# Golang 语法 —— init 方法

- 无参数
- 无返回值
- 包被引入的时候执行，**只执行一次！**
- 执行顺序不定（目前是按照所在文件名排序）
- 可以有多个（一个包，一个文件内都可以有多个）

```go
// CfgMap 服务名到Config的映射
var CfgMap map[string]*Config
func init() {
    // CfgMap = make(map[string]Config) 没有指定容量，采用默认容量
    CfgMap = make(map[string]*Config, 4) // 4 是容量
    CfgMap["hello"] =&Config{
        Endpoint: "http://localhost:8080",
    }
}
```

```go
29  func init() {
30      // 第一个
31  }
32
33  func init() {
34      // 第二个
35  }
```

# Golang 语法 —— init 方法

如果我要保证 init 方法按照顺序执行，该如何做？

—— 用一个 init 方法，里面调用真实的逻辑

```go
// init.go
// 强制要求所有人把自己的有依赖关系的 init 方法放在这里
func init() {
    initBeforeSomething()
    initSomething()
    initAfterSomething()
}

func initBeforeSomething() {

}

func initSomething()  {

}

func initAfterSomething()  {

}
```

# RPC —— 获得配置 endpoint

- 要求所有的服务，都实现我们的 Service 接口
- 找不到配置的时候，返回一个错误

```
func SetFuncField(val Service) {

    v := reflect.ValueOf(val) // 这是指针的反
    ele := v.Elem() // 拿到了指针指向的结构体
```

```
name := val.(Service).ServiceName()

cfg, ok := CfgMap[name]
if !ok {
    return []reflect.Value{reflect.ValueOf(out), reflect.ValueOf(errors.New("找不到服务配置: " + name))}
}

resp, err := client.Post(cfg.Endpoint, contentType: "application/json", bytes.NewReader(inData))
```

```
type Service interface {
    ServiceName() string
}
```

# RPC —— 设计一个配置模块

- CfgMap是一个写死的东西，我们期望能够从不同的地方读取配置

```go
type ConfigProvider interface {
    GetServiceConfig(serviceName string) (*Config, error)
}

type InMemoryConfigProvider struct {
    cfg map[string]*Config
}

var ErrServiceNotFound = errors.New( text: "service not found")

func (i *InMemoryConfigProvider) GetServiceConfig(serviceName string) (*Config, error) {
    cfg, ok := i.cfg[serviceName]
    if !ok {
        return nil, ErrServiceNotFound
    }
    return cfg, nil
}
```

# Golang 语法 —— sentinel error

- ErrServiceNotFound 是一个预定义的错误
- sentinel error 是 golang 里面常用的一种解决方案，用于向调用方指示特定错误
- 例如ErrNoRows, KeyNotFound, ...
- 只有在你觉得，调用方需要区分具体某些错误的时候，才需要使用

```go
type ConfigProvider interface {
    GetServiceConfig(serviceName string) (*Config, error)
}

type InMemoryConfigProvider struct {
    cfg map[string]*Config
}

var ErrServiceNotFound = errors.New( text: "service not found")

func (i *InMemoryConfigProvider) GetServiceConfig(serviceName string) (*Config, error) {
    cfg, ok := i.cfg[serviceName]
    if !ok {
        return nil, ErrServiceNotFound
    }
    return cfg, nil
}
```

# RPC —— 提供一个基于文件的实现

- 我需要读取文件，解析文件内容
- 所以我需要知道从哪里读取数据

# RPC —— 提供一个基于文件的实现

- 我需要读取文件，解析文件内容
- 所以我需要知道从哪里读取数据

- 我们需要将文件内容解析成为这个结构体

```go
type YamlConfigProvider struct {
    // 作为名字到配置的映射
    Services map[string]*Config `yaml:"services"`
}

func (y *YamlConfigProvider) GetServiceConfig(serviceName string) (*Config, error) {
    cfg, ok := y.Services[serviceName]
    if !ok {
        return nil, ErrServiceNotFound
    }
    return cfg, nil
}
```

# RPC —— 创建一个Provider

- 我需要读取文件，解析文件内容
- 所以我需要知道从哪里读取数据

- 我们需要将文件内容解析成为这个结构体

```go
func NewYamlConfigProvider(filepath string) (*YamlConfigProvider, error) {
    content, err := os.ReadFile(filepath)
    if err != nil {
        return nil, err
    }

    ycp := &YamlConfigProvider{}
    err = yaml.Unmarshal(content, ycp)
    return ycp, err
}
```

