

第3章 使用 Go 语言实现 HTTP/1.0 客户端

[日] 涩川喜规 · 详解HTTP：协议基础与Go语言实现

第1章介绍了HTTP的如下4个基本元素，以及发送和接收这些基本元素的curl命令的写法。

方法和路径

首部

主体

状态码

本章我们来学习一下如何使用Go语言来编写发送和接收上述内容的代码。



在前面的章节中，curl命令都带有HTTP/1.0 中的运行选项（`--http1.0`），本章会省略该选项。虽然在执行curl命令时会像“HTTP/1.1”这样显示协议版本，但动作与HTTP/1.0并没有太大区别。

3.1 为何使用 Go 语言

Google公司在2007年发布了Go语言。近年来，Go语言得到广泛应用。Go语言的优点体现在以下几个方面。

拥有轻量级语言规范和丰富的标准库

编译速度快，可与动态脚本语言的运行速度匹敌，会进行类型检查

运行速度快，易于实现多核性能，节省内存

易于交叉编译

输出的是单一的二进制文件，易于发布

另外，本书之所以采用 Go 语言来实现 HTTP 代码，也是因为它是一门优秀的教学语言，具体体现在以下几个方面。

语言规范少于其他编程语言，使用其他编程语言的用户也能轻松理解 Go 语言的运行方式，因此 Go 作为描述语言也非常优秀

Go 语言是编译语言，能够检查语法结构和类型，易于发现输入错误

仅使用标准库即可创建访问 HTTP 的程序

实际上，Go 语言也用作各种 Web 服务的 CLI 客户端的实现语言

单看每个特征，我们也能发现 Go 语言相比其他编程语言的优秀之处。例如，Go 语言的运行速度比 C 和 C++ 快，生成的二进制文件也比较小。不过，交叉编译环境搭建起来比较费事，而且在编译和发布时，要把需要用到的库都集中起来创建编译环境，这也很麻烦。拥有包管理器的 Python、Ruby 和 Node.js 等脚本语言无须编译，但不会输出单一的二进制文件，运行速度比 Go 语言慢一点儿，内存消耗量方面也比 Go 语言大一点儿。我们在学生时代都参加过模拟考试，应该知道用自己擅长的学科弥补不擅长的学科是一件很难的事情，只有所有学科都取得好成绩，才能提高偏差值¹。Go 语言就是一种能够完美解决各种问题的语言。

本书不会详细介绍 Go 语言的基础知识，其他编程语言的使用者可通过在线教程“*A Tour of Go*”了解 Go 语言。如今，Go 语言的相关图书不断增多，学习也变得更加方便。大家还可以参考集中介绍了 Go 语法的“Java to Go in-depth tutorial”。

3.2 Go 语言的 API 结构

Go 语言中提供的 HTTP 的客户端 API 大致分为如下 3 类。

功能有限，但易于使用的 API

能使用 Cookie 的、可稍加控制的 API

原生的、可访问所有功能的 API

前两种 API 比较简单，写出来的代码也比较短，但在处理更详细的内容时，就需要使用最后一种 API 了。前两种 API 是基于后一种 API 构建的。我们首先使用比较简单的 API 来实现频繁使用的处理的代码，然后学习如何使用原生 API 来实现更加详细的 HTTP 访问。

3.3 本章的主要内容

第 1 章和第 2 章介绍了 HTTP 的基本结构，以及浏览器发送和接收信息的模式（表 3-1）。下面我们来学习一下如何使用 Go 语言来实现其中几个比较典型的模式。

表 3-1 浏览器发送和接收信息的模式

模式	方法	Go 语言的 API
使用 GET 获取信息	GET	<code>http.Get</code>
获取带查询的信息	GET	<code>http.Get</code>
使用 HEAD 获取首部	HEAD	<code>http.Head</code>
使用 <code>x-www-form-urlencoded</code> 发送表单	POST	<code>http.PostForm</code>
使用 POST 发送文件	POST	<code>http.Post</code>
使用 <code>multipart/form-data</code> 发送文件	POST	<code>http.PostForm</code>
发送和接收 Cookie	GET、HEAD、POST	<code>http.Client</code>
代理	GET、HEAD、POST	<code>http.Client</code>
访问文件系统	GET、HEAD、POST	<code>http.Client</code>
发送任意方法	所有	<code>http.Request</code> 、 <code>http</code>
发送首部	所有	<code>http.Request</code> 、 <code>http</code>

表 3-1 中的前 6 项使用简单的 API，接下来笔者会首先介绍这部分内容，之后介绍用于把 Cookie 和代理设置为有效的 `http.Client` 的使用方法，最后介绍可以访问所有功能的 API，即 `http.Request` 和 `http.Client` 的组合。

很多读者可能使用的并不是 Go 语言，但各种编程语言的库基本上会提供与常用的 GET 和 POST 相关的 API 和专用方法，然后追加提供可以访问所有功能的 API。

3.4 GET 方法的发送及主体、状态码和首部的接收

首先来看最简单的发送 GET 方法的代码，它与下面的 curl 命令是等价的。

 复制代码

```
1 $ curl http://localhost:18888
```

代码清单 3-1 中的代码用于发送 GET 方法，并将响应主体输出到画面上。代码中也包含了错误处理。这是保持良好运行的最低限度的代码。

代码清单 3-1 发送 GET 方法，并将响应主体输出到画面上

 复制代码

```
1 package main          // ①
2
3 import (
4     "io/ioutil"
5     "log"
6     "net/http"
7 )
8
9 func main() {          // ③
10    resp, err := http.Get("http://localhost:18888")
11    if err != nil {      // ④
12        panic(err)
13    }
14    defer resp.Body.Close() // ⑤
15    body, err := ioutil.ReadAll(resp.Body) // ⑥
16    if err != nil {      // ④
17        panic(err)
```

```
18     }
19     log.Println(string(body)) // ⑦
20 }
```

