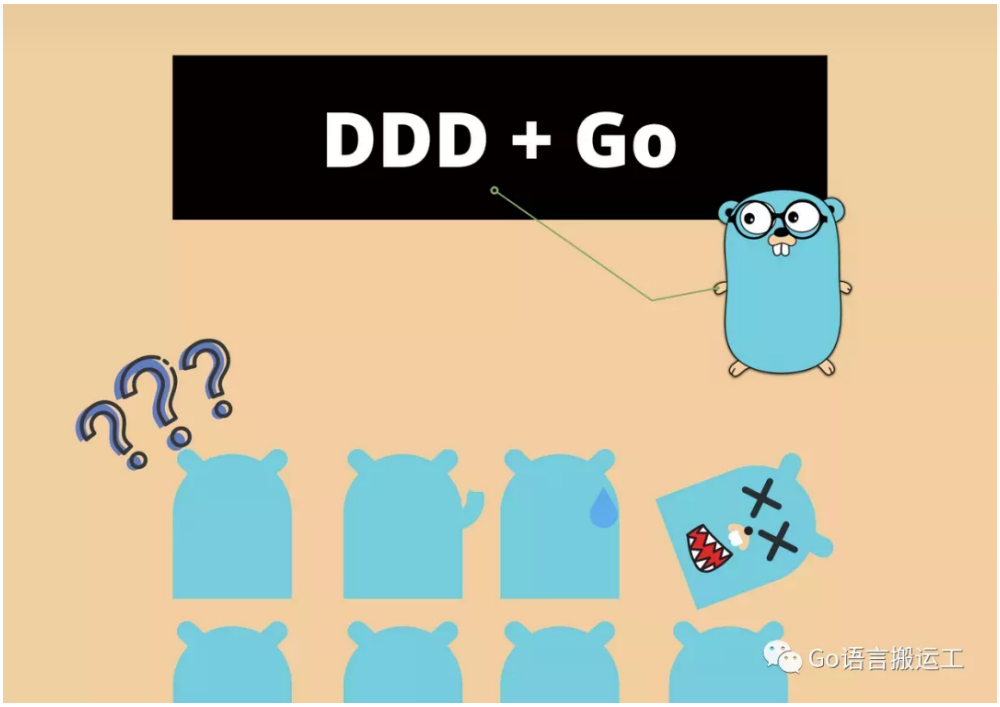


Go如何实现领域驱动设计(DDD)

码农小军

学习如何在Go应用程序中使用DDD的简单方法。

近年来，微服务已经成为一种非常流行的构建软件的方法。微服务用于构建可伸缩、灵活的软件。然而，跨多团队随机构建微服务可能会带来很大的挫折和复杂性。不久前我还没有听说过领域驱动设计——DDD，但现在无论走到哪里似乎每个人都在谈论它。

在本文，我将从头开始构建一个在线酒店应用来一步步地探索DDD的各种概念。希望每实现一部分，对理解DDD会更容易。采用这种方法的原因是，每次我在阅读DDD资料时都很头疼。有这么多的概念，很宽泛和不清楚，不清楚什么是什么。如果你不知道为什么我在研究DDD时头疼，下面的图可能会让你认识到这一点。

从上面的图片可以看得出来，为什么Eric Evans在他的《领域驱动设计：解决软件核心的复杂性》要用500页来解释什么是领域驱动设计。如果你对学习DDD有兴趣可以阅读本书。

首先，我想指出的是，本文描述了我对DDD的理解，我在本文中展示的实现是基于我对go相关项目的经验得出的最佳实践。我们将创建的实现绝不是社区所接受的最佳实践。我还将在项目中以DDD方法命名文件夹，以使其易于理解，但我不确定这是否是我想要的代码框架的样子。基于此，我将创建另一个分支来修正代码结构，这个重构将在其他文章解释。

我在网上看到很多关于DDD和如何正确实现的激烈讨论。让我印象深刻的是，多数时候人们似乎忘记了DDD背后的目的，都以讨论一些小的实现细节而告终。我认为重要的是遵循Evan提出的方法，而不是命名为X或Y。

DDD是一个很大的领域，我们将主要关注它的实现，但在我们实现任何东西之前，我将对DDD中的一些概念做一个快速的概述。

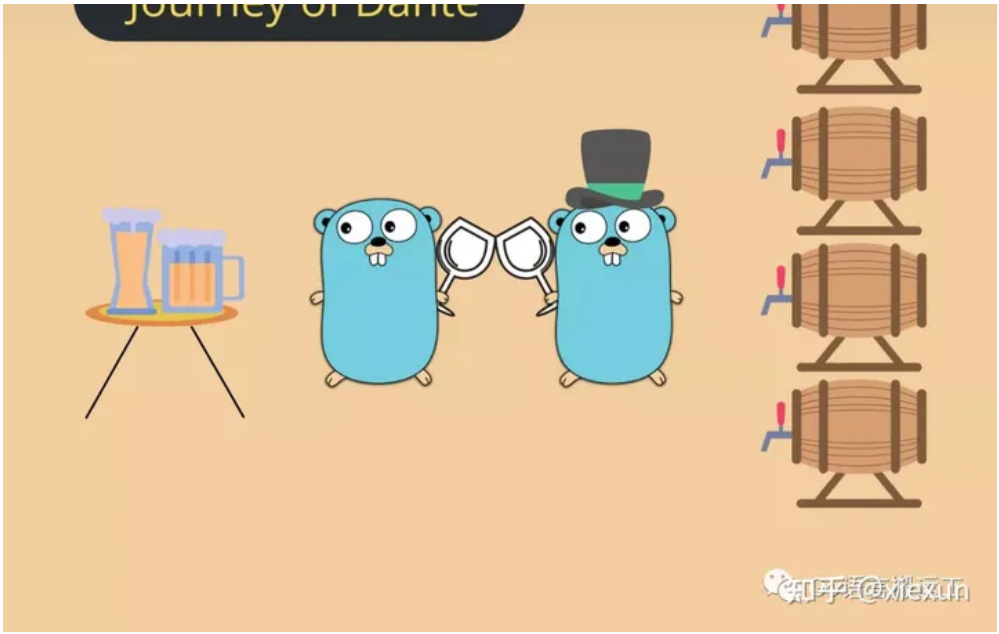
什么是DDD?

