CQRS简单入门

1. 简单入门之入门

CQRS/ES和领域驱动设计更搭，故整体分层沿用经典的DDD四层。其实要实现的功能概要很简单，如下图。



基础框架选择了https://github.com/looplab/eventhorizon，该框架功能强大、示例都挺复杂的，囊括的概念太多，不太适合入门，所以决定在其基础上，进行简化。

1. 简化使用eventhorizon

Eventhorizon已经提供了详尽的使用案例（https://github.com/looplab/eventhorizon/tree/master/examples），只是概念比较多，也不符合之前的一般使用方式，故按照概要图进行简化使用。

**1.presentation**

使用github.com/gin-gonic/gin，路由功能等，与业务无关都可以委托出去，同时抽象了一个核心的函数，作为衔接presentation 和application层。

从gin上下文中读取输入数据，并根据约定的Command Key,转交给application层进行相应的Command解析。

func handles(command string) gin.HandlerFunc {

return func(c \*gin.Context) {

data, err := c.GetRawData()

if err != nil {

c.JSON(http.StatusBadRequest, "")

return

}

result := application.HandCommand(data, command)

c.JSON(http.StatusOK, result)

}

}

**2. application**

Application很薄的一层，依然是与业务无关的，重点在于将计算机领域的数据、模型，转换为业务领域建模所需。

核心函数依然只有一个，主要功能为：创建正确的Command；将presentation层传递上来数据转为为领域层所需要的模型（Command来承载）；委托“命令总线”发布命令，不必关系命令的接收方会怎样，解除对命令执行方的依赖，只关心命令是否正确发送出去；想presentation层报告命令发布情况。

func HandCommand(postBody []byte, commandKey string) (result Result) {

cmd, err := eh.CreateCommand(eh.CommandType(commandKey))

if err != nil {

result.Msg = "could not create command: " + err.Error()

return

}

if err := json.Unmarshal(postBody, &cmd); err != nil {

result.Msg = "could not decode Json" + err.Error()

return

}

ctx := context.Background()

if err := bus.HandleCommand(ctx, cmd); err != nil {

result.Msg = "could not handle command: " + err.Error()

return

}

result.Succ = true

result.Msg = "ok"

return

}

**3. domain**

Domain层，核心的业务逻辑层，不进行累赘的表述，重点需要介绍下domain/Bus。总线也可以放置到infrastructure层，不过根据个人习惯写在了domain层里。

Domain/Bus，整个CQRS的核心、负责命令、事件的发布、注册等功能。核心功能主要有：命令的注册、命令的执行、事件的注册、事件的发布（异步）和存储、EventStore的构建等。

**4. infrastructure**

由于是简单入门infrastructure层进行了抽象简化，提供基本的仓储功能。领域层进行业务处理根据所需进行数据的持久化以及读取等。

1. 简要总结

一种方法是使其足够简单以至于不存在明显的缺陷，另外一种是使其足够复杂以至于看不出有什么问题。

以上组合已经能够支持最基础的CQRS/ES的概念和功能了。

现在看看如何利用已有的东西，对具体业务进行融合。

1. 小试牛刀

支付项目中第三方支付公司需要和客户进行签约。需要调用支付公司的接口将用户提交的基本信息提交给支付公司，支付公司发送验证码并告知用户须知，签约成功之后需要将协约基本信息进行保存，以后使用该协约进行代收付等资金业务。

单纯演示，将概要设计简化如下：获取客户端提交的用户信息，校验数据，调用第三方支付的接口，持久化到SQLite数据库，激活领域事件存储到MongoDB，领域事件的处理。

1. presentation

这里偷懒，没有进行API路由和命令的映射，统一使用了"/api/sign\_protocol"。核心代码注册API。

signProtocolAPI := "/api/sign\_protocol"

router.POST(signProtocolAPI, handles(signProtocolAPI))

2. application

Application层不需要额外代码

3. domain

domain层只需要commands.go、protocol.go；代码也很简单，command主要两个功能承载入参、承接应用层到聚合根。

protocol.go聚合根，主要的业务逻辑。这里也很简单，进行领域服务请求、并且进行持久化。

4. infrastructure

Infrastructure一般的持久化。