一、课前准备

- 了解Web编程基本概念
- GoLand 2019.2 EAP
- GoLang 1.10+

二、课堂主题

说明：

熟悉 Go 原生的 Web 编程包，socket、net/http等

三、课堂目标

说明：完成本章课程案例

四、知识点（140分钟）

目前 Go 社区已经有非常多关于 Web 开发的库或框架。大而全的有beego、revel、iris。超高性能的有echo、fasthttp、gin（目前 GitHub 星标最多）。还有不少专注于具体某个方面的，最多要属路由了，例如：mux、httprouter。

课程从Go语言原生的net/http包开始学起。

为什么还要从最原始的 net/http 包开始呢？因为这些库/框架大多是基于 net/http 包做了包装，提供易于使用的功能，如路由参数（`/:name/:age`）/路由分组等。熟练掌握了基础知识和net/http，学习其他框架必然能有事半功倍的效果。

而且由于现代Web项目的复杂化，在baidu、aliyun、字节跳动等大型团队的项目中，可以发现开发中并没有解决所有业务场景的框架，那么选择从原生包的基础上进行修改和拓展，也就是成本最优的解决方案。（类似的大型项目管理思想，在微服务案例阶段，会穿插讲解）

1. 现代Web服务（20分钟）

我们平时浏览网页的时候,会打开浏览器，输入网址后按下回车键，然后就会显示出你想要浏览的内容。

在这个看似简单的用户行为背后，到底隐藏了些什么呢?

对于普通的上网过程，系统其实是这样做的：

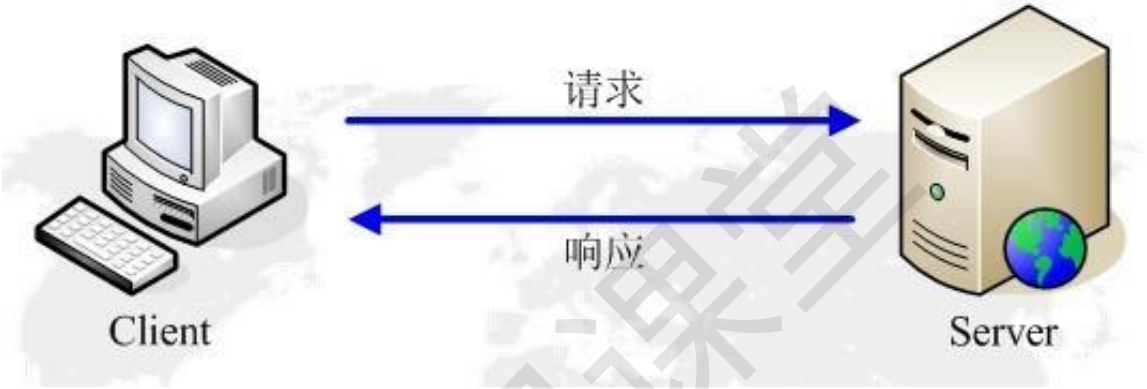
> 浏览器本身是一个客户端，当你输入URL的时候，首先浏览器会去请求**DNS服务器**，通过DNS获取相应的域名对应的IP，然后通过**IP地址**找到IP对应的服务器后，要求**建立TCP连接**，等浏览器**发送完HTTP Request（请求）**包后，服务器接收到请求包之后才开始处理请求包，服务器调用自身服务，**返回HTTP Response（响应）包**；客户端收到来自服务器的响应后开始渲染这个Response包里的主体（body），等收到全部的内容随后**断开与该服务器之间的TCP连接**。

以上是典型的HTTP方式的Web运行，作为服务器端开发，我们需要的就是处理请求和返回响应。

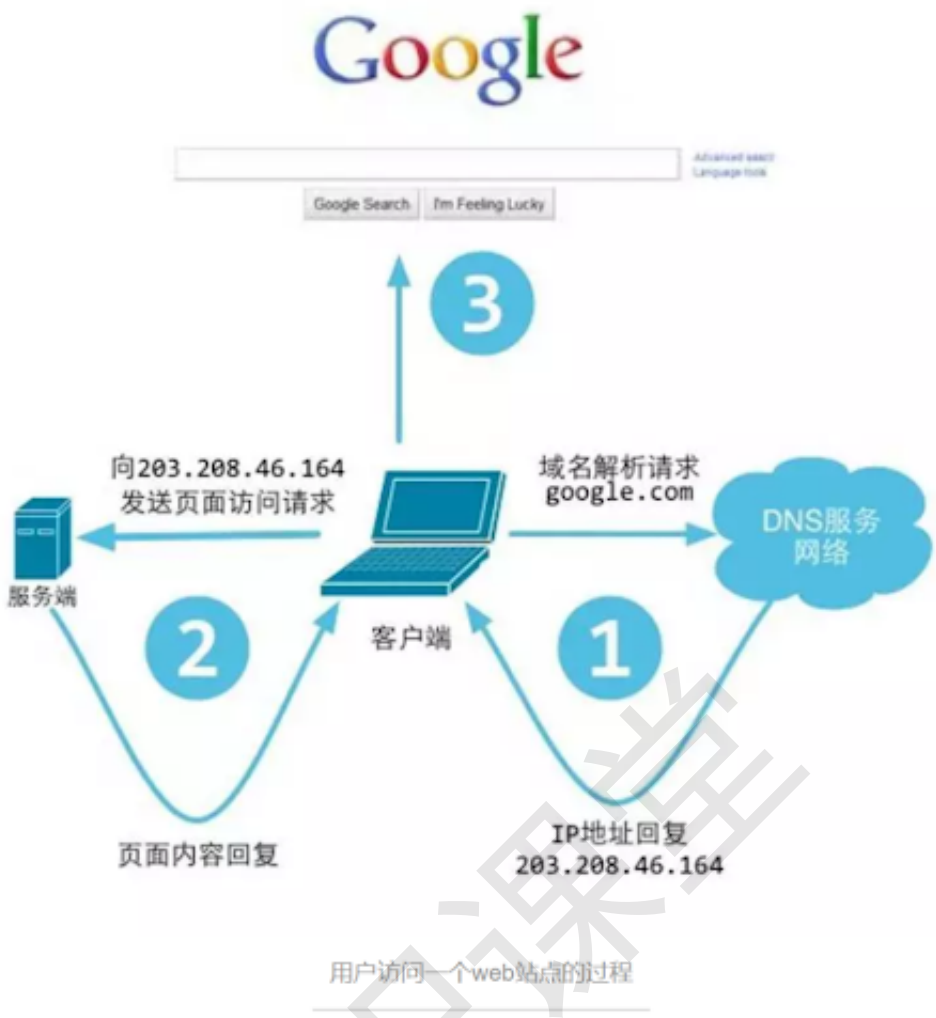
建立在HTTP协议之上，通过XML或者JSON来交换信息的都可以称作是：**Web服务**。

Web工作方式

HTTP协议工作于客户端-服务端架构为上。浏览器作为HTTP客户端通过URL向HTTP服务端即WEB服务器发送所有请求。Web服务器根据接收到的请求后，向客户端发送响应信息。



我们平时浏览网页的时候，会打开浏览器，输入网址后按下回车键，然后就会显示出你想要浏览的内容。在这个看似简单的用户行为背后，到底隐藏了些什么呢? 对于普通的上网过程，系统其实是这样做的：浏览器本身是一个客户端，当你输入URL的时候，首先浏览器会去请求DNS服务器，通过DNS获取相应的域名对应的IP，然后通过IP地址找到IP对应的服务器后，要求建立TCP连接，等浏览器发送完HTTP Request (请求)包后，服务器接收到请求包之后才开始处理请求包，服务器调用自身服务，返回HTTP Response (响应)包；客户端收到来自服务器的响应后开始渲染这个Response包里的主体(body)，等收到全部的内容随后断开与该服务器之间的TCP连接。



用户访问一个web站点的过程

一个Web服务器也被称为HTTP服务器，它通过HTTP协议与客户端通信。这个客户端通常指的是Web浏览器(其实手机端客户端内部也是浏览器实现的)。 Web服务器的工作原理可以简单地归纳为:

- 客户端通过TCP/IP协议建立到服务器的TCP连接
- 客户端向服务器发送HTTP协议请求包，请求服务器里的资源文档
- 服务器向客户端发送HTTP协议应答包，如果请求的资源包含有动态语言的内容，那么服务器会调用动态语言 的解释引擎负责处理“ 动态内容”，并将处理得到的数据返回给客户端
- 客户端与服务器断开。由客户端解释HTML文档，在客户端屏幕上渲染图形结果

一个简单的HTTP事务就是这样实现的，看起来很复杂，原理其实是挺简单的。需要注意的是客户端与服务器之间的通信是非持久连接的，也就是当服务器发送了应答后就与客户端断开连接，等待下一次请求。

> 第一次请求url，服务器返回的是html页面，然后浏览器开始渲染HTML。当解析到HTML DOM里面的图片连接，css脚本和js脚本的连接，浏览器会自动发起一个请求静态资源的HTTP请求，获取相应静态资源，然后浏览器会渲染出来，最终将所有资源整合、渲染、完整展现在屏幕上。(网页优化有一向措施是减少HTTP请求次数，把尽量多的css和js资源合并在一起)。

URL和DNS解析

URL

我们浏览网页都是通过URL访问的，那么URL到底是怎么样的呢? URL (Uni form Resource Locator)是“统一资源定位符” 的英文缩写，用于描述一个网络 上的资源，基本格式如下
`scheme://host[:port#]/path/.../[?query-string][#anchor]`

- scheme

  指定低层使用的协议(例如: http, https, ftp)
- host

  HTTP服务器的I P地址或者域名
- port#

  HTTP服务器的默认端口是80,这种情况下端口号可以省略。如果使用了别的端口,必须指明，例
- path

  访问资源的路径
- query-string

  发送给http服务器的数据
- anchor

  锚
- `https://www.baidu.com/s?wd=程潇`

举个例子：

让我们来解析一下下面这一段：

http://mail.163.com/index.html

1、http://:这个是协议，也就是HTTP超文本传输协议，也就是网页在网上传输的协议。

2、mail：这个是服务器名，代表着是一个邮箱服务器，所以是mail.

3、163.com:这个是域名，是用来定位网站的独一无二的名字。

4、mail.163.com：这个是网站名，由服务器名+域名组成。

5、/：这个是根目录，也就是说，通过网站名找到服务器，然后在服务器存放网页的根目录

6、index.html：这个是根目录下的默认网页（当然，163的默认网页是不是这个我不知道，只是大部分的默认网页，都是index.html）

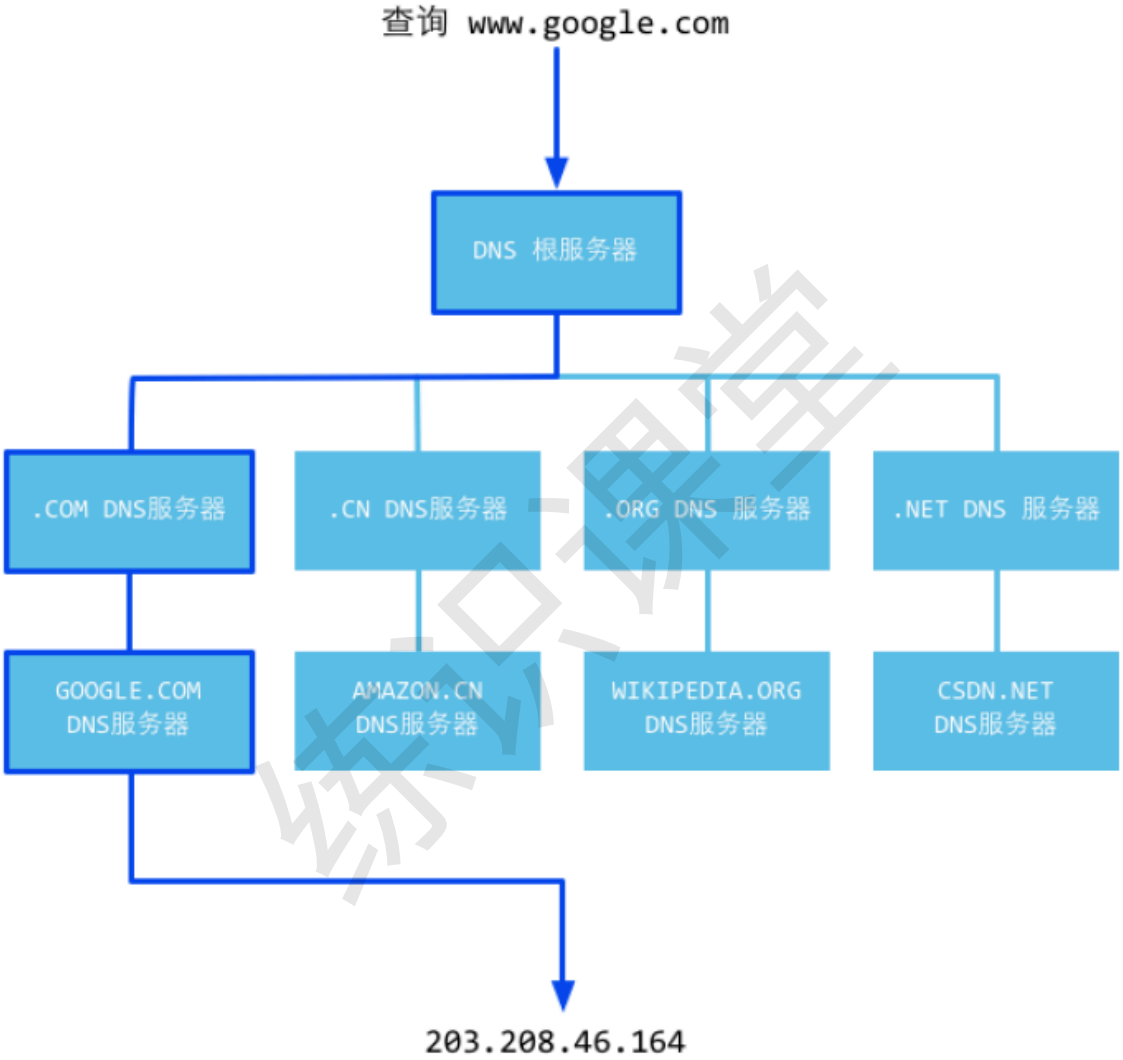
7、http://mail.163.com/index.html:这个叫做URL，统一资源定位符，全球性地址，用于定位网上的资源。