# golang 语法—— 文件操作

- 读文件：
  - ReadFile
  - Open
- 打开文件，用于后续读写：OpenFile
- 创建文件：Create

文件操作不用死记硬背，知道在 os 包就可以，有需要的时候，再去翻文档。

```go
func TestFile(t *testing.T) {
    path := `D:\workspace\go\src\geektime\sparrow\test\client.yaml`
    // 打开一个文件，只能读
    file, _ := os.Open(path)
    fmt.Printf( format: "%s\n", file.Name())

    // 创建了一个文件。如果文件已经存在，会被清空
    file, _ = os.Create( name: `D:\workspace\go\src\geektime\sparrow\test\test.txt`)

    // 写入数据，cnt 是写入数据字节数
    cnt, _ := file.WriteString( s: "你好")
    fmt.Printf( format: "写入字节数：%d", cnt)

    // 打开一个append only 模式的文件
    file, _ = os.OpenFile( name: `D:\workspace\go\src\geektime\sparrow\test\test.txt`, os.O_APPEND, fs.ModeAppend)
    _, _ = file.WriteString( s: "hello, world")
}
```

# golang —— 文件 flag 和 mode

- OpenFile 调用
- os.O_XXXX | os.O_YYY 用于设置 flag
- fs.ModeXXX | fs.ModeYYY 用于设置 Mode

同样不要死记硬背，记住在哪里，IDE 里面点过去看，自己写测试

```go
// OpenFile is the generalized open call; most users will use Open
// or Create instead. It opens the named file with specified flag
// (O_RDONLY etc.). If the file does not exist, and the O_CREATE flag
// is passed, it is created with mode perm (before umask). If successful,
// methods on the returned File can be used for I/O.
// If there is an error, it will be of type *PathError.
func OpenFile(name string, flag int, perm FileMode) (*File, error) {
    testlog.Open(name)
    f, err := openFileNolog(name, flag, perm)
    if err != nil { return nil, err }
    f.appendMode = flag&O_APPEND != 0

    return f, nil
}
```

```go
// Open opens the named file for reading. If successful, methods on
// the returned file can be used for reading; the associated file
// descriptor has mode O_RDONLY.
// If there is an error, it will be of type *PathError.
func Open(name string) (*File, error) {
    return OpenFile(name, O_RDONLY, perm: 0)
}

// Create creates or truncates the named file. If the file already exists,
// it is truncated. If the file does not exist, it is created with mode 0666
// (before umask). If successful, methods on the returned File can
// be used for I/O; the associated file descriptor has mode O_RDWR.
// If there is an error, it will be of type *PathError.
func Create(name string) (*File, error) {
    return OpenFile(name, O_RDWR|O_CREATE|O_TRUNC, perm: 0666)
}
```

# golang —— 文件操作最常见错误原因

- 文件不存在
- 文件存在但是没权限

找不到文件的最常见原因，路径写错或者使用了相对路径。相对路径定位是相对于当前工作目录，用 `os.GetWd()` 来查看

**将路径打印出来 DEBUG！**

**不要猜！不要相信自己推测！**

# RPC —— 集成 ConfigProvider

问题：我怎么知道使用哪种 Provider 的实现？我有in-memory（基于内存），我也有file（基于文件），将来用户可能扩展基于远程配置中心的……

答案：要在整个 RPC 启动的时候指定一个。为此我们需要引入一个 Application 的概念，表达当下整个 RPC 应用。

```go
type application struct {
    CfgProvider ConfigProvider
}

// App 必须要显示使用 InitApplication 来创建
var App *application
```

# RPC —— Application 概念

- App 是全局唯一
- 考虑到我们的扩展性，我们使用 golang 一种常见的设计模式来设计 application 的初始化过程

```go
type application struct {
    CfgProvider ConfigProvider
}

// App 必须要显示使用 InitApplication 来创建
var App *application
```

# golang 设计模式 —— Options

1. 定义结构体：这里是一个 application，里面含有各个字段
2. 定义 Option，一般是一个方法，其参数是结构体指针
3. 定义使用Option的方法，这里是初始化application。里面一般先是使用默认数据创建实例，而后是用 option 来改变默认值
4. 提供各种 Option 实现

```go
type application struct {
    CfgProvider ConfigProvider
}

// App 必须要显示使用 InitApplication 来创建
var App *application

type AppOption func(app *application) error

func InitApplication(opts ... AppOption) error {
    App = &application{
        // 默认实现
        CfgProvider: NewInMemoryConfigProvider(),
    }
    for _, opt := range opts {
        err := opt(App)
        if err != nil {
            return err
        }
    }
    return nil
}

func WithCfgProvider(cfp ConfigProvider) AppOption {
    return func(app *application) error {
        app.CfgProvider = cfp
        return nil
    }
}
```

# RPC —— 集成 ConfigProvider

用户在应用初始化的时候初始化 App

启动失败的时候直接panic，不必执行了

```go
func main() {
    fcg, err := NewYamlConfigProvider( filepath: "your path")
    if err != nil {
        panic( v: "初始化配置失败")
    }
    err = InitApplication(WithCfgProvider(fcg))
    if err != nil {
        panic( v: "初始化应用失败")
    }
}
```

```go
name := val.ServiceName()

cfg, err := App.CfgProvider.GetServiceConfig(name)

if err != nil {
    return []reflect.Value{reflect.ValueOf(out), reflect.ValueOf(err)}
}

resp, err := client.Post(cfg.Endpoint, contentType: "application/json", bytes.NewReader(inData))
```

