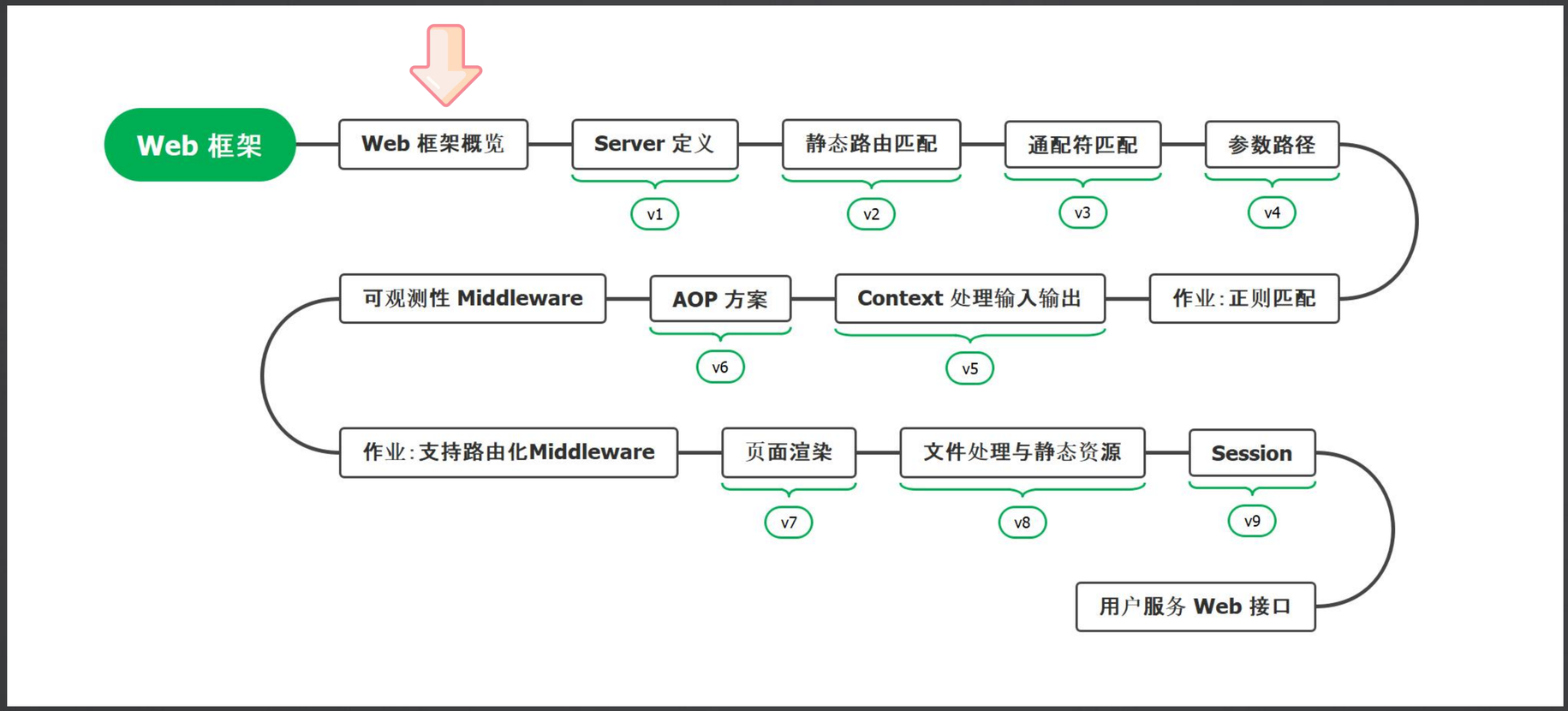
Web 模块——Web 框架概览

大明

学习路线——Web 框架概览

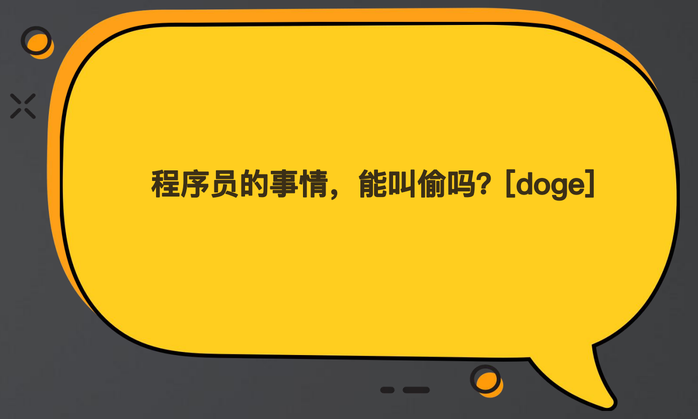


Web 框架概览

在我们开始学习新一类框架的时候，大多数时候只有一个模糊的印象，比如说我们对于 Web 框架的印象大概就是：

处理 HTTP 请求的

好在因为我们造的轮子，其实都不是从 0 到 1，所以我们可以通过参考其它 Web 框架来设计自己的框架。



Beego —— Controller 抽象

Beego 是基于 MVC (Model-View-Controller) 的，所以它定义了一个核心接口 ControllerInterface。ControllerInterface 定义了一个控制器必须要解决什么问题。

