kratos框架介绍及示例

Go微服务课程学习资料，七米整理，更多Go学习资料👉 liwenzhou.com

资料

kratos官方资料

https://github.com/go-kratos/kratos

https://go-kratos.dev/

复习资料

Protocol Buffers V3语法

前置依赖

推荐安装并使用最新稳定版本的Go。

推荐国内同学配置国内的GOPROXY节点。

```
go env -w GO111MODULE=on
go env -w GOPROXY=https://goproxy.cn,direct
```

安装CLI脚手架工具

https://go-kratos.dev/docs/getting-started/usage

kratos 是与 Kratos 框架配套的脚手架工具，kratos 能够

- 通过模板快速创建项目
- 快速创建与生成 protoc 文件
- 使用开发过程中常用的命令
- 极大提高开发效率，减轻心智负担

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

快速开始

创建项目

1. 创建一个名为helloworld的项目

```
kratos new helloworld
```

上面的命令会从github拉取项目目录模板，国内网络可能会失败。如果失败使用以下命令指定从gitee拉取。

```
# 如在国内环境拉取失败，可 -r 指定源
kratos new helloworld -r https://gitee.com/go-kratos/kratos-layout.git
```

2. 进入项目目录

```
cd helloworld
```

3. 拉取项目依赖

```
go mod download
```

生成代码

生成所有proto源码、wire等等

```
go generate ./...
```

运行

使用kratos运行项目

```
kratos run
```

或者手动编译后执行

```
# 编译
go build -o ./bin/ ./...
# 执行
./bin/helloworld -conf ./configs/config.yaml
```

输出

```
# 输出
INFO msg=config loaded: config.yaml format: yaml # 默认载入 configs/config.yaml 配置文件
INFO msg=[gRPC] server listening on: [::]:9000 # gRPC服务监听 9000 端口
INFO msg=[HTTP] server listening on: [::]:8000 # HTTP服务监听 8000 端口
```

项目的目录结构

```
.
├── Dockerfile
├── LICENSE
├── Makefile
├── README.md
├── api // 下面维护了微服务使用的proto文件以及根据它们所生成的go文件
|   └── helloworld
|       └── v1
|           ├── error_reason.pb.go
|           ├── error_reason.proto
|           ├── greeter.pb.go
|           ├── greeter.proto
|           ├── greeter_grpc.pb.go
|           └── greeter_http.pb.go
├── bin // 编译好的二进制可执行文件存放目录
|   └── helloworld
├── cmd // 整个项目启动的入口文件
|   └── helloworld
|       ├── main.go
|       ├── wire.go
|       └── wire_gen.go
├── configs // 配置文件目录
|   └── config.yaml
├── go.mod
├── go.sum
├── internal  // 该服务所有不对外暴露的代码，通常的业务逻辑都在这下面，使用internal避免错误引用
|   ├── biz
|   |   ├── README.md
|   |   ├── biz.go
|   |   └── greeter.go
|   ├── conf // 内部使用的config的结构定义，使用proto格式生成
|   |   ├── conf.pb.go
|   |   └── conf.proto
|   ├── data // 业务数据访问，包含 cache、db 等封装，实现了 biz 的 repo 接口。
|   |   ├── README.md
|   |   ├── data.go
|   |   └── greeter.go
|   ├── server // http和grpc实例的创建和配置
|   |   ├── grpc.go
```

```
|   |   ├── http.go
|   |   └── server.go
|   └── service // 实现了 api 定义的服务层，格式化输出数据&协同各类 biz 交互，但是不应处理复杂逻辑
|       ├── README.md
|       ├── greeter.go
|       └── service.go
├── openapi.yaml
└── third_party // api 依赖的第三方proto
    ├── README.md
    ├── errors
    |   └── errors.proto
    ├── google
    |   ├── api
    |   |   ├── annotations.proto
    |   |   ├── client.proto
    |   |   ├── field_behavior.proto
    |   |   ├── http.proto
    |   |   └── httpbody.proto
    |   └── protobuf
    |       ├── any.proto
    |       ├── api.proto
    |       ├── compiler
    |       |   └── plugin.proto
    |       ├── descriptor.proto
    |       ├── duration.proto
    |       ├── empty.proto
    |       ├── field_mask.proto
    |       ├── source_context.proto
    |       ├── struct.proto
    |       ├── timestamp.proto
    |       ├── type.proto
    |       └── wrappers.proto
    ├── openapi
    |   └── v3
    |       ├── annotations.proto
    |       └── openapi.proto
    └── validate
        ├── README.md
        └── validate.proto
```

