11_Http协议

11.1 互联网时代的诞生

早在20世纪70年代的时候，在互联网的雏形——ARPA基础之上， TCP、 IP等协议已经诞生并在计算机世 界当中流行，而时至20世纪80年代，大量的计算机都使用了UNIX操作系统，并使用Berkeley Socket接入到网络当中。在当时的情况下，网络中任意两台计算机之间可以直接进行网络通信。从计算机的层面上来说，信息已经可以在互联网中任意两台计算机之间传递了，但是对用户而言，当时的互联网使用起来非常地不便—— 因为在当时，信息的展示非常不符合人类的阅读习惯，而且也非常难以聚集和传播。

1989 年，任职于欧洲核子研究中心(CERN)的蒂姆·伯纳斯 - 李(Tim Berners- Lee)发表了一篇论文，提出了在互联网上构建超链接文档系统的构想。这篇论文中他确立了三项关键技术:

- URI ：即统一资源标识符，作为互联网上资源的唯一身份;
- HTML：即超文本标记语言，描述超文本文档;

> 普通的文本通常是只能携带文本信息的，如果需要携带图片、音频等多媒体数据，那么就需要将这 些多媒体的二进制数据包含进文本文件当中， 一种自然的做法将这些二进制数据和原本的文本数据 打包在一起(比如富文本文档、 Microsoft Word之类)，但是这种混杂的数据存储模式使得解析 程序的代码编写变得十分困难。

> 超文本的解决方案是不直接包含多媒体的二进制数据，而是一个描述其存储位置的路径来指向目标数据，即链接，链接除了可以充当文件系统中的路径来使用以外，还可以指向互联网上的资源。

> 链接可以指向多媒体数据，还可以指向布局描述文件、网页脚本文件和其他网页。这样一个完整Web文档就能由多个不同的子文档构建形成。

> Web文档一般用HTML文件描述，布局描述文件是CSS文件，网页脚本文件则是由JavaScript语言编写的程序。

- HTTP：即超文本传输协议，用来传输超文本。

这三项技术解决了网络信息在计算机上定位、展示和传播的问题。用户可以使用URI来定位某个网络资源(比如某个网页、一张图片或者其他多媒体资源)，使用浏览器可以从服务器当中获取超文本形式的网络资源并且在计算机上展示，期间用来承载网络资源的协议是超文本协议。这样一套超文本系统完美 地运行在网络上，让各地的人们能够自由地共享信息，蒂姆把这个系统称为“万维网”(WorldWide Web)，也就是我们现在所熟知的 Web。

11.2 HTTP的基本特点

超文本传输协议(**HTTP**) 是一个用于传输超媒体文档(例如 HTML)的协议。它最初是为 Web 浏览器 与 Web 服务器之间的通信而设计的。

HTTP的基本特点如下:

- 载荷数据的类型：最初HTTP携带超文本数据，而随着互联网的不断发展， HTTP也可以用于携带其 他类型(不限于超媒体文档)的数据载荷。载荷数据通称为资源。
- 文本协议： HTTP是文本协议，这意味着HTTP报文的头部数据是文本类型的，用户可以直接阅读报 文内容。
- 传输模型： HTTP采用客户端-服务端的传输模型。
- 协议的分层： HTTP是一种应用层协议，在TCP/IP协议族处于第4层，在ISO/OSI体系中处于第7层， 其依赖的传输层是可靠的协议(比如TCP协议)。

- 状态的保存：　HTTP是一种无状态协议。

11.2.1 客户端-服务端模型

HTTP遵循经典的客户端-服务端模型，客户端打开一个连接以发出请求，然后等待直到收到服务器端响应。其中，永远由客户端先发起请求，而服务端每收到一个请求就对应回复一个响应。 一个请求连同它对应的响应合起来称作事务。从通信模型的角度来看， HTTP和TCP有着显著的差异： TCP会存在一个持续不断的数据流，通信双方按照流的方式交换数据；在HTTP当中，不同的事务之间是彼此独立的。

11.2.2 无状态协议

HTTP 是一种无状态协议，这意味着服务器在事务结束之后不会保留任何数据(即状态)，不同事务在 协议层面上彼此分离。这种设计确实限制了HTTP的使用范围，对于有着复杂状态的业务，比如游戏，HTTP就不适合作为直接承载业务的协议(要么需要选择其他协议，要么基于HTTP构造更上层的协议)。但是对于大多数Web应用，用户可以采用分层的设计，将状态转移到数据库层(比如redis、 mysql)或者是客户端(比如cookie)进行存储，这样的话，用户在处理HTTP事务时，就不需要考虑协议交互之间的状态变更，也不需要考虑其他事务的影响，使用起来更加简洁。

无状态协议有助于服务端的可扩展性，如果需要支持更高的并发量，那么只需要简单地部署更多机器就可以解决问题，即**支持水平扩展。**

11.2.3 可靠性

HTTP一般基于 TCP/IP 层， HTTP报文的可靠性是由传输层提供的。所以HTTP的底层一般是TCP协议，但是新版本的HTTP可以不采用TCP。

11.2.4 文本协议

HTTP协议是文本协议，至少HTTP请求和响应报文的头部是文本形式的。客户端和服务端需要使用字符串解析的方式来处理协议内容。在报文传递过程中，用户可以抓取报文，然后直接阅读报文的内容。这种设计让HTTP报文十分易于阅读，但是代价就是解析会更加困难，传输的数据体积会更大，效率偏低。

