手把手带你写一个 Web 框架

go doc net/http | grep "^func"

命令行能查询出 net/http 库所有的对外库函数：

第一层，标准库创建 HTTP 服务是通过创建一个 Server 数据结构完成的；
第二层，Server 数据结构在 for 循环中不断监听每一个连接；

第三层，每个连接默认开启一个 Goroutine 为其服务；

第四、五层，serverHandler 结构代表请求对应的处理逻辑，并且通过这个结构进行具体业务逻辑处理；

第六层，Server 数据结构如果没有设置处理函数 Handler，默认使用 DefaultServerMux 处理请求；

第七层，DefaultServerMux 是使用 map 结构来存储和查找路由规则。

其实在 Golang 的设计中，每个 Goroutine 都是独立存在的，父 Goroutine 一旦使用 Go 关键字开启了一个子 Goroutine，父子 Goroutine 就是平等存在的，他们互相不能干扰。而在异常面前，所有 Goroutine 的异常都需要自己管理，不会存在父 Goroutine 捕获子 Goroutine 异常的操作。

每个goroutine都需要做好自己的异常捕获，不然任意一处panic就会导致整个进程崩溃！

需求 1：HTTP 方法匹配

需求 2：静态路由匹配

需求 3：批量通用前缀

需求 4：动态路由匹配

动态路由的匹配:

如果你对算法比较熟悉，会联想到这个问题本质是一个字符串匹配，而字符串匹配，比较通用的高效方法就是字典树，也叫 trie 树。

把超时控制做成中间件：

func TimeoutHandler(fun ControllerHandler, d time.Duration) ControllerHandler {

}.

对pipeline进行设置的位置包括 core、group和单个路由

常见Body内容格式

JSON、XML、JSONP、HTML；

开发比较完整的功能模块：先定义接口，再具体实现。

通过JSONP，回调函数实现跨域访问

如何优雅的关闭,主要关注的两个问题:

- 如何控制关闭进程的操作” 和 “如何等待所有逻辑都处理结束”。

信号	操作	是否可处理
SIGINT	Ctrl+C	可以
SIGQUIT	Ctrl+\	可以
SIGTERM	kill	可以
SIGKILL	kill -9	不可以

极客时间

```
quit := make(chan os.Signal)
```

ticker 是 Golang 中最优的定时写法。(比sleep好)

衡量一个框架的评判标准：

标准	说明
核心模块	服务启动方式、路由分发机制是怎么样的 上下文context封装性如何、中间件机制是怎么设计
功能完备性	是否有提供日志模块、是否有提供命令行工具、是否有提供缓存机制等
框架扩展性	需要扩展某个功能，是否改动较大、是否支持功能实现的可插拔
框架性能	框架每秒能支持多少请求、是否有性能问题
文档完备度/社区活跃度	是否有完善的文档支持、社区是否足够活跃、咨询问题多久能得到回复

极客时间

最应该考虑功能完备性，明显使用 Beego 的收益会远远大于使用 Gin 和 Echo。框架扩展性放在最高级。

更优的recover机制：

区分了网络连接错误的异常和普通的逻辑异常，并且进行了不同的处理逻辑

golang中字节数组和字符串转换优化：

> 直接通过string关键字转换，会先在内存空间中重新开辟一段字符串空间，然后将 byte 数组复制进入这个字符串空间

```
package bytesconv

// 字符串转化为字符数组，不需要创建新的内存
func StringToBytes(s string) []byte {
  return *(*[]byte)(unsafe.Pointer(
    &struct {
      string
      Cap int
    }{s, len(s)},
  ))
}

// 字符数组转换为字符串，不需要创建新的内存
func BytesToString(b []byte) string {
  return *(*string)(unsafe.Pointer(&b))
}
```

最主流的开源许可证有 6 种：Apache、BSD、GPL、LGPL、MIT、Mozillia

定制使用第三方库的方法：对于很强的定制第三方库的需求，我们只能选择复制源码的方式。

面向接口编程的思想：

抽象业务 、 屏蔽具体实现

面向接口 / 对象 / 过程：

“面向过程编程”是指进行业务抽象的时候，我们定义一个一个的过程方法，通过这些过程方法的串联完成具体的业务。

“面向对象编程”表示的是在业务抽象的时候，我们先定义业务中的对象，通过这些对象之间的关联来表示整个业务。

面对业务，我们并不先定义具体的对象、思考对象有哪些属性，而是先思考如何抽象接口，把接口的定义放在第一步，然后多个模块之间梳理如何通过接口进行交互，最后才是实现具体的模块。