# golang 语法 —— panic

简单理解，就是不可挽回的错误。

panic 和 error 的区别：

1. panic 更加严重，强调的是不可挽回；
2. error 代表的是错误，属于可以修复的类型；error 更加接近 java 里面的 exception

```go
func main() {
    fcg, err := NewYamlConfigProvider( filepath: "your path")
    if err != nil {
        panic( v: "初始化配置失败")
    }
    err = InitApplication(WithCfgProvider(fcg))
    if err != nil {
        panic( v: "初始化应用失败")
    }
}
```

# golang 语法 —— panic 和 error 我选哪个？

## 遇事不决选 error

当你怀疑可以用 error 的时候，就说明你不需要 panic

一般情况下，只有快速失败的过程，才会考虑 panic

```go
func main() {
    fcg, err := NewYamlConfigProvider( filepath: "your path")
    if err != nil {
        panic( v: "初始化配置失败")
    }
    err = InitApplication(WithCfgProvider(fcg))
    if err != nil {
        panic( v: "初始化应用失败")
    }
}
```

# golang 语法 —— 我就要从 panic 中恢复过来

某些时候，你可能需要从 panic 中恢复过来：

比如某个库，发生 panic 的场景是你不希望发生的场景。

这时候，你需要我们的 recover

```
// The recover built-in function allows a program to manage behavior of a
// panicking goroutine. Executing a call to recover inside a deferred
// function (but not any function called by it) stops the panicking sequence
// by restoring normal execution and retrieves the error value passed to the
// call of panic. If recover is called outside the deferred function it will
// not stop a panicking sequence. In this case, or when the goroutine is not
// panicking, or if the argument supplied to panic was nil, recover returns
// nil. Thus the return value from recover reports whether the goroutine is
// panicking.
func recover() interface{}
```

# golang 语法 —— recover 惯常使用

recover 往往和 defer 一起使用。

记住右边的这个例子

```go
defer func() {
    if data := recover(); data != nil {
        fmt.Printf( format: "hello, panic: %v \n", data)
    }

    fmt.Print( a...: "恢复是在这里继续执行")
}()

panic( v: "boom !")

fmt.Print( a...: "即便恢复，也不会继续执行")
```

```
API server listening at: [::]:58341
=== RUN   TestInitApplication
hello, panic: boom !
恢复是在这里继续执行--- PASS: TestInitApplication (0.00s)
PASS

Debugger finished with the exit code 0
```

# Golang 语法 —— defer

- 用于在方法返回之前执行某些动作
- 像栈一样，先进后出

defer 语义接近 java 的 finally 块
所以我们经常使用 defer 来释放资源，例如释放锁

```
func Defer() {
    defer func() {
        println( args...: "A")
    }()

    defer func() {
        println( args...: "B")
    }()

    defer func() {
        println( args...: "C")
    }()

    defer func() {
        println( args...: "D")
    }()
}
```

# Server 走起

终于来了

```
10
11  func handler(w http.ResponseWriter, r *http.Request) {
12      data, _ := ioutil.ReadAll(r.Body)
13      fmt.Printf("input: #{string(data)} \n")
14      input := &Input{}
15      _ = json.Unmarshal(data, input)
16      output, _ := json.Marshal(&Output{
17          Msg: "Hello, " + input.Name,
18      })
19      fmt.Fprintf(w, "#{string(output)}")
20  }
```

```go
func handler(w http.ResponseWriter, r *http.Request) {
    data, _ := ioutil.ReadAll(r.Body)
    fmt.Printf("input: #{string(data)} \n")
    input := &Input{}
    _ = json.Unmarshal(data, input)
    output, _ := json.Marshal(&Output{
        Msg: "Hello, " + input.Name,
    })
    fmt.Fprintf(w, "#{string(output)}")
}

func main() {
    http.HandleFunc( pattern: "/", handler)
    log.Fatal(http.ListenAndServe( addr: ":8080",  handler: nil))
}
```

```go
type Service interface {
    ServiceName() string
}
```

```go
type HelloService interface {
    Service
    Hello(input *Input) (*Output, error)
}

type UserService interface {
    Service
    GetUser(req *GetUserReq) (*GetUserResp, error)
}
```

server 怎么知道，调用的是 HelloService 还是 UserService

# Golang 语法 —— 组合

```
type Service interface {
    ServiceName() string
}

type HelloService interface {
    Service
    Hello(input *Input) (*Output, error)
}
```

```
type Base struct {
}

type Concrete struct {
    Base
}

type Concrete1 struct {
    *Base
}
```

