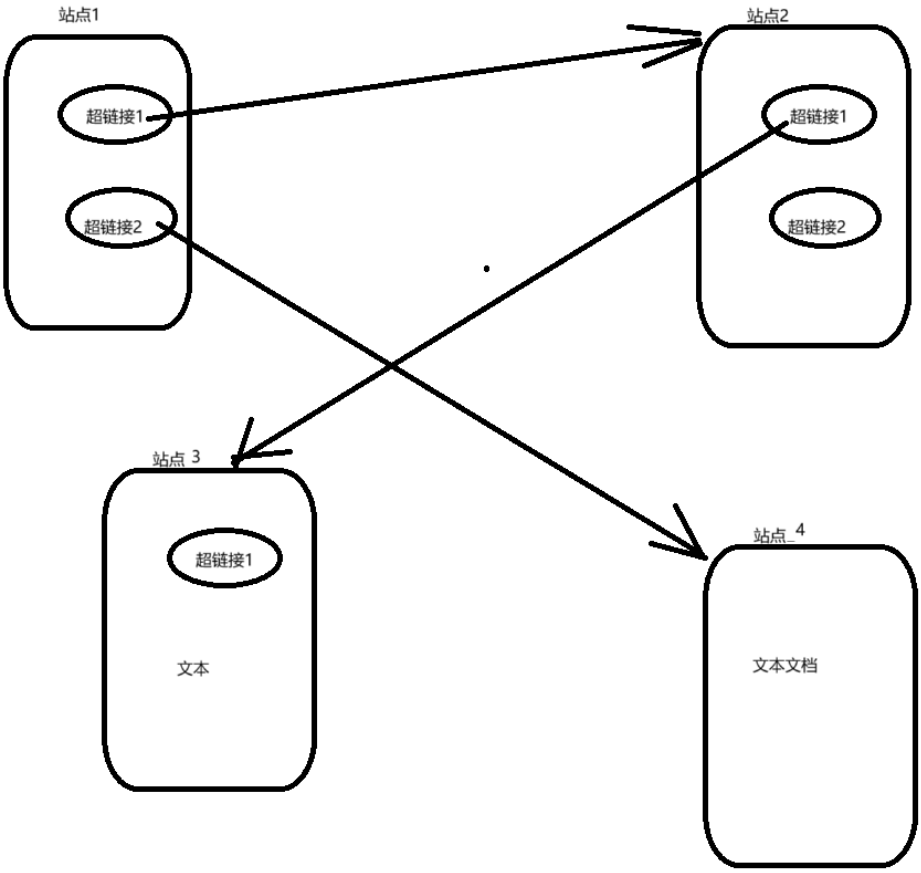
第六章：Http服务器

一、基本相关概念

1.1 万维网www

1> 万维网（world wide web）:是一个大规模的、**联机式**的信息储藏所，简称web。能够实现使用**链接的方式**方便的从一个站点访问到另一个站点，从而主动的按需获取丰富的信息。



2> 超文本：**包含了指向其他文档的链接的文本文档成为超文本**，能够被浏览器所识别。一个超文本是由多个信息源组成，而这些信息源可以分布在世界各地，并且数量不受限制。超文本是万维网的基础。

3> 万维网使用的是BS模型，即浏览器服务器模型。浏览器向服务器发送请求，服务器响应浏览器并发送客户端所需要的超文本

4> URL(Uniform Resource Locator):统一资源定位符。用来标识超文本在万维网中的唯一标识，表明了该超文本的信息，具有唯一性。网络访问中，就是依照此来进行请求以及响应的。

URL由四部分组成：**协议** ： //**主机名** ： **端口号** / **路径**

例如：http://www.baidu.com.cn

协议：指出使用的何种协议来获取万维网的文档，常用的协议就是http（超文本传输协议），：//是规定格式，必须写上

主机名：是万维网文档所在的主机域名（ip），通常以www开头，但也不是硬性要求，也可以直接使用主机的ip地址来代替

端口号：就是该协议对应的端口号，对应合同谈判协议而言，默认的端口号是80

路径：较长的字符串，超文本文档所在的路径

1.2 超文本传输协议http

1> 概念：HTTP定义了浏览器是怎样从万维网服务器中请求超文本，以及服务器如何把文档传递给浏览器的。

2> 从层次角度来说：HTTP是面向**事务**的**应用层协议**，能够实现**可靠的**文件交换

事务：指的是一系列的信息交换（建立链接、发送请求、响应请求、展示信息并断开请求），而这一系列的信息脚后是一个不可分割的整体，要执行就全部执行结束，要不执行就全部不执行

