12 | 代码实现（下）：怎样更加“面向对象”？

2022-12-31 钟敬 来自北京

《手把手教你落地DDD》

讲述：钟敬

时长 16:00　大小 14.61M

你好，我是钟敬。今天咱们继续研究编码。

上节课我们学习了**领域服务**和**工厂**两个模式，分别用于实现领域逻辑以及创建领域对象。今天我们考虑再增加一些面向对象的元素。

面向对象的三个特征是封装、继承和多态。其中多态我们暂时还不涉及。而上节课我们完成的**添加组织**功能，在封装和继承方面还不到位。

今天我们会继续开发**修改组织**功能，结合这个功能的开发，看看怎样运用封装和继承。

为“修改组织”功能开卡

为了完成**修改组织**的功能，我们再一次把产品经理老王请过来进行“开卡”，看需求的细节是否理解到位。

首先要确认的一个问题是，到底要修改组织的哪些信息。目前的 Org 对象一共有 11 个属性，我们列成一个表，和老王逐一确认。

属性	创建	修改	备注
组织 ID	–	?	系统自动生成，不可更改；后续操作作为 url 路径参数传入
租户	是	?	操作不能跨租户，因此不可更改，但需要作为后续操作的输入参数
上级组织	是	?	–
组织类别	是	?	–
负责人	是	?	–
组织名称	是	?	–
状态	–	?	–
创建时间	–	–	系统自动生成，不可更改
创建人	–	–	系统根据用户信息自动设置
最后修改时间	–	–	系统自动生成，不可更改
最后修改人	–	–	系统根据用户信息自动设置

这个表的含义是，在完成某个功能时，客户端要提供哪些数据。其中“创建”一栏是创建组织时需要哪些参数。“修改”一栏中的问号，是我们觉得在修改时需要客户提供，但需要与业务确认的。

老王轻轻一笑，眼神中的意思是“你们这些搞 IT 的还是不懂业务呀”，然后给出了后面四条意见。

1. 如果修改“上级组织”的话，实际上是在调整组织结构。一般不作为普通的修改功能，建议增加专门的“调整组织结构”功能。

2.“组织类别”不能修改，也就是说，一个开发组不能直接变成开发中心。如果真有这种需求，需要另外创建新的开发中心，把人迁移过去，然后撤销当前的开发组。

3.“状态”不能直接改，应该通过“撤销组织”功能间接修改。

4. 最终只有“负责人”和“名称”可以直接修改，建议这个功能叫做“修改组织基本信息”。

根据老王的建议，我们把表格改成了下面的样子。由于调整组织结构功能比较复杂，所以以后再考虑，这里只增加了“撤销组织”功能。

属性	创建组织	修改组织基本信息	撤销组织	备注
组织 ID	–	–	–	系统自动生成，不可更改；后续操作作为 url 路径参数传入
租户	是	是	是	操作不能跨租户，因此不可更改，但需要作为后续操作的输入参数
上级组织	是	–	–	在“调整组织架构”中更改
组织类别	是	–	–	–
负责人	是	是	–	–
组织名称	是	是	–	–
状态	–	–	是	–
创建时间	–	–	–	系统自动生成，不可更改
创建人	–	–	–	系统根据用户信息自动设置
最后修改时间	–	–	–	系统自动生成，不可更改
最后修改人	–	–	–	系统根据用户信息自动设置

接着，老王又为撤销组织增加了 2 个业务规则。

规则编号	限界上下文	模块	规则描述	举例	影响的主要功能
R023	卷卷通	组织管理	组织下没有员工，才能撤销该组织	组织下面还有小王和小李，这时不能撤销组织，需要先把这两个人迁移到其他部门，才能撤销。	撤销组织
R024	卷卷通	组织管理	只有有效的组织才能被撤销		撤销组织