非常非常类似于继承，可以访问字段，也可以访问方法

# Golang 语法 —— 组合

```go
type Base struct {
}

type Concrete struct {
    Base
}

type Concrete1 struct {
    *Base
}
```

```go
func (b Base) SayHello() {
    fmt.Printf("Base say hello, " + b.Name())
}

func (b Base) Name() string {
    return "Base"
}

func (c Concrete) Name() string {
    return "Concrete"
}
```

```go
base := Base{}
c := Concrete{
    Base: base,
}
c.SayHello()
```

输出什么？

# Golang 语法 —— 组合

```go
func (b Base) SayHello() {
    fmt.Printf("Base say hello, " + b.Name())
}

func (b Base) Name() string {
    return "Base"
}

func (c Concrete) Name() string {
    return "Concrete"
}
```

```go
base := Base{}
c := Concrete{
    Base: base,
}
c.SayHello()
```

组合不是继承！组合不是继承！组合不是继承！

Golang 没有重写！ Golang 也没有多态！

只要方法跑过去 Base 里面，后续所有的调用，执行的都是 Base 的方法

# 组合不是继承

没有重写没有多态

# RPC —— server 维护所有 services

采用一个线程安全的 map 来维护

```go
type Service interface {
    ServiceName() string
}

var services sync.Map

func AddService(service Service) {
    services.Store(service.ServiceName(), service)
}

var ErrServiceNotFound = errors.New( text: "service not found")

func GetService(name string) (Service, error) {
    if service, ok := services.Load(name); ok {
        return service.(Service), nil
    }
    return nil, ErrServiceNotFound
}
```

# golang 包 —— sync

sync 包提供了基本的并发工具

- sync.Map：并发安全 map
- sync.Mutex：锁
- sync.RWMutex：读写锁
- sync.Once：只执行一次

```go
type Service interface {
    ServiceName() string
}

var services sync.Map

func AddService(service Service) {
    services.Store(service.ServiceName(), service)
}

var ErrServiceNotFound = errors.New( text: "service not found")

func GetService(name string) (Service, error) {
    if service, ok := services.Load(name); ok {
        return service.(Service), nil
    }
    return nil, ErrServiceNotFound
}
```

# golang 包 —— sync.Mutex 和 sync.RWMutex

sync 包提供了基本的并发工具

- sync.Map：并发安全 map
- sync.Mutex：锁
- sync.RWMutex：读写锁
- sync.Once：只执行一次

```go
func Mutex() {
    var lock sync.Mutex
    lock.Lock()
    defer lock.Unlock()
}
```

```go
func RwMutex() {
    var lock sync.RWMutex
    // 加读锁
    lock.RLock()
    defer lock.RUnlock()

    // 加写锁
    lock.Lock()
    defer lock.Unlock()
}
```

# golang 包 —— mutex家族注意事项

- 尽量用 RWMutext
- 尽量用 defer 来释放锁，防止panic没有释放锁
- 不可重入：lock 之后，即便是同一个线程(goroutine)，也无法再次加锁（写递归函数要小心）
- 不可升级：加了读锁之后，如果试图加写锁，锁不升级

不可重入和不可升级，和很多语言的实现都是不同的，因此要小心使用

```
var lock sync.Mutex
lock.Lock()
lock.Lock() // 死锁
fmt.Print( a...: "hello")
lock.Unlock()
lock.Unlock()
```

```
var lock sync.RWMutex
lock.RLock()
lock.Lock() // 死锁，卡主
fmt.Print( a...: "aa")
lock.Unlock()
lock.RUnlock()
```

# golang 包 —— sync.Once

sync 包提供了基本的并发工具

- sync.Map：并发安全 map
- sync.Mutex：锁
- sync.RWMutex：读写锁
- sync.Once：只执行一次

```go
// App 必须要显示使用 InitApplication 来创建
var App *application
var appOnce sync.Once

type AppOption func(app *application) error

func InitApplication(opts ... AppOption) error {
    var err error
    appOnce.Do(func() {
        App = &application{
            // 默认实现
            CfgProvider: NewInMemoryConfigProvider(),
        }
        for _, opt := range opts {
            err = opt(App)
            if err != nil {
                return
            }
        }
    })
    return err
}
```

确保只初始化一次

# golang 包 —— sync

一点都不花里胡哨

比 java 并发框架简单多了

三分钟就学会

# RPC —— 服务分发

```go
func handler(w http.ResponseWriter, r *http.Request) {
    data, _ := ioutil.ReadAll(r.Body)
    fmt.Printf( format: "input: %s \n", string(data))
    input := &Input{}
    _ = json.Unmarshal(data, input)
    output, _ := json.Marshal(&Output{
        Msg: "Hello, " + input.Name,
    })
    fmt.Fprintf(w, format: "%s", string(output))
}

func main() {
    AddService(&userService{})
    AddService(&helloService{})
    http.HandleFunc( pattern: "/", handler)
    log.Fatal(http.ListenAndServe( addr: ":8080", handler: nil))
}
```