同时 ContollerInterface 的默认实现 Controller 提供了实现自定义控制器的各种辅助方法，所以在 Beego 里面，一般都是组合 Controller 来实现自己的 Controller。

```go
// ControllerInterface is an interface to unifo
type ControllerInterface interface {
    Init(ct *context.Context, controllerName, a
    Prepare()
    Get()
    Post()
    Delete()
    Put()
    Head()
    Patch()
    Options()
    Trace()
    Finish()
    Render() error
    XSRFToken() string
    CheckXSRFCookie() bool
    HandlerFunc(fn string) bool
    URLMapping()
}
```

Beego —— Controller 抽象

```go
// ControllerInterface is an interface to unifo
type ControllerInterface interface {
    Init(ct *context.Context, controllerName, a
    Prepare()
    Get()
    Post()
    Delete()
    Put()
    Head()
    Patch()
    Options()
    Trace()
    Finish()
    Render() error
    XSRFToken() string
    CheckXSRFCookie() bool
    HandlerFunc(fn string) bool
    URLMapping()
}
```

```go
type UserController struct {
    web.Controller
}

func (c *UserController) GetUser() {
    c.Ctx.WriteString( content: "你好，我是大明")
}
```

```go
func TestUserController(t *testing.T) {
    c := &UserController{}
    web.Router( rootpath: "/user", c, mappingMethods...: "get:GetUser")
    // 监听 8081 端口
    web.Run( params...: ":8081")
}
```

注意到用户虽然被要求组合 Controller，但是路由注册和服务器启动是通过另外一套机制来完成的。

Beego —— HttpServer 和 ControllerRegister

ControllerInterface 可以看做核心接口，因为它直接体现了 Beego 的设计初衷：MVC 模式。同时它也是用户核心接入点。

但是如果从功能特性上来说，HttpServer 和 ControllerRegister 才是核心。

- HttpServer：代表一个”服务器”，大多数时候它就是一个进程。
- ControllerRegister：真正干活的人。注册路由，路由匹配和执行业务代码都是透过它来完成的。

```
// HttpServer defines beego application with a
type HttpServer struct {
    Handlers            *ControllerRegister
    Server              *http.Server
    Cfg                 *Config
    LifeCycleCallbacks []LifeCycleCallback
}
```

```
// ControllerRegister containers registered router rules,
type ControllerRegister struct {...}
```

Beego —— Context 抽象

用户操作请求和响应是通过 Ctx 来达成的。它代表的是整个请求执行过程的上下文。

进一步，Beego 将 Context 细分了几个部分：

- Input：定义了很多和处理请求有关的方法
- Output：定义了很多和响应有关的方法
- Response：对 http.ResponseWriter 的二次封装

```go
// Context Http request context stru
// BeegoInput and BeegoOutput provid
type Context struct {
    Input          *BeegoInput
    Output         *BeegoOutput
    Request        *http.Request
    ResponseWriter *Response
    _xsrfToken     string
}
```

```go
func (c *UserController) CreateUser() {
    u := &User{}
    err := c.Ctx.BindJSON(u)
    if err != nil {
        c.Ctx.WriteString(err.Error())
        return
    }

    _ = c.Ctx.JSONResp(u)
}
```

