如果要问大家这样一个问题：“在过去半个世纪中，哪种信息技术对人类的影响最为深远？”，我想很多人的答案是Web（World Wide Web、WWW、W3或者万维网），因为它改变了我们的生活方式和思维方式。如果各位阅读过W3C介绍WWW的官方文档（“http://www.w3.org/WWW/”），应该对它的第一句话记忆犹新——“The World Wide Web (known as "WWW', "Web" or "W3") is the universe of network-accessible information, the embodiment of human knowledge”。如果将这句话翻译成简洁的中文，就是“Web是（网络）信息的来源，知识的化身”。

Web为我们提供了一种利用HTTP协议获取和操作网络资源的方式，这些将Web服务器作为宿主的资源不仅仅包含像文字和图片这些传统的信息载体，还包含音频和视频这些多媒体信息。Web的核心主要体现在三个方面，即HTTP、超文本（Hypertext）和超媒体（Hypermedia），超文本和超媒体规范了网络信息的表现形式，而HTTP则提供了网络访问的标准协议。接下来我们就以围绕着HTTP对Web作一下基本的介绍。

下面主要讲的是HTTP1.1协议

# **TCP/IP与HTTP**

TCP/IP是以IP和TCP协议为核心的一整套网络协议的总称，所以有时候我们也称其为TCP/IP协议簇。毫不夸张地说，TCP/IP支撑着整个互联网，因为它就是互联网采用的网络协议。TCP/IP协议簇划分为如右图所示的4个层次（应用层、传输层、网络层和链路层），构成整个协议簇的各个子协议处于相应层次中。

既然将整个协议簇命名为TCP/IP，那么IP和TCP自然就是其中最为核心的两个协议了。处于网络层的IP协议提供的IP数据报传输是不可靠的，因为它只承诺尽可能地将数据报发送出去，但不能保证发送的数据报能够成功地抵达目的地。IP协议的不可靠性还体现在它不能检测数据在传输过程中是否发生了改变，也就是说数据的完整性得不到保证。IP协议是一个无连接（Connectionless）的网络协议，每次数据报的处理对它来说均是独立的，因此IP协议也不能提供针对有序传输（数据接收的顺序与发送的顺序一致）的保证。

虽然IP协议只能提供不可靠的数据传输，同时有序传输也得不到保证，但是建立在它之上的传输层协议TCP有效地解决了这两个问题。TCP是一个基于连接的协议，数据交换双方在进行报文传输之前需要建立连接，报文传输结束之后需要关闭连接。这是一个双工（Duplex）连接，数据交换的双工均可以利用它向对方发送数据。

TCP利用“接收确认”和“超时重传”机制确保了数据能够成功抵达目的地。具体来说，接收方在成功接收到数据之后会回复一个确认消息。发送方在本地具有一个存放尚未得到确认的已发消息的缓冲区，如果发送方在一个设定的时限内没有接收到针对某个已发报文的确认消息，它会从该缓存区中选择对应的报文进行重新发送。在接收到确认之后，相应的报文会从缓存区中移除。

为了解决有序传输的问题，发送方会为每个报文进行编号，报文的序号体现了它们被发送的顺序。接收端在接收到某个报文之后，它会利用此序号判断是否具有尚未成功接收的已发报文，如果有的话，该报文会被存放到本地的缓冲区中。等到之前发送的报文全部被接收之后，接收方按照序号对接收的报文依次向上（应用层）递交，成功递交的报文会被从缓存区中移除。除了接收到“失序”的报文之外，接收方还有可能接收到重复的报文，因为没有报文均具有一个唯一的序号，如果该序号小于已经成功递交或者添加到缓存区中的报文序号，它会被认为是重复接收的报文而被丢弃。

由于每个TCP报文段都具有一个16位的检验和（Checksum），所以接收方可以根据它确认数据在传输过程中是否被篡改。除此之外，TCP还提供了“流量控制”功能避免了双方因缓存区大小不一致而导致报文丢失。具体来说，如果发送方的缓冲区大于接收方的缓存区，会导致接收方在缓冲区已满的情况下无法处理后续接收的报文，所以接收方会将自己缓存区剩余的大小及时通知给发送端，后者据此控制报文发送“流量”。

HTTP（Hypertext Transfer Protocol），全称为“超文本传输协议”，是TCP/IP协议簇的一部分。从图1-1可以看出，这是一个位于应用层的网络协议，在它之下的就是TCP协议。由于TCP协议是一个“可靠”的协议，HTTP自然也能提供可靠数据传输功能。

IP协议利用IP地址来定位数据报发送的目的地，而利用域名系统（DNS）可以实现域名与IP地址之间的转换。TCP协议利用端口号标识应用程序，所以某个应用程序在使用TCP协议进行通信的时候必须指定目标应用的IP地址（或者域名）和端口号。HTTP默认采用的端口号为80，而HTTPS（利用TLS/SSL为HTTP提供传输安全保障）的默认端口号则为443，当然在网络可达的前提下，我们可以指定任意的端口。

