Building web framework with golang

Miclle Zheng

@miclle

一、Simple HTTP Server

使用 `net/http#ListenAndServe` 包实现一个最简单、最基础的 HTTP 服务。

```go
package main

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

func main() {
        // Hello world, the web server
        helloHandler := func(w http.ResponseWriter, req *http.Request) {
                io.WriteString(w, "Hello, world!\n")
        }

        http.HandleFunc("/hello", helloHandler)
        log.Fatal(http.ListenAndServe(":8080", nil))
}
```

```
go run main.go
```

测试访问 /hello 路由：

```
curl -i 127.0.0.1:8080/hello
```

```
HTTP/1.1 200 OK
Date: Fri, 11 Dec 2020 12:09:21 GMT
Content-Length: 14
Content-Type: text/plain; charset=utf-8

Hello, world!
```

测试访问不存在的 /test 路由：

```
curl -i 127.0.0.1:8080/test
```

```
HTTP/1.1 404 Not Found
Content-Type: text/plain; charset=utf-8
X-Content-Type-Options: nosniff
Date: Fri, 11 Dec 2020 12:09:55 GMT
Content-Length: 19

404 page not found
```

handler

handler 函数有两个参数，http.ResponseWriter 和 http.Request。 response writer 被用于写入 HTTP 响应数据，这里我们简单的返回 "Hello, world!\n"。

```
helloHandler := func(w http.ResponseWriter, req *http.Request) {
        io.WriteString(w, "Hello, world!\n")
}
```

源码分析

http.HandleFunc

```go
// HandleFunc registers the handler function for the given pattern
// in the DefaultServeMux.
// The documentation for ServeMux explains how patterns are matched.
func HandleFunc(pattern string, handler func(ResponseWriter, *Request)) {
	DefaultServeMux.HandleFunc(pattern, handler)
}
```

http.ListenAndServe

```go
// ListenAndServe listens on the TCP network address addr and then calls
// Serve with handler to handle requests on incoming connections.
// Accepted connections are configured to enable TCP keep-alives.
//
// The handler is typically nil, in which case the DefaultServeMux is used.
//
// ListenAndServe always returns a non-nil error.
func ListenAndServe(addr string, handler Handler) error {
	server := &Server{Addr: addr, Handler: handler}
	return server.ListenAndServe()
}
```

两个结论：

1. `http.HandleFunc` 会将指定 `pattern` (模式、路由) 的 `handler` 注册在 `DefaultServeMux` 上面
2. `http.ListenAndServe` 如果 `handler` 为 `nil` ，在这种情况下使用 `DefaultServeMux` 。

那么问题来了 DefaultServeMux 是啥?

```go
type ServeMux struct {
	mu    sync.RWMutex
	m     map[string]muxEntry
	es    []muxEntry // slice of entries sorted from longest to shortest.
	hosts bool       // whether any patterns contain hostnames
}

type muxEntry struct {
	h       Handler
	pattern string
}

// NewServeMux allocates and returns a new ServeMux.
func NewServeMux() *ServeMux { return new(ServeMux) }

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

var defaultServeMux ServeMux

func (mux *ServeMux) Handle(pattern string, handler Handler)
func (mux *ServeMux) HandleFunc(pattern string, handler func(ResponseWriter, *Request))
func (mux *ServeMux) Handler(r *Request) (h Handler, pattern string)
func (mux *ServeMux) ServeHTTP(w ResponseWriter, r *Request)
```

二、ServeMux

https://golang.org/pkg/net/http/#ServeMux

> ServeMux is an HTTP request multiplexer. It matches the URL of each incoming request against a list of registered patterns and calls the handler for the pattern that most closely matches the URL.

ServeMux 是一个 HTTP 请求多路复用器。它根据已注册模式列表匹配每个传入请求的 URL，并调用与 URL 最匹配的模式的处理程序。

> Patterns name fixed, rooted paths, like "/favicon.ico", or rooted subtrees, like "/images/" (note the trailing slash). Longer patterns take precedence over shorter ones, so that if there are handlers registered for both "/images/" and "/images/thumbnails/", the latter handler will be called for paths beginning "/images/thumbnails/" and the former will receive requests for any other paths in the "/images/" subtree.

