hydra服务开发规范

创建hydra服务需要如下步骤:

1. 创建hydra.MicroApp实例，并指定平名名称，系统名称，服务器类型，集群名称等参数，并调用Start()
2. 通过app.Conf...指定配置参数。如启动端口，数据库连接串等，需区分开发环境与生产环境配置
3. 处理服务运行时钩子，如启动前检查配置是否正确，每个请求处理前检查权限或验证合法性等
4. 编写服务实现函数或类，并注册到app中

推荐项目结构如下:

|-----main.go

|-----init.go

|-----handing.go

|-----conf.dev.go

|-----conf.prod.go

|-----services

|----------order

|---------------order.handler.go

|-----modules

|----------const

|--------------keys

|------------------cache.keys.go

|--------------sqls

|------------------order.go

|----------order

|---------------order.go

|----------order.db.go

|----------order.remote.go

|----------order.cache.go

文件/目录	说明
main.go	应用入口。 创建、启动服务器实例
init.go	应用初始化。检查配置是否正确，注册服务等
conf.dev.go	开发环境配置
conf.prod.go	生产环境配置
handing.go	请求预处理。如登录检查，权限检查等
services	服务目录。存放所有服务实现类，init.go直接调用进行注册
modules	业务模块目录。业务具体实现，如数据库操作，缓存操作，队列发送等
modules/const	存放业务操作的静态配置信息。如sql语句、缓存key、系统枚举、错误码等

1. hydra实例创建与启动

`main.go` 中存放MicroApp实例代码,建议对APP对象进行包装：

```
package main

import "github.com/micro-plat/hydra/hydra"

type apiserver struct {
        *hydra.MicroApp
}

func main() {
        app := &apiserver{
                hydra.NewApp(
                        hydra.WithPlatName("mall"),
                        hydra.WithSystemName("apiserver"),
                        hydra.WithServerTypes("api")),
        }
        app.Start()
}
```

2. 启动和运行时参数配置

conf.dev.go 和 conf.prod.go 存放配置数据。
并使用 +build 进行条件编译

conf.dev.go 内容如下:

```
// +build !prod

package main

func (api *apiserver) config() {
    api.IsDebug = true //打印详细的日志信息
        api.Conf.API.SetMainConf(`{"address":":8090"}`)
        api.Conf.Plat.SetVarConf("db", "db", `{
                        "provider":"mysql",
                        "connString":"mrss:123456@tcp(192.168.0.36)/mrss?charset=utf8",
                        "maxOpen":20,
                        "maxIdle":10,
                        "lifeTime":600
        }`)
}
```

> 配置中指定了 api server 的启动端口和数据库连接串，数据库连接串属于平台公共配置，只能通过 app.Conf.Plat.SetVarConf 进行指定，当前示例指定的配置将创建到 /mall/var/db/db 节点。请通过 conf 命令查看。

conf.prod.go 内容如下:

```
// +build prod

package main

func (api *apiserver) config() {
        api.Conf.API.SetMainConf(`{"address":":8090"}`)
        api.Conf.Plat.SetVarConf("db", "db", `{
                        "provider":"mysql",
                        "connString":"#connString",
                        "maxOpen":200,
                        "maxIdle":20,
                        "lifeTime":600
        }`)
}
```

> 只有编译时指定 prod tag时，才会使用 conf.prod.go 内容。请使用 go install -tags "prod" 或 go build -tags "prod" 进行编译安装

> 由于生产环境的数据库连接信息不能直接写在代码中，可通过 # 开头的变量名替代，发布时调用 install 或 registry 会引导用户通过终端输入。

3. 初始化检查与服务注册

服务初始化时需检查配置是否正确，拉取应用配置，注册服务等。

```
package main

import "github.com/micro-plat/hydra/component"

//init 检查应用程序配置文件，并根据配置初始化服务
func (api *apiserver) init() {
        api.Initializing(func(c component.IContainer) error {
                //检查db配置是否正确
                if _, err := c.GetDB(); err != nil {
                        return err
                }
                //检查消息队列配置

                //拉取应用程序配置

                return nil
        })
        //服务注册
}
```

