目录

均衡算法选择与权重配置实例

负载均衡算法比较分析

加权轮询算法

- **原理**：轮询调度算法就是以轮询的方式依次将请求调度到不同的服务器，即每次调度执行i = (i + 1) mod n，选出第 i 台服务器。加权轮询调度算法可以解决服务器间性能不一的情况，它用相应的权值表示服务器的处理性能，按权值的高低和轮询方式分配请求到各服务器。权值高的服务器先收到连接，权值高的服务器比权值低的服务器处理更多的连接，相同权值的服务器处理相同数目的连接数。
- **优势**：算法简洁。它无需记录当前所有连接的状态，所以它是一种无状态调度。
- **劣势**：不适用于请求服务时间变化比较大，或者每个请求所消耗的时间不一致的情况，此时轮询调度算法容易导致服务器间的负载不平衡。
- **适用场景**：每个请求所占用的后端服务器时间基本相同，常用于短连接服务，例如 HTTP 等服务。
- **用户推荐**：用户可知每个请求所占用后端时间基本相同或相差较小时，如已知后端服务器处理的都是同类型或者相似类型的请求时，推荐选择加权轮询的方式，因为该实现方式消耗小，无需遍历，效率较高。

加权最小连接数算法

- **原理**：在实际情况中，客户端的每一次请求服务在服务器停留的时间可能会有较大的差异，随着工作时间的延伸，如果采用简单的轮询算法，每一台服务器上的连接进程数目可能会产生极大的不同，这样实际上并没有达到真正的负载均衡。最小连接调度是一种动态调度算法，它通过服务器当前所活跃的连接数来估计服务器的负载情况。最小连接调度是一种动态调度算法，它通过服务器当前所活跃的连接数来估计服务器的负载情况。调度器需要记录各个服务器已建立连接的数目，当一个请求被调度到某台服务器，其连接数加1; 当连接中止或超时，其连接数减一。加权最小连接数算法是在最小连接数调度算法的基础上，根据服务器的不同处理能力，给每个服务器分配不同的权值，使其能够接受相应权值数的服务请求。

  1） 假设各台 RS 的权值依次为 wi,当前连接数依次为 ci,依次计算 ci/wi,值最小的 RS 作为下一个分配的 RS

  2） 如果存在ci/wi相同的 RS，这些 RS 再使用加权轮询的方式调度

- **优势**：此种均衡算法适合处理长时的请求服务，如 FTP 等应用。
- **劣势**：相较于加权轮询算法，加权最小连接数算法需要保存服务器现有的连接数目，它是一种有状态调度。
- **适用场景**：每个请求所占用的后端时间相差较大的场景，常用于长连接服务。
- **用户推荐**：如果用户需要处理不同的请求，且请求所占用后端时间相差较大，如 2 ms 和 2s 这种数量级的差距时，推荐使用加权最小连接数算法实现负载均衡。

均衡算法选取及权重配置实例

为了让用户在不同场景下，能够让 RS 集群稳定的承接业务，我们给出几个负载均衡选择与权重配置的实例供用户进行参考。

场景1：

若用户首次接触云服务，且建站时间不长，网站负载较低，则建议购买相同配置的 RS，因此 RS 都是无差别的接入层服务器。在此场景下，用户可以将 RS 的权重设置为相同的值，采用加权轮询的方式进行流量分发。

场景2：

设有2台配置相同（CPU 和 内存）的 RS，由于性能一致，用户可以将 RS 权重都设置为10。设现在每台 RS 与客户端端建立了50个 TCP 连接，此时新增一台 RS。在此场景下，推荐用户使用最小连接数的均衡方式，这样能快速的提升新加入 RS 的负载，降低另外2台 RS 的压力。

场景3： 用户有4台服务器，用于承载简单的静态网站访问，且4台服务器的计算能力的比例为 6：3：2：1（按CPU、内存换算）。在此场景下，用户可以依次将 RS 权重比例设置为60，30，20，10，由于静态网站访问大多数是短连接请求，因此可以采用加权轮询的均衡方式，让 SLB 按 RS 的性能比例分配请求。

场景4： 某用户有12台 RS 用于承担海量的 WEB 访问请求，且不希望多购置 RS 增加支出。 某台 RS 经常会因为负载过高，导致服务器重启。在此场景下，建议用户根据 RS 的性能设置相应的权重，给负载过高的 RS 设置较小的权值。除此之外，可以采用最小连接数的负载均衡方式，将请求分配到活跃连接数较少的 RS 上，从而解决某台 RS 负载过高的问题。

场景5： 某用户有3台 RS 用于处理若干长连接请求，且这3太服务器的计算能力比例为4：2：1（按CPU、内存换算）。此时性能最好的服务器处理请求较多，用户不希望过载此服务器，希望能够将新的请求分配到空闲服务器上。在此场景下，可

以采用加权最小连接数的均衡方式，并适当降低繁忙服务器的权重，便于 SLB 将请求分配到活跃数较少的 RS 上，实现负载均衡。

SSL 证书格式要求及格式转换说明

证书格式要求

- 用户要申请的证书为：linux 环境下 pem 格式的证书。
- 如果是通过 root CA 机构颁发的证书， 您拿到的证书为唯一的一份，不需要额外的证书，配置的站点即可被浏览器等访问设备认为可信。
- 如果是通过中级 CA 机构颁发的证书，您拿到的证书文件包含多份证书，需要人为的将服务器证书与中间证书合并在一起上传。
- 当您的证书有证书链时，请将证书链内容，转化为 PEM 格式内容，与证书内容合并上传。
- 拼接规则为：服务器证书放第一份，中间证书放第二份，中间不要有空行。注：一般情况下，机构在颁发证书的时候会有对应说明， 请注意规则说明。

以下为证书格式和证书链格式范例，请确认格式正确后上传：

1、root CA 机构颁发的证书：证书格式为 linux 环境下 pem 格式。样例如下：

证书规则为：

- [————BEGIN CERTIFICATE————, ————END CERTIFICATE————] 开头和结尾；请将这些内容一并上传；
- 每行 64 字符，最后一行不超过 64 字符；

2、中级机构颁发的证书链： -----BEGIN CERTIFICATE----- -----END CERTIFICATE----- -----BEGIN CERTIFICATE----- -----END CERTIFICATE----- -----BEGIN CERTIFICATE----- -----END CERTIFICATE-----

证书链规则：

- 证书之间不能有空行；
- 每一份证书遵守第一点关于证书的格式说明；

RSA 私钥格式要求

样例如下：

rsa 私钥规则：

- [-----BEGIN RSA PRIVATE KEY-----, -----END RSA PRIVATE KEY-----] 开头结尾；请将这些内容一并上传；
- 每行64字符，最后一行长度可以不足64字符。