在有些情况下，可以对HTTP传输的数据进行加密。 TLS/SSL就是一种加密工具，工作在HTTP和TCP之间。我们把基于TLS协议的HTTP称作HTTPS协议，它提供了更好的安全性。

11.3 HTML、CSS和Javascript

我们所浏览的网页的本质是由HTML编写的超文本文档。 **HTML** (超文本标记语言—— HyperText Markup Language)构成了 Web 世界的一砖一瓦。它定义了网页内容的含义和结构。 **CSS** 则用来描述一个网页的表现与展示效果。而一些在网页端的交互行为则由 **JavaScript** 语言编写的网页脚本进行控制。超文本(**Hypertext**) 是指文档当中除了有普通文本内容以外，还存在着一些指向其他内容的链接。只要将内容上传到互联网，就可以通过链接和多媒体资源、 CSS文件、脚本文件以及他人创建的页面相连接。

HTML 使用标记(**markup**) 来注明文本、图片和其他内容，以便于在 Web 浏览器中显示。 HTML 标记包含一些特殊元素，比如 `<head>` `<title>` `<body>` `<header>` `<footer>` `<article>` `<section>` `<p>` `<div>` `<span>` `<img>` `<video>`等等。这些标记用来指定网页当中不同的元素。这些 HTML 元素可以通过**标签** (**tag**) 将文本从文档中引出，标签由在 < 和 > 中包含的元素名组成，HTML 标签里的元素名不区分大小写。也就是说，它们可以用大写，小写或混合形式书写。例如，`<title>` 标签可以写成 `<Title>` ， `<TITLE>` 或以任何其他方式。

下面我们以王道论坛为例子。在浏览器中可以打开论坛首页，然后右键点击网页空白处可以查看页面源代码。可以发现整个HTML采用一个树形的结构：

```
<!DOCTYPE html>
<html>
<head>
...
</head>
<body>
...
</body>
</html>
```

在文档当中，各个元素可以使用 href属性来指向所对应资源的位置。 `<style>`元素里面的内容就是所谓的CSS，它专门用来控制各个部分层叠样式(比如文字大小、颜色等等)。 `<script>`元素里面的内容就是 **JavaScript** 语言写好的网页脚本了，有些脚本代码是直接写到HTML文档当中，有些是通过 src属性来指定脚本文件的链接。

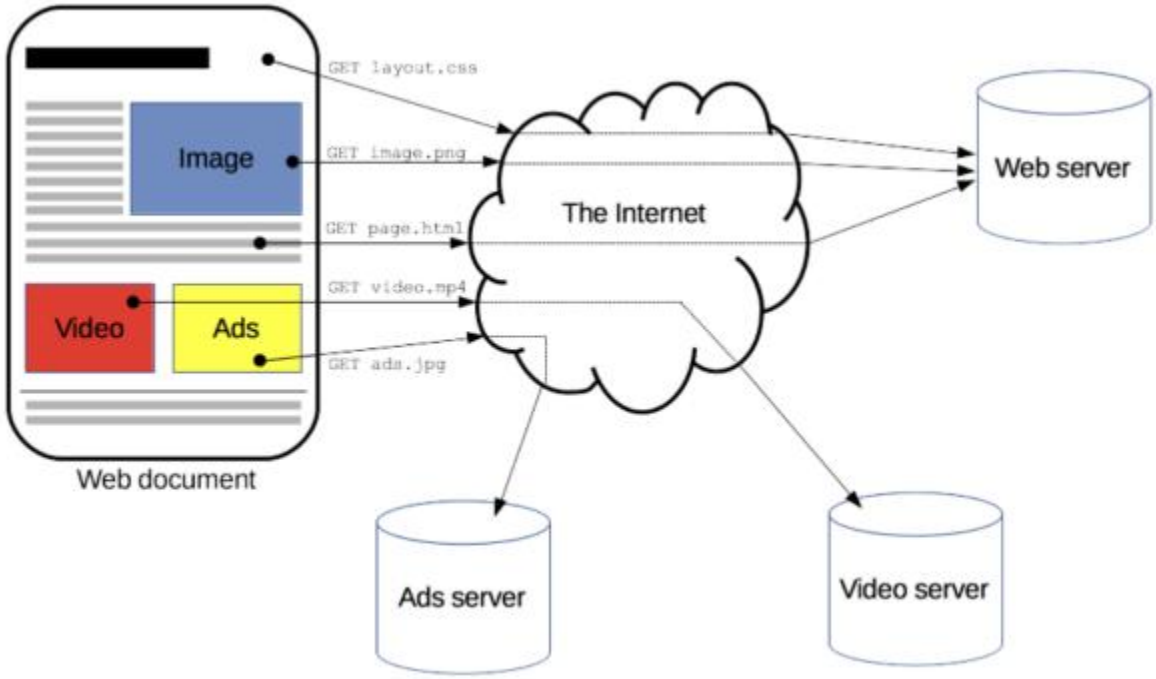
随便点开一个脚本文件内容看一下：

```
function lsSubmit(op) {
    var op = !op ? 0 : op;
    if(op) {
        $('lsform').cookietime.value = 2592000;
    }
    if($('ls_username').value == '' || $('ls_password').value == '') {
        showWindow('login', 'member.php?mod=logging&action=login' + (op ?
            '&cookietime=1' : ''));
    } else {
        ajaxpost('lsform', 'return_ls', 'return_ls');
    }
    return false;
}

function errorhandle_ls(str, param) {
    if(!param['type']) {
        showError(str);
    }
}
```