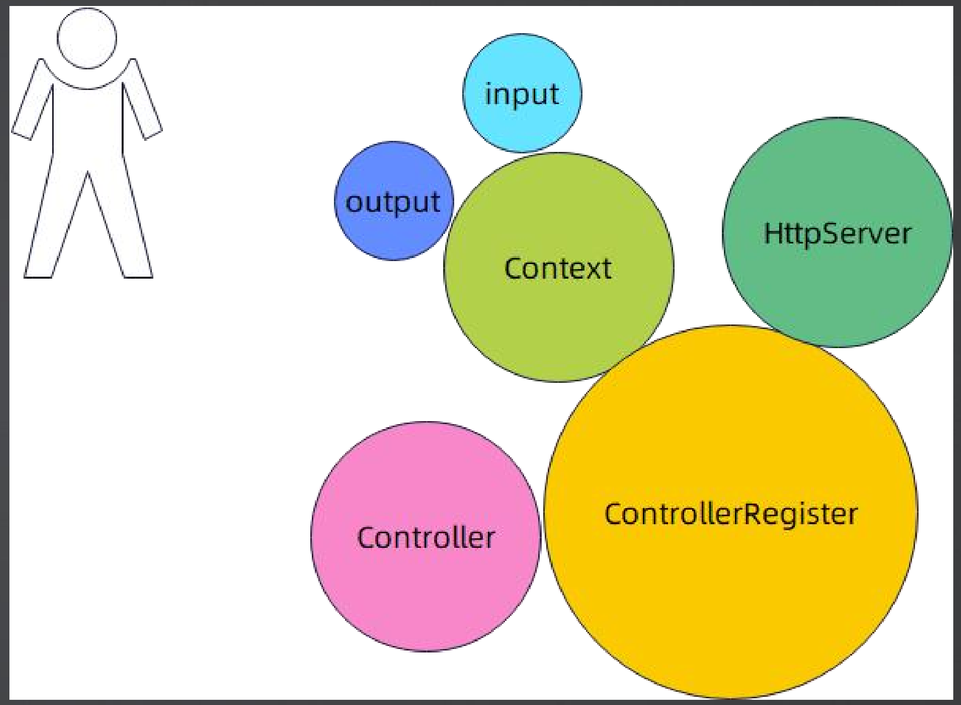
```go
// Controller defines some basic http requ
// http context, template and view, sessio
type Controller struct {
    // context data
    Ctx  *context.Context
    Data map[interface{}]interface{}
```

Beego —— 核心抽象总结

ControllerRegister 最为基础，它解决了路由注册和路由匹配这个基础问题。

Context 和 Controller 为用户提供了丰富 API，用于辅助构建系统。

HttpServer 作为服务器抽象，用于管理应用生命周期和资源隔离单位。



Gin —— IRoutes 接口

核心接口 IRoutes：提供的是注册路由的抽象。它的实现类 Engine 类似于 ControllerRegister。

Use 方法提供了用户接入自定义逻辑的能力，这个一般情况下也被看做是插件机制。

还额外提供了静态文件的接口。

Gin 没有 Controller 的抽象。所以在这方面我和 Gin 的倾向比较一致，即 MVC 应该是用户组织 Web 项目的模式，而不是我们中间件设计者要考虑的。

```go
// IRoutes defines all router handle interface.
type IRoutes interface {
    Use(...HandlerFunc) IRoutes

    Handle(string, string, ...HandlerFunc) IRoutes
    Any(string, ...HandlerFunc) IRoutes
    GET(string, ...HandlerFunc) IRoutes
    POST(string, ...HandlerFunc) IRoutes
    DELETE(string, ...HandlerFunc) IRoutes
    PATCH(string, ...HandlerFunc) IRoutes
    PUT(string, ...HandlerFunc) IRoutes
    OPTIONS(string, ...HandlerFunc) IRoutes
    HEAD(string, ...HandlerFunc) IRoutes

    StaticFile(string, string) IRoutes
    Static(string, string) IRoutes
    StaticFS(string, http.FileSystem) IRoutes
}
```

Gin —— Engine 实现

Engine 可以看做是 Beego 中 HttpServer 和 ControllerRegister 的合体。

- 实现了路由树功能，提供了注册和匹配路由的功能
- 它本身可以作为一个 Handler 传递到 http 包，用于启动服务器

Engine 的路由树功能本质上是依赖于 methodTree 的。

```
func setupRouter() *gin.Engine {
        // Disable Console Color
        // gin.DisableConsoleColor()
        r := gin.Default()

        // Ping test
        r.GET("/ping", func(c *gin.Context) {
                c.String(http.StatusOK, "pong")
        })

        // Get user value
        r.GET("/user/:name", func(c *gin.Context) {
                user := c.Params.ByName("name")
                value, ok := db[user]
                if ok {
                        c.JSON(http.StatusOK, gin.H{"user": user, "value":
                } else {
                        c.JSON(http.StatusOK, gin.H{"user": user, "status"
                }
        })
```

Gin —— methodTrees 和 methodTree

methodTree 才是真实的路由树。

Gin 定义了 methodTrees，它实际上代表的是森林，即每一个 HTTP 方法都对应到一棵树。

```
noRoute          HandlersChain
noMethod         HandlersChain
pool             sync.Pool
trees            methodTrees
maxParams        uint16
maxSections      uint16
trustedProxies   []string
trustedCIDRs     []*net.IPNet
```

Engine 字段

```
type methodTree struct {
    method string
    root   *node
}

type methodTrees []methodTree
```

