构建API服务器一

本示例构建一个api服务器，提供两个API接口：会员充值订单提交，充值结果查询。接口通过md5签名验证。

接口地址	功能	说明
/order/request	充值订单提交	订单保存到数据库，并发送消息队列进行订单扣款
/order/query	充值结果查询	查询数据库返回订单信息

知识点:

- 数据库配置与操作
- 消息队列配置与发送
- 自定义服务名
- md5签名

1. 搭建基础代码

main.go

```go
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.init()
	app.Start()
}
```

> app.init 用于挂载服务配置，注册等处理

config.dev.go

```go
// +build !prod

package main

func (api *apiserver) config() {
	api.IsDebug = true
	api.Conf.API.SetMainConf(`{"address":":8090","trace":true}`)
	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
	}`)
	api.Conf.Plat.SetVarConf("queue", "queue", `
		{
				"proto":"redis",
				"addrs":[
						"192.168.0.111:6379",
						"192.168.0.112:6379",
						"192.168.0.113:6379"
				],
				"db":1,
				"dial_timeout":10,
				"read_timeout":10,
				"write_timeout":10,
				"pool_size":10
	}
	`)
}
```

> 数据库配置，消息队列等属于平台共用配置，需使用 `api.Conf.Plat` 提供的函数进行设置

config.prod.go

```go
// +build prod

package main

func (api *apiserver) config() {
        api.Conf.API.SetMainConf(`{"address":":8090","trace":true}`)
        api.Conf.Plat.SetVarConf("db", "db", `{
                        "provider":"mysql",
                        "connString":"#connString",
                        "maxOpen":20,
                        "maxIdle":10,
                        "lifeTime":600
        }`)
        api.Conf.Plat.SetVarConf("queue", "queue", `
                {
                        "proto":"redis",
                        "addrs":[
                                        #redis_addr
                        ],
                        "db":1,
                        "dial_timeout":10,
                        "read_timeout":10,
                        "write_timeout":10,
                        "pool_size":20
        }
        `)
}
```

2. 初始化检查与服务注册

```go
package main

import (
	"github.com/micro-plat/hydra/component"
	"github.com/micro-plat/hydra/quickstart/demo/apiserver11/services/order"
)

//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() {
	app.config()
	app.handling()

	api.Initializing(func(c component.IContainer) error {
		//检查db配置是否正确
		if _, err := c.GetDB(); err != nil {
			return err
		}
		if _, err := c.GetQueue(); err != nil {
			return err
		}

		return nil
	})

	//服务注册
	api.Micro("/order", order.NewOrderHandler)
}
```

> 初始化时创建数据库，消息队列对象，创建失败则返回错误系统，系统则会启动失败

> Initializing函数为每个服务器都会执行，不同的服务器处理不同的逻辑，则可以使用 `component.IContainer` 提供的 `服务器类型` 等参数进行判断

3. 请求预处理，验证签名

```go
package main

import (
	"fmt"

	"github.com/micro-plat/hydra/context"
	"github.com/micro-plat/hydra/quickstart/demo/apiserver11/modules/merchant"
)

func (api *apiserver) handling() {
	api.MicroApp.Handling(func(ctx *context.Context) (rt interface{}) {
		if err := ctx.Request.Check("merchant_id"); err != nil {
			return err
		}
		key, err := merchant.GetKey(ctx,ctx.Request.GetInt(merchant_id))
		if err != nil {
			return err
		}
		if !ctx.Request.CheckSign(key) {
			return fmt.Errorf(908, "商户签名错误")
		}
		return nil
	})
}
```

3. 构建服务

`servers/order.go`

```go
package order

import (
	"github.com/micro-plat/hydra/component"
	"github.com/micro-plat/hydra/context"
	"github.com/micro-plat/hydra/quickstart/demo/apiserver10/modules/order"
)

type OrderHandler struct {
	container component.IContainer
	o         order.IOrder
}

func NewOrderHandler(container component.IContainer) (u *OrderHandler) {
	return &OrderHandler{
		container: container,
		o:         order.NewOrder(container),
	}
}

//RequestHandle 会员充值订单请求
func (u *OrderHandler) RequestHandle(ctx *context.Context) (r interface{}) {
	ctx.Log.Info("--------------会员充值订单请求---------------")
	ctx.Log.Info("1.检查请求参数")
	if err := ctx.Request.Check("merchant_id","order_no", "account", "face", "num");
		return context.NewError(context.ERR_NOT_ACCEPTABLE, err)
	}

	ctx.Log.Info("2. 创建充值订单")
	result, err := u.o.Create(
		ctx.Request.GetString("merchant_id")
		ctx.Request.GetString("order_no"),
		ctx.Request.GetString("account"),
		ctx.Request.GetInt("face"),
		ctx.Request.GetInt("num"))
	if err != nil {
		return err
	}
	return result
}

//QueryHandle 充值结果查询
func (u *OrderHandler) QueryHandle(ctx *context.Context) (r interface{}) {
	ctx.Log.Info("--------------充值结果查询---------------")
	ctx.Log.Info("1.检查请求参数")
	if err := ctx.Request.Check("merchant_id","order_no"); err != nil {
		return context.NewError(context.ERR_NOT_ACCEPTABLE, err)
	}

	ctx.Log.Info("2. 查询充值结果")
	result, err := u.o.Query(
                ctx.Request.GetString("merchant_id",
                ctx.Request.GetString("order_no"))
        if err != nil {
                return err
        }
        return result
}
```