领域驱动设计是在软件所属领域之后对软件进行结构化和建模的一种方法。这意味着必须首先考虑所编写的软件的领域。领域是软件将处理的主题或问题。软件的编写应该反映该领域。

DDD主张工程团队必须与主题专家(SME)交谈，他们是领域内的专家。这样做的原因是SME拥有关于领域的知识，这些知识应该反映在软件中。想想看，如果我要做一个股票交易平台，作为一名工程师，我对这个领域的了解够不够去做一个好的股票交易平台？如果我能和沃伦·巴菲特谈谈这个领域，这个平台可能会好得多。

代码中的架构也应该反映领域。当我们开始编写我们的酒店应用时，我们将体会到领域内涵。

Gopher的DDD之路



让我们开始学习如何实现DDD，在开始之前我将给你讲述一个Gopher和Dante的故事，他们想创建一个在线酒店应用。Dante知道如何写代码，但是对如何运营一个酒店一无所知。

在Dante决定开始创建酒店应用的那天，他遇到了一个问题，从哪里开始，如何开始？他出去散步，思考这个问题。在公交站标志前等待的时候，一个戴大礼帽的男人走近Dante说：

> “看起来你好像在担心什么事情，年轻人，需要帮忙建一个酒店应用吗?”

Dante和大礼帽男一起散步，他们讨论了酒店以及如何经营。Dante问如何处理酒徒（drinker），大礼帽男纠正说是客户（Customer）不是酒徒（drinker）。大礼帽男还向Dante解释了酒店还需要一些东西来运作，比如顾客、员工、银行和供应商。

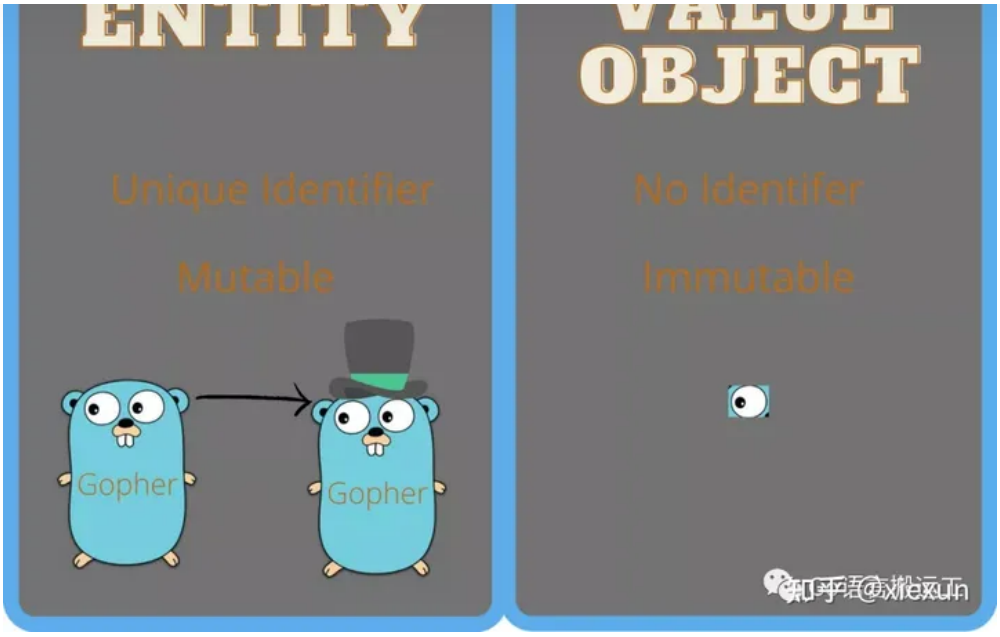
领域、模型、统一语言和子领域

我希望你们喜欢Dante的故事，我写它是有原因的。我们可以用这个故事来解释DDD中使用的一些概念，这些词如果没有上下文很难解释，比如一个短篇故事。

Dante和大礼帽男已经讨论了一个领域模型会话。大礼帽男作为该方面的专家而Dante作为工程师讨论了领域空间并找到了共同点。这样做是为了学习模型，模型是处理领域所需组件的抽象。当Dante和大礼帽男在讨论酒店，他们正是在讨论相关领域。该领域是软件运行的关注点，我将把酒店（Tavern）称为核心/根领域。

大礼帽男还指出，它不叫饮酒徒，而叫顾客。这说明了在SMO和开发人员之间找到一种通用语言是多么重要。如果不是项目中的每个人都有通用语言，那将会非常令人困惑。我们还得到了一些子领域，这是大礼帽男提到的酒店应用所需要的东西。子领域是一个单独的领域，用于解决根领域内的相关东西。