Gin —— HandlerFunc 和 HandlersChain

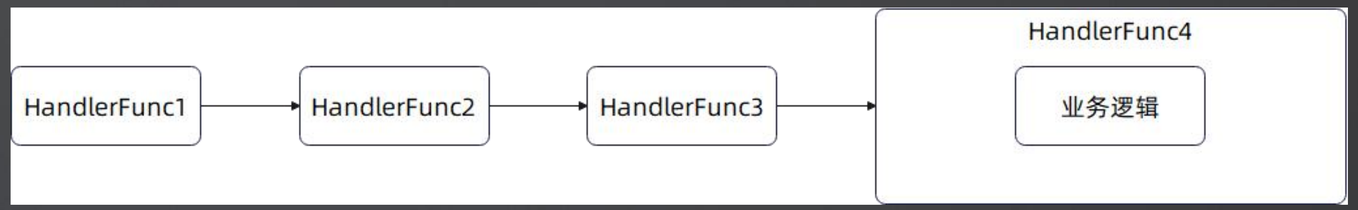
HandlerFunc 定义了核心抽象——处理逻辑。

在默认情况下， 它代表了注册路由的业务代码。

HandlersChain 则是构造了责任链模式。

```go
// HandlerFunc defines the handler used by gin mi
type HandlerFunc func(*Context)

// HandlersChain defines a HandlerFunc array.
type HandlersChain []HandlerFunc
```



最后一个则是封装了业务逻辑的 HandlerFunc

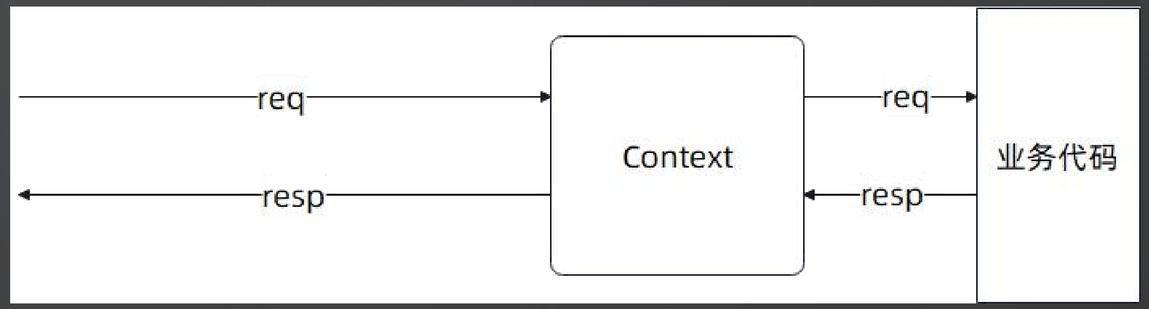
Gin —— Context 抽象

Context 也是代表了执行的上下文，提供了丰富的 API：

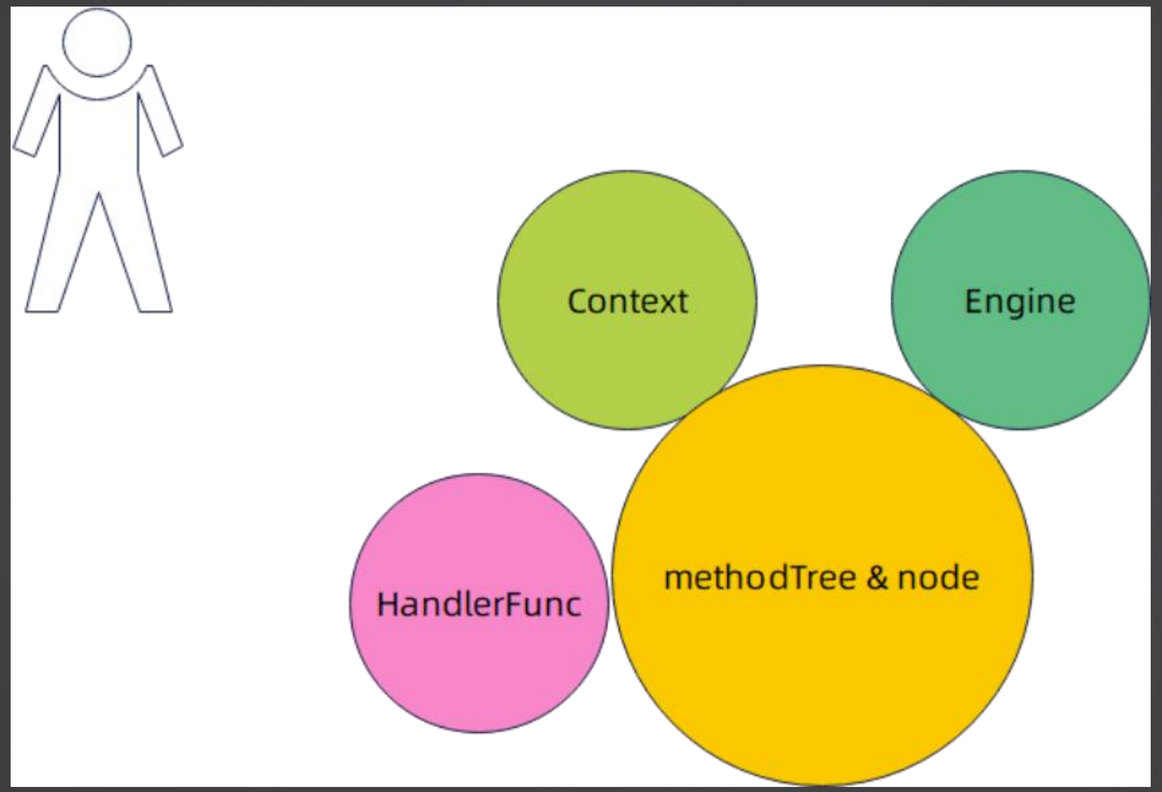
- 处理请求的 API，代表的是以 Get 和 Bind 为前缀的方法
- 处理响应的 API，例如返回 JSON 或者 XML 响应的方法
- 渲染页面，如 HTML 方法

```
// manage the flow, validate the JS
type Context struct {
    writermem responseWriter
    Request   *http.Request
    Writer    ResponseWriter

    Params   Params
    handlers HandlersChain
    index    int8
    fullPath string
```