初步完成程序功能

采用前两节课的方法，我们初步完成了“修改组织基本信息”和“撤销组织”功能。

修改后的 OrgServcie 是这样的。

复制代码

```
package chapter12.unjuanable.application.orgmng;
// imports ...

@Service
public class OrgService {
    private final OrgBuilderFactory orgBuilderFactory;
    private final OrgRepository orgRepository;
    private final OrgHandler orgHandler;  //新增了一个领域服务依赖

    @Autowired
    public OrgService(OrgBuilderFactory orgBuilderFactory
            , OrgHandler orgHandler
            , OrgRepository orgRepository) {
        // 为依赖注入赋值 ...
    }

    @Transactional
    public OrgDto addOrg(OrgDto request, Long userId) {
        // 添加组织的功能已完成，这里省略 ...
    }

    //修改组织基本信息
    @Transactional
    public OrgDto updateOrgBasic(Long id, OrgDto request, Long userId) {
        Org org = orgRepository.findById(request.getTenant(), id)
                .orElseThrow(() -> {
                    throw new BusinessException("要修改的组织(id ="
                                + id + "  )不存在！");
                });

        orgHandler.updateBasic(org, request.getName()
                , request.getLeader(), userId);

        orgRepository.update(org);

        return buildOrgDto(org);
    }

    //取消组织
    @Transactional
    public Long cancelOrg(Long id, Long tenant, Long userId) {
        Org org = orgRepository.findById(tenant, id)
                .orElseThrow(() -> {
                    throw new BusinessException("要取消的组织(id ="
                                + id + "  )不存在！");
```

```
                });

        orgHandler.cancel(org, userId);
        orgRepository.update(org);

        return org.getId();
    }

    private static OrgDto buildOrgDto(Org org) {
        // 将领域对象转换成DTO
    }

}
```


我们写了一个 OrgHandler 来协助完成功能。和 Validator 一样，这也是一个**领域服务，**但命名为 OrgDomainService 显得有点繁琐，我们这里按 XxxHandler 命名，也就是 Xxx 处理器。

OrgHandler 的代码是这样的。

复制代码

```
package chapter12.unjuanable.domain.orgmng.org;
// imports...

@Component
public class OrgHandler {
    private final CommonValidator commonValidator;
    private final OrgNameValidator nameValidator;
    private final OrgLeaderValidator leaderValidator;
    private CancelOrgValidator cancelValidator;

    public OrgHandler(CommonValidator commonValidator
            , OrgNameValidator nameValidator
            , OrgLeaderValidator leaderValidator
            , CancelOrgValidator cancelValidator) {
        // 为依赖注入赋值...
    }

    public void updateBasic(Org org, String newName
                , Long newLeader, Long userId) {
        updateName(org, newName);
        updateLeader(org, newLeader);
        updateAuditInfo(org, userId);
    }

    public void cancel(Org org, Long userId) {
        cancelValidator.cancelledOrgShouldNotHasEmp(org.getTenant()
                        , org.getId());
        cancelValidator.OnlyEffectiveOrgCanBeCancelled(org);
```

```
        org.setStatus(OrgStatus.CANCELLED);
        updateAuditInfo(org, userId);
    }

    private void updateLeader(Org org, Long newLeader) {
        if (newLeader != null && !newLeader.equals(org.getLeader())) {
            leaderValidator.leaderShouldBeEffective(org.getTenant()
                                , newLeader);
            org.setLeader(newLeader);
        }
    }

    private void updateName(Org org, String newName) {
        if (newName != null && !newName.equals(org.getName())) {
            nameValidator.orgNameShouldNotEmpty(newName);
            nameValidator.nameShouldNotDuplicatedInSameSuperior(
                  org.getTenant(), org.getSuperior(), newName);
            org.setName(newName);
        }
    }

    private void updateAuditInfo(Org org, Long userId) {
        // 设置最后修改人和时间
    }
}
```


与 OrgBuilder 类似，这个类中的方法基本上也是先校验，再赋值，只不过这里是更改而不是新建。同时 OrgHandler 没有可变属性，因此可以直接注入到应用服务。

