练识课堂 -- GoWeb编程基础（一）

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（10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**。

常见Web服务简介

Web服务背后的**关键在于平台的无关性**，你可以运行你的服务在Linux/Unix系统，可以与其他Window的浏览器程序或者iOS/Android App交互。主流的Web服务有：REST、SOAP。

- REST：是基于HTTP协议的一个补充，他的每一次请求都是一个HTTP请求，然后根据不同的method来处理不同的逻辑，很多Web开发者都熟悉HTTP协议，所以学习REST是一件比较容易的事情。
- SOAP：W3C在跨网络信息传递和远程计算机函数调用方面的一个标准。但是SOAP非常复杂，其完整的规范篇幅很长，而且内容仍然在增加。

除此之外，非http协议的Web服务还有很多：

- Socket：指TCP/IP网络环境中的两个连接端，是API提供的一组接口用于组织数据以符合协议。
- Websocket：可以看作HTTP的降级或修改，使得浏览器能够直接进行TCP连接，减少HTTP长连接方式的资源浪费。
- RPC：远程过程调用协议，一种进程间通信方式，其调用包含了传输协议（可以是HTTP）和编码协议，允许一个程序调用另一个地址空间的过程或函数。RPC是现阶段替换SOAP标准的最佳实现。
- RMI：远程方法调用，在客户端Java虚拟机（JVM）上的对象像调用本地对象一样调用服务器端JVM上对象的方法。
- ……

（以上是混淆了web服务的一些特性，做出简单的分类，有一篇文档详细介绍各种Web服务和名词）

HTTP

HTTP 协议是整个互联网的基石。我们每天浏览网页都在使用 HTTP。现在很多 APP 也都在内部使用 HTTP 与服务器交互。所以学习 Web 编程，HTTP 协议是必须要掌握的。

HTTP 是一个无状态的，基于文本的协议。它灵活，稳定，强大。自 1991 年发布以来只进行了几次修订。下面是 HTTP 发展简史：

- 1991 年，HTTP 的第一个版本 0.9 由 Tim Berners-Lee 创建。最初只有一个方法 GET，而且规定服务器返回的只能是 HTML 格式的数据。
- 1996 年，HTTP 1.0 发布，支持了 POST 和 HEAD 方法。
- 1999 年，HTTP 1.1 发布，添加了 PUT/DELETE/OPTIONS/TRACE/CONNECT 这 5 个方法，**并允许开发者自行添加更多方法**。
- 2015 年，HTTP 2.0 发布，为提升性能做出了不少修改，如采用二进制格式，完全多路复用。

HTTP 是一种请求——响应模式的协议，所有操作以一个请求开始，以一个响应结束。

HTTP 请求

HTTP 请求的格式非常简单。一个请求由以下 4 个部分组成：

- 请求行（request-line）；
- 零个或多个首部（header）；
- 一个空行；
- 一个可选的报文主体（body）。

第一个重要的部分为请求行，其格式如下：

```
Method Path Version
```

- `Method`：请求的方法，表示对资源进行的操作。常用的方法有`GET/POST/PUT/DELETE`；
- `Path`：请求资源的路径，如`/user/info.html`；
- `Version`：即 HTTP 的版本号，1.1 版本写做`HTTP/1.1`。

第二个部分为请求的首部，每个首部占一行。首部使用由冒号（`:`）分隔的键值对表示，如`Content-Type: x-www-form-urlencoded`。

第三个部分为一个空行。注意，**即使没有首部，后面的空行也不能省略**。

最后为一个可选的报文主体。如果有主体，服务器会根据首部中`Content-Type`指定的格式来解析这部分内容。

HTTP 响应

HTTP 响应的格式与请求非常相似。一个响应由以下 4 个部分组成：

- 响应行（response line）；
- 零个或多个首部（header）；
- 一个空行；
- 一个可选的报文主体（body）。

响应行的格式为：

```
StatusCode Description
```

- `StatusCode`：状态码，表示请求状态；
- `Description`：对状态码的简短描述。

响应的首部与主体和请求的一样，这里就不多说了。

这里简单介绍一下状态码。HTTP 将状态码分为了 5 大类，`1XX/2XX/3XX/4XX/5XX`。

- 1xx

  情报状态码，又叫做信息状态码。服务器通过这类状态码告知客户端，自己已经收到了客户端发送的请求。几个常见的状态码如下：

  - `100 Continue`：表示服务器到目前为止收到的内容都正常，客户端应该继续请求。如果已

经完成，则忽略它。
    - `101 Switching Protocol`：这个状态码是响应客户端的`Upgrade`首部发送的，并且指示服务器也正在切换的协议。如客户端请求切换协议，服务器将协议切换至 Websocket，就会发送该状态码给客户端，并且在`Upgrade`首部中填上 Websocket。
- 2xx