- ① 库之外的源代码必须以 `package main` 开始。
- ② 导入需要用到的包。在 Go 语言中，无法使用文件中未声明的包。
- ③ 所有程序在最开始都会调用 `main` 包（① 中创建的包）中的 `main` 函数。
- ④ Go 语言的错误处理代码。Go 语言的函数会将错误作为返回值返回，因此需要确认返回值是否为 `nil`。这里没有异常处理。`panic` 用来显示错误并结束程序。库擅自结束程序会给使用的应用程序带来麻烦，因此在打包为库时，Go 语言通常会像 `http.Get()` 那样，将错误作为返回值的最终项返回。
- ⑤ 这是后处理的代码。如果加上 `defer`，在跳出 `main` 函数后，就会执行该语句。这是通过套接字读取主体之后的处理。
- ⑥ 将主体内容读取为字节序列。
- ⑦ 将字节序列转换为 UTF-8 字符串并输出到画面上。



关于 ⑤ 处的 `defer`，虽然运行的时间点不一样，但与资源的创建（接收）在相同的作用域内，所以可以利用 `defer` 机制编写结束处理，或者使其自动执行。在 C++ 中，可以通过调用本地的析构函数，使用智能指针（smart pointer）来销毁栈。在 Ruby 中，可以像 `File.open` 方法那样使用代码块。在 Python 中，可以使用 `with` 语法结构。在异常处理中，无论是否发生错误，`finally` 代码块都会被调用。拥有这种 `finally` 代码块的语言也会使用 `defer`。

在 `simpleget` 文件夹中，创建一个名为 `main.go` 的文件（文件名任意），输入前面的代码，然后像下面这样在 `simpleget` 文件夹中进行编译。这里会生成一个与文件夹同名的执行文件，我们试着运行这个执行文件。

复制代码

```
1 $ go build
2 $ ./simpleget
```

在 Windows 中，程序如下运行。

 复制代码

```
1 > go build  
2 > simpleget.exe
```

如果像下面这样启动，则也可以像脚本语言一样，在运行时不关心编译内容。

 复制代码

```
1 $ go run main.go
```

代码清单 3-2 中的代码省略了错误检查，更加简短。

代码清单 3-2 无错误检查的代码

 复制代码

```
1 package main  
2  
3 import (  
4     "io/ioutil"  
5     "log"  
6     "net/http"  
7 )  
8  
9 func main() {  
10     resp, _ := http.Get("http://localhost:18888")  
11     defer resp.Body.Close()  
12     body, _ := ioutil.ReadAll(resp.Body)  
13     log.Println(string(body))  
14 }
```

`resp` 变量中保存的元素是 `http.Response` 类型的对象。该对象中保存着从服务器返回的各种信息。主体保存在成员变量 `Body` 中，`StatusCode` 中使用 `200` 之类的数值来保存

状态码，`Status` 中使用 `"200 OK"` 之类的字符串来保存状态码。将代码清单 3-3 的代码插入到 `resp` 变量的定义后面，就会在控制台上显示状态码。

代码清单 3-3 在控制台上显示状态码

 复制代码

```
1 // 字符串 "200 OK"
2 log.Println("Status:", resp.Status)
3 // 数值 200
4 log.Println("StatusCode:", resp.StatusCode)
```

同样，`resp.Header` 中保存着首部。首部是字符串数组 `map` 类型。下面这行代码用于在控制台上输出响应首部。

输出响应首部

```
log.Println("Headers:", resp.Header)
```

也存在仅获取指定项目的开头元素的 `Get` 方法，如下所示。

获取指定的首部

```
log.Println("Content-Length:", resp.Header.Get("Content-Length"))
```

运行结果如下所示。

 复制代码

```
1 2016/06/08 01:43:38 <html><body>hello</body></html>
2 2016/06/08 01:43:38 Status: 200 OK
3 2016/06/08 01:43:38 StatusCode: 200
4 2016/06/08 01:43:38 Headers: map[Date:[Tue, 07 Jun 2016 16:43:38 GMT] Content-
5 Length:[32] Content-Type:[text/html; charset=utf-8]]
6 2016/06/08 01:43:38 Content-Length: 32
```

`http.Get()` 函数是一种高级 API，即使什么也不指定，也可以用它进行 10 次重定向等，而无须关心与浏览器的动作的差别。

io.Reader

代码清单 3-1 中存在如下代码。

读取整个主体数据，返回字节数组

```
body, err := ioutil.ReadAll(resp.Body) // ⑥
```

在 Go 语言中，数据流的输入和输出会抽象化为 `io.Reader` 接口和 `io.Writer` 接口。文件和套接字等处都会用到这些接口。`http.Response` 类型的变量 `resp` 的成员变量 `Body` 也是一个持有 `io.Reader` 接口的对象。

准备一个字节数组的缓冲区，并调用 `Read()` 方法，即可直接处理 `io.Reader`。

`io.Writer()` 也一样，只需准备一个字节数组，然后调用 `Write()` 方法即可。它们都是原生的，因此实际中如果这样使用，代码行数就会增多。Go 语言提供了既能处理文件又能处理套接字的许多高级功能。

```
func ioutil.ReadAll(reader io.Reader) ([]byte, error)
```