# **Web资源**

这里所说的资源是一个宽泛的概念，任何寄宿于Web服务器可以利用HTTP协议获取或者操作的“事物”均可以称为资源。这也是一个抽象的概念，不仅仅是寄宿于Web服务器的某个静态物理文件可以视为Web资源，通过Web应用根据请求动态生成的数据也是Web资源。

## **媒体类型**

资源实际上是一种承载着某种信息的数据，相同的信息可以采用不同形态的数据来展现，数据的“形态”主要体现为展示数据所采用的格式，比如一个数据对象可以通过XML格式来表示，也可以通过JSON格式来表示。数据的处理必须依赖于一种已知的格式，所以将Web资源的形态以一种标准化的方式固定下来显得尤为重要，这就是我们接下来着重介绍的媒体媒体（Media Type）。

不论是通过HTTP请求从Web服务器上获取资源，还是利用请求向服务器提交资源，响应或者请求的主体（Body）除了包含承载资源本身的数据之外，其报头（Header）部分还应该包含表示数据形态的媒体类型。

媒体类型又被称为MIME（Multipurpose Internet Mail Extension）类型，MIME是一个互联网标准，它扩展了电子邮件标准，使其能够支持非ASCII字符、二进制格式附件等多种格式的邮件消息。由于MIME在电子邮件系统应用得非常好，所以被HTTP用于描述并标记多媒体内容。下面的列表给出了一种常用的媒体类型。

* text/html：HTML格式的文档。
* text/xml（application/xml）：XML格式的文本。
* text/json（application/json）： JSON格式的文本。
* image/gif（image/jpeg、image/png）：GIF（JPEG、PNG）格式的图片。
* audio/mp4（audio/mpeg、audio/vnd.wave）：MP4（MPEG、WAVE）格式的音频文件。
* video/mp4（video/mpeg、video/quicktime）：MP4（MPEG、QUICKTIME）格式的视频文件。

## **URI、URL和URN**

可操作的Web资源应该具有一个 唯一的标识。虽然具有很多唯一性标志符的种类可供选择（比如GUID），但是采用URI来标识Web资源已经成为了一种共识，实际上URI的全称为“统一资源标志符（Uniform Resource Identifier）”。

我想有很多人弄不清楚URI和URL之间的区别，有人甚至觉得这是同一概念的不同表述而已。一个URL肯定是一个URI，但是一个URI并不一定是一个URL，URL仅仅是URI的一种表现形式而已。两者的差异其实可以直接从其命名来区分，URI是Web资源的标志符，所以只要求它具有“标识性”即可；URL全称为“统一资源定位符（Uniform Resource Locator）”，所以除了标识性之外，它还具有定位的功能，用于描述Web资源所在的位置。

URL不仅仅用于定位目标资源所在的位置，还指名了获取资源所采用的协议，一个完整的URL包含协议名称、主机名称（IP地址或者域名）、端口号、路径和查询字符串5个部分。比如对于“ http://www.artech.com:8080/images/photo.png?size=small”这样一个URL，上述的5个部分分别是“http”、“www.artech.com”、“8080”、“/images/photo.png”和“?size=small”。

除了URL，URN也是URI的一种表现形式，URN全称“统一资源定位符（Uniform Resource Name）”。URN与资源所在的位置无关，倘若采用URN来唯一标识某个资源，在位置发生改变的时候标志符依然可以保持不变。URN一般也不会涉及到获取被标识资源采用的网络协议，所以不需要为利用不同协议访问的相同资源定义不同的标志符。

# **HTTP事务**

虽然TCP是一种基于连接的传输层协议，并且保存双方针对同一个连接的多轮消息交换的会话状态，但是建立其上的HTTP则是一种无状态的网络协议。HTTP采用简单的“请求/响应”消息交换模式，一次HTTP事务（Transaction）始于请求的发送，止于响应的接收。针对客户端和Web服务器的多次消息交换来说，每个HTTP事务均是相互独立的。

## **HTTP方法**

HTTP采用简单的请求/响应模式的消息交换旨在实现针对某个Web资源的某种操作。至于针对资源的操作类型，不外乎CRUD（Create、Retrieve、Update和Delete）而已。一个HTTP请求除了利用URI标志目标资源之外，还需要通过HTTP方法（HTTP Method或者HTTP Verb）指名针对资源的操作类型。我们常用的HTTP方法 包括GET、POST、PUT、DELETE、HEAD、OPTIONS、TRACE、CONNECTION和PATCH等

