方法将会返回一个 EOF，而程序则会在检测到这个 EOF 之后退出 for 循环。

我们可以通过执行以下命令来运行这个 JSON 解码器：

```
go run json.go
```

如果一切正常，将会看到以下结果：

```
{1 Hello World! {2 Sau Sheong} [{1 Have a great day! Adam} {2 How are you today? Betty}]}
```

最后，在面对 JSON 数据时，我们可以根据输入决定使用 Decoder 还是 Unmarshal：如果 JSON 数据来源于 io.Reader 流，如 http.Request 的 Body，那么使用 Decoder 更好；如果 JSON 数据来源于字符串或者内存的某个地方，那么使用 Unmarshal 更好。

7.5.2 创建 JSON

正如上一个小节所示，分析 JSON 的方法和分析 XML 的方法是非常相似的。同样地，如图 7-9 所示，创建 JSON 的方法和创建 XML 的方法也是相似的。

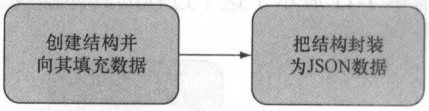

图 7-9 使用 Go 创建 JSON：创建结构并将其封装为 JSON 数据

代码清单 7-12 展示了把 Go 结构封装为 JSON 数据的具体代码。

代码清单 7-12 将结构封装为 JSON

```
package main

import (
  "encoding/json"
  "fmt"
  "io/ioutil"
)

type Post struct {
  Id       int       `json:"id"`
  Content  string    `json:"content"`
  Author   Author    `json:"author"`
  Comments []Comment `json:"comments"`
}

type Author struct {
  Id   int    `json:"id"`
  Name string `json:"name"`
}

type Comment struct {
  Id      int    `json:"id"`
  Content string `json:"content"`
  Author  string `json:"author"`
}
```

创建结构并向里面填充数据

```
func main() {
  post := Post{
    Id:      1,
    Content: "Hello World!",
    Author: Author{
      Id:   2,
      Name: "Sau Sheong",
    },
    Comments: []Comment{
      Comment{
        Id:      3,
        Content: "Have a great day!",
        Author:  "Adam",
      },
      Comment{
        Id:      4,
        Content: "How are you today?",
        Author:  "Betty",
      },
    },
  }

  output, err := json.MarshalIndent(&post, "", "\t\t")
  if err != nil {
    fmt.Println("Error marshalling to JSON:", err)
    return
  }
  err = ioutil.WriteFile("post.json", output, 0644)
  if err != nil {
    fmt.Println("Error writing JSON to file:", err)
    return
  }
}
```

把结构封装为由字节切片组成的 JSON 数据

跟处理 XML 时的情况一样，这个封装程序使用的结构和之前分析 JSON 时使用的结构是相同的。程序首先会创建一些结构，然后通过调用 `MarshalIndent` 函数将结构封装为由字节切片组成的 JSON 数据（`json` 库的 `MarshalIndent` 函数和 `xml` 库的 `MarshalIndent` 函数的作用是类似的）。最后，程序会将封装所得的 JSON 数据存储到指定的文件中。

正如我们可以通过编码器手动创建 XML 一样，我们也可以通过编码器手动将 Go 结构编码为 JSON 数据，图 7-10 展示了这个过程。

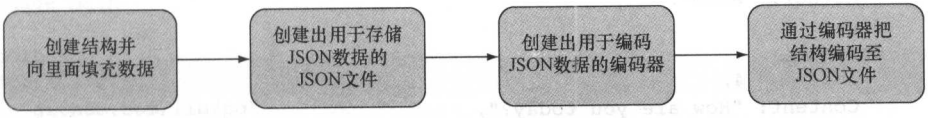

图 7-10 使用 Go 创建 JSON 数据：通过编码器把结构编码为 JSON

代码清单 7-13 展示了 `json.go` 文件中包含的代码，这些代码可以根据给定的 Go 结构创建

相应的 JSON 文件。

代码清单 7-13 使用 Encoder 把结构编码为 JSON

```
package main

import (
  "encoding/json"
  "fmt"
  "io"
  "os"
)

type Post struct {
  Id       int         `json:"id"`
  Content  string      `json:"content"`
  Author   Author      `json:"author"`
  Comments []Comment   `json:"comments"`
}

type Author struct {
  Id   int    `json:"id"`
  Name string `json:"name"`
}

type Comment struct {
  Id      int    `json:"id"`
  Content string `json:"content"`
  Author  string `json:"author"`
}

func main() {

  post := Post{
    Id:      1,
    Content: "Hello World!",
    Author: Author{
      Id:   2,
      Name: "Sau Sheong",
    },
    Comments: []Comment{
      Comment{
        Id:      3,
        Content: "Have a great day!",
        Author: "Adam",
      },
      Comment{
        Id:      4,
        Content: "How are you today?",
        Author: "Betty",
      },
    },
  },
```