使用Go编写一个DDD应用-Entities（实体）和Value Object（值对象）



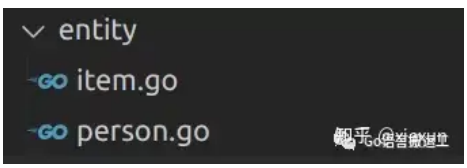
我们已经了解了酒店应用的相关东西，是时候编写酒店系统代码了。通过创建go module来配制本项目。

```
mkdir ddd-go
go mod init github.com/percybolmer/ddd-go
```

我们将创建一个domain目录，存放所有的子领域，但在实现领域之前，我们需要在根目录下创建另一个目录。出于说明的目的，我们将其命名为entity，因为它将保存DDD方法中所谓的实体。一个实体是一个结构体包含标志符，其状态可能会变，改变状态的意思是实体的值可以改变。

首先我们将创建两个实体，Person和Item。我喜欢将实体保存在一个单独的包中，以便它们可以被所有其他领域使用。

为了保持代码整洁，我喜欢小文件，并使文件夹结构易于浏览。因此，我建议创建两个文件，每个文件对应一个实体，并以实体命名。现在，仅仅包含结构体定义，稍后会添加一些其他逻辑。



为领域创建第一个实体

```
//entities包保存所有子领域共享的所有实体
package entity

import (
    "github.com/google/uuid"
)

// Person 在所有领域中代表人
type Person struct {
```

```
    //Name就是人的名字
    Name string `json:"name" bson:"name"`
    // 人的年龄
    Age int `json:"age" name:"age"`
}
```

```
package entity

import "github.com/google/uuid"

// Item表示所有子领域的Item
type Item struct {
    ID          uuid.UUID `json:"id" bson:"id"`
    Name        string    `json:"name" bson:"name"`
    Description string    `json:"description" bson:"description"`
}
```

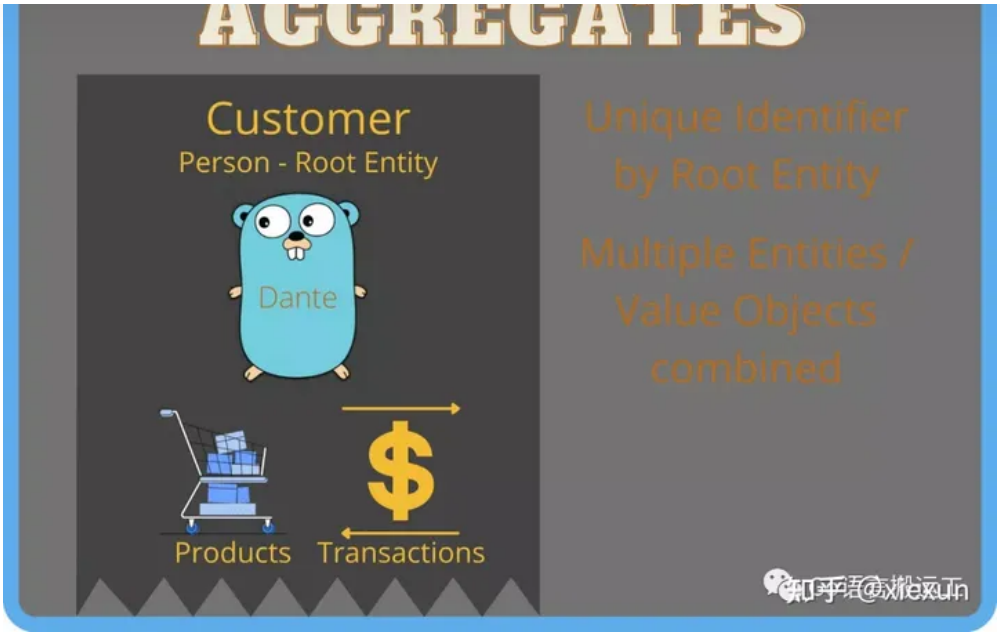
ok，现在我们已经定义了一些实体并了解了什么是实体。一个结构体具有唯一标识符来引用，状态可变。

有些结构体是不可变的，不需要唯一标识符，这些结构体被称为值对象。所以结构体在创建后没有标识符和持久化值。值对象通常位于领域内，用于描述该领域中的某些方面。我们现在将创建一个值对象，它是Transaction，一旦事务被执行，它就不能改变状态。

> 在真实的应用程序中，通过ID跟踪事务是一个好主意，这里只是为了演示

```
package valueobject

import (
    "time"
)

// Transaction表示双方用于支付
type Transaction struct {
    Amount    int       `json:"amount" bson:"amount"`
    From      uuid.UUID `json:"from" bson:"from"`
    To        uuid.UUID `json:"to" bson:"to"`
    CreatedAt time.Time `json:"createdAt" bson:"createdAt"`
}
```

聚合（Aggregates）—组合实体（Entities）和值对象（Value Objects）



现在我们来看看DDD的下一个组件，聚合。聚合是一组实体和值对象的组合。因此，在本例中，我们可以首先创建一个新的聚合，即Customer。

> DDD聚合是领域概念(例如订单、诊所访问、播放列表)——Martin Fowler

聚合(aggregate)的原因是业务逻辑将应用于Customer聚合，而不是每个持有该逻辑的实体。聚合不允许直接访问底层实体。在现实生活中，也经常需要多个实体来正确表示数据，例如Customer。它是一个Person，但是他/她可以持有Products并执行事务。

DDD聚合中的一个重要规则是，它们应该只有一个实体作为根实体。这意味着根实体的引用也用于引用聚合。对于我们的customer聚合，这意味着Person ID是惟一标识符。

