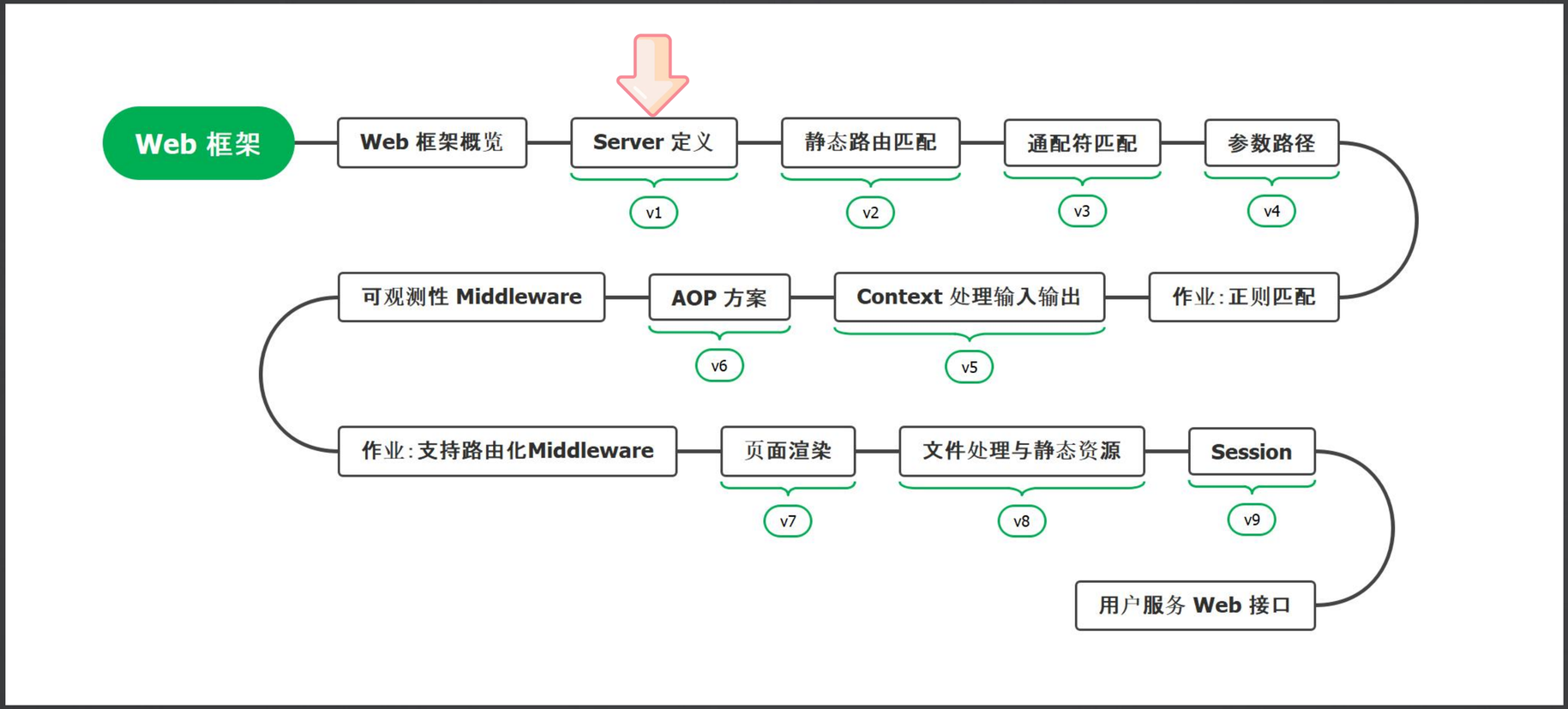
Web 模块——Server

大明

学习路线—— Server 定义



Web 核心

在框架对比的时候，我们注意到对于一个 Web 框架来说，至少要提供三个抽象：

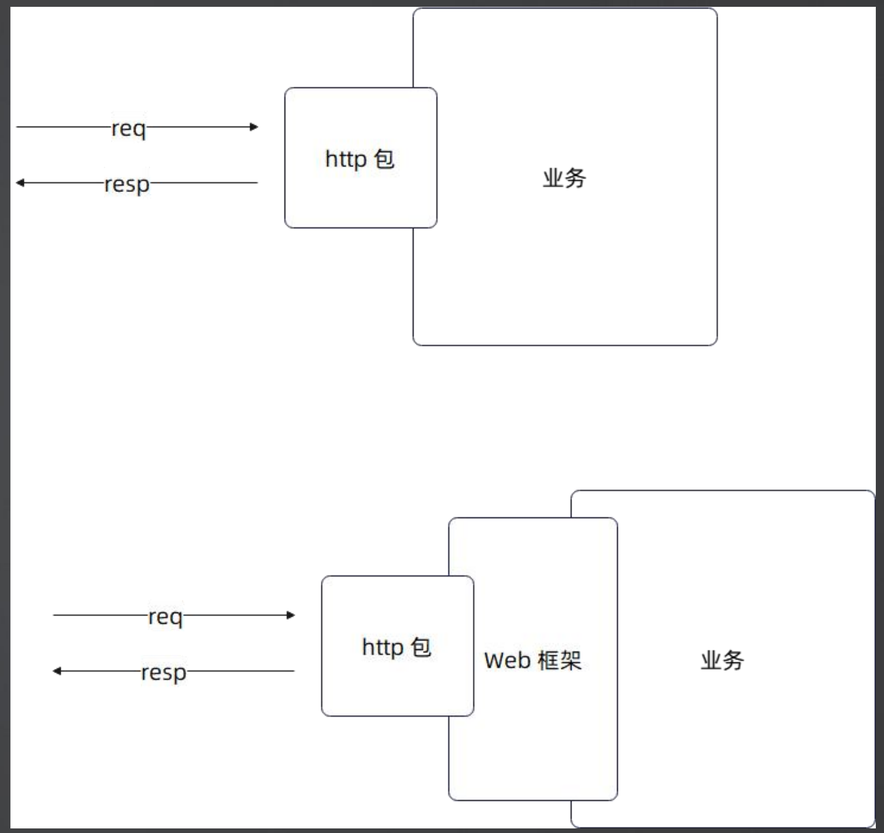
- 代表服务器的抽象，这里我们称之为 Server
- 代表上下文的抽象，这里我们称之为 Context
- 路由树

Web 核心 —— Server

从前面框架对比来看，对于一个 Web 框架来说，我们首先要有一个整体代表服务器的抽象，也就是 Server。

Server 从特性上来说，至少要提供三部分功能：

- 生命周期控制：即启动、关闭。如果在后期，我们还要考虑增加生命周期回调特性
- 路由注册接口：提供路由注册功能
- 作为 http 包到 Web 框架的桥梁



Server —— http.Handler 接口

http 包暴露了一个接口，Handler。

它是我们引入自定义 Web 框架相关的连接点。

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

Server —— 接口定义

Server 定义版本一：只组合 http.Handler。

优点：

- 用户在使用的时候只需要调用 http.ListenAndServe 就可以
- 和 HTTPS 协议完全无缝衔接
- 极简设计

缺点：

- 难以控制生命周期，并且在控制生命周期的时候增加回调支持
- 缺乏控制力：如果将来希望支持优雅退出的功能，将难以支持

```go
type Server interface {
    http.Handler
}
```

```go
func TestServer(t *testing.T) {
    var s Server
    http.ListenAndServe( addr: ":8080", s)

    http.ListenAndServeTLS( addr: ":4000",
        certFile: "cret file", keyFile: "key file", s)
}
```

Server —— 接口定义

Server 定义版本二：组合 http.Handler 并且增加 Start 方法。

```go
type Server interface {
    http.Handler
    Start(addr string) error
}
```

优点：