匹配模式固定，较长的模式优先于较短的模式，"/" 匹配子树中任何其他路径的请求

我们这次不使用默认的 `ServeMux` 来完成路由功能：

```go
func main() {
        // 这里生成一个 ServeMux 实例
        handler := http.NewServeMux()

        // 注册路由 /hello/
        handler.HandleFunc("/hello/", func(w http.ResponseWriter, r *http.Request) {
                name := strings.Replace(r.URL.Path, "/hello/", "", 1)
                io.WriteString(w, fmt.Sprintf("Hello %s\n", name))
        })

        // 注册路由 /hello
        handler.HandleFunc("/hello", func(w http.ResponseWriter, r *http.Request) {
                io.WriteString(w, "Hello, world!\n")
        })

        // 注册路由 /
        handler.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
                w.Header().Set("Content-Type", "text/plain")
                w.WriteHeader(http.StatusNotFound)
                io.WriteString(w, fmt.Sprintf("Oops Not found\nURL: %s\n", r.URL.Path))
        })

        log.Fatal(http.ListenAndServe(":8080", handler))
}
```

测试访问 /hello 路由：

```
curl -i 127.0.0.1:8080/hello
```

```
HTTP/1.1 200 OK
Date: Sun, 13 Dec 2020 08:46:13 GMT
Content-Length: 14
Content-Type: text/plain; charset=utf-8

Hello, world!
```

测试访问 /hello/ 路由：

```
curl -i 127.0.0.1:8080/hello/
```

```
HTTP/1.1 200 OK
Date: Sun, 13 Dec 2020 08:46:16 GMT
Content-Length: 7
Content-Type: text/plain; charset=utf-8

Hello
```

测试访问 /hello/foo 路由：

```
curl -i 127.0.0.1:8080/hello/foo
```

```
HTTP/1.1 200 OK
Date: Sun, 13 Dec 2020 08:48:17 GMT
Content-Length: 10
Content-Type: text/plain; charset=utf-8

Hello foo
```

测试访问 /hello/foo/boo 路由：

```
curl -i 127.0.0.1:8080/hello/foo/boo
```

```
HTTP/1.1 200 OK
Date: Sun, 13 Dec 2020 08:48:30 GMT
Content-Length: 14
Content-Type: text/plain; charset=utf-8

Hello foo/boo
```

测试访问不存在的 /test 路由：

```
curl -i 127.0.0.1:8080/test
```

```
HTTP/1.1 404 Not Found
Content-Type: text/plain
Date: Sun, 13 Dec 2020 09:11:45 GMT
Content-Length: 26

Oops Not found
URL: /test
```

测试访问不存在的 /hel/foo 路由：

```
curl -i 127.0.0.1:8080/hel/foo
```

```
HTTP/1.1 404 Not Found
Content-Type: text/plain
Date: Sun, 13 Dec 2020 09:12:18 GMT
Content-Length: 29

Oops Not found
URL: /hel/foo
```

这里发生两处变化：

1. 所有 `/hello/` 的子路径都被路由 1 接管， `/hello/` 后的子路径被赋值给 `name` 。
2. 注册了 `/` 的路由，所以所有没有匹配到前两个路由的 URL 都会被路由 3 接管

默认的 DefaultServeMux 和自己定义的 ServeMux 对象有什么区别呢?

没有太大区别，完全可以把上面代码中的 `handler := http.NewServeMux()` 这一行改为 `handler := http.DefaultServeMux` 。
其实 `http.DefaultServeMux` 本身就是一个 `ServeMux` 类型的变量，只是为了方便，为 http 包添加必要的 API 提供了便利罢了。类似 log 包下的 `std`

```
var std = New(os.Stderr, "", LstdFlags)

// Fatal is equivalent to Print() followed by a call to os.Exit(1).
func Fatal(v ...interface{}) {
        std.Output(2, fmt.Sprint(v...))
        os.Exit(1)
}
```

ServeMux 如何注册 handler? `HandleFunc` 与 `Handle`

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

```go
func (mux *ServeMux) Handle(pattern string, handler Handler) {
	mux.mu.Lock()
	defer mux.mu.Unlock()

	...

	e := muxEntry{h: handler, pattern: pattern}
	mux.m[pattern] = e // 放在 map 里
	if pattern[len(pattern)-1] == '/' {
		mux.es = appendSorted(mux.es, e) // 排序后放在 slice 里
	}

	...
}
```

ServeMux 如何匹配路由并分配处理器?