汇总 `io.Reader` 的内容，并将其读入字节数组。

```
func Copy(dst io.Writer, src io.Reader) (written int64, err error)
```

将所有内容从 `io.Reader` 复制到 `io.Writer` 中。

```
bufio.NewReader(reader io.Reader) *bufio.Reader
```

包装 `io.Reader`，添加缓冲结构，创建能够新增许多便捷方法的对象。

```
bytes.Buffer
```

能够读写内存，并且可用作 `io.Reader` 和 `io.Writer`。

```
strings.Reader
```

能够读取字符串，并且可用作 `io.Reader`。

另外，以下函数也很常用。

```
fmt.Fprintf(writer io.Writer, format string, ... variant{})
```

将格式化的字符串写出到 `io.Writer` 中。

Go 语言拥有通过组合使用接口来提高代码抽象度的结构。因为我们需要记住一些库，所以一开始可能感觉比较难，不过熟悉之后，就可以对所有的输入和输出目标执行高级操作了²。

3.5 使用 GET 方法发送查询

接下来介绍如何使用 `GET` 方法发送查询。我们来编写与下面的 curl 命令基本等价的代码。

 复制代码

```
1 $ curl -G --data-urlencode "query=hello world" http://localhost:18888
```

如果查询中没有空格或不可以在 URL 中使用的字符，就可以使用 `--data` 或者其缩写 `-d` 来代替 `--data-urlencode`。`-G` 是 `--get` 的缩写。

代码清单 3-4 是使用 Go 语言实现的代码。

代码清单 3-4 使用 GET 方法发送查询

 复制代码

```
1 package main
2
3 import (
4     "io/ioutil"
5     "log"
6     "net/http"
7     "net/url"
8 )
9
10 func main() {
11     values := url.Values{    // ①
12         "query": {"hello world"}, // ②
13     }
14     // ③
15     resp, _ := http.Get("http://localhost:18888" + "?" + values.Encode())
16     defer resp.Body.Close()
```

```
17     body, _ := ioutil.ReadAll(resp.Body)
18     log.Println(string(body))
19 }
```

- ① 首先，创建查询字符串。查询字符串使用 `url.Values` 类型进行声明，调用 `value.Encode()` 方法来创建。字符串的转义也由该方法执行。
- ② 最后，在 URL 的末尾添加结果字符串。

如下所示，查看服务器端的日志就可以发现，URL 中附加了查询。另外，我们还能得知查询被正确传递，成了字符串数组的映射。

接收查询的服务器端的日志

 复制代码

```
1 URL: "/?query=hello+world"
2 Query: {
3     "query": []string{
4         "hello world",
5     },
6 }
```

3.6 使用 HEAD 方法获取首部

`HEAD` 方法的实现很简单，只要将 `http.Get` 替换为 `http.Head()` 即可。下面的 `curl` 命令也可实现 `HEAD` 方法的处理。

 复制代码

```
1 $ curl --head http://localhost:18888
```

代码清单 3-5 发送 HEAD 方法来获取首部

 复制代码

```
1 package main
```

```
2
3 import (
4     "log"
5     "net/http"
6 )
7
8 func main() {
9     resp, err := http.Head("http://localhost:18888")
10    if err != nil {
11        panic(err)
12    }
13    log.Println("Status:", resp.Status)
14    log.Println("Headers:", resp.Header)
15 }
```

正如 `HEAD` 方法定义的那样，它无法获取主体。如果像 `GET` 方法那样读取主体，虽然不会发生错误，但会得到一个长度为 0 的字节数组。

3.7 使用 `POST` 方法发送 `x-www-form-urlencoded` 形式的表单

笔者最开始介绍了如何使用 `GET` 方法实现与普通的 Web 浏览器相同的操作，然后介绍了如何使用 `GET` 方法发送查询。从本节到 3.9 节，笔者将介绍 HTTP/1.0 中使用 `POST` 方法发送表单的相关内容。本节主要介绍在未改变编码的情况下，如何使用默认的 `x-www-form-urlencoded` 形式发送表单。

我们来回顾一下第 2 章的内容。在使用 `curl` 命令以与 `x-www-form-urlencoded` 相同的形式发送表单时，使用如下命令。

 复制代码

```
1 $ curl -d test=value http://localhost:18888
```

实际的 Go 语言代码如代码清单 3-6 所示。由于该代码仅进行发送，所以访问发送结果的主体等的相关内容请参考其他代码。

代码清单 3-6 发送 `x-www-form-urlencoded` 形式的表单

 复制代码

```
1 package main
2
3 import (
4     "log"
5     "net/http"
6     "net/url"
7 )
8
9 func main() {
10     values := url.Values{
11         "test": {"value"},
12     }
13
14     resp, err := http.PostForm("http://localhost:18888", values)
15     if err != nil {
16         // 发送失败
17         panic(err)
18     }
19     log.Println("Status:", resp.Status)
20 }
```

上面的代码中再次出现了 `url.Values`，但这里并未调用 `url.Values.Encode()` 方法，而是将 `url.Values` 传递给了 `http.PostForm()` 函数。第 2 章开头介绍过使用 `x-www-form-urlencoded` 发送表单的相关内容，其中提到了 URL 编码可分为两类，但如果在 Go 语言中使用这种调用方法，就会执行基于 RFC 3986 的转换。

3.8 使用 POST 方法发送任意主体

使用 `POST` 方法可以将任意内容放入主体中发送。虽然无法使用 HTTP/1.0 的浏览器发送主体，但可以使用 HTTP/1.1 之后出现的 `XMLHttpRequest` 来实现。由于在介绍如何编写发送 `multipart` 表单的代码时需要用到这些内容，所以笔者提前对此进行介绍。