提高应用 API 的封装性

下面我们看看怎么提高程序的封装性。

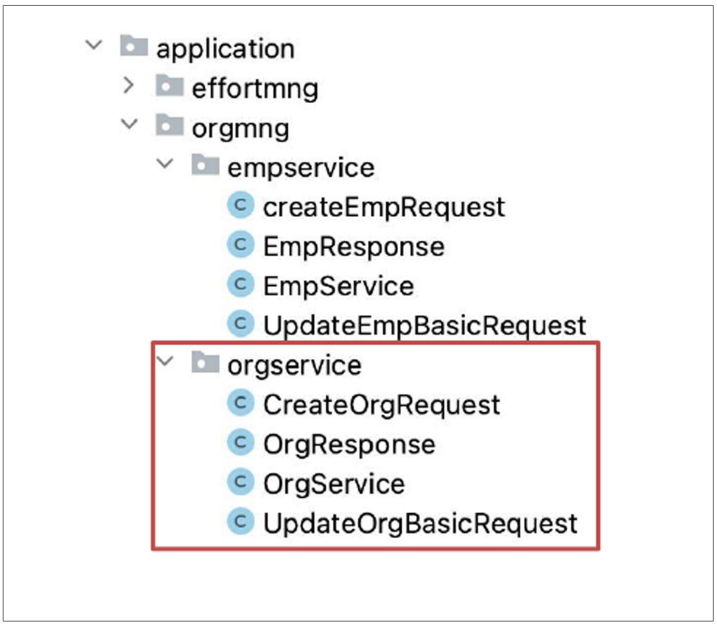
所谓“封装”，指的是将一个模块的实现细节尽量隐藏在内部，只向外界暴露最小的可访问接口，也叫信息隐藏原则，或最小接口原则，目的是减小模块间的耦合，提高程序的可维护性。这里说的模块是广义的，一个函数、一个类、一个包乃至整个应用系统，都可以看作模块，而我们之前领域建模中说的**模块**模式是狭义的，专门指领域模型里的领域对象所组成的包。

先看一下系统对外提供的 API。我们的系统采用 RestfulAPI，以 JSON 作为参数格式，JSON 的结构则是与 OrgDto 一致的。目前的 DTO 中有领域对象中所有 11 个属性，因此，不论增加还是修改，入口参数的 JSON 格式都是下面这样。

```
{
  "createdAt": "2022-10-05T06:49:39.659Z",
  "createdBy": 0,
  "id": 0,
  "lastUpdatedAt": "2022-10-05T06:49:39.659Z",
  "lastUpdatedBy": 0,
  "leader": 0,
  "name": "string",
  "orgType": "string",
  "status": "string",
  "superior": 0,
  "tenant": 0
}
```


我们已经知道，添加和修改操作，实际上都只是用了这些属性的一个子集。现在这样简单粗暴地把所有属性都暴露给客户端，不仅违反了最小接口原则，也容易在理解上发生混淆。

为了解决这个问题，我们要为每个功能编写单独的参数 DTO，只包含必要的参数。DTO 本身的修改比较简单，你可以自己试一下。这里我们重点来看一下修改后应用层的包结构。



CreateOrgRequest 和 UpdateOrgBasicRequest 分别是添加和修改组织的参数 DTO。原来的 OrgDto 改名为 OrgResponse，作为两个功能共同的返回参数类型。另外，这里又出现了一个包结构打横分还是打竖分的问题。

一些伙伴更喜欢打横分，也就是创建一个 dto 包，把 EmpService 和 OrgService 的 DTO 都放进去。而我们这里采用的是**打竖分**，也就是把 service 和这个 service 用到的 DTO 放在同一个包内，提高了模块的内聚性。

修改之后，添加组织的 JSON 参数格式成了下面这样。

```
{
  "leader": 0,
  "name": "string",
  "orgType": "string",
  "superior": 0,
  "tenant": 0
}
```

这就简洁多了，关键是缩小了 API，系统的封装性提高了。修改组织功能的 JSON 的优化也是类似的。

提高领域对象的封装性

改进了系统 API 的封装之后，我们再来考虑领域对象层面。

现在的 Org 对象只是纯粹的 DTO，所有属性都通过 getter 和 setter 暴露在外，没有任何封装性可言。要提高封装性，可以从两个角度考虑。第一是限制 getter 和 setter 的数量；第二是用表示业务含义的接口代替简单的 setter 和 getter。

由于 getter 只是用来查询，不会破坏数据，而不恰当的 setter 则可能破坏数据，导致程序出错，所以相对而言，限制 setter 比限制 getter 更重要一些。为了限制 setter，我们再列一个表，研究一下在创建了组织以后，哪些属性是可以修改的。

属性	是否可修改	备注
组织 ID	–	创建成功后不能修改，但存盘后会被 Repository 更新
租户	–	–
上级组织	是	–
组织类别	–	–
负责人	是	–
组织名称	是	–
状态	是	只在撤销组织功能中进行修改
创建时间	–	–
创建人	–	–
最后修改时间	是	–
最后修改人	是	–

