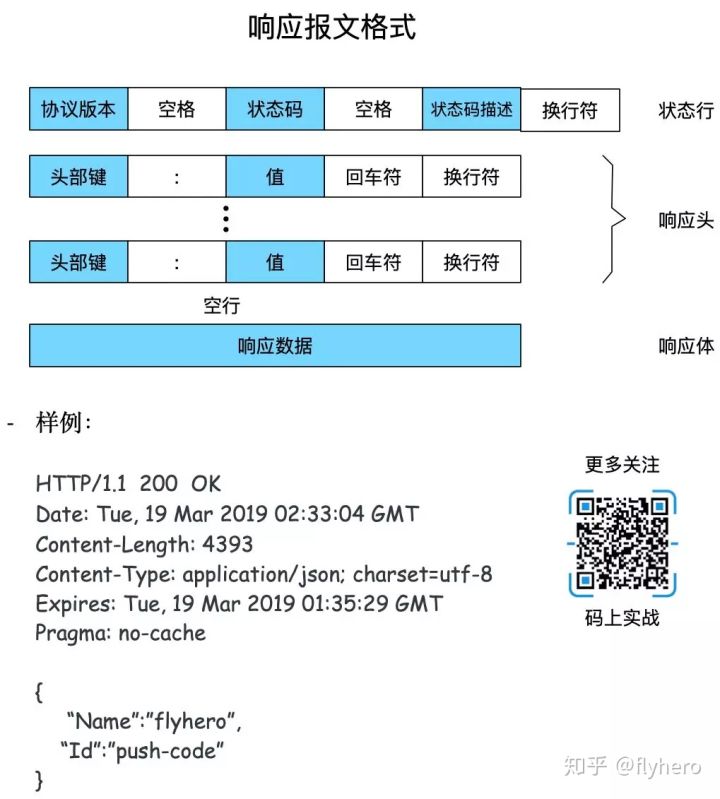
响应（Response）

服务端响应客户端格式：

状态行

响应头

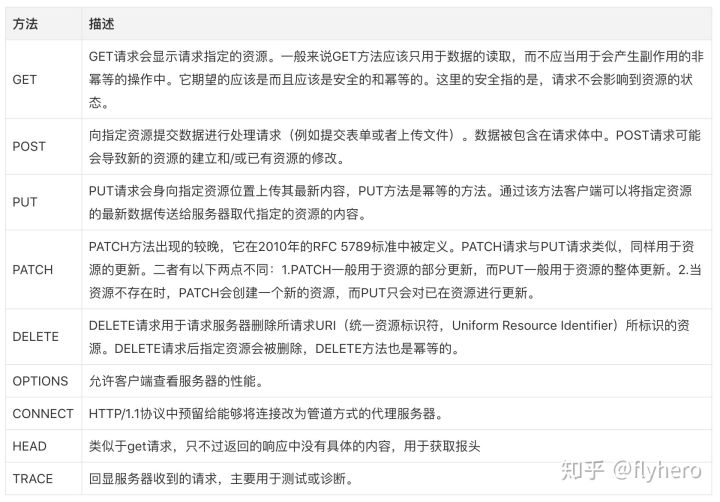
响应体



### 状态码

HTTP状态码由三个十进制数字组成，第一个十进制数字定义了状态码的类型，后两个数字没有分类的作用。HTTP状态码共分为5种类型：











# 主图



Restful:使用统一的资源接口(uri)来访问资源

### 内容协商:通过这些方法单一的url可以代表不同的资源

**（1）客户端驱动**  
    客户端发起请求，服务器发送可选项列表，客户端作出选择后在发送第二次请求。  
    优点：比较容易实现；  
    缺点：增加了时延，至少要发送两次请求，第一次请求获取资源列表，第二次获取选择的副本；

**（2）服务器驱动**  
    服务器检查客户端的请求首部集并决定提供哪个版本的页面。  
    优点：比客户端驱动的协商要快。HTTP提供了q机制，允许服务器近似匹配，还提供了vary首部供服务器告知下游的设备（如代理服务器）如何对请求估值；  
    缺点：首部集不匹配，服务器要做猜测；

**（3）透明协商**  
    某个中间设备（通常是缓存代理）代表客户端进行协商。  
    优点：免除了web服务器的协商开销，比客户端驱动的协商要快；  
    缺点：HTTP并没有提供相应的规范；

    其中，服务器驱动的解决方案应用的较为广泛。

Accept：告知服务器发送何种媒体类型；  
Accept-Language：告知服务器发送何种语言；  
Accept-Charset：告知服务器发送何种字符集；  
Accept-Encoding：告知服务器采用何种编码；

**四、q质量值的应用场景**

假设客户端的Accept-Language指定的是西班牙语，但是服务端只有英语与法语版本，这个客户端希望在没有西班牙语的时候优先返回英语。这就意味着，我们需要一种HTTP机制更详细的描述偏好。这种机制就是质量值（q值）。示例如下：

Accept-Language: en;q=0.5, fr;q=0.0, nl;q=1.0, tr;q=0.0

这个首部表示：用户最愿意接受荷兰语（nl），英文也行（en）,就是不愿意接受法语（fr）或者土耳其语(tr)