让我们创建一个aggregate文件夹，然后在里面创建一个名为customer.go的文件。

```
mkdir aggregate
cd aggregate
touch customer.go
```

在该文件中，我们将添加一个名为Customer的新结构，它将包含表示Customer所需的所有实体。注意，所有字段都以大写字母开头，这在Go中使它们可以从包外部访问。这与我们所说的聚合不允许访问底层实体的说法相违背，但是我们需要它来使聚合可序列化。另一种方法是添加自定义序列化，但我发现有时跳过一些规则是有意义的。

```
// Package aggregates holds aggregates that combines many entities into a full
package aggregate

import (
    "github.com/percybolmer/ddd-go/entity"
    "github.com/percybolmer/ddd-go/valueobject"
)

// Customer 聚合包含了代表一个客户所需的所有实体
type Customer struct {
    // Person是客户的根实体
    // person.ID是聚合的主标识符
    Person *entity.Person `bson:"person"`
    //一个客户可以持有许多产品
    Products []*entity.Item `bson:"products"`
```

```
}
```

我将所有实体设置为指针，这是因为实体可以改变状态，我想让它反映在运行时所有访问它的实例中。值对象被保存为非指针，因为它们不能改变状态。

工厂函数-封装复杂的逻辑

到目前为止，我们只定义了不同的实体、值对象和聚合。现在开始实现一些实际业务逻辑，我们从工厂函数开始。工厂模式是一种设计模式，用于在创建所需实例的函数中封装复杂逻辑，调用者不知道任何实现细节。

工厂模式是一种非常常见的模式，您甚至可以在DDD应用程序之外使用它，而且您可能已经使用过很多次了。官方Go Elasticsearch客户端就是一个很好的例子。您将一个配置传入到NewClient函数中，该函数是一个工厂函数，返回客户端连接到弹性集群，可以插入/删除文档。对于其他开发人员来说很容易使用，在NewClient中做了很多事情：

```
func NewClient(cfg Config) (*Client, error) {
    var addrs []string

    if len(cfg.Addresses) == 0 && cfg.CloudID == "" {
        addrs = addrsFromEnvironment()
    } else {
        if len(cfg.Addresses) > 0 && cfg.CloudID != "" {
            return nil, errors.New("cannot create client: both Addresses and C
        }

        if cfg.CloudID != "" {
            cloudAddr, err := addrFromCloudID(cfg.CloudID)
            if err != nil {
                return nil, fmt.Errorf("cannot create client: cannot parse Clou
            }
            addrs = append(addrs, cloudAddr)
        }

        if len(cfg.Addresses) > 0 {
            addrs = append(addrs, cfg.Addresses...)
```

```
urls, err := addrsToURLs(addrs)
if err != nil {
    return nil, fmt.Errorf("cannot create client: %s", err)
}

if len(urls) == 0 {
    u, _ := url.Parse(defaultURL) // errcheck exclude
    urls = append(urls, u)
}

// TODO(karmi): Refactor
if urls[0].User != nil {
    cfg.Username = urls[0].User.Username()
    pw, _ := urls[0].User.Password()
    cfg.Password = pw
}

tp, err := estransport.New(estransport.Config{
    URLs:         urls,
    Username:     cfg.Username,
    Password:     cfg.Password,
    APIKey:       cfg.APIKey,
    ServiceToken: cfg.ServiceToken,

    Header: cfg.Header,
    CACert: cfg.CACert,

    RetryOnStatus:        cfg.RetryOnStatus,
    DisableRetry:         cfg.DisableRetry,
    EnableRetryOnTimeout: cfg.EnableRetryOnTimeout,
    MaxRetries:           cfg.MaxRetries,
    RetryBackoff:         cfg.RetryBackoff,

    CompressRequestBody: cfg.CompressRequestBody,

    EnableMetrics:     cfg.EnableMetrics,
    EnableDebugLogger: cfg.EnableDebugLogger,

    DisableMetaHeader: cfg.DisableMetaHeader,

    DiscoverNodesInterval: cfg.DiscoverNodesInterval,

    Transport:          cfg.Transport,
    Logger:             cfg.Logger,
    Selector:           cfg.Selector,
    ConnectionPoolFunc: cfg.ConnectionPoolFunc,
})
if err != nil {
    return nil, fmt.Errorf("error creating transport: %s", err)
}

client := &Client{Transport: tp}
client.API = esapi.New(client)

if cfg.DiscoverNodesOnStart {
    go client.DiscoverNodes()
}
```

DDD建议使用工厂来创建复杂的聚合、仓库和服务。我们将实现一个工厂函数，该函数将创建一个新的Customer实例。将创建一个名为NewCustomer的函数，它接受一个name参数，函数内部发生的事情不需要创建新customer的领域所知。

