# Go 进阶训练营
# Kratos 框架设计的思考与实现

毛剑

# 目录

- 面向包的设计理念
- 配置规范的思考
- 业务错误处理
- 日志接口设计
- **Metadata** 传递和使用
- **Transport HTTP/gRPC**
- **Middleware** 插件化的使用
- **Kratos** 启动管理器

# What is Go ?

Go is:

- *open source*
- *concurrent*
- *garbage-collected*
- *efficient*
- *scalable*
- *simple*
- *fun*
- *...*

http://golang.org

```
for {
    // do somthing
}
for n < 5 {
    // do somthing
}
for i := 0; i < 5; i++ {
    // do somthing
}
for i, s := range strings {
    // do somthing
}
```

**loop、while、do..while、for、range**

# 面向包的设计理念

**Go** 是一个面向包名设计的语言。

**Package** 在 **Go** 程序中主要起到功能隔离的作用：

- *程序的各个部分可以分开*
- *在大型团队中处理复杂的项目*
- *包名可改善团队成员之间的沟通*

**Go** 对于包支持其实很棒的，标准库就是很好的设计。

但是，我们在开发过程中，如果没有采用很好的代码组织方式，可能会让项目非常地难以理解。

# 面向包的设计理念

在 **Go** 标准库，也提供很多常用的 **Packages**：

- *fmt*
- *strings*
- *bytes*
- *io*
- *errors*
- *encoding*
- *sync*
- *time*
- *net/http*
- *net/rpc*
- *database/sql*

# 面向包的设计理念

为了有目的，包必须提供，而不是包含。

包的命名必须旨在描述它提供的内容：

- *包的目的是为特定问题域而提供的。*
- *每个包的目的越集中，就越清楚知道包提供了什么。*
- *包的名称必须描述它提供的内容，如果包的名称不能立即暗示这一点，则它可能包含一组零散的功能。*
- *这些零散的功能，可能包含一些 util、common、helpers 相关。*

包不能成为不同问题域的聚合地，随着时间的推移，它将影响项目的简洁和重构、适应、扩展和分离的能力。

# 面向包的设计理念

为了可用，包的设计必须以用户为中心。

包的存在是为了解决开发人员特定问题而提供的支持。

包应该有以下几个特点：

- *必须直观且易于使用；*
- *必须包对资源和性能的影响；*
- *必须防止需要对具体的类型断言；*
- *必须减少、最小化和简化引入其代码库；*

# 面向包的设计理念

为了便于携带，包的设计必须考虑到可重用性。

- *包装必须追求极高的便携性*
  - *一个包与其它包依赖越少，一个包的可重用性就越高。*
- *包必须减少对其它包设置策略*
  - *当包具有依赖性时，通常接受来自其它包导入的所有策略。*
- *包不能成为单点依赖*
  - *当包被单一的依赖点时，就像一个公共包（common），会给项目带来很高的耦合性。*

# 面向包的设计理念

从包的设计理念中，我们了解到标准库其实就是比较好的例子。

其实，我们 **Kit** 和 **Application** 项目，也是可以按包进行组织代码结构。

让大家遵循良好的包规范设计准则，有助于项目架构的可持续迭代和维护。

而 **Kit** 项目视为公司的标准库，应该具备：

- *统一*
- *高度抽象*
- *支持插件*
- *尽量减少依赖*
- *可持续维护*

# 面向包的设计理念

在 **Kratos** 框架中，我们主要参考了基础库的包设计理念。

包名按功能进行划分，每个包具有唯一的职责。

当 用户不可见 或者 不稳定 的接口需要放到 /internal 目录中。

然而，我们也主要分为这几层的设计思想：

- ***API Layer***
- ***Transport Layer***
- ***Middleware Layer***
- ***Service Layer***
- ***Data Layer***

# 面向包的设计理念

Kratos 每个包功能特性：

- */cmd，可以通过 **go install** 一键安装生成工具，使用户更加方便地使用框架。*
- */errors，统一的业务错误封装，方便返回错误码和业务原因。*
- */config，支持多数据源方式，进行配置合并铺平，通过 **Atomic** 方式支热更配置。*
- */transport，传输层（HTTP/gRPC）的抽象封装，可以方便获取对应的接口信息。*
- */middleware，中间件抽象接口，主要跟 **transport** 和 **service** 之间的桥梁适配器。*
- */metadata，跨服务间的元信息传递和使用。*
- */registry，注册中心适配接口，可以实现各种服务发现，例如：etcd/consul/zookeeper。*

```
github.com/go-kratos/kratos
├── cmd
├── docs
├── internal
├── examples
├── api
├── errors
├── config
├── encoding
├── log
├── metrics
├── metadata
├── middleware
├── transport
├── registry
├── third_party
├── app.go
├── options.go
├── go.mod
└── go.sum
```