4. 请求预处理

检查用户是否登录，是否有权限等

```
package main

import (
        "github.com/micro-plat/hydra/context"
)

func (api *apiserver) handing() {
        api.Handling(func(ctx *context.Context) (rt interface{}) {
                //检验登录状态或权限
                return nil
        })
}
```

5. 微服务入口

服务的实现一般会分多层实现，对于入口层服务负责参数的检查，转换，业务逻辑的调用，和结果输出。此层服务可编写服务类进行实现：

```
package order

import (
        "github.com/micro-plat/hydra/component"
        "github.com/micro-plat/hydra/context"
)

type RequestHandler struct {
        container component.IContainer
}

func NewRequestHandler(container component.IContainer) (u *RequestHandler, err error) {
        return &RequestHandler{container: container}, nil
}

//Handle .
func (u *RequestHandler) Handle(ctx *context.Context) (r interface{}) {
        return "success"
}
```

服务注册:

```
    api.Micro("/order/request", order.NewRequestHandler)
```

> 在 `Handle` 中实现服务逻辑

> 使用app.Micro("/order/request",order.NewRequestHandler)进行服务注册。NewRequestHandler可进行服务初始化，如初始化modules层提供的业务对象。初始化失败时返回error信息，服务启动时会显示错误消息，并提示启动失败。

6. 业务逻辑实现

在modules层，可实现数据库操作，远程服务访问，缓存处理等操作，但需放在不同的文件中。由业务外层代码统一调用。

modules/order/order.go 内容如下:

```go
package order

import "github.com/micro-plat/hydra/component"

type IOrder interface {
}

type Order struct {
        c component.IContainer
}

func NewOrder(c component.IContainer) *Order {
        return &Order{
                c: c,
        }
}
```

modules/order/order.db.go 内容如下:

```go
package order

import "github.com/micro-plat/hydra/component"

type IOrderDB interface {
}

type OrderDB struct {
        c component.IContainer
}

func NewOrderDB(c component.IContainer) *OrderDB {
        return &OrderDB{
                c: c,
        }
}
```

modules/order/order.cache.go 内容如下:

```
package order

import "github.com/micro-plat/hydra/component"

type IOrderCache interface {
}

type OrderCache struct {
        c component.IContainer
}

func NewOrderCache(c component.IContainer) *OrderCache {
        return &OrderCache{
                c: c,
        }
}
```

> component.IContainer提供了数据，缓存，消息队列等常用组件，这些组件进行了系统优化，并且直接从配置中拉取配置进行初始化，使用非常方便

数据库查询:

```
func (o *OrderDB) Query(orderNO string) (db.QueryRows, error) {
        db := o.c.GetRegularDB()
        rows, _, _, err := db.Query(sqls.ORDER_QUERY, map[string]interface{}{
                "order_no": orderNO,
        })
        return rows, err
}
```

> GetRegularDB 为获取已创建的DB对象，实际上在app.Initializing时调用GetDB()时已经创建了DB实例，此处只是从缓存中取得实例。

> 数据库事务及其它操作，自行参考db.IDB接口

缓存查询：

```
func (o *OrderCache) Query(orderNO string) (string, error) {
        cache := o.c.GetRegularCache()
        rows, err := cache.Get(keys.ORDER_QUERY)
}
```

> 其它缓存操作,自行学习

modules/const/keys/cache.keys.go内容如下:

```
package keys

const ORDER_QUERY = "MALL:APISERVER:ORDER:QUERY_BY_ORDER_NO"
```

modules/const/sqls/order.go内容如下:

```
package sqls

const ORDER_QUERY = "SELECT * FROM ORDER_MAIN T WHERE T.ORDER_NO=@order_no"
```

> 数据库操作使用了 `github.com/micro-plat/lib4go/db` 库。SQL语句中可使用 `@`，`#` 等占位符进行参数翻译处理。