## **响应状态码**

针对客户端向Web服务器发送的任意一个HTTP请求，不论在何种情况下得到一个响应，每个响应均具有一个由3位数字表示的状态码和相应的描述文字。不同数值的状态码体现了不同类型的响应状态，W3C对响应状态码的范围作了如下的规范。

* 100~199：信息状态码，代表请求已被接受，需要继续处理。
* 200~299：成功状态码，代表请求已成功被服务器接收、理解、并接受。
* 300~399：重定向状态码，代表需要客户端采取进一步的操作才能完成请求。
* 400~499：客户端错误状态码，代表了客户端看起来可能发生了错误，妨碍了服务器的处理。
* 500~599：服务器错误状态码，代表了服务器在处理请求的过程中有错误或者异常状态发生，也有可能是服务器意识到以当前的软硬件资源无法完成对请求的处理。

具体的响应码如下：

### 1XX: 信息提示，消息响应

状态码 原因短语 代表含义

100 Continue 继续,客户端应继续其请求

101 Switching Protocols 切换协议,服务器根据客户端的请求切换协议,只能切换到更高级的协议.例如:切换到HTTP的新版本协议

### 2XX: 成功响应

状态码 原因短语 代表含义

200 OK 请求成功,一般用于GET与POST请求

201 Created 成功请求并创建了新的资源

202 Accepted 已经接受请求，但未处理完成

203 Non-Authoritative Information 非授权信息。请求成功。但返回的meta信息不在原始的服务器，而是一个副本

204 No Content 服务器成功处理，但未返回内容。在未更新网页的情况下，可确保浏览器继续显示当前文档

205 Reset Content 重置内容。服务器处理成功，用户终端（例如：浏览器）应重置文档视图.可通过此返回码清除浏览器的表单域

206 Partial Content 服务器成功处理了部分GET请求

### 3XX: 重定向

状态码 原因短语 代表含义

300 Multiple Choices 多种选择。请求的资源可包括多个位置，相应可返回一个资源特征与地址的列表用于用户终端（例如：浏览器）选择

301 Moved Permanently 永久移动。请求的资源已被永久的移动到新URI，返回信息会包括新的URI，浏览器会自动定向到新URI。今后任何新的请求都应使用新的URI代替

302 Found 临时移动。与301类似。但资源只是临时被移动。客户端应继续使用原有URI

303 See Other 查看其它地址。与301类似。使用GET和POST请求查看

304 Not Modified 未修改。所请求的资源未修改，服务器返回此状态码时，不会返回任何资源。客户端通常会缓存访问过的资源，通过提供一个头信息指出客户端希望只返回在指定日期之后修改的资源

305 Use Proxy 使用代理。所请求的资源必须通过代理访问

306 Unused 已经被废弃的HTTP状态码

307 Temporary Redirect 临时重定向。与302类似。使用GET请求重定向

### 4XX: 错误类信息，客户端错误

状态码 原因短语 代表含义

400 Bad Request 客户端请求的语法错误，服务器无法理解

401 Unauthorized 请求要求用户的身份认证

402 Payment Required 保留，将来使用

403 Forbidden 服务器理解请求客户端的请求，但是拒绝执行此请求

404 Not Found 服务器无法根据客户端的请求找到资源（网页）。通过此代码，网站设计人员可设置"您所请求的资源无法找到"的个性页面

405 Method Not Allowed 客户端请求中的方法被禁止

406 Not Acceptable 服务器无法根据客户端请求的内容特性完成请求

407 Proxy Authentication Required 请求要求代理的身份认证，与401类似，但请求者应当使用代理进行授权

408 Request Time-out 服务器等待客户端发送的请求时间过长，超时

409 Conflict 服务器完成客户端的PUT请求是可能返回此代码，服务器处理请求时发生了冲突

410 Gone 客户端请求的资源已经不存在。410不同于404，如果资源以前有现在被永久删除了可使用410代码，网站设计人员可通过301代码指定资源的新位置

411 412 Precondition Failed 客户端请求信息的先决条件错误

413 Request Entity Too Large 由于请求的实体过大，服务器无法处理，因此拒绝请求。为防止客户端的连续请求，服务器可能会关闭连接。如果只是服务器暂时无法处理，则会包含一个Retry-After的响应信息

414 Request-URI Too Large 请求的URI过长（URI通常为网址），服务器无法处理

415 Unsupported Media Type 服务器无法处理请求附带的媒体格式

416 Requested range not satisfiable 客户端请求的范围无效

417 Expectation Failed 服务器无法满足Expect的请求头信息

### 5XX: 错误类信息，服务器端错误

状态码 原因短语 代表含义

500 Internal Server Error 服务器内部错误，无法完成请求

501 Not Implemented 服务器不支持请求的功能，无法完成请求

502 Bad Gateway 作为网关或者代理工作的服务器尝试执行请求时，从远程服务器接收到了一个无效的响应

503 Service Unavailable 由于超载或系统维护，服务器暂时的无法处理客户端的请求。延时的长度可包含在服务器的Retry-After头信息中

504 Gateway Time-out 充当网关或代理的服务器，未及时从远端服务器获取请求

505 HTTP Version not supported 服务器不支持请求的HTTP协议的版本，无法完成处理Length Required 服务器无法处理客户端发送的不带Content-Length的请求信息

