26 | 泛化的实现（下）：怎样为泛化编写代码？

2023-02-09 钟敬 来自北京

《手把手教你落地DDD》

课程介绍 >

讲述：钟敬

时长 18:24　大小 16.81M

你好，我是钟敬。

上节课，我们学习了泛化的数据库设计，这节课我们接着看看怎样为泛化编写代码。

泛化在程序里，体现为一套有继承关系的对象，而在数据库里体现为若干张表。所以，泛化的编码主要解决的问题就是，怎么把内存中的对象和数据库表里的数据进行相互转换。这个问题解决了，其他部分就和常规的面向对象编程没有什么区别了。

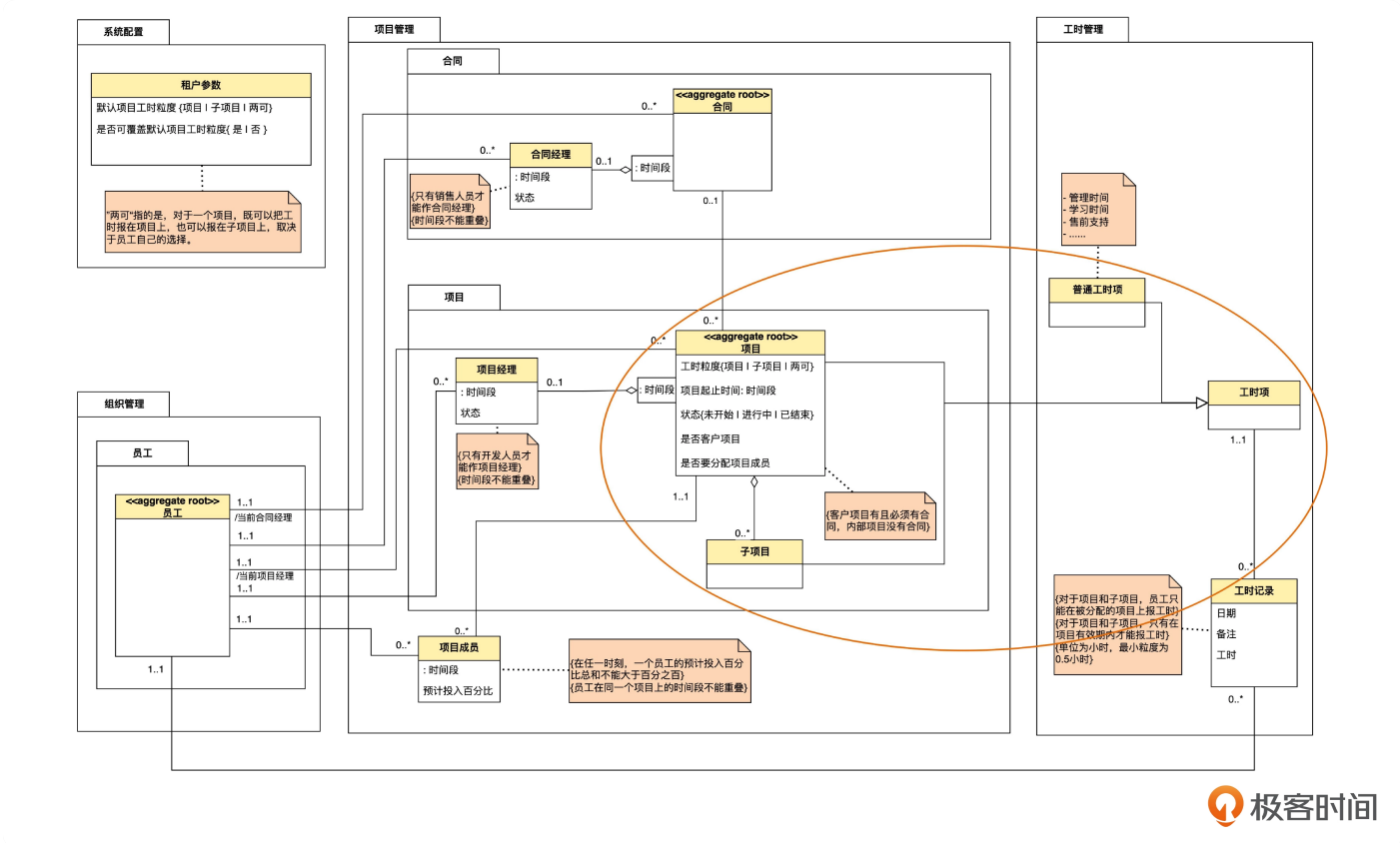
同一个泛化结构，在内存中的对象布局是一样的，但根据不同的数据库设计策略，数据库里的表结构却是不一样的，上节课我们讲过主要有三种。这就造成了泛化关系的持久化问题，比关联关系的持久化要更加复杂一些。