使用 `curl` 命令发送从文件中读取的任意内容的代码如下所示。

 复制代码

```
1 $ curl -T main.go -H "Content-Type: text/plain" http://localhost:18888
```

代码清单 3-7 是使用 Go 语言实现上述处理的代码。

代码清单 3-7 使用 POST 方法发送任意的主体

 复制代码

```
1 package main
2
3 import (
4     "log"
5     "net/http"
6     "os"
7 )
8
9 func main() {
10     file, err := os.Open("main.go")
11     if err != nil {
12         panic(err)
13     }
14     resp, err := http.Post("http://localhost:18888", "text/plain", file)
15     if err != nil {
16         // 发送失败
17         panic(err)
18     }
19     log.Println("Status:", resp.Status)
20 }
```

`http.Post()` 方法的第 2 个参数中指定了 `Content-Type` 首部的内容。另外，发送的内容并不是采用文本形式传递的，而是采用了 `io.Reader` 的形式。`os.Open()` 函数创建的 `os.File` 对象属于 `io.Reader` 接口，因此可以直接传递给 `http.Post()`。

在将程序中创建的文本（而不是文件）传递给 `http.Post` 时，如代码清单 3-8 所示，使用 `bytes.Buffer` 或 `strings.Reader` 将字符串实现为 `io.Reader` 接口。

代码清单 3-8 将字符串实现为 io.Reader 接口

 复制代码

```
1 package main
2
```

```
3 import (
4     "log"
5     "net/http"
6     "strings"
7 )
8
9 func main() {
10    reader := strings.NewReader("文本")
11    resp, err := http.Post("http://localhost:18888", "text/plain", reader)
12    if err != nil {
13        // 发送失败
14        panic(err)
15    }
16    log.Println("Status:", resp.Status)
17 }
```

3.9 使用 multipart/form-data 形式发送文件

在 HTTP/1.0 的 HTML 中，最复杂的发送处理就是使用 `multipart/form-data` 发送文件，相应的 curl 命令如下所示。

```
$ curl -F "name=Michael Jackson" -F "thumbnail=@photo.jpg"
http://localhost:18888
```

该 curl 命令用于发送文本数据和文件。使用 Go 语言实现这一处理的代码如代码清单 3-9 所示。

代码清单 3-9 使用 multipart/form-data 形式发送文件

 复制代码

```

1 package main
2
3 import (
4     "bytes"
5     "io"
6     "log"
7     "mime/multipart"
8     "net/http"
9     "os"
10    )
11
12 func main() {
13     var buffer bytes.Buffer           // ①
14     writer := multipart.NewWriter(&buffer) // ②
15     writer.WriteField("name", "Michael Jackson") // ③
16     // ④
17     fileWriter, err := writer.CreateFormFile("thumbnail", "photo.jpg")
18     if err != nil {
19         panic(err)
20     }
21     readFile, err := os.Open("photo.jpg") // ⑤
22     if err != nil {
23         // 文件读取失败
24         panic(err)
25     }
26     defer readFile.Close()
27     io.Copy(fileWriter, readFile)        // ⑥
28     writer.Close()                      // ⑦
29
30     resp, err := http.Post("http://localhost:18888", writer.FormDataContentType()
31     &buffer)
32     if err != nil {
33         // 发送失败
34         panic(err)
35     }
36     log.Println("Status:", resp.Status)
37 }

```

- ① 声明缓冲区，用来存储 multipart 部分构建后的字节序列。
- ② 创建 `Writer`，用来构建 multipart。
- ③ 文件以外的字段通过 `WriteField()` 方法进行登录。
- ④ ④ ~ ⑥ 是文件的读取操作。首先创建写入单个文件的 `io.Writer`。
- ⑤ 打开文件。
- ⑥ 使用 `io.Copy()`，将文件的所有内容复制到用于写入文件的 `io.Writer` 中。

⑦ 最后关闭 multipart 的 `io.Writer`, 将所有内容写入到缓冲区中。

之后, 像上一节一样使用 `http.Post()` 发送该缓冲区, 就可以发送表单了。

代码行数有点多。最开始在作为 `io.Reader` 的 `bytes.Buffer` 中创建了 multipart 表单的发送内容。创建该内容时使用的是 `multipart.Writer` 对象。当通过该对象写入表单项和文件时, 表单项和文件就会写入到作为 `multipart.NewWriter()` 的参数传递的 `bytes.Buffer` 中。这个 `bytes.Buffer` 既是 `io.Reader`, 又是 `io.Writer`。

`Content-Type` 中需要放入边界字符串。边界字符串是 `multipart.Writer` 对象在内部使用随机数生成的。边界字符串可以通过 `Boundary()` 方法获取, 因此, 我们也可以按照代码清单 3-10 的方法构建 `Content-Type`。`FormDataContentType()` 方法是该代码的“快捷方式”。

代码清单 3-10 与 `FormDataContentType()` 作用相同的代码

 复制代码

```
1 "multipart/form-data; boundary=" + writer.Boundary()
```

对发送的文件设置任意的 MIME 类型

上一节介绍的代码可以用来发送文件, 不过各个文件的 `Content-Type` 实际变成了 `application/octet-stream` 类型。我们可以使用 `textproto.MIMEHeader` 来设置任意的 MIME 类型³。试着对代码清单 3-9 中 ④ ~ ⑥ 的部分进行改写, 将 MIME 类型设置为 `image/jpeg`, 改写的部分如代码清单 3-11 所示。

