Networking Services in Go

Miti Bhat

The Packages

- One of the many reasons for Go's popularity, as a system language, is its inherent support for creating networked programs. The standard library exposes APIs ranging from lowlevel socket primitives to higher-level service afundamental topics abbstractions such as HTTP and RPC. Connected applications can be created by including the following:
- The net package
- A TCP API server
- The HTTP package
- A JSON API server

The net package

The starting point for all networked programs in Go is the*net* package (h t t p s : / / g o l a n g . o rg / p k g / n e t). It provides a rich API to handle low-level networking primitives as well as application-level protocols such as HTTP. Each logical component of a network is represented by a Go type including hardware interfaces, networks, packets, addresses, protocols, and connections. Furthermore, each type exposes a multitude of methods giving Go one of the most complete standard libraries for network programming supporting both IPv4 and IPv6.

Whether creating a client or a server program, Go programmers will need, at a minimum, the network primitives covered in the following sections. These primitives are offered as functions and types to facilitate clients connecting to remote services and servers to handle incoming requests.

Addressing

One of the basic primitives, when doing network programming, is the *address*. The types and functions of the net package use a string literal to represent an address such as "127.0.0.1". The address can also include a service port separated by a colon such as "74.125.21.113:80". Functions and methods in the net package also support string literal representation for IPv6 addresses such as "::1" or [2607:f8b0:4002:c06::65]:80" for an address with a service port of 80.

The net.Conn Type

The net.Conn interface represents a generic connection established between two nodes on the network. It implements io.Reader and io.Writer interfaces which allow connected nodes to exchange data using streaming IO primitives. The net package offers network protocol-specific implementations of the net.Conn interface such as *IPConn*, *UDPConn*, and *TCPConn*. Each implementation exposes additional methods specific to its respective network and protocol. However, the default method set defined in net.Conn is adequate for most uses.

The HTTP package

Due to its importance and ubiquity, HTTP is one of a handful of protocols directly implemented in Go. The net/http package (https://golang.org/pkg/net/http/) provides code to implement both HTTP clients and HTTP servers. This section explores the fundamentals of creating HTTP clients and servers using the net/http package. Later, we will return our attention back to building versions of our currency service using HTTP.

The http.Client type

The http.Client struct represents an HTTP client and is used to create HTTP requests and retrieve responses from a server. The following illustrates how to retrieve the text content of Beowulf from Project Gutenberg's website located at h t t p : / / g u t e n b e r g . o r g / c ac h e / e p u b / 1 6 3 2 8 / p g 1 6 3 2 8 . t x t, using the client variable of the http.Client type and prints its content to a standard output:

Demo: httpclient.go

The previous example uses the client.Get method to retrieve content from the remote server using the HTTP protocol method GET internally. The GET method is part of several convenience methods offered, by the Client type, to interact with HTTP servers as summarized in the following table. Notice that all of these methods return a value of the *http.Response type (discussed later) to handle responses returned by the HTTP server.

Method	Description

`Client.Get`	As discussed earlier, `Get` is a convenience method that issues an HTTP `GET` method to retrieve the resource specified by the `url` parameter from the server:
	`Get(url string,`
	`) (resp *http.Response, err error)`

`Client.Post`	The `Post` method is a convenience method that issues an HTTP `POST` method to send the content specified by the `body` parameter to the server specified by the `url` parameter:
	`Post(`
	`  url string,`
	`  bodyType string,`
	`  body io.Reader,`
	`) (resp *http.Response, err error)`

`Client.PostForm`	The `PostForm` method is a convenience method that uses the HTTP `POST` method to send form `data`, specified as mapped key/value pairs, to the server:
	`PostForm(`
	`  url string,`
	`  data url.Values,`
	`) (resp *http.Response, err error)`

`Client.Head`	The `Head` method is a convenience method that issues an HTTP method, `HEAD`, to the remote server specified by the `url` parameter:
	`Head(url string,`
	`)(resp *http.Response, err error)`

`Client.Do`	This method generalizes the request and response interaction with a remote HTTP server. It is wrapped internally by the methods listed in this table. Section *Handling client requests and responses* discusses how to use this method to talk to the server.

http.Client

It should be noted that the HTTP package uses an internal http.Client variable designed to mirror the preceding methods as package functions for further convenience. They include http.Get, *http.Post*, http.PostForm, and http.Head. The following snippet shows the previous example using http.Get instead of the method from the http.Client:

Demo: httpclient1a.go

Configuring the client

- Besides the methods to communicate with the remote server, the http.Client type exposes additional attributes that can be used to modify and control the behavior of the client. For instance, the following source snippet sets the timeout to handle a client request to complete within 21 seconds using the Timeout attribute of the Client type:

- Demo: httpclient2.go
- The Transport field of the Client type provides further means of controlling the settings

of a client. For instance, the following snippet creates a client that disables the connection reuse between successive HTTP requests with the DisableKeepAlive field. The code also uses the Dial function to specify further granular control over the HTTP connection used by the underlying client, setting its timeout value to 30 seconds:

Handling client requests and responses

An http.Request value can be explicitly created using the http.NewRequest function. A request value can be used to configure HTTP settings, add headers, and specify the content body of the request. The following source snippet uses the http.Request type to create a new request which is used to specify the headers sent to the server:

Demo: httpclient3.go

The http.NewRequest function has the following signature:

```
func NewRequest(method, uStr string, body io.Reader) (*http.Request, error)
```

It takes a string that specifies the HTTP method as its first argument. The next argument specifies the destination URL. The last argument is an io.Reader that can be used to specify the content of the request (or set to nil if the request has no content). The function returns a pointer to a http.Request struct value (or a non-nil error if one occurs). Once the request value is created, the code uses the Header field to add HTTP headers to the request to be sent to the server.

Handling client requests and responses

Once a request is prepared (as shown in the previous source snippet), it is sent to the server using the *Do* method of the http.Client type and has the following signature:

```
Do(req *http.Request) (*http.Response, error)
```

The method accepts a pointer to an http.Request value, as discussed in the previous section. It then returns a pointer to an http.Response value or an error if the request fails.

In the previous source code, resp, err := client.Do(req) is used to send the request to the server and assigns the response to the resp variable.

Handling client requests and responses

The response from the server is encapsulated in struct http.Response which contains

several fields to describe the response including the HTTP response status, content length, headers, and the response body. The response body, exposed as the http.Response.Body field, implements the io.Reader which affords the use streaming IO primitives to consume the response content.

The Body field also implements *io.Closer* which allows the closing of IO resources. The previous source uses defer resp.Body.Close() to close the IO resource associated with the response body. This is a recommended idiom when the server is expected to return a non-nil body.

A simple HTTP server

The HTTP package provides two main components to accept HTTP requests and serve responses:
The http.Server type uses the http.Handler interface type, defined in the following
listing, to receive requests and server responses:

```
type Handler interface {
ServeHTTP(ResponseWriter, *Request)
}
```

Any type that implementshttp.Handler can be registered (explained next) as a valid handler. The Go http.Server type is used to create a new server. It is a struct whose values can be configured, at a minimum, with the TCP address of the service and a handler that will respond to incoming requests. The following code snippet shows a simple HTTP server that defines the msg type as handler registered to handle incoming client requests:e http.Server type

Demo: httpserv0.go

A simple HTTP server

In the previous code, the msg type, which uses a string as its underlying type, implements the ServeHTTP() method making it a valid HTTP handler. Its ServeHTTP method uses the response parameter, resp, to print response headers "200 OK" and "Content-Type: text/html". The method also writes the string value m to the response variable using fmt.Fprint(resp, m) which is sent back to the client.

In the code, the variable server is initialized as http.Server{Addr: ":4040", Handler: msgHandler}. This means the server will listen on all network interfaces at port 4040 and will use variable msgHandler as its http.Handler implementation. Once initialized, the server is started with the server.ListenAndServe() method call that is used to block and listen for incoming requests.

The default server

Besides the Addr and Handler, the http.Server struct exposes several additional fields that can be used to control different aspects of the HTTP service such as connection, timeout values, header sizes, and TLS configuration. For instance, the following snippet shows an updated example which specifies the server's read and write timeouts:

Demo: httpserv1.go

In the code, the http.ListenAndServe(":4040", msgHandler) function is used to start a server which is declared as a variable in the HTTP package. The server is configured with the local address ":4040" and the handler msgHandler (as was done earlier) to handle all incoming requests.

Routing requests with http.ServeMux

The http.Handler implementation introduced in the previous section is not sophisticated. No matter what URL path is sent with the request, it sends the same response back to the client. That is not very useful. In most cases, you want to map each path of a request URL to a different response.

Fortunately, the HTTP package comes with the http.ServeMux type which can multiplex incoming requests based on URL patterns. When an http.ServeMux handler receives a request, associated with a URL path, it dispatches a function that is mapped to that URL.