> URI：uniform resource identifier， 统一资源标识符，用来唯一的标识一个资源。
>
> URL：uniform resource locator， 统一资源定位器，它是一种具体的URI，即URL可以用来标识一个资源，而且还指明了如何locate这个资源。

> URN：uniform resource name， 统一资源命名，是通过名字来标识资源，比如mailto:lianshi@lianshiclass.com。
>
> 也就是说，URI是以一种抽象的，高层次概念定义统一资源标识，而URL和URN则是具体的资源标识的方式。URL和URN都是一种URI。

DNS

DNS (Domain Name System)是 “城名系统”的英文缩写，是一种组织成域层次结构的计算机和网络服务命名系统，它用于TCP/IP网络，它从事将主机名或域名转换为实际IP地址的工作。DNS就是这样的一位“翻译官”，它的基本工作原理可用下图来表示。



DNS解析过程

1. 浏览器中输入域名，操作系统会先检查自己本地的hosts文件是否有这个网络映射关系，如果有，就先调用这个IP地址映射，完成域名解析。
2. 如果hosts没有域名，查找本地DNS解析器缓存，如果有直接返回，完成域名解析。
3. 如果还没找到，会查找TCP/IP参数中设置的首选DNS服务器，我们叫它本地DNS服务器，此服务收到查询时，如果要查询的域名包含在本地配置区域资源中，则返回解析结果给客户机，完成域名解析，此解析具有权威性。
4. 如果要查询的域名，不由本地DNS服务器区域解析，但该服务已经缓存了地址映射关系，则调用这个IP地址映射，完成域名解析，此解析不具有权威性。
5. 如果上述过程失败，则根据本地DNS服务器的设置进行查询，如果未用转发模式，则把请求发给根服务器，根服务器返回一个负责该顶级服务器的IP，本地DNS服务器收到IP信息后，再连接该IP上

的服务器进行解析，如果仍然无法解析，则发送下一级DNS服务器，重复操作，直到找到。

6. 如果是转发模式则把请求转发至上一级DNS服务器，假如仍然不能解析，再转发给上上级。不管是否转发，最后都把结果返回给本地DNS服务器上述一个是迭代查询，一个是递归查询。递归查询的过程是查询者发生了更替，而迭代查询过程，查询者不变。

通过上面的步骤，我们最后获取的是IP地址，也就是浏览器最后发起请求的时候是基于IP来和服务器做信息交互的。

举个例子来说，你想知道某个一起上法律课的女孩的电话，并且你偷偷拍了她的照片，回到寝室告诉一个很仗义的哥们儿，这个哥们儿二话没说，拍着胸脯告诉你，甭急，我替你查(此处完成了-次递归查询，即，问询者的角色更替)。然后他拿着照片问了学院大四学长，学长告诉他，这姑娘是xx系的；然后这哥们儿马不停蹄又问了xx系的办公室主任助理同学，助理同学说是xx系yy班的，然后很仗义的哥们儿去xx系yy班的班长那里取到了该女孩儿电话。(此处完成若千次迭代查询，即，问询者角色不变，但反复更替问询对象)最后，他把号码交到了你手里。完成整个查询过程。

HTTP协议

1 什么是HTTP协议

HTTP协议是Web工作的核心，所以要了解清楚Web的工作方式就需要详细的了解清楚HTTP是怎么样工作的。

HTTP协议是Hyper Text Transfer Protocol（超文本传输协议）的缩写是一个基于TCP/IP通信协议来传递数据，服务器传输超文本到本地浏览器的传送协议，建立在TCP协议之上，一般采用80端口。HTTP协议工作于客户端-服务端架构上。浏览器可作为HTTP客户端通过URL向HTTP服务端即WEB服务器发送所有请求。Web服务器根据接收到的请求后，向客户端发送响应信息。它是一个无状态的请求/响应协议。

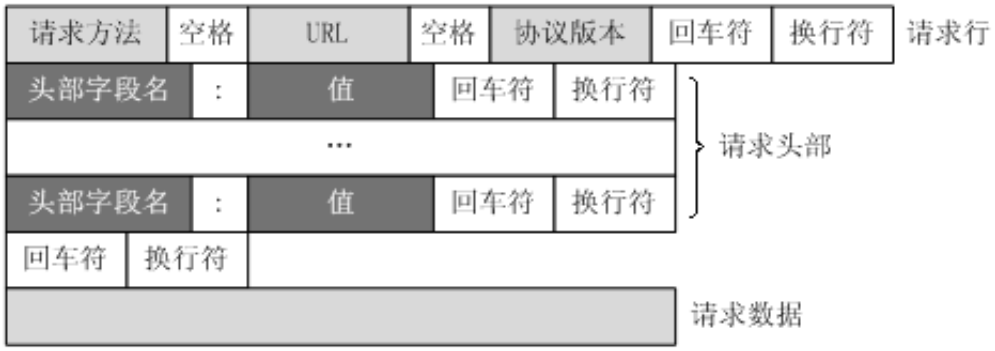
客户端请求消息和服务器响应消息都会包含请求头和请求体。HTTP请求头提供了关于请求或响应，发送实体的信息，如：Content-Type、Content-Length、Date等。当浏览器接收并显示网页前，此网页所在的服务器会返回一个包含HTTP状态码的信息头（server header）用以响应浏览器的请求。

> HTTP是一种让web服务器与客户端通过Internet发送与接受数据的协议，它是一个请求、响应协议，客户端建立连接并发送请求。服务器不能主动去与客户端联系，也不能发送一个回调连接，客户端可提前中断连接。
>
> HTTP请求是无状态的，同一个客户端的每个请求之间没有关联，对HTTP服务器来说，它并不知道这两个请求是否来自同一个客户端。为了解决这个问题引入了cookie机制来维护链接的可持续状态。

2 HTTP请求包

我们先来看看Request包的结构：

由 请求行（request line）、请求头部（header）、空行和请求数据四个部分组成。



第一部分：请求行，用来说明请求类型,要访问的资源以及所使用的HTTP版本

GET说明请求类型为GET,`/`为要访问的资源，该行的最后一部分说明使用的是`HTTP1.1`版本。

第二部分：请求头部，紧接着请求行（即第一行）之后的部分，用来说明服务器要使用的附加信息

从第二行起为请求头部，HOST将指出请求的目的地.User-Agent,服务器端和客户端脚本都能访问它,它是浏览器类型检测逻辑的重要基础.该信息由你的浏览器来定义,并且在每个请求中自动发送等等

第三部分：空行，请求头部后面的空行是必须的

即使第四部分的请求数据为空，也必须有空行。

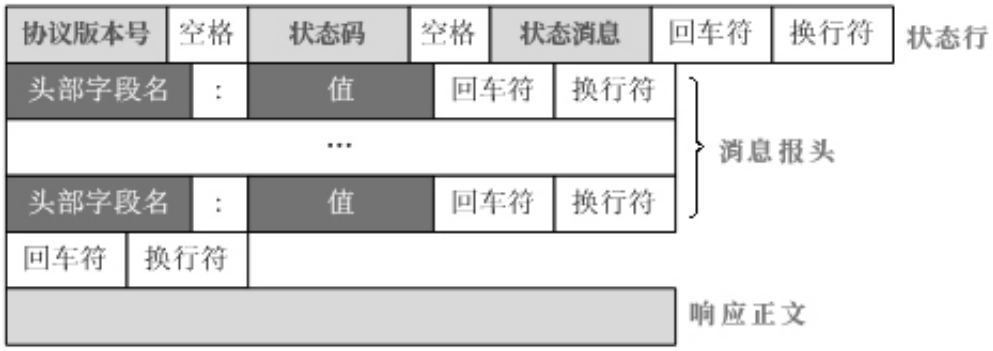
第四部分：请求数据也叫主体，可以添加任意的其他数据

请求包的例子：

```
GET http://edu.kongyixueyuan.com/ HTTP/1.1                  //请求行：请求方法请求URI
HTTP协议/ 协议版本
Accept: application/x-ms-application, image/jpeg, application/xaml+xml,
image/gif, image/pjpeg, application/x-ms-xbap, */*
//客户端能接收的数据格式
Accept-Language: zh-CN
User-Agent: Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 6.1; Win64; x64;
Trident/4.0; .NET CLR 2.0.50727; SLCC2; .NET CLR 3.5.30729; .NET CLR 3.0.30729;
Media Center PC 6.0; .NET4.0C; .NET4.0E)
UA-CPU: AMD64
Accept-Encoding: gzip, deflate                              //是否支持流压缩
Host: edu.kongyixueyuan.com                                 //服务端的主机名
Connection: Keep-Alive
//空行，用于分割请求头和消息体
//消息体,请求资源参数,例如POST传递的参数
```

3 HTTP响应包

我们再来看看HTTP的response包，也由四个部分组成，分别是：状态行、消息报头、空行和响应正文。



第一部分：状态行，由HTTP协议版本号， 状态码， 状态消息 三部分组成。

第一行为状态行，（HTTP/1.1）表明HTTP版本为1.1版本，状态码为200，状态消息为（ok）

第二部分：消息报头，用来说明客户端要使用的一些附加信息

第二行和第三行为消息报头， Date:生成响应的日期和时间；Content-Type:指定了MIME类型的HTML(text/html),编码类型是UTF-8

第三部分：空行，消息报头后面的空行是必须的

第四部分：响应正文，服务器返回给客户端的文本信息。

空行后面的html部分为响应正文。

响应包的例子：

```
HTTP/1.1 200 OK                                          //状态行
Server: nginx                                            //服务器使用的WEB软件名及版本
Content-Type: text/html; charset=UTF-8                   //服务器发送信息的类型
Connection: keep-alive                                   //保持连接状态
Set-Cookie: PHPSESSID=mjup58ggbefu7ni9jea7908kub; path=/; HttpOnly
Cache-Control: no-cache
Date: Wed, 14 Nov 2018 08:27:32 GMT                      //发送时间
Content-Length: 99324                                    //主体内容长度
                                                         //空行用来分割消息头和主体
<!DOCTYPE html>...                                       //消息体
```

状态码用来告诉HTTP客户端, HTTP服务器是否产生了预期的Response。HTTP/1. 1协议中定义了5类状态码，状态码由三位数字组成，第一个数字定义了响应的类别。(HTTP状态码的英文为HTTP Status Code)

- 1XX 提示信息—表示请求已被成功接收，继续处理
- 2XX 成功—表示请求已被成功接收，理解，接受
- 3XX 重定向-要完成请求必须进行更进一步的处理
- 4XX 客户端错误-请求有语法错误或请求无法实现
- 5XX 服务器端错误-服务器未能实现合法的请求

> 常见状态码：

```
200 OK                          //客户端请求成功
400 Bad Request                 //客户端请求有语法错误，不能被服务器所理解
401 Unauthorized                //请求未经授权，这个状态代码必须和WWW-
Authenticate报头域一起使用
403 Forbidden                   //服务器收到请求，但是拒绝提供服务
404 Not Found                   //请求资源不存在，eg：输入了错误的URL
500 Internal Server Error       //服务器发生不可预期的错误
503 Server Unavailable          //服务器当前不能处理客户端的请求，一段时间后可能恢复
正常
```

HTTP协议是无状态的和Connection: keep alive的区别 无状态是指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。从另一方面讲，打开一个服务器上的网页和你之前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(面对无连接)。 从HTTP/1.1起，默认都开启了Keep-Alive保持连接特性，简单地说，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，如果客户端再次访问这个服务器上的网页，会继续使用这一条已经建立的TCP连接。 Keep-Alive不会永久保持连接，它有一个保持时间，可以在不同服务器软件(如Apache) 中设置这个时间。

4 请求方法

根据HTTP标准，HTTP请求可以使用多种请求方法。 HTTP1.0定义了三种请求方法： GET, POST 和HEAD方法。 HTTP1.1新增了五种请求方法：OPTIONS, PUT, DELETE, TRACE 和 CONNECT 方法。

```
GET         请求指定的页面信息，并返回实体主体。
HEAD        类似于get请求，只不过返回的响应中没有具体的内容，用于获取报头
POST        向指定资源提交数据进行处理请求（例如提交表单或者上传文件）。数据被包含在请求体
中。POST请求可能会导致新的资源的建立和/或已有资源的修改。
PUT         从客户端向服务器传送的数据取代指定的文档的内容。
DELETE      请求服务器删除指定的页面。
CONNECT     HTTP/1.1协议中预留给能够将连接改为管道方式的代理服务器。
OPTIONS     允许客户端查看服务器的性能。
TRACE       回显服务器收到的请求，主要用于测试或诊断。
```