如果您不是按照上述方案生成私钥，得到[-----BEGIN PRIVATE KEY-----, -----END PRIVATE KEY-----] 这种样式的私钥，您可以按照如下方式转换：

```
openssl rsa -in old_server_key.pem -out new_server_key.pem
```

然后将 new_server_key.pem 的内容与证书一起上传。

证书转换为 PEM 格式说明

目前负载均衡只支持PEM格式的证书，其他格式的证书需要转换成PEM格式后才能上传到负载均衡中，建议通过openssl 工具进行转换。下面是几种比较流行的证书格式转换为PEM格式的方法。

DER 转换为 PEM

DER 格式一般出现在 java 平台中。

证书转换：`openssl x509 -inform der -in certificate.cer -out certificate.pem`

私钥转换：`openssl rsa -inform DER -outform PEM -in privatekey.der -out privatekey.pem`

P7B 转换为 PEM

P7B格式一般出现在 windows server 和 tomcat 中。

证书转换：`openssl pkcs7 -print_certs -in incertificat.p7b -out outcertificate.cer`

获取outcertificat.cer里面 [-----BEGIN CERTIFICATE-----， -----END CERTIFICATE-----] 的内容作为证书上传。

私钥转换：无私钥

PFX 转换为 PEM

PFX 格式一般出现在 windows server 中。

证书转换：`openssl pkcs12 -in certname.pfx -nokeys -out cert.pem`

私钥转换：`openssl pkcs12 -in certname.pfx -nocerts -out key.pem -nodes`

七层负载均衡后端主机获取用户的真实IP方法

apache、nginx、tomcat 日志中获取用户的真实IP方法

- 由于 4 层负载均衡（ TCP 协议）服务可以直接在后端服务器上获取来访者真实 IP 地址，无需进行额外的配置，以下介绍的内容均是针对 7 层（ HTTP 协议）的负载均衡服务而言。
- 7 层负载均衡系统提供 X-Forwarded-For 的方式获取访问者真实 IP ，LB 侧默认开启，需要后端服务做相应配置来获取 client ip 。 以下针对常见的应用服务器配置方案进行介绍。

Apache配置方案

Windows 2003 Server + Apache 解决方案：

1. 打开文件：\apache\conf\httpd.conf。
2. 在文件中查找：”CustomLog”,找到如下配置块: 查看到当前使用的LogFormat为”combined”(如果实际启用的为其他日志格式，替换相应的格式定义即可)。
3. 在文件中查找：”LogFormat”,找到如下配置块（combined格式定义）:

```
LogFormat "%h %l %u %t \"%r\" %>s %b \"%{Referer}i\" \"%{User-Agent}i\"" combined
```

将其修改为：

```
LogFormat "%h %l %u %t \"%r\" %>s %b \"%{Referer}i\" \"%{User-Agent}i\"  \"%{X-Forwarded-For}i\"" combined
```

4. 保存并关闭文件 \apache\conf\httpd.conf。
5. 重启 Apache 服务。

Linux + Apache 解决方案：

1. 打开文件：/etc/httpd/conf/httd.conf。
2. 在文件中查找：”CustomLog”,找到如下配置块: 查看到当前使用的 LogFormat 为”combined” (如果实际启用的为其他日志格式，替换相应的格式定义即可)。

```
#
# For a single logfile with access, agent, and referer information
# (Combined Logfile Format), use the following directive:
#
CustomLog  logs/access_log combined
```

3. 在文件中查找：”LogFormat”,找到如下配置块（combined格式定义）:

```
LogFormat "%h %l %u %t \"%r\" %>s %b \"%{Referer}i\" \"%{User-Agent}i\"" combined
```

将其修改为：

```
LogFormat "%h %l %u %t \"%r\" %>s %b \"%{Referer}i\" \"%{User-Agent}i\" \"%{X-Forwarded-For}i\" " combined
```

4. 保存并关闭文件 /etc/httpd/conf/httd.conf。
5. 重启 Apache 服务。

2. Nginx 配置方案

解决方案如下：

6. 打开文件：/etc/nginx/nginx.conf。
7. 在文件中查找：”CustomLog”,找到如下配置块:

```
server {
    listen   80; ## listen for ipv4
    listen   [::]:80 default ipv6only=on; ## listen for ipv6
    server_name  localhost;
    access_log  /var/log/nginx/localhost.access.log main;
```

8. 将 access_log 这一行替换为如下内容:

```
log_format main '$remote_addr - $remote_user [$time_local] '
            '"$request" $status $body_bytes_sent "$http_referer" '
           '"$http_user_agent" "$http_x_forwarded_for"' ;
access_log /var/log/nginx/localhost.access.log  main;
```

9. 保存并关闭文件 /etc/nginx/nginx.conf。

10. 重启 Nginx 服务。

Tomcat 日志中获取访客真实IP的解决方案

11. 修改 tomcat 的 server.xml ，如：vi /usr/local/tomcat7/conf/server.xml。

```
        Note: The pattern used is equivalent to using pattern="common" -->
      <Valve className="org.apache.catalina.valves.AccessLogValve" directory="logs"
             prefix="localhost_access_log." suffix=".txt"
             pattern='%{X-Forwarded-For}i %h %l %u %t "%r" %s %b' />
```

2. 修改 pattern 为 pattern='%{X-Forwarded-For}i %h %l %u %t "%r" %s %b'，则会记录 headers 头中的 X-Forwarded-For 信息。

```
        Note: The pattern used is equivalent to using pattern="common" -->
      <Valve className="org.apache.catalina.valves.AccessLogValve" directory="logs"
             prefix="localhost_access_log." suffix=".txt"
             pattern="%h %l %u %t &quot;%r&quot; %s %b" />
```