你应该已经想到了，这里说的内存和数据库数据的相互转换问题，是在仓库（repository）里解决的。或者说，仓库屏蔽了不同的表结构的差别，我会结合工时项和客户的例子带你体会这

一点。

为领域模型编码

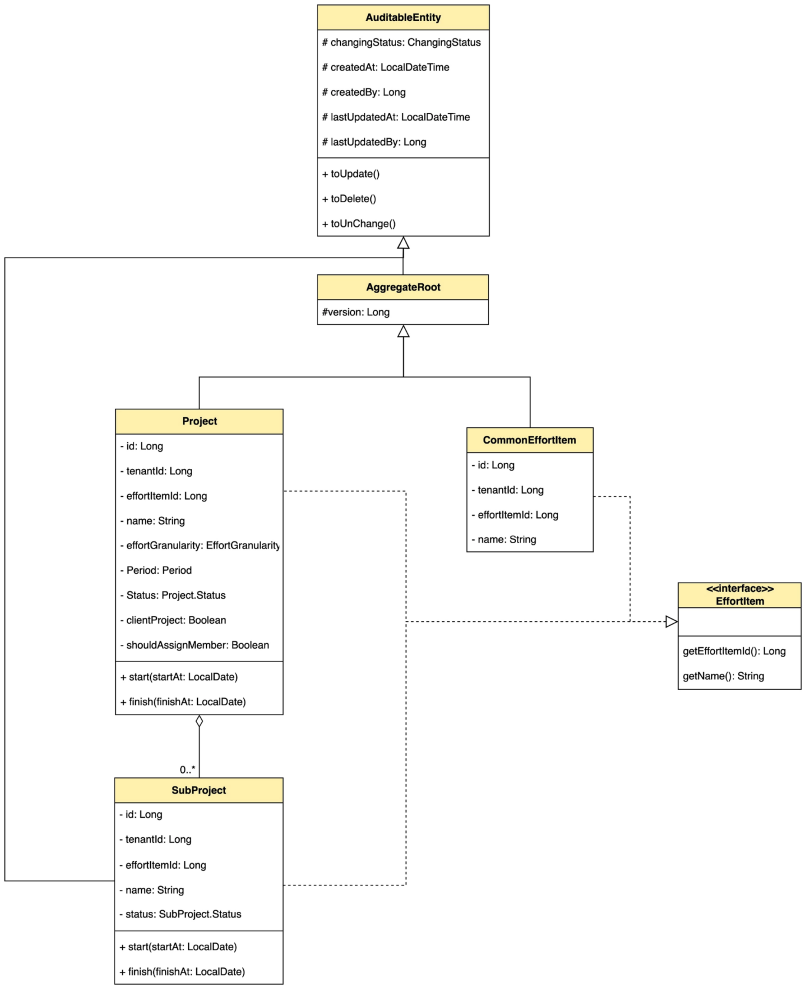
我们首先为工时项以及它的子类编写领域层代码。之前说过，我们要养成边看领域模型，边写代码的习惯，所以先回顾一下领域模型。