代码清单 3-11 将 MIME 类型设置为 `image/jpeg`

 复制代码

```
1 import (
2     "net/textproto"
3 )
4 :
```

```
5 part := make(textproto.MIMEHeader)
6 part.Set("Content-Type", "image/jpeg")
7 part.Set("Content-Disposition", `form-data; name="thumbnail"; filename="photo.jpg"
8 fileWriter, err := writer.CreatePart(part)
9 if err != nil {
10     panic(err)
11 }
12 readFile, err := os.Open("photo.jpg")
13 if err != nil {
14     panic(err)
15 }
16 io.Copy(fileWriter, readFile)
17 :
```

上一节使用的 `multipart.Writer.WriteField()` 方法和 `multipart.Writer.CreateFormFile()` 方法属于高级 API，可以在不直接操作 `multipart` 的构成元素的情况下创建内容，各部分持有的首部也可以自动设置。代码清单 3-11 中的代码是通过抽出各个方法中执行的详细处理来指定任意的 `Content-Type` 的。

3.10 Cookie 的发送和接收

前面介绍的代码仅执行一次发送 HTTP 的请求。HTTP 是无状态的，因此每次发送请求时使用的函数之间不会相互影响。如果服务器未持有状态，那么即使改变函数的调用顺序，结果也不会发生改变。

在使用 Cookie 的情况下，浏览器内部必须持有状态，这时就需要使用 `http.Client` 结构体，而不是前面介绍的各个函数，如代码清单 3-12 所示。

代码清单 3-12 Cookie 的发送和接收

 复制代码

```
1 package main
2
3 import (
4     "log"
5     "net/http"
6     "net/http/cookiejar"
7     "net/http/httputil"
```

```

8 )
9
10 func main() {
11     jar, err := cookiejar.New(nil)    // ①
12     if err != nil {
13         panic(err)
14     }
15     client := http.Client{           // ②
16         Jar: jar,
17     }
18     for i := 0; i < 2; i++ {        // ③
19         // ④
20         resp, err := client.Get("http://localhost:18888/cookie")
21         if err != nil {
22             panic(err)
23         }
24         dump, err := httputil.DumpResponse(resp, true)
25         if err != nil {
26             panic(err)
27         }
28         log.Println(string(dump))
29     }
30 }

```

① 创建 `cookiejar` (Cookie 容器) 的实例来保存 Cookie。

② 创建可保存 Cookie 的 `http.Client` 实例。

③ 由于 Cookie 的结构是客户端第一次访问服务器时接收 Cookie，第二次及之后访问时向服务器发送 Cookie，因此这里进行两次访问。

④ 使用所创建的客户端的 `Get()` 方法 (而非 `http.Get()`) 进行访问。

`net/http/cookiejar` 是作为内置库实现的 Cookie 的一个功能。Go 语言的 `http.Client` 将实现 `http.CookieJar` 接口的任意对象用作处理 Cookie 的后台。因为在默认的实现中，Cookie 仅保存在内存中，所以当程序再次启动时，Cookie 就被删掉了。标准库中并未提供可以将 Cookie 永久保存在文件中的实现，因此我们需要使用第三方实现，或者自己动手进行实现。

`http.Client` 结构体与 `http` 包一样，也实现了 `Get()`、`Head()`、`Post()` 和 `PostForm()`，我们可以直接按照前面介绍的方法使用这些函数。`http` 包中的大部分函数在内部成了默认的 `http.Client` 结构体实例的各个方法的别名。像代码清单 3-13 那样改

写默认的客户端可以使 Cookie 有效。当然，这会影响该流程中 `net/http` 包的运行，因此在改写默认的客户端时要注意其影响范围，比如只在可通过批处理立刻执行完毕的程序中执行该操作。

代码清单 3-13 改写默认的客户端，使 Cookie 有效

 复制代码

```
1 http.DefaultClient = &http.Client{  
2     Jar: jar,  
3 }  
4 :  
5 resp, err := http.Get("http://localhost:18888/cookie")
```

在本书中，我们主要通过赋初始值来创建 Go 语言的对象，如代码清单 3-14 所示。

代码清单 3-14 使用 Go 语言创建对象的方法

 复制代码

```
1 // 通过赋初始值来创建  
2 a := Struct{  
3     Member: "Value",  
4 }  
5  
6 // 使用 new() 函数进行初始化  
7 a := new(Struct)  
8  
9 // 使用 make() 函数进行初始化  
10 // 数组切片、映射和通道专用  
11 a := make(map[string]string)  
12  
13 // 使用库提供的构造函数来创建  
14 // 内部使用上述其中一种语法  
15 a := library.New()
```

具体来说有两种做法：一种是结构体使用 `{}` 赋初始值，创建实例；另一种是使用 `new()` 函数进行初始化。另外，内置类型的数组切片、映射和通道使用 `make()` 函数进行初始

化。在设置任意的 MIME 类型的示例代码中出现的 `textproto.MIMEHeader` 是 `map` 的别名，因此也使用 `make()` 函数。最后的方法是使用库提供的构造函数，当然其内部也使用了前面几种语法之一。

3.11 使用代理

在使用代理时也会用到 `http.Client`。在 `Cookie` 的情况下，可以通过设置 `Jar` 成员变量而使其有效，但这里使用的是 `Transport`。`Transport` 是进行实际的通信的后台。下面是设置代理的 curl 命令。