常用方法: Get\Post\Head

Http定义了与服务器交互的不同方法，最基本的方法有4种，分别是**GET，POST，PUT，DELETE，**对应着对这个资源的查 ，改 ，增 ，删 4个操作。

另外还有个Head方法. 类似GET方法，只请求页面的首部，不响应页面Body部分，用于获取资源的基本信息，即检查链接的可访问性及资源是否修改。

GET和POST的区别：

- GET在浏览器回退时是无害的，而POST会再次提交请求。
- GET产生的URL地址可以被Bookmark，而POST不可以。
- GET请求会被浏览器主动cache，而POST不会，除非手动设置。
- GET请求只能进行url编码，而POST支持多种编码方式。
- GET请求参数会被完整保留在浏览器历史记录里，而POST中的参数不会被保留。
- GET请求在URL中传送的参数是有长度限制的，而POST没有。
- 对参数的数据类型，GET只接受ASCII字符，而POST没有限制。

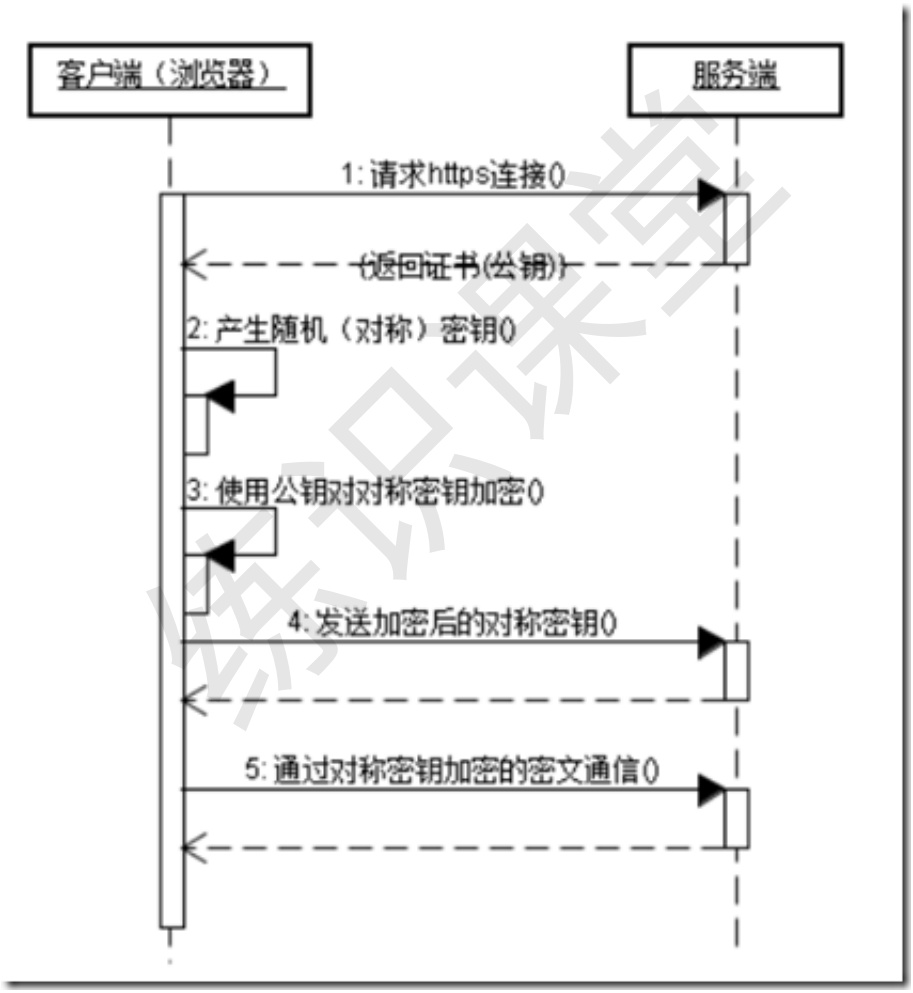
- GET比POST更不安全，因为参数直接暴露在URL上，所以不能用来传递敏感信息。
- GET参数通过URL传递，POST放在Request body中。

HTTP的底层是TCP/IP。所以GET和POST的底层也是TCP/IP，也就是说，GET/POST都是TCP链接。GET产生一个TCP数据包；POST产生两个TCP数据包。对于GET方式的请求，浏览器会把http header和data一并发送出去，服务器响应200（返回数据）；而对于POST，浏览器先发送header，服务器响应100 continue，浏览器再发送data，服务器响应200 ok（返回数据））

HTTPS通信原理

HTTPS（Secure Hypertext Transfer Protocol）安全超文本传输协议 它是一个安全通信通道

HTTPS是HTTP over SSL/TLS，HTTP是应用层协议，TCP是传输层协议，在应用层和传输层之间，增加了一个安全套接层SSL。



服务器 用RSA生成公钥和私钥把公钥放在证书里发送给客户端，私钥自己保存客户端首先向一个权威的服务器检查证书的合法性，如果证书合法，客户端产生一段随机数，这个随机数就作为通信的密钥，我们称之为对称密钥，用公钥加密这段随机数，然后发送到服务器服务器用密钥解密获取对称密钥，然后，双方就已对称密钥进行加密解密通信了。

Https的作用

- 内容加密 建立一个信息安全通道，来保证数据传输的安全；
- 身份认证 确认网站的真实性
- 数据完整性 防止内容被第三方冒充或者篡改

Https和Http的区别

- https协议需要到CA申请证书。
- http是超文本传输协议，信息是明文传输；https 则是具有安全性的ssl加密传输协议。
- http和https使用的是完全不同的连接方式，用的端口也不一样，前者是80，后者是443。
- http的连接很简单，是无状态的；HTTPS协议是由SSL+HTTP协议构建的可进行加密传输、身份认证的网络协议，比http协议安全。

IP TCP UDP HTTP

> 通过对七层网络协议的了解，IP协议对应网络层，TCP协议对应于传输层，而http协议对应于应用层，从本质上来说，三者是不同层面的东西，如果打个比方的话，IP就像高速公路，TCP就如同卡车，http就如同货物，货物要装载在卡车并通过高速公路才能从一个地点送到另一个地点。

那TCP与UDP的区别又是什么呢?

- **TCP** 传输控制协议，Transmission Control Protocol TCP是一种面向连接的、可靠的、基于字节流的传输层通信协议。
- **UDP** 用户数据报协议,User Datagram Protocol UDP是OSI（Open System Interconnection，开放式系统互联） 参考模型中一种无连接的传输层协议，提供面向事务的简单不可靠信息传送服务。

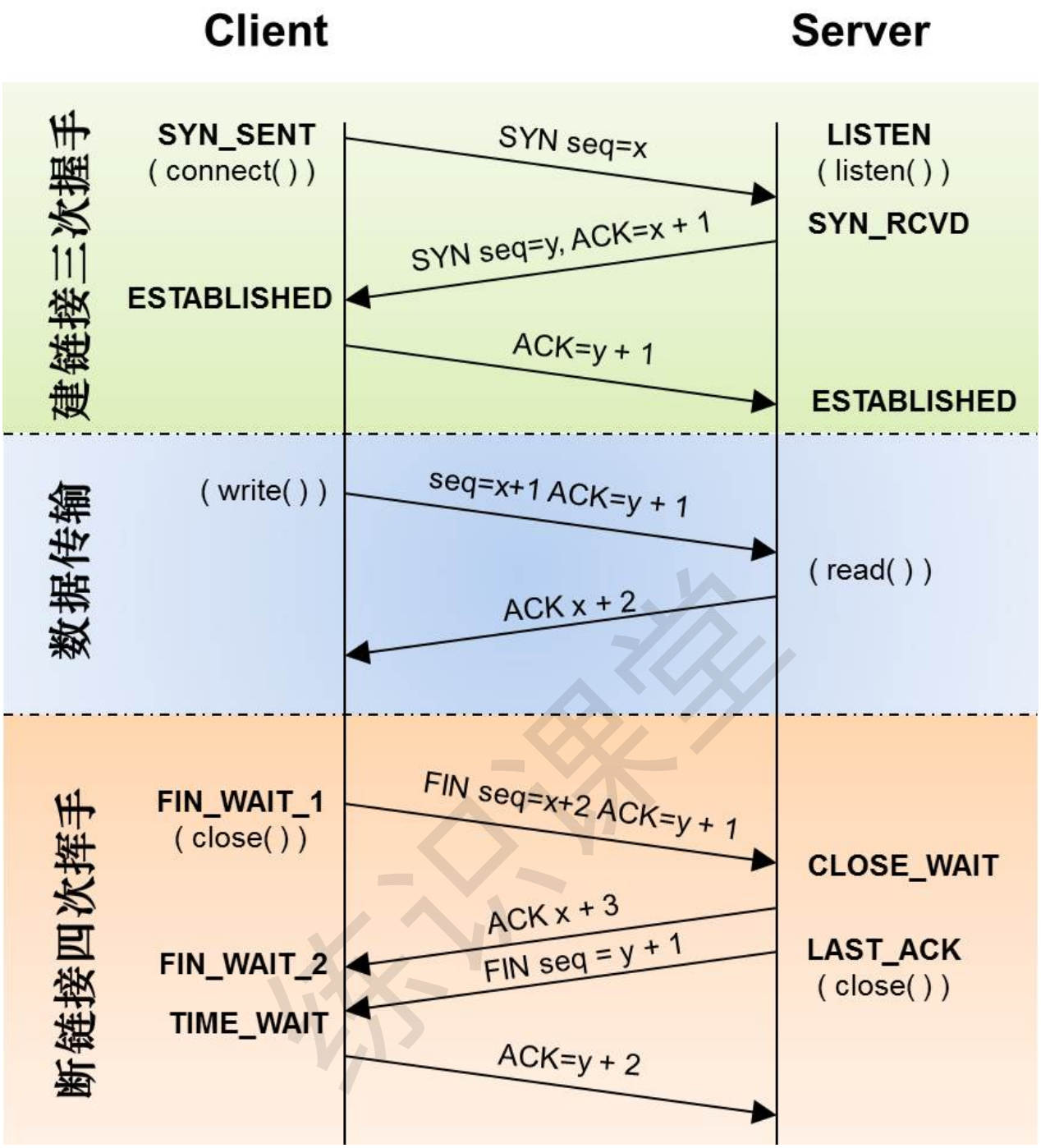
TCP是面向连接的传输控制协议，提供可靠的数据服务（类似于打电话） UDP是提供无连接的数据报服务，传输不可靠，可能丢包（类似于发短信） TCP首部开销20字节，UDP首部开销8字节 TCP只能是点到点的连接，UDP支持一对一，一对多，多对一，多对多的交互通信 TCP逻辑通信信道是全双工的可靠信道，UDP则是不可靠信道

注:什么是单工、半双工、全工通信？ 信息只能单向传送为单工； 信息能双向传送但不能同时双向传送称为半双工； 信息能够同时双向传送则称为全双工。

TCP的三次握手

> TCP建立一个连接需要3次握手IP数据包，断开连接需要4次握手。TCP因为建立连接、释放连接、IP分组校验排序等需要额外工作，速度较UDP慢许多。TCP适合传输数据，UDP适合流媒体

第一次握手：客户端发送syn包(syn=j)到服务器，并进入SYN_SEND状态，等待服务器确认； 第二次握手：服务器收到syn包，必须确认客户的SYN（ack=j+1），同时自己也发送一个SYN包（syn=k），即SYN+ACK包，此时服务器进入SYN_RECV状态； 第三次握手：客户端收到服务器的SYN＋ACK包，向服务器发送确认包ACK(ack=k+1)，此包发送完毕，客户端和服务器进入ESTABLISHED状态，完成三次握手。