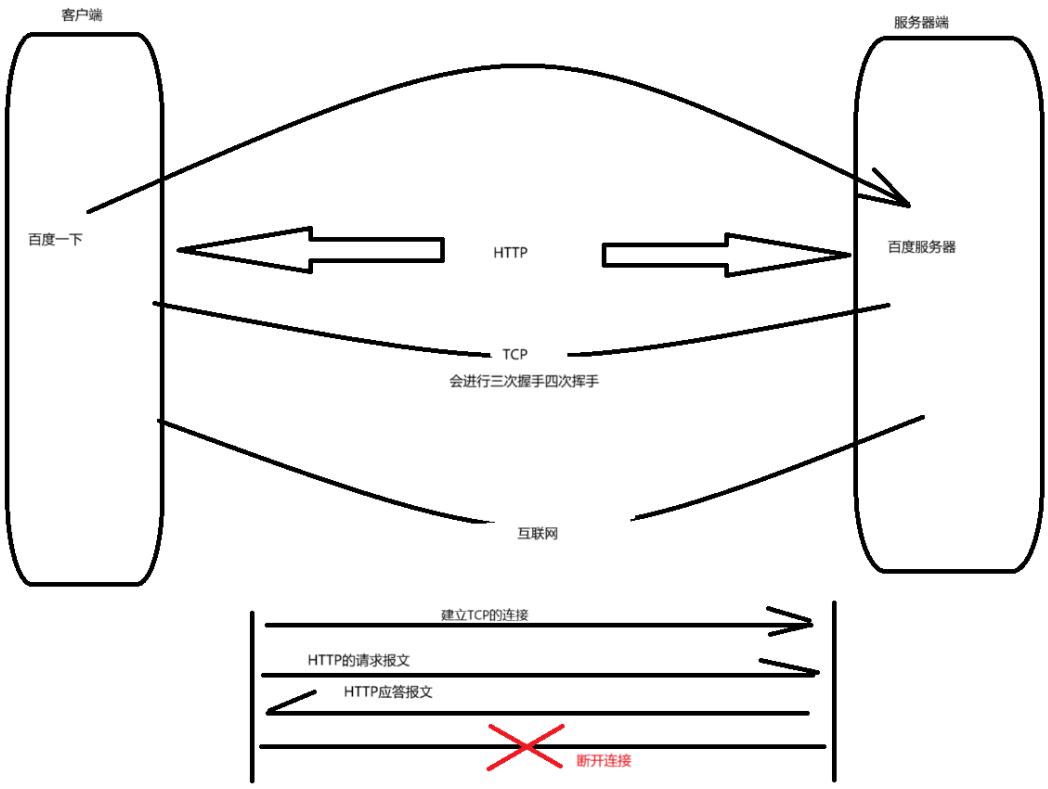
可靠：该协议是应用层协议，其传输层使用的是TCP协议进行的可靠传输

3> HTTP永远是**客户端发起请求，服务器端进行响应。**这样就限制了HTTP协议使用时，无法实现在客户端没有发起请求时，服务器主动向客户端发送消息。

4> HTTP协议时**无状态的协议**，同一个客户端的本次连接服务器和下一次的连接服务器没有任何联系。服务器没有记忆功能。

5> HTTP协议本身是**无连接的**，也就是说限制每个连接只能处理一个请求，服务器处理完客户端的请求并收到客户端的应答后就会立即断开连接。

6> 工作流程



1.3 超文本标记语言HTML

1.3.1 HTML的基本内容

1> HTML（HyperText Markup Language）:超文本标记语言，时用来完成超文本文档的编写的。html并不是协议，而是万维网浏览器能够识别的一种语言

2> 该语言中定义了需要使用的排版命令，这些命令我们称之为 “标签”

3> 使用html语言编写的文档称为超文本，后缀为 .html 或者 .htm

4> 浏览器主要解析的就是该语言编写出来的超文本，能将相关标签解析成界面

5> html编辑器可以是任意编辑器，如文本编辑、vscode、sublime等待

1.3.2 常用标签

1> 标签格式：使用双尖括号包裹着关键字，通常成对出现

2> 标签分类：

```
1、单标签：也称空标签  <标签名/>      例如：<br/>
2、双标签：成对出现 <标签名>  内容 </标签名>
```

3> 常用的标签

```
h1--h6:标题标签
p:段落标签
div:盒子标签
input:输入标签
。。。
```

www.runoob.com/html/html-tutorial.html 参考菜鸟教程

1.3.3 第一个HTML文件

1> vscode中创建一个后缀为.html的文档

2> 输入html:5 或者！ 后按tab键补齐

3> 在拓展部分，加一个live server的扩展

```
<!-- 第一行：文档类型签，html说明是超文本文档 -->
<!DOCTYPE html>
<!-- 第二行表示整个界面的开始标签，是一个双标签 -->
<html lang="en">
    <!-- 浏览器头部信息 -->
<head>
    <!-- 表示编码格式 -->
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <!-- 浏览器的标题 -->
    <title>My first HTML</title>
    <!-- 浏览器头部的结束标签 -->
</head>
<!-- body中才是整个界面的主体 -->

    <!-- h标签表示这是一个标题 -->
    <h1>这是我的第一个界面，使用的是一级标题</h1>
    <!-- a标签表示这是一个超连接 -->
    <a href="www.baidu.com">百度一下</a>
    <!-- 换行 -->
    <br>
    账号：<input type="text">
    <br>
    密码：<input type="password">
<body>
```

```
</body>
<!-- html标签的结束 -->
</html>
```

二、HTTP版本

2.1 http1.0版本

从上世纪90年代开始，浏览器诞生后的第一个标准，是一个**短连接**的版本。

该版本的很大的一个问题就是每发起一个请求，都要建立一次TCP的连接，而且是串行请求，做了很多无谓的TCP连接和断开，增加了开销，并且一般请求时使用的是明文模式，容易被攻击，安全性能比较低。

2.2 http1.1版本（正在使用的）

1> 在http1.0版本的基础上**增加了持续连接**的功能。服务器在发送响应后的一段时间内仍然保持这条连接，使同一个客户端和服务器可以在这条连接上继续后续的请求和响应工作。直到某一段主动断开连接或者长时间不响应时也会断开连接

2> http1.1版本支持两种工作模式

非流水线方式：客户端在收到前一个响应之后才能发出下一个请求

流水线方式：客户端在收到前一个响应之前就能直接发送下一个请求

3> 特点：简单（报文首部和报文内容）、灵活易于拓展（请求方法和状态码没有固定死）

4> 不足：依然是无状态、铭文传输不安全

5> http1.1版本所作的优化

```
1、使用长连接的方式，解决了1.0版本的短连接的性能问题：只要任意一端没有明确提出断开连接，那么则一直保持TCP的连接状态
2、使用管道网络传输：允许在同一个TCP连接中，建立多个管道，客户端可以发送多个请求，不需要等到客户端响应后才发送另一个请求。这样就提高了整体的响应时间。解决了请求的队头阻塞
        不足是：服务器端必须按照接收请求的书匈奴发送这些请求的响应，如果服务器在处理请求A花费了很长时间，那么后续的请求的处理就都会被阻塞。这成为响应队头阻塞
3、实际上HTTP1.1的管道化技术并不是默认开启的，而且很多浏览器技术都不支持。所以，大家只需要知道http1.1版本提供了该功能，但是并没有得到广泛应用
```