# **HTTP报文**

## 协议的作用及版本

(1). HTTP是Hyper Text Transfer Protocol（超文本(html)传输协议）。

(2). HTTP协议的作用用户描述客户端与服务器间的数据传递的。

(3). Http协议的版本：

HTTP/1.0:特点：

每次请求服务器上的资源都要建立新的连接，响应完毕后都会关闭连接。

是无状态的协议。

HTTP/1.1:特点：

在一次TCP/IP连接的基础上可以发出多次请求和得到多次的响应。

多个请求和相应可以叠加。

比1.0多了一些请求和响应头。

## 协议的组成

(1).请求部分：

GET /App1/1.html HTTP/1.1

Accept: \*/\*

Accept-Language: zh-cn

Accept-Encoding: gzip, deflate

User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR 2.0.50727; .NET CLR 3.0.4506.2152; .NET CLR 3.5.30729)

Host: localhost:8080

Connection: Keep-Alive

第一个空行-->

username=admin&password=123

请求部分由三部分组成的：

请求行：位于第一行,GET /App1/1.html HTTP/1.1

请求消息头：从第二行开始至第一个空行结束

请求正文：从第一个空行之后的都是正文

(2).响应部分：

HTTP/1.1 200 OK

Server: Apache-Coyote/1.1

Accept-Ranges: bytes

ETag: W/"65-1366335797484"

Last-Modified: Fri, 19 Apr 2013 01:43:17 GMT

Content-Type: text/html

Content-Length: 65

Date: Fri, 19 Apr 2013 02:06:23 GMT

Warnning：w1

Warnning: w2

hello http

<a href="b.html">click</a>

<img src="1.jpg"/>

响应部分由三部分组成的：

响应行：位于第一行

响应消息头：从第二行开始至第一个空行结束

响应正文：从第一个空行之后的都是正文

## 协议详细讲解

(1).请求行：GET /App1/1.html HTTP/1.1

GET：请求方式。默认的请求方式。其他常用的请求方式还有POST。

GET的特点：默认的请求方式。

/App1/c.html?username=sdsfds&password=234324

把表单的请求的数据放在了请求的URI的后面。?username=sdsfds&password=234324

这样不好:暴露数据；请求行长度有限。

POST的特点(经常使用的)：借助HTML中的form表单。<form action="c.html" method="post">

请求参数出现在正文部分。长度木有限制。相对安全。

---------------------------

/App1/1.html:请求的资源的URI。

---------------------------

HTTP/1.1：客户端使用的协议的版本

(2).响应行：HTTP/1.1 200 OK

HTTP/1.1：服务器用的协议版本

200：响应码。代表服务器处理的结果的一种表示

200：正常

302/307：重定向

304:服务器的资源没有被修改

404：请求的资源不存在

500：服务器报错了

OK：响应码描述

(3).请求消息头：向服务器传递附加信息

Accept:通知服务器，浏览器可以接受的MIME类型。（文件系统中用文件扩展名区分数据的类型。

网络上用MIME类型来区分数据类型。Tomcat\conf\web.mxl） MIME类型名称：大类型/小类型

Accept-Charset:通知服务器，浏览器支持的字符集

Accept-Encoding:通知服务器，浏览器能够解码的数据压缩方式。比如：gzip

Accept-language:通知服务器，所希望的语言

Host：请求的主机和端口

Referer：是一个URL地址。取值是当前页面之前的那个页面地址的。防盗链

Content-Type:通知服务器，请求正文的MIME类型。

取值：application/x-www-form-urlencoded默认值

对应的是form表单的enctype属性

If-Modified-Since:通知服务器，缓存的文件的最后修改时间。

User-Agent:通知服务器，浏览器类型.

Content-Length:表示请求消息正文的长度

Connection:表示是否需要持久连接。如果服务器看到这里的值为“Keep -Alive”，或者看到请求使用的是

HTTP 1.1（HTTP 1.1默认进行持久连接

Cookie:这是最重要的请求头信息之一（会话有关）一个浏览器会产生一个cookieID。

(4).响应消息头：

Location:通知客户端，指示新的资源的位置（结合302/307来用。请求重定向）

Server:通知客户端，服务器的类型

Content-Encoding:通知客户端，响应正文的压缩编码方式。常用的是gzip。

Content-Length：通知客户端响应正文的数据大小

Content-Type:通知客户端响应正文的MIME类型

Refresh：让浏览器自动刷新。取值为整数（刷新的时间间隔，单位是秒）

Refresh:3

Refresh:3;URL=其他资源的URI

Content-Disposition：通知客户端，以下载的方式打开资源。

Content-Disposition：attachment;filename=1.jpg

Set-Cookie:SS=Q0=5Lb\_nQ; path=/search服务器端发送的Cookie（会话有关）

Expires: -1 网页的有效时间。单位是毫秒

Cache-Control: no-cache (1.1)