依赖注入工具wire

安装wire

依赖注入工具，必须安装。

```
go install github.com/google/wire/cmd/wire@latest
```

开始新的项目

使用CLI工具创建项目，我们要新建一个名为`bubble`的项目，那么就执行下面的命令：

```
kratos new bubble
```

创建好之后，执行以下命令切换到项目目录下。

```
cd bubble
```

定义API&生成代码

定义proto文件

1. 先添加模板，在项目目录下执行以下命令。

   ```
   kratos proto add api/bubble/v1/todo.proto
   ```

   此时会在项目目录的`api/bubble/v1`目录下创建一个`todo.proto`文件。

   > 你必须提前掌握 protobuf v3的语法、以及 如何使用注释指定HTTP/JSON到gRPC转换。

2. 根据自己的业务来修改`api/bubble/v1/todo.proto`文件。

   注意：添加HTTP注解需要导入`google/api/annotations.proto`

   ```
   syntax = "proto3";

   package api.bubble.v1;

   import "google/api/annotations.proto";

   option go_package = "bubble/api/bubble/v1;v1";
   option java_multiple_files = true;
   option java_package = "api.bubble.v1";

   service Todo {
     rpc CreateTodo (CreateTodoRequest) returns (CreateTodoReply){
       option (google.api.http) = {
         post: "/v1/todo",
         body: "*",
       };
     }
   ```

```
  rpc UpdateTodo (UpdateTodoRequest) returns (UpdateTodoReply){
    option (google.api.http) = {
      put: "/v1/todo/{id}",
      body: "*",
    };
  }
  rpc DeleteTodo (DeleteTodoRequest) returns (DeleteTodoReply){
    option (google.api.http) = {
      delete: "/v1/todo/{id}",
    };
  }
  rpc GetTodo (GetTodoRequest) returns (GetTodoReply){
    option (google.api.http) = {
      get: "/v1/todo/{id}",
    };
  }
  rpc ListTodo (ListTodoRequest) returns (ListTodoReply){
    option (google.api.http) = {
      get: "/v1/todos",
    };
  }
}

message todo {
  int64 id = 1;
  string title = 2;
  bool status = 3;
}

message CreateTodoRequest {
  string title = 1;
}
message CreateTodoReply {
  todo todo = 1;
}

message UpdateTodoRequest {
  int64 id = 1;
  string title = 2;
  bool status = 3;
}
message UpdateTodoReply {
}

message DeleteTodoRequest {
  int64 id = 1;
}
message DeleteTodoReply {
}
```

```
message GetTodoRequest {
  int64 id = 1;
}
message GetTodoReply {
  todo todo = 1;
}

message ListTodoRequest {}
message ListTodoReply {
  repeated todo data = 1;
}
```

生成代码

生成proto代码

根据你写的proto文件生成go代码。有以下两种方式，根据自己的实际情况任选一个。

1. 第一种方式是使用make命令。前提是你电脑上有`make`命令并且项目根目录下有`Makefile`文件。

```
make api
```

2. 第二种方式是使用kratos命令，需要指定你的proto文件。

```
kratos proto client api/bubble/v1/todo.proto
```

上述命令执行完后会生成以下文件。

```
api/bubble/v1/todo.pb.go
api/bubble/v1/todo_grpc.pb.go
# 注意 http 代码只会在 proto 文件中声明了 http 时才会生成
api/bubble/v1/todo_http.pb.go
```