这个就是网页脚本的例子，可以发现到，除了语法上有一部分差异(主要是没有类型的声明)，其形式和C代码还是比较类似的，自然运行的逻辑也差不多。

至此，用户使用浏览器访问网页的流程就可以整理出来了：

- 用户输入一个网址，就可以找到Web服务器并向其发送一个HTTP协议请求
- 当收到请求以后， Web服务端也用HTTP协议回复一个响应，这个响应报文中包含了一个HTML文档。
- 浏览器收到响应的HTML文档后就可以解析文档，并且将其中的所有元素按照CSS的规定展示出来，如果其中存在一些超链接，比如图片、视频之类的，则会继续发送HTTP请求来获取这些资源的内容。
- 用户可以和网页的各个元素进行交互。对某些元素效果就是直接给服务端发送HTTP请求(比如提交表单)，而对另一些元素的操作就比较复杂，它会调用对应的网页脚本，网页脚本可以给用户返回结果，也可以发送HTTP请求，等得到响应之后再做后续处理返回给用户。



11.4 HTTP的各个组件

HTTP是一个客户端-服务端协议：请求通过一个客户端实体(称作Agent，用户代理)被发出。大多数情况下， Agent都是指浏览器，当然它也可能是其他的东西，比如一个命令行程序(如curl)，或者是图形界面程序(如Postman)，又或者是任何其他可以发送HTTP请求的应用程序，比如用户自行编写的程序。

每一个发送到服务器的请求，都会被服务器处理并返回一个消息，也就是响应。在这个请求与响应之间，还有许许多多的被称为代理的实体，他们的作用与表现各不相同，有些代理的目的是转发请求和转换协议，这称作网关(gateway)，另一些代理负责将一些结果保存起来以提升访问效率，即缓存(cache)。

注意：路由器和交换机之类的设备工作在网络层及以下，在Web应用的设计中这些设备是透明的，用户无需考虑和关心。

11.4.1 客户端

客户端指任何用来让用户发起请求的工具。这个角色通常都是由浏览器来扮演，程序员还会使用curl或者postman等工具当做客户端。

浏览器

浏览器总是作为发起一个请求的实体，它永远不是服务器。当用户使用浏览器查看网页时，浏览器首先发送一个请求来获取页面的HTML文档，再解析文档中的资源信息发送其他请求，获取可执行脚本或 CSS 样式来进行页面布局渲染，以及一些其它页面资源(如图片和视频等)。然后，浏览器将这些资源整合到一起，展现出一个完整的文档，也就是网页。浏览器执行的脚本可以在之后的阶段获取更多资源，并相应地更新网页。

一个网页就是一个超文本文档。也就是说，有一部分显示的文本可能是链接，启动它(通常是鼠标的点击)就可以获取一个新的网页，使得用户可以控制客户端进行网上冲浪。浏览器来负责发送HTTP请求，并进一步解析HTTP返回的消息，以向用户提供明确的响应。

11.4.2 服务端

在通信过程的另一端，是由Web服务端来接收用户请求并通过响应向客户端提供所请求的文档。服务端只是逻辑意义上代表一个机器：它可以是单个服务器，也可以是由一组服务器组成的计算机集群，还可以是一个用来实现复杂业务的分布式系统。 一台机器上可以部署多个服务端， 一个进程内也是可以部署多个服务端。采用一定的手段，可以让同一个机器或者同一个进程的不同服务端共享同一个端口号。

11.4.3 代理

在浏览器和服务器之间，有一些工作在应用上面的中间设备程序负责转发HTTP消息——称作代理 （**Proxies**）。代理可以以透明地方式工作，这样用户在使用过程中根本就感知不到，比如网站内部的反向代理。代理也可以不透明地工作，用户明确地知道自己的请求会经过代理进行转换。代理主要有如 下几种作用：

- 缓存：缓存上下行数据，减轻后端服务器的压力；
- 过滤：比如反病毒扫描、家长控制；
- 负载均衡：让多个后端服务器服务不同的请求；
- 认证：对不同资源进行权限管理；
- 日志记录：记录客户端的连接情况。

值得注意的是，代理和交换机、路由器是有显著区别的：代理工作在应用层，它会显式处理HTTP请求和 响应，而交换机、路由器工作在更低的层次， HTTP把这些低层设备看成是透明的。

11.5 URI和URL

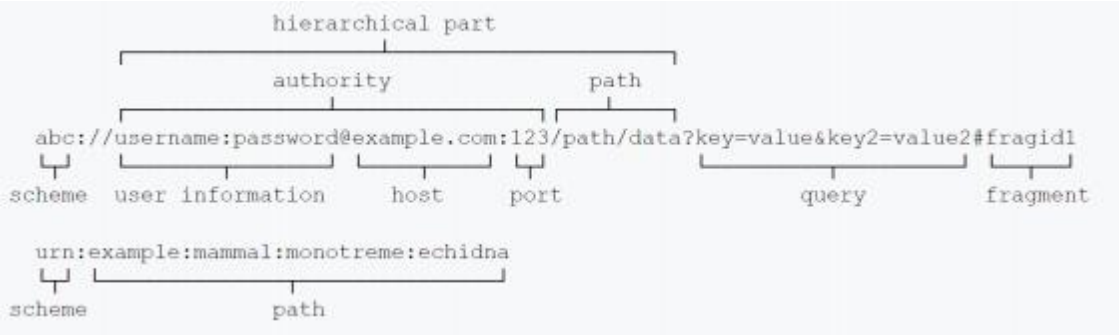
客户端如果想要给目的服务器发送请求(不局限于HTTP协议)，客户端需要做这样一些事情：