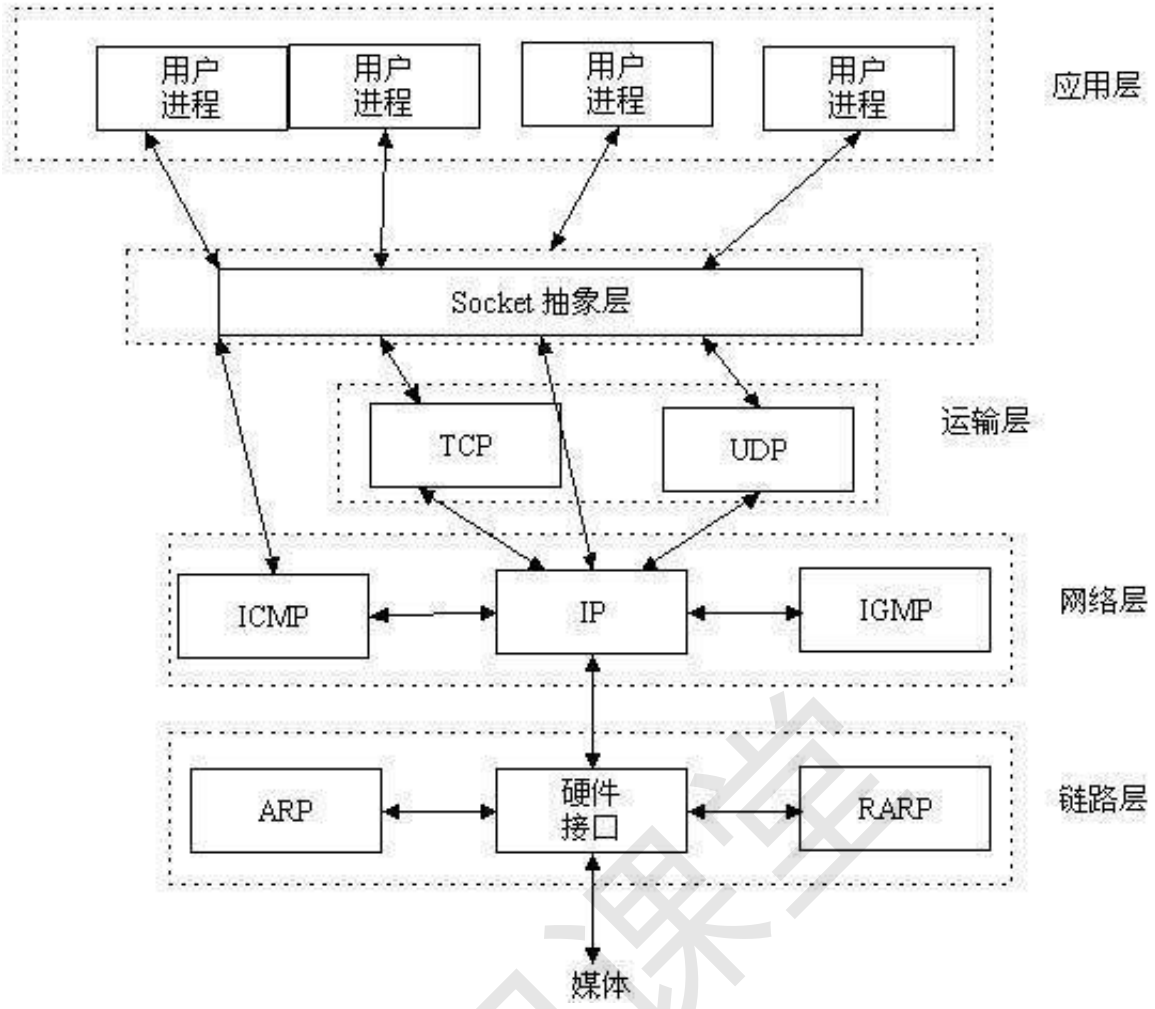


tcp-ip-handshark.png

Socket

我们知道两个进程如果需要进行通讯最基本的一个前提能能够唯一的标示一个进程，在本地进程通讯中我们可以使用PID来唯一标示一个进程，但PID只在本地唯一，网络中的两个进程PID冲突几率很大，这时候我们需要另辟它径了，我们知道IP层的ip地址可以唯一标示主机，而TCP层协议和端口号可以唯一标示主机的一个进程，这样我们可以利用ip地址＋协议＋端口号唯一标示网络中的一个进程。

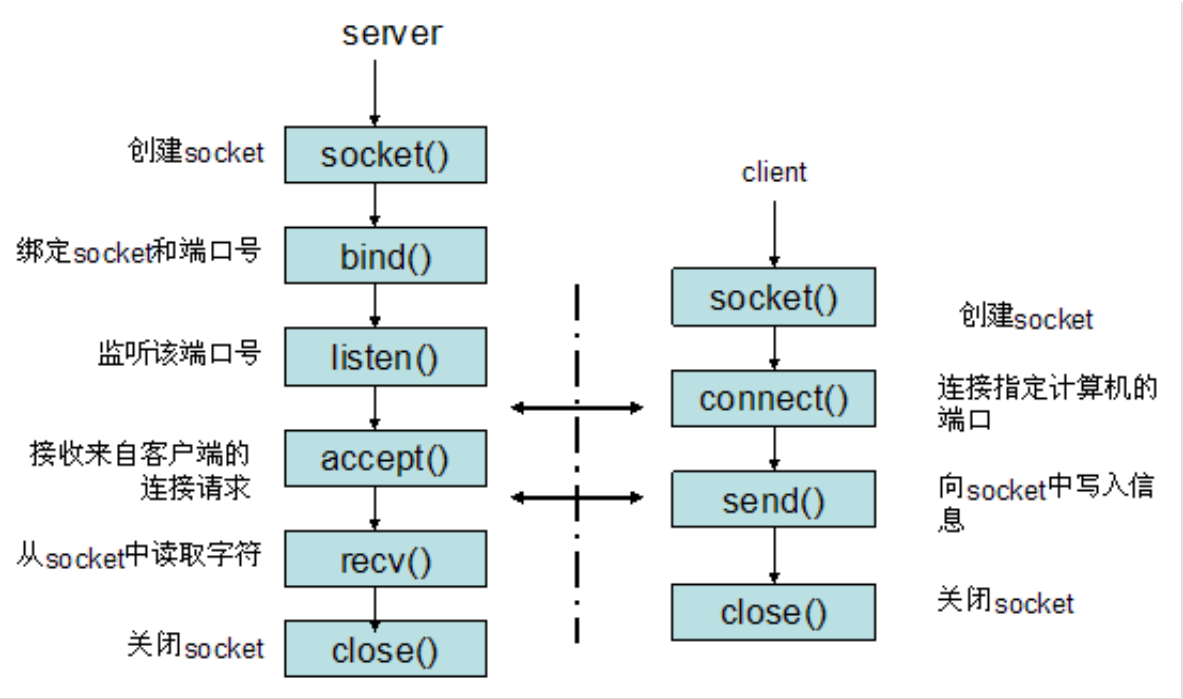
能够唯一标示网络中的进程后，它们就可以利用socket进行通信了，什么是socket呢？我们经常把socket翻译为套接字，socket是在应用层和传输层之间的一个抽象层，它把TCP/IP层复杂的操作抽象为几个简单的接口供应用层调用已实现进程在网络中通信。



socket.jpg

socket起源于UNIX，在Unix一切皆文件哲学的思想下，socket是一种"打开—读/写—关闭"模式的实现，服务器和客户端各自维护一个"文件"，在建立连接打开后，可以向自己文件写入内容供对方读取或者读取对方内容，通讯结束时关闭文件。

Socket通信流程 socket是"打开—读/写—关闭"模式的实现，以使用TCP协议通讯的socket为例，其交互流程大概是下图这样的：



Go代码示例

服务端

```
// tcp/server/main.go

// TCP server端

// 处理函数
func process(conn net.Conn) {
  defer conn.Close() // 关闭连接
  for {
    reader := bufio.NewReader(conn)
    var buf [128]byte
    n, err := reader.Read(buf[:]) // 读取数据
    if err != nil {
      fmt.Println("read from client failed, err:", err)
      break
    }
    recvStr := string(buf[:n])
    fmt.Println("收到client端发来的数据：", recvStr)
    conn.Write([]byte(recvStr)) // 发送数据
  }
}

func main() {
  listen, err := net.Listen("tcp", "127.0.0.1:20000")
  if err != nil {
    fmt.Println("listen failed, err:", err)
    return
  }
  for {
```

```
    conn, err := listen.Accept() // 建立连接
    if err != nil {
      fmt.Println("accept failed, err:", err)
      continue
    }
    go process(conn) // 启动一个goroutine处理连接
  }
}
```

client端

```
// tcp/client/main.go

// 客户端
func main() {
  conn, err := net.Dial("tcp", "127.0.0.1:20000")
  if err != nil {
    fmt.Println("err :", err)
    return
  }
  defer conn.Close() // 关闭连接
  inputReader := bufio.NewReader(os.Stdin)
  for {
    input, _ := inputReader.ReadString('\n') // 读取用户输入
    inputInfo := strings.Trim(input, "\r\n")
    if strings.ToUpper(inputInfo) == "Q" { // 如果输入q就退出
      return
    }
    _, err = conn.Write([]byte(inputInfo)) // 发送数据
    if err != nil {
      return
    }
    buf := [512]byte{}
    n, err := conn.Read(buf[:])
    if err != nil {
      fmt.Println("recv failed, err:", err)
      return
    }
    fmt.Println(string(buf[:n]))
  }
}
```

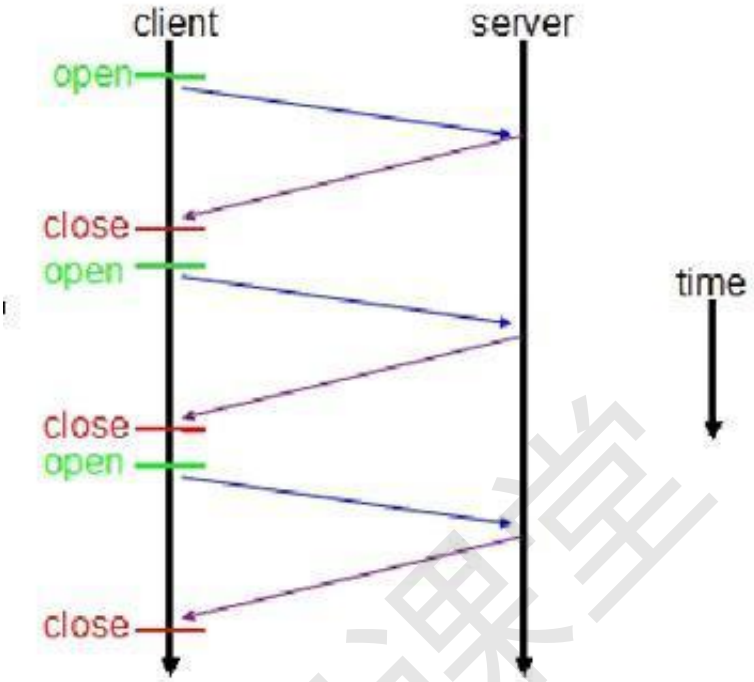

WebSocket

WebSocket protocol 是HTML5一种新的协议。它实现了浏览器与服务器全双工通信，能更好的节省服务器资源和带宽并达到实时通讯它建立在TCP之上，同 HTTP一样通过TCP来传输数据。WebSocket同HTTP一样也是应用层的协议，并且一开始的握手也需要借助HTTP请求完成。

它和 HTTP 最大不同是：

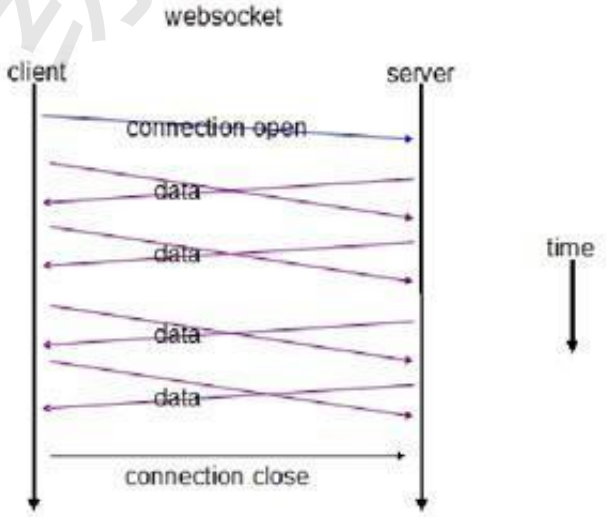
- WebSocket 是一种双向通信协议，在建立连接后，WebSocket 服务器和 Browser/Client Agent 都能主动的向对方发送或接收数据，就像 Socket 一样；
- WebSocket 需要类似 TCP 的客户端和服务器端通过握手连接，连接成功后才能相互通信。

HTTP请求客户端服务器交互图



http-client-server.jpg

WebSocket客户端服务器交互图



websocket-client-server.jpg

上图对比可以看出，相对于传统 HTTP 每次请求-应答都需要客户端与服务端建立连接的模式，WebSocket 是类似 Socket 的 TCP 长连接的通讯模式，一旦 WebSocket 连接建立后，后续数据都以帧序列的形式传输。在客户端断开 WebSocket 连接或 Server 端断掉连接前，不需要客户端和服务端重新发起连接请求。在海量并发及客户端与服务器交互负载流量大的情况下，极大的节省了网络带宽资源的消耗，有明显的性能优势，且客户端发送和接受消息是在同一个持久连接上发起，实时性优势明显。

WebSocket连接过程（握手）

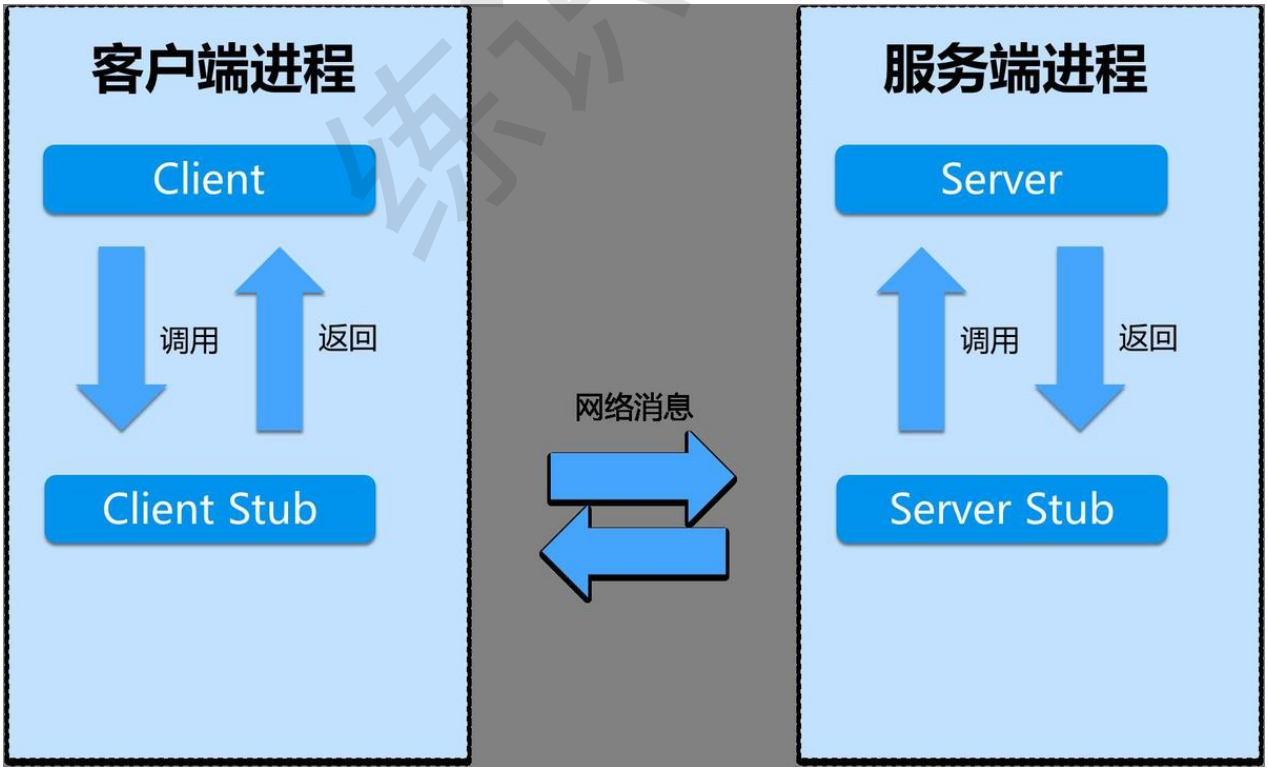
从WebSocket客户端服务器交互图可以看出，在WebSocket中，只需要服务器和浏览器通过HTTP协议进行一个握手的动作，然后单独建立一条TCP的通信通道进行数据的传送。