NewCustomer将验证输入是否包含创建Customer所需的所有参数：

> 在实际的应用程序中，我可能会建议在领域/客户中包含聚合的Customer和工厂。

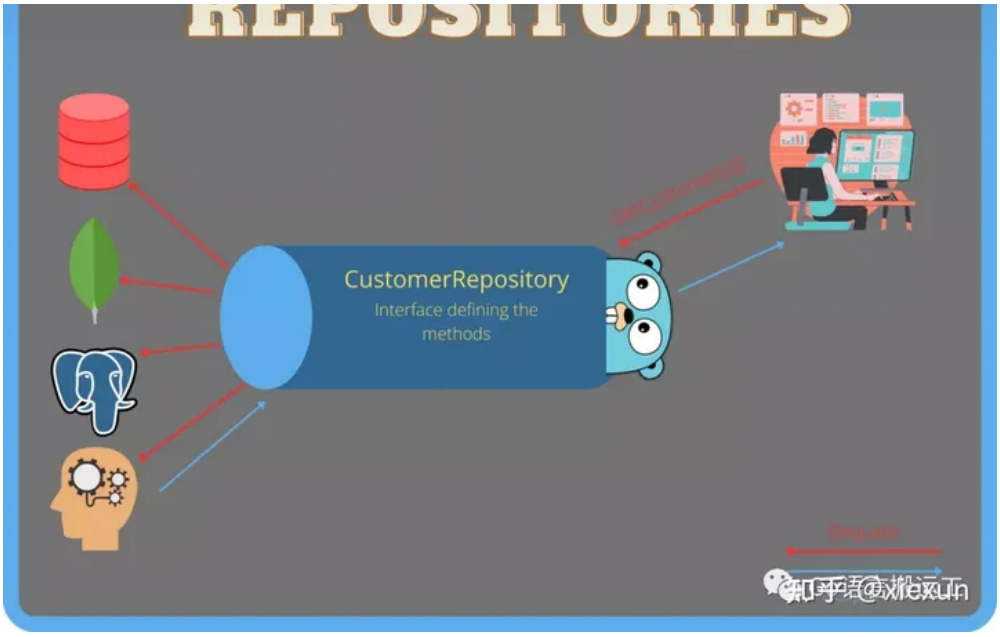
```
package aggregate

import (
    "errors"

    "github.com/google/uuid"
    "github.com/percybolmer/ddd-go/entity"
    "github.com/percybolmer/ddd-go/valueobject"
)

var (
    // 当person在newcustom工厂中无效时返回ErrInvalidPerson
    ErrInvalidPerson = errors.New("a customer has to have an valid person")
)

type Customer struct {
    // Person是客户的根实体
    // person.ID是aggregate的主标志符
    Person *entity.Person `bson:"person"`
    // 一个客户可以持有许多产品
    Products []*entity.Item `bson:"products"`
    // 一个客户可以执行许多事务
    Transactions []valueobject.Transaction `bson:"transactions"`
}

// NewCustomer是创建新的Customer聚合的工厂
// 它将验证名称是否为空
func NewCustomer(name string) (Customer, error) {
    // 验证Name不是空的
    if name == "" {
        return Customer{}, ErrInvalidPerson
    }

    // 创建一个新person并生成ID
    person := &entity.Person{
        Name: name,
        ID:   uuid.New(),
    }
    // 创建一个customer对象并初始化所有的值以避免空指针异常
    return Customer{
        Person:       person,
        Products:     make([]*entity.Item, 0),
        Transactions: make([]valueobject.Transaction, 0),
    }, nil
}
```

customer工厂函数现在帮助验证输入、创建新ID并确保正确初始化所有值。
现在我们已经有了一些业务逻辑，可以开始添加单元测试了。我将在aggregate包中创建一个customer_test.go，在其中测试与Customer相关的逻辑。

```go
import (
    "testing"

    "github.com/percybolmer/ddd-go/aggregate"
)

func TestCustomer_NewCustomer(t *testing.T) {
    // 构建我们需要的测试用例数据结构
    type testCase struct {
        test        string
        name        string
        expectedErr error
    }
    //创建新的测试用例
    testCases := []testCase{
        {
            test:        "Empty Name validation",
            name:        "",
            expectedErr: aggregate.ErrInvalidPerson,
        }, {
            test:        "Valid Name",
            name:        "Percy Bolmer",
            expectedErr: nil,
        },
    }

    for _, tc := range testCases {
        // Run Tests
        t.Run(tc.test, func(t *testing.T) {
            //创建新的customer
            _, err := aggregate.NewCustomer(tc.name)
            //检查错误是否与预期的错误匹配
            if err != tc.expectedErr {
                t.Errorf("Expected error %v, got %v", tc.expectedErr, err)
            }

        })
    }

}
```

我们不会在创造新customer方面深入，现在开始寻找我所知道的最佳设计模式的时候了。

仓库-仓库模式



DDD描述了应该使用仓库来存储和管理聚合。这是其中一种模式，一旦我学会了，我就知道我永远不会停止使用它。这种模式依赖于通过接口隐藏存储/数据库解决方案的实现。这允许我们定义一组必须使用的方法，如果它们被实现了，就可以被用作一个仓库。

这种设计模式的优点是，它允许我们在不破坏任何东西的情况下切换解决方案。我们可以在开发阶段使用内存存储，然后在生产阶段将其切换到MongoDB存储。它不仅有助于在不破坏任何利用仓库的东西的情况下更改所使用的底层技术，而且在测试中也非常有用。您可以简单地为单元测试等实现一个新的仓库。

我们将首先创建一个名为repository的文件。进入domain/customer包。在该文件中，我们将定义仓库所需的函数。我们需要Get、Add和Update函数处理customers。我们不会删除任何客户，一旦有客户在酒店，就永远是客户。我们还将在客户包中实现一些通用错误，不同的仓库实现可以使用这些错误。

```
// Customer包保存了客户领域的所有域逻辑

import (
    "github.com/google/uuid"
    "github.com/percybolmer/ddd-go/aggregate"
)
var (
    // 当没有找到客户时返回ErrCustomerNotFound。
    ErrCustomerNotFound = errors.New("the customer was not found in the reposi
    // ErrFailedToAddCustomer在无法将客户添加到存储库时返回。
    ErrFailedToAddCustomer = errors.New("failed to add the customer to the repo
    // 当无法在存储库中更新客户时，将返回ErrUpdateCustomer。
    ErrUpdateCustomer = errors.New("failed to update the customer in the repos
)
//  CustomerRepository是一个接口，它定义了围绕客户仓库的规则
// 必须实现的函数
type CustomerRepository interface {
    Get(uuid.UUID) (aggregate.Customer, error)
    Add(aggregate.Customer) error
    Update(aggregate.Customer) error
}
```