6> http1.1的瓶颈

```
1、头部巨大且重复：http头部会含有很多固定的字段，并且加起来会有几百字节甚至上千字节，未经压缩就直接发送。
                    大量的请求和响应报文中会有很多重复的字段；
                    http协议是无状态的，想要记录之前的操作，在首部外加了一个cookie的技术，也会使得首部比较巨大
2、http1.1版本传输请求和响应的过程使用的是ASCII编码完成，而不是二进制编码
3、并发连接数是有限的，以谷歌为例，最大的并发连接数是6个，而且每个连接都要经过TCP的三次握手，以及TCP的慢启动过程
4、响应对头阻塞问题没有解决
5、不支持服务器推送过程，只能客户端请求，服务器响应
```

2.3 http2.0版本

1> 2015年推出了http2.0版本，目的是为了改善http的性能，并且兼容了http1.1版本

2> http2.0版本只在应用层做了改变，传输层还是使用的TCP传输协议。HTTP2.0把http分成【语义】和【语法】两部分，其中语义层不做改动，跟http1.1保持一致。但是在[语法]部分做了很多改动，基本上改变了HTTP的报文传输格式

3> HTTP2.0完成了头部压缩，HTTP2.0使用**了HPACK的算法**，发送的是压缩后的内容，这样就可以节约带宽，解决了http1.1版本的头部巨大且重复问题。HPACK算法将传输过程中进行的同一个TCP传输的内容进行合并，将重复的内容不需要进行传输了。提高传输效率

4> http2.0能够实现并发传输：引入了stream的概念，可以实现并发发送stream，只需要建立一个TCP的连接，节约了多次建立连接过程中握手机制的事件

5> http2.0实现了数据推送功能：服务器端和客户端双方都会建立一个stream，并且使用stream ID进行区分。客户端建立的是奇数号，服务器建立的是偶数号。服务器推送数据时，会先发送PUSH_PROMISS帧，告诉客户端接下来哪个stream发送资源。

6> HTTP2.0的不足

```
1、也存在响应队头阻塞问题
2、TCP的握手延时问题
3、网络迁移时需要进行重新连接：服务器和客户端建立连接的四要素:源ip地址、源端口号、目的ip地址、目的端口号
总结：以上问题主要的原因是应用层使用的是TCP服务，无论http层再怎么设计都无法逃避。如果想要从根本上解决问题，那么就需要彻底改变传输层的相关协议，将传输层的TCP协议改成UDP协议，这就是http3.0的重要改革部分
```

2.4 http3.0版本

1> http3.0是最新推出的传输协议

2> http3.0将传输层的TCP协议更改成了UDP协议，使用的是基于UDP的QUIC的协议

UDP：面向无连接的，不关心是否丢包问题，所以，**解决了队头阻塞问题**

3> QUIC协议：基于UDP实现的买那些无连接的协议，也不许进行三次握手和四次挥手。再协议中实现了类似于TCP的连接管理、拥塞控制、流量控制等相关操作。相当于将UDP的不可靠传输改成可靠的传输协议了

4> QUIC吸引中也有一个stream的概念，保证数据的可靠性，每个sream数据报都有一个唯一的ID标识。当一个数据报一旦丢失后，该数据会被重传，不会影响其他数据包的传输。

5> 能实现更快的连接建立：QUIC协议也需要进行握手连接，相当雨将TCP的握手机制结合在一起，基于一个ID进行连接，而不是基于源ip地址、源端口号、目的ip地址、目的端口号

6> 不是基于四元组进行连接的，是通过一个stream ID完成，那么就可以实现在不同网络下的迁移

2.5 https版本

1> 为了传输的安全性，引入了HTTPS，加密的超文本传输协议

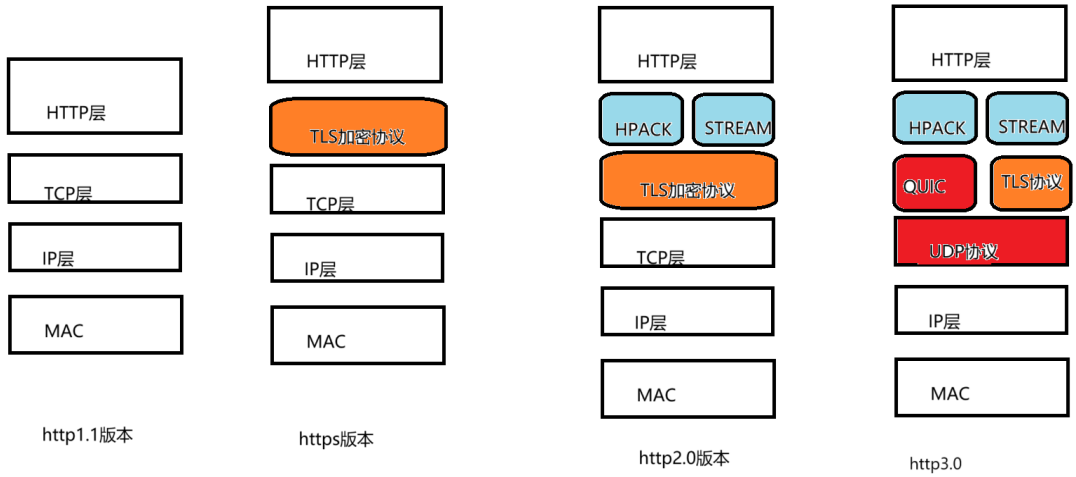
2> 在http层和TCP层之间加入了TLS协议，该协议提供了信息加密、校验机制以及身份证数组证书

3> TLS协议需要进行四次握手机制

1、客户端向服务器发送随机数C、TLS协议版本号、密钥套件系列列表
2、服务器收到客户端信息后回复一个ACK
3、服务器在发送一个随机数S、切人TLS版本号、使用的密钥套件
4、客户端回复一个ACK