再回顾一下 `http.ListenAndServe` 的第二个参数:

```
func ListenAndServe(addr string, handler Handler) error
```

Go 支持外部实现路由器, `ListenAndServe` 的第二个参数就是配置外部路由器, 它是一个 `Handler` 接口。即外部路由器实现 `Hanlder` 接口。

https://golang.org/pkg/net/http/#Handler

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

ServeMux 实现了 ServeHTTP 方法

ServeMux ServeHTTP

```go
// ServeHTTP dispatches the request to the handler whose
// pattern most closely matches the request URL.
func (mux *ServeMux) ServeHTTP(w ResponseWriter, r *Request) {
        if r.RequestURI == "*" {
                if r.ProtoAtLeast(1, 1) {
                        w.Header().Set("Connection", "close")
                }
                w.WriteHeader(StatusBadRequest)
                return
        }
        h, _ := mux.Handler(r) // 找到对应的 Handler
        h.ServeHTTP(w, r)      // 响应请求
}
```

ServeMux Handler

```go
func (mux *ServeMux) Handler(r *Request) (h Handler, pattern string) {

        ...

        return mux.handler(host, r.URL.Path)
}

...

// handler is the main implementation of Handler.
// The path is known to be in canonical form, except for CONNECT methods.
func (mux *ServeMux) handler(host, path string) (h Handler, pattern string) {
        mux.mu.RLock()
        defer mux.mu.RUnlock()

        // Host-specific pattern takes precedence over generic ones
        if mux.hosts {
                h, pattern = mux.match(host + path)
        }
        if h == nil {
                h, pattern = mux.match(path)
        }
        if h == nil {
                h, pattern = NotFoundHandler(), ""
        }
        return
}
```

```go
// Find a handler on a handler map given a path string.
// Most-specific (longest) pattern wins.
func (mux *ServeMux) match(path string) (h Handler, pattern string) {
        // Check for exact match first.
        v, ok := mux.m[path]
        if ok {
                return v.h, v.pattern
        }

        // Check for longest valid match.  mux.es contains all patterns
        // that end in / sorted from longest to shortest.
        for _, e := range mux.es {
                if strings.HasPrefix(path, e.pattern) {
                        return e.h, e.pattern
                }
        }
        return nil, ""
}
```