Gin —— 抽象总结



Iris —— Application

https://github.com/kataras/iris

Application 是 Iris 的核心抽象，它代表的是”应用”。实际上这个语义更加接近 Beego 的 HttpServer 和 Gin 的 Engine。

```
func TestHelloWorld(t *testing.T) {
    app := iris.New()

    app.Get( relativePath: "/", func(ctx iris.Context)
        _, _ = ctx.HTML( format: "Hello <strong>%s</st
    })

    _ = app.Listen( hostPort: ":8083")
}
```

它提供了：

- 生命周期控制功能，如 Shutdown 等方法
- 注册路由的 API

个人认为 Application 这个名字不是很合适，比如说有些应用会监听多个端口。不同的端口提供不同的功能，也就是常说的多 Server 应用。相比之下，HttpServer 或者 Engine 会更合适一点。

Iris —— 路由相关

Iris 的设计非常复杂。在 Beego 和 Gin 里面能够明显看到路由树的痕迹，但是在 Iris 里面就很难看出来。

和处理路由相关的三个抽象：

- Route：直接代表了已经注册的路由。在 Beego 和 Gin 里面，对应的是路由树的节点
- APIBuilder：创建 Route 的 Builder 模式，Party 也是它创建的
- repository：存储了所有的 Routes，有点接近 Gin 的 methodTrees 的概念

个人观点：过于复杂，职责不清晰，不符合一般人的直觉，新人学习和维护门槛高。不要学。

```go
// repository passed to all parties(subrouters),
// all the routes.
type repository struct {
    routes []*Route
    pos    map[string]int
}
```

```go
// Route contains the information about a registered Route.
// If any of the following fields are changed then the
// caller should Refresh the router.
type Route struct {
    Name       string         `json:"name"`   // "userRoute
    Method     string         `json:"method"` // "GET"
```