- 确定客户端所使用的协议
- 定位服务端在网络中的位置
- 定位服务端当中的某个进程
- 找到服务端进程当中的具体某个服务
- 给服务端提供本次请求的参数

所有的上述行为都可以通过一个名为URL (在本课程中，我们可以认为URL和URI是相同的东西)字符串来描述。 URL 代表着是统一资源定位符(**Uniform Resource Locator**) 。HTTP 请求的内容通称为资源。资源这一概念非常宽泛，它可以是一份文档， 一张图片，或所有其他你能够想到的格式。 URL 就是一个给定资源在 Web 上的地址，每个有效的 URL 都指向一个唯一的资源，这个资源可以是一个 HTML 页面， 一个 CSS 文档或者一幅图像等等。由于通过 URL 呈现的资源和 URL 本身由Web 服务器处理， 因此 web 服务器的拥有者需要认真地维护资源以及与它关联的URL。

一个典型的URL采用这样的书写方式：

- 方案字段说明将要采用的协议，然后是连接符 :// (如果不写方案字段则表示采用默认的HTTP协 议)；
- 用户信息字段说明登录网站的用户名和密码，中间用 : 隔开，然后的连接符是 @ (如果不需要登录认证，这个部分可以省略)；
- 主机字段说明服务端的IP地址或者是域名；
- 端口字段说明服务端的端口号(如果不写端口字段说明端口是默认端口， http是80 ， https是443)，前面的连接符是：；
- 路径字段用来定位服务的具体内容，当然具体内容的解析方式由服务端决定，中间的分隔符是 /；查询字段用来给服务端传递请求信息，由若干的键值对组成，键与值之间用 = 分隔，键值对与键值 对之间用 & 分隔；
- 分段字段和服务端无关，客户端用其来定位网页的阅读起始位置，前面的连接符是 #；



下面给出两个典型的例子：

`http://www.baidu.com/s?wd=123`

`https://www.zhihu.com/search?type=wangdao&q=123`

11.6 HTTP报文

HTTP报文分为两种——请求报文和响应报文。

11.6.1 GET请求示例

我们可以书写一个**简单的服务端程序**，它监听1280端口，然后读取所有的客户端请求并显示在标准输出上，然后回复一个默认的HTTP响应。

```
int main() {
     int listenfd = socket(AF_INET, SOCK_STREAM, 0);
     if(listenfd < 0) {
          perror("socket");
          return;
     }

     int on = 1;
     int ret = setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR,
               &on,sizeof(on));
     if(ret < 0) {
          perror("setsockopt");
          return;
     }

     struct sockaddr_in serveraddr;
     bzero(&serveraddr, sizeof(serveraddr));
     serveraddr.sin_family = AF_INET;
     serveraddr.sin_port = htons(_port);
     serveraddr.sin_addr.s_addr = inet_addr(_ip.c_str());
     ret = bind(listenfd, (struct sockaddr*)&serveraddr,
                sizeof(serveraddr));
     if(ret < 0) {
          perror("bind");
          return;
     }

     ret = listen(listenfd, 10);
     if(ret < 0) {
          perror("listen");
          return;
     }
```

```
    while(1) {
        struct sockaddr_in peeraddr;
        socklen_t len = sizeof(peeraddr);

        int peerfd = accept(listenfd, (struct sockaddr_in*)&perraddr, &len);
        if(peerfd < 0) {
            perror("accept");
            close(listenfd);return -1;
        }
        //打印客户端的信息
        printf("client: %s:%d has connected.\n",
            inet_ntoa(peeraddr.sin_addr),
            ntohs(peeraddr.sin_port));

        char buff[1024] = {0};
        ret = recv(peerfd, buff, sizeof(buff), 0);
        printf("recv ret: %d\n%s\n", ret, buff);

        //进行响应,构造一个响应报文
        const char * startLine = "HTTP/1.1 200 OK\r\n";
        const char * headers =
            "Server: MyHttpServer\r\n"
            "Content-Type: text/html\r\n"
            "Content-Length: ";

        const char * emptyLine = "\r\n";
        const char * body = "<html><head>response</head>"
                "<body>this is http test</body></html>";
        int length = strlen(body);

        memset(buff, 0, sizeof(buff));
        sprintf(buff, "%s%s%d\r\n%s%s",
            startLine,
            headers,
            length,
            emptyLine,
            body);

        //发送响应报文
        send(peerfd, buff, strlen(buff), 0);
        close(peerfd);
    }
    close(listenfd);
}//end main
```

下面我们分别使用**浏览器**、**curl**和**postman**工具发送一下HTTP请求：

首先是使用浏览器，在地址栏当中输入 http://192.168.30.129:8080(这个IP和端口需要根据服务端的实际bind情况来书写)，后面可以书写任意的路径和请求字段，然后可以看到标准输出上面的结果：

```
GET / HTTP/1.1
Host: 192.168.30.129:8080
Connection: keep-alive
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML,
like Gecko) Chrome/116.0.0.0 Safari/537.36
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,imag
e/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
Accept-Encoding: gzip, deflate
Accept-Language: zh-CN,zh;q=0.9,en;q=0.8,zh-TW;q=0.7
```