本来，传统上的泛化既可以用继承来实现，也可以用不同的属性值来实现。不过根据 DDD 的思路，我们在领域建模的时候，已经有意识地考虑了领域模型和程序实现一致性，所以，对于上图里的泛化，我们直接用继承来实现就可以了。

那么，在程序设计上，应该把工时项建成一个父类吗？

如果用 Java 的话，我们发现无法做到这一点。为了说明这个问题，我们画一下设计模型图。



上面就是设计模型的类图（后面简称为设计图）。

我们首先回味一下设计图和领域模型图的一个区别。领域模型图里只有带实线的空心箭头，而设计图里有实线和虚线两种空心箭头。设计图里的实线箭头代表继承，也就是 Java 里的 extends；虚线箭头代表对接口的实现，也就是 Java 里的 implements。使用继承还是实现，都是代码设计中的考虑，不是业务概念，因此在领域模型图里不需要区分，所以在领域模型图里只需要表示“泛化”的实线箭头。

好，我们继续讨论是否可以用类的继承来实现这个泛化体系的问题。本来，如果没有 SubProject （子项目）的话，可以让 EffortItem（工时项）成为 AggregateRoot（聚合根）的子类，让 Project（项目）和 CommonEffortItem（普通工时项）继承 EffortItem。EffortItem 类里面有 EffortItemId（工时项 ID）属性。

但是，如果让 SubProject 也继承 EffortItem 类的话，SubProject 就成了聚合根，问题在于 SubProject 并不是聚合根。所以 SubProject 只能继承 AuditableEntity（可审计的实体）。由

于 Java 不支持多继承，我们就没有别的选择了，只能把 EffortItem 设计成一个接口，而公共的 EffortItemId 属性只能放在各个子类里了。

AuditableEntity 和 AggregateRoot 是技术实现时候的考虑，没有业务概念，所以在领域模型图里并不存在。

接下来，我们再复习一下有关设计图的其他几个知识点。

设计图里用了英文，目的是接近代码实现；领域模型图里用中文，目的是便于和领域专家交流。要根据词汇表进行中英文的转换。

在设计图里有权限修饰符，加号（+）代表 public，减号（-）代表 private，井号（#）代表 protected，波浪号（~）代表包级私有。而领域模型图里所有属性都是业务可感知的概念，都可以认为是公共的，所以不需要写权限修饰符。

在传统的面向对象方法学里，理论上要根据领域模型图绘制详细的设计图，再进行编码。但在敏捷的实践中，只需要在必要的时候才画部分设计图，多数情况下直接按照领域模型图写代码就行了。我们目前这个设计图只能算是一个示意图。其中省略了 getter 和 setter，也省略了所在的包图。

那么，有了设计，领域层的代码实现就比较简单了。 EffortItem 接口的代码是这样。

复制代码

```
package chapter26.unjuanable.domain.effortmng.effortitem;

public interface EffortItem {
    Long getEffortItemId();
    String getName();
}
```


这应该不用解释了。Project 类的部分代码是后面这样。

复制代码

```
package chapter26.unjuanable.domain.projectmng.project;

// imports ...

```

```java
public class Project extends AggregateRoot
        implements EffortItem {

    private final Long tenantId;              // 租户ID
    private final Long effortItemId;          // 工时项ID
    private String name;                      // 项目名称
    private Period period;                    // 起止时间段
    private Status status;                    // 项目状态
    private Boolean clientProject;            // 是否客户项目
    private Boolean shouldAssignMember;       // 是否要分配人员
    private EffortGranularity effoRtGranulArIty; // 工时粒度

    // 项目经理
    private final Map<Period, ProjectMng> mngs = new HashMap<>();
    // 子项目
    private final Collection<SubProject> subProjects = new ArrayList<>();

    // 构造器 ...

    // 实现 EffortItem 接口里的两个方法
    @Override
    public Long getEffortItemId() {
        return effortItemId;
    }
    @Override
    public String getName() {
        return name;
    }

    // 其他方法...

}
```