创建结构并向里面填充数据

```
    jsonFile, err := os.Create("post.json")
    if err != nil {
      fmt.Println("Error creating JSON file:", err)
      return
    }
    encoder := json.NewEncoder(jsonFile)
    err = encoder.Encode(&post)
    if err != nil {
      fmt.Println("Error encoding JSON to file:", err)
      return
    }
  }
```

创建用于存储数据的 JSON 文件

根据给定的 JSON 文件创建出相应的编码器

把结构编码到 JSON 文件里面

跟之前一样，程序会创建一个用于存储 JSON 数据的 JSON 文件，并通过把这个文件传递给 `NewEncoder` 函数来创建一个编码器。接着，程序会调用编码器的 `Encode` 方法，并向其传递一个指向 `Post` 结构的引用。在此之后，`Encode` 方法会从结构里面提取数据并将其编码为 JSON 数据，然后把这些 JSON 数据写入创建编码器时给定的 JSON 文件里面。

关于分析和创建 XML 和 JSON 的介绍到这里就结束了。虽然最近这两节介绍的内容可能会因为模式相似而显得有些乏味，但这些基础知识对于接下来的一节学习如何创建 Go Web 服务是不可或缺的，因此花时间学习和掌握这些知识是非常值得的。

7.6 创建 Go Web 服务

创建 Go Web 服务并不是一件困难的事情：如果你仔细地阅读并理解了前面各个章节介绍的内容，那么掌握接下来要介绍的知识对你来说应该是轻而易举的。

本节将要构建一个简单的基于 REST 的 Web 服务，它允许我们对论坛帖子执行创建、获取、更新以及删除操作。具体来说，我们将会使用第 6 章介绍过的 CRUD 函数来包裹起一个 Web 服务接口，并通过 JSON 格式来传输数据。除了本章之外，后续的章节也会沿用这个 Web 服务作为例子，对其他概念进行介绍。

代码清单 7-14 展示了实现 Web 服务需要用到的数据库操作，这些操作和 6.4 节介绍过的操作基本相同，只是做了一些简化。这些代码定义了 Web 服务需要对数据库执行的所有操作，它们都隶属于 `main` 包，并且被放置到了 `data.go` 文件中。

代码清单 7-14 使用 data.go 访问数据库

```
package main

import (
  "database/sql"
  _ "github.com/lib/pq"
)

var Db *sql.DB
```

```
func init() {                                          连接到数据库
  var err error
  Db, err = sql.Open("postgres", "user=gwp dbname=gwp password=gwp sslmode=
   disable")
  if err != nil {
    panic(err)
  }
}

                                                       获取指定的帖子
func retrieve(id int) (post Post, err error) {
  post = Post{}
  err = Db.QueryRow("select id, content, author from posts where id = $1",
   id).Scan(&post.Id, &post.Content, &post.Author)
  return
}

                                                       创建一篇新帖子
func (post *Post) create() (err error) {
  statement := "insert into posts (content, author) values ($1, $2) returning
   id"
  stmt, err := Db.Prepare(statement)
  if err != nil {
    return
  }
  defer stmt.Close()
  err = stmt.QueryRow(post.Content, post.Author).Scan(&post.Id)
  return
}

                                                       更新指定的帖子
func (post *Post) update() (err error) {
  _, err = Db.Exec("update posts set content = $2, author = $3 where id =
  $1", post.Id, post.Content, post.Author)
  return
}

                                                       删除指定的帖子
func (post *Post) delete() (err error) {
  _, err = Db.Exec("delete from posts where id = $1", post.Id)
  return
}
```

正如所见，这些代码跟前面代码清单 6-6 展示过的代码非常相似，只是在函数名和方法名上稍有区别，因此我们在这里就不再一一解释了。如果你需要重温一下这些代码的作用，那么可以去复习一下 6.4 节。

在拥有了对数据库执行 CRUD 操作的能力之后，让我们来学习一下如何实现真正的 Web 服务。代码清单 7-15 展示了整个 Web 服务的实现代码，这些代码保存在文件 server.go 中。