1. 浏览器，服务器建立TCP连接，三次握手。这是通信的基础，传输控制层，若失败后续都不执行。
2. TCP连接成功后，浏览器通过HTTP协议向服务器传送WebSocket支持的版本号等信息。（开始前的HTTP握手）
3. 服务器收到客户端的握手请求后，同样采用HTTP协议回馈数据。
4. 当收到了连接成功的消息后，通过TCP通道进行传输通信。

RPC

> Remote Procedure Call 远程过程调用 它是一种通过网络从远程计算机程序上请求服务，而不需要了解底层网络技术的协议。RPC协议假定某些传输协议的存在，如TCP或UDP，为通信程序之间携带信息数据。在OSI网络通信模型中，RPC跨越了传输层和应用层。RPC使得开发包括网络分布式多程序在内的应用程序更加容易。

先说说RPC服务的基本架构吧。一个完整的RPC架构里面包含了四个核心的组件，分别是Client ,Server,Client Stub以及Server Stub，这个Stub大家可以理解为存根。



rpc-architecture.jpg

- 客户端（Client），服务的调用方。

- 服务端（Server），真正的服务提供者。
- 客户端存根，存放服务端的地址消息，再将客户端的请求参数打包成网络消息，然后通过网络远程发送给服务方。
- 服务端存根，接收客户端发送过来的消息，将消息解包，并调用本地的方法。

RPC采用客户机/服务器模式，通信是建立在Socket之上的,出于一种类比的愿望,在一台机器上运行的主程序,可以调用另一台机器上准备好的子程序,就像LPC(本地过程调用)。请求程序就是一个客户机，而服务提供程序就是一个服务器。首先，调用进程发送一个有进程参数的调用信息到服务进程，然后等待应答信息。在服务器端，进程保持睡眠状态直到调用信息的到达为止。当一个调用信息到达，服务器获得进程参数，计算结果，发送答复信息，然后等待下一个调用信息，最后，客户端调用过程接收答复信息，获得进程结果，然后调用执行继续进行。

RPC vs HTTP

- 论复杂度，RPC框架肯定是高于简单的HTTP接口的。但毋庸置疑，HTTP接口由于受限于HTTP协议，需要带HTTP请求头，还有三次握手，导致传输起来效率或者说安全性不如RPC。
- HTTP是一种协议,RPC可以通过HTTP来实现,也可以通过Socket自己实现一套协议来实现.
- RPC更是一个软件结构概念，是构建分布式应用的理论基础。就好比为啥你家可以用到发电厂发出来的电？是因为电是可以传输的。至于用铜线还是用铁丝还是其他 种类的导线，也就是用http还是用其他协议的问题了。

Rest & Restful

> Rest全称是Representational State Transfer，中文意思是表述性状态转移。Rest指的是一组架构约束条件和原则。如果一个架构符合Rest的约束条件和原则，我们就称它为Restful架构。

然而Rest本身并没有创造新的技术、组件或服务，而隐藏在Restful背后的理念就是使用Web的现有特征和能力， 更好地使用现有Web标准中的一些准则和约束。我们现在所说的Rest是基于HTTP协议之上来讲的，但Rest架构风格并不是绑定在HTTP上，只不过目前HTTP是唯一与Rest相关的实例。

Rest架构的主要原则

- 在Rest中的一切都被认为是一种资源。
- 每个资源由URI标识。
- 使用统一的接口。处理资源使用POST，GET，PUT，DELETE操作类似创建，读取，更新和删除（CRUD）操作。
- 无状态：每个请求是一个独立的请求。从客户端到服务器的每个请求* * 都必须包含所有必要的信息，以便于理解。
- 同一个资源具有多种表现形式，例如XML，JSON

Restful API 简单例子

```
[POST]     http://localhost/users    // 新增
[GET]      http://localhost/users/1  // 查询
[PATCH]    http://localhost/users/1  // 更新
[PUT]      http://localhost/users/1  // 覆盖，全部更新
[DELETE]   http://localhost/users/1  // 删除
```

2. Go Hello Web！（30分钟）

接下来，我们来编写一个 Web 版本的 "Hello World" 程序。使用 Go 语言提供的 net/http 包。

- 创建`server.go`文件，输入下面内容：

```
package main

import (
    "fmt"
    "log"
    "net/http"
)

func hello(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Hello, Web!")
}

func main() {
    http.HandleFunc("/", hello)
    if err := http.ListenAndServe(":8080", nil); err != nil {
        log.Fatal(err)
    }
}
```

打开命令行，进入项目目录，输入命令：`go run server.go`，我们的第一个服务器程序就跑起来了。

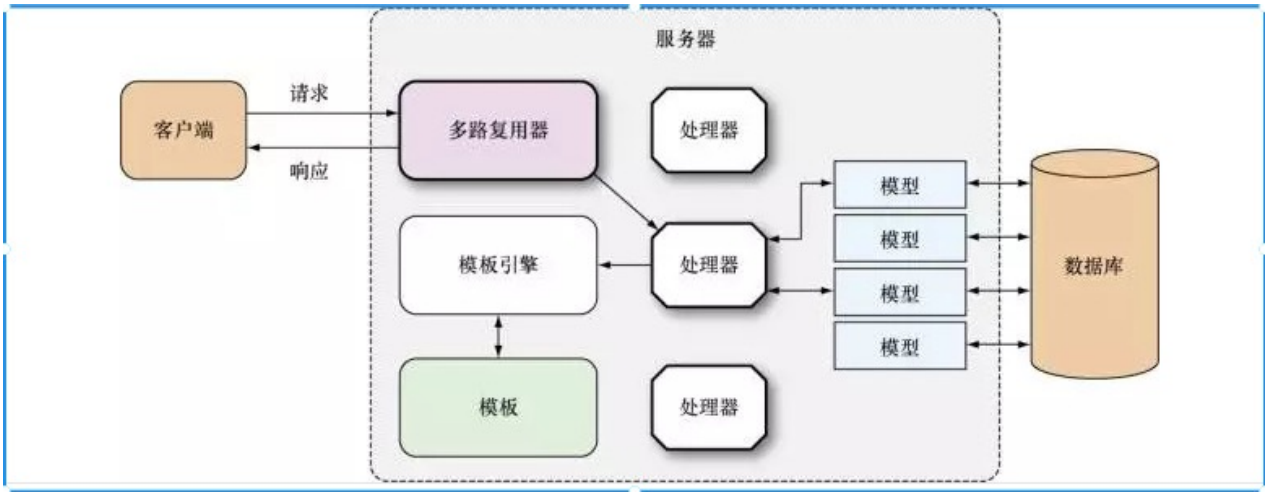
之后打开浏览器，输入网址`localhost:8080`，"Hello, Web!"就在网页上显示出来了。

解析该程序：

> `http.HandleFunc`将`hello`函数注册到根路径`/`上，`hello`函数我们也叫做处理器。它接收两个参数：
>
> 第一个参数为一个类型为`http.ResponseWriter`的接口，响应就是通过它发送给客户端的。
>
> 第二个参数是一个类型为`http.Request`的结构指针，客户端发送的信息都可以通过这个结构获取。
>
> `http.ListenAndServe`将在 8080 端口上监听请求，最后交由`hello`处理。

多路复用器

一个典型的 Go Web 程序结构如下：



- 客户端发送请求；
- 服务器中的多路复用器收到请求；
- 多路复用器根据请求的 URL 找到注册的处理器，将请求交由处理器处理；
- 处理器执行程序逻辑，必要时与数据库进行交互，得到处理结果；
- 处理器调用模板引擎将指定的模板和上一步得到的结果渲染成客户端可识别的数据格式（通常是 HTML）；
- 最后将数据通过响应返回给客户端；
- 客户端拿到数据，执行对应的操作，例如渲染出来呈现给用户。

net/http 包内置了一个默认的多路复用器 `DefaultServeMux`。定义如下：

```
// src/net/http/server.go

// DefaultServeMux is the default ServeMux used by Serve.
var DefaultServeMux = &defaultServeMux

var defaultServeMux ServeMux
```

net/http 包中很多方法都在内部调用 `DefaultServeMux` 的对应方法，如 `HandleFunc`。我们知道，`HandleFunc` 是为指定的 URL 注册一个处理器（准确来说，`hello` 是处理器函数，见下文）。其内部实现如下：

```
// src/net/http/server.go
func HandleFunc(pattern string, handler func(ResponseWriter, *Request)) {
  DefaultServeMux.HandleFunc(pattern, handler)
}
```

实际上，`http.HandleFunc` 方法是将处理器注册到 `DefaultServeMux` 中的。

另外，我们使用 ":8080" 和 `nil` 作为参数调用 `http.ListenAndServe` 时，会创建一个默认的服务器：

```
// src/net/http/server.go
func ListenAndServe(addr string, handler Handler) {
    server := &Server{Addr: addr, Handler: handler}
    return server.ListenAndServe()
}
```

这个服务器默认使用`DefaultServeMux`来处理器请求：

```
type serverHandler struct {
  srv *Server
}

func (sh serverHandler) ServeHTTP(rw ResponseWriter, req *Request) {
  handler := sh.srv.Handler
  if handler == nil {
    handler = DefaultServeMux
  }
  handler.ServeHTTP(rw, req)
}
```

服务器收到的每个请求会调用对应多路复用器（即`ServeMux`）的`ServeHTTP`方法。在`ServeMux`的`ServeHTTP`方法中，根据 URL 查找我们注册的处理器，然后将请求交由它处理。

虽然默认的多路复用器使用起来很方便，但是在生产环境中不建议使用。由于`DefaultServeMux`是一个全局变量，所有代码，**包括第三方代码**都可以修改它。 有些第三方代码会在`DefaultServeMux`注册一些处理器，这可能与我们注册的处理器冲突。

创建多路复用器

创建多路复用器也比较简单，直接调用`http.NewServeMux`方法即可。然后，在新创建的多路复用器上注册处理器：

```
package main

import (
  "fmt"
  "log"
  "net/http"
)

func hello(w http.ResponseWriter, r *http.Request) {
  fmt.Fprintf(w, "Hello, Web")
}

func main() {
  //创建Mux
  mux := http.NewServeMux()
  mux.HandleFunc("/", hello)
```

```
  server := &http.Server{
    Addr:    ":8080",
    Handler: mux, //注册处理器
  }

  if err := server.ListenAndServe(); err != nil {
    log.Fatal(err)
  }
}
```

这里我们还自己创建了服务器对象。通过**指定服务器的参数**，我们可以创建定制化的服务器。

```
server := &http.Server{
  Addr:           ":8080",
  Handler:        mux,
  ReadTimeout:    1 * time.Second,
  WriteTimeout:   1 * time.Second,
}
```

在上面代码，创建了一个读超时和写超时均为 1s 的服务器。

处理器和处理器函数

服务器收到请求后，会根据其 URL 将请求交给相应的处理器处理。处理器实现了`Handler`接口的结构，`Handler`接口定义在 net/http 包中：

```
// src/net/http/server.go
type Handler interface {
    func ServeHTTP(w Response.Writer, r *Request)
}
```

可以定义一个实现该接口的结构，注册这个结构类型的对象到多路复用器中：

```
package main

import (
    "fmt"
    "log"
    "net/http"
)

type GreetingHandler struct {
    Language string
}
```

```
func (h GreetingHandler) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "%s", h.Language)
}

func main() {
    mux := http.NewServeMux()
    mux.Handle("/chinese", GreetingHandler{Language: "你好"})
    mux.Handle("/english", GreetingHandler{Language: "Hello"})

    server := &http.Server {
        Addr:   ":8080",
        Handler: mux,
    }

    if err := server.ListenAndServe(); err != nil {
        log.Fatal(err)
    }
}
```

解析：

定义一个实现`Handler`接口的结构`GreetingHandler`。然后，创建该结构的两个对象，分别将它注册到多路复用器的`/hello`和`/world`路径上。注意，这里注册使用的是`Handle`方法，注意与`HandleFunc`方法对比。

启动服务器之后，在浏览器的地址栏中输入`localhost:8080/chinese`，浏览器中将显示`你好`，输入`localhost:8080/english`将显示`Hello`。