我们看到，Project 继承了 AggregateRoot， 同时实现了 EffortItem 接口。EffortItem 的其他几个实现类也是类似的，就不一一展开了。

查询工时项

对于工时项，一个最重要的功能是给定一个员工，查询这个员工可以报工时的**工时项**列表。这个功能的入口在工时项的应用服务 EffortService 里。

我们先设计一下这个功能的返回值类型。对于报工时的需求，前端只需要得到每个工时项的 ID 和名称就可以了。我们先编写 EffortItemDTO 来存放这两个属性。


```java
package chapter26.unjuanable.application.effortmng;
```

```
public class EffortItemDTO {
    private Long effortItemId;
    private String name;
    // 构造器、getter ...
}
```

然后，把 DTO 组织在一起，成为返回值类型 AvailableEffortItems。

复制代码

```
package chapter26.unjuanable.application.effortmng;
// imports ...

public class AvailableEffortItems {
    List<EffortItemDTO> assignments = new ArrayList<>();
    List<EffortItemDTO> subProjects = new ArrayList<>();
    List<EffortItemDTO> commonProjects = new ArrayList<>();
    List<EffortItemDTO> commonEffortItems = new ArrayList<>();

    void addItem(ItemType type, Long effortItemId, String name) {
        switch (type) {
            case ASSIGNED_PROJECT:
                assignments.add(new EffortItemDTO(effortItemId, name));
                break;
            case COMMON_PROJECT:
                commoanProjects.add(new EffortItemDTO(effortItemId, name));
                break;
            case SUB_PROJECT:
                subProjects.add(new EffortItemDTO(effortItemId, name));
                break;
            case COMMON:
                commontEffortItems.add(new EffortItemDTO(effortItemId, name));
                break;
        }
    }

    // getters ...

    public enum ItemType {
        ASSIGNED_PROJECT, COMMON_PROJECT, SUB_PROJECT, COMMON
    }
}

```


为了便于前端显式，返回值把工时项分成了分配的项目（assignments）、通用项目（commonProjects）、子项目（subProjects）、普通工时项（commonEffortItems） 4 个列

表。

编写了返回值类型，就可以编写查询工时项功能了。下面是应用服务的代码。

复制代码

```
package chapter26.unjuanable.application.effortmng;

// imports ...

@Service
public class EffortService {
    // 项目仓库
    private final ProjectRepository projectRepository;
    // 普通工时项仓库
    private final CommonEffortItemRepository commonEffortItemRepository;

    @Autowired
    public EffortService(ProjectRepository projectRepository
            , CommonEffortItemRepository commonEffortItemRepository) {
        // 仓库的依赖注入 ...
    }

    // 查找员工可用的工时项
    public AvailableEffortItems findAvailableEffortItems(Long empId) {
        Collection<Project> assignments
                = projectRepository.findAssignmentsByEmpId(empId);
        Collection<Project> commonProjects
                = projectRepository.findCommonProjects();
        Collection<CommonEffortItem> commonEffortItems
                = commonEffortItemRepository.findAll();

        var result = new AvailableEffortItems();

        assignments.forEach( p ->
                result.addItem(ASSIGNED_PROJECT,p.getEffortItemId(), p.getName(

        commonProjects.forEach( p ->
                result.addItem(COMMON_PROJECT, p.getEffortItemId(), p.getName()

        commonEffortItems.forEach( i ->
                result.addItem(COMMON_ITEM, i.getEffortItemId(), i.getName()));

        return result;
    }
}
```

这个服务的算法是分别从数据库进行三次查询，查出分配给这个员工的项目、不需要分配人员的项目（也就是通用项目）以及普通工时项。然后分别利用它们的**工时项 ID** 和**名称**，构造查询结果。