可以发现请求报文由3个部分组成：

- 第一行用来说明请求的方法、 URL的路径部分以及HTTP协议的版本，随后是一个换行\r\n（使用wireshark抓包可以发现，注意HTTP协议头部的所有换行都是\r\n ）；
- 随后若干行是若干个键值对，这些键值对称为首部字段。每个键值对的键和值都是字符串（所以HTTP是文本协议），中间用冒号：隔开，键值对与键值对之间用换行 \r\n 隔开；
- 首部字段结束之后是一个空的换行 \r\n ；
- 最后是请求的请求体部分，目前的请求体是空的。

接下来使用curl命令来发送HTTP请求。

```
curl 192.168.30.129:8080
```

下面是服务端看到的请求内容：

```
GET / HTTP/1.1
Host: 192.168.30.129:8080
User-Agent: curl/7.58.0
Accept: */*
```

还可以使用postman工具来发送请求，首先创建一个新的Collection，然后在Collection当中添加一个新的request，选择一个合适方法(比如GET)，再输入URL最后send即可。下面是服务端看到的请求内容：

```
GET / HTTP/1.1
User-Agent: PostmanRuntime/7.32.3
Accept: */*
Cache-Control: no-cache
Postman-Token: 3b1b8335-ce53-4092-b552-8fbacf625632
Host: 192.168.30.128:8080
Accept-Encoding: gzip, deflate, br
Connection: keep-alive
```

11.6.2 POST请求示例

前述的例子都是发送方法为GET的请求，并且的所有请求都没有携带请求体，接下我们就分别使用浏览器、curl和postman来发送一些其他类型的请求。

先使用curl来发送一个post请求：

```
curl -d 'key1=value1' -d 'key2=value2' -X POST 192.168.30.129:8080
```

下面是显示的内容：

```
POST / HTTP/1.1
Host: 192.168.30.129:8080
User-Agent: curl/7.58.0
Accept: */*
Content-Length: 23
Content-Type: application/x-www-form-urlencoded

key1=value1&key2=value2
```

和之前的GET请求相比，这个POST请求的报文多了一个请求体的部分。
当然，也可以使用postman来发送http请求，这种发送的方法使用起来更加简洁。只需要把方法改成POST，然后在Body当中选择x-www-form-urlencoded 的方式增加两个键值对即可。

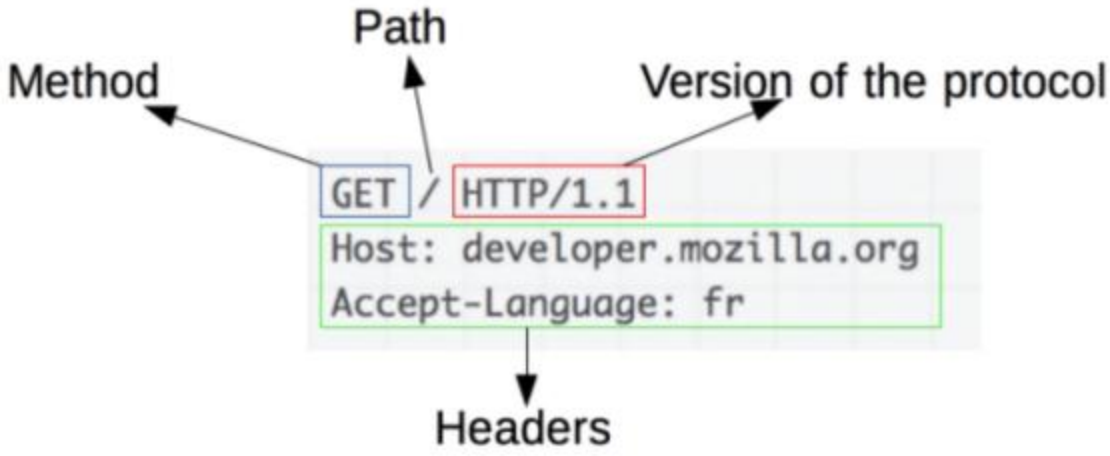
下面是显示的内容：

```
POST / HTTP/1.1
User-Agent: PostmanRuntime/7.32.3
Accept: */*
Cache-Control: no-cache
Postman-Token: 916f0960-1704-4f1f-a42f-40a093c3eb8f
Host: 192.168.30.128:8080
Accept-Encoding: gzip, deflate, br
Connection: keep-alive
Content-Type: application/x-www-form-urlencoded
Content-Length: 23

key1=value1&key2=value2
```

在浏览器中输入URL并确认是发起GET请求，倘若需要使用浏览器来发送POST请求的话，就需要先通过服务器传输一个HTML网页 `post_form.html`给浏览器，浏览器接收到数据，再打开该网页，其中存在着form表单元素，然后通过这个表单发送POST请求。

```
<!DOCTYPE html>
<html>
    <head>
        <title> title </title>
    </head>
    <body>
        <h1> POST请求 </h1>
        <p> Body类型: application/x-www-form-urlencoded </p>
        <form method="post" enctype="application/x-www-form-urlencoded">
            <input type="label">input:
            <input type="text" name="input"><br>
        </form>
    </body>
</html>
```