Pragma: no-cache (1.0) 通知客户端不要缓存

# Keep-Alive

在http早期，每个http请求都要求打开一个tpc socket连接，并且使用一次之后就断开这个tcp连接。

使用keep-alive可以改善这种状态，即在一次TCP连接中可以持续发送多份数据而不会断开连接。通过使用keep-alive机制，可以减少tcp连接建立次数，也意味着可以减少TIME\_WAIT状态连接，以此提高性能和提高httpd服务器的吞吐率(更少的tcp连接意味着更少的系统内核调用,socket的accept()和close()调用)。

但是，keep-alive并不是免费的午餐,长时间的tcp连接容易导致系统资源无效占用。配置不当的keep-alive，有时比重复利用连接带来的损失还更大。所以，正确地设置keep-alive timeout时间非常重要。

## keepalvie timeout

Httpd守护进程，一般都提供了keep-alive timeout时间设置参数。比如nginx的keepalive\_timeout，和Apache的KeepAliveTimeout。这个keepalive\_timout时间值意味着：一个http产生的tcp连接在传送完最后一个响应后，还需要hold住keepalive\_timeout秒后，才开始关闭这个连接。

当httpd守护进程发送完一个响应后，理应马上主动关闭相应的tcp连接，设置 keepalive\_timeout后，httpd守护进程会想说：”再等等吧，看看浏览器还有没有请求过来”，这一等，便是keepalive\_timeout时间。如果守护进程在这个等待的时间里，一直没有收到浏览发过来http请求，则关闭这个http连接。

下面写一个脚本，方便测试：

sleep(60); //为了便于分析测试，会根据测试进行调整

echo "www.example.com";

1. 当keepalive\_timeout时间为0时，即不启用Keep-Alive时，一个tcp连接的生命周期：

#tcpdump -n host 218.1.57.236 and port 80

20:36:50.792731 IP 218.1.57.236.43052 > 222.73.211.215.http: S 1520902589:1520902589(0) win 65535

20:36:50.792798 IP 222.73.211.215.http > 218.1.57.236.43052: S 290378256:290378256(0) ack 1520902590 win 5840

20:36:50.801629 IP 218.1.57.236.43052 > 222.73.211.215.http: . ack 1 win 32768

20:36:50.801838 IP 218.1.57.236.43052 > 222.73.211.215.http: P 1:797(796) ack 1 win 32768

20:36:50.801843 IP 222.73.211.215.http > 218.1.57.236.43052: . ack 797 win 59

20:37:50.803230 IP 222.73.211.215.http > 218.1.57.236.43052: P 1:287(286) ack 797 win 59

20:37:50.803289 IP 222.73.211.215.http > 218.1.57.236.43052: F 287:287(0) ack 797 win 59

20:37:50.893396 IP 218.1.57.236.43052 > 222.73.211.215.http: . ack 288 win 32625

20:37:50.894249 IP 218.1.57.236.43052 > 222.73.211.215.http: F 797:797(0) ack 288 win 32625

20:37:50.894252 IP 222.73.211.215.http > 218.1.57.236.43052: . ack 798 win 59

第1~3行建立tcp三次握手，建立连接。用时8898μs

第4~5行通过建立的连接发送第一个http请求，服务端确认收到请求。用时5μs

第5~6行，可以知道脚本执行用时60s1387μs,与php脚本相符。

第6、8行服务端发送http响应。发送响应用时90166μs。

第7行，表明由服务端守护进程主动关闭连接。结合第6、8行，说明http响应一旦发送完毕，服务端马上关闭这个tcp连接

第7、9、10说明tcp连接顺序关闭,用时90963μs。需要注意,这里socket资源并没有立即释放，需要等待2MSL时间（60s）后才被真正释放。

由此可见，在没有设置 keepalive\_timeout 情况下，一个socket资源从建立到真正释放需要经过的时间是:建立tcp连接 + 传送http请求 + php脚本执行 + 传送http响应 + 关闭tcp连接 + 2MSL 。(注:这里的时间只能做参考，具体的时间主要由网络带宽，和响应大小而定)

2. 当keepalive\_timeout时间大于0时，即启用Keep-Alive时，一个tcp连接的生命周期。为了便于分析，我们将keepalive\_timeout设置为300s

#tcpdump -n host 218.1.57.236 and port 80

21:38:05.471129 IP 218.1.57.236.54049 > 222.73.211.215.http: S 1669618600:1669618600(0) win 65535