ServeMux 路由器设计思路

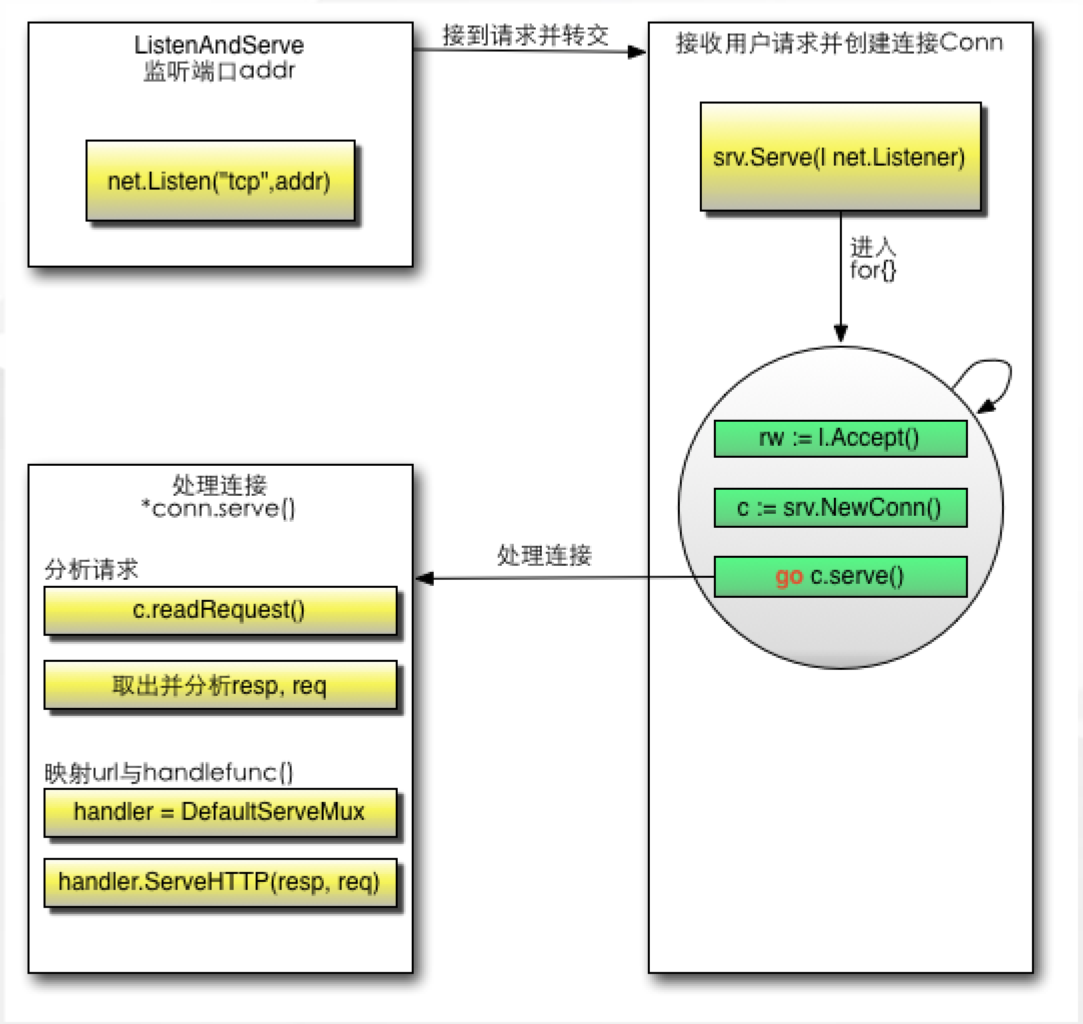
注册路由：

1. 使用 `ServeMux.HandlerFunc` 注册 `func(ResponseWriter, *Request)` 签名的函数作为处理器：
   1.1 在内部转换为 `http.HandlerFunc` 对象， `http.HandlerFunc` 类型实现了 `http.Handler` 接口
   1.2 之后再调用 `ServeMux.Handle` 方法注册路由
2. 使用 `ServeMuxHandle` 注册 `http.Handler` 对象作为处理器
   2.1 将 handler 保存在 ServeMux 内置的 muxEntry map 和 slice 中

匹配并处理路由：

1. 通过 `http.ListenAndServe(addr, mux)` ServeMux.ServeHTTP 接收请求
2. 使用 ServeMux.Handler 匹配合适路由，并返回 handler
   2.1 ServeMux.Handler -> ServeMux.handler(host, r.URL.Path)
   2.2 ServeMux.handler(host, r.URL.Path) -> ServeMux.match(host + path | path) 匹配路由
3. 调用 handler.ServeHTTP(w, r) 处理请求

扩展阅读：一个 HTTP 连接处理流程



https://astaxie.gitbooks.io/build-web-application-with-golang/content/zh/03.3.html

ServeMux 的问题

1. `pattern` 不支持路由占位符、通配符

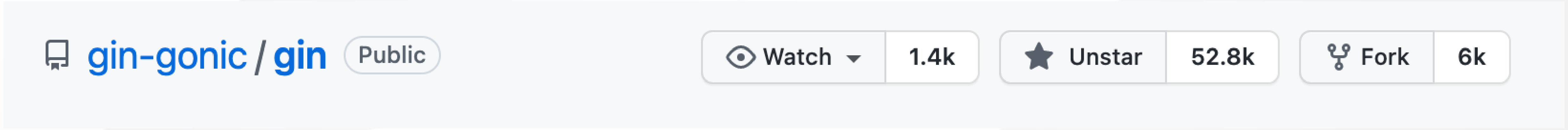
```
"/users/:id"
"/users/{id}"
"/users/{\^[0-9]*$\}"
```

2. 不支持路由 **Middleware Handler**

```
// Router → ...Middleware Handler → Application Handler
var basicAuth   = func(w http.ResponseWriter, r *http.Request) { /* TODO: check username and password */ }
var userProfile = func(w http.ResponseWriter, r *http.Request) { /* TODO: get user profile */ }
DefaultServeMux.HandleFunc("/users/:id", basicAuth, userProfile)
```

3. HandlerFunc 过于原始，复杂业务会导致大量重复代码：
   - 没有便捷的请求参数绑定方法：
     - `/users/:id` => `req.Get("id")` / `req.GetUint("id")`
     - Request Query Params => `req.BindParams(&struct)`
     - Request Body => `req.BindJSON(&struct)`
   - 没有 `context`，如何传递上下文参数
   - Debug Logger: `X-Request-ID` 如何在 handler 调用链中传递?