- Server 既可以当成普通的 http.Handler 来使用，又可以作为一个独立的实体，拥有自己的管理生命周期的能力
- 完全的控制，可以为所欲为

缺点：

- 如果用户不希望使用 ListenAndServeTLS，那么 Server 需要提供 HTTPS 的支持

```go
func TestServer(t *testing.T) {
    var s Server
    http.ListenAndServe(addr: ":8080", s)

    http.ListenAndServeTLS(addr: ":4000",
        certFile: "cret file", keyFile: "key file", s)

    s.Start(addr: ":8081")
}
```

注意：Start 方法可以不需要 addr 参数，那么在创建实现类的时候传入地址就可以。

版本一和版本二都直接耦合了 Go 自带的 http 包，如果我们希望切换为 fasthttp 或者类似的 http 包，则会非常困难。

Server —— HTTPServer 实现

该实现直接使用 http.ListenAndServe 来启动，后续可以根据需要替换为：

- 内部创建 http.Server 来启动
- 使用 http.Serve 来启动，换取更大的灵活性，如将端口监听和服务器启动分离等

ServeHTTP 则是我们整个 Web 框架的核心入口。我们将在整个方法内部完成：

- Context 构建
- 路由匹配
- 执行业务逻辑

```go
// 确保 HTTPServer 肯定实现了 Server 接口
var _ Server = &HTTPServer{}

type HTTPServer struct {
}

// ServeHTTP HTTPServer 处理请求的入口
func (s *HTTPServer) ServeHTTP(writer http.ResponseWriter, request *http.Request) {

}

func (s *HTTPServer) Start(addr string) error {
    return http.ListenAndServe(addr, s)
}
```

```go
//
// Serve always returns a non-nil error.
func Serve(l net.Listener, handler Handler) error {
    srv := &Server{Handler: handler}
    return srv.Serve(l)
}
```