代码清单 7-15 定义在 server.go 文件内的 Go Web 服务

```
package main

import (
  "encoding/json"
  "net/http"
```

```
  "path"
  "strconv"
)

type Post struct {
  Id      int    `json:"id"`
  Content string `json:"content"`
  Author  string `json:"author"`
}

func main() {
  server := http.Server{
    Addr: "127.0.0.1:8080",
  }
  http.HandleFunc("/post/", handleRequest)
  server.ListenAndServe()
}

func handleRequest(w http.ResponseWriter, r *http.Request) {
  var err error
  switch r.Method {
  case "GET":
    err = handleGet(w, r)
  case "POST":
    err = handlePost(w, r)
  case "PUT":
    err = handlePut(w, r)
  case "DELETE":
    err = handleDelete(w, r)
  }
  if err != nil {
    http.Error(w, err.Error(), http.StatusInternalServerError)
    return
  }
}

func handleGet(w http.ResponseWriter, r *http.Request) (err error) {
  id, err := strconv.Atoi(path.Base(r.URL.Path))
  if err != nil {
    return
  }
  post, err := retrieve(id)
  if err != nil {
    return
  }
  output, err := json.MarshalIndent(&post, "", "\t\t")
  if err != nil {
    return
  }
  w.Header().Set("Content-Type", "application/json")
  w.Write(output)
  return
}
```

多路复用器负责将请求转发给正确的处理器函数

获取指定的帖子

```
func handlePost(w http.ResponseWriter, r *http.Request) (err error) {
  len := r.ContentLength
  body := make([]byte, len)
  r.Body.Read(body)
  var post Post
  json.Unmarshal(body, &post)
  err = post.create()
  if err != nil {
    return
  }
  w.WriteHeader(200)
  return
}

func handlePut(w http.ResponseWriter, r *http.Request) (err error) {
  id, err := strconv.Atoi(path.Base(r.URL.Path))
  if err != nil {
    return
  }
  post, err := retrieve(id)
  if err != nil {
    return
  }
  len := r.ContentLength
  body := make([]byte, len)
  r.Body.Read(body)
  json.Unmarshal(body, &post)
  err = post.update()
  if err != nil {
    return
  }
  w.WriteHeader(200)
  return
}

func handleDelete(w http.ResponseWriter, r *http.Request) (err error) {
  id, err := strconv.Atoi(path.Base(r.URL.Path))
  if err != nil {
    return
  }
  post, err := retrieve(id)
  if err != nil {
    return
  }
  err = post.delete()
  if err != nil {
    return
  }
  w.WriteHeader(200)
  return
}
```

创建新的帖子

更新指定的帖子

删除指定的帖子

这段代码的结构非常直观：handleRequest多路复用器会根据请求使用的HTTP方法，把请求转发给相应的CRUD处理器函数，这些函数都接受一个ResponseWriter和一个Request

作为参数，并返回可能出现的错误作为函数的执行结果；handleRequest 会检查这些函数的执行结果，并在发现错误时通过 StatusInternalServerError 返回一个 500 状态码。

接下来，让我们首先从帖子的创建操作开始，对 Go Web 服务的各个部分进行详细的解释，handlePost 函数如代码清单 7-16 所示。

代码清单 7-16 用于创建帖子的函数

```
func handlePost(w http.ResponseWriter, r *http.Request) (err error) {
  len := r.ContentLength
  body := make([]byte, len)
  r.Body.Read(body)
  var post Post
  json.Unmarshal(body, &post)
  err = post.create()
  if err != nil {
    return
  }
  w.WriteHeader(200)
  return
}
```

读取请求主体，并将其存储在字节切片中

创建一个字节切片

把切片存储的数据解封至 Post 结构

创建数据库记录

handlePost 函数首先会根据内容的长度创建出一个字节切片，然后将请求主体记录的 JSON 字符串读取到字节切片里面。之后，函数会声明一个 Post 结构，并将字节切片存储的内容解封到这个结构里面。这样一来，函数就拥有了一个填充了数据的 Post 结构，于是它调用结构的 Create 方法，把记录在结构中的数据存储到了数据库里面。

为了调用 Web 服务，我们需要用到第 3 章介绍过的 cURL，并在终端中执行以下命令：

```
curl -i -X POST -H "Content-Type: application/json"  -d '{"content":"My first
➥post","author":"Sau Sheong"}' http://127.0.0.1:8080/post/
```

这个命令首先会把 Content-Type 首部设置为 application/json，然后通过 POST 方法，向地址 http://127.0.0.1/post/发送一条主体为 JSON 字符串的 HTTP 请求。如果一切顺利，应该会看到以下结果：

```
HTTP/1.1 200 OK
Date: Sun, 12 Apr 2015 13:32:14 GMT
Content-Length: 0
Content-Type: text/plain; charset=utf-8
```