我们可以只为那些可以修改的属性保留 setter，其他的只有 getter，成为只读属性。再为 Org 类增加一个包含只读属性的构造器，以便创建对象。

修改后的 Org 类是这样的。


```
public class Org {
    private Long id;
    private Long tenantId;
    private Long superiorId;
    private String orgTypeCoDe;
```

```
    private Long leaderId;
    private String name;
    private OrgStatus status;
    private LocalDateTime createdAt;
    private Long createdBy;
    private LocalDateTime lastUpdatedAt;
    private Long lastUpdatedBy;

    public Org(Long tenantId, String orgTypeCoDe
          , LocalDateTime createdAt, Long createdBy) {
        // 为属性赋值 ...
    }

    // 所有属性的 getter ...

    // 保留了 superiorId, leaderId, name,
    // lastUpdateAt, lastUPdateBy 和 status 的 setter ...

}
```


其中 OrgId 比较特殊，大部分情况下是只读的，但当 Repository 将新建的 Org 保存到数据库时，由于 id 是数据库自动生成的，需要回填 id 值。这可以通过反射等技巧来绕过去。事实上，很多数据库访问框架就是利用反射，直接为私有属性赋值，以便在不破坏封装的前提下，从数据库中取出对象。

通过减少领域对象的 Setter，我们进一步提高了程序的封装性。

通过“表意接口”提高封装性

做了这些改进以后，我们再看看在 OrgHandler 中完成“撤销组织”功能的代码。

复制代码

```
package chapter12.unjuanable.domain.orgmng.org;
// imports ...

@Component
public class OrgHandler {
    //...

    public void cancel(Org org, Long userId) {
        cancelValidator.OrgToBeCancelledShouldNotHasEmp(
                                    org.getTenantId(), org.getId());
        cancelValidator.OrgToBeCancelledShouldBeEffective(org);
        org.setStatus(OrgStatus.CANCELLED); // 直接为 Status 赋值
        updateAuditInfo(org, userId);
```

```
    }

    // ...
}
```


其中，org.setStatus(OrgStatus.CANCELLED) 直接将组织的状态设置成了“已撤销”。从面向对象的角度来看，更好的做法是 Org 类提供一个 cancel() 方法，像下面这样：

复制代码

```
package chapter12.unjuanable.domain.orgmng.org;
// imports ...

public class Org {
    //...

    //Org 自己管理自己的状态
    public void cancel() {
        this.status = OrgStatus.CANCELLED;
    }

    //...
}
```

这样，Org 类就可以自己管理自己的状态，OrgHandler 就不必了解 Org 内部状态的转换细节，只需告诉 Org 需要撤销就可以了，像下面这样。

复制代码

```
package chapter12.unjuanable.domain.orgmng.org;
// imports ...

@Component
public class OrgHandler {
    //...

    public void cancel(Org org, Long userId) {
        cancelValidator.OrgToBeCancelledShouldNotHasEmp(
                        org.getTenant(), org.getId());
        cancelValidator.OrgToBeCancelledShouldBeEffective(org);
        org.cancel();   // 只需告诉 Org 要进行撤销，但不必了解 Org 内部细节
        updateAuditInfo(org, userId);
    }

    // ...
}
```

类似的，我们看一下“只有有效的组织才能被撤销”这条规则的实现代码。


```
package chapter12.unjuanable.domain.orgmng.org.validator;
// imports...

@Component
public class CancelOrgValidator {
    //...

    // 只有有效的组织才能被撤销
    public void OnlyEffectiveOrgCanBeCancelled(Org org) {
        //直接访问了状态属性
        if (!org.getStatus().equals(OrgStatus.EFFECTIVE)) {
            throw new BusinessException("该组织不是有效状态，不能撤销！");
        }
    }

    //...
}
```

我们看到 CancelOrgValidator 直接访问了 Org 的状态，判断是否为有效组织。这实际上又是重构中的一种坏味道，叫做**特性依恋**（Feature Envy）。Status 这个特性是属于 Org 类的，而 CancelOrgValidator 要通过访问这个特性来实现自己的逻辑，所以 CancelOrgValidator “依恋”了 Org 的特性。这是对象封装性被破坏的征兆。

解决方法是将这段判断逻辑移动到 Org 类内部，重构后的 Org 类是这样的。

复制代码

```
package chapter12.unjuanable.domain.orgmng.org;
// imports ...

public class Org {
    //...

    public boolean isEffective() {
        return status.equals(OrgStatus.EFFECTIVE);
    }

    //...
}
```