# 面向包的设计理念

Kratos 主要的框架工具：

- *cmd/kratos*
  - *kratos new project_name，创建项目模板，默认通过 **kratos-layout** 仓库下载，也可以通过 -r 或者 环境变量 **KRATOS_LAYOUT_REPO** 指定自定义模板仓库。*
  - *kratos proto add，添加一个 **proto** 模板文件（CURD）。*
  - *kratos proto client，生成 **proto** 源码文件。*
  - *kratos proto server，生成 **proto service** 实现代码文件。*
  - *kratos upgrade，更新 **kratos** 工具到最新版本。*
- *cmd/protoc-gen-go-errors，为 **errors** 生成对应 **IsXxx**、**ErrorXxx** 辅助代码。*
- *cmd/protoc-gen-go-http，为 **HTTP** 生成对应的接口定义，根据 google.api.http 规范实现。*

*go install github.com/go-kratos/kratos/cmd/kratos/v2@latest*

*kratos new helloworld*

# 面向包的设计理念

在 **Application** 项目结构中，其实是包含一起部署的程序集。

程序集可以包括 **server**、**cli**工具 和 **task** 等应用，通常会放到 /cmd 目录中。

如果在一个单体大型项目中，可以按包名类似划分为：

- */api*
- */cmd*
  - *xxxcli*
  - *xxxd*
- */internal*
  - *order*
  - *payment*
  - *platform*
    - *mysql*
    - *redis*

# 面向包的设计理念

如果在微服务中，通常我们也可以通过统一仓库（**mono-repo**）进行管理项目。

把不同的业务域划分出对应的微服务，再通过 **HTTP/gRPC** 进行进程之间的通信。

微服务结构划分，跟单体项目有所不同，可以把一个个项目放到 **app** 中：

- *api*
- *app*
  - *user*
  - *order*
  - *payment*
- *pkg*
  - *backoff*
  - *pagination*

# 面向包的设计理念

在 **Kratos** 项目中，我们主要是以微服务类型为标准的项目布局。

通过 **Kratos** 工具生成的项目结构为：

- *api*
- *cmd*
  - *myapp*
- *configs*
  - *application.yaml*
- *internal*
  - *service*
  - *biz*
  - *data*
- *go.mod*
- *go.sum*

# 配置规范的思考

在 **Kratos** 项目中，配置源可以指定多个，并且 **config** 会进行合并成 **key/value** 。

然后用户通过 **Scan** 或者 **Value** 获取对应键值内容；

主要功能特性：

- *默认实现了本地文件数据源。*
- *用户可以自定义数据源实现。*
- *支持配置热更新，通过 **Atomic** 方式变更已有 **Value**。*
- *支持自定义数据源解码实现。*

后续计划：

- *增加对 **flags**、环境变量 占位符替换。*
- *通过铺平的 **key/value**，进行二次赋值替换。*
- *例如，a.b.c = {{ xx.xxx }}*

```go
// Observer is config observer.
type Observer func(string, Value)

// Config is a config interface.
type Config interface {
        Load() error
        Scan(v interface{}) error
        Value(key string) Value
        Watch(key string, o Observer) error
        Close() error
}

// KeyValue is config key value.
type KeyValue struct {
        Key    string
        Value  []byte
        Format string
}

// Source is config source.
type Source interface {
        Load() ([]*KeyValue, error)
        Watch() (Watcher, error)
}

// Watcher watches a source for changes.
type Watcher interface {
        Next() ([]*KeyValue, error)
        Stop() error
}
```