> 返回 `error` 则http状态码为400，返回其它状态码可使用 `context.NewError` 设置，或使用 `ctx.Response.SetStatus` 设置

> 返回非 `error` 类型值，http状态码为200

> 使用ctx.Log进行日志输出，可保证同一个请求过程有相同的 `session id`，便于对执行流程进行准确分析

> ctx.Request中提供了请求参数获取，检查等功能

> ctx.Response可处理响应参数，如修改返回类型为 `json`，`xml`，`plain`， 设置状态码等

> RequestHandle, QueryHandle 在Handle前有其它名称（非GET,POST,PUT,DELETE）则会注册为路由的一部分。当前注册代码为 `api.Micro("/order", order.NewOrderHandler)`，则实际注册的服务有 `/order/request`，`/order/query`

4. 业务逻辑

保存订单到数据库，并发送到消息队列

根据请求参数，查询订单信息并返回

`modules/order/order.go` 调用数据库保存，并调用qtask提供的队列管理工具进行消息任务发送

```go
package order

import (
	"github.com/micro-plat/hydra/component"
	"github.com/micro-plat/hydra/quickstart/demo/apiserver11/modules/const/keys"
	"github.com/micro-plat/qtask"
)

type IOrder interface {
	Create(merchantID string, orderNO string, account string, face int, num int) (ma
	Query(merchantID string, orderNO string) (map[string]interface{}, error)
}

type Order struct {
	c  component.IContainer
	db IOrderDB
}

func NewOrder(c component.IContainer) *Order {
	return &Order{
		c:  c,
		db: NewOrderDB(c),
	}
}
func (d *OrderDB) Create(merchantID string, orderNO string, account string, face int, nu
	order, err := d.db.Create(merchantID, orderNO, account, face, num)
	if err != nil {
		return nil, err
	}
	qtask.Create(d.c, "订单支付", order, 60, keys.ORDER_PAY)
	return order, err
}
```

modules/order/order.db.go 保存订单信息

```go
package order

import (
	"fmt"

	"github.com/micro-plat/hydra/component"
	"github.com/micro-plat/hydra/quickstart/demo/apiserver11/modules/const/sqls"
)

type IOrderDB interface {
	Create(merchantID string, orderNO string, account string, face int, num int) (ma
	Query(merchantID string, orderNO string) (map[string]interface{}, error)
}

type OrderDB struct {
	c component.IContainer
}

func NewOrderDB(c component.IContainer) *OrderDB {
	return &OrderDB{
		c: c,
	}
}
func (d *OrderDB) Create(merchantID string, orderNO string, account string, face int, nu
	db := d.c.GetRegularDB()
	input := map[string]interface{}{
		"merchant_id": merchantID,
		"order_no":    orderNO,
		"account":     account,
		"face":        face,
		"num":         num,
	}
	orderID, _, _, err := db.Scalar(sqls.Get_ORDER_ID, input)
	if err != nil {
		return nil, err
	}
	input["order_id"] = orderID
	row, _, _, err := db.Execute(sqls.ORDER_CREATE, input)
	if err != nil || row == 0 {
		return nil, fmt.Errorf("系统错误暂时无法创建订单%v", err)
	}
	return map[string]interface{}{
		"order_id": orderID,
	}, nil
}

func (d *OrderDB) Query(merchantID string, orderNO string) (map[string]interface{}, erro
	db := d.c.GetRegularDB()
	row, sql, param, err := db.Execute(sqls.ORDER_QUERY, map[string]interface{}{})
	if err != nil {
		return nil, fmt.Errorf("sql执行错误:%s %+v,%v",sql,param,err)
```

```
        }
        if row.IsEmpty(){
                return nil,context.NewError(901,"订单不存在")
        }
        return row, nil
}
```

> 返回指定的状态码可使用 `context.NewError`

> 数据库执行失败可打印执行SQL与输入参数