问题来了，我怎么知道client是调用helloService还是userService，调用的是哪个方法？

# RPC —— 服务分发

只能客户端提供这种信息，我们要考虑让客户端带上这些数据

例如，当用户调用 `HelloService` 的 `SayHello` 的时候，我们把服务名和方法名都传过来

```go
func handler(w http.ResponseWriter, r *http.Request) {
    data, _ := ioutil.ReadAll(r.Body)
    fmt.Printf( format: "input: %s \n", string(data))
    input := &Input{}
    _ = json.Unmarshal(data, input)
    output, _ := json.Marshal(&Output{
        Msg: "Hello, " + input.Name,
    })
    fmt.Fprintf(w, format: "%s", string(output))
}

func main() {
    AddService(&userService{})
    AddService(&helloService{})
    http.HandleFunc( pattern: "/", handler)
    log.Fatal(http.ListenAndServe( addr: ":8080", handler: nil))
}
```

问题来了，我怎么知道client是调用helloService还是userService，调用的是哪个方法？

# RPC —— 元数据传递

我们把服务名和方法名这种东西称为元数据，
也可以称为调用上下文

那么 HTTP 协议 body 被我们用了，还有
啥可以用的？

- header
- path

# RPC —— header 传递元数据

```
    }
    req, err := http.NewRequest( method: "POST", cfg.Endpoint, bytes.NewReader(inData))

    if err != nil {
        return []reflect.Value{reflect.ValueOf(out), reflect.ValueOf(err)}
    }

    req.Header.Set( key: "Content-Type", value: "application/json")
    req.Header.Set( key: "sparrow-service", name)
    req.Header.Set( key: "sparrow-service-method", field.Name)

    resp, err := client.Do(req)
```

```
func handler(w http.ResponseWriter, r *http.Request) {
    data, _ := ioutil.ReadAll(r.Body)
    serviceName := r.Header.Get( key: "sparrow-service")
    methodName := r.Header.Get( key: "sparrow-service-method")

    service, _ := GetService(serviceName)

    val := reflect.ValueOf(service)

    method := val.MethodByName(methodName)
    inType := method.Type().In( i: 0)
    in := reflect.New(inType.Elem())
    _ = json.Unmarshal(data, in.Interface())
    res := method.Call([]reflect.Value{in})

    output, _ := json.Marshal(res[0].Interface())
    fmt.Fprintf(w, format: "%s", string(output))
}
```

# RPC —— 返回error

你这咋没处理 error 呀

这是作思考题~~

即，如何返回错误，让 client 知道 server 发生了错误，调用方能拿到准确的 error 实例

```go
func handler(w http.ResponseWriter, r *http.Request) {
    data, _ := ioutil.ReadAll(r.Body)
    serviceName := r.Header.Get( key: "sparrow-service")
    methodName := r.Header.Get( key: "sparrow-service-method")

    service, _ := GetService(serviceName)

    val := reflect.ValueOf(service)

    method := val.MethodByName(methodName)
    inType := method.Type().In( i: 0)
    in := reflect.New(inType.Elem())
    _ = json.Unmarshal(data, in.Interface())
    res := method.Call([]reflect.Value{in})

    output, _ := json.Marshal(res[0].Interface())
    fmt.Fprintf(w, format: "%s", string(output))
}
```

# 一个大概能用的 server也有了

它虽然丑，但是能用呀

# 改造我们的RPC吧

一直在重构

# RPC —— 优雅退出，一个 channel 的场景

为啥要优雅退出：退出前清理一些资源。

- 下线节点
- 不再接收新请求
- 执行完当前请求
- 释放数据库连接
- 执行用户自定义动作
- 彻底关闭

监听 OS 信号量

# RPC —— 优雅退出，一个 channel 的场景

```go
// 启动服务器
func main() {
    AddService(&userService{})
    AddService(&helloService{})

    go func() {
        listenSignal()
    }()

    // go listenSignal()

    http.HandleFunc(pattern: "/", handler)
    log.Fatal(http.ListenAndServe(addr: ":8080", handler: nil))
}
```

```go
func listenSignal() {
    signals := make(chan os.Signal, 1)

    signal.Notify(signals, sysSignals...)

    select {
    case <- signals:
        // 设置超时，强制关闭
        forceShutdownIfNeed()
        shutdown()
        os.Exit(code: 0)
    }
}
```

# golang 语法 —— goroutine

- 一段异步执行的代码
- 用关键字 go 启动

```go
// 启动服务器
func main() {
    AddService(&userService{})
    AddService(&helloService{})

    go func() {
        listenSignal()
    }()

    // go listenSignal()

    http.HandleFunc( pattern: "/", handler)
    log.Fatal(http.ListenAndServe( addr: ":8080",  handler: nil))
}
```

```go
func Goroutine() {
    go func() {
        time.Sleep(10 * time.Second)
    }()
    fmt.Print( a...: "hello, world")
}
```