虽然，自定义处理器这种方式比较灵活，强大，但是需要定义一个新的结构，实现`ServeHTTP`方法，还是比较繁琐的。

为了方便使用，net/http 包提供了以函数的方式注册处理器，即使用`HandleFunc`注册。函数必须满足签名：`func (w http.ResponseWriter, r *http.Request)`。这个函数称为处理器函数。`HandleFunc`方法内部，会将传入的处理器函数转换为`HandlerFunc`类型。

```
// src/net/http/server.go
func (mux *ServeMux) HandleFunc(pattern string, handler func(ResponseWriter,
*Request)) {
  if handler == nil {
    panic("http: nil handler")
  }
  mux.Handle(pattern, HandlerFunc(handler))
}
```

`HandlerFunc`是底层类型为`func (w ResponseWriter, r *Request)`的新类型，它可以自定义其方法。由于`HandlerFunc`类型实现了`Handler`接口，所以它也是一个处理器类型，最终使用`Handle`注册。

```
// src/net/http/server.go
type HandlerFunc func(w *ResponseWriter, r *Request)

func (f HandlerFunc) ServeHTTP(w ResponseWriter, r *Request) {
    f(w, r)
}
```

注意，这几个接口和方法名很容易混淆：

- `Handler`：处理器接口，定义在 net/http 包中。实现该接口的类型，其对象可以注册到多路复用器中；
- `Handle`：注册处理器的方法；
- `HandleFunc`：注册处理器函数的方法；
- `HandlerFunc`：底层类型为`func (w ResponseWriter, r *Request)`的新类型，实现了`Handler`接口。它连接了**处理器函数**与**处理器**。

URL 匹配

一般的 Web 服务器有非常多的 URL 绑定，不同的 URL 对应不同的处理器。但是服务器是怎么决定使用哪个处理器的呢？例如，我们现在绑定了 3 个 URL，`/`和`/hello`和`/hello/world`。

显然，

如果请求的 URL 为`/`，则调用`/`对应的处理器。

如果请求的 URL 为`/hello`，则调用`/hello`对应的处理器。

如果请求的 URL 为`/hello/world`，则调用`/hello/world`对应的处理器。

但是，如果请求的是`/hello/others`，那么使用哪一个处理器呢？ 匹配遵循以下规则：

- 首先，精确匹配。即查找是否有`/hello/others`对应的处理器。如果有，则查找结束。如果没有，执行下一步；
- 将路径中最后一个部分去掉，再次查找。即查找`/hello/`对应的处理器。如果有，则查找结束。如果没有，继续执行这一步。即查找`/`对应的处理器。

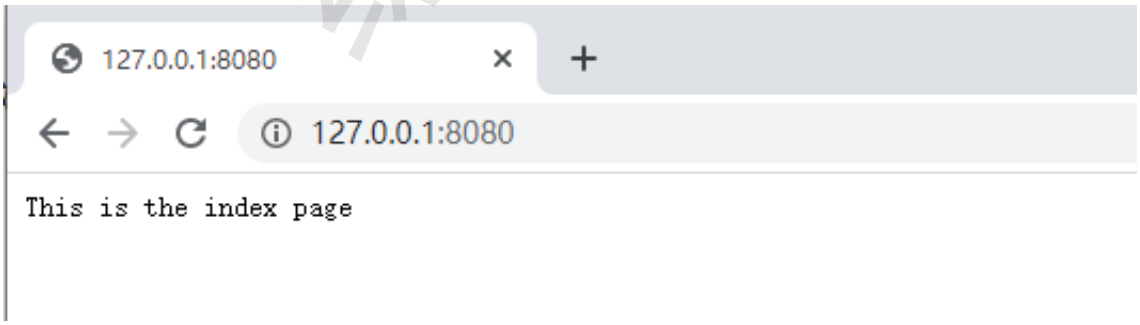
这里有一个注意点，**如果注册的 URL 不是以`/`结尾的，那么它只能精确匹配请求的 URL**。反之，即使请求的 URL 只有**前缀**与被绑定的 URL 相同，`ServeMux`也认为它们是匹配的。

这也是为什么上面步骤进行到`/hello/`时，不能匹配`/hello`的原因。因为`/hello`不以`/`结尾，必须要精确匹配。 如果，我们绑定的 URL 为`/hello/`，那么当服务器找不到与`/hello/others`完全匹配的处理器时，就会退而求其次，开始寻找能够与`/hello/`匹配的处理器。

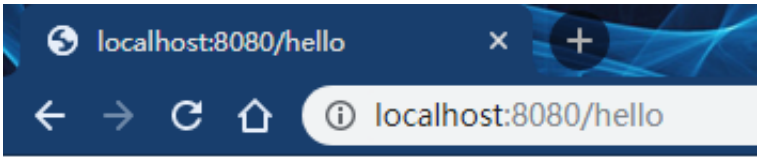
```
package main

import (
  "fmt"
  "log"
  "net/http"
```

```
)

func indexHandler(w http.ResponseWriter, r *http.Request) {
  fmt.Fprintf(w, "This is the index page")
}

func helloHandler(w http.ResponseWriter, r *http.Request) {
  fmt.Fprintf(w, "This is the hello page")
}

func worldHandler(w http.ResponseWriter, r *http.Request) {
  fmt.Fprintf(w, "This is the world page")
}

func main() {
  mux := http.NewServeMux()
  mux.HandleFunc("/", indexHandler)
  mux.HandleFunc("/hello", helloHandler)
  mux.HandleFunc("/hello/world", worldHandler)

  server := &http.Server{
    Addr:    ":8080",
    Handler: mux,
  }

  if err := server.ListenAndServe(); err != nil {
    log.Fatal(err)
  }
}
```

- 浏览器请求`localhost:8080/`将返回`"This is the index page"`，因为`/`精确匹配；

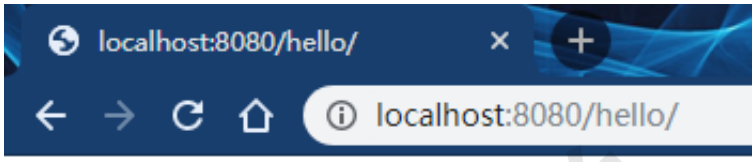


- 浏览器请求`localhost:8080/hello`将返回`"This is the hello page"`，因为`/hello`精确匹配；



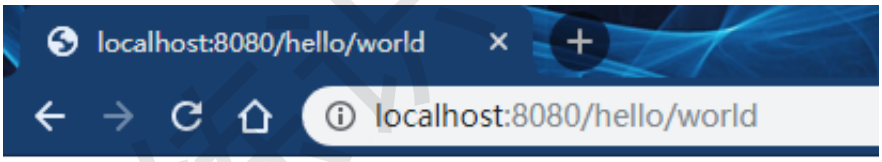
This is the hello page

- 浏览器请求`localhost:8080/hello/`将返回`"This is the index page"`。**注意这里不是`hello`，因为绑定的`/hello`需要精确匹配，而请求的`/hello/`不能与之精确匹配。故而向上查找到`/`；**



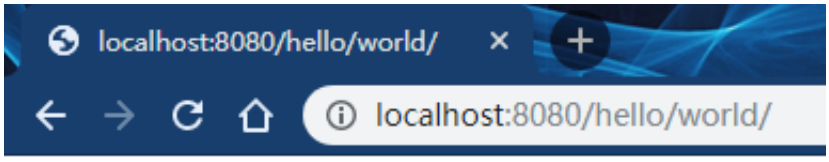
This is the index page

- 浏览器请求`localhost:8080/hello/world`将返回`"This is the world page"`，因为`/hello/world`精确匹配；



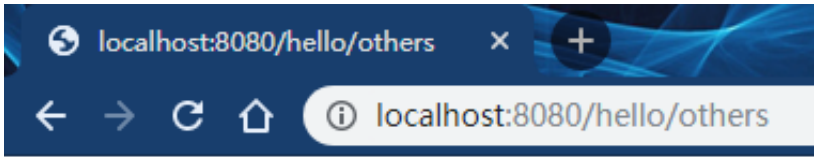
This is the world page

- 浏览器请求`localhost:8080/hello/world/`将返回`"This is the index page"`，**查找步骤为`/hello/world/`（不能与`/hello/world`精确匹配）-> `/hello/`（不能与`/hello/`精确匹配）-> `/`；**



This is the index page

- 浏览器请求`localhost:8080/hello/other`将返回`"This is the index page"`，**查找步骤为`/hello/others` -> `/hello/`（不能与`/hello`精确匹配） -> `/`**；



This is the index page

如果注册时，将`/hello`改为`/hello/`，那么请求`localhost:8080/hello/`和`localhost:8080/hello/world/`都将返回`"This is the hello page"`。自己试试吧！

检查点

> 思考：
>
> 使用`/hello/`注册处理器时，`localhost:8080/hello/`返回什么？

3. http 请求（50分钟）

请求的结构

处理器函数：

```
func (w http.ResponseWriter, r *http.Request)
```

其中，`http.Request`就是请求的类型。客户端传递的数据都可以通过这个结构来获取。结构`Request`定义在包 net/http 中：

```
// src/net/http/request.go

type Request struct {
    Method           string
    URL              *url.URL
    Proto            string
    ProtoMajor       int
    ProtoMinor       int
    Header           Header
    Body             io.ReadCloser
    ContentLength    int
    // 省略一些字段...
}
```

Method

请求中的`Method`字段表示客户端想要调用服务器的 HTTP 协议方法。其取值有`GET/POST/PUT/DELETE`等。服务器根据请求方法的不同会进行不同的处理，例如`GET`方法只是获取信息（用户基本信息，商品信息等），`POST`方法创建新的资源（注册新用户，上架新商品等）。

URL

Tim Berners-Lee 在创建万维网的同时，也引入了使用字符串来表示互联网资源的概念。他称该字符串为**统一资源标识符**（URI，Uniform Resource Identifier）。URI 由两部分组成。一部分表示资源的名称，即**统一资源名称**（URN，Uniform Resource Name）。另一部分表示资源的位置，即**统一资源定位符**（URL，Uniform Resource Location）。

在 HTTP 请求中，使用 URL 来对要操作的资源位置进行描述。URL 的一般格式为：

```
[scheme:][//[userinfo@]host][/]path[?query][#fragment]
```

- `scheme`：协议名，常见的有`http/https/ftp`；
- `userInfo`：若有，则表示用户信息，如用户名和密码可写作`ls:password`；
- `host`：表示主机域名或地址，和一个可选的端口信息。若端口未指定，则默认为 80。例如`www.example.com`，`www.example.com:8080`，`127.0.0.1:8080`；
- `path`：资源在主机上的路径，以`/`分隔，如`/posts`；
- `query`：可选的查询字符串，客户端传输过来的键值对参数，键值直接用`=`，多个键值对之间用`&`连接，如`page=1&count=10`；
- `fragment`：片段，又叫锚点。表示一个页面中的位置信息。由浏览器发起的请求 URL 中，通常没有这部分信息。但是可以通过`ajax`等代码的方式发送这个数据；

我们来看一个完整的 URL：

```
https://ls:password@www.lianshiclass.com/posts?page=1&count=10#fmt
```

Go 中的 URL 结构定义在`net/url`包中：

```
// net/url/url.go
type URL struct {
    Scheme      string
    Opaque      string
    User        *Userinfo
    Host        string
    Path        string
    RawPath     string
    RawQuery    string
    Fragment    string
}
```

可以通过请求对象中的`URL`字段获取这些信息。接下来，我们编写一个程序来具体看看（使用上一篇文章讲的 Web 程序基本结构，只需要增加处理器函数和注册即可）：

```go
func urlHandler(w http.ResponseWriter, r *http.Request) {
    URL := r.URL

    fmt.Fprintf(w, "Scheme: %s\n", URL.Scheme)
    fmt.Fprintf(w, "Host: %s\n", URL.Host)
    fmt.Fprintf(w, "Path: %s\n", URL.Path)
    fmt.Fprintf(w, "RawPath: %s\n", URL.RawPath)
    fmt.Fprintf(w, "RawQuery: %s\n", URL.RawQuery)
    fmt.Fprintf(w, "Fragment: %s\n", URL.Fragment)
}

// 注册
mux.HandleFunc("/url", urlHandler)
```

运行服务器，通过浏览器访问`localhost:8080/url/posts?page=1&count=10#main`：

```
Scheme:
Host:
Path: /url/posts
RawPath:
RawQuery: page=1&count=10
Fragment:
```

为什么会出现空字段？注意到源码`Request`结构中`URL`字段上有一段注释：

```go
// URL specifies either the URI being requested (for server
// requests) or the URL to access (for client requests).
//
// For server requests, the URL is parsed from the URI
// supplied on the Request-Line as stored in RequestURI.  For
// most requests, fields other than Path and RawQuery will be
// empty. (See RFC 7230, Section 5.3)
//
// For client requests, the URL's Host specifies the server to
// connect to, while the Request's Host field optionally
// specifies the Host header value to send in the HTTP
// request.
```

大意是作为服务器收到的请求时，`URL`中除了`Path`和`RawQuery`，其它字段大多为空。

我们还可以通过`URL`结构得到一个 URL 字符串：

```
URL := &net.URL {
    Scheme:     "http",
    Host:       "example.com",
    Path:       "/posts",
    RawQuery:   "page=1&count=10",
    Fragment:   "main",
}
fmt.Println(URL.String())
```