3. 然后重启 tomcat 即可。

测试结果如下：

本次测试中 100.97 是修改前的 slb 内网地址，182.92 是修改后的获取的真实地址。

```
[root@iZ25b7u5z1aZ logs]# tail -f localhost_access_log.2015-06-21.txt |grep -v "test.html"
100.97.137.213 - - [21/Jun/2015:04:44:26 +0800] "GET /docs/security-howto.html HTTP/1.0" 200 39835
100.97.137.246 - - [21/Jun/2015:04:44:26 +0800] "GET /docs/images/tomcat.gif HTTP/1.0" 200 2066
100.97.136.139 - - [21/Jun/2015:04:44:26 +0800] "GET /docs/images/asf-logo.gif HTTP/1.0" 200 7279
100.97.136.183 - - [21/Jun/2015:04:49:19 +0800] "GET /jsp.jsp HTTP/1.0" 200 14935
100.97.136.6 - - [21/Jun/2015:04:49:32 +0800] "GET /favicon.ico HTTP/1.0" 304 -
100.97.136.34 - - [21/Jun/2015:04:49:43 +0800] "GET /jsp.jsp HTTP/1.0" 200 14934
182.92.253.20 100.97.136.231 - - [21/Jun/2015:04:53:29 +0800] "GET /jsp.jsp HTTP/1.0" 200 14935
182.92.253.20 100.97.135.223 - - [21/Jun/2015:04:53:31 +0800] "GET /docs/security-howto.html HTTP/
1.0" 304 -
182.92.253.20 100.97.135.237 - - [21/Jun/2015:04:53:31 +0800] "GET /docs/images/tomcat.gif HTTP/1.
0" 304 -
182.92.253.20 100.97.136.46 - - [21/Jun/2015:04:53:31 +0800] "GET /docs/images/asf-logo.gif HTTP/1
.0" 304 -
```

健康检查异常排查思路

四层排查

TCP协议下，负载均衡使用 SYN包进行探测；

在监听器页面，选中查看的监听器，选择【服务器信息】选项，查看负载均衡后端服务器的健康状态，若不健康，排查思路如下：

- 确定 SLB 后端服务器是否有配置防火墙影响了服务，如果有请关闭
- 使用 netstat 命令，确定后端服务器的端口是否有进程在监听，若未启动，则重新启动服务

七层排查

针对7层（HTTP协议）服务，当某一监听出现健康检查状态为“不健康”时，可以通过如下方面进行排查：

- 由于负载均衡的七层健康检查服务与后端 KEC 之间的通讯是走内网的，您需要登录服务器检查应用服务器端口是否正常监听在内网地址上，如果没有监听在内网地址，请将应用服务器端口监听到内网上，从而确保负载均衡系统和后端 KEC 之间的通讯正常。

假设负载均衡前端端口是80，KEC 后端端口也是80，KEC 内网IP是：10.1.1.2

Windows系统服务器使用如下命令：

```
netstat -ano | findstr :80
```

Linux系统服务器使用如下命令：

```
netstat -anp | grep :80
```

如果能看到 10.1.1.2:80 的监听或 0.0.0.0:80 的监听则说明这部分正常。

- 请确保后端服务器开启了相应的端口，该端口必须与您在负载均衡监听配置中配置的后端端口保持一致。

如果是4层负载均衡，只要后端端口 telnet有响应即可，可以使用telnet 10.1.1.2 80来测试。如果是7层负载均衡，需要 HTTP 状态码是200 等代表正常的状态码。检验方法如下：

Windows系统可以直接在 KEC 内的浏览器输入内网IP测试是否正常，本例为：http://10.1.1.2 Linux系统可以通过curl -I命令看看状态是否为HTTP/1.1 200 OK，本例是：curl -I 10.1.1.2

- 检查后端 KEC 内部是否有防火墙或其他安全类防护软件，这类软件很容易将负载均衡系统的本地IP地址屏蔽，从而导致负载均衡系统无法跟后端服务器进行通讯。

检查服务器内网防火墙是否放行80端口，可以暂时关闭防火墙进行测试。

Windows系统可以运行输入firewall.cpl操作关闭 Linux系统可以输入/etc/init.d/iptables stop关闭

- 检查负载均衡健康检查参数设置是否正确，请参考监听器 健康检查 文档。
- 健康检查指定的检测文件，建议是以html形式的简单页面，只用于检查返回结果，不建议用php等动态脚本语言。
- 检查后端是否有较高负载导致 KEC 对外提供服务响应慢。

会话保持原理

什么是会话保持？

会话保持是负载均衡最常见的问题之一，会话保持是指在负载均衡器上的一种机制，可以识别客户端与服务器之间交互过程的关联性，在负载均衡的同时保证一系列相关联的访问请求会分配到一台服务器上

连接和会话的区别？

在讨论会话保持前，我们先区分一些概念：什么是连接（Connection）、什么是会话（Session）。需要特别强调的是，如果我们仅仅是谈论负载均衡，会话和连接往往具有相同的含义。从简单的角度来看，如果用户需要登录，那么就可以理解为会话；如果不需要登录，那么就是连接。

什么时候需要会话保持？

对于同一个连接中的数据包，负载均衡会将其进行 NAT 转换后，转发至后端固定的服务器进行处理。负载均衡系统内部会专门有一张表来记录这些连接的状况，包括：[源IP：端口]、[目的IP：端口]、[服务器IP：端口]、空闲超时时间（Idle Timeout）等等。由于负载均衡内部记录连接状态的这张表需要消耗系统的内存资源，因此这张表不可能无限大，所有传统厂商都会有一定的限制。这张表的大小一般称之为最大并发连接数，也就是系统同时能够容纳的连接数量。负载均衡的当前连接状态表项中，设计了一个空闲超时时间（Idle Timeout）的参数。当该连接在Idle Timeout 内无流量通过时，负载均衡会自动删除该连接条目，释放系统资源。

删除连接后，客户端的请求将无法保证继续发往同一个后端服务器，需要遵循负载均衡器的流量分发策略。

在某些要求登录状态的情境下，要求客户端和服务器之间保持一个会话（session）以记录客户端的各种信息。比如在大多数电子商务的应用系统或者需要进行用户身份认证的在线系统中，一个客户端与服务器通常需要经过好几次的交互过程才能完成一笔交易或者是一个请求的完成。由于这几次交互过程是密切相关的，服务器在进行这些交互过程的某一个交互步骤时往往需要了解上一次或上几次的交互过程处理结果，这就要求所有这些相关的交互过程都由一台服务器完成，而不能被负载均衡器分散到不同的服务器上　否则可能出现异常情景：

- 客户端输入了正确的用户名和密码，但却反复跳到登录页面；
- 用户输入了正确的验证码，但是总提示验证码错误；
- 客户端放入购物车的物品丢失；

因此会话保持机制的意义就在于，确保在合适的情境下，将来自相同客户端的请求转发至后端相同的服务器进行处理。如果在客户端和服务器之间部署了负载均衡设备，很有可能这多个连接会被转发至不同的服务器进行处理。如果服务器之间没有会话信息的同步机制，会导致其他服务器无法识别用户身份，造成用户在和应用系统发生交互时出现异常。

负载均衡希望将来自客户端的连接、请求均衡的转发至后端的多台服务器，以避免单台服务器负载过高；而会话保持机制却要求将某些请求转发至同一台服务器进行处理。因此，在实际的部署环境中，我们要根据应用环境的特点，选择适当的会话保持机制。