4> 注意：TLS的四次握手，是发生在TCP的三次握手建立连接之后执行的安全性检查。是一个耗时操作，但是确保了信息的安全性。

2.6 一个模型记住各个版本的http



三、HTTP报文格式

1> http报文分为两类

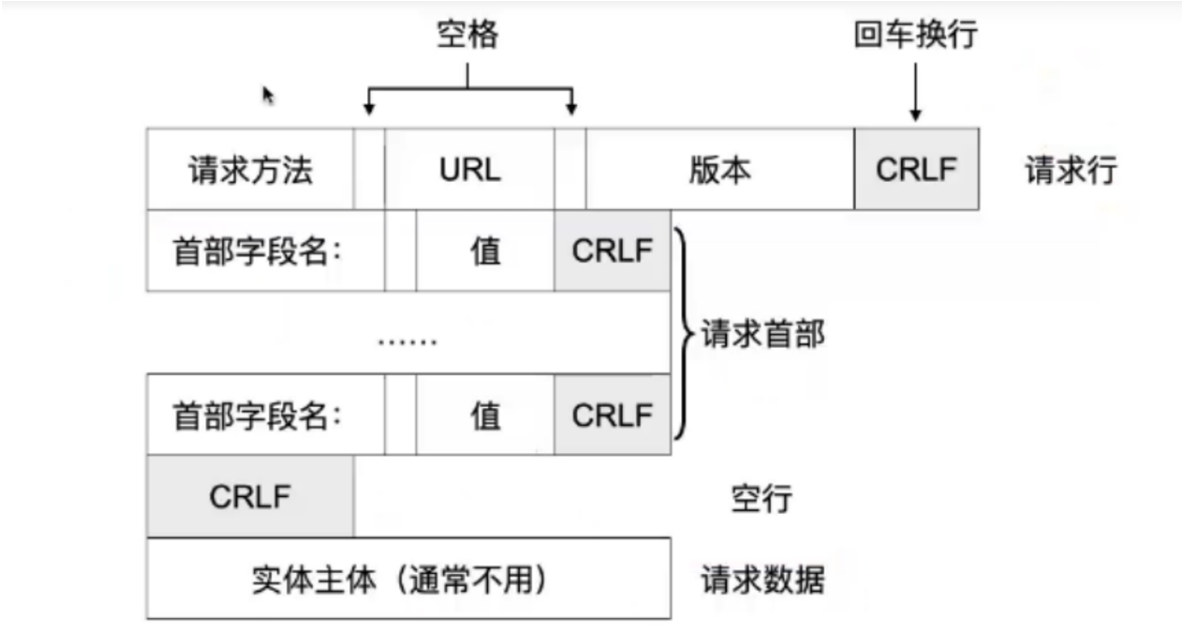
请求报文：从客户端向服务器发送的请求报文

响应报文：从服务器端向客户端发送的应答

2> 报文由三部分组成：开始行、首部行和实体主体

1、开始行：用于区分是请求报文还是响应报文
请求报文中，开始行也成为请求行
响应报文中，开始行也成为状态行
2、首部行：用来说明浏览器、服务器以及报文主体的一些信息
首部行可以由很多行，也可以没有
在每一个首部行都有一个首部的字段名和它的值，中间使用冒号隔开，每一行的首部行中间使用回车换行隔开
整个首部行结束后，还有一个空行将首部行和后面的实体主体进行分割
3、实体主体：在请求报文中，一般不用。有时响应报文也可以没有

3.1 客户端请求报文格式



1> 请求行由三部分组成，分别是请求方法、URL和版本，中间使用空格隔开

2> 请求方法：就是对所请求的对象进行的操作，因此这些方法实际上也就是一些操作指令。因此，请求报文的类型是由它所指定的方法来决定的

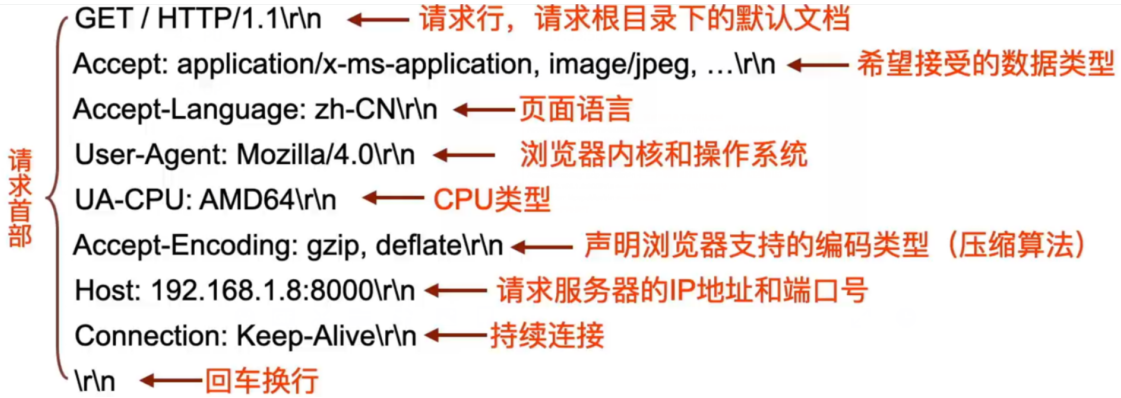
常用的请求方法：
1、GET：用来请求URL的资源，并返回实体主体
2、POST：向指定的资源提交数据并进行处理（例如提交的是表单或者上传文件）。数据被包含在实体主体部分。POST请求可能会导致新的资源的建立或者对已有资源的修改
3、HEAD：类似于GET请求，但是返回的是响应中的首部，没有内容主体
4、PUT：从客户端向服务器传送的是指定的文档内容
5、DELETE：请求服务器去删除某些页面
6、OPTION：请求一些选项的信息
7、TRACE：用来进行环回测试使用的报文请求
8、CONNECT：HTTP/1.1版本中预留给能够将连接改为管道方式的代理服务