21:38:05.471140 IP 222.73.211.215.http > 218.1.57.236.54049: S 4166993862:4166993862(0) ack 1669618601 win 5840

21:38:05.481731 IP 218.1.57.236.54049 > 222.73.211.215.http: . ack 1 win 32768

21:38:05.481976 IP 218.1.57.236.54049 > 222.73.211.215.http: P 1:797(796) ack 1 win 32768

21:38:05.481985 IP 222.73.211.215.http > 218.1.57.236.54049: . ack 797 win 59

21:38:07.483626 IP 222.73.211.215.http > 218.1.57.236.54049: P 1:326(325) ack 797 win 59

21:38:07.747614 IP 218.1.57.236.54049 > 222.73.211.215.http: . ack 326 win 32605

21:43:07.448454 IP 222.73.211.215.http > 218.1.57.236.54049: F 326:326(0) ack 797 win 59

21:43:07.560316 IP 218.1.57.236.54049 > 222.73.211.215.http: . ack 327 win 32605

21:43:11.759102 IP 218.1.57.236.54049 > 222.73.211.215.http: F 797:797(0) ack 327 win 32605

21:43:11.759111 IP 222.73.211.215.http > 218.1.57.236.54049: . ack 798 win 59

我们先看一下，第6~8行，跟上次示例不一样的是，服务端httpd守护进程发完响应后，没有立即主动关闭tcp连接。

第8行，结合第6行，我们可以看到，5分钟(300s)后，服务端主动关闭这个tcp连接。这个时间，正是我们设置的keepalive\_timeout的时间。

由此可见，设置了keepalive\_timout时间情况下，一个socket建立到释放需要的时间是多了keepalive\_timeout时间。

3. 当keepalive\_timeout时间大于0，并且在同一个tcp连接发送多个http响应。这里为了便于分析，我们将keepalive\_timeout设置为180s

通过这个测试，我们想弄清楚,keepalive\_timeout是从第一个响应结束开启计时，还是最后一个响应结束开启计时。测试结果证实是后者，这里，我们每隔120s发一次请求，通过一个tcp连接发送了3个请求。

# tcpdump -n host 218.1.57.236 and port 80

22:43:57.102448 IP 218.1.57.236.49955 > 222.73.211.215.http: S 4009392741:4009392741(0) win 65535

22:43:57.102527 IP 222.73.211.215.http > 218.1.57.236.49955: S 4036426778:4036426778(0) ack 4009392742 win 5840

22:43:57.111337 IP 218.1.57.236.49955 > 222.73.211.215.http: . ack 1 win 32768

22:43:57.111522 IP 218.1.57.236.49955 > 222.73.211.215.http: P 1:797(796) ack 1 win 32768

22:43:57.111530 IP 222.73.211.215.http > 218.1.57.236.49955: . ack 797 win 59

22:43:59.114663 IP 222.73.211.215.http > 218.1.57.236.49955: P 1:326(325) ack 797 win 59

22:43:59.350143 IP 218.1.57.236.49955 > 222.73.211.215.http: . ack 326 win 32605

22:45:59.226102 IP 218.1.57.236.49955 > 222.73.211.215.http: P 1593:2389(796) ack 650 win 32443

22:45:59.226109 IP 222.73.211.215.http > 218.1.57.236.49955: . ack 2389 win 83

22:46:01.227187 IP 222.73.211.215.http > 218.1.57.236.49955: P 650:974(324) ack 2389 win 83

22:46:01.450364 IP 218.1.57.236.49955 > 222.73.211.215.http: . ack 974 win 32281

22:47:57.377707 IP 218.1.57.236.49955 > 222.73.211.215.http: P 3185:3981(796) ack 1298 win 32119

22:47:57.377714 IP 222.73.211.215.http > 218.1.57.236.49955: . ack 3981 win 108

22:47:59.379496 IP 222.73.211.215.http > 218.1.57.236.49955: P 1298:1622(324) ack 3981 win 108

22:47:59.628964 IP 218.1.57.236.49955 > 222.73.211.215.http: . ack 1622 win 32768

22:50:59.358537 IP 222.73.211.215.http > 218.1.57.236.49955: F 1622:1622(0) ack 3981 win 108

22:50:59.367911 IP 218.1.57.236.49955 > 222.73.211.215.http: . ack 1623 win 32768

22:50:59.686527 IP 218.1.57.236.49955 > 222.73.211.215.http: F 3981:3981(0) ack 1623 win 32768