会话保持的分类

简单会话保持（四层会话保持）

简单会话保持（也称作基于源地址的会话保持、基于 IP 的会话保持）是指负载均衡器在作负载均衡时根据访问请求的源地址作为判断关联会话的依据。对来自同一IP地址的所有访问请求在作负载均时都会被保持到一台服务器上去

简单会话保持中一个很重要的参数就是连接超时值，负载均衡器会为每一个处于保持状态中的会话设定一个时间值。若一个会话从上一次完成到下次再来之间的间隔时间小于超时值时，负载均衡器将会将新的连接进行会话保持；但如果这个间隔大于该超时值，负载均衡器会将新来的连接认为是新的会话然后进行负载平衡。

简单会话保持实现简单，只需要根据数据包三　四层的信息就可以实现，效率比较高

但此种方式存在的问题就在于，当多个客户端通过代理或地址转换的方式访问服务器时，由于来源地址一样，请求都被分配到同一台服务器上，会导致服务器之间的负载严重失衡。

另外一种情况是，同一个客户端产生大量并发，要求分配到多个服务器上处理的同时进行会话保持。这时基于客户端源地址的会话保持方法也会导致负载均衡失效。

以上情况出现时，就必须要考虑使用其他的会话保持方式。

4.2. 存会话（Session）的会话保持

此种方式通过多个后端服务器共享 Session 的方式，实现与负载均衡同时的会话保持。主要有以下几种形式：

1） 数据库存放

Session 信息存储到数据库表以实现不同应用服务器间 Session 信息的共享。此种方式适合数据库访问量不大的网站。

- 优点：实现简单
- 缺点：由于数据库服务器相对于应用服务器更难扩展且资源更为宝贵，在高并发的 Web 应用中，最大的性能瓶颈通常出现在数据库服务器。因此如果将 Session 存储到数据库表，频繁的数据库操作会影响业务。

2） 文件系统存放

通过文件系统（比如NFS）来实现各台服务器间的 Session 共享。此种方式适合并发量不大的网站。

- 优点：各台服务器只需要 mount 存储 Session 的磁盘即可，实现较为简单。
- 缺点：NFS 对高并发读写的性能并不高，在硬盘 I/O 性能和网络带宽上存在较大瓶颈，尤其是对于 Session 这样的小文件的频繁读写操作。

3） Memcached 存放

利用 Memcached 来保存 Session 数据，直接通过内存的方式读取。

- 优点：效率高，在读写速度上会比存放在文件系统时快很多，而且多个服务器共用 Session 也更加方便，将这些服务器都配置成使用同一组 memcached 服务器就可以，减少了额外的工作量。
- 缺点：一旦宕机内存中的数据将会丢失，但对 Session 数据来说并不是严重的问题。如果网站访问量太大、Session 太多的时候 memcached 会将不常用的部分删除，但是如果用户隔离了一段时间之后继续使用，将会发生读取失败的问题。

基于cookie的会话保持（七层会话保持）

基于cookie的会话保持可使用植入 cookie 和重写 cookie 来进行会话保持。

植入 cookie 是什么？

植入 cookie 是指由负载均衡服务器来给客户端设置 cookie，即 HTTP/HTTPS 响应报文中插入 SERVERID 字串和客户配置时指定的超时时间，在此时间内会将同一客户端的请求传入到同一个后端 KEC 服务器，当客户端浏览器再次通过此 cookie 访问时，负载均衡不会传给后端的 KEC 服务器，即插入 cookie 关键字与值对后端 KEC 来说是完全不需要知道的。

重写 cookie 是什么？

重写 cookie 是指负载均衡实例的拥有者可以按照自己的需要自定义在后端的 KEC 服务器回复 HTTP/HTTPS 响应中插入 cookie 关键字与值，后端的 KEC 服务器上同时需要维护此 cookie 的超时时间与生存时间，在此响应报文经过负载均衡时，负载均衡会基于一定规则重写 cookie 的值字串用于会话保持，当携带 cookie 关键字与值的请求到来时会将此 cookie 关键字与值传入到初始插入 cookie 的后端 KEC 服务器；但 cookie 值的内容已经与初始相比已经改变。

金山云 SLB 的7层会话保持能力，可以选择重写 cookie 或者植入 cookie 的方式实现。

cookie原理说明

什么是cookie?

HTTP 协议是无状态的，也就是说客户端和服务器端不需要建立持久的连接。由于客户端和服务器端的连接是基于一种请求应答模式，即第一步：客户端和服务器建立一个连接；第二步：客户端提交一个请求；第三步：服务器端收到请求后返回一个响应，第四步：断开连接。

若客户端和服务器在完成一次请求以后就断开了连接，二者之间就不再有任何关系了；那么，当用户在页面1进行了登录后跳转到了同一个 Web 应用的页面2时，如何在页面2知道用户已经进行了登录呢？即当客户端再次发起请求的时候，服务器端如何判断两次不同的请求来自同一个客户端呢？

HTTP协议下，服务器是无法区分每一次请求之间的联系的。要判断这种联系就需要有一个状态来标识每一次请求，如果两次请求的状态标识是一样的，这就表明这两个请求是从同一个客户端发起的。

Cookie就是这样一个用来标识每一次请求的状态位。经过多年的发展Cookie变得越来越规范，后来直接成为了一个通用标准。

cookie 的工作原理

1) 当首次向金山云发起请求时，HTTP请求头如下：

```
Accept:text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,/;q=0.8
Accept-Encoding:gzip, deflate, sdch
Accept-Language:en,zh-CN;q=0.8,zh;q=0.6
Connection:keep-alive
Host:www.ksyun.com
```

2) 请求到达金山云的服务器以后，金山云的服务器生成响应，并在响应的头部写入cookie信息：

```
Set-Cookie:BD_HOME=1; path=/
Set-Cookie:__bsi=14934756243064632384_00_0_I_R_174_0303_C02F_N_I_I_0; expires=Thu, 19-Nov-16 16:20:45 GMT; domain=www.ksyun; path=/
Set-Cookie:BDSVRTM=172; path=/
```

3) 当客户端浏览器接收到响应头以后，会将cookie信息写入本地进行管理。

4) 再次向服务器发起请求时，客户端通过发送一个带有Cookie: name=value; name2=value2的HTTP请求头将之前存在本地的cookie一起发送过去。请求的头部信息为：

```
Accept:text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,/;q=0.8
Accept-Encoding:gzip, deflate, sdch
Accept-Language:en,zh-CN;q=0.8,zh;q=0.6
Connection:keep-alive
Cookie:BD_HOME=1; BDSVRTM=0; BD_LAST_QID=15071962345319158759570 57
Host:www.ksyun.com
```