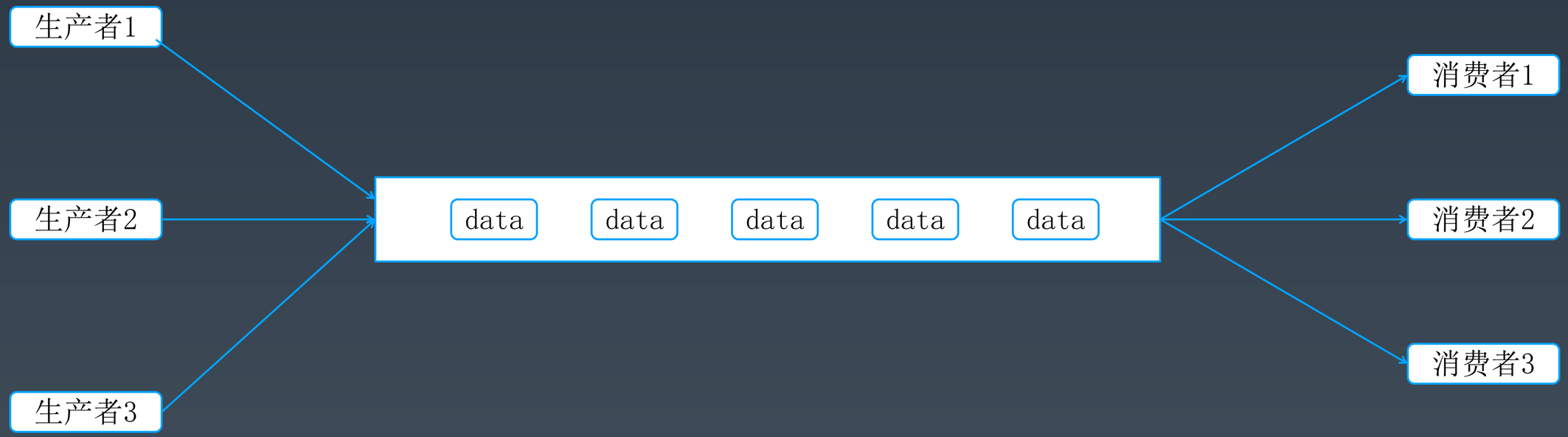
直接输出，而不会等待十秒

# golang 语法—— channel

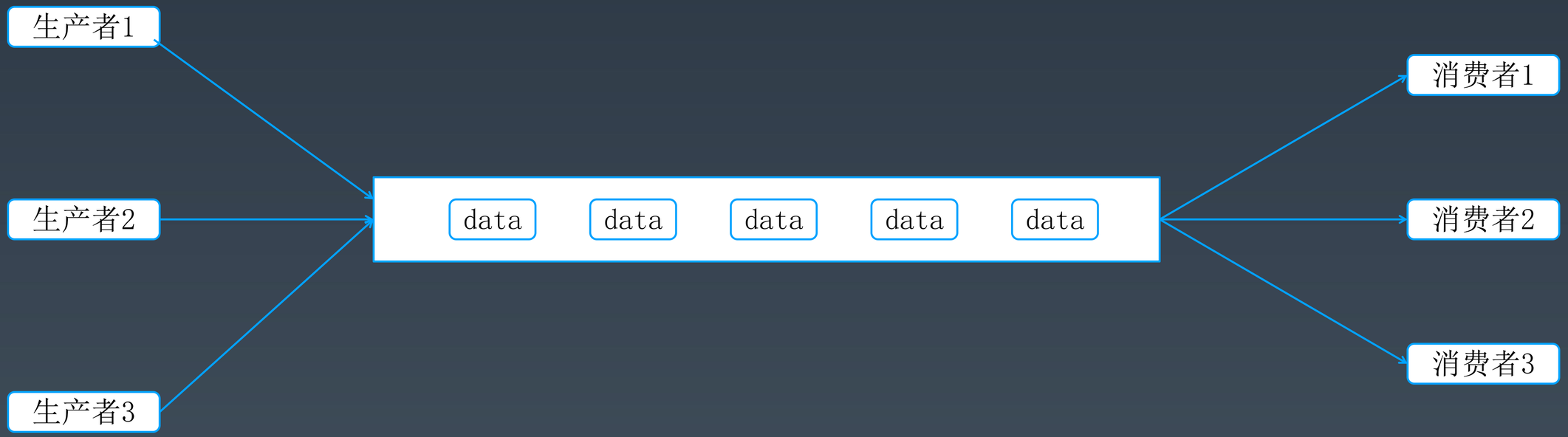
1. 使用 make 创建 channel
2. 带缓冲和不带缓冲的channel

```go
func listenSignal() {
    signals := make(chan os.Signal, 1) // 1 表示缓冲大小为1

    // signals := make(chan os.Signal) // 不带缓冲

    signal.Notify(signals, sysSignals...)

    select {
    case <- signals:
        // 设置超时，强制关闭
        forceShutdownIfNeed()
        shutdown()
        os.Exit( code: 0)
    }
}
```