Server —— 注册路由 API 设计

暂时我们可以考虑先站在用户的角度，考虑如何注册路由。

```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

```
None(relativePath string, handle
Get(relativePath string, handler
Post(relativePath string, handle
Put(relativePath string, handler
Delete(relativePath string, hand
Connect(relativePath string, han
Head(relativePath string, handle
Options(relativePath string, han
Patch(relativePath string, handl
```

```go
// Returns a *Route, app will throw any errors later on.
func (api *APIBuilder) Handle(method string, relativePath string,
```

Iris

```
Pre(middleware ...Midd
Use(middleware ...Midd
CONNECT(path string, h
DELETE(path string, h
GET(path string, h Hanc
HEAD(path string, h Har
OPTIONS(path string, h
PATCH(path string, h Ha
POST(path string, h Har
PUT(path string, h Hanc
```

```go
// Add registers a new route for an HTTP method and path wit
// in the router with optional route-level middleware.
func (e *Echo) Add(method, path string, handler HandlerFunc,
```

Echo

Server —— 注册路由 API 设计

大体上有两类方法：

- 针对任意方法的：如 Gin 和 Iris 的 Handle 方法、Echo 的 Add 方法
- 针对不同 HTTP 方法的：如 Get、POST、Delete，这一类方法基本上都是委托给前一类方法

```go
// PUT is a shortcut for router.Handle("PUT", path, handle).
func (group *RouterGroup) PUT(relativePath string, handlers ...HandlerFunc) IRoutes {
    return group.handle(http.MethodPut, relativePath, handlers)
}
```

Gin

```go
func (e *Echo) DELETE(path string, h HandlerFunc, m ...MiddlewareFunc) *Route {
  return e.Add(http.MethodDelete, path, h, m...)
}
```

Echo

```go
// Returns a *Route and an error which will be filled if route wasn't registered succes
func (api *APIBuilder) Get(relativePath string, handlers ...context.Handler) *Route {
    return api.Handle(http.MethodGet, relativePath, handlers...)
}
```

Iris

所以实际上，核心方法只需要有一个，例如 Handle。其它的方法都建立在这上面。

Server —— AddRoute 方法

思考点：

- AddRoute 方法只接收一个 HandleFunc。因为我希望它只注册业务逻辑。即便真有多个的场景，用户可以自己组合成一个。
- 如果允许注册多个，那么在实现的时候就要考虑，其中一个失败了，是否还允许继续执行下去；反过来，如果其中一个 HandleFunc 要中断执行，怎么中断。
- 这里我采用了新的名字 AddRoute，我认为这更加贴近这个方法本意。
  - Handle：看上去像是处理什么东西，而实质上我们这里只是注册路由，所以用 AddRoute 会更加合适

```go
type Server interface {
    http.Handler
    // Start 启动服务器
    // addr 是监听地址。如果只指定端口，可以使用 ":8081"
    // 或者 "localhost:8082"
    Start(addr string) error
    // AddRoute 注册一个路由
    // method 是 HTTP 方法
    // path 是路径，必须以 / 为开头
    AddRoute(method string, path string, handler HandleFunc)
    // 我们并不采取这种设计方案
    // AddRoute(method string, path string, handlers... HandleFunc)
}
```

```go
type HandleFunc func(ctx *Context)
```

这里我们叫做 HandleFunc 而不是 HandlerFunc，采用动词 Handle 更加符合 Go 的命名风格。

Server —— AddRoute 方法

```go
type Server interface {
    http.Handler
    // Start 启动服务器
    // addr 是监听地址。如果只指定端口，可以使用 ":8081"
    // 或者 "localhost:8082"
    Start(addr string) error
    // AddRoute 注册一个路由
    // method 是 HTTP 方法
    // path 是路径，必须以 / 为开头
    AddRoute(method string, path string, handler HandleFunc)
    // 我们并不采取这种设计方案
    // AddRoute(method string, path string, handlers... HandleFunc)
}
```