3> URL:统一资源定位符，就是要申请的资源路径

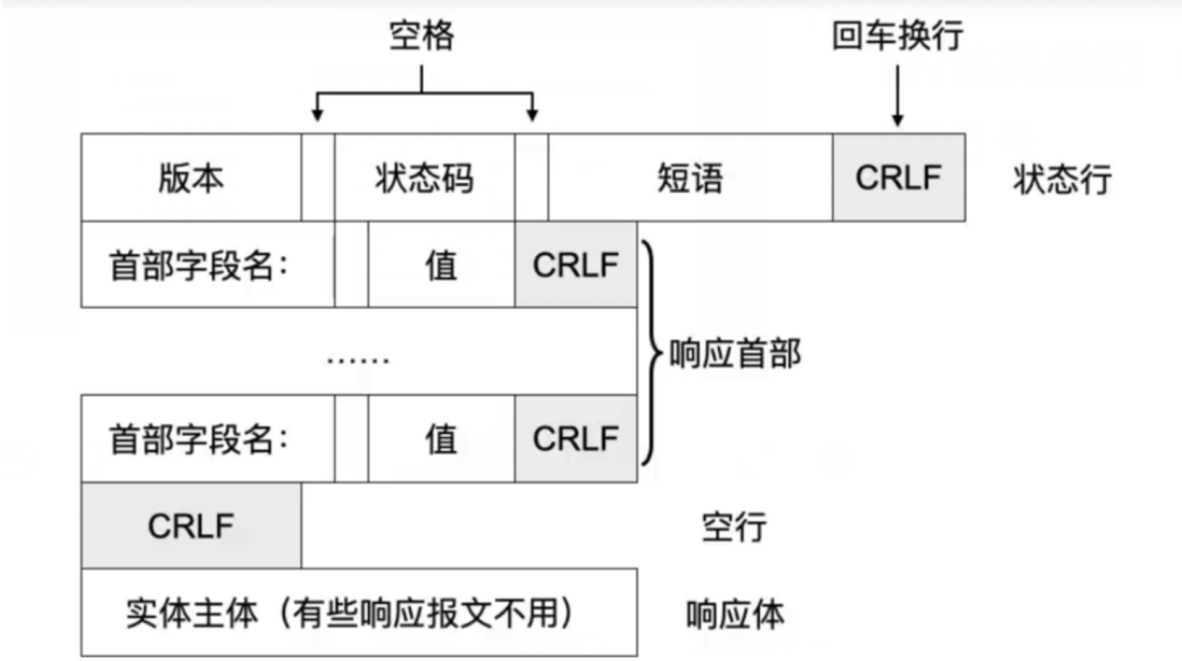
4> 版本：HTTP版本，目前常用的使用HTTP 1.1版本

5> CR和LF：回车和换行

6> 报文请求的实例



3.2 响应报文格式



1> 响应报文的首部也成为状态行：也由三部分组成，分别是版本、状态码和解释状态码的短语，中间也是有空格隔开

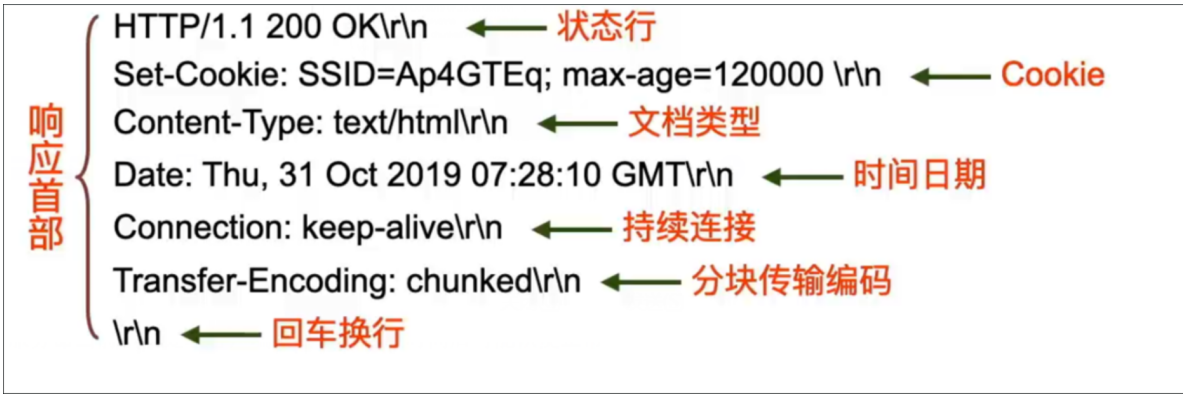
2> 版本：就是HTTP版本

3> 状态码：状态码是由三位数字组成，第一个数字定义了响应的类别，共分为5类

```
1XX：指示信息---表示请求已经接收，继续处理
2XX：成功---表示请求已经被成功接收、理解以及处理。如200表示成功
3XX：重定向---表示要完成请求必须要进行更进一步的操作。如304
4XX：客户端错误---请求的语法错误或者请求无法实现
        432：HTTP错误
        400：客户端请求有语法错误，不能被服务器所解析
        401：认证失败，请求未被接受
        403：服务器收到请求，但是拒绝提供服务
        404：请求的资源不存在，或者输入的URL有误
5XX：服务器错误---表示服务器未能实现合法的请求
        500：服务器发生了不可预测的错误
        503：服务器不能处理当前客户端的请求，一段时间后可能恢复正常
```