生成Service代码

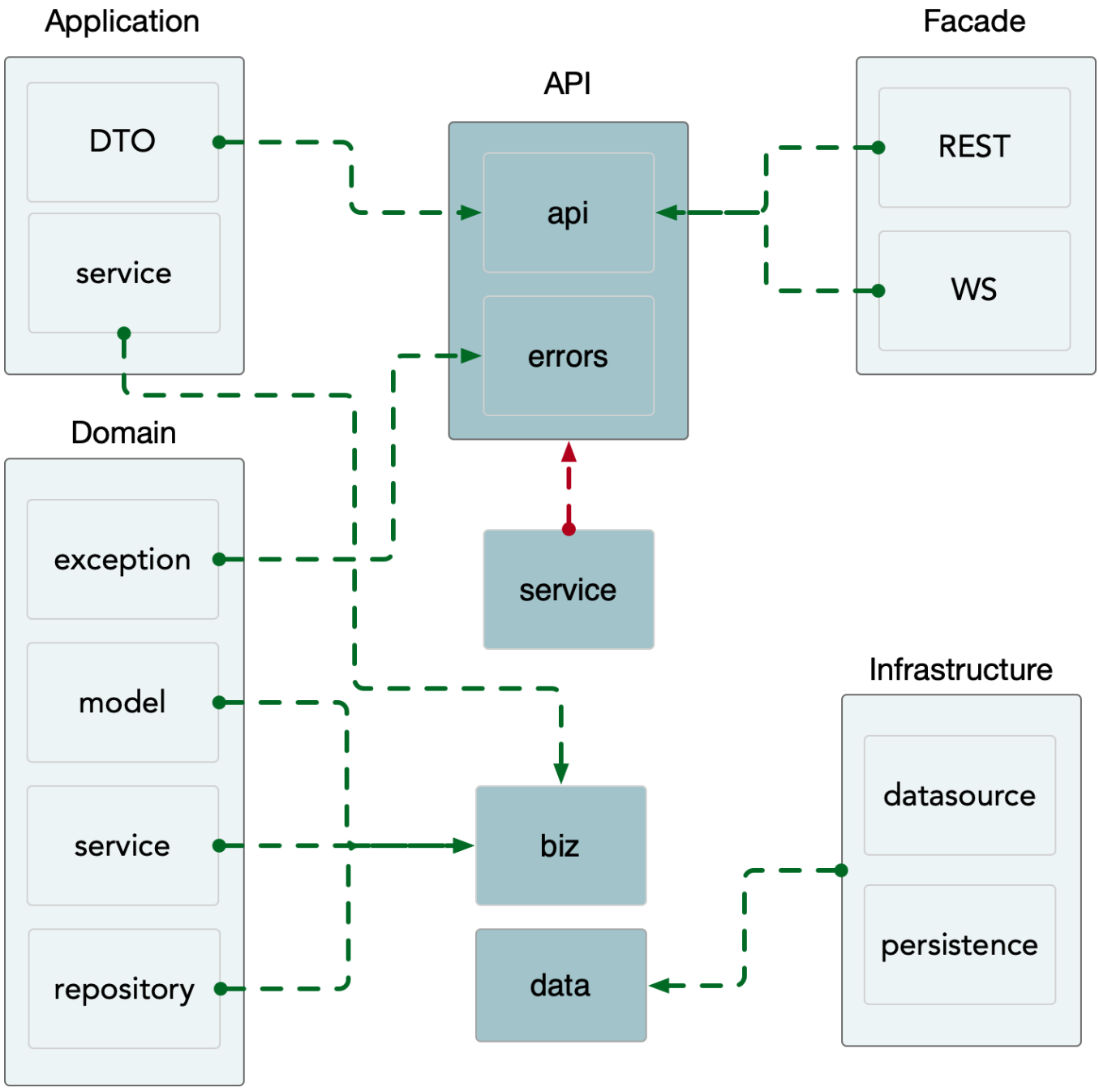
通过你的`proto`文件，可以直接生成对应的 Service 实现代码：

使用 kratos 命令并且通过 `-t` 指定生成代码的保存目录

```
kratos proto server api/bubble/v1/todo.proto -t internal/service
```