 复制代码

```
1 $ curl -x http://localhost:18888 http://github.com
```

等价的 Go 语言代码如代码清单 3-15 所示。

代码清单 3-15 通过 Transport 使用代理

 复制代码

```
1 package main
2
3 import (
4     "log"
5     "net/http"
6     "net/http/httputil"
7     "net/url"
8 )
9
10 func main() {
11     proxyUrl, err := url.Parse("http://localhost:18888")
12     if err != nil {
13         panic(err)
14     }
15     client := http.Client{
16         Transport: &http.Transport{
17             Proxy: http.ProxyURL(proxyUrl),
18         },
19     }
20     resp, err := client.Get("http://github.com")
21     if err != nil {
```

```
22     panic(err)
23 }
24 dump, err := httputil.DumpResponse(resp, true)
25 if err != nil {
26     panic(err)
27 }
28 log.Println(string(dump))
29 }
```

虽然 `client.Get()` 的访问目标是外部站点，但代理指向的是本地的测试服务器。在运行上面的代码后，请求并没有被直接发送到外部，而是由本地服务器暂时接收。但由于本地服务器会直接返回响应，所以使用该代码并不能访问 `github.com`。

如果将代理的 URL 改为代码清单 3-16 的形式，就可以赋上 BASIC 认证中的用户名和密码了。

代码清单 3-16 给代理设置 BASIC 认证信息

 复制代码

```
1 http://用户名:密码@github.com
```

因为 `http.Client` 默认使用的代理参数会从环境变量中获取信息并设置代理，所以在设置了各种程序共用的环境变量的 `HTTP_PROXY` 和 `HTTPS_PROXY` 的情况下，客户端就会向环境变量中设置的代理发送请求。另外，在 `NO_PROXY` 中写上略去设置的主机名，客户端就可以在不经过代理的情况下直接与所设置的主机进行通信。

与使用 Cookie 时相同，如果像代码清单 3-17 那样添加代码，将 `http.Client` 中 `Transport` 类型的成员变量设置为 `http` 包的 `DefaultTransport`，`http.Get()` 等全局函数就可以使用代理了。

代码清单 3-17 让全局函数也能使用代理

```
1 http.DefaultTransport = &http.Transport{  
2     Proxy: http.ProxyURL(proxyUrl),  
3 }
```

 复制代码



mattn 的博客中介绍了如何通过 Go 语言使用 Internet Explorer 中设置的代理信息（仅限于 Windows）。

3.12 访问文件系统

第 1 章在介绍 HTTP/1.0 时提到了 URL 结构中的方案。到目前为止的程序访问的都是 `http` 或 `https` 方案。`file` 方案用于访问本地文件。如果执行下面的 curl 命令，作业文件夹中的文件内容就会输出到控制台上。

```
1 $ curl file://main.go
```

 复制代码

通信后台的 `http.Transport` 中存在 `RegisterProtocol` 方法，该方法用来添加这些方案之外的方案所用的 `Transport`。`http.NewFileTransport()` 是用于文件访问的后台。使用这些方法和后台就可以访问本地文件。与前面示例中 URL 的记述方法稍有不同，本地文件的内容会存储在响应主体中返回（代码清单 3-18）。

代码清单 3-18 访问本地文件

```
1 package main  
2  
3 import (  
4     "log"  
5     "net/http"  
6     "net/http/httputil"  
7 )  
8  
9 func main() {  
10     transport := &http.Transport{}
```

 复制代码

```
11     transport.RegisterProtocol("file", http.NewFileTransport(http.Dir(".")))
12     client := http.Client{
13         Transport: transport,
14     }
15     resp, err := client.Get("file://./main.go")
16     if err != nil {
17         panic(err)
18     }
19     dump, err := httputil.DumpResponse(resp, true)
20     if err != nil {
21         panic(err)
22     }
23     log.Println(string(dump))
24 }
```

3.13 发送任意方法

前面的代码使用的都是 `http` 模块的函数或者 `http.Client` 结构体的方法，这些方法仅支持 `GET`、`HEAD` 和 `POST`。当使用其他方法发送请求时，需要使用 `http.Request` 结构体的对象。使用下面的 curl 命令可以发送 `DELETE` 方法。

 复制代码

```
1 $ curl -X DELETE http://localhost:18888
```

代码清单 3-19 是使用 Go 语实现的代码。

代码清单 3-19 使用 `http.Request` 结构体发送 `DELETE` 方法

 复制代码

```
1 package main
2
3 import (
4     "log"
5     "net/http"
6     "net/http/httputil"
7 )
8
9 func main() {
10     client := &http.Client{}
```

```
11     request, err := http.NewRequest("DELETE", "http://localhost:18888", nil)
12     if err != nil {
13         panic(err)
14     }
15     resp, err := client.Do(request)
16     if err != nil {
17         panic(err)
18     }
19     dump, err := httputil.DumpResponse(resp, true)
20     if err != nil {
21         panic(err)
22     }
23     log.Println(string(dump))
24 }
```

`http.Request` 结构体是通过构造函数 `http.NewRequest()` 创建的。函数的参数为方法、URL 和主体。与 `POST` 一样，主体中可以使用 `io.Reader`。因此，本章介绍的发送文件的代码和发送 multipart 表单的代码都可以直接使用。要想转换为通过 `PostForm` 发送的形式，就需要将 `url.Values` 转换为符合 `io.Reader` 接口的对象，如代码清单 3-20 所示。

代码清单 3-20 转换为使用 PostForm 发送结构体的形式

 复制代码

```
1 import (
2     "strings"    // 添加该内容
3     "net/url"
4 )
5
6 values := url.Values{"test": {"value"}}
7 reader := strings.NewReader(values.Encode())
```