接下来，我们需要实现接口的实际业务逻辑，我们将从内存存储方式开始。在本文的最后，我们将了解如何在不破坏其他任何东西的情况下将其更改为MongoDB存储方案。

> 另一种方式是在customer包中创建memory.go， 但我发现在更大的系统中，它会很快变得混乱

```
mkdir memory
touch memory/memory.go
```

让我们首先在memory.go文件中设置正确的结构，我们希望创建一个具有实现CustomerRepository接口的结构，并且不要忘记创建新仓库的工厂函数。

```
// memory包是客户仓库的内存中实现
package memory

import (
    "sync"

    "github.com/google/uuid"
    "github.com/percybolmer/ddd-go/aggregate"
)

// MemoryRepository实现了CustomerRepository接口
type MemoryRepository struct {
    customers map[uuid.UUID]aggregate.Customer
    sync.Mutex
}

// New是一个工厂函数，用于生成新的客户仓库
func New() *MemoryRepository {
    return &MemoryRepository{
        customers: make(map[uuid.UUID]aggregate.Customer),
    }
}

// Get根据ID查找Customer
func (mr *MemoryRepository) Get(uuid.UUID) (aggregate.Customer, error) {
    return aggregate.Customer{}, nil
}

// Add将向存储库添加一个新Customer
func (mr *MemoryRepository) Add(aggregate.Customer) error {
    return nil
}

// Update将用新的客户信息替换现有的Customer信息
func (mr *MemoryRepository) Update(aggregate.Customer) error {
    return nil
}
```

我们需要添加一种从Customer聚合中检索信息的方法，例如来自根实体的ID。所以我们应该用一个获取ID的函数和一个更改名称的函数来更新聚合。

```
// GetID返回客户的根实体ID
func (c *Customer) GetID() uuid.UUID {
    return c.Person.ID
}
// SetName更改客户的名称
func (c *Customer) SetName(name string) {
```

让我们向内存仓库添加一些非常基本的功能，以便它能按预期工作。

```go
// memory包是客户仓库的内存实现
package memory

import (
    "fmt"
    "sync"

    "github.com/google/uuid"
    "github.com/percybolmer/ddd-go/aggregate"
    "github.com/percybolmer/ddd-go/domain/customer"
)

// MemoryRepository实现了CustomerRepository接口
type MemoryRepository struct {
    customers map[uuid.UUID]aggregate.Customer
    sync.Mutex
}

// New是一个工厂函数，用于生成新的客户存储库
func New() *MemoryRepository {
    return &MemoryRepository{
        customers: make(map[uuid.UUID]aggregate.Customer),
    }
}

// Get根据ID查找Customer
func (mr *MemoryRepository) Get(id uuid.UUID) (aggregate.Customer, error) {
    if customer, ok := mr.customers[id]; ok {
        return customer, nil
    }

    return aggregate.Customer{}, customer.ErrCustomerNotFound
}

// Add将向存储库添加一个新Customer
func (mr *MemoryRepository) Add(c aggregate.Customer) error {
    if mr.customers == nil {
    // 安全检查如果Customer没创建，在使用工厂是不应该发生，但你永远不知道
        mr.Lock()
        mr.customers = make(map[uuid.UUID]aggregate.Customer)
        mr.Unlock()
    }
    // 确保Customer不在仓库中
    if _, ok := mr.customers[c.GetID()]; ok {
        return fmt.Errorf("customer already exists: %w", customer.ErrFailedToA
    }
    mr.Lock()
    mr.customers[c.GetID()] = c
    mr.Unlock()
    return nil
}

// Update 将用新的Customer信息替换现有Customer户信息
func (mr *MemoryRepository) Update(c aggregate.Customer) error {
    // 确保Customer在存储库中
    if _, ok := mr.customers[c.GetID()]; !ok {
        return fmt.Errorf("customer does not exist: %w", customer.ErrUpdateCus
```

```
    mr.customers[c.GetID()] = c
    mr.Unlock()
    return nil
}
```

和前面一样，我们应该为代码添加单元测试。我想从测试的角度指出仓库模式有多好。在单元测试中，使用仅为测试创建的仓库替换部分逻辑非常容易，这使得发现测试中的已知错误变得更加容易。

```
package memory

import (
    "testing"

    "github.com/google/uuid"
    "github.com/percybolmer/ddd-go/aggregate"
    "github.com/percybolmer/ddd-go/domain/customer"
)

func TestMemory_GetCustomer(t *testing.T) {
    type testCase struct {
        name        string
        id          uuid.UUID
        expectedErr error
    }

    //创建要添加到存储库中的模拟Customer
    cust, err := aggregate.NewCustomer("Percy")
    if err != nil {
        t.Fatal(err)
    }
    id := cust.GetID()
    // 创建要使用的仓库，并添加一些测试数据进行测试
    // 跳过工厂
    repo := MemoryRepository{
        customers: map[uuid.UUID]aggregate.Customer{
            id: cust,
        },
    }

    testCases := []testCase{
        {
            name:        "No Customer By ID",
            id:          uuid.MustParse("f47ac10b-58cc-0372-8567-0e02b2c3d479"),
            expectedErr: customer.ErrCustomerNotFound,
        }, {
            name:        "Customer By ID",
            id:          id,
            expectedErr: nil,
        },
    }

    for _, tc := range testCases {
        t.Run(tc.name, func(t *testing.T) {

            _, err := repo.Get(tc.id)
            if err != tc.expectedErr {
                t.Errorf("Expected error %v, got %v", tc.expectedErr, err)
            }
        })
```