5) 服务器接收到请求以后，从请求头中获得cookie信息，分析cookie数据后向客户端返回响应。

以上就是cookie在客户端和服务器之间进行传递信息的基本过程。

cookie的生命周期

cookie 是有生命周期的。一旦到了cookie 的失效日期，客户端的 cookie 就会被删除，服务器在创建 cookie 时可以控制一个 cookie 在客户端的“存活”时间。在以下几种情况下，cookie 会结束自己的生命周期：

- 未指定过期时间的 cookie。当服务器创建一个 cookie 的时候没有指定过期时间时，客户端会将这类 cookie 写入浏览器开辟的一块内存中，当关闭浏览器以后，这块内存也就被释放了，对应的 cookie 也就是结束了它的生命；
- 指定过期时间的cookie。当服务器创建一个 cookie 的时候指定了对应的过期时间时，当到达了过期时间时，对应的 cookie 就会被删除；

- 当浏览器中的 cookie 数量达到了限制时。浏览器会按照某种策略删除一些旧的 cookie，为新的 cookie 腾出空间；
- 人为删除cookie。

cookie管理

服务器端创建一个cookie时，一般都会指定以下两个选项：

- domain
- path

这两个选项决定了创建的 cookie 属于哪个域名下的哪个位置。

domain 选项

默认情况下，domain 会被设置为创建该 cookie 的页面所在的域名。当客户端再次给相同域名发送请求时，cookie 会一起被发送至服务器。当 cookie 的 domain 选项被设置为一个一级域名时，此域名下的所有二级都将同时拥有相同的 cookie，经常会出现顶级域名和二级域名的 cookie 冲突问题。

我们在发送请求时，浏览器会把 domain 的值与请求的域名做一个比较，并将匹配的 cookie 发送至服务器。

- 当我们未指定 domain 时，默认的 domain 为访问地址的域名。如果是顶级域名访问，那么设置的 cookie 也可以被其他二级域名所共享，因此登录等操作一般都在顶级域名下进行操作。
- 二级域名可以读取设置了 domain 为顶级域名或者自身的 cookie，但是不能读取其他二级域名 domain 的 cookie，因此想要 cookie 在多个二级域名中共享的时候，需要设置 domain 为顶级域名，这样就可以在所有二级域名里面使用该 cookie。这里需要注意的是顶级域名只能获取到 domain 设置为顶级域名的 cookie，无法获取 domain 设置为二级域名的 cookie。

path选项

path 选项规定，客户端请求的 URL 只有在 path 指定的路径时，才会发送 cookie 消息头，它决定了客户端发送 cookie 到服务器端的匹配规则。通常是将 path 选项的值与请求的 URL 从头开始逐字符比较，如果字符匹配，则发送 cookie 消息头。需要注意的是，只有在 domain 选项满足之后才会对 path 属性进行比较。path属性的默认值是发送 Set-Cookie 消息头所对应的 URL 中的 path 部分。

以上从浏览器本身的限制和生成 cookie 时的选项对 cookie 的管理进行了简单的总结。接下来就通过一些简单的代码来演示如何创建和获取 cookie。

服务器端创建cookie

金山云服务器通过发送一个带有Set-Cookie的 HTTP 消息响应头来创建一个 cookie。例如：

```
// 创建一个cookie对象
Cookie co = new Cookie(“site”, “http://www.ksyun.com “);
co.setDomain(“mine.com”);
// 通过响应头，将cookie发送到客户端
response.addCookie(co);

Cookie co = new Cookie(“site”, “http://www.ksyun.com “);
co.setDomain(“mine.com”);
co.setPath(“/homes”);
co.setMaxAge(3600); // 单位为秒
co.setHttpOnly(true);
co.setSecure(false);
response.addCookie(co);
```