11.6.3请求报文详解



方法

在HTTP请求当中，第一行的第一个字段就是请求的方法。

- GET：用来获取资源；
- HEAD：用来获取资源的首部字段(不获取响应体)；
- POST：用来提交表单给对应的资源，这个请求通常来说会修改资源的状态(称为有副作用的)；
- PUT：用请求的内容替换掉目标资源的内容；
- DELETE：删除目标资源；
- TRACE：用来做环回测试；
- OPTIONS：描述目标资源的通信选项。

一般情况下， GET、 POST请求是最常使用的。

> curl命令的用法可以参考curl 的用法指南 - 阮一峰的网络日志 (ruanyifeng.com)

路径

HTTP请求的第一行的第二个字段是请求的路径。它对应了URL当中的路径字段。路径字段和对应的行为之间的具体分配策略由服务端程序员自行设计。服务端要完成的一个重要工作就是根据请求路径的内容，为不同的路径分配不同的服务以完成任务。

一种典型的设计就是将路径和服务端文件系统的路径对应起来，不同的路径就意味着访问不同的文件。

另一种设计方案会更加通用，例如，用 /api/add表示访问服务端代码的 add 函数，用 /api/sub表示访问服务端代码的 sub 函数。

版本

HTTP请求的第一行的第三个字段是HTTP协议的版本。最初的HTTP是0.9版本，随后出现了1.0，1.1， 2.0和3.0等版本。目前主流的HTTP协议版本是1.1 ， 1.1之后的2.0、3.0版本正在蓬勃发展中。

需要特别的注意是， HTTP/1.1版本是支持持久化连接的，在完成一个事务之后，服务端和客户端之间的 TCP连接可以不马上断开，这样如果后续HTTP请求是由同一个客户端(这里指拥有相同的IP端口)发起的话，就可以复用之前建好的TCP连接，不用重复3次握手的过程了。而且HTTP/1.1版本增加管线化技术，允许在第一个应答被完全发送之前就发送第二个请求，以降低通信延迟。

HTTP 的发展 - HTTP | MDN (mozilla.org)

首部字段

从请求的第一行结束开始， 一直到第一个空的换行为止(\r\n\r\n)都是属于首部字段。首部字段是由若干个键值对组成，有一些首部字段的含义是相对固定的，其含义是由RFC规范规定，另外一些首部字段 是由业务方自己设计的，通信的客户端和服务端自行根据首部字段的内容进行处理。

下面是一些常见的首部字段：

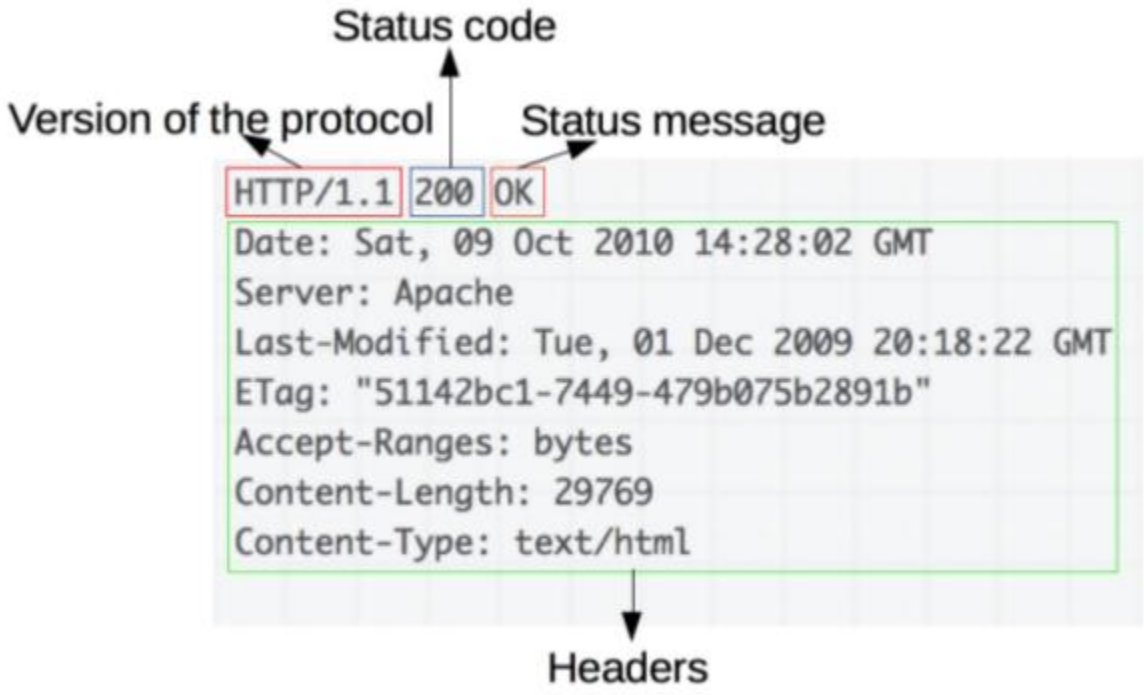
- Host请求头指明了请求将要发送到的服务器主机名和端口号。如果没有包含端口号，会自动使用被请求服务的默认端口（比如HTTPS URL使用443端口， HTTP URL使用80端口）。引入Host字段以后，可以让多个域名对应同一个IP地址。
- Accept请求头用来告知(服务器)客户端可以处理的内容类型，这种类型使用MIME的方式进行描述。服务端可以和客户端进行内容协商，然后回复一个合适类型的数据，以便客户端可以合理地表现资源。 (MIME是一种描述资源类型的方式， MIME)。
- Connection请求头决定当前的事务(一次请求连同对应的响应称为事务)完成后，是否会关闭网络连接。如果该值是“keep-alive”，网络连接就是持久的，不会关闭，使得对同一个服务器的请求可以继续在该连接上完成。
- Content-Length请求头用来指明发送给接收方的请求体的大小，以字节为单位。
- User-Agent首部包含了一个特征字符串，用来让网络协议的对端来识别发起请求的用户代理软件的应用类型、操作系统、软件开发商以及版本号。