上述命令会在项目的 internal/service 目录下生成 todo.go 文件。

项目结构



依赖注入

kratos中使用 wire 实现依赖注入。

wire介绍和用法：https://www.liwenzhou.com/posts/Go/wire/

wire的使用：

1. 一般大型项目会用到。
2. 编写代码时注意使用依赖注入，把用到的依赖项使用参数传入，而不是自己直接写死。
3. 使用wire把构造函数连接起来，编写一个注入器。

4. 命令行工具wire生成Go代码到wire_gen.go文件。
5. 调用`wire_gen.go`中生成的函数。

注意的地方：

1. wire.go 最上面要加`//go:build wireinject`
2. wire.go 需要和最终产出对象在同一个包内。在哪里用就在哪里创建wire.go文件。

推荐阅读：

https://github.com/google/wire/blob/main/_tutorial/README.md

https://segmentfault.com/a/1190000039185137

https://segmentfault.com/a/1190000016140106?utm_source=sf-similar-article

配置

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

记住三个关键点和一个核心流程！

三个关键点

1. 配置源
    1. 文件json yaml
    2. 环境变量 env
    3. 配置中心 apollo consul
2. 读取配置
    1. c.Load 和 c.Scan, 取值c.Value
    2. c.Watch 配置热加载
    3. 合并
    4. Resolver 自定义
    5. Codec自定义
3. 配置结构体
    1. `internal/conf/conf.pb.go` 文件中的`Bootstrap`

```go
    // 创建配置对象
    c := config.New(
        config.WithSource(
            file.NewSource(flagconf), // 指定配置的来源
        ),
    )
    defer c.Close()
    // 加载配置（从配置文件/配置中心/环境变量加载配置）
    if err := c.Load(); err != nil {
        panic(err)
    }

    // 创建配置结构体变量bc
    var bc conf.Bootstrap
    // 将配置数据扫描到结构体变量bc中
    if err := c.Scan(&bc); err != nil {
        panic(err)
    }
```

一个核心流程

1. 程序添加配置的核心流程
    1. 修改`internal/conf/conf.proto`文件， 用pb定义配置
    2. 生成配置对应的Go代码
        - 项目目录下执行`make config`
        - 如果没有装make命令，那么就执行

        ```
        protoc --proto_path=./internal \
                       --proto_path=./third_party \
                       --go_out=paths=source_relative:./internal \
                       internal/conf/conf.proto
        ```

    3. 修改configs配置文件

注意事项：

1. pb文件要语法写对了

2. 改完pb文件一定记得要生成Go代码
3. 配置文件要跟Go代码里的结构体对应上！
4. 配置文件的语法要写对。

业务逻辑开发

牢记一个思路！

顺着请求流程，去写代码。

service →biz→data

- service：服务的入口，实现了API层定义的服务
- biz：业务逻辑层，复杂的业务逻辑写在这里
- data：数据层，数据（data、cache）有关的操作写在这里

小清单，待办事项的增删改查

数据表，三个字段。

- id: 唯一id
- title：吃饭
- status: true

如何自定义HTTP返回

可以覆盖默认的 DefaultResponseEncoder, 通过 http.ResponseEncoder() 配置，注入到 http.Server() 中

自定义HTTP响应的格式

什么时候用这个?

1. 你需要对外提供一套HTTP RESTful接口
2. 对外提供的RESTful 接口 要求有一套固定格式的响应数据。

```
{
  "code": 200,
  "msg": "success",
  "data": []
}
```

kratos默认的HTTP响应编码器

```
func DefaultResponseEncoder(w http.ResponseWriter, r *http.Request, v interface{})
error {
  if v == nil {
    return nil
  }
  if rd, ok := v.(Redirector); ok {
    url, code := rd.Redirect()
    http.Redirect(w, r, url, code)
    return nil
  }
  codec, _ := CodecForRequest(r, "Accept")
  data, err := codec.Marshal(v)
  if err != nil {
    return err
  }
  w.Header().Set("Content-Type", httputil.ContentType(codec.Name()))
  _, err = w.Write(data)
  if err != nil {
    return err
  }
  return nil
}
```