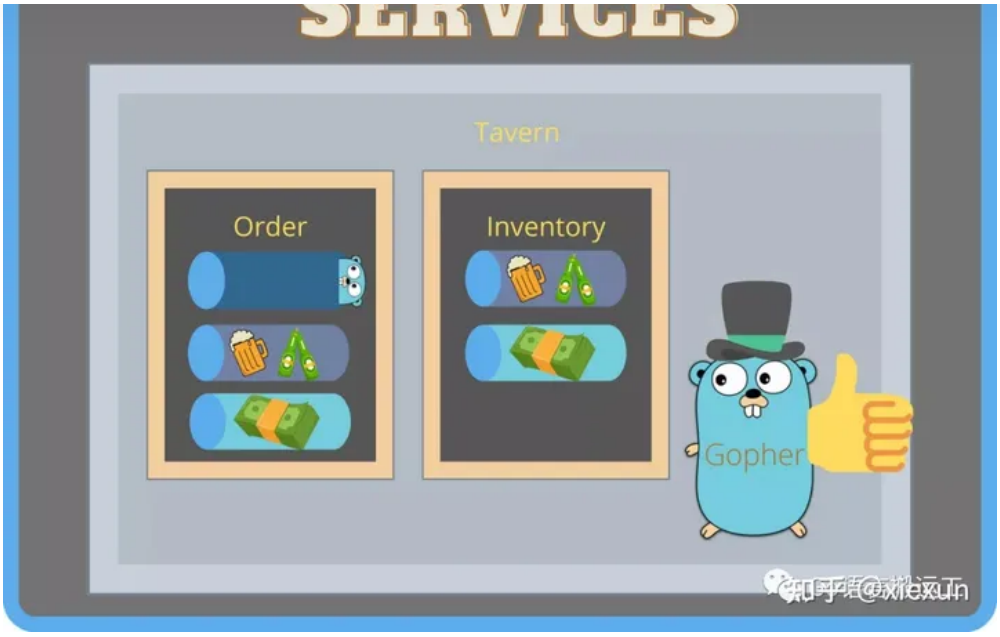
```
func TestMemory_AddCustomer(t *testing.T) {
    type testCase struct {
        name        string
        cust        string
        expectedErr error
    }

    testCases := []testCase{
        {
            name:        "Add Customer",
            cust:        "Percy",
            expectedErr: nil,
        },
    }

    for _, tc := range testCases {
        t.Run(tc.name, func(t *testing.T) {
            repo := MemoryRepository{
                customers: map[uuid.UUID]aggregate.Customer{},
            }

            cust, err := aggregate.NewCustomer(tc.cust)
            if err != nil {
                t.Fatal(err)
            }

            err = repo.Add(cust)
            if err != tc.expectedErr {
                t.Errorf("Expected error %v, got %v", tc.expectedErr, err)
            }

            found, err := repo.Get(cust.GetID())
            if err != nil {
                t.Fatal(err)
            }
            if found.GetID() != cust.GetID() {
                t.Errorf("Expected %v, got %v", cust.GetID(), found.GetID())
            }
        })
    }
}
```

很好，我们有了第一个仓库。记住要保持仓库与它们的领域相关。在这种情况下，仓库只处理Customer聚合，它应该只这样做。永远不要让仓库与任何其他聚合耦合，我们想要松耦合。

那么我们如何处理酒店的逻辑流呢，我们不能简单地依赖客户仓库？我们将在某一点上开始耦合不同的仓库，并构建一个表示酒店逻辑的流。

进入Services，这是我们需要学习的最后一部分。

Services——连接业务逻辑



我们有这些实体，一个聚合，和一个仓库，但它还不像一个应用程序，不是吗？这就是为什么我们需要下一个组件Service。

Service将把所有松散耦合的仓库绑定到满足特定领域需求的业务逻辑中。在酒店应用中，我们可能有一个Order服务，负责将仓库链接在一起以执行订单。因此，服务将拥有对CustomerRepository和ProductRepository的访问权。

Service通常包含执行某个业务逻辑流(如Order、Api或Billing)所需的所有仓库。你甚至可以在一个服务中包含另一个服务。

我们将实现Order服务，它随后可以成为酒店（Tavern）服务的一部分。让我们创建一个名为services的新文件夹，该文件夹将保存我们实现的服务。我们首先创建一个名为order.go的文件将持有OrderService，我们将使用它来处理酒店中的新订单。我们仍然缺少一些领域，因此我们将只从CustomerRepository开始，但很快会添加更多领域。