# 配置规范的思考

在 **Kratos** 项目中，我们默认通过 **proto** 进行定义配置文件。

主要的以下几点好处：

- *可以定义统一的模板配置*
- *添加对应的配置校验*
- *更好地管理配置*
- *跨语言支持*

```yaml
server:
  http:
    addr: 0.0.0.0:8000
    timeout: 1s
  grpc:
    addr: 0.0.0.0:9000
    timeout: 1s
data:
  database:
    driver: mysql
    source: root:root@tcp(127.0.0.1:3306)/test
  redis:
    addr: 127.0.0.1:6379
    read_timeout: 0.2s
    write_timeout: 0.2s
```

```protobuf
message Bootstrap {
  Server server = 1;
  Data data = 2;
}

message Server {
  message HTTP {
    string network = 1;
    string addr = 2;
    google.protobuf.Duration timeout = 3;
  }
  message GRPC {
    string network = 1;
    string addr = 2;
    google.protobuf.Duration timeout = 3;
  }
  HTTP http = 1;
  GRPC grpc = 2;
}

message Data {
  message Database {
    string driver = 1;
    string source = 2;
  }
  message Redis {
    string network = 1;
    string addr = 2;
    google.protobuf.Duration read_timeout = 3;
    google.protobuf.Duration write_timeout = 4;
  }
  Database database = 1;
  Redis redis = 2;
}
```

# 业务错误处理

在 API 中，业务错误主要通过 proto 进行定义，并且通过工具自动生成辅助代码。

在 errors.Error 中，主要实现了 HTTP 和 gRPC 的接口：

- *StatusCode() int*
- *GRPCStatus() *grpc.Status*

业务错误，主要参考了 gRPC  errdetails.ErrorInfo 的实现：

- *code：错误码，跟 http-status 一致，并且在 grpc 中可以转换成 grpc-status。*
- *message：错误信息，为用户可读的信息，可作为用户提示内容。*
- *reason：错误原因，定义为业务判定错误码。*
- *metadata：错误元信息，为错误添加附加可扩展信息。*

```
{
    code: 400,
    reason: "custom_error",
    message: "invalid argument error",
    metadata: { }
}
```

# 业务错误处理

安装 errors 辅助代码生成工具：

- *go install github.com/go-kratos/kratos/cmd/protoc-gen-go-errors/v2*

通过 proto 生成对应的代码：

```
protoc --proto_path=. \
       --proto_path=./third_party \
       --go_out=paths=source_relative:. \
       --go-errors_out=paths=source_relative:. \
       error_reason.proto
```

使用生成的 errors 辅助代码：

```
v1.ErrorUserNotFound("user %s not found", "kratos")
v1.IsUserNotFound(err)
```

```proto
syntax = "proto3";

package api.blog.v1;
import "errors/errors.proto";

// 多语言特定包名，用于源代码引用
option go_package = "github.com/go-kratos/kratos/examples/blog/api/v1;v1";
option java_multiple_files = true;
option java_package = "blog.v1.errors";
option objc_class_prefix = "APIBlogErrors";

enum ErrorReason {
  // 设置缺省错误码
  option (errors.default_code) = 500;

  // 为某个枚举单独设置错误码
  USER_NOT_FOUND = 0 [(errors.code) = 404];

  CONTENT_MISSING = 1 [(errors.code) = 400];;
}
```

```go
func IsUserNotFound(err error) bool {
	e := errors.FromError(err)
	return e.Reason == ErrorReason_USER_NOT_FOUND.String() && e.Code == 404
}

func ErrorUserNotFound(format string, args ...interface{}) *errors.Error {
	return errors.New(404, ErrorReason_USER_NOT_FOUND.String(), fmt.Sprintf(format, args...))
}
```

# 日志接口设计

在 Kratos 日志中， 主要分为 Logger、Valuer、Filter、Helper 的实现。

为了方便扩展，Logger 接口定义非常简单：

```
type Logger interface {
    Log(level Level, keyvals ...interface{}) error
}
```

这个 Logger 接口，非常容易组合和扩展：

```
// 也可以定义多种日志输出 log.MultiLogger(out, err)，例如：info/warn/error，file/agent
logger := log.NewStdLogger(os.Stdout)
// 根据日志级别进行过虑日志，或者 Key/Value/FilterFunc
logger := log.NewFilter(logger, log.FilterLevel(log.LevelInfo))
// 输出结构化日志
logger.Log(log.LevelInfo, "msg", "log info")
```

# 日志接口设计

如果需要过滤日志中某些不应该被打印明文的字段，例如 password 等信息，可以通过 log.NewFilter() 来实现过滤功能。

```
logger := log.NewFilter(
    log.DefaultLogger,
    log.FilterLevel(log.LevelInfo), // 通过 Level 过滤日志
    log.FilterKey("password"),      // 通过 Key 过滤日志
    log.FilterValue("123456"),      // 通过 Value 过滤日志
    log.FilterFunc(func(level Level, keyvals ...interface{}) bool {  // 通过自定义 FilterFunc
        return level == log.LevelError
    },
)
logger.Log(log.LevelInfo, "password", "123456") // 输出格式为：password=***
```