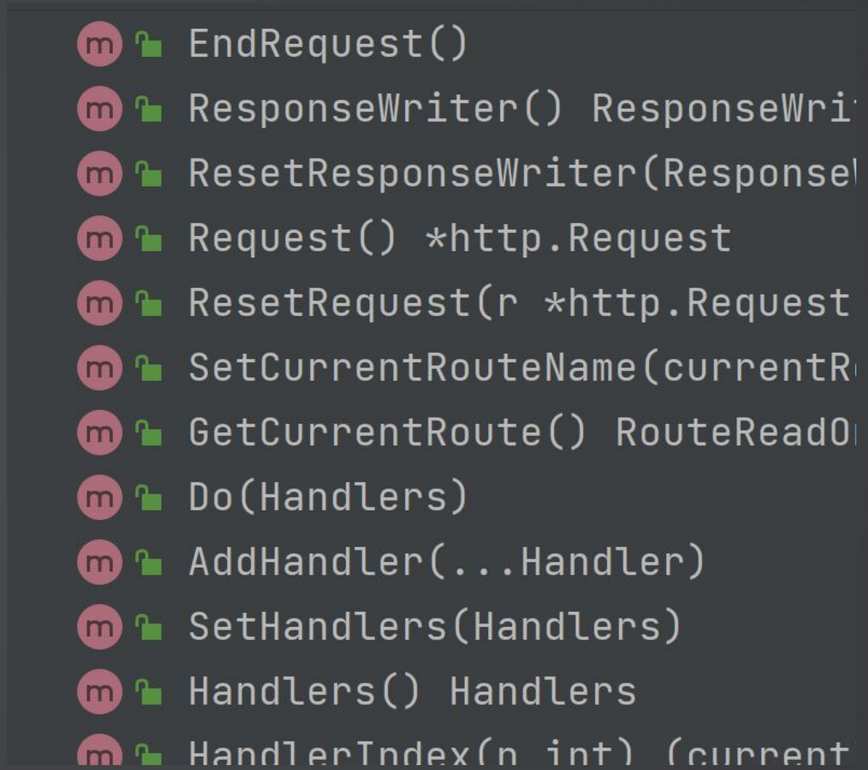
Iris —— Context 抽象

Context 也是代表上下文。

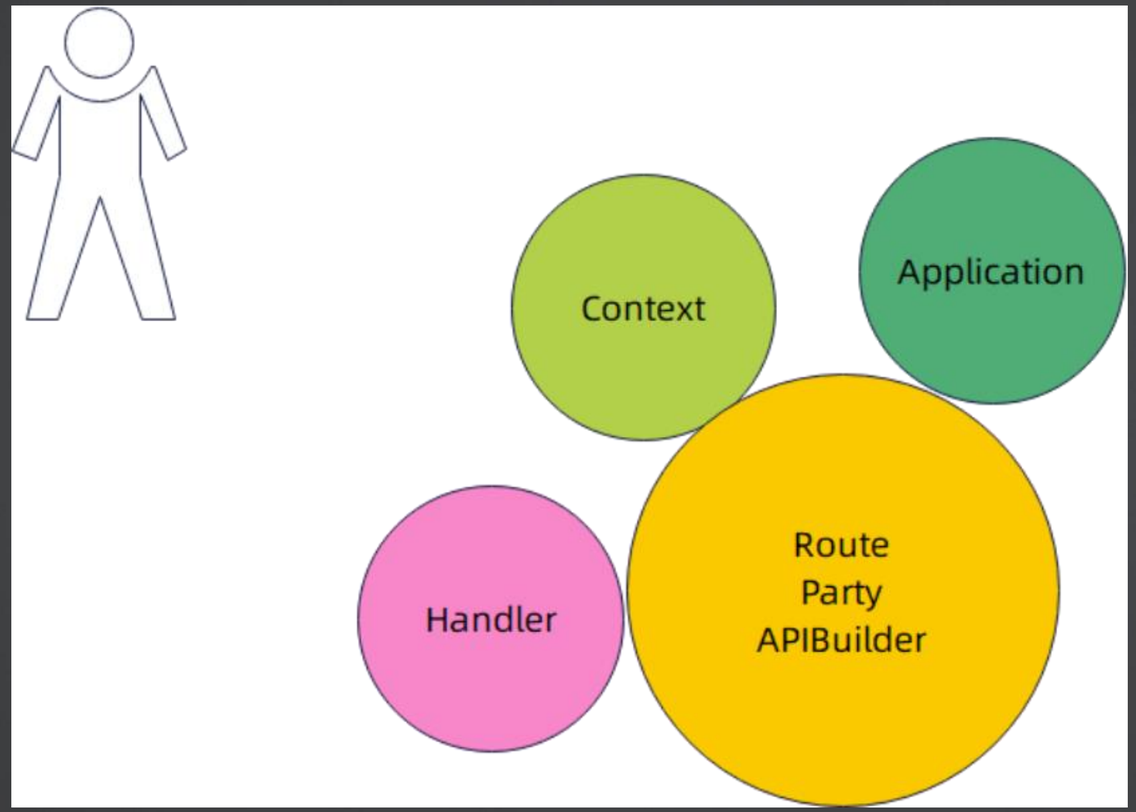
Context 本身也是提供了各种处理请求和响应的方法。

基本上和 Beego 和 Gin 的 Context 没啥区别。

比较有特色的是它的 Context 支持请求级别的添加 Handler，即 AddHandler 方法。



Iris —— 抽象总结



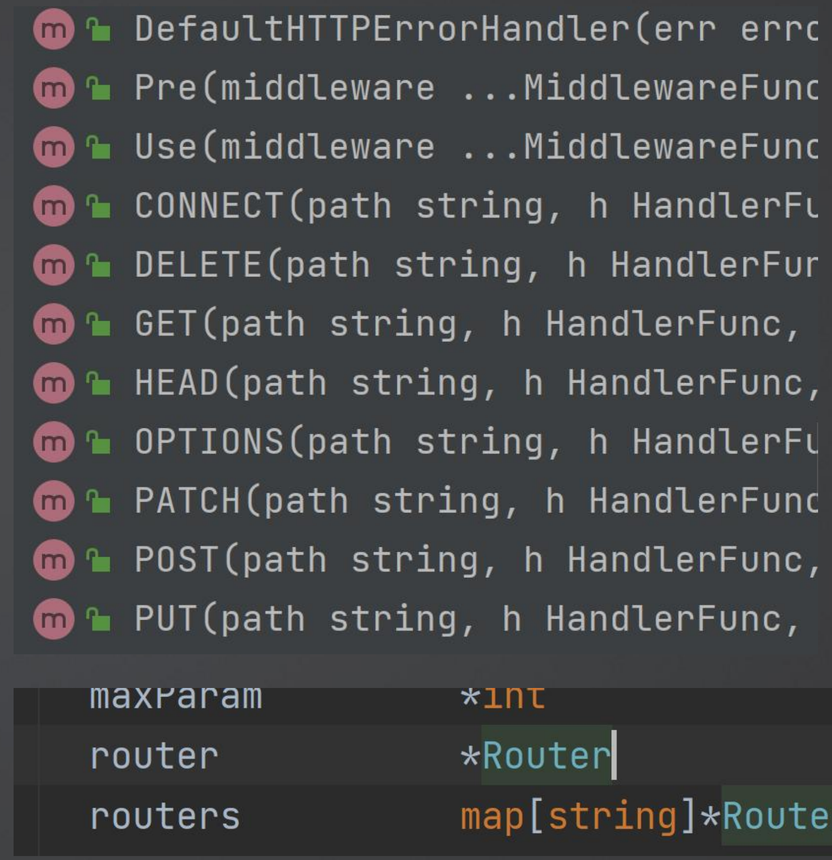
Echo —— Echo

Echo 是它内部的一个结构体，类似于 Beego 的 HttpServer 和 Gin 的 Engine：

- 暴露了注册路由的方法，但是它并不是路由树的载体
- 生命周期管理：如 Shutdown 和 Start 等方法

在 Echo 里面有两个相似的字段：

- router：这其实就是代表路由树
- routers：这代表的是根据 Host 来进行分组组织，可以看做是近似于 namespace 之类的概念，既是一种组织方式，也是一种隔离机制



Echo —— Route 和 node

Router 代表的就是路由树，node 代表的是路由树上的节点。

node 里面有一个很有意思的设计：staticChildren、paramChild 和 anyChild。利用这种设计可以轻松实现路由优先级和路由冲突检测。