  成功状态码。表示服务器已经收到了客户端的请求，并成功对请求进行了处理。几个常见的状态码如下：
    - `200 OK`：这最常见的状态码了，表示请求成功。
    - `201 Created`：请求已成功，并因此创建了一个新的资源。
- 3xx

  重定向状态码。服务器收到了请求，但是为了完整地处理该请求，客户端还需要执行指定的动作。一般用于 URL 重定向。几个常见的状态码如下：
    - `300 Multiple Choice`：被请求的资源有一系列可供选择的回馈信息，每个都有自己特定的地址。用户或浏览器可自行选择一个地址进行重定向。
    - `302 Moved Permanently`：被请求的资源已经永久移动到新的位置了。
- 4xx

  客户端错误状态码。表示客户端发送的请求中有错误，如格式不对。常见的状态码如下：
    - `404 Not Found`：最常见的状态码了，表示页面不存在。
    - `405 Method Not Allowed`：请求的方法不允许。
- 5xx

  服务器错误状态码。表示服务器由于某些原因无法正确处理请求。常见的状态码如下：
    - `500 Internal Server Error`：服务器遇到了不知道如何处理的情况。
    - `501 Not Implemented`：此请求方法不被服务器支持。

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，~~摘自《Go Web 编程》（需要画一个高清图）~~：

- 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：

```go
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。

```go
// 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。

```go
// 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。

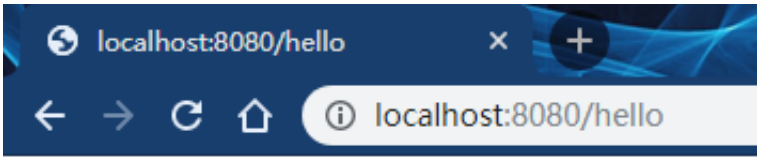
这里有一个注意点，**如果注册的 URL 不是以`/`结尾的，那么它只能精确匹配请求的 URL**。反之，即使请求的 URL 只有**前缀**与被绑定的 URL 相同，`ServeMux`也认为它们是匹配的。

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。

```go
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)
  }
}
```

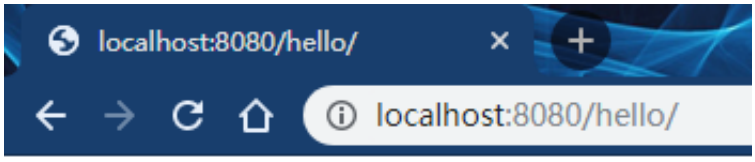
- 浏览器请求`localhost:8080/`将返回`"This is the index page"`，因为`/`精确匹配；

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



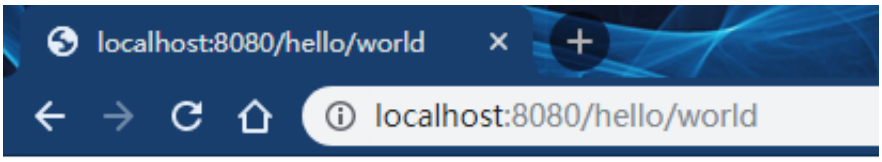
This is the hello page

- 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`/`**；



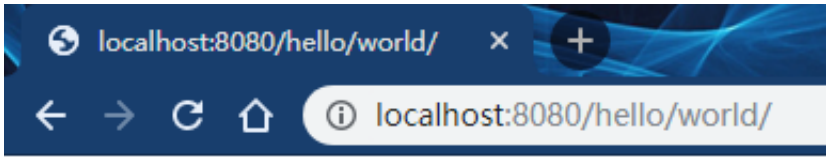
This is the index page

- 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；



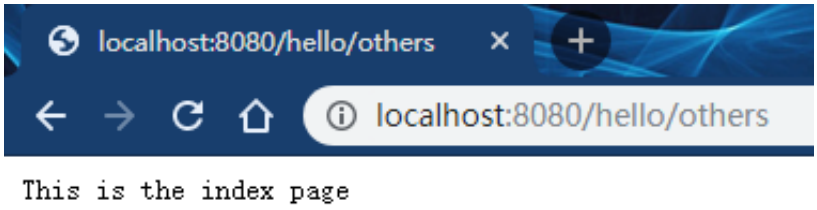
This is the world page

- 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） -> `/`**；



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：

```go
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)
    }
}
```