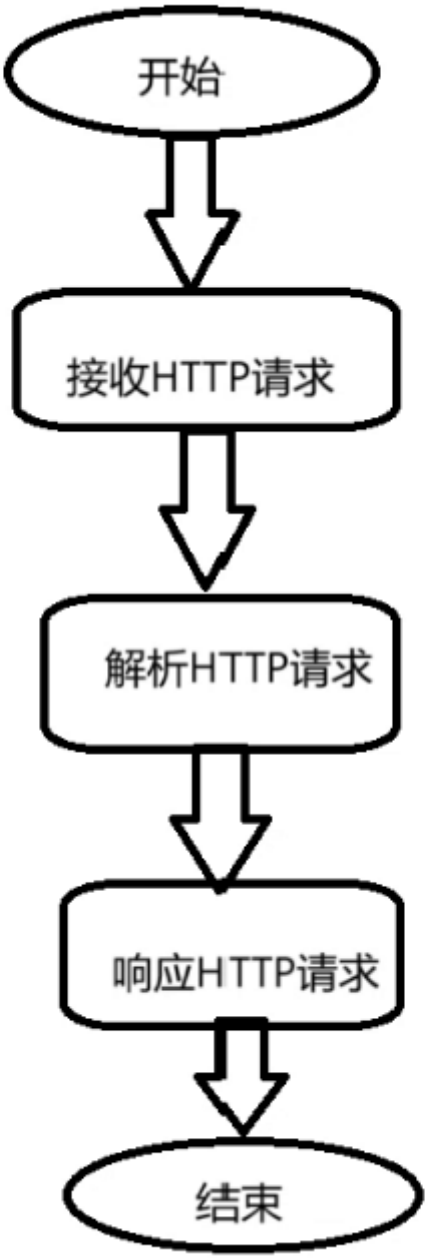
4> 实体主体：可以有可以无，如果服务器需向客户端发送数据，则将数据放入到该处

5> 响应报文实例：



四、HTTP服务器

4.1 原理



4.2 源码

1> main.cpp

```
#include <myhead.h>
#include"http.h"

//定义线程体函数
void *msg_request(void *arg)
{
    //解析到传进来的客户端的套接字文件描述符
    int sock = *(int *)arg;

    //调用信息处理函数，该函数是所有请求的入口
    handler_msg(sock);
}
```

```c
/******************主程序***************************** */
int main(int argc, const char *argv[])
{
    int port = 80;          //默认使用80端口号

    //判断是否手动传入端口号
    if(argc>1)
    {
        port = atoi(argv[1]);
        //atoi：功能：将给定的字符串转变成整数
        //参数：字符串（只能包含数字字符）
        //返回值：整数       例如：“123”  ---> 123
    }

    //初始化服务器
    //自定义函数：功能完成tcp服务器的初始化操作  创建套接字、绑定端口号、监听
    //参数：端口号
    //返回值：服务器套接字
    int lis_socket = init_server(port);

    //循环接收客户端的连接请求
    //并发服务器使用多线程完成
    while(1)
    {
        struct sockaddr_in peer;  //接收客户端的地址信息结构体
        socklen_t len = sizeof(peer);    //接收长度

        //接收客户端的连接请求
        //返回的套接字就是服务器端创建处理的用于跟客户端进行通信的套接字文件描述符
        int sock = accept(lis_socket, (struct sockaddr*)&peer, &len);
        if(sock==-1)
        {
            perror("accept error");
            return -1;
        }

        printf("您有新的客户端发来连接请求了\n");

        //创建一个分支线程用户跟客户端进行通信
        pthread_t tid = -1;
        if(pthread_create(&tid, NULL, msg_request, &sock) >0)
        {
            printf("pthread_create error\n");
            return -1;
        }

        //将线程设置成分离态
        pthread_detach(tid);
    }

    //关闭服务器
    close(lis_socket);
    return 0;
}
```