三、Gin Web Framework



1. 多种路由注册方式

```go
func main() {
	router := gin.Default()

	router.GET("/someGet", gettingHandler)
	router.POST("/somePost", postingHandler)
	router.PUT("/somePut", puttingHandler)
	router.DELETE("/someDelete", deletingHandler)
	router.PATCH("/somePatch", patchingHandler)
	router.HEAD("/someHead", headHandler)
	router.OPTIONS("/someOptions", optionsHandler)

	// By default it serves on :8080 unless a
	// PORT environment variable was defined.
	router.Run()
	// router.Run(":3000") for a hard coded port
}
```

```
[GIN-debug] [WARNING] Running in "debug" mode. Switch to "release" mode in production.
 - using env:   export GIN_MODE=release
 - using code:  gin.SetMode(gin.ReleaseMode)

[GIN-debug] GET     /someGet     -->  gettingHandler  (3 handlers)
[GIN-debug] POST    /somePost    -->  postingHandler  (3 handlers)
[GIN-debug] PUT     /somePut     -->  puttingHandler  (3 handlers)
[GIN-debug] DELETE  /someDelete  -->  deletingHandler (3 handlers)
[GIN-debug] PATCH   /somePatch   -->  patchingHandler (3 handlers)
[GIN-debug] HEAD    /someHead    -->  headHandler     (3 handlers)
[GIN-debug] OPTIONS /someOptions -->  optionsHandler  (3 handlers)

[GIN-debug] Listening and serving HTTP on :8080
```

2. 支持路由组

```go
func main() {
        router := gin.Default()

        // 简单的路由组: v1
        v1 := router.Group("/v1")
        {
                v1.POST("/login", loginEndpoint)
                v1.POST("/submit", submitEndpoint)
                v1.POST("/read", readEndpoint)
        }

        // 简单的路由组: v2
        v2 := router.Group("/v2")
        {
                v2.POST("/login", loginEndpoint)
                v2.POST("/submit", submitEndpoint)
                v2.POST("/read", readEndpoint)
        }

        router.Run(":8080")
}
```

```
[GIN-debug] [WARNING] Running in "debug" mode. Switch to "release" mode in production.
 - using env:   export GIN_MODE=release
 - using code:  gin.SetMode(gin.ReleaseMode)

[GIN-debug] POST    /v1/login     --> loginEndpoint    (3 handlers)
[GIN-debug] POST    /v1/submit    --> submitEndpoint   (3 handlers)
[GIN-debug] POST    /v1/read      --> readEndpoint     (3 handlers)

[GIN-debug] POST    /v2/login     --> loginEndpoint    (3 handlers)
[GIN-debug] POST    /v2/submit    --> submitEndpoint   (3 handlers)
[GIN-debug] POST    /v2/read      --> readEndpoint     (3 handlers)

[GIN-debug] Listening and serving HTTP on :8080
```

3. 丰富的模型绑定和验证方法

```go
func (c *Context) Bind(obj interface{}) error
func (c *Context) BindJSON(obj interface{}) error
func (c *Context) BindXML(obj interface{}) error
func (c *Context) BindQuery(obj interface{}) error
func (c *Context) BindYAML(obj interface{}) error
func (c *Context) BindHeader(obj interface{}) error
func (c *Context) BindUri(obj interface{}) error
func (c *Context) MustBindWith(obj interface{}, b binding.Binding) error
func (c *Context) ShouldBind(obj interface{}) error
func (c *Context) ShouldBindJSON(obj interface{}) error
func (c *Context) ShouldBindXML(obj interface{}) error
func (c *Context) ShouldBindQuery(obj interface{}) error
func (c *Context) ShouldBindYAML(obj interface{}) error
func (c *Context) ShouldBindHeader(obj interface{}) error
func (c *Context) ShouldBindUri(obj interface{}) error
func (c *Context) ShouldBindWith(obj interface{}, b binding.Binding) error
func (c *Context) ShouldBindBodyWith(obj interface{}, bb binding.BindingBody) (err error)

type Auth struct {
        Username string `form:"user"     json:"user"     xml:"user"     binding:"required"`
        Password string `form:"password" json:"password" xml:"password" binding:"required"`
}

func main() {
        router := gin.Default()

        // 绑定 JSON ({"username": "manu", "password": "123"})
        router.POST("/login", func(c *gin.Context) {
                var auth Auth
                if err := c.ShouldBindJSON(&auth); err != nil {
                        c.JSON(http.StatusBadRequest, gin.H{"error": err.Error()})
                        return
                }
                // TODO(m) check username and password
                c.JSON(http.StatusOK, gin.H{"status": "you are logged in"})
        })
}
```