# 日志接口设计

为了方便打印传统的日志，我们也可以通过 log.NewHelper() 进行封装日志接口，并且提供 传统日志 和 结构化日志 的使用。

```
logger := log.NewHelper(
    log.NewFilter(log.DefaultLogger, log.FilterLevel(log.LevelWarn)),
)
logger.Log(LevelInfo "msg", "info log")   // 通过指定 Level 等级输出
logger.Info("info log")                   // 通过 Info 等级输出
logger.Infof("info %s", "log")            //  通过 Info 等级 和 fmt.Sprinf 输出
logger.Infow("msg", "info log")           // 通过 结构化日志 输出
```

# 日志接口设计

通常在使用日志过程中，我们需要 log.With() 和 Hook 定制 Fields，例如：timestamp、caller、trace 等等。

在 Kratos 日志中，主要通过实现 Valuer 进行定制化：

```
type Valuer func(ctx context.Context) interface{}

func Value(ctx context.Context, v interface{}) interface{} {
    if v, ok := v.(Valuer); ok {
        return v(ctx)
    }
    return v
}
```

```
// Timestamp returns a timestamp Valuer with a custom time format.
func Timestamp(layout string) Valuer {
        return func(context.Context) interface{} {
                return time.Now().Format(layout)
        }
}

// TraceID returns a traceid valuer.
func TraceID() Valuer {
        return func(ctx context.Context) interface{} {
                if span := trace.SpanContextFromContext(ctx); span.HasTraceID() {
                        return span.TraceID().String()
                }
                return ""
        }
}
```

# 日志接口设计

所以，我们在 Kratos 可以这样子使用日志：

```
logger := log.NewStdLogger(os.Stdout)
logger = log.NewFilter(logger, log.FilterLevel(log.LevelInfo))
logger = log.With(logger,
        "app", "helloworld",
        "ts", log.DefaultTimestamp,
        "caller", log.DefaultCaller,
        "trace_id", log.TraceID(),
        "span_id", log.SpanID(),
    )
helper := log.NewHelper(logger)
helper.WithContext(ctx).Info("info log")
```

# Metadata 传递和使用

微服务之间主要通过 HTTP/gRPC 进行接口交互，所有在服务构架应该进行统一 Metadata 传递使用。

在 HTTP/gRPC 中，其实是通过 HTTP Header 进行传递，在框架中先通过 metadata 包封装成 key/value 结构，然后携带到 Transport Header 中。

```
// Metadata is our way of representing request headers internally.
// They're used at the RPC level and translate back and forth
// from Transport headers.
type Metadata map[string]string
```

Metadata 默认 Key 格式为：

- *x-md-global-xxx，全局传递，例如* ***mirror/color/criticality***
- *x-md-local-xxx，局部传递，例如* ***caller***

*可以在 middleware/metadata 定制自己的 key prefix，配置固定的元信息传递*

# Metadata 传递和使用

Metadata 的主要用法为：

- *配置 **client/server** 对应的 middleware/metadata 插件，可以自定义传递 **key prefix**，或者 **metadata** 常量，例如 **caller**。*
- *然后通过 metadata 包，NewClientContext 或者 FromServerContext 进行配置或者获取。*

```go
// SayHello implements helloworld.GreeterServer
func (s *server) SayHello(ctx context.Context, in *helloworld.HelloRequest) (*helloworld.HelloReply, error) {
        var extra string
        if md, ok := metadata.FromServerContext(ctx); ok {
                extra = md.Get("x-md-global-extra")
        }
        info, _ := kratos.FromContext(ctx)
        return &helloworld.HelloReply{Message: fmt.Sprintf("Hello %s extra: %s name: %s", in.Name, extra, info.Name())}, nil
}

func main() {
        grpcSrv := grpc.NewServer(
                grpc.Address(":9000"),
                grpc.Middleware(
                        mmd.Server(),
                ))
        httpSrv := http.NewServer(
                http.Address(":8000"),
                http.Middleware(
                        mmd.Server(),
                ),
        )
```

```go
conn, err := grpc.DialInsecure(
        context.Background(),
        grpc.WithEndpoint("127.0.0.1:9000"),
        grpc.WithMiddleware(
                mmd.Client(),
        ),
)
if err != nil {
        log.Fatal(err)
}
client := helloworld.NewGreeterClient(conn)
ctx := context.Background()
ctx = metadata.AppendToClientContext(ctx, "x-md-global-extra", "2233")
reply, err := client.SayHello(ctx, &helloworld.HelloRequest{Name: "kratos"})
if err != nil {
        log.Fatal(err)
}
```