2> http.cpp

```
#include"http.h"
#include"custom_handle.h"
#include<myhead.h>

//声明初始化服务器函数
int  init_server(int _port)
{
    //创建套接字
    int sock = socket(AF_INET, SOCK_STREAM, 0);
    if(sock == -1)
    {
        perror("socke error");
        return  -1;
    }

    //端口号快速重用
    int reuse = 1;
    if(setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &reuse, sizeof(reuse))==-1)
    {
        perror("setsockopt error");
        return -1;
    }

    //绑定ip和端口号
    //填充地址信息结构体
    struct sockaddr_in local;
    local.sin_family = AF_INET;    //协议族
    local.sin_port = htons(_port);    //端口号
    local.sin_addr.s_addr = INADDR_ANY;     //表示可以接收任意主机的连接请求

    //绑定工作
    if(bind(sock, (struct sockaddr*)&local, sizeof(local))==-1)
    {
        perror("bind error");
        return -1;
    }


    //启动监听
    if(listen(sock, 128) == -1)
    {
        perror("listen error");
        return -1;
    }

    //程序执行至此，说明服务器创建成功
    return sock;

}


//自定义从客户端套接字中读取一行信息函数
int get_line(int sock, char *buf)
{
    char ch = '\0';     //读取一个字符
```

```
    int i=0;      //填充变量，用于填充buf，所以从0开始
    int ret = 0;       //记录读取数据的返回值

    while(i<SIZE && ch!='\n')       //当buf没有满并且，没有读取到换行时，继续读取
    {
        ret = recv(sock, &ch, 1, 0);       //从缓冲区中读取一字节数据放入ch中
        if(ret>0 && ch=='\r')
        {
            //下一个字符可能是'\n'
            int s = recv(sock, &ch, 1, MSG_PEEK);      //将下一个字符拿出来看看
            if(s>0 && ch=='\n')
            {
                recv(sock, &ch, 1, 0);      //将回车读取出来
            }else
            {
                ch = '\n';         //直接放入回车，结束
            }
        }

        //程序执行至此，表示该字符需要放入字符串中
        buf[i] = ch;       //将读取下来的字符放入数组中
        i++;              //继续填充下一个位置
    }
    buf[i] = '\0';         //将字符串补充完整
    return i;           //将字符数组中字符串实际长度返回
}

//定义清空头部的函数
static void clear_header(int sock)
{
    char buf[4096] = "";     //读取消息的容器
    int ret = 0;       //读取的个数
    do
    {
        ret = get_line(sock, buf);           //循环将每一行数据全部读取出来
    } while (ret!=1 && strcmp(buf, "\n")!=0);

}



//定义展示404界面的函数
static void show_404(int sock)
{
    //将获取的消息头部全部清除掉，上面解析数据时，只解析了请求行
    clear_header(sock);

    //组装首部信息
    const char *msg = "HTTP/1.1 404 Not Found\r\n";
    //发送首部
    send(sock, msg, strlen(msg), 0);
    send(sock, "\r\n", strlen("\r\n"), 0);      //发送一个空行，切记切记

    //打开一个html页面文档
    struct  stat st;      //定义一个文件状态结构体
    stat("../wwwroot/404.html", &st);          //获取wwwroot目录下的404界面信息，主要
使用该信息中文件大小的成员
    int fd = open("../wwwroot/404.html", O_RDONLY);     //以只读的形式打开文件
```

```
    if(fd == -1)
    {
        perror("open 404 error");
        return;
    }

    //将整个文件发送出去
    //函数原型：ssize_t sendfile(int out_fd, int in_fd, off_t *offset, size_t
count);
    //功能：将一个文件描述符中的数据发送到另一个文件描述符中
    //参数1：要被拷贝到的文件描述符
    //参数2：从哪个文件中拷贝
    //参数3：从该文件的哪个位置开始拷贝
    //参数4：拷贝的文件大小
    sendfile(sock, fd, NULL, st.st_size);
    close(fd);

}


//定义处理错误信息的响应
//参数1：套接字文件描述符
//参数2：错误码
void echo_error(int sock, int err_code)
{
    //对错误码进行多分枝选择
    switch(err_code)
    {
        case 403:
            break;
        case 404:
            show_404(sock);     //调用展示404界面
            break;
        case 405:
            break;
        case 500:
            break;
        default:
            break;
    }
}


//定义响应普通页面的函数
static int echo_www(int sock, const char *path, size_t s)
{
    //以只读的形式打开文件
    int fd = open(path, O_RDONLY);
    if(fd == -1)
    {
        perror("open error");
        return -1;
    }

    //组装响应首部
    const char *msg = "HTTP/1.1 200 OK\r\n";
    send(sock, msg, strlen(msg), 0);        //发送给客户端网页
    send(sock, "\r\n", strlen("\r\r"), 0);  //发送空行
```

```
    //将整个文件发送给网页端
    if(sendfile(sock, fd, NULL, s) ==-1)
    {
        echo_error(sock, 500);        //返回错误界面
        return -1;
    }

    close(fd);
    return 0;
}

//定义处理POST请求或携带数据的GET请求
static int  handle_request(int sock, const char *method,const char * path, const
char * querry_string)
{
    char line[SIZE] = "";     //存储一行信息
    int ret = 0;          //接收读取的内容
    int content_len = -1;    //文本长度

    //判断是什么请求
    if(strcasecmp(method, "GET") == 0)          //GET请求
    {
        //清空消息报头
        clear_header(sock);
    }else
    {
        //获取POST请求的参数大小
        do
        {
            ret = get_line(sock, line);
            if(strncasecmp(line, "content-length", 14) == 0)        //找到该长度
            {
                content_len = atoi(line+16);    //将数字字符串转换为整数
            }
        }while(ret!=1 && (strcmp(line, "\n"))!=0);
    }

    //输出相关数据
    printf("method = %s\n", method);
    printf("query_string = %s\n", querry_string);
    printf("content_len = %d\n", content_len);

    char req_buf[4096] = "";       //用于回复的消息
    //如果是POST请求，那么肯定会携带数据，需要将数据解析出来
    if(strcasecmp(method, "POST") == 0)
    {
        int len = recv(sock, req_buf, content_len, 0);       //将所需信息读取出来
        printf("len = %d\n", len);
        printf("req_buf = %s\n", req_buf);
    }

    //发送给客户端消息
    const char *msg = "HTTP/1.1 200 OK\r\n\r\n";        //回复报文响应首部
    send(sock, msg, strlen(msg), 0);

    //请求交给自定义的函数来处理业务逻辑
    parse_and_process(sock, querry_string, req_buf);
```

```
}



//用于处理客户端信息函数的定义
int handler_msg(int sock)
{
    //定义容器存储客户端发来的数据
    char del_buf[SIZE] = "";

    //读取客户端发来的请求
    //通常我们使用recv接收客户端请求时，最后一个参数如果是0表示阻塞读取，并将数据读走
    //而最后一个参数为MSG_PEEK时，仅仅指示表示查看套接字中的数据，并不读走
    recv(sock, del_buf, SIZE, MSG_PEEK);

    #if 1
    //看一看客户端发来的数据
    printf("--------------------------------------------------\n");
    printf("%s\n", del_buf);
    printf("--------------------------------------------------\n");
    #endif

    //接下来就是解析HTTP请求
    //获取请求行
    char buf[SIZE] = "";         //用于存储读取下来的客户端请求信息的一行
    int count = get_line(sock, buf);
    //功能：获取套接字中的一行信息
    //参数1：套接字文件描述符
    //参数2：读取一行的字符串
    //返回值：当前行的字符串长度

    //程序执行至此，表示buf中已经存储了请求行的信息
    //接下来需要解析请求行中的请求方法、请求url。。。
    char method[32] = "";       //存储请求方法的容器
    int k = 0;              //填充请求方法

    //遍历请求行首部  ---> buf
    int i = 0;          //用于遍历buf的循环变量
    for(i; i<count; i++)
    {
        //找到了任意一个字符
        if(isspace(buf[i]))        //如果该字符是空格，直接结束遍历
        {
            break;
        }

        method[k++] = buf[i];       //将字符放入方法串中
    }
    method[k] = '\0';           //将字符串补充完整
    //程序执行至此，method数组中就存储了请求方法

    //将空格跳过
    while(isspace(buf[i]) && i<SIZE)
    {
        i++;
    }
    //程序执行至此，i就记录了buf中后面的第一个非空字符串
```

```
    //判断请求方法是GET请求还是POST请求
    if(strcasecmp(method, "GET")!=0 && strcasecmp(method,"POST")!=0)
    {
        //说明该请求方法既不是GET请求也不是POST请求
        printf("method error\n");
        //echo_error(sock, 405);       //向客户端回复一个错误页面
        close(sock);            //关闭套接字
        return -1;
    }

    //判断是否为POST请求
    int need_handle = 0;           //标识是否要进行手动处理界面，如果是1，则需要对数据处理，0表示不需要处理
    if(strcasecmp(method, "POST") == 0)
    {
        need_handle = 1;
    }

    //拿去要处理的url以及发送的数据（如果有?的话）
    char url[SIZE] = "";        //存储要解析的url
    int t = 0;                 //填充url字符串的变量
    char *querry_string = NULL;     //指向url中，是否有要处理的数据，如果有，则指向要处理数据的起始地址

    for(i; i<SIZE; i++)      //继续向后遍历请求首部
    {
        //可能还会出现空格
        if(isspace(buf[i]))
        {
            break;                 //表示url读取结束
        }

        //此时表示buf[i]是一个url中的一个字符
        if(buf[i] == '?')            //说明请求中有数据要处理
        {
            //将资源路径保存至url字符数组中，并且使用querry_string指向附加数据
            querry_string = &url[t];
            querry_string ++;           //表示指向问号后的字符串
            url[t] = '\0';              //表示结束
        }else
        {
            url[t] = buf[i];       //其余普通字符，直接放入url容器中
        }

        t++;          //继续填充下一个url内容
    }
    url[t] = '\0';         //将字符串补充完整

    printf("url = %s\n", url);
    printf("querry_string = %s\n", querry_string);

    //程序执行至此，表示url数据也已经拿取下来了

    //如果是GET请求，并且有附带数据，也是需要手动处理数据的
    //例如：http://192.168.31.245:8080/index.html?tt=234，需要进行额外的处理
    if(strcasecmp(method, "GET")==0 && querry_string!=NULL)
    {
```

```
        need_handle = 1;            //也需要进行手动处理
    }

    //我们可以把请求资源路径固定为 wwwroot 下的资源
    char path[SIZE] = "";           //用于确定要响应的文件路径
    sprintf(path, "../wwwroot%s", url);         //将url合成一个服务器中的路径


    //如果请求的没有的地址没有任何携带资源，那么默认返回index.html
    if(path[strlen(path)-1] == '/')
    {
        strcat(path, "index.html");
    }

    printf("path = %s\n", path);



    //判断当前服务器中是否有该path
    struct stat st;          //定义文件状态类型的结构体
    if(stat(path, &st) ==-1)                //如果指定的文件存在，则会把该文件的信息放入
st结构体中，如果不存在，函数返回-1
    {
        //说明要访问的文件不存在
        printf("can not find file\n");
        echo_error(sock, 404);
        close(sock);
        return -1;
    }

    //程序执行至此，表示能够确定是否需要自己来处理后续逻辑了

    //如果是POST请求或者是携带数据的GET请求，都需要手动书写逻辑进行处理
    if(need_handle == 1)
    {
        handle_request(sock, method, path, querry_string);
        //调用处理请求函数
        //参数1：套接字文件描述符
        //参数2：请求方法
        //参数3：请求的路径
        //参数4：请求附带的数据
    }else
    {
        //调用清除 请求首部剩余的内容
        clear_header(sock);

        //此时表示是GET请求,并且没有附加数据，则直接返回请求的界面即可
        echo_www(sock, path, st.st_size);
    }


    //表示所有请求都正常处理了
    close(sock);         //关闭客户端
    return 0;
}
```