# golang 语法 —— channel 带缓冲



# golang 语法—— channel 带缓冲




放满阻塞生产者
空了阻塞消费者

# golang 语法—— channel 不带缓冲


缺了任何一边都要阻塞另外一边

# golang 语法—— channel

用 <- 符号来表达收发

```go
func channel()  {
    ch := make(chan string)
    go func() {
        time.Sleep(time.Second)
        ch <- "Hello, msg from channel"
    }()

    msg := <- ch

    fmt.Println(msg)
}
```

# golang 语法 —— select

等待多个 channel

想象一下，自己对接了好多个产品。然后哪个产品来了需求就做哪个，做完就做下一个。那么问题来了，如果同时来了，怎么办？—— 当然是随便挑一个

select 比较常见和 for 循环一起使用

```go
func Select() {
    ch1 := make(chan string)
    ch2 := make(chan string)

    go func() {
        time.Sleep(time.Second)
        ch1 <- "msg from ch1"
    }()

    go func() {
        time.Sleep(time.Second)
        ch2 <- "msg from ch2"
    }()

    for i := 0; i < 2; i++ {
        select {
        case msg := <- ch1:
            fmt.Println(msg)
        case msg := <- ch2:
            fmt.Println(msg)
        }
    }
```

# RPC —— 优雅退出，一个 channel 的场景

```go
func forceShutdownIfNeed() {
    time.AfterFunc(time.Minute, func() {
        os.Exit( code: 1)
    })
}

func shutdown() {
    // 执行各种动作
    services.Range(func(key, value interface{}) bool {
        service := value.(Service)
        go func() {
            service.ShutDown()
        }()
        return true
    })
}
```

超时强制关闭

多个goroutine一起关

# RPC - handler 分析

整个 handler 的过程，分成三类：

1. HTTP 协议相关 （“传输”协议）
2. 反射相关 （内部实现）
3. json 相关 （序列化协议）

```go
func handler(w http.ResponseWriter, r *http.Request) {
    data, _ := ioutil.ReadAll(r.Body)
    serviceName := r.Header.Get( key: "sparrow-service")
    methodName := r.Header.Get( key: "sparrow-service-method")

    service, _ := GetService(serviceName)

    val := reflect.ValueOf(service)

    method := val.MethodByName(methodName)
    inType := method.Type().In( i: 0)
    in := reflect.New(inType.Elem())
    _ = json.Unmarshal(data, in.Interface())
    res := method.Call([]reflect.Value{in})

    output, _ := json.Marshal(res[0].Interface())
    fmt.Fprintf(w, "#{string(output)}")
}
```

# RPC - 所以为何 RPC 框架那么复杂

- 直接基于 TCP 的
- 基于 HTTP 的
- 基于 gRPC 的
- ...

- json
- xml
- protobuf
- thrift
- ...

笛卡尔积一下，在加上服务治理，就是我们日常了解的五花八门的RPC了

# RPC - invoker 抽象

将和HTTP无关的部分抽取出来作为一个 Invoker 的抽象：

1. 考虑将来扩展非 HTTP 协议，例如直接使用TCP 协议传输；
2. 引入 Filter 等机制

```
func handler(w http.ResponseWriter, r *http.Request) {
    data, _ := ioutil.ReadAll(r.Body)
    serviceName := r.Header.Get( key: "sparrow-service")
    methodName := r.Header.Get( key: "sparrow-service-method")

    service, _ := GetService(serviceName)

    val := reflect.ValueOf(service)

    method := val.MethodByName(methodName)
    inType := method.Type().In( i: 0)
    in := reflect.New(inType.Elem())
    _ = json.Unmarshal(data, in.Interface())
    res := method.Call([]reflect.Value{in})

    output, _ := json.Marshal(res[0].Interface())
    fmt.Fprintf(w, "#{string(output)}")
}
```

# RPC - invoker 抽象

将和HTTP无关的部分抽取出来作为一个 Invoker 的抽象：

1. 考虑将来扩展非 HTTP 协议，例如直接使用TCP 协议传输；
2. 引入 Filter 等机制

```go
func handler(w http.ResponseWriter, r *http.Request) {
    data, _ := ioutil.ReadAll(r.Body)
    serviceName := r.Header.Get( key: "sparrow-service")
    methodName := r.Header.Get( key: "sparrow-service-method")

    service, _ := GetService(serviceName)

    val := reflect.ValueOf(service)

    method := val.MethodByName(methodName)
    inType := method.Type().In( i: 0)
    in := reflect.New(inType.Elem())
    _ = json.Unmarshal(data, in.Interface())
    res := method.Call([]reflect.Value{in})

    output, _ := json.Marshal(res[0].Interface())
    fmt.Fprintf(w, "#{string(output)}")
}
```

```go
type Invoker interface {
    Invoke(inv *Invocation) ([]byte, error)
}

type Invocation struct {
    MethodName string
    ServiceName string
    Input []byte
}
```