这样，CancelOrgValidator 就可以不依赖 Org 的内部状态了。

复制代码

```
package chapter12.unjuanable.domain.orgmng.org.validator;
// imports...

@Component
public class CancelOrgValidator {
    //...

    // 只有有效的组织才能被撤销
    public void OnlyEffectiveOrgCanBeCancelled(Org org) {
        //不再依赖 Org 的内部状态
        if (!org.isEffective()) {
            throw new BusinessException("该组织不是有效状态，不能撤销！");
        }
    }

    //...
}
```


Org.cancel() 和 Org.isEffective() 既提高了对象的封装性，也表达了这个功能的业务含义，因此也是**表意接口**模式的一种应用。

上一节的表意接口体现在“领域服务”，这一节体现在“领域对象”。对“表意接口”的运用，往往可以避免“特性依恋”的坏味道，反之，发现和消除“特性依恋”，也会重构出“表意接口”。

用继承消除重复

下面再来看看 Org 类还有什么可以优化的地方。

我们发现，其实每个实体都包含 4 个审计字段 createdAt、createdBy、lastUpdatedAt 和 lastUpdatedBy。那么我们可以考虑抽出一个包含这四个属性的父类，减少编写各个类的重复工作。抽出的父类是下面的样子。

复制代码

```
package chapter12.unjuanable.common.framework.domain;

import java.time.LocalDateTime;

public abstract class AuditableEntity {
    protected LocalDateTime createdAt;
    protected Long createdBy;
    protected LocalDateTime lastUpdatedAt;
    protected Long lastUpdatedBy;
```

```
    public AuditableEntity(LocalDateTime createdAt, Long createdBy) {
        this.createdAt = createdAt;
        this.createdBy = createdBy;
    }

    // lastUPdatedAt 和 lastUpdatedBy 的 setter 以及所有属性的 getter ...

}
```


这个类放在 common.framework 包里面，因为这种基类属于框架层面。

在面向对象设计中一个常见的陷阱就是滥用继承。要防止这一倾向，要记住一个原则，**不要仅仅为了复用而使用继承**。你还要问自己一个问题，父类和子类的关系，在语义上，**是否有分类关系，或者概念的普遍和特殊关系**。只有符合这种关系的，才能采用继承，否则应该用“组合”来实现复用。

在我们的例子中，我们发现审计字段可以复用，只是可能可以采用继承的一个征兆。然后我们再从语义上进行分析。可以说，每个实体都是可审计的，或者说，每类实体都是一类特殊的可审计实体。**我们发现了这种普遍和特殊的分类关系，所以在这里运用继承是合理的**。

编程风格回顾

在第 10 节课中我们说过，具体的编程风格是多种多样的，每种都各有利弊。因此，不可能也没有必要强求一致。关键是理解背后的原理，作出取舍，然后在自己的开发团队中形成比较统一的风格。

下面总结一下我们这几节课采用的风格的要点，说明背后的原因，以及其他可选的方式。

第一，领域对象不访问数据库。

我们在领域对象中既不会显式，也不会隐式访问数据库，目的是使领域对象和数据库解耦，便于维护和测试。

如果使用 JPA 并结合延迟加载，领域对象就会隐式地访问数据库；如果在领域对象的方法中写 SQL，则会显式地访问数据库。领域对象访问数据库，才能实现更加纯粹的面向对象风格，代价是一般需要对 ORM 框架有深入的理解，或者增加了领域对象和数据库访问机制的耦合，还可能无意间导致性能等问题。你可以根据这两者的利弊进行取舍。

第二，领域服务只能读数据库。

在实现一些业务规则时，需要访问数据库中的数据，因此领域服务需要读数据库。而写库的功能通常可以由应用服务来做，从而减轻领域层的负担。有些伙伴喜欢把写功能也放在领域服务，从而使应用服务非常薄，这样也可以。

第三，应用服务可以读写数据库。

纯粹的（通过调用仓库）读写数据库并不包含领域知识，因此放在应用服务似乎更合适一些。比如更新一个对象前，要把这个对象先从数据库中读出来；创建或修改对象后，要存回数据库，这些本身都没有领域逻辑。

第四，用 ID 表示对象之间的关联。

如果按偏面向对象的风格，对象之间的关联应该用对象导航的方式来实现。比如说，Org 对象的 leader 属性的类型，应该是 Emp（员工），这样就可以在内存中直接从组织对象导航到充当负责人的员工对象了。不过我们的程序中，leader 属性仅保存了员工的 ID，而不是员工对象。这是考虑到在企业应用中节省带宽和内存。关于对象导航风格，我们会在下个迭代中继续探讨。