3.14 发送首部

最后，我们来看一下如何发送首部。在 curl 命令中，需要指定 `-H` 选项来发送首部。

[复制代码](#)

```
1 $ curl -H "Content-Type=image/jpeg" -d "@image.jpeg" http://localhost:18888
```

用于发送任意方法的 `http.Request` 结构体持有 `Header` 字段，它与 `http.Response` 结构体的 `Header` 字段是同一个字段。`GET` 方法的示例代码中使用了 `Get()` 方法来获取首部的值，而添加首部需要使用 `Add()` 方法。

[复制代码](#)

```
1 // 添加首部
2 request.Header.Add("Content-Type", "image/jpeg")
```

`curl` 命令中为特定的首部提供了便捷选项，比如用于 BASIC 认证的 “`--basic -u 用户名：密码`” 选项，以及用于 Cookie 的 “`-c 文件`” 选项等。Go 语言中也提供了类似辅助方法作为 `http.Request` 结构体的方法，具体如下所示。

[复制代码](#)

```
1 // BASIC 认证
2 request.SetBasicAuth("用户名", "密码")
3 // 手动添加一个 Cookie
4 request.AddCookie(&http.Cookie)
```

即使首部中未设置 Cookie，也可以通过在 `http.Client` 的 `Jar` 中设置 `cookiejar.Jar` 的实例来发送和接收 Cookie。通过手动进行设置，也可以自由地发送未接收的 Cookie。

3.15 超时

我们可以在 `curl` 命令中使用 `-m` 选项来设置超时时间，具体如下所示。

 复制代码

```
1 $ curl -m 2 http://localhost:8080/slow_page
```

从版本 1.7 开始，Go 的标准库中引入了 Context 机制。Context 有以下两个功能。

从外部停止所有的处理（类似于 C# 中的 `CancellationToken`）

在与同一个请求相关的处理中（请求范围）共享值（类似于 Java 等编程语言中的线程本地存储）

第一个功能可以用于实现超时处理。Context 是 Go 语言中与应用程序设计紧密相关的功能，但本书的重点并不是介绍 Go 语言的编程方法，因此这里就不再详细介绍 Context 了。



网上有许多关于 Context 的详细介绍，请读者自行查阅。本节仅介绍超时的相关内容。第二个功能将在第 12 章介绍。

Context 具有父子关系。根 Context（使用 `context.Background()` 创建）不会自动结束。拥有超时控制的 Context 要使用 `context.WithTimeout()` 函数创建。在创建请求时，使用 `http.NewRequestWithContext()` 代替 `http.NewRequest()`，在传入 Context 后，超时就设置好了。多个请求可以共享同一个 Context，这样一来，相关的所有请求就可以同时停止。

 复制代码

```
1 // 先创建 Context
2 ctx, cancel := context.WithTimeout(context.Background(), 2 * time.Second)
3 defer cancel()
4
5 // 创建拥有根 Context 的请求
6 req, _ := http.NewRequestWithContext(ctx, "GET", "http://localhost:8080/slow_page",
7 nil)
8 // 然后正常发送请求
9 res, err := http.DefaultClient.Do(req)
```

在 Go 1.7 之前的版本中，我们可以使用 `http.Client` 的 `Timeout` 属性或 `http.Transport` 等方法实现超时，而如今，如无特殊要求，最好使用 `Context`。

3.16 国际化域名

第 1 章中介绍了与 URL 国际化有关的内容。当时介绍了用于转换的服务，而使用 Go 语言也可以进行转换。

`idna.ToASCII()` 和 `idna.ToUnicode()` 函数用来转换域名。在发送请求之前，通过 `idna.ToASCII()` 转换域名并使用前面介绍的 API，就可以获取域名的站点信息（代码清单 3-21）。

代码清单 3-21 URL 国际化的应对方法

 复制代码

```
1 package main
2
3 import (
4     "fmt"
5     "golang.org/x/net/idna"
6 )
7
8 func main() {
9     src := "握力王"
10    ascii, err := idna.ToASCII(src)
11    if err != nil {
12        panic(err)
13    }
14    fmt.Printf("%s -> %s\n", src, ascii)
15 }
```

3.17 本章小结

本章介绍了如何使用 Go 语言实现 HTTP/1.0 中出现的各种请求模式。关于 Go 语言，我们对在阅读伪代码时比较难理解的实例的创建，以及只看语法的话难以理解其用法的 `io.Reader` 和 `io.Writer` 的处理进行了补充说明。