它里面还有一个字段叫做 echo, 维护的是使用 Route 的 Echo 实例。这种设计形态在别的地方也能见到，比如说在 sql.Tx 里面维持了一个 sql.DB 的实例。

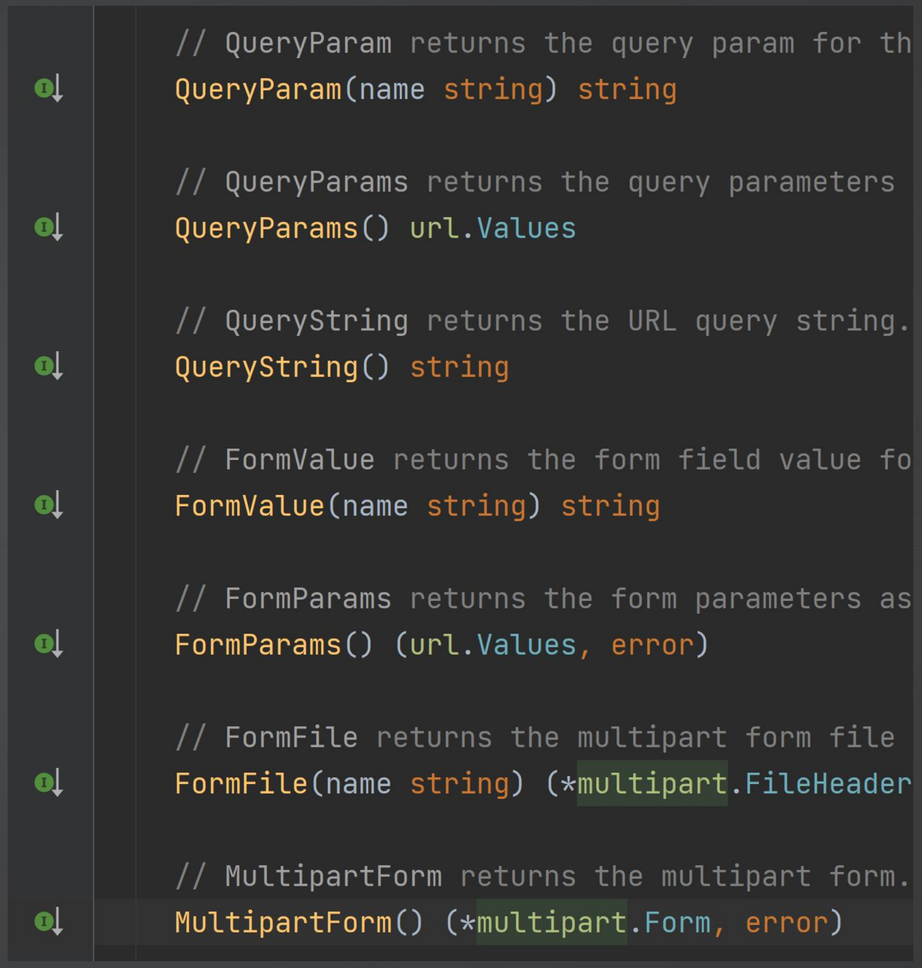
```
// Router is the registry of all r
// request matching and URL path p
Router struct {
    tree   *node
    routes map[string]*Route
    echo   *Echo
}
```

```
node struct {
    kind           kind
    label          byte
    prefix         string
    parent         *node
    staticChildren children
    ppath          string
    pnames         []string
    methodHandler  *methodHandler
    paramChild     *node
    anyChild       *node
    // isLeaf indicates that node does not
```