第五，领域对象有自己的领域服务。

如果是偏面向对象风格，很多领域逻辑可以在领域对象内部完成，因此不一定需要领域服务。但对于偏过程式的风格，由于领域对象不能访问数据库，很多领域逻辑就要外化到领域服务中，因此多数领域对象都会有相应的领域服务。

第六，在以上前提下利用封装和继承。

即使是我们这种偏过程式的风格，如果善于利用封装、合理利用继承，也能有效提高程序质量。一个技巧是识别**特性依恋**的坏味道，并重构到**表意接口**。

最后要说的是，我们这个例子其实比较简单，但体现出的原理和复杂功能是一样的，咱们要以小见大，举一反三。

总结

好，今天的内容先讲到这里，现在来总结一下。

这节课我们主要是利用封装和继承，进一步提高了代码质量。我们可以通过两个层面来提高封装性。

第一个层面是 API 的封装。添加和修改组织两种 API 的参数是不同的，我们只对外暴露出每个 API 必须的参数，从而缩小了接口，提高了封装性。

第二个层面是领域对象的封装。我们分析了哪些属性是不需要修改的，把这些属性变成只读的，从而缩小了对象的接口。另一方面，我们用**表意接口**封装了状态属性，使外界不需要直接关心状态转换的细节，同时消除了**特性依恋**的坏味道。

另外，我们还可以利用继承减少代码的重复。在我们的例子里，是通过抽象出 AuditableEntity 作为所有实体的父类来实现的。

最后，我们对这三节课所使用的编程风格进行了回顾，说明了这么做的理由以及一些其他做法。这里没有唯一正确的方式，重要的是根据实际情况进行权衡。

讲完今天的课，第一个迭代就结束了。通过实现**组织**的**创建**和**修改**功能，我已经把这个迭代需要掌握的知识点都介绍清楚了（为了让你关注重点，我没有把所有的代码实现都写进课程里）。你可以尝试运用这些知识，自己实现其他的功能。

学完这个迭代，应该可以利用 DDD 开发一些不太复杂的需求了，你不妨在自己的项目中试一试。

思考题

1. 对于领域对象，可否利用 Java 的包级私有权限，进一步增强封装？

2. 在现在的 Org 类里，只对 setter 进行了限制，而所有属性都有 getter，能否举例说明什么样的属性不应该有 getter 呢？

好，今天的课程结束了，有什么问题欢迎在评论区留言。从下节课开始，我们将进入第二个迭代，学习 DDD 中几个有些难度的内容，包括**聚合**和**值对象**模式以及**泛化**等技能。相信在咱们的共同努力下，你一定能顺利掌握。

赞 5　提建议

上一篇　11丨代码实现（中）：怎样创建领域对象、实现领域逻辑？

下一篇　13丨迭代二概述：怎样更深刻地理解领域知识？

精选留言 (13)

写留言

bin
2022-12-31 来自广东

handler如果不依赖数据库，这部分业务逻辑好像也可以放到领域对象里面。有没有一个好的边界区分？

> 作者回复: 目前的风格就是以是否依赖数据库来区分，参考课程中对代码风格的总结。当然，也可以制定你自己的代码规则。

3

Jaising
2023-01-20 来自安徽

还在犹豫是否要翻过墙的时候，就先把帽子丢到墙的另一边，剩下的自然会有办法。

以我切身体会仅仅迭代一学完就治好了不少DDD恐惧症，真的感谢钟老师和编辑小新的最小闭环设计，对零基础入门DDD不要太好，平时不愿记笔记的我都能跟着老钟医通过重新演绎更新了五篇——事件风暴、领域建模、数据库设计、分层架构和代码实现，小结与导航在这里：https://juejin.cn/post/7190270005072625723。

辞旧迎新，给自己的最好礼物就是持续学习，学以致用，换取职业生涯不断上台阶的可能，终身成长就有了立身之本，用来抵御各类危机。新的一年从入门DDD开始，会发现软件开发的世界照进了新的光，让我们一起加油！