4. 丰富的数据渲染方法

```go
func (c *Context) Render(code int, r render.Render)
func (c *Context) HTML(code int, name string, obj interface{})
func (c *Context) IndentedJSON(code int, obj interface{})
func (c *Context) SecureJSON(code int, obj interface{})
func (c *Context) JSONP(code int, obj interface{})
func (c *Context) JSON(code int, obj interface{})
func (c *Context) AsciiJSON(code int, obj interface{})
func (c *Context) PureJSON(code int, obj interface{})
func (c *Context) XML(code int, obj interface{})
func (c *Context) YAML(code int, obj interface{})
func (c *Context) ProtoBuf(code int, obj interface{})
func (c *Context) String(code int, format string, values ...interface{})
func (c *Context) Redirect(code int, location string)
func (c *Context) Data(code int, contentType string, data []byte)
func (c *Context) DataFromReader(code int, contentLength int64, contentType string, reader io.Reader, extraHeaders map[string]string)
func (c *Context) File(filepath string)
func (c *Context) FileFromFS(filepath string, fs http.FileSystem)
func (c *Context) FileAttachment(filepath, filename string)
func (c *Context) SSEvent(name string, message interface{})
func (c *Context) Stream(step func(w io.Writer) bool) bool
```

5. `gin.Context` Metadata Management （在 Handler 调用链中传递数据）

```go
func (c *Context) Set(key string, value interface{})
func (c *Context) Get(key string) (value interface{}, exists bool)
func (c *Context) MustGet(key string) interface{}
func (c *Context) GetString(key string) (s string)
func (c *Context) GetBool(key string) (b bool)
func (c *Context) GetInt(key string) (i int)
func (c *Context) GetInt64(key string) (i64 int64)
func (c *Context) GetUint(key string) (ui uint)
func (c *Context) GetUint64(key string) (ui64 uint64)
func (c *Context) GetFloat64(key string) (f64 float64)
func (c *Context) GetTime(key string) (t time.Time)
func (c *Context) GetDuration(key string) (d time.Duration)
func (c *Context) GetStringSlice(key string) (ss []string)
func (c *Context) GetStringMap(key string) (sm map[string]interface{})
func (c *Context) GetStringMapString(key string) (sms map[string]string)
func (c *Context) GetStringMapStringSlice(key string) (smss map[string][]string)


func main() {
        router := gin.New()

        var middleware = func(c *gin.Context) {
                c.Set("example", "12345")
                c.Next()
        }

        router.GET("/posts/:id", middleware, func(c *gin.Context) {
                example := c.MustGet("example").(string)
                log.Println(example) // it would print: "12345"
        })
        router.Run()
}
```

6. 强大的 `Middleware` 扩展机制

```go
func Logger() gin.HandlerFunc {
        return func(c *gin.Context) {
                c.Set("example", "12345")
                c.Next()
        }
}

func main() {
        router := gin.New()

        router.Use(Logger())

        router.GET("/test", func(c *gin.Context) {
                example := c.MustGet("example").(string)
                log.Println(example) // it would print: "12345"
        })
        router.Run()
}
```

Gin 的问题

1. `gin.Context` 过于强大，几乎所有的 `Helper` 都挂载在 `Context` 上
   - gin.Context 处理参数绑定
   - gin.Context 处理上下文档参数传递
   - gin.Context 处理数据渲染
   - gin.Context 接管请求流程控制， `c.Next()`, `c.Abort()`
2. 无贯穿 Handler 调用链的 Logger，在每一个 Handler 中从 Context 中，或 Header 中初始化一个带有 `X-Request-ID` 信息的 Logger
3. `gin.Context` 不是 interface 设计，无法扩展
4. gin 与 Ruby on Rails 相比，gin 在底层协议封装与业务逻辑处理之间，更多的是在路由层面，以及提供丰富的帮助方法；gin 仍属于高度抽象的 HTTP Rack 框架；

> 在 Rack 的协议中，将 Rack 应用描述成一个可以响应 call 方法的 Ruby 对象，它仅接受来自外界的一个参数，也就是环境，然后返回一个只包含三个值的数组，按照顺序分别是状态码、HTTP Headers 以及响应请求的正文。