请求体

目前我们只考虑方法为POST的请求体。最为常见的请求体的形式有着两种： `application/x-www-form-urlencoded`和 `multipart/form-data` ：前者通常是普通的表单数据，而后者一般是通过表单来提交的二进制数据，比如通过表单上传的文件等等。下面是请求内容的示例：

```
//这是使用x-www-form-urlencoded形式的请求，请求体采用=和&将多个键值对进行连接 Content-Type: application/x-www-form-urlencoded
Content-Length: 23

key1=value1&key2=value2

//这是使用form-data形式的请求
//首先在首部字段定义一个boundary，这个boundary会作为各个不同的键值对之间的分隔符，最后一个boundary后面会添加--- 作为整个请求体的结尾
Content-Type: multipart/form-data; boundary=--------------------------------------------
-730798861229637558155649
Content-Length: 274

----------------------------730798861229637558155649
Content-Disposition: form-data; name="key1"

value1
----------------------------730798861229637558155649
Content-Disposition: form-data; name="key2"

value2
----------------------------730798861229637558155649--
```

11.6.4 响应报文



类似于请求报文，响应报文也由3个部分组成：

- 第一行用来指明HTTP协议的版本和响应状态；
- 和请求一样，接下来是若干行首部字段，每个首部字段用 \r\n换行；
- 首部字段部分以一个空行结束(即 \r\n\r\n)；
- 最后是响应体，存储报文真正承载的数据。

下面是一个HTTP响应的示例： (使用 `curl -i` 选项可以获取完整的响应报文)

响应的状态

HTTP响应的状态可以用来通知客户端请求执行的状态， 一般使用一个3位数字的状态码来描述，随后的字符串用来描述详细内容。不同大种类的状态依靠第一位数字来区分，某个大类型下具体的详细状态则由后续两位数字进行区分。下面是状态码大类的分类：

- 100 ~ 103：信息提示 ；
- 200 ~ 206：成功，其中200表示OK；
- 300 ~ 305：重定向，其中301表示URL已经被移除了，资源位于另一URL；
- 400 ~ 415：客户端错误；
- 500 ~ 505：服务端错误。

下面是常见的状态码的具体用法；

- 200 OK：表明请求已经成功。
- 301 Moved Permanently：说明请求的资源已经被移动到了由 Location 头部指定的URL上，是固定的不会再改变。服务端在使用301状态时应该在响应报文中加上Location首部字段，这样浏览器会自动重定向到URL所指的资源。

```
//下面稍微调整下原来的server代码
const char * firstLine = "HTTP/1.1 301 Moved Permanently\r\n";

const char * headers ="ContentType:text/html\r\n"
                      "Location:http://www.baidu.com\r\n";
```

- 302 Found ： 302重定向状态码表明请求的资源被暂时的移动到了由该HTTP响应的响应头 Location指定的 URL 上。浏览器会重定向到这个URL。使用时和301类似，也需要加上一个 Location首部字段。
- 304 Not Modified ： 304说明无需再次传输请求的内容，也就是说可以使用缓存的内容。
- 404 Not Found ： 404代表客户端错误，指的是服务器端无法找到所请求的资源。
- 403 Forbidden ： 403代表客户端错误，指的是服务器端有能力处理该请求，但是拒绝授权访问。
- 500 Internal Server Error：在 HTTP 协议中， 500 Internal Server Error是表示服务器端错误的响应状态码，意味着所请求的服务器遇到意外的情况并阻止其执行请求。这个错误代码是一个通用的 “万能”响应代码。

11.7 RESTful设计风格

客户端可以通过多种手段去给服务端发送数据(方法、 URL的路径部分、首部字段、请求体)，这多种多样的方式会对编程风格的统一性产生巨大的影响。为了规范接口的设计和解析方式，著名的HTTP专家Roy Fielding提出一种接口设计的风格，称为REST (表示状态转换Representational state transfer)。

如果用户的接口设计符合REST风格，就称作是RESTful的接口设计。 REST风格的特点如下：

- 使用HTTP方法表示行为。 (GET PUT POST DELETE就分别表述 查改增删)
- 所有的对象(网络实体或者具体信息)都抽象成资源，每一个资源都使用URL的路径来标识；
- 所有传递的参数都放在请求和响应的报文体当中
  - 使用JSON或者是XML编码数据
  - 客户端不再得到完整的HTML，数据的展示效果由客户端完成

REST风格的设计目的，是为了提高性能、可扩展性、可靠性。具体到每一个方法，都有各自的特点：

- GET方法是安全无副作用的，不会修改任何状态；
- GET、 PUT和DELETE是幂等的，所谓的**幂等**是指执行一次和执行多次的效果是一致的；
- GET和POST是可缓存的，它们的响应是可以缓存起来，以便下次请求会快速得到响应。