本章介绍了许多模式，我们可以通过图 3-1 来了解在各种模式下使用什么方法。

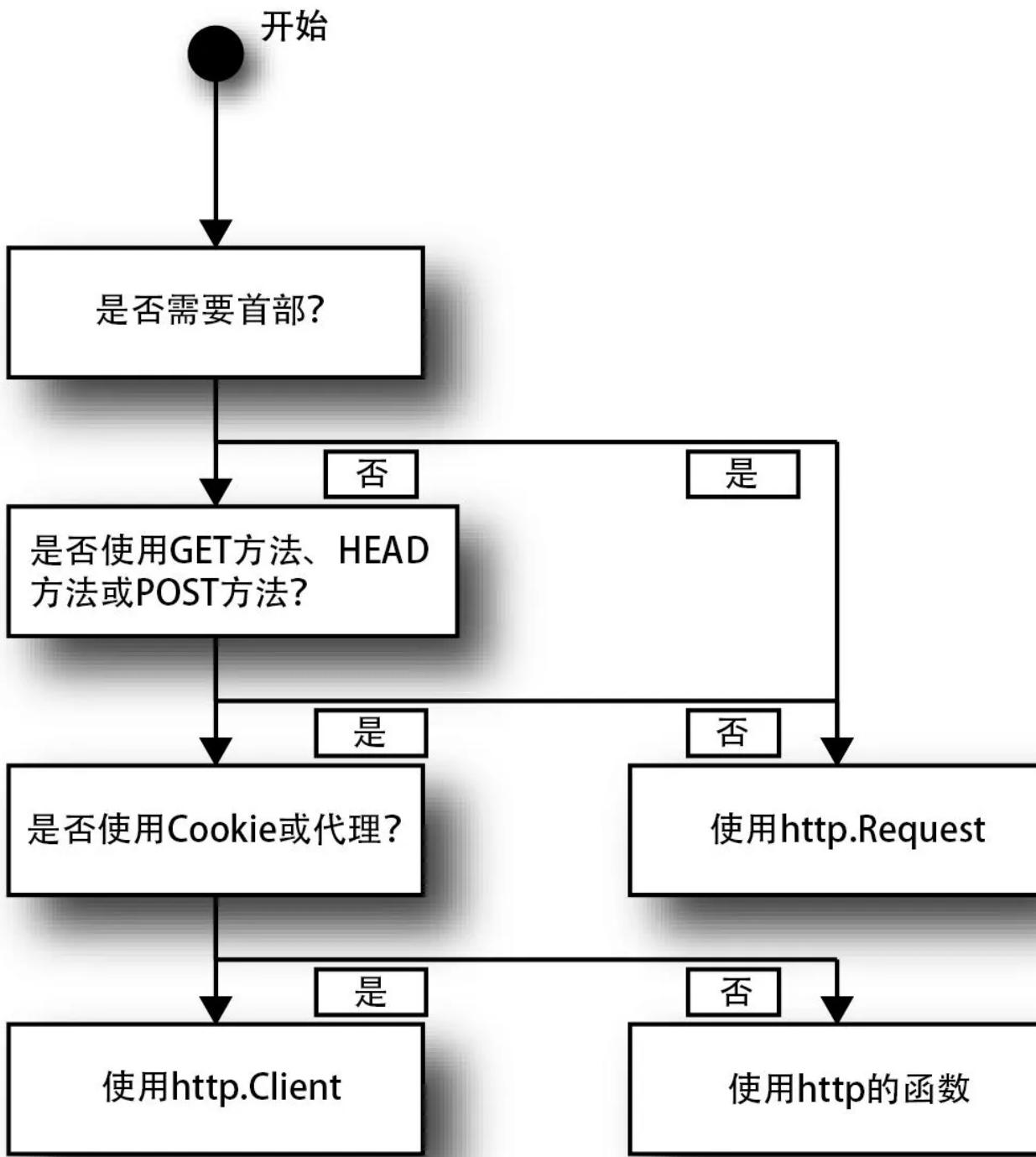


图 3-1 各种模式下使用的方法

第 1 章介绍了使用 curl 命令处理 4 个基本元素的方法，本章介绍了使用编程语言处理 4 个基本元素的方法。第 2 章提到过，HTTP 的许多功能是通过运用 4 个基本元素来实现的。如果按照浏览器来实现缓存结构，就要编写大量代码，而本章至少涵盖了与服务器直接通信的

相关内容。本章介绍的 API 对 HTTP/2 也适用。如果只是使用 Web API，阅读本章的内容就足够了。

AI智能总结

本文介绍了使用 Go 语言实现 HTTP/1.0 客户端的方法。首先强调了选择 Go 语言的优点，如轻量级语言规范、丰富的标准库、编译速度快、易于交叉编译等。接着详细介绍了 Go 语言的 API 结构，包括功能有限但易于使用的 API、能使用 Cookie 的、可稍加控制的 API 以及原生的、可访问所有功能的 API。文章还列举了浏览器发送和接收信息的模式，并对比了使用 Go 语言和其他编程语言的区别。重点介绍了使用 Go 语言实现几个典型的模式，包括发送 GET 方法、获取带查询的信息、使用 HEAD 获取首部、使用 x-www-form-urlencoded 发送表单等。通过代码清单和对应的 curl 命令的对比，读者可以清晰地了解如何使用 Go 语言实现这些功能。整体而言，本文通过简洁清晰的语言和实际的代码示例，帮助读者快速了解了使用 Go 语言实现 HTTP/1.0 客户端的方法和技术特点。文章内容涵盖了发送 x-www-form-urlencoded 形式的表单、使用 POST 方法发送任意主体、使用 multipart/form-data 形式发送文件以及 Cookie 的发送和接收等内容，为读者提供了全面的技术指导。文章还介绍了如何使用代理和访问文件系统，以及发送任意方法和首部，为读者提供了更加全面的技术知识。文章内容丰富，适合对 Go 语言和 HTTP/1.0 客户端实现感兴趣的读者阅读学习。

-
- [1]: 指相对平均值的偏差数值，在日本常用于考察学生的智力和学习能力。计算公式为“偏差值 = (个人成绩 - 平均成绩) × 10 / 标准差 + 50”。——译者注
 - [2]: 笔者在 ASCII PROGRAMMING+ 的网站上连载了文章，其中有更加详细的说明和示例。
 - [3]: 笔者在书中加入了 mattn 的博客中介绍的内容，并对这部分内容进行了适当的改动。

精选留言

由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。