3> custom_handle.cpp

```c
#include"custom_handle.h"
#include<myhead.h>

#define KB 1024
#define HTML_SIZE (64*1024)

//处理求和的相关逻辑
static int handle_add(int sock, const char * req_buf)
{
    int num1, num2;     //存储传过来的两个数据

    //使用函数，将数据解析出来
    sscanf(req_buf, "\"data1=%ddata2=%d\"", &num1, &num2);
    printf("num1 = %d, num2 = %d\n", num1, num2);

    char reply_buf[HTML_SIZE] = "";
    sprintf(reply_buf, "%d", num1+num2);       //将和转换为字符串
    printf("reply_buf = %s\n", reply_buf);

    send(sock, reply_buf, strlen(reply_buf), 0);


    return 0;

}

//定义处理登录界面的函数
int handle_login(int sock, char *req_buf)
{
    char reply_buf[HTML_SIZE] = "";     //定义用于回复消息的容器

    //解析数据
    char *uname = strstr(req_buf, "username=");     //将uname锁定到账号的起始位置
    //将指针偏移
    uname += strlen("username=");
    char *ptr = strstr(req_buf,"password=");       //锁定密码的位置
    *(ptr-1) = '\0';       //这样账号就成一个新的字符串

    //锁定密码
    char *passwd = ptr+strlen("password=");
    printf("账号：%s    密码：%s", uname, passwd);

    //判断账号和密码，登录逻辑：只要账号和密码一致，就说明登录成功
    if(strcmp(uname, passwd) == 0)
    {
        //此处表示登录成功，登录成功后，我们跳转到首页
        sprintf(reply_buf, "<script>localStorage.setItem('usr_user_name', '%s');
</script>", uname);
        strcat(reply_buf, "<script>window.location.href='/index.html';
</script>");

        //将消息发送给网页端
        send(sock, reply_buf, strlen(reply_buf), 0);
    }


}
```

```
//处理相关数据使用的函数
int parse_and_process(int sock, const char *querry_string, char * req_buf)
{
    //先处理求和请求  "hello world"   , world
    if(strstr(req_buf, "data1=") && strstr(req_buf, "data2="))
    {
        return handle_add(sock, req_buf);

    }else if(strstr(req_buf,"username=") && strstr(req_buf,"password="))    //处
理登录界面
    {
        return handle_login(sock, req_buf);

    }else
    {
        //处理其他的请求
    }


}
```

4> CMakeLists.txt

```
# 规定最小版本
cmake_minimum_required(VERSION 2.8)

# 项目名称
project(thttpd)

# 添加可执行文件
add_executable(thttpd.out main.cpp http.cpp custom_handle.cpp)

# 链接线程支持库
target_link_libraries(thttpd.out pthread)
```