# RPC - invoker 抽象

将和HTTP无关的部分抽取出来作为一个 Invoker 的抽象：

1. 考虑将来扩展非 HTTP 协议，例如直接使用TCP 协议传输；
2. 将 Invoker 的实现像洋葱一样层层包裹，扩展实现各种功能，如 filter 支持，cluster 支持

```go
func handler(w http.ResponseWriter, r *http.Request) {
    data, _ := ioutil.ReadAll(r.Body)
    serviceName := r.Header.Get( key: "sparrow-service")
    methodName := r.Header.Get( key: "sparrow-service-method")

    service, _ := GetService(serviceName)

    val := reflect.ValueOf(service)

    method := val.MethodByName(methodName)
    inType := method.Type().In( i: 0)
    in := reflect.New(inType.Elem())
    _ = json.Unmarshal(data, in.Interface())
    res := method.Call([]reflect.Value{in})

    output, _ := json.Marshal(res[0].Interface())
    fmt.Fprintf(w, "#{string(output)}")
}
```

```go
type Invoker interface {
    Invoke(inv *Invocation) ([]byte, error)
}

type Invocation struct {
    MethodName string
    ServiceName string
    Input []byte
}
```

# RPC - invoker 抽象

```go
func (h *httpInvoker) Invoke(inv *Invocation) ([]byte, error) {
    serviceName := inv.ServiceName
    methodName := inv.MethodName
    data := inv.Input
    service, err := GetService(serviceName)
    if err != nil {
        return nil, err
    }
    val := reflect.ValueOf(service)
    method := val.MethodByName(methodName)
    inType := method.Type().In( i: 0)
    in := reflect.New(inType.Elem())
    err = json.Unmarshal(data, in.Interface())
    if err != nil {
        return nil, err
    }
    res := method.Call([]reflect.Value{in})
    output, err := json.Marshal(res[0].Interface())
    if err != nil {
        return nil, err
    }
    return output, nil
}
```

# RPC - Invoker装饰器模式解决 filter 设计

```go
// 也是一个装饰器模式，我们会在这里组织filter
type filterInvoker struct {
    Invoker
    filters []Filter
}

func (f *filterInvoker)Invoke(inv *Invocation) ([]byte, error) {
    for _, flt := range f.filters {
        flt(inv)
    }
    return f.Invoker.Invoke(inv)
}

// 通过扩展 filter 来完成别的工作，例如接入 metrics
type Filter func(inv *Invocation)

func logFilter(inv *Invocation) {
    fmt.Printf( format: "log filter ===== service name: %s, method name: %s \n",
        inv.ServiceName, inv.MethodName)
}
```

# RPC - 装饰器模式解决filter 设计

```go
func handler(w http.ResponseWriter, r *http.Request) {
    data, _ := ioutil.ReadAll(r.Body)
    serviceName := r.Header.Get(key:"sparrow-service")
    methodName := r.Header.Get(key:"sparrow-service-method")

    filterIvk := &filterInvoker{
        Invoker:  &httpInvoker{},
        filters: []Filter{logFilter},
    }
    output, _ := filterIvk.Invoke(&Invocation{
        MethodName: methodName,
        ServiceName: serviceName,
        Input: data,
    })
    fmt.Fprintf(w, format:"%s", string(output))
}
```

# Client 也可以进行类似改造

照猫画虎，依葫芦画瓢

# 作业

1. 开启三个 goroutine 分别输出 A, B, C，要求顺序必然是第一个goroutine 输出A之后第二个goroutine 输出B，然后第三个 goroutine 输出C，如此循环一百次
2. 尝试写一个 Filter，能够将请求参数输出到一个日志文件
3. 尝试改造一下 Filter 和 FilterInvoker 的设计，使得它能够输出请求的执行时间。注意，此时我们的filter是只在业务代码执行之前执行，也就是我们能够获得请求的开始时间，但是我们的filter目前不支持在业务代码执行之后执行，因此我们不能获得请求的结束时间，因而无法获得

# 作业

3. 尝试改造一下 Filter 和 FilterInvoker 的设计，使得它能够输出请求的执行时间。注意，此时我们的filter是只在业务代码执行之前执行，也就是我们能够获得请求的开始时间，但是我们的filter目前不支持在业务代码执行之后执行，因此我们不能获得请求的结束时间，因而无法获得执行时间（结束时间-执行时间）

# THANKS