我想从创建一个新的Service的Factory开始，并演示一个非常简单的技巧，这是我从Jon Calhoun的web开发书中学到的。我们将为一个函数创建一个别名，该函数接受一个Service指针并修改它，然后允许使用这些别名的可变参数。通过这种方式更改Service的行为或替换仓库非常容易。

```
// service包，包含将仓库连接到业务流的所有服务
package services

import (
    "github.com/percybolmer/ddd-go/domain/customer"
)

// OrderConfiguration是一个函数的别名，该函数将接受一个指向OrderService的指针并对其进行
type OrderConfiguration func(os *OrderService) error

//OrderService是OrderService的一个实现
type OrderService struct {
    customers customer.CustomerRepository
}

// NewOrderService接受可变数量的OrderConfiguration函数，并返回一个新的OrderService
// 将按照传入的顺序调用每个OrderConfiguration
func NewOrderService(cfgs ...OrderConfiguration) (*OrderService, error) {
    // 创建orderservice
    os := &OrderService{}
```

```
            // 将service传递到configuration函数
            err := cfg(os)
            if err != nil {
                return nil, err
            }
        }
        return os, nil
    }
```

看看我们如何在工厂方法中接受可变数量的OrderConfiguration？这是一种允许动态工厂，并允许开发人员配置代码结构的非常整洁的方法，前提是已经实现了相关函数。这个技巧对于单元测试非常有用，因为您可以用所需的仓库替换服务中的某些部分。

对于较小的服务，这种方法似乎有点复杂了。我想指出的是，在示例中，我们只使用configurations来修改仓库，但这也可以用于内部设置和选项。对于较小的服务，也可以创建一个简单的工厂函数，例如接受CustomerRepository。

让我们创建一个应用CustomerRepository的OrderConfiguration，这样我们就可以开始创建Order的业务逻辑。

```
// WithCustomerRepository将给定的客户仓库应用到OrderService
func WithCustomerRepository(cr customer.CustomerRepository) OrderConfiguration
    // 返回一个与OrderConfiguration别名匹配的函数，
    // 您需要返回这个，以便父函数可以接受所有需要的参数
    return func(os *OrderService) error {
        os.customers = cr
        return nil
    }
}

// WithMemoryCustomerRepository将内存客户仓库应用到OrderService
func WithMemoryCustomerRepository() OrderConfiguration {
    // 创建内存仓库，如果我们需要参数，如连接字符串，它们可以在这里输入
    cr := memory.New()
    return WithCustomerRepository(cr)
}
```

现在，要使用这个，您可以在创建服务时简单地链接所有configurations，从而使我们能够轻松地更换组件。

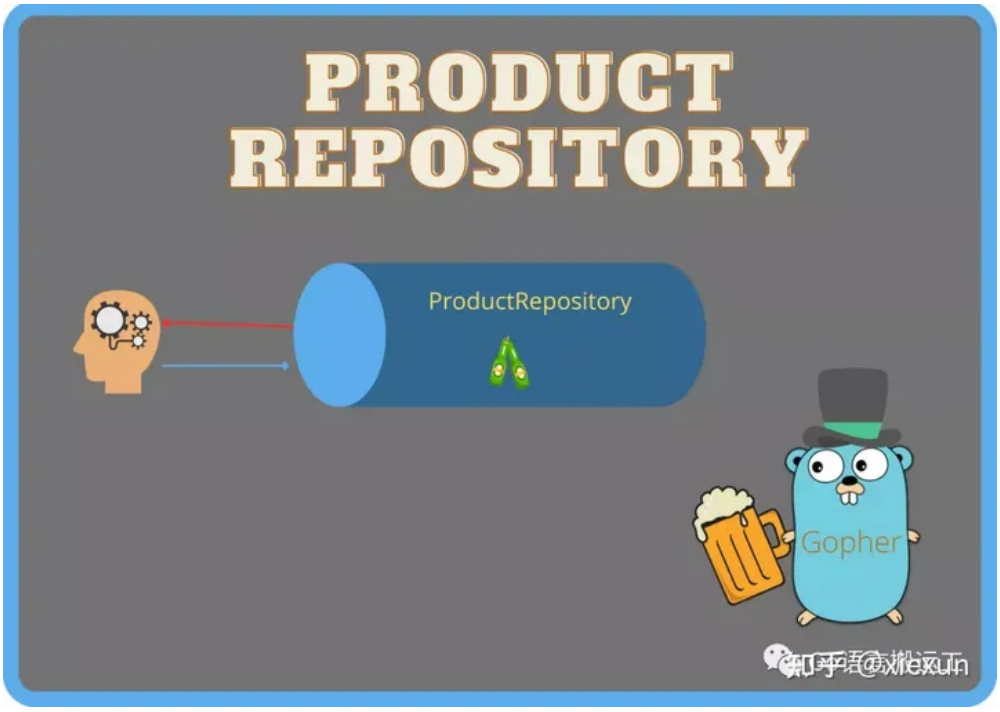
```
// 在开发中使用的内存示例
NewOrderService(WithMemoryCustomerRepository())
// 我们将来可以像这样切换到MongoDB
NewOrderService(WithMongoCustomerRepository())
```

让我们开始为Order服务添加功能，这样顾客就可以在酒店里购买东西。

```
// CreateOrder将所有仓库链接在一起，为客户创建订单
func (o *OrderService) CreateOrder(customerID uuid.UUID, productIDs []uuid.UUI
    // 获取customer
    c, err := o.customers.Get(customerID)
    if err != nil {
        return err
    }

    // 获取每个产品，我们需要一个Product Repository
```

哎呀，我们的酒店没有任何产品供应。你肯定知道怎么解决吧？让我们实现更多的仓库，并通过使用OrderConfiguration将它们应用到服务中。

ProductRepository -酒店系统的最后一部分



可以参考customer Repository实现完成。

发布于 2021-09-05 00:00

领域驱动设计（DDD） 领域驱动设计（书籍） 软件