示例代码

```go
// Login 认证用户
func Login(c *gin.Context) {
	var (
		services = ctrl.GetServices(c)
		log      = logger.New(c)
		info     = &params.AuthInfo{}
	)

	if err := c.BindJSON(info); err != nil {
		log.Error(err.Error())
		c.AbortWithError(http.StatusBadRequest, err)
		return
	}

	account, err := services.Authenticator.Auth(log, info)
	if err != nil {
		log.Error(err.Error())
		c.AbortWithStatusJSON(http.StatusBadRequest, e)
		return
	}

	cookie := &http.Cookie{
		Name:     "APPNAME",
		Value:    account.Token,
		Path:     "/",
		MaxAge:   3600,
		Domain:   "domain.com",
		Secure:   config.GetBool("cookie_secure"),
		HttpOnly: true,
		SameSite: http.SameSiteNoneMode,
	}

	http.SetCookie(c.Writer, cookie)

	c.JSON(http.StatusOK, account)
}
```

```go
// Signup 用户注册
func Signup(c *gin.Context) {
	var (
		services = ctrl.GetServices(c)
		log      = logger.New(c)
		info     = &params.CreateUserArgs{}
	)

	if err := c.BindJSON(info); err != nil {
		log.Error(err.Error())
		c.AbortWithError(http.StatusBadRequest, err)
		return
	}

	account, err := services.User.Create(log, info)
	if err != nil {
		log.Error(err.Error())
		c.AbortWithStatusJSON(http.StatusBadRequest, e)
		return
	}

	cookie := &http.Cookie{
		Name:     "APPNAME",
		Value:    token.Token,
		Path:     "/",
		MaxAge:   3600,
		Domain:   "domain.com",
		Secure:   config.GetBool("cookie_secure"),
		HttpOnly: true,
		SameSite: http.SameSiteNoneMode,
	}

	http.SetCookie(c.Writer, cookie)

	c.JSON(http.StatusOK, account)
}
```

四、如何实现一个 Web Engine？

1. http 请求从 `http.ListenAndServe` 方法开始，所以必须实现 `http.Handler` 接口

```go
// ServeHTTP conforms to the http.Handler interface.
func (engine *Engine) ServeHTTP(w http.ResponseWriter, req *http.Request) {

}

// 为了启动方便，实现一个 `Run` 方法
func (engine *Engine) Run(addr string) (err error) {
        err = http.ListenAndServe(addr, engine)
        return
}

func (engine *Engine) RunTLS(addr, certFile, keyFile string) (err error) {
        err = http.ListenAndServeTLS(addr, certFile, keyFile, engine)
        return
}
```

2. 实现一个路由注册方法 `Handle`

```
func (engine *Engine) Handle(httpMethod, relativePath string, handlers ...HandlerFunc)

func (engine *Engine) GET(relativePath string, handlers ...HandlerFunc){
        return engine.Handle(http.MethodGet, relativePath, handlers...)
}

func (engine *Engine) POST(relativePath string, handlers ...HandlerFunc)
func (engine *Engine) DELETE(relativePath string, handlers ...HandlerFunc)
func (engine *Engine) PATCH(relativePath string, handlers ...HandlerFunc)
func (engine *Engine) PUT(relativePath string, handlers ...HandlerFunc)
func (engine *Engine) OPTIONS(relativePath string, handlers ...HandlerFunc)
func (engine *Engine) HEAD(relativePath string, handlers ...HandlerFunc)
```

3. 实现一个路由 `match` 方法，在 `ServeHTTP` 方法中调用

```
func (engine *Engine) match(path string) (h Handler)
```

- 路由支持占位符、通配符：
  - `/posts/:id`
  - `/posts/*`
- 支持具名路由
  - `/posts/latest`
- 匹配优先级使用定义顺序
  1. `/posts/latest`
  2. `/posts/:id`

```go
func main() {
        engine := engine.New()

        engine.POST("/posts", createPostsHandler)
        engine.GET("/posts", getPostsHandler)
        engine.GET("/posts/latest", getLatestPostsHandler)

        engine.Use(logger.Logger)

        engine.Group("/posts/:id", getPostMiddleware, func(group *Router) {
                group.GET("", getPostHandler)
                group.PATCH("", updatePostHandler)
                group.DELETE("", deletePostHandler)
        })

        engine.Run()
}
```