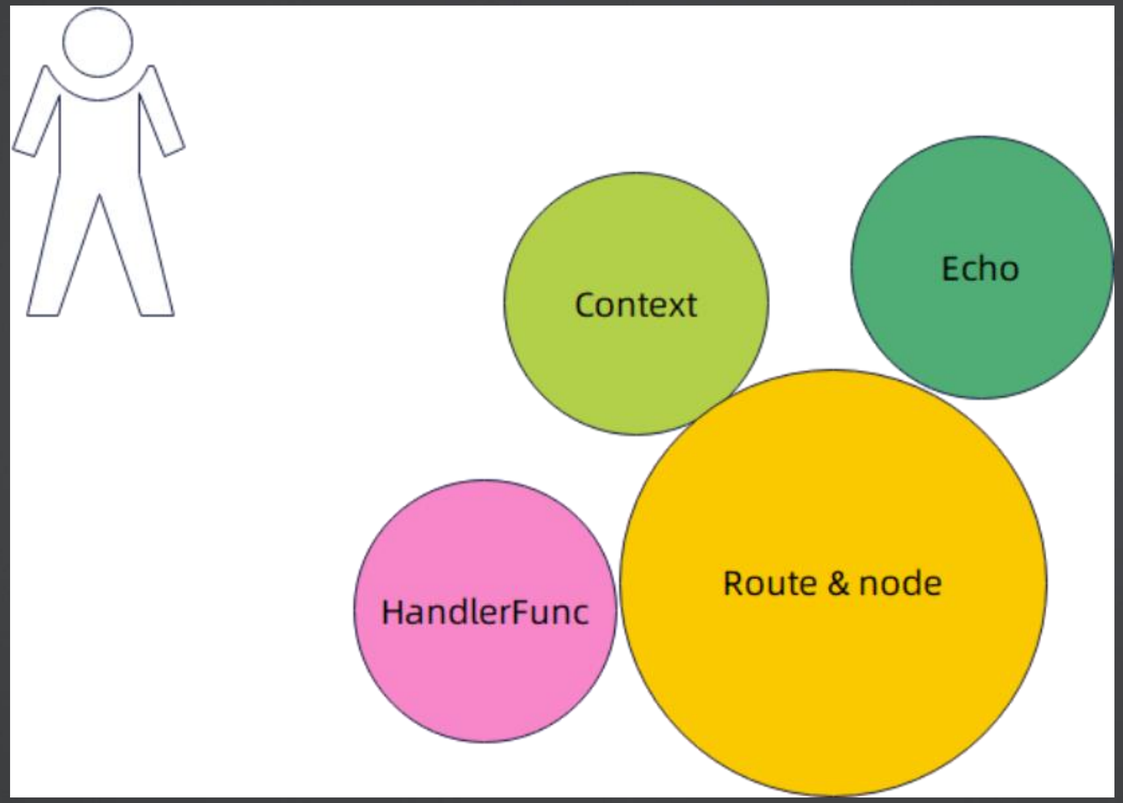
Echo —— Context

一个大而全的接口，定义了处理请求和响应的各种方法。

和 Beego、Gin、Iris 的 Context 没有什么区别。



Echo —— 核心抽象



框架对比

	Beego	Gin	Iris	Echo
代表服务器	HttpServer	Engine	Application	Echo
代表路由	ControllerRegister	methodTree	Route	Route
上下文	Context	Context	Context	Context
处理逻辑	HandlerFunc	HandlerFunc	HandlerFunc	HandlerFunc

所以实际上我们要造一个 Web 框架，就是要建立我们自己的这几个抽象。

Web 框架面试题

实际上，面整体 Web 框架的还是比较少的。大多数时候，面试都是聊具体的某个 Web 框架。

从整体上来面的话，比较高频率的问题：

- Web 框架拿来做什么？处理 HTTP 请求，为用户提供便捷 API，为用户提供无侵入式的插件机制，提供如上传下载等默认功能
- 为什么都已经有了 http 包，还要开发 Web 框架？高级路由功能、封装 HTTP 上下文以提供简单 API、封装 Server 以提供生命周期控制、设计插件机制以提供无侵入式解决方案
- Web 框架的核心？路由树、上下文 Context、Server（按照我理解的重要性排序）

更多面试题在接下来的课程内容里。

Q & A

THANKS