上面程序运行输出字符串：

```
http://example.com/posts?page=1&count=10#main
```

Proto

`Proto`表示 HTTP 协议版本，如`HTTP/1.1`，`ProtoMajor`表示大版本，`ProtoMinor`表示小版本。

```
func protoFunc(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Proto: %s\n", r.Proto)
    fmt.Fprintf(w, "ProtoMajor: %d\n", r.ProtoMajor)
    fmt.Fprintf(w, "ProtoMinor: %d\n", r.ProtoMinor)
}

mux.HandleFunc("/proto", protoFunc)
```

启动服务器，浏览器请求`localhost:8080`返回：

```
Proto: HTTP/1.1
ProtoMajor: 1
ProtoMinor: 1
```

Header

`Header`中存放的客户端发送过来的首部信息，键-值对的形式。`Header`类型底层其实是`map[string][]string`：

```
// src/net/http/header.go
type Header map[string][]string
```

每个首部的键和值都是字符串，可以设置多个相同的键。注意到`Header`值为`[]string`类型，存放相同的键的多个值。浏览器发起 HTTP 请求的时候，会自动添加一些首部。我们编写一个程序来看看：

```
func headerHandler(w http.ResponseWriter, r *http.Request) {
    for key, value := range r.Header {
        fmt.Fprintf(w, "%s: %v\n", key, value)
    }
}

mux.HandleFunc("/header", headerHandler)
```

启动服务器，浏览器请求`localhost:8080/header`返回：

```
Accept-Enreading: [gzip, deflate, br]
Sec-Fetch-Site: [none]
Sec-Fetch-Mode: [navigate]
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/79.0.1904.108 Safari/537.36]
Sec-Fetch-User: [?1]
Accept:
[text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/
*;q=0.8,application/signed-exchange;v=b3]
Accept-Language: [zh-CN,zh;q=0.9,en-US;q=0.8,en;q=0.7]
```

我使用的是 Chrome 浏览器，不同的浏览器添加的首部不完全相同。

常见的首部有：

- `Accept`：客户端想要服务器发送的内容类型；
- `Accept-Charset`：表示客户端能接受的字符编码；
- `Content-Length`：请求主体的**字节**长度，一般在 POST/PUT 请求中较多；
- `Content-Type`：当包含请求主体的时候，这个首部用于记录主体内容的类型。在发送 POST 或 PUT 请求时，内容的类型默认为`x-www-form-urlecoded`。但是在上传文件时，应该设置类型为`multipart/form-data`。
- `User-Agent`：用于描述发起请求的客户端信息，如什么浏览器。

Content-Length/Body

`Content-Length`表示请求体的**字节**长度，请求体的内容可以从`Body`字段中读取。细心的朋友可能发现了`Body`字段是一个`io.ReadCloser`接口。**在读取之后要关闭它，否则会有资源泄露。可以使用`defer`简化代码编写。**

```
func bodyHandler(w http.ResponseWriter, r *http.Request) {
    data := make([]byte, r.ContentLength)
    r.Body.Read(data) // 忽略错误处理
    defer r.Body.Close()

    fmt.Fprintln(w, string(data))
}

mux.HandleFunc("/body", bodyHandler)
```

上面代码将客户端传来的请求体内容回传给客户端。还可以使用`io/ioutil`包简化读取操作：

```
data, _ := ioutil.ReadAll(r.Body)
```

直接在浏览器中输入 URL 发起的是`GET`请求，无法携带请求体。有很多种方式可以发起带请求体的请求，下面介绍两种：

使用表单

通过 HTML 的表单我们可以向服务器发送 POST 请求，将表单中的内容作为请求体发送。

```
func indexHandler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprint(w, `
<html>
    <head>
        <title>Go Web</title>
    </head>
    <body>
        <form method="post" action="/body">
            <label for="username">用户名: </label>
            <input type="text" id="username" name="username">
            <label for="email">邮箱: </label>
            <input type="text" id="email" name="email">
            <button type="submit">提交</button>
        </form>
    </body>
</html>
`)
}

mux.HandleFunc("/", indexHandler)
```

在 HTML 中使用`form`来显示一个表单。点击提交按钮后，浏览器会发送一个 POST 请求到路径`/body`上，将用户名和邮箱作为请求包体。

启动服务器，进入主页`localhost:8080/`，显示表单。填写完成后，点击提交。浏览器向服务器发送 POST 请求，URL 为`/body`，`bodyHandler`处理完成后将包体回传给客户端。

上面的数据使用了`x-www-form-urlencoded`编码，这是表单的默认编码。

调试工具

- Postman
- Postwoman
- Paw

获取请求参数

上面我们分析了 Go 中 HTTP 请求的常见字段。在实际开发中，客户端通常需要在请求中传递一些参数。参数传递的方式一般有两种方式：

- URL 中的键值对，又叫查询字符串，即`query string`；
- 表单。

URL 键值对

前文中介绍 URL 的一般格式时提到过，URL 的后面可以跟一个可选的查询字符串，以`?`与路径分隔，形如`key1=value1&key2=value2`。

URL 结构中有一个`RawQuery`字段。这个字段就是查询字符串。

```
func queryHandler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintln(w, r.URL.RawQuery)
}

mux.HandleFunc("/query", queryHandler)
```

如果我们以`localhost:8080/query?name=ls&age=20`请求，查询字符串`name=ls&age=20`会传回客户端。

表单

表单狭义上说是通过表单发送请求，广义上说可以将数据放在请求体中发送到服务器。编写一个 HTML 页面，通过页面表单发送 HTTP 请求：

```
<html>
    <head>
        <title>Go Web</title>
    </head>

    <body>
        <form action="/form?lang=cpp&name=ls" method="post"
enctype="application/x-www-form-urlencoded">
            <label>Form:</label>
```

```
            <input type="text" name="lang" />
            <input type="text" name="age" />
            <button type="submit">提交</button>
        </form>
    </body>
</html>
```

- `action`表示提交表单时请求的 URL，`method`表示请求的方法。**如果使用`GET`请求，由于`GET`方法没有请求体，参数将会拼接到 URL 尾部**；
- `enctype`指定请求体的编码方式，默认为`application/x-www-form-urlencoded`。如果需要发送文件，必须指定为`multipart/form-data`；

`urlencoded`编码：RFC 3986 中定义了 URL 中的保留字以及非保留字，所有保留字符都需要进行 URL 编码。URL 编码会把字符转换成它在 ASCII 编码中对应的字节值，接着把这个字节值表示为一个两位长的十六进制数字，最后在这个数字前面加上一个百分号（%）。例如空格的 ASCII 编码为 32，十六进制为 20，故 URL 编码为`%20`。

`Form`字段

使用`x-www-form-urlencoded`编码的请求体，在处理时首先调用请求的`ParseForm`方法解析，然后从`Form`字段中取数据：

```
func formHandler(w http.ResponseWriter, r *http.Request) {
    r.ParseForm()
    fmt.Fprintln(w, r.Form)
}

mux.HandleFunc("/form", formHandler)
```

`Form`字段的类型`url.Values`底层实际上是`map[string][]string`。调用`ParseForm`方法之后，可以使用`url.Values`的方法操作数据。

使用`ParseForm`还能解析查询字符串，将上面的表单改为：

```
<html>
    <head>
        <title>Go Web</title>
    </head>

    <body>
        <form action="/form?lang=cpp&name=ls" method="post"
enctype="application/x-www-form-urlencoded">
            <label>Form:</label>
            <input type="text" name="lang" />
            <input type="text" name="age" />
            <button type="submit">提交</button>
```

```
        </form>
    </body>
</html>
```

查询字符串中的键值对和表单中解析处理的合并到一起了。同一个键下，表单值总是排在前面，如`[golang cpp]`。

`PostForm`字段

如果一个请求，同时有 URL 键值对和表单数据，而用户只想获取表单数据，可以使用`PostForm`字段。使用`PostForm`只会返回表单数据，不包括 URL 键值。

`MultipartForm`字段

如果要处理上传的文件，那么就必须使用`multipart/form-data`编码。与之前的`Form/PostForm`类似，处理`multipart/form-data`编码的请求时，也需要先解析后使用。只不过使用的方法不同，解析使用`ParseMultipartForm`，之后从`MultipartForm`字段取值。

```
<form action="/multipartform?lang=cpp&name=dj" method="post"
enctype="multipart/form-data">
  <label>MultipartForm:</label>
    <input type="text" name="lang" />
    <input type="text" name="age" />
    <input type="file" name="uploaded" />
    <button type="submit">提交</button>
</form>
```

```
func multipartFormHandler(w http.ResponseWriter, r *http.Request) {
    r.ParseMultipartForm(1024)
    fmt.Fprintln(w, r.MultipartForm)

    fileHeader := r.MultipartForm.File["uploaded"][0]
  file, err := fileHeader.Open()
  if err != nil {
    fmt.Println("Open failed: ", err)
    return
  }

  data, err := ioutil.ReadAll(file)
  if err == nil {
    fmt.Fprintln(w, string(data))
  }
}

mux.HandleFunc("/multipartform", multipartFormHandler)
```

`MultipartForm`包含两个`map`类型的字段，一个表示表单键值对，另一个为上传的文件信息。

使用表单中文件控件名获取`MultipartForm.File`得到通过该控件上传的文件，可以是多个。得到的是`multipart.FileHeader`类型，通过该类型可以获取文件的各个属性。

需要注意的是，这种方式用来处理文件。为了安全，`ParseMultipartForm`方法需要传一个参数，表示最大使用内存，避免上传的文件占用空间过大。

`FormValue/PostFormValue`

为了方便地获取值，`net/http`包提供了`FormValue/PostFormValue`方法。它们在需要时会自动调用`ParseForm/ParseMultipartForm`方法。

`FormValue`方法返回请求的`Form`字段中指定键的值。**如果同一个键对应多个值，那么返回第一个**。如果需要获取全部值，直接使用`Form`字段。下面代码将返回`hello`对应的第一个值：

```
fmt.Fprintln(w, r.FormValue("hello"))
```

`PostFormValue`方法返回请求的`PostForm`字段中指定键的值。**如果同一个键对应多个值，那么返回第一个**。如果需要获取全部值，直接使用`PostForm`字段

注意： 当编码被指定为`multipart/form-data`时，`FormValue/PostFormValue`将不会返回任何值，它们读取的是`Form/PostForm`字段，而`ParseMultipartForm`将数据写入`MultipartForm`字段。

检查点

> 其他格式：
>
> 通过 AJAX 之类的技术可以发送其它格式的数据，例如`application/json`等。这种情况下：
>
> - 首先通过首部`Content-Type`来获知具体是什么格式；
> - 通过`r.Body`读取字节流；
> - 解码使用。

4.http 响应（50分钟）

接下来是如何响应客户端的请求。最简单的方式是通过`http.ResponseWriter`发送字符串给客户端。但是这种方式仅限于发送字符串。

`ResponseWriter`

```
func (w http.ResponseWriter, r *http.Request)
```

这里的`ResponseWriter`其实是定义在`net/http`包中的一个接口：

```
// src/net/http/
type ReponseWriter interface {
    Header() Header
    Write([]byte) (int, error)
    WriteHeader(statusCode int)
}
```

我们响应客户端请求都是通过该接口的 3 个方法进行的。例如之前`fmt.Fprintln(w, "Hello, Web")`其实底层调用了`Write`方法。

收到请求后，多路复用器会自动创建一个`http.response`对象，它实现了`http.ResponseWriter`接口，然后将该对象和请求对象作为参数传给处理器。那为什么请求对象使用的时结构指针`*http.Request`，而响应要使用接口呢?

实际上，请求对象使用指针是为了能在处理逻辑中方便地获取请求信息。而响应使用接口来操作，底层也是对象指针，可以保存修改。

接口`ResponseWriter`有 3 个方法：

- `Write`；
- `WriteHeader`；
- `Header`。

`Write`方法

由于接口`ResponseWriter`拥有方法`Write([]byte) (int, error)`，所以实现了`ResponseWriter`接口的结构也实现了`io.Writer`接口：

```
// src/io/io.go
type Writer interface {
  Write(p []byte) (n int, err error)
}
```

这也是为什么`http.ResponseWriter`类型的变量`w`能在下面代码中使用的原因（`fmt.Fprintln`的第一个参数接收一个`io.Writer`接口）：

```
fmt.Fprintln(w, "Hello World")
```

我们也可以直接调用`Write`方法来向响应中写入数据：

```
func writeHandler(w http.ResponseWriter, r *http.Request) {
    str := `<html>
<head><title>Go Web</title></head>
<body><h1>直接使用 Write 方法<h1></body>
</html>`
    w.Write([]byte(str))
}

mux.HandleFunc("/write", writeHandler)
```

curl 工具

工具`curl`可以测试我们的 Web 应用。由于浏览器只会展示响应中主体的内容，其它元信息需要进行一些操作才能查看，不够直观。`curl`是一个 Linux 命令行程序，可用来发起 HTTP 请求，功能非常强大，如设置首部/请求体，展示响应首部等。

通常 Linux 系统会自带`curl`命令。简单介绍几种 Windows 上安装`curl`的方式。

- 直接在curl官网下载可执行程序，下载完成后放在`PATH`目录中即可在`Cmd`或`Powershell`界面中使用；
- Windows 提供了一个软件包管理工具chocolatey，类似mac上的Homebrew，可以安装/更新/删除 Windows 软件。安装`chocolatey`后，直接在`Cmd`或`Powershell`界面执行以下命令即可安装`curl`，也比较方便：

```
choco install curl
```