服务提供者的接口定义：

- 获取服务凭证的能力 Name；
- 创建服务实例化方法的能力 Register；
- 获取服务实例化方法参数的能力 Params；
- 两个与实例化控制相关的方法，控制实例化时机方法 IsDefer、实例化预处理的方法 Boot。

```
package framework

// NewInstance 定义了如何创建一个新实例，所有服务容器的创建服务
type NewInstance func(...interface{}) (interface{}, error){}

// ServiceProvider 定义一个服务提供者需要实现的接口
type ServiceProvider interface {
  // Register 在服务容器中注册了一个实例化服务的方法，是否在注册的时候就实例化这个服务，需要参考 IsDefer 接口。
  Register(Container) NewInstance
  // Boot 在调用实例化服务的时候会调用，可以把一些准备工作：基础配置，初始化参数的操作放在这个里面。
  // 如果 Boot 返回 error，整个服务实例化就会实例化失败，返回错误
  Boot(Container) error
  // IsDefer 决定是否在注册的时候实例化这个服务，如果不是注册的时候实例化，那就是在第一次 make 的时候进行实例化操作
  // false 表示不需要延迟实例化，在注册的时候就实例化。true 表示延迟实例化
  IsDefer() bool
  // Params params 定义传递给 NewInstance 的参数，可以自定义多个，建议将 container 作为第一个参数
  Params(Container) []interface{}
  // Name 代表了这个服务提供者的凭证
  Name() string
}
```

一个服务容器主要的功能是：为服务提供注册绑定、提供获取服务实例

.

服务容器实现主要方法: Build + Make

服务容器的优点：设计的拓展性非常好，之后在实际业务中我们只要保证服务协议不变，而不用担心具体的某个服务实现进行了变化

目录设计原则：

业务的目录结构也是一个服务，是一个应用目录服务。在这个服务中，我们制定框架要求的最小化的工程化规范，即框架要求业务至少有哪些目录结构。

golang cobra库使用：

```
// InitFoo 初始化 Foo 命令
func InitFoo() *cobra.Command {
   FooCommand.AddCommand(Foo1Command)
   return FooCommand
}
// FooCommand 代表 Foo 命令
var FooCommand = &cobra.Command{
   Use:     "foo",
   Short:   "foo 的简要说明",
   Long:    "foo 的长说明",
   Aliases: []string{"fo", "f"},
   Example: "foo 命令的例子",
   RunE: func(c *cobra.Command, args []string) error {
      container := c.GetContainer()
      log.Println(container)
      return nil
   },
}
// Foo1Command 代表 Foo 命令的子命令 Foo1
var Foo1Command = &cobra.Command{
   Use:     "foo1",
   Short:   "foo1 的简要说明",
   Long:    "foo1 的长说明",
   Aliases: []string{"fo1", "f1"},
   Example: "foo1 命令的例子",
   RunE: func(c *cobra.Command, args []string) error {
      container := c.GetContainer()
      log.Println(container)
      return nil
   },
}
```

cron的使用：

```
// 创建一个cron实例
c := cron.New()

// 每整点30分钟执行一次
```

```
c.AddFunc("30 * * * *", func() {
  fmt.Println("Every hour on the half hour")
})
// 上午3-6点，下午8-11点的30分钟执行
c.AddFunc("30 3-6,20-23 * * *", func() {
  fmt.Println(".. in the range 3-6am, 8-11pm")
})
// 东京时间4:30执行一次
c.AddFunc("CRON_TZ=Asia/Tokyo 30 04 * * *", func() {
  fmt.Println("Runs at 04:30 Tokyo time every day")
})
// 从现在开始每小时执行一次
c.AddFunc("@hourly",      func() {
  fmt.Println("Every hour, starting an hour from now")
})
// 从现在开始，每一个半小时执行一次
c.AddFunc("@every 1h30m", func() {
  fmt.Println("Every hour thirty, starting an hour thirty from now")
})

// 启动cron
c.Start()
```

Golang中启动子进程方式:

1. 因为在 Golang 中，fork 可以启动一个子进程，但是这个子进程是无法继承父进程的调度模型的。Golang 的调度模型是在用户态的 runtime 中自己进行调度的，而系统调用 fork 出来的子进程默认只会有单线程。所以在 Golang 中尽量不要使用 fork 的方式来复制启动当前进程。
2. 另一个办法是使用 os.StartProcess 来启动一个进程，执行当前进程相同的二进制文件以及当前进程相同的参数.开源库 go-daemon。

分布式定时器的实现方法:

通过抢占本地文件锁的方式实现

// 尝试独占文件锁 err = syscall.Flock(int(lock.Fd()), syscall.LOCK_EX|syscall.LOCK_NB)

获取配置的方式:

读取本地配置文件、读取远端配置服务、获取环境变量

常见配置的文件格式:

INI XML Properties YAML

`go-yaml`

配置文件和环境变量结合方法： 笔者在文件中配置env(XXX)读取后再用环境变量替换

配置文件热更新的实现方式:

可以自动监控配置文件目录下的所有文件，当配置文件有修改和更新的时候，能自动更新程序中的配置文件信息，也就是实现配置文件热更新。

使用 fsnotify 库对文件夹进行监控。

日志级别分类：

级别	描述
OFF	最高级别，用于关闭日志记录。
FATAL	导致应用程序提前终止的严重错误。一般这些信息将立即呈现在状态控制台上。
ERROR	其他运行时错误或意外情况。一般这些信息将立即呈现在状态控制台上。
WARN	使用已过时的API，API的滥用，潜在错误，其他不良的或意外的运行时的状况（但不一定是错误的）。 一般这些信息将立即呈现在状态控制台上。
INFO	令人感兴趣的运行时事件（启动/关闭）。 一般这些信息将立即呈现在状态控制台上。因而要保守使用，并保持到最低限度。
DEBUG	流经系统的详细信息。一般这些信息只记录到日志文件中。
TRACE	最详细的信息。一般这些信息只记录到日志文件中。自版本1.2.12。

日志常见几种输出方式：

- 控制台输出本地单个日志文件
- 输出本地单个日志文件
- 自动进行切割输出
- 自定义输出

类型嵌套对接口实现的优化：

```
// HadeConsoleLog 代表控制台输出
type HadeConsoleLog struct {
    // 类型嵌套HadeLog
    HadeLog
}
```

file-rotatelogs 进行文件切割实现滚动文件

前后端一体化改造方法：

将 Vue 的项目代码集成到业务代码中，然后确定其编译结果文件夹，在路由中，将某个请求路由到我们的编译结果文件夹，就完成了上面架构提到的同一个项目同时支持前端请求和后端请求了。

```
ue-init webpack hade
```

把请求路由到结果文件夹的方法；

所以，顺序应该是先看静态文件在 /dist 文件夹中是否存在，如果存在则返回静态文件，如果不存在，则访问动态请求。Gin的中间件 static 实现这个功能。

前后端一体化编译命令改造：

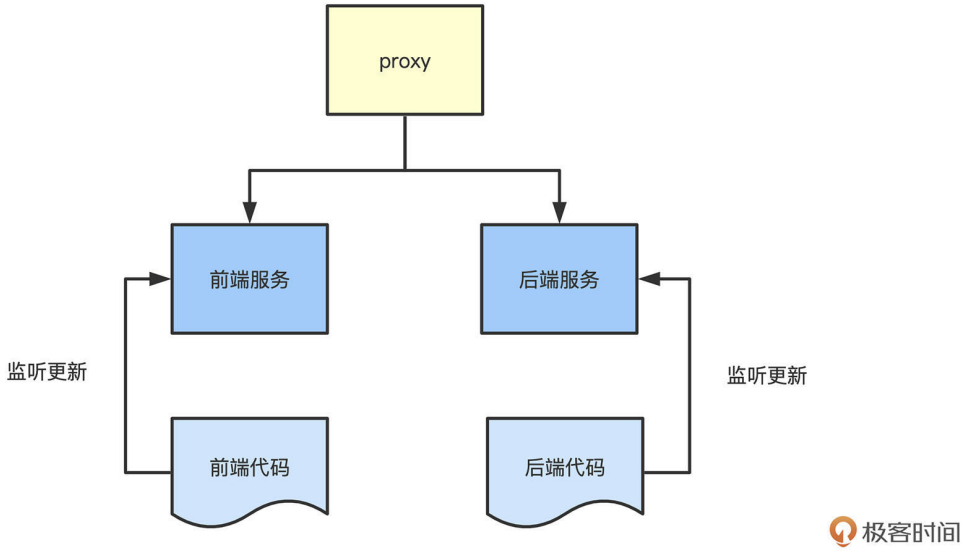
笔者通过Command注册。代码调用 exec.Command(path, "run", "build")方式封装前后端一体化编译命令；可执行：

```
编译前端   ./hade build frontend
编译后端   ./hade build backend
同时编译前后端 ./hade build all
自编译 ./hade build self
```

Go现有调试模式:

只能先通过 go build 编译出二进制文件，通过运行二进制文件再启动服务

后端动态编译调试模式实现方案:



只有两个服务都启动了，我们才进行上一节课说的：先请求后端服务，遇到 404 了，再请求前端服务

监控后端文件变化的方法:

使用 fsnotify 库对目录进行监控

优秀程序员应该有三大美德：懒惰、急躁和傲慢

创建自动化生成Provider的方法:

第三方库 survey进行交互输入；

text/template 库创建预定义模板文件方法

自动化创建命令行工具：

text/template 通过模板文件创建cmd类

swagger介绍:

REST 接口调试和文档解决方案

swagger-editor提供一个开源网站，在线编辑 swagger 文件。swagger-codegen提供一个 Java 命令行工具，通过 swagger 文件生成 client 端代码。而swagger-ui，通过提供一个开源网站，将 swagger 接口在线展示出来，并且可以让调用者查看、调试。

Golang自动集成库swag介绍：

1. 在 API 接口中编写注释。注释的详细写法需要参考说明文档。
2. 下载 swag 工具或者安装 swag 库
3. 使用工具或者库将指定代码生成 swagger.json