22:50:59.686531 IP 222.73.211.215.http > 218.1.57.236.49955: . ack 3982 win 108

第一组，三个ip包表示tcp三次握手建立连接，由浏览器建立。

第二组，发送第一次http请求并且得到响应，服务端守护进程输出响应之后，并没马上主动关闭tcp连接。而是启动keepalive\_timout计时。

第三组，2分钟后，发送第二次http请求并且得到响应，同样服务端守护进程也没有马上主动关闭tcp连接，重新启动keepalive\_timout计时。

第四组，又2分钟后，发送了第三次http请求并且得到响应。服务器守护进程依然没有主动关地闭tcp连接（距第一次http响应有4分钟了,大于keepalive\_timeout值），而是重新启动了keepalive\_timout计时。

第五组，跟最后一个响应keepalive\_timeout(180s)内，守护进程再没有收到请求。计时结束，服务端守护进程主动关闭连接。4次挥手后，服务端进入TIME\_WAIT状态。

这说明，当设定了keepalive\_timeout，一个socket由建立到释放，需要时间是：tcp建立 + (最后一个响应时间 – 第一个请求时间) + tcp关闭 + 2MSL。红色加粗表示每一次请求发送时间、每一次请求脚本执行时间、每一次响应发送时间，还有两两请求相隔时间。进一步测试，正在关闭或者TIME\_WAIT状态的tcp连接，不能传输http请求和响应。即，当一个连接结束keepalive\_timeout计时，服务端守护进程发送第一个FIN标志ip包后，该连接不能再使用了。

## http keep-alive与tcp keep-alive

http keep-alive与tcp keep-alive，不是同一回事，意图不一样。http keep-alive是为了让tcp活得更久一点，以便在同一个连接上传送多个http，提高socket的效率。而tcp keep-alive是TCP的一种检测TCP连接状况的保鲜机制。tcp keep-alive保鲜定时器，支持三个系统内核配置参数：

echo 1800 > /proc/sys/net/ipv4/tcp\_keepalive\_time

echo 15 > /proc/sys/net/ipv4/tcp\_keepalive\_intvl

echo 5 > /proc/sys/net/ipv4/tcp\_keepalive\_probes

keepalive是TCP保鲜定时器，当网络两端建立了TCP连接之后，闲置idle（双方没有任何数据流发送往来）了tcp\_keepalive\_time后，服务器内核就会尝试向客户端发送侦测包，来判断TCP连接状况(有可能客户端崩溃、强制关闭了应用、主机不可达等等)。如果没有收到对方的回答(ack包)，则会在 tcp\_keepalive\_intvl后再次尝试发送侦测包，直到收到对对方的ack,如果一直没有收到对方的ack,一共会尝试 tcp\_keepalive\_probes次，每次的间隔时间在这里分别是15s, 30s, 45s, 60s, 75s。如果尝试tcp\_keepalive\_probes,依然没有收到对方的ack包，则会丢弃该TCP连接。TCP连接默认闲置时间是2小时，一般设置为30分钟足够了。

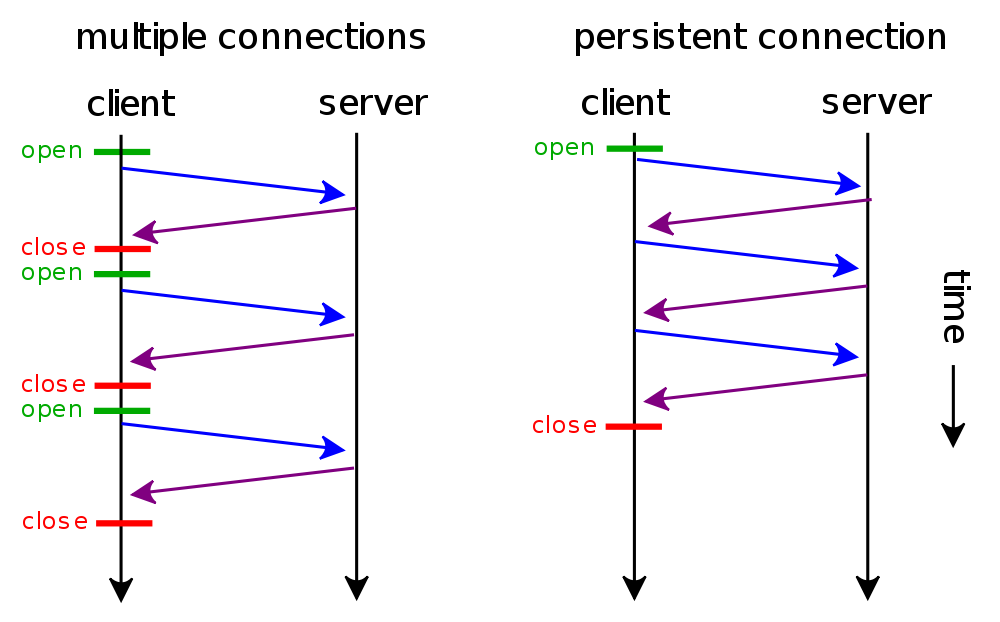
也就是说，仅当nginx的keepalive\_timeout值设置高于tcp\_keepalive\_time，并且距此tcp连接传输的最后一个http响应，经过了tcp\_keepalive\_time时间之后，操作系统才会发送侦测包来决定是否要丢弃这个TCP连接。一般不会出现这种情况，除非你需要这样做。