祝各位兔年快乐！

2

Hesher

2023-02-02 来自北京

老师，如果用JPA，会在实体对象上添加各种注解，框架就要求所有属性必须有setter，这就很难保证实体的封装性了，这种情况有更好的方案吗？

H·H

2023-01-30 来自上海

领域对象是否可以通过依赖注入的方式注入 仓储Repository？

> 作者回复: 你是指把repository注入领域对象，从而使领域对象可以访问数据库吗？理论上可以，就看你要不要选这种编程风格。

共 2 条评论 >

escray

2023-01-29 来自北京

从 org.setStatus(OrgStatus.CANCELLED) 到 org.cancel() 的重构，这个是不是应该在面向对象分析的时候就能够做到？org.isEffective() 也有类似的问题。

1. 领域对象不访问数据库
2. 领域服务只读数据库
3. 应用服务可以写数据库
4. 组合优于继承

这三条编程风格，其实从开始设计的时候就采纳。

对于思考题，

1. 对于 Java 的包级私有权限，我的理解是可以默认为私有，只有在需要的时候才改为 protect 或者是 public，也就是说默认采用相对严格的封装。
2. 对于不适合 getter 的属性，我的理解是所有没有明显业务含义的，都不需要 getter，比如 ID 字段，以及一些采用 ID 来表示对象之间关系的字段，例如 leaderID 就不需要 getter，可以补充一个 getLeader 的公开方法。

课程看到这里，感觉在讲领域驱动的同时，也在说明面向对象的分析和设计，甚至包括代码实现和重构。

Johar
2023-01-25 来自重庆

1. 对于领域对象，可否利用 Java 的包级私有权限，进一步增强封装？
可以
2. 在现在的 Org 类里，只对 setter 进行了限制，而所有属性都有 getter，能否举例说明什么样的属性不应该有 getter 呢？
应用层不需要的字段，或者是需要进行装换字段

Sam Jiang
2023-01-23 来自上海

问题一：个人觉得可以。
问题二：如果getter返回的是容器类对象，需要防止客户程序通过引用修改容器。一种解决办法是返回不可变的容器，就无法增删元素了，但还是可以修改其中某个元素的属性。

请叫我和尚
2023-01-17 来自北京

一些问题：
5. 为什么在 addOrg 加事务？里面就一个 save 操作，是否没必要加@Transactional
6. 很多用@Autowired 注入的属性，有些是 final，有些不是 final，这个有什么讲究吗？

leesper
2023-01-08 来自广东

思考题：
1. 可以，这种领域对象只在领域内使用，出了包完全访问不到，应该是一种纯的领域对象抽象
2. 我觉得应该是某种类似于密码，轻易不可随便读的属性，可以没有getter方法

作者回复: 关于第2题，密码确实是一种。还有些只用于技术实现，业务不可见的，也未必要暴露出来。

共 2 条评论 >

燃

2023-01-04 来自浙江

使用聚合对象的时候，比如电商领域，主单子单1-n关系，应该统一由聚合根暴露get，不能随意其他地方也能访问到子单对象。

共 1 条评论 >

aoe

2023-01-03 来自广东

原来我现在的编程风格是偏过程式的

思考题

1. 可以。这样确实可以增强封装。再疯狂一点，可以使用 Java 模块化技术，让反射都无法打破封装。
2. 没发现不应该有 getter 的属性。

> 作者回复: 关于第2问，可以扩展一下。比如对于状态，如果用 isEffective(), isTerminated()之类的方法封装了，那么未必要用 getStatus()了。

共 2 条评论 >

龙腾

2023-01-02 来自广东

老师请教一下，“第三，应用服务可以读写数据库。”这块没太明白，什么情况下读写数据库不包含领域知识呢？咱们的数据库设计不就是按照领域来划分设计的吗，只要读写都是在领域内进行的吧？

> 作者回复: “调用Repository，从数据库里取出一个领域对象”这句话是不需要和业务人员谈的，因此这句话反应的逻辑不包含领域知识，

虚竹

2022-12-31 来自广东

1 这里的领域对象对应传统写法里的跟数据库表对应的XXEnity对象，传统上基本不会在XXEntity里写逻辑，而是在XXService里封装方法写逻辑，但似乎确实某些逻辑写这里最合适，比如数据状态是否有效，方便复用和修改

2 如果某个校验涉及到5个领域对象（即需要5张表中的数据），是需要把5个领域对象都传入XXValidator，然后调用各个领域对象各自的方法组合起来使用？还是会基于这5个对象再进行一层封装，然后再传入XXValidator？

> 作者回复: 关于第2点，要还要看着5个对象之间有没有联系。最好举出具体的案例，才好回答。

共 3 条评论 >