**q值的范围从0.0~1.0（1.0优先级最高）**

**五、vary首部的应用场景**

**服务器的决策不是依据Accept首部集（常规的内容协商首部集），而是比如Accept-Encoding**

假设整个请求过程是这样的：客户端 -> 代理服务器（具备缓存功能） ->web服务器。

第一个支持gzip压缩的客户端向中间代理服务器发送请求，代理服务器转发该请求，向web服务器拉取内容，拿到内容后代理服务器缓存该内容（由于请求首部有Accept-Encoding: gzip 所以内容会被压缩）。

第二个不支持gzip压缩的客户端也向中间代理服务器发送同一个请求，代理服务器发现该请求已经被缓存了，于是就把压缩后的内容响应给该客户端。悲剧了，因为该客户端根本不支持gzip压缩，也就没法解压。

**六、Vary首部的工作原理**

HTTP的Vary响应首部中列出了所有客户端请求首部，缓存服务器可以用这些首部来选择文档或者产生定制的内容。比如：若给客户端的响应内容取决于Accept-Encoding，Vary首部就必须包含Accept-Encoding。

当新的请求到达时，缓存服务器会根据内容协商首部集来寻找最佳匹配。但是在把文档提供给客户端之前，它必须检查web服务器有没有在已缓存响应中发生Vary首部。如果有，那么新请求中那些首部的值必须与旧的已缓存请求里相应的首部相同。因为web服务器可能会根据客户端请求的首部来改变响应，为了实现透明协商，**缓存服务器必须为每个已缓存变体保存客户端请求首部和相应的服务器响应首部**。

简单的讲：

web服务器添加响应首部Vary: Accept-Encoding 告知代理服务器根据客户端的请求首部Accept-Encoding缓存不同的版本，这样下次客户端请求同一资源时，根据Accept-Encoding选择相应的缓存版本响应。

http认证

# HTTP Basic认证

每次客户端请求都需带上Authorization请求头， 值为"Basic xxx"。xxx为对用户名和密码进行Base64编码后的值。 若客户端是浏览器，则浏览器会提供一个输入用户名和密码的对话框，用户输入用户名和密码后，浏览器会保存用户名和密码，用于构造Authorization值。当关闭浏览器后，用户名和密码将不再保存。

用户名/密码经过Base64加码后，这个Base64码值可以轻易被解码并获得用户名/密码，所以此认证方式并不安全。为了传输安全，需要配合SSL使用。

# HTTP另外 Digest认证

Digest认证步骤如下：  
第一步：客户端访问Http资源服务器。由于需要Digest认证，服务器返回了两个重要字段nonce（随机数）和realm。

第二步： 客户端构造Authorization请求头，值包含username、realm、nouce、uri和response的字段信息。其中，realm和nouce就是第一步返回的值。nouce只能被服务端使用一次。uri(digest-uri)即Request-URI的值，但考虑到经代理转发后Request-URI的值可能被修改、因此实现会复制一份副本保存在uri内。response也可叫做Request-digest，存放经过MD5运算后的密码字符串，形成响应码。

第三步：服务器验证包含Authorization值的请求，若验证通过则可访问资源。

Digest认证可以防止密码泄露和请求重放，但没办法防假冒。所以安全级别较低。

Digest和Basic认证一样，每次都会发送Authorization请求头，也就相当于重新构造此值。所以两者易用性都较差。

# HTTP SSL Client认证

SSL认证安全级别较高，但需要承担证书费用。SSL认证过程中涉及到一些重要的概念，数字证书机构的公钥、证书的私钥和公钥、非对称算法（配合证书的私钥和公钥使用）、对称密钥、对称算法（配合对称密钥使用）。

大致的认证步骤如下：  
第一步：客户端请求服务资源，服务器要求客户端出示数字证书。  
第二步：客户端发送数字证书  
第三步：服务器通过数字证书机构的公钥验证数字证书的合法性，验证通过后取出证书的公钥。  
第四步：服务端随机生成一个随机数即为对称密钥，并使用非对称算法和证书公钥加密。这个加密后的字符串，只有发送的客户端能解。  
第五步：客服端使用非对称解密算法和证书私钥获取服务端发送的对称密钥。后续客户端和服务端的请求直接基于对称算法和对称密钥。由于只有客户端和服务端有对称密钥，所以后续发送的请求较安全。

SSL可以防泄漏、假冒、重放，所以在Web系统中得到了广泛的应用。

SSL客户端认证在实际中用得不多，因为一来需要在客户端中安装证书（升级麻烦）、二来需要承担证书费用。

# HTTP 表单认证

基于表单的认证方式并不存在于HTTP规范。所以实现方式也呈现多样化。表单认证一般都会配合cookie+sessiond的使用，现在绝大多数的Web站点都是使用此认证方式。用户在登录页中填写用户名和密码，服务端认证通过后会将sessionId返回给浏览器端，浏览器会保存sessionId到浏览器的Cookie中。因为Http是无状态的，所以浏览器使用Cookie来保存sessionId。下次客户端发送的请求中会包含sessionId值，服务端发现sessionId存在并认证过则会提供资源访问。

# http缓存

https://www.cnblogs.com/ranyonsue/p/8918908.html

# http cookie

<https://www.cnblogs.com/bq-med/p/8603664.html>