The following abbreviated code snippet shows http.ServeMux variable mux configured to handle two URL paths "/hello" and "/goodbye":

Demo: httpserv3.go

Routing requests with http.ServeMux

The code declares two functions assigned to variables hello and goodbye. Each function is mapped to a path "/hello" and "/goodbye" respectively using the

mux.HandleFunc("/hello", hello) and mux.HandleFunc("/goodbye", goodbye)

method calls. When the server is launched, with http.ListenAndServe(":4040", mux), its handler will route the request "http://localhost:4040/hello" to the hello function and requests with the path "http://localhost:4040/goodbye" to the goodbye function.

The default ServeMux

It is worth pointing out that the HTTP package makes available a default ServeMux internally. When used, it is not necessary to explicitly declare a ServeMux variable. Instead the code uses the package function, http.HandleFunc, to map a path to a handler function as illustrated in the following code snippet:

Demo: httpserv4.go

To launch the server, the code calls http.ListenAndServe(":4040", nil) where the ServerMux parameter is set to nil. This implies that the server will default to the perdeclared package instance of http.ServeMux to handle incoming requests.

A JSON API server

it is possible to use the HTTP package to create services over HTTP. Earlier we discussed the perils of creating services using raw

TCP directly when we created a server for our global monetary currency service. In this section, we explore how to create an API server for the same service using HTTP as the underlying protocol. The new HTTP-based service has the following design goals:

Use HTTP as the transport protocol Use JSON for structured communication between client and server

Clients query the server for currency information using JSON-formatted requests The server respond using JSON-formatted responses The following shows the code involved in the implementation of the new service. This time, the server will use the curr1 package

to load and query ISO 4217 currency data from a local CSV file.

The code in the curr1 package defines two types, CurrencyRequest and Currency, intended to represent the client request and currency data returned by the server, respectively as listed here:

Demo: currency.go

A JSON API server

Note that the preceding struct types shown are annotated with tags that describe the JSON properties for each field. This information is used by the JSON encoder to encode the key name of JSON objects (see Chapter 10, *Data IO in Go*, for detail on encoding). The remainder of the code, listed in the following snippet, defines the functions that set up the server and the handler function for incoming requests:

Demo: jsonserv0.go

Since we are leveraging HTTP as the transport protocol for the service, you can see the code is now much smaller than the prior implementation which used pure TCP. The currs function implements the handler responsible for incoming requests. It sets up a decoder to decode the incoming JSON-encoded request to a value of the curr1.CurrencyRequest type as highlighted in the following snippet:

A JSON API server

Since we are leveraging HTTP as the transport protocol for the service, you can see the code is now much smaller than the prior implementation which used pure TCP. The currs function implements the handler responsible for incoming requests. It sets up a decoder to decode the incoming JSON-encoded request to a value of the curr1.CurrencyRequest type as highlighted in the following snippet:

```
var currRequest curr1.CurrencyRequest

dec := json.NewDecoder(req.Body)

if err := dec.Decode(&currRequest); err != nil { ... }
```

A JSON API server

Next, the function executes the currency search by calling curr1.Find(currencies,

currRequest.Get) which returns the slice []Currency assigned to the result variable.

The code then creates an encoder to encode the result as a JSON payload, highlighted in the following snippet:

result := curr1.Find(currencies, currRequest.Get)

enc := json.NewEncoder(resp)

if err := enc.Encode(&result); err != nil { ... }

Lastly, the handler function is mapped to the "/currency" path in the main function with the call to mux.HandleFunc("/currency", currs). When the server receives a request for that path, it automatically executes the currs function.

A JSON API server

Test: curl -X POST -d '{"get":"Euro"}' http://localhost:4040/currency

The cURL command posts a JSON-formatted request object to the server using the -X POST -d '{"get":"Euro"}' parameters. The output (formatted for readability) from the server is comprised of a JSON array of the preceding currency items.

An API server client in Go

An HTTP client can also be built in Go to consume the service with minimal efforts. As is shown in the following code snippet, the client code uses the http.Client type to communicate with the server. It also uses the encoding/json sub-package to decode incoming data (note that the client also makes use of the curr1 package, shown earlier, which contains the types needed to communicate with the server):

Demo: jsonclient0.go

In the previous code, an HTTP client is created to send JSON-encoded request values as

currRequest := &curr1.CurrencyRequest{Get: param} where param is the currency string to retrieve. The response from the server is a payload that represents an array of JSON-encoded objects (see the JSON array in the section, *Testing the API Server with cURL*).

The code then uses a JSON decoder, json.NewDecoder(resp.Body).Decode(&currencies) , to decode the payload from the

response body into the slice, []curr1.Currency.

Thank you