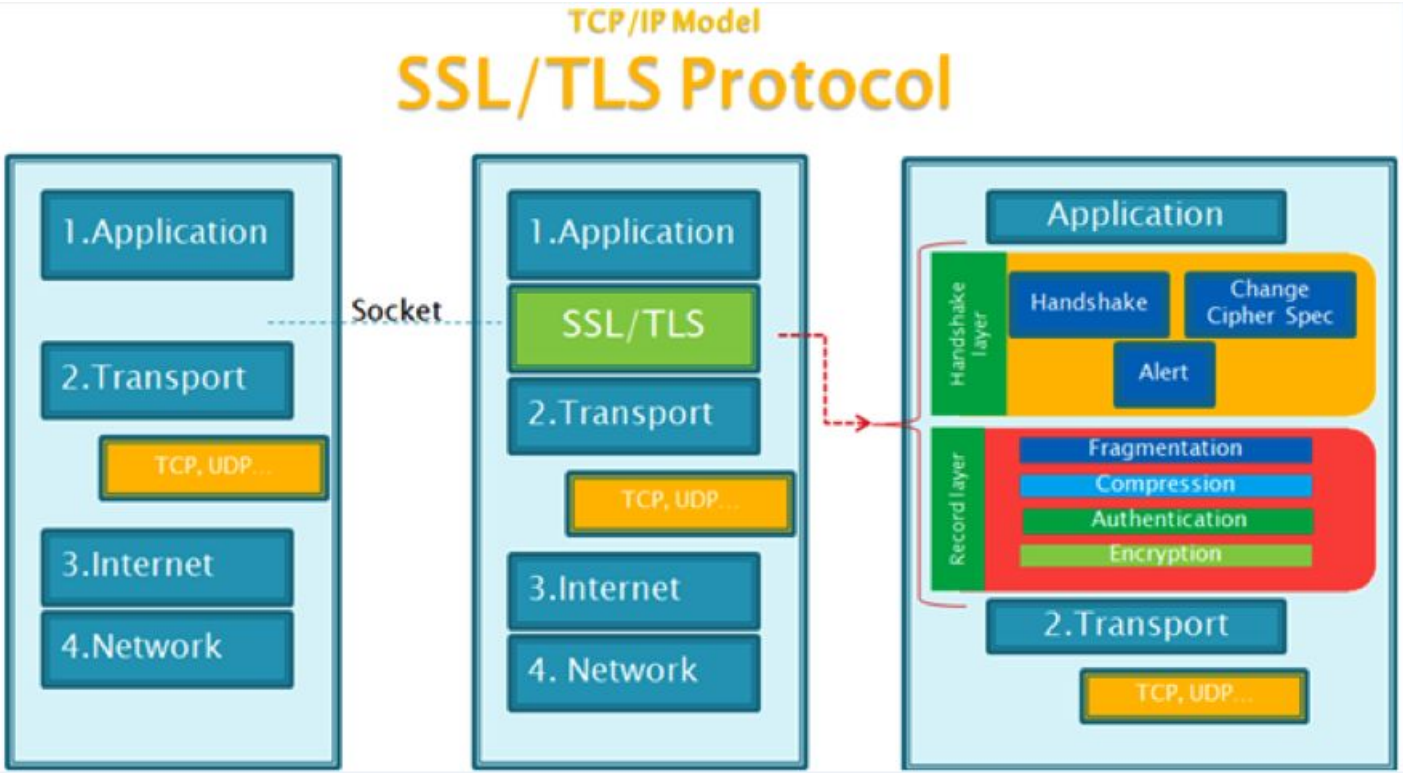
客户端读取cookie

客户端向服务器发起请求时，在 domain 和 path 匹配的情况下，会将对应的 cookie 一起发送到服务器端。如果一个 path 下设置的 cookie 太多，就可能出现 http 请求头超长的问题。请求到达服务器端以后，我们可以这样读取cookies：

```
Cookie[] cookies = request.getCookies();
if (cookies != null) {
for (int i = 0; i < cookies.length; ++i) {
// 获得具体的Cookie
Cookie cookie = cookies[i];
// 获得Cookie的名称
String name = cookie.getName();
String value = cookie.getValue();
out.print(“Cookie名:” + name + ”    Cookie值:” + value + “
”);
}
}
```

SSL 原理说明

SSL/TLS 是一个介于应用层（HTTP 协议）与传输层（TCP 协议）之间的一个可选协议，其协议架构可参照下图：



当HTTP通信不使用 SSL/TLS 时，所有信息均以明文形式传播，会有以下风险：

- 窃听风险：第三方可以获得通信内容
- 篡改风险：第三方可以修改通信内容
- 冒充风险：第三方可以冒充他人身份参与通信

SSL/TLS 协议可以解决这些通信风险，协议设计的目标为：

- 所有信息都是加密传播，第三方无法窃听。
- 具有校验机制，一旦被篡改，通信双方可以立即发现。
- 配备身份证书，防止身份被冒充。

目前，主流浏览器都已经支持了 SSL/TLS 的支持。

SSL/TLS 协议基本运行过程

SSL/TLS 协议的基本思路是采用公钥加密的方法。即客户端先向服务器端索要公钥，然后使用公钥加密信息并发送至服务器端；服务器收到密文后，用自己的私钥解密。

这里需要解决两个问题：

1）如何保证公钥不被篡改？

解决方法：将公钥放在数字证书中。只要证书是可信的，公钥就是可信的。

2）公钥加密计算量太大，如何减少耗用的时间？

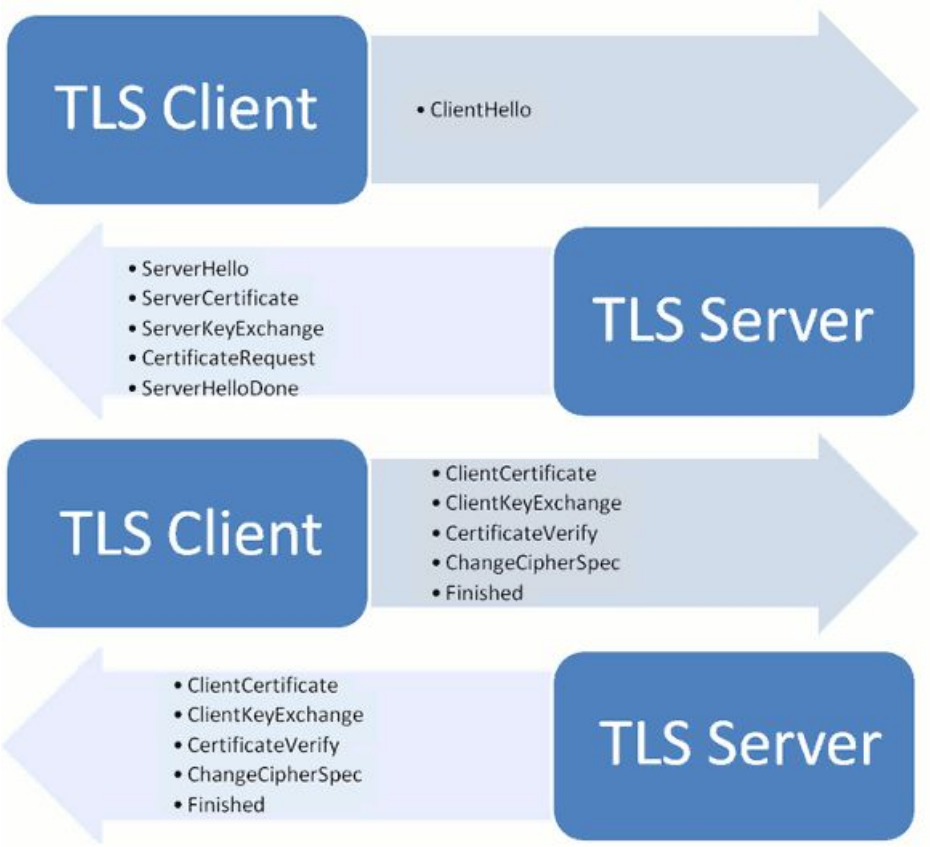
解决方法：每一次对话（session），客户端和服务器端都生成一个"对话密钥"（session key），用它来加密信息。由于"对话密钥"是对称加密，所以运算速度非常快，而服务器公钥只用于加密"对话密钥"本身，这样就减少了加密运算的消耗时间。

SSL/TLS协议的基本过程是这样的：

- 客户端向服务器端索要并验证公钥
- 双方协商生成"对话密钥"
- 双方采用"对话密钥"进行加密通信。

上面过程的前两步，又称为"握手阶段"（handshake）。

握手阶段的详细过程



"握手阶段"涉及四次通信且所有通信都是明文传输的。

客户端发出请求（ClientHello）

客户端（通常是浏览器）先向服务器发出加密通信的请求，通常被称为ClientHello请求。

在这一步，客户端主要向服务器提供以下信息。

- 支持的协议版本，比如TLS 1.0
- 一个客户端生成的随机数，稍后用于生成"对话密钥"
- 支持的加密方法，比如RSA公钥加密
- 支持的压缩方法

这里需要注意的是，客户端发送的信息之中不包括服务器的域名。也就是说，理论上服务器只能包含一个网站，否则会分不清应该向客户端提供哪一个网站的数字证书。这就是为什么通常一台服务器只能有一张数字证书的。