11.8 HTTPS的原理

HTTP本身的内容是按照明文进行传递，无论是URL、报文体还是报文头部，都有可能被恶意截取，造成安全隐患，严重影响用户体验，甚至可能会给用户造成财产损失。 一种解决方案就是将HTTP报文的内容 进行加密——这就是SSL (Secure Sockets Layer ，安全套接层)的工作内容， SSL工作在HTTP协议和 TCP协议之间，在ISO/OSI 7层模型对应第6层表示层，在TCP/IP协议族当中属于应用层，但是在HTTP协 议的下层。由于商业竞争原因， SSL协议现在已经被另一种加密方式TLS (Transport Layer Security)取 代。虽然SSL和TLS在实现上有相当的差异，但是其在网络协议的位置相同，提供功能也是一致的，所以 我们一般称为SSL/TLS层。

11.8.1 对称加密和非对称加密

对称加密：客户端使用密钥和原始文档可以生成加密文档，服务端收到加密文档后，利用相同的密钥可以还原为原始文档。当加密算法或者密钥当中任意一个都没有泄漏时，其他人是无法截获原始文档内容的。 一种简单的对称加密算法就是使用异或操作。通常而言，对称加密的效率会更高一些。

非对称加密：客户端和服务端之间会存在一对互补的密钥，称作公钥和私钥。客户端使用公钥加密原始文档得到加密文档，而服务端会使用相对的私钥解密加密文档得到原始文档。除了可以传输加密文档以外，由于成对的公钥和私钥之间是互补的，所以还可以提供身份验证的功能。客户端要想安全的访问服务端，客户端首先先生成一对公钥和私钥，然后将公钥拷贝给服务端。当某个客户端登录，服务端会检查对应客户端的私钥是否和已经保存的某个公钥互补，倘若互补，就可以验证本次登录的客户端的身份了。

11.8.2 TLS的实现机制

TLS的实现要完成四个目的：验证身份、达成安全套件(使用的算法类型+密钥)共识、传递密钥、加密 通讯。

下面我们来看看TLS的执行步骤：

- 在使用HTTPS协议访问服务端时，首先需要使用TLS进行加密握手。

1. Client Hello：客户端将浏览器的信息(支持的加密算法，通常使用的RSA，一种非对称加密算法) 发送给服务端；
2. Server Hello：服务端回复客户端，内容包含所使用的加密算法和一个随机数密钥，如果这个客户端在一段时间内曾经连接过，可以复用密钥；
3. Server Certificate：服务端回复客户端自己的证书信息，从而浏览器可以验证证书的有效性；
4. Server key exchange & Server finished：根据选择的加密算法，选择回复的内容：可以是一个随机数；也可以生成一对公钥私钥，并回复公钥；有些算法可以不回复内容；
5. Client key exchange：根据算法的不同客户端有不同的选择：可能是将服务端发来的随机数连同证书部门的信息加密得到新字符串；也可能是通过算法(比如RSA)生成一对密钥，将公钥发送给服务端；还有可能什么都不发送；
6. Key generation：双方根据随机数和生成的新字符串或者是根据彼此公钥和私钥，生成一个新的相同的密钥(这个密钥是对称加密算法AES/DES的密钥)；
7. Client CipherSpec Exchange & Finish：通知服务端要切换密码了，客户端握手完成；
8. Server CipherSpec Exchange & Finish ：回复客户端已经切换密码了，服务端握手完成。

- 在客户端和服务端完成TLS加密握手之后，就得到了一个对称加密算法的密钥
- 之后，所有的通信内容都通过对称加密算法进行加密解密，只要公钥证书是可靠的，那么通信内容就不会被泄漏了。

使用 `curl -v` 访问一个https站点就可以看清TLS加密过程。(也可以使用wireshark抓包，并输入表达式 ssl进行筛选)

```
curl -v https://www.baidu.com

* Rebuilt URL to: https://www.baidu.com/
*   Trying 14.119.104.254...
* TCP_NODELAY set
* Connected to www.baidu.com (14.119.104.254) port 443 (#0)
* ALPN, offering h2
* ALPN, offering http/1.1
* successfully set certificate verify locations:
*   CAfile: /etc/ssl/certs/ca-certificates.crt
  CApath: /etc/ssl/certs
* TLSv1.3 (OUT), TLS handshake, Client hello (1):
* TLSv1.3 (IN), TLS handshake, Server hello (2):
* TLSv1.2 (IN), TLS handshake, Certificate (11):
* TLSv1.2 (IN), TLS handshake, Server key exchange (12):
* TLSv1.2 (IN), TLS handshake, Server finished (14):
* TLSv1.2 (OUT), TLS handshake, Client key exchange (16):
* TLSv1.2 (OUT), TLS change cipher, Client hello (1):
* TLSv1.2 (OUT), TLS handshake, Finished (20):
* TLSv1.2 (IN), TLS handshake, Finished (20):
* SSL connection using TLSv1.2 / ECDHE-RSA-AES128-GCM-SHA256
* ALPN, server accepted to use http/1.1
* Server certificate:
*  subject: C=CN; ST=beijing; L=beijing; O=Beijing Baidu Netcom Science
Technology Co., Ltd; CN=baidu.com
*  start date: Jul  6 01:51:06 2023 GMT
*  expire date: Aug  6 01:51:05 2024 GMT
*  subjectAltName: host "www.baidu.com" matched cert's "*.baidu.com"
```

```
*  issuer: C=BE; O=GlobalSign nv-sa; CN=GlobalSign RSA OV SSL CA 2018
*  SSL certificate verify ok.
```