4. 贯穿 Handler 调用链的 Logger

```go
func Login(c *gin.Context) {
	var (
		log     = logger.New(c)
		info    = &params.AuthInfo{}
	)

	if err := c.BindJSON(info); err != nil {
		log.Error(err.Error())
		c.AbortWithError(http.StatusBadRequest, err)
		return
	}

	...
}
```

→

```go
func Login(c *gin.Context) {
	var (
		// log = logger.New(c)
		info = &params.AuthInfo{}
	)

	if err := c.BindJSON(info); err != nil {
		c.Logger.Error(err.Error())
		c.AbortWithError(http.StatusBadRequest, err)
		return
	}

	...
}
```

5. Handler 改造

如果把 web 服务 API 看成一个函数集，可以有如下思考：

- 用户发出的请求作为函数的入参
- 函数的返回值作为请求的响应
- 将请求参数的数据绑定，以及请求返回结果渲染这些与业务无关逻辑，抽象到框架中

伪代码：

```
// posts
RoutePattern1(req *http.Request) (resp http.ResponseWriter, err error)

// posts/latest
RoutePattern2(req *http.Request) (resp http.ResponseWriter, err error)

// posts/:id
RoutePattern3(req *http.Request) (resp http.ResponseWriter, err error)
```

进一步把 http.Request 封装在 Context 中

```
// posts
RoutePattern1(c *Context) (resp http.ResponseWriter, err error)

// posts/latest
RoutePattern2(c *Context) (resp http.ResponseWriter, err error)

// posts/:id
RoutePattern3(c *Context) (resp http.ResponseWriter, err error)
```

把反复出现的请求数据绑定提升到函数参数中

```go
// 重复代码
if err := c.BindJSON(&args); err != nil {
        log.Error(err.Error())
        c.AbortWithError(http.StatusBadRequest, err)
        return
}
```

```go
type Info struct {
        ID uint `uri:"id" binding:"required"`
}

RoutePattern1(c *Context, info *Info) (resp http.ResponseWriter, err error)
```

```go
type AuthInfo struct {
        Login     string `json:"login"    binding:"required"`
        Password  string `json:"password" binding:"required"`
}

RoutePattern1(c *Context, info *AuthInfo) (resp http.ResponseWriter, err error)
```

把 Response 数据渲染交给返回值，并通过 Request `Content-Type` 来指定返回数据的格式

```go
type Info struct {
	ID uint `uri:"id" binding:"required"`
}

RoutePattern1(c *Context, info *Info) (post *Post, err error)
```

```go
type AuthInfo struct {
	Login    string `json:"login"    binding:"required"`
	Password string `json:"password" binding:"required"`
}

RoutePattern1(c *Context, info *AuthInfo) (account *Account, err error)
```

- 通过函数本身 `return` 来做流程控制，判断 `err` 是否为 `nil` 来决定是否继续执行
- 可以自定义 error 来决定 Response 的状态码

完整示例：

```go
type CreatePostsArgs struct {
	Title   string `json:"title"   binding:"required"`
	Content string `json:"content" binding:"required"`
}

type Post struct {
	Title     string `json:"title"      xml:"title"`
	Content   string `json:"content"    xml:"content"`
	CreatedAt int64  `json:"created_at" xml:"created_at"`
	UpdatedAt int64  `json:"updated_at" xml:"updated_at"`
}

func CreatePostsHandler(c *engine.Context, args *CreatePostsArgs)(post *Post, err error)
	if post, err = service.CreatePost(args); err != nil {
		c.Logger.Error(err.Error()) // 带了 X-Request-ID 信息的 Logger
		return
	}
	return post, nil
}

func main() {
	engine := engine.New()
	engine.POST("/posts", CreatePostsHandler)
	engine.Run()
}
```

如果把 Handler 返回值进一步抽象：

```go
type Response struct {
        Data interface{}
        Error error
        Code int
}

func CreatePostsHandler(c *engine.Context, args *CreatePostsArgs)(resp Response)
        if post, err := service.CreatePost(args); err != nil {
                c.Logger.Error(err.Error())
                resp.Error = err
                return
        }

        resp.Data = post
        resp.Code = http.StatusCreated

        return
}
```

讲了这么多，这个 Web Engine 该怎么实现呢？

以后再说，这个 PPT 就不讲了。

Q & A