```go
// PUT is a shortcut for router.Handle("PUT", path, handle).
func (group *RouterGroup) PUT(relativePath string, handlers ...HandlerFunc) IRoutes {
    return group.handle(http.MethodPut, relativePath, handlers)
}
```

Gin

```go
func (e *Echo) DELETE(path string, h HandlerFunc, m ...MiddlewareFunc) *Route {
    return e.Add(http.MethodDelete, path, h, m...)
}
```

Echo

```go
// Returns a *Route and an error which will be filled if route wasn't registered succes
func (api *APIBuilder) Get(relativePath string, handlers ...context.Handler) *Route {
    return api.Handle(http.MethodGet, relativePath, handlers...)
}
```

Iris

其它三个框架都是允许注册多个，但其实用起来体验不会很好。

- Gin 和 Iris 最后一个是不定参数，那么完全可以一个都不传，如 PUT(“path”)。这个在编译期无法发现
- Echo 则是存在我希望 h 传入 nil 的可能。实际上 Echo 是将中间件注册逻辑和路由注册逻辑合并在了一起

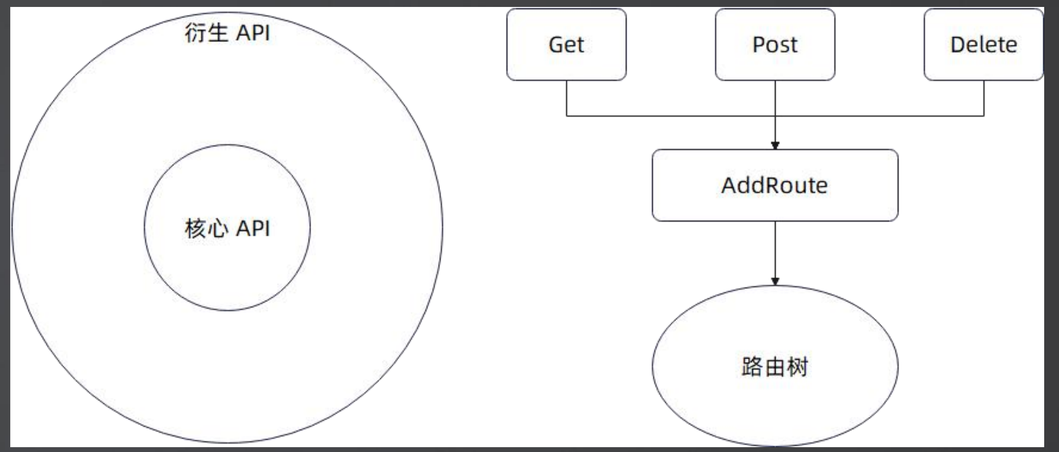
Server —— AddRoute 衍生方法

针对不同 HTTP 方法的注册 API，都可以委托给 Handle 方法。这种设计思路很常用。

```go
func (s *HTTPServer) Post(path string, handler HandleFunc) {
    s.AddRoute(http.MethodPost, path, handler)
}

func (s *HTTPServer) Get(path string, handler HandleFunc) {
    s.AddRoute(http.MethodGet, path, handler)
}

func (s *HTTPServer) AddRoute(method string, path string, handler HandleFunc) {
    panic( v: "implement me")
}
```



AddRoute 最终会和路由树交互，我们后面再考虑。

Server —— ServeHTTP 方法

ServeHTTP 方法是作为 http 包与 Web 框架的关联点，需要在 ServeHTTP 内部，执行：

- 构建起 Web 框架的上下文
- 查找路由树，并执行命中路由的代码

```go
type Context struct {
    Req  *http.Request
    Resp http.ResponseWriter
}
```

```go
// ServeHTTP HTTPServer 处理请求的入口
func (s *HTTPServer) ServeHTTP(writer http.ResponseWriter, request *http.Request) {
    ctx := &Context{
        Req:  request,
        Resp: writer,
    }
    s.serve(ctx)    查找路由，执行代码
}
```

到目前为止，代码放在 v1 里面。

面试要点

- HTTP 服务器的生命周期？一般来说就是启动、运行和关闭。在这三个阶段的前后都可以插入生命周期回调。一般来说，面试生命周期，多半都是为了问生命周期回调。例如说怎么做 Web 服务的服务发现？就是利用生命周期回调的启动后回调，将 Web 服务注册到服务中心。
- HTTP Server 功能？记住在不同的框架里面有不同的叫法，比如说在 Gin 里面叫做 Engine，它们的基本功能都是提供路由注册，生命周期控制以及作为与 http 包结合的桥梁。

Q & A

THANKS