自定义的HTTP响应编码器

```
// 自定义编码器
func responseEncoder(w http.ResponseWriter, r *http.Request, v interface{}) error {
  if v == nil {
    return nil
  }
  // 判断是不是重定向
  if rd, ok := v.(kratoshttp.Redirector); ok {
    url, code := rd.Redirect()
    http.Redirect(w, r, url, code)
    return nil
  }
  // 构造自定义的相应结构体
  resp := &httpResponse{
    Code: http.StatusOK,
    Msg:  "success",
    Data: v,
  }
  codec, _ := kratoshttp.CodecForRequest(r, "Accept")
  data, err := codec.Marshal(resp) // json.Marshal
  if err != nil {
    return err
  }
  // 设置响应头 Content-Type:application/json
```

```
  w.Header().Set("Content-Type", "application/"+codec.Name())
  _, err = w.Write(data)
  return err
}
```

自定义HTTP错误响应

原理和自定义HTTP响应一样的。

```
// 自定义的错误响应编码器
func errorEncoder(w http.ResponseWriter, r *http.Request, err error) {
  if err == nil {
    return
  }
  resp := new(httpResponse)
  // 能从err里面解析出错误码的
  if gs, ok := status.FromError(err); ok {
    resp = &httpResponse{
      Code: kratosstatus.FromGRPCCode(gs.Code()),
      Msg:  gs.Message(),
      Data: nil,
    }
  } else {
    resp = &httpResponse{
      Code: http.StatusInternalServerError, // 500
      Msg:  "内部错误",
    }
  }

  codec, _ := kratoshttp.CodecForRequest(r, "Accept")
  body, err := codec.Marshal(resp)
  if err != nil {
    w.WriteHeader(http.StatusInternalServerError)
    return
  }
  w.Header().Set("Content-Type", "application/"+codec.Name())
  w.WriteHeader(resp.Code)
  _, _ = w.Write(body)
}
```

注册自定义的编码器

在 `internal/server/http.go` 中

```
// 替换默认的HTTP响应编码器
opts = append(opts, http.ResponseEncoder(responseEncoder))
// 替换默认的错误响应编码器
opts = append(opts, http.ErrorEncoder(errorEncoder))
```

自定义错误状态码枚举

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

安装生成错误状态码代码的工具

```
go install github.com/go-kratos/kratos/cmd/protoc-gen-go-errors/v2@latest
```

定义错误码 proto 文件

```
syntax = "proto3";

package api.bubble.v1;

import "errors/errors.proto";

option go_package = "bubble/api/bubble/v1;v1";
option java_multiple_files = true;
option java_package = "api.bubble.v1";


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
    TODO_NOT_FOUND = 0 [(errors.code) = 404];

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

生成错误相关Go代码

```
protoc --proto_path=. \
         --proto_path=./third_party \
         --go_out=paths=source_relative:. \
         --go-errors_out=paths=source_relative:. \
         api/bubble/v1/todo_error.proto
```

日志

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

四个类型

- Logger - 适配各种日志输出方式
- Helper - 在项目代码中打日志
- Valuer 设置全局字段
- Filter 日志过滤

两种用法

- 简单/小项目用全局日志方式
- kratos-layout

中间件

https://go-kratos.dev/docs/component/middleware/overview

中间件的原理

```
// Middleware 自定义中间件
// type middleware func(Handler) Handler
// type Handler func(ctx context.Context, req interface{}) (interface{}, error)
func Middleware() middleware.Middleware {
  return func(handler middleware.Handler) middleware.Handler {
    return func(ctx context.Context, req interface{}) (interface{}, error) {
      // 执行之前做点事
      fmt.Println("Middleware：执行handle之前")
      defer func() {
        fmt.Println("Middleware：执行handle之后")
      }()
      return handler(ctx, req) // 执行目标handler
    }
  }
}
```

中间件的自定义注册

```
import "github.com/go-kratos/kratos/v2/middleware/selector"

grpc.Middleware(
            selector.Client(recovery.Recovery(), tracing.Server(),testMiddleware).
                Path("/hello.Update/UpdateUser", "/hello.kratos/SayHello").
                Regex(`/test.hello/Get[0-9]+`).
                Prefix("/kratos.", "/go-kratos.", "/helloworld.Greeter/").
                Build(),
        )
```

—— 七米整理，更多Go学习资料👉 liwenzhou.com