## keep-alive与TIME\_WAIT

使用http keep-alvie，可以减少服务端TIME\_WAIT数量(因为由服务端httpd守护进程主动关闭连接)。道理很简单，相较而言，启用keep-alive，建立的tcp连接更少了，自然要被关闭的tcp连接也相应更少了。

最后

我想用一张示意图片来说明使用启用keepalive的不同。另外，http keepalive是客户端浏览器与服务端httpd守护进程协作的结果，所以，我们另外安排篇幅介绍不同浏览器的各种情况对keepalive的利用。



# http与https

<https://www.jianshu.com/p/26285148ec32>

* http是不安全的

数据容易被拦截，被篡改，被攻击等

* https是安全的

对数据进行加密，对请求进行身份验证，数据是完整的不易被篡改

* https的连接

https://mp.weixin.qq.com/s/Sqco2jlApWpDAJ9XD7Mgcw

## https使用非对称加密

对称加密：就是使用一个密钥，一个加密也可以解密

非对称加密：就是加密使用公钥，解密必须使用私钥；加密使用私钥，解密必须使用私钥；

公钥是可以公开的，而私钥是只有一个人知道的（在浏览器中浏览器有自己的私钥）

## HTTPS双向认证指南

客户端和服务器端都需要验证对方的身份，在建立Https连接的过程中，握手的流程比单向认证多了几步。单向认证的过程，客户端从服务器端下载服务器端公钥证书进行验证，然后建立安全通信通道。双向通信流程，客户端除了需要从服务器端下载服务器的公钥证书进行验证外，还需要把客户端的公钥证书上传到服务器端给服务器端进行验证，等双方都认证通过了，才开始建立安全通信通道进行数据传输。

<https://www.jianshu.com/p/2b2d1f511959?utm_campaign=haruki>

<https://help.aliyun.com/document_detail/160093.html?spm=5176.22414175.sslink.1.92b21e432Ub9Fd>

<https://help.aliyun.com/document_detail/85954.html?spm=5176.22414175.sslink.2.92b21e432Ub9Fd>

HTTPS双向认证方式通信在一些安全级别较高的场景非常有用，拥有合法证书的客户端才能正常访问业务。

# 常见面试题

## 浏览器与服务器建立TCP连接后，是否会在一个http请求完成后断开？什么情况下回断开？

http/1.0 的时候，服务器完成一个http连接之后就会断开这个连接。每次请求每次重连，代码很大。到了http/1.1 后 ，增加了 Connection: keep-alive，在请求之后会保持连接一定时间。当 Header 中有 collection: close 的时候，TCP连接会关闭；或者 当keep-alive 设置的时间到了，自动断开。

## 一个 TCP 连接可以对应几个http请求？

一个 TCP 创建出来后，会保持一定时间。这个时间内，可以发送多个http请求。

## 一个 TCP 连接中，http请求可以一起发送吗？

在http/1.1 中， 单个 TCP 同时只能处理一个请求。也就是当我们打开一个页面的时候，有多个较大的请求，他们应该各个会创建一个 TCP 连接。其实就是两个不同的http请求，开始到结束的时间在同一个 TCP 连接里不能重叠。但是在http/2， 提供了multiplexing，支持多路传输

## 为什么有的时候刷新页面不需要重新创建SSL连接？

TCP 连接有的时候会被浏览器和服务端维持一段时间。TCP 不需要重新建立，SSL 自然也会用之前的。

## 浏览器对同一个host建立的TCP连接数量是否有限制？

假设我们还处在 HTTP/1.1 时代，那个时候没有多路传输，当浏览器拿到一个有几十张图片的网页该怎么办呢？肯定不能只开一个 TCP 连接顺序下载，那样用户肯定等的很难受但是如果每个图片都开一个 TCP 连接发 HTTP 请求，那电脑或者服务器都可能受不了要是有 1000 张图片的话总不能开 1000 个TCP 连接吧，你的电脑同意 NAT 也不一定会同意。Chrome 最多允许对同一个 Host 建立六个 TCP 连接。不同的浏览器有一些区别。

## HTTP长连接、短连接究竟是什么？

<https://www.cnblogs.com/gotodsp/p/6366163.html>

## http常见面试题

<https://mp.weixin.qq.com/s/t7ZYT6wBBbFYVBPOSztpRg>

HTTP Status Code详解： <https://mp.weixin.qq.com/s/UaVuQEUehZxJdv0i-NiZmQ>