对于虚拟主机的用户来说，这当然很不方便。2006年，TLS 协议加入了一个 Server Name Indication 扩展，允许客户端向服务器提供它所请求的域名。

服务器回应（SeverHello）

服务器收到客户端请求后，向客户端发出回应，通常被称为SeverHello。服务器的回应包含以下内容：

- 确认使用的加密通信协议版本，比如TLS 1.0版本。如果浏览器与服务器支持的版本不一致，服务器将关闭加密通信
- 一个服务器生成的随机数，稍后用于生成"对话密钥"
- 确认使用的加密方法，比如RSA公钥加密
- 服务器证书

除了以上信息，若服务器需要确认客户端的身份，则会再包含一项请求，要求客户端提供"客户端证书"。比如，金融机构往往只允许认证客户连入自己的网络，就会向正式客户提供 USB 密钥，里面就包含了一张客户端证书。

客户端回应

客户端收到服务器回应以后会首先验证服务器证书。如果证书不是由可信机构颁布、证书中的域名与实际域名不一致或者证书已经过期，就会向访问者显示一个警告，由其选择是否还要继续通信。

如果证书没有问题，客户端就会从证书中取出服务器的公钥并向服务器发送以下信息：

- 一个随机数。该随机数用服务器公钥加密，防止被窃听。
- 编码改变通知，表示随后的信息都将用双方商定的加密方法和密钥发送。
- 客户端握手结束通知，表示客户端的握手阶段已经结束。这一项同时也是前面发送的所有内容的 hash 值，用来供服务器校验。

上面第一项的随机数，是整个握手阶段出现的第三个随机数，又称 "pre-master key"。有了它以后，客户端和服务器就同时有了三个随机数，接着双方就用事先商定的加密方法，各自生成本次会话所用的同一把"会话密钥"。

对于RSA密钥交换算法来说，pre-master-key 本身就是一个随机数，再加上 hello 消息中的随机，三个随机数通过一个密钥导出器最终导出一个对称密钥。

pre-master 的存在原因是 SSL 协议不信任每个主机都能产生完全随机的随机数。客户端和服务器加上 pre-master 三个随机数一同生成的密钥不容易被猜出了。因为一个伪随机可能完全不随机，可是三个伪随机则十分接近随机。

此外，如果前一步，服务器要求客户端证书，客户端会在这一步发送证书及相关信息。

服务器的最后回应

服务器收到客户端的第三个随机数 pre-master-key 之后，计算生成本次会话所用的"会话密钥"。然后，向客户端最后发送如下信息：

- 编码改变通知，表示随后的信息都将用双方商定的加密方法和密钥发送。
- 服务器握手结束通知，表示服务器的握手阶段已经结束。这一项同时也是前面发送的所有内容的hash值，用来供客户端校验。

至此，整个握手阶段全部结束。接下来，客户端与服务器进入加密通信，完全是使用 HTTP 协议，只不过用"会话密钥"加密内容。

HTTP 长连接说明

如何理解 HTTP 协议是无状态的

HTTP 协议是无状态的，指的是协议对于事务处理没有记忆能力，服务器不知道客户端是什么状态。也就是说，打开一个服务器上的网页和上一次打开这个服务器上的网页之间没有任何联系。HTTP 是一个无状态的面向连接的协议，无状态不代表 HTTP 不能保持 TCP 连接，更不能代表 HTTP 使用的是 UDP 协议（无连接协议）。

HTTP 协议与 TCP/IP 协议的关系

HTTP 的长连接和短连接本质上是 TCP 长连接和短连接。HTTP 属于应用层协议，在传输层使用 TCP 协议，在网络层使用 IP 协议。 IP 协议主要解决网络路由和寻址问题，TCP 协议主要解决如何在 IP 层之上可靠地传递数据包，使得网络上接收端收到发送端发出的所有数据包，并且接收顺序与发送顺序一致。TCP 协议是可靠的、面向连接的。

什么是长连接、短连接？

在 HTTP/1.0 中默认使用短连接。也就是说，客户端和服务器每进行一次 HTTP 操作，就建立一次连接，任务结束就中断连接。当客户端浏览器访问的某个 HTML 或其他类型的 Web 页中包含有其他的 Web 资源（如JavaScript文件、图像文件、CSS文件等），每遇到这样一个 Web 资源，浏览器就会重新建立一个 HTTP 会话。

而从 HTTP/1.1 起，默认使用长连接，用以保持连接特性。使用长连接的 HTTP 协议，会在响应头加入这行代码：

```
Connection:keep-alive
```

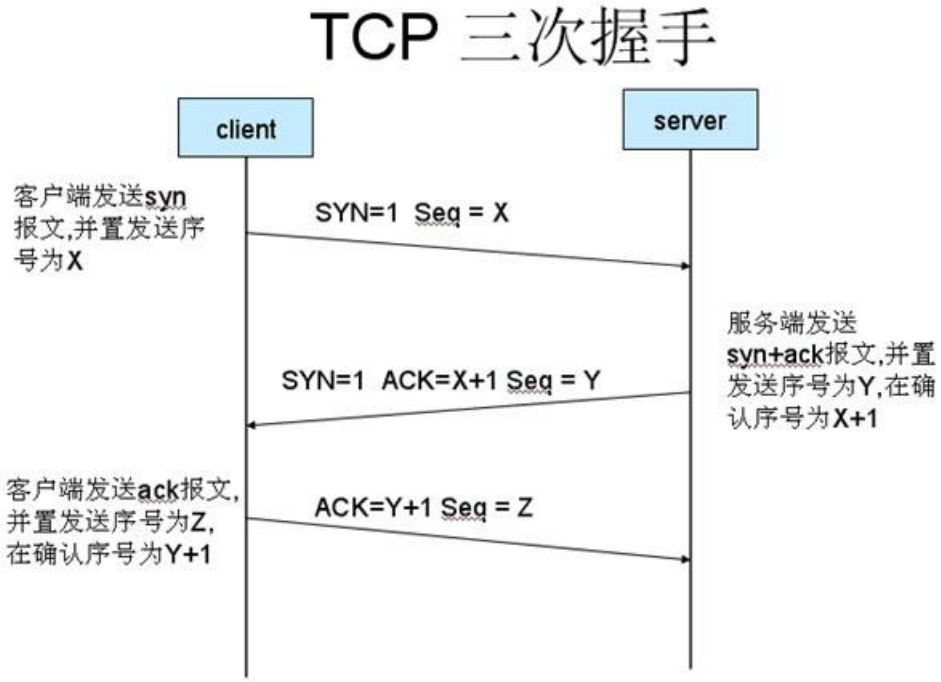
在使用长连接的情况下，当一个网页打开完成后，客户端和服务器之间用于传输 HTTP 数据的 TCP 连接不会关闭，客户端再次访问这个服务器时，会继续使用这一条已经建立的连接。Keep-Alive 不会永久保持连接，它有一个保持时间，可以在不同的服务器软件（如Apache）中设定这个时间。长连接的实现需要客户端和服务端都支持长连接。