# Transport HTTP/gRPC

**Kratos** 框架对传输层进行了抽象，用户也可以实现自己的传输层，框架默认实现了 **gRPC** 和 **HTTP** 两种通信协议传输层。

Transport 主要的接口：

- *Server*
  - *服务的启动和停止，用于管理服务生命周期。*
- *Transporter*
  - *Kind，代表实现的通讯协议的类型。*
  - *Endpoint，提供的服务终端地址。*
  - *Operation，用于标识服务的方法路径*
  - *Header，请求头的元数据*
- *Endpointer*
  - *用于实现注册到注册中心的终端地址*
  - *如果不实现这个方法则不会注册到注册中心*

```go
// Server is transport server.
type Server interface {
        Start(context.Context) error
        Stop(context.Context) error
}

// Endpointer is registry endpoint.
type Endpointer interface {
        Endpoint() (*url.URL, error)
}

// Header is the storage medium used by a Header.
type Header interface {
        Get(key string) string
        Set(key string, value string)
        Keys() []string
}

// Transporter is transport context value interface.
type Transporter interface {
        // grpc
        // http
        Kind() Kind
        // Server Transport: grpc://127.0.0.1:9000
        // Client Transport: discovery:///provider-demo
        Endpoint() string
        // Service full method selector generated by protobuf
        // example: /helloworld.Greeter/SayHello
        Operation() string
        // http: http.Header
        // grpc: metadata.MD
        Header() Header
}
```

# Middleware 插件化的使用

Kratos 内置了一系列的中间件用于处理日志、指标、跟踪链等通用场景。用户也可以通过实现 **Middleware** 接口，开发自定义 **middleware**，进行通用的业务处理，比如用户鉴权等。

主要的内置中间件：

- *recovery，用于 **recovery panic***
- *tracing，用于启用 **trace***
- *logging，用于请求日志的记录*
- *metrics，用于启用 **metrics***
- *validate，用于处理参数校验*
- *metadata，用于启用元信息传递*

```go
// Handler defines the handler invoked by Middleware.
type Handler func(ctx context.Context, req interface{}) (interface{}, error)

// Middleware is HTTP/gRPC transport middleware.
type Middleware func(Handler) Handler

// Chain returns a Middleware that specifies the chained handler for endpoint.
func Chain(m ...Middleware) Middleware {
        return func(next Handler) Handler {
                for i := len(m) - 1; i >= 0; i-- {
                        next = m[i](next)
                }
                return next
        }
}
```

# Kratos 启动管理器

在 **Kratos** 中，可以通过实现 transprt.Server 接口，然后通过 kratos.New 启动器进行管理服务生命周期。

启动器主要处理：

- *server 生命周期管理*
- *registry 注册中心管理*

```go
// Server is transport server.
type Server interface {
        Start(context.Context) error
        Stop(context.Context) error
}

// Endpointer is registry endpoint.
type Endpointer interface {
        Endpoint() (*url.URL, error)
}
```

```go
func main() {
        s := &server{}
        grpcSrv := grpc.NewServer(
                grpc.Address(":9000"),
                grpc.Middleware(
                        recovery.Recovery(),
                ),
        )
        httpSrv := http.NewServer(
                http.Address(":8000"),
                http.Middleware(
                        recovery.Recovery(),
                ),
        )
        helloworld.RegisterGreeterServer(grpcSrv, s)
        helloworld.RegisterGreeterHTTPServer(httpSrv, s)

        app := kratos.New(
                kratos.Name(Name),
                kratos.Server(
                        httpSrv,
                        grpcSrv,
                ),
        )

        if err := app.Run(); err != nil {
                log.Fatal(err)
        }
}
```

# References


https://www.ardanlabs.com/blog/2017/02/package-oriented-design.html

https://www.ardanlabs.com/blog/2017/01/develop-your-design-philosophy.html

https://www.ardanlabs.com/blog/2017/02/design-philosophy-on-packaging.html

https://docs.microsoft.com/en-us/previous-versions/msp-n-p/ee658109(v=pandp.10)

https://github.com/danceyoung/paper-code/blob/master/package-oriented-design/packageorienteddesign.md

https://go-kratos.dev/blog/go-project-layout/

https://go-kratos.dev/docs/