不过这个结果只能证明处理器函数在处理这个请求的时候没有发生任何错误，却无法说明帖子真的已经创建成功了。为了验证这一点，我们需要通过执行以下 SQL 查询来检视一下数据库：

```
psql -U gwp -d gwp -c "select * from posts;"
```

如果帖子创建成功了，应该会看到以下结果：

```
 id |    content     |   author
----+----------------+------------
  1 | My first post | Sau Sheong
```

```
(1 row)
```

除了 handlePost 函数之外，我们的 Web 服务的每个处理器函数都会假设目标帖子的 id 已经包含在了 URL 里面。比如说，当用户想要获取一篇帖子时，Web 服务接收到的请求应该指向以下 URL：

```
/post/<id>
```

而这个 URL 中的<id>记录的就是帖子的 id。代码清单 7-17 展示了函数是如何通过这一机制来获取帖子的。

代码清单 7-17 用于获取帖子的函数

```
func handleGet(w http.ResponseWriter, r *http.Request) (err error) {
  id, err := strconv.Atoi(path.Base(r.URL.Path))
  if err != nil {
    return
  }
  post, err := retrieve(id)
  if err != nil {
    return
  }
  output, err := json.MarshalIndent(&post, "", "\t\t")
  if err != nil {
    return
  }
  w.Header().Set("Content-Type", "application/json")
  w.Write(output)
  return
}
```

从数据库里获取数据，并将其填充到 Post 结构中

把 Post 结构封装为 JSON 字符串

把 JSON 数据写入 ResponseWriter

handleGet 函数首先通过 path.Base 函数，从 URL 的路径中提取出字符串格式的帖子 id，接着使用 strconv.Atoi 函数把这个 id 转换成整数格式，然后通过把这个 id 传递给 retrivePost 函数来获得填充了帖子数据的 Post 结构。

在此之后，程序通过 json.MarshalIndent 函数，把 Post 结构转换成了 JSON 格式的字节切片。最后，程序把 Content-Type 首部设置成了 application/json，并把字节切片中的 JSON 数据写入 ResponseWriter，以此来将 JSON 数据返回给调用者。

为了观察 handleGet 函数是如何工作的，我们需要在终端里面执行以下命令：

```
curl -i -X GET http://127.0.0.1:8080/post/1
```

这条命令会向给定的 URL 发送一个 GET 请求，尝试获取 id 为 1 的帖子。如果一切正常，那么这条命令应该会返回以下结果：

```
HTTP/1.1 200 OK
Content-Type: application/json
Date: Sun, 12 Apr 2015 13:32:18 GMT
Content-Length: 69
```

```
{
    "id": 1,
    "content": "My first post",
    "author": "Sau Sheong"
}
```

在更新帖子的时候，程序同样需要先获取帖子的数据，具体细节如代码清单 7-18 所示。

代码清单 7-18 用于更新帖子的函数

```
func handlePut(w http.ResponseWriter, r *http.Request) (err error) {
id, err := strconv.Atoi(path.Base(r.URL.Path))
  if err != nil {
    return
  }
  post, err := retrieve(id)
  if err != nil {
    return
  }
  len := r.ContentLength
  body := make([]byte, len)
  r.Body.Read(body)
  json.Unmarshal(body, &post)
  err = post.update()
  if err != nil {
    return
  }
  w.WriteHeader(200)
  return
}
```

从数据库里获取指定帖子的数据，并将其填充至 Post 结构

从请求主体中读取 JSON 数据

把 JSON 数据解封至 Post 结构

对数据库进行更新

在更新帖子时，handlePut 函数首先会获取指定的帖子，然后再根据 PUT 请求发送的信息对帖子进行更新。在获取了帖子对应的 Post 结构之后，程序会读取请求的主体，并将主体中的内容解封至 Post 结构，最后通过调用 Post 结构的 update 方法更新帖子。

通过在终端里面执行以下命令，我们可以对之前创建的帖子进行更新：

```
curl -i -X PUT -H "Content-Type: application/json" -d '{"content":"Updated
➥post","author":"Sau Sheong"}' http://127.0.0.1:8080/post/1
```

需要注意的是，跟使用 POST 方法创建帖子时不一样，这次我们需要通过 URL 来指定被更新帖子的 ID。如果一切正常，这条命令应该会返回以下结果：

```
HTTP/1.1 200 OK
Date: Sun, 12 Apr 2015 14:29:39 GMT
Content-Length: 0
Content-Type: text/plain; charset=utf-8
```

现在，我们可以通过再次执行以下 SQL 查询来确认更新是否已经成功：

```
psql -U gwp -d gwp -c "select * from posts;"
```

如无意外，应该会看到以下内容：