HTTP 协议的长连接和短连接，实质上是 TCP 协议的长连接和短连接。

HTTP长连接的请求数量限定是最多连续发送100个请求，超过限定将关闭这条连接。

TCP 连接

当网络通信时采用 TCP 协议时，在真正的读写操作之前，客户端与服务器端之间必须建立一个连接，当读写操作完成后，双方不再需要这个连接时可以释放这个连接。连接的建立依靠“三次握手”，所以每个连接的建立都需要消耗时间和资源。



经典的三次握手建立连接示意图：

TCP短连接

TCP 短连接的情况如下：client 向 server 发起连接请求，server 接到请求，然后双方建立连接。client 向 server 发送消息，server 回应 client，然后一次请求就完成了。这时候双方任意都可以发起 close 操作，不过一般都是 client 先发起 close 操作。上述可知，短连接一般只会在 client/server 间传递一次请求操作。

短连接的优点是：管理简单，存在的连接都是有用的连接，不需要额外的控制手段。

TCP长连接

TCP 长连接的情况如下：client 向 server 发起连接， server 接受 client 连接，双方建立连接，client 与 server 完成一次请求后，它们之间的连接并不会主动关闭，后续的读写操作会继续使用这个连接。

TCP 的保活功能主要为服务器应用提供。如果客户端已经消失而连接未断开，则会使得服务器上保留一个半开放的连接，而服务器又在等待来自客户端的数据，此时服务器将永远等待客户端的数据。保活功能就是试图在服务端器端检测到这种半开放的连接。

如果一个给定的连接在两小时内没有任何动作，服务器就向客户发送一个探测报文段，根据客户端主机响应探测客户端状态，客户端状态有如下四种：

- 客户主机依然正常运行，且服务器可达。此时客户的 TCP 响应正常，服务器将保活定时器复位。
- 客户主机已经崩溃，并且关闭或者正在重新启动。上述情况下客户端都不能响应 TCP。服务端将无法收到客户端对探测的响应。服务器总共发送10个这样的探测。若服务器没有收到任何一个响应，它就认为客户端已经关闭并终止连接。
- 客户端崩溃并已经重新启动。服务器将收到一个对其保活探测的响应，这个响应是一个复位，使得服务器终止这个连接。
- 客户机正常运行，但是服务器不可达。这种情况与第二种状态类似。

长连接和短连接的优点和缺点

由上可以看出，长连接可以省去较多的 TCP 建立和关闭的操作，减少浪费，节约时间。对于频繁请求资源的客户端适合使用长连接。在长连接的应用场景下，client 端一般不会主动关闭连接，当 client 与 server 之间的连接一直不关闭，随着客户端连接越来越多，server 会保持过多连接。这时候 server 端需要采取一些策略，如关闭一些长时间没有请求发生的连接，这样可以避免一些恶意连接导致 server 端服务受损；如果条件允许则可以限制每个客户端的最大长连接数，这样可以完全避免恶意的客户端拖垮整体后端服务。

短连接对于服务器来说管理较为简单，存在的连接都是有用的连接，不需要额外的控制手段。但如果客户请求频繁，将在 TCP 的建立和关闭操作上浪费较多时间和带宽。

长连接和短连接的产生在于 client 和 server 采取的关闭策略。不同的应用场景适合采用不同的策略。

HTTP返回值说明

HTTP 状态码（HTTP Status Code）是用以表示网页服务器 HTTP 响应状态的3位数字代码。它由 RFC 2616 规范定义的，并得到RFC 2518、RFC 2817、RFC 2295、RFC 2774、RFC 4918等规范扩展。

所有状态码的第一个数字代表了响应的五种状态之一：

1xx ： 信息响应类，表示接收到请求并且继续处理 2xx ： 处理成功响应类，表示动作被成功接收、理解和接受 3xx ： 重定向响应类，为了完成指定的动作，必须接受进一步处理 4xx ： 客户端错误，客户请求包含语法错误或者不能正确执行 5xx ： 服务端错误，服务器不能正确执行一个正确的请求

下面给出了常见响应返回值的说明：

100——客户必须继续发出请求 101——客户要求服务器根据请求转换 HTTP 协议版本

200——交易成功 201——提示知道新文件的 URL 202——接受和处理、但处理未完成 203——返回信息不确定或不完整 204——请求收到，但返回信息为空 205——服务器完成了请求，用户代理必须复位当前已经浏览过的文件 206——服务器已经完成了部分用户的 GET 请求

300——请求的资源可在多处得到 301——删除请求数据 302——在其他地址发现了请求数据 303——建议客户访问其他 URL 或访问方式 304——客户端已经执行了GET，但文件未变化 305——请求的资源必须从服务器指定的地址得到 306——前一版本HTTP中使用的代码，现行版本中不再使用 307——申明请求的资源临时性删除

400——错误请求，如语法错误 401——请求授权失败 402——保留有效 ChargeTo 头响应 403——请求不允许 404——没有发现文件、查询或 URl 405——用户在 Request-Line 字段定义的方法不允许 406——请求的资源的内容特性无法满足请求头中的条件，请求资源不可访问 407——类似401，用户必须首先在代理服务器上得到授权 408——客户端没有在用户指定的时间内完成请求 409——对当前资源状态，请求不能完成 410——服务器上不再有此资源且无进一步的参考地址 411——服务器拒绝用户定义的 Content-Length 属性请求 412——一个或多个请求头字段在当前请求中错误 413——请求的资源大于服务器允许的大小 414——请求的资源 URL 长于服务器允许的长度 415——请求资源不支持请求项目格式 416——请求中包含 Range 请求头字段，在当前请求资源范围内没有 range 指示值，请求也不包含 If-Range 请求头字段 417——服务器不满足请求 Expect 头字段指定的期望值，如果是代理服务器，可能是下一级服务器不能满足请求

500——服务器产生内部错误 501——服务器不支持请求的函数 502——服务器暂时不可用，有时是为了防止发生系统过载 503——服务器过载或暂停维修 504——关口过载，服务器使用另一个关口或服务来响应用户，等待时间设定值较长 505——服务器不支持请求头中所使用的HTTP协议版本