启动服务器，使用下面命令测试`Write`方法：

```
curl -i localhost:8080/write
```

选项`-i`的作用是显示响应首部。该命令返回：

```
HTTP/1.1 200 OK
Date:  Wed, 1 Jan 2020 13:36:32 GMT
Content-Length: 113
Content-Type: text/html; charset=utf-8

<html>
<head><title>Go Web</title></head>
<body><h1>直接使用 Write 方法<h1></body>
</html>
```

更多的响应内容，可以通过Chrome的开发者工具查看。

注意，我们没有设置内容类型，但是返回的首部中有`Content-Type: text/html; charset=utf-8`，说明`net/http`会自动推断。`net/http`包是通过读取响应体中前面的若干个字节来推断的，并不是百分百准确的。

如何设置状态码和响应内容的类型呢？这就是`WriteHeader`和`Header()`两个方法的作用。

`WriteHeader`方法

`WriteHeader`方法的名字带有一点误导性，它并不能用于设置响应首部。`WriteHeader`接收一个整数，并将这个整数作为 HTTP 响应的状态码返回。调用这个返回之后，可以继续对`ResponseWriter`进行写入，但是**不能对响应的首部进行任何修改操作**。如果用户在调用`Write`方法之前没有执行过`WriteHeader`方法，那么程序默认会使用 200 作为响应的状态码。

如果，我们定义了一个 API，还未定义其实现。那么请求这个 API 时，可以返回一个 501 Not Implemented 作为状态码。

```
func writeHeaderHandler(w http.ResponseWriter, r *http.Request) {
    w.WriteHeader(501)
    fmt.Fprintln(w, "This API not implemented!!!")
}

mux.HandleFunc("/writeheader", writeHeaderHandler)
```

使用`curl`来测试刚刚编写的处理器：

```
curl -i localhost:8080/writeheader
```

返回：

```
HTTP/1.1 501 Not Implemented
Date:  Wed, 1 Jan 2020 14:15:16 GMT
Content-Length: 28
Content-Type: text/plain; charset=utf-8

This API not implemented!!!
```

`Header`方法

`Header`方法其实返回的是一个`http.Header`类型，该类型的底层类型为`map[string][]string`：

```
// src/net/http/header.go
type Header map[string][]string
```

类型`Header`定义了 CRUD 方法，可以通过这些方法操作首部。

```
func headerHandler(w http.ResponseWriter, r *http.Request) {
    w.Header().Set("Location", "http://baidu.com")
    w.WriteHeader(302)
}
```

通过第一篇文章我们知道 302 表示重定向，浏览器收到该状态码时会再发起一个请求到首部中`Location`指向的地址。使用`curl`测试：

```
curl -i localhost:8080/header
```

返回：

```
HTTP/1.1 302 Found
Location: http://baidu.com
Date:  Wed, 1 Jan 2020 14:17:49 GMT
Content-Length: 0
```

如何在浏览器中打开`localhost:8080/header`，网页会重定向到百度首页。

接下来，我们看看如何设置自定义的内容类型。通过`Header.Set`方法设置响应的首部`Contet-Type`即可。我们编写一个返回 JSON 数据的处理器：

```
type User struct {
    FirstName    string      `json:"first_name"`
    LastName     string      `json:"last_name"`
    Age          int         `json:"age"`
    Hobbies      []string    `json:"hobbies"`
}

func jsonHandler(w http.ResponseWriter, r *http.Request) {
    w.Header().Set("Content-Type", "application/json")
    u := &User {
        FirstName:  "ls",
        LastName:   "ls",
        Age:        18,
        Hobbies:    []string{"reading", "learning"},
    }
    data, _ := json.Marshal(u)
    w.Write(data)
}

mux.HandleFunc("/json", jsonHandler)
```

通过`curl`发送请求：

```
curl -i localhost:8080/json
```

返回：

```
HTTP/1.1 200 OK
Content-Type: application/json
Date:  Wed, 1 Jan 2020 14:31:03 GMT
Content-Length: 78

{"first_name":"ls","last_name":"ds","age":18,"hobbies":["reading","learning"]}
```

可以看到响应首部中类型`Content-Type`被设置成了`application/json`。类似的格式还有xml（`application/xml`）/pdf（`application/pdf`）/png（`image/png`）等等。

cookie

cookie 的出现是为了解决 HTTP 协议的无状态性的。客户端通过 HTTP 协议与服务器通信，多次请求之间无法记录状态。服务器可以在响应中设置 cookie，客户端保存这些 cookie。然后每次请求时都带上这些 cookie，服务器就可以通过这些 cookie 记录状态，辨别用户身份等。

> 整个计算机行业的收入都建立在 cookie 机制之上，广告领域更是如此。

上面的说法虽然有些夸张，但是可见 cookie 的重要性。

我们知道广告是互联网最常见的盈利方式。其中有一个很厉害的广告模式，叫做**联盟广告**。最常见的就是，刚刚在百度上搜索了某个关键字，然后打开淘宝或京东后发现相关的商品已经被推荐到首页或边栏了。这是由于这些网站组成了广告联盟，只要加入它们，就可以共享用户浏览器的 cookie 数据。

Go 中 cookie 使用`http.Cookie`结构表示，在`net/http`包中定义：

```
// src/net/http/cookie.go
type Cookie struct {
  Name         string
  Value        string
  Path         string
  Domain       string
  Expires      time.Time
  RawExpires   string
  MaxAge       int
  Secure       bool
  HttpOnly     bool
  SameSite     SameSite
  Raw          string
  Unparsed     []string
}
```

- `Name/Value`：cookie 的键值对，都是字符串类型；
- 没有设置`Expires`字段的 cookie 被称为**会话 cookie** 或**临时 cookie**，这种 cookie 在浏览器关闭时就会自动删除。设置了`Expires`字段的 cookie 称为**持久 cookie**，这种 cookie 会一直存在，直到指定的时间来临或手动删除；
- `HttpOnly`字段设置为`true`时，该 cookie 只能通过 HTTP 访问，不能使用其它方式操作，如 JavaScript。提高安全性；

注意：

> `Expires`和`MaxAge`都可以用于设置 cookie 的过期时间。`Expires`字段设置的是 cookie 在什么**时间点**过期，而`MaxAge`字段表示 cookie 自创建之后能够存活多少秒。虽然 HTTP 1.1 中废弃了`Expires`，推荐使用`MaxAge`代替。但是几乎所有的浏览器都仍然支持`Expires`；而且，微软的 IE6/IE7/IE8 都不支持`MaxAge`。所以为了更好的可移植性，可以只使用`Expires`或同时使用这两个字段。

cookie 需要通过响应的首部发送给客户端。浏览器收到`Set-Cookie`首部时，会将其中的值解析成 cookie 格式保存在浏览器中。下面我们来具体看看如何设置 cookie：

```
func setCookie(w http.ResponseWriter, r *http.Request) {
    c1 := &http.Cookie {
        Name:       "name",
        Value:      "lianshi",
        HttpOnly:   true,
    }
    c2 := &http.Cookie {
        Name:       "age",
        Value:      18,
        HttpOnly:   true,
    }
    w.Header().Set("Set-Cookie", c1.String())
    w.Header().Add("Set-Cookie", c2.String())
}

mux.HandleFunc("/set_cookie", setCookie)
```

运行程序，打开浏览器输入`localhost:8080/set_cookie`，在浏览器开发者工具，切换到 Application（应用）标签，查看 cookie。在左侧 Cookies 下点击测试的 URL，右侧即可显示我们刚刚设置的 cookie：

上面构造 cookie 的代码中，有几点需要注意：

> - 首部名称为`Set-Cookie`；
> - 首部的值需要是字符串，所以调用了`Cookie`类型的`String`方法将其转为字符串再设置；
> - 设置第一个 cookie 调用`Header`类型的`Set`方法，添加第二个 cookie 时调用`Add`方法。`Set`会将同名的键覆盖掉。如果第二个也调用`Set`方法，那么第一个 cookie 将会被覆盖。

为了使用的便捷，`net/http`包还提供了`SetCookie`方法。用法如下：

```
func setCookie2(w http.ResponseWriter, r *http.Request) {
    c1 := &http.Cookie {
        Name:       "name",
        Value:      "lianshi",
        HttpOnly:   true,
    }
```

```
    c2 := &http.Cookie {
        Name:       "age",
        Value:      "18",
        HttpOnly:   true,
    }
    http.SetCookie(w, c1)
    http.SetCookie(w, c2)
}

mux.HandleFunc("/set_cookie2", setCookie2)
```

如果收到的响应中有 cookie 信息，浏览器会将这些 cookie 保存下来。只有没有过期，在向同一个主机发送请求时都会带上这些 cookie。在服务端，我们可以从请求的`Header`字段读取`Cookie`属性来获得 cookie：

```
func getCookie(w http.ResponseWriter, r *http.Request) {
  fmt.Fprintln(w, "Host:", r.Host)
  fmt.Fprintln(w, "Cookies:", r.Header["Cookie"])
}

mux.HandleFunc("/get_cookie", getCookie)
```

第一次启动服务器，请求`localhost:8080/get_cookie`时，没有 cookie 信息。

先请求一次`localhost:8080/set_cookie`，然后再次请求`localhost:8080/get_cookie`，浏览器就将 cookie 传过来了。

`r.Header["Cookie"]`返回一个切片，这个切片又包含了一个字符串，而这个字符串又包含了客户端发送的任意多个 cookie。如果想要取得单个键值对格式的 cookie，就需要解析这个字符串。 为此，`net/http`包在`http.Request`上提供了一些方法使我们更容易地获取 cookie：

```
func getCookie2(w http.ResponseWriter, r *http.Request) {
    name, err := r.Cookie("name")
    if err != nil {
        fmt.Fprintln(w, "cannot get cookie of name")
    }

    cookies := r.Cookies()
    fmt.Fprintln(w, c1)
    fmt.Fprintln(w, cookies)
}

mux.HandleFunc("/get_cookies", getCookies2)
```

- `Cookie`方法返回以传入参数为键的 cookie，如果该 cookie 不存在，则返回一个错误；
- `Cookies`方法返回客户端传过来的所有 cookie。

有一点需要注意，cookie 是与主机名绑定的，不考虑端口。我们上面查看 cookie 的图中有一列`Domain`表示的就是主机名。可以这样来验证一下，创建两个服务器，一个绑定在 8080 端口，一个绑定在 8081 端口，先请求`localhost:8080/set_cookie`设置 cookie，然后请求`localhost:8081/get_cookie`：

```
func main() {
  mux1 := http.NewServeMux()
  mux1.HandleFunc("/set_cookie", setCookie)
  mux1.HandleFunc("/get_cookie", getCookie)

  server1 := &http.Server{
    Addr:    ":8080",
    Handler: mux1,
  }

  mux2 := http.NewServeMux()
  mux2.HandleFunc("/get_cookie", getCookie)

  server2 := &http.Server {
    Addr:    ":8081",
    Handler: mux2,
  }

  wg := sync.WaitGroup{}
  wg.Add(2)

  go func () {
    defer wg.Done()

    if err := server1.ListenAndServe(); err != nil {
      log.Fatal(err)
    }
  }()


  go func() {
    defer wg.Done()

    if err := server2.ListenAndServe(); err != nil {
      log.Fatal(err)
    }
  }()

  wg.Wait()
}
```

发送给端口 8081 的请求同样可以获取 cookie。

上面代码中，不能直接在主 goroutine 中依次`ListenAndServe`两个服务器。因为`ListenAndServe`只有在出错或关闭时才会返回。在此之前，第二个服务器永远得不到机会运行。所以，我创建两个 goroutine 各自运行一个服务器，并且使用`sync.WaitGroup`来同步。否则，主 goroutine 运行结束之后，整个程序就退出了。

五、拓展点（10分钟）

> 梳理一下net/http代码的执行流程
>
> - 首先调用Http.HandleFunc
>
> 按顺序做了几件事：
>
> 1 调用了DefaultServeMux的HandleFunc
>
> 2 调用了DefaultServeMux的Handle
>
> 3 往DefaultServeMux的map[string]muxEntry中增加对应的handler和路由规则
> - 其次调用http.ListenAndServe(":8080", nil)
>
> 按顺序做了几件事情：
>
> 1 实例化Server
>
> 2 调用Server的ListenAndServe()
>
> 3 调用net.Listen("tcp", addr)监听端口
>
> 4 启动一个for循环，在循环体中Accept请求
>
> 5 对每个请求实例化一个Conn，并且开启一个goroutine为这个请求进行服务go c.serve()
>
> 6 读取每个请求的内容w, err := c.readRequest()
>
> 7 判断handler是否为空，如果没有设置handler（这个例子就没有设置handler），handler就设置为DefaultServeMux
>
> 8 调用handler的ServeHttp
>
> 9 在这个例子中，下面就进入到DefaultServeMux.ServeHttp
>
> 10 根据request选择handler，并且进入到这个handler的ServeHTTP

六、总结（5分钟）

> 说明：
>
> 回顾本堂课所有知识点；

Go Web 的基本形式如下：

```go
package main

import (
    "fmt"
    "log"
    "net/http"
)

func helloHandler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Hello, World")
}

type greetingHandler struct {
    Name string
}

func (h greetingHandler) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Hello, %s", h.Name)
}

func main() {
    mux := http.NewServeMux()
    // 注册处理器函数
    mux.HandleFunc("/hello", helloHandler)

    // 注册处理器
    mux.Handle("/greeting/golang", greetingHandler{Name: "Golang"})

    server := &http.Server {
        Addr:       ":8080",
        Handler:    mux,
    }
    if err := server.ListenAndServe(); err != nil {
        log.Fatal(err)
    }
}
```