查询工时项的改进

现在我们想一想，这段代码还有什么改进空间。

这段代码的问题是，从 29 行到 36 行非常相似，似乎可以抽取出来。那么我们就试着抽一下。变成了下面的样子。

复制代码

```
1  package chapter26.unjuanable.application.effortmng;
2
3  // imports ...
4
5  @Service
6  public class EffortService {
7      // 仓库和构造器没有变化 ...
8
9      public AvailableEffortItems findAvailableEffortItems(Long empId) {
10         Collection<Project> assignments
11                 = projectRepository.findAssignmentsByEmpId(empId);
12         Collection<Project> commonProjects
13                 = projectRepository.findCommonProjects();
14         Collection<CommonEffortItem> commonEffortItems
15                 = commonEffortItemRepository.findAll();
16
17         var result = new AvailableEffortItems();
18
19         //使用抽取出的方法
20         appendResult(ASSIGNED_PROJECT, assignments, result);
21         appendResult(COMMON_PROJECT, commonProjects, result);
22
23         //由于类型不匹配，不能使用抽取的方法
24         commonEffortItems.forEach( i ->
25                 result.addItem(COMMON_ITEM, i.getEffortItemId(), i.getName()));
26
27         return result;
28     }
29
30     // 抽取出的公共方法
31     private void appendResult(AvailableEffortItems.ItemType type
32             , Collection<Project> items
33             , AvailableEffortItems result) {
34         items.forEach(p ->
35                 result.addItem(type, p.getEffortItemId(), p.getName()));
```

```
    }
}
```

只有存放项目的 Collection，也就是 assignments 和 commonProjects ，才能使用抽出的公共方法 appendResult()，而 commonEffortItems 则无法使用。这是因为它的类型不是 Collection<Project>，而是 Collection<CommonEffortItem>，不符合 appendResult() 的签名。

那么，怎么让 commonEffortItems 也能使用这个公共的方法呢？

由于 Project 和 CommonEffortItem 都是 EffortItem 的子类，所以我们可以利用泛型的技巧来解决。

复制代码

```
package chapter26.unjuanable.application.effortmng;

// imports ...

@Service
public class EffortService {
    // 仓库和构造器没有变化 ...

    public AvailableEffortItems findAvailableEffortItems(Long empId) {
        Collection<Project> assignments
                = projectRepository.findAssignmentsByEmpId(empId);
        Collection<Project> commonProjects
                = projectRepository.findCommonProjects();
        Collection<CommonEffortItem> commonEffortItems
                = commonEffortItemRepository.findAll();

        var result = new AvailableEffortItems();

        appendResult(ASSIGNED_PROJECT, assignments, result);
        appendResult(COMMON_PROJECT, commonProjects, result);

        //commonEffortItems 也可以使用公共方法了
        appendResult(COMMON_ITEM, commonEffortItems, result);

        return result;
    }

    private void appendResult(AvailableEffortItems.ItemType type
            , Collection<? extends EffortItem> items //使用通配符
            , AvailableEffortItems result) {
        items.forEach(p ->
```

```
                result.addItem(type, p.getEffortItemId(), p.getName()));
    }
}
```

我们在 29 行使用类型通配符，这样就可以像 23 行那样使用公共方法了。如果你不是 Java 背景的话，可以忽略这个技巧。只需要知道由于接口 EffortItem 的存在，我们可以更方便地抽取公共逻辑就可以了。

最后，可以利用“内联”的重构手法，去除多余的局部变量定义，把代码再简化一点。

复制代码

```
package chapter26.unjuanable.application.effortmng;
//imports ...

@Service
public class EffortService {
    // 仓库和构造器没有变化 ...

    public AvailableEffortItems findAvailableEffortItems(Long empId) {

        var result = new AvailableEffortItems();

        // 用"内联"重构，去除多余的局部变量
        appendResult(ASSIGNED_PROJECT
                , projectRepository.findAssignmentsByEmpId(empId)
                , result);

        appendResult(COMMON_PROJECT
                , projectRepository.findCommonProjects()
                , result);

        appendResult(COMMON_ITEM
                , commonEffortItemRepository.findAll()
                , result);

        return result;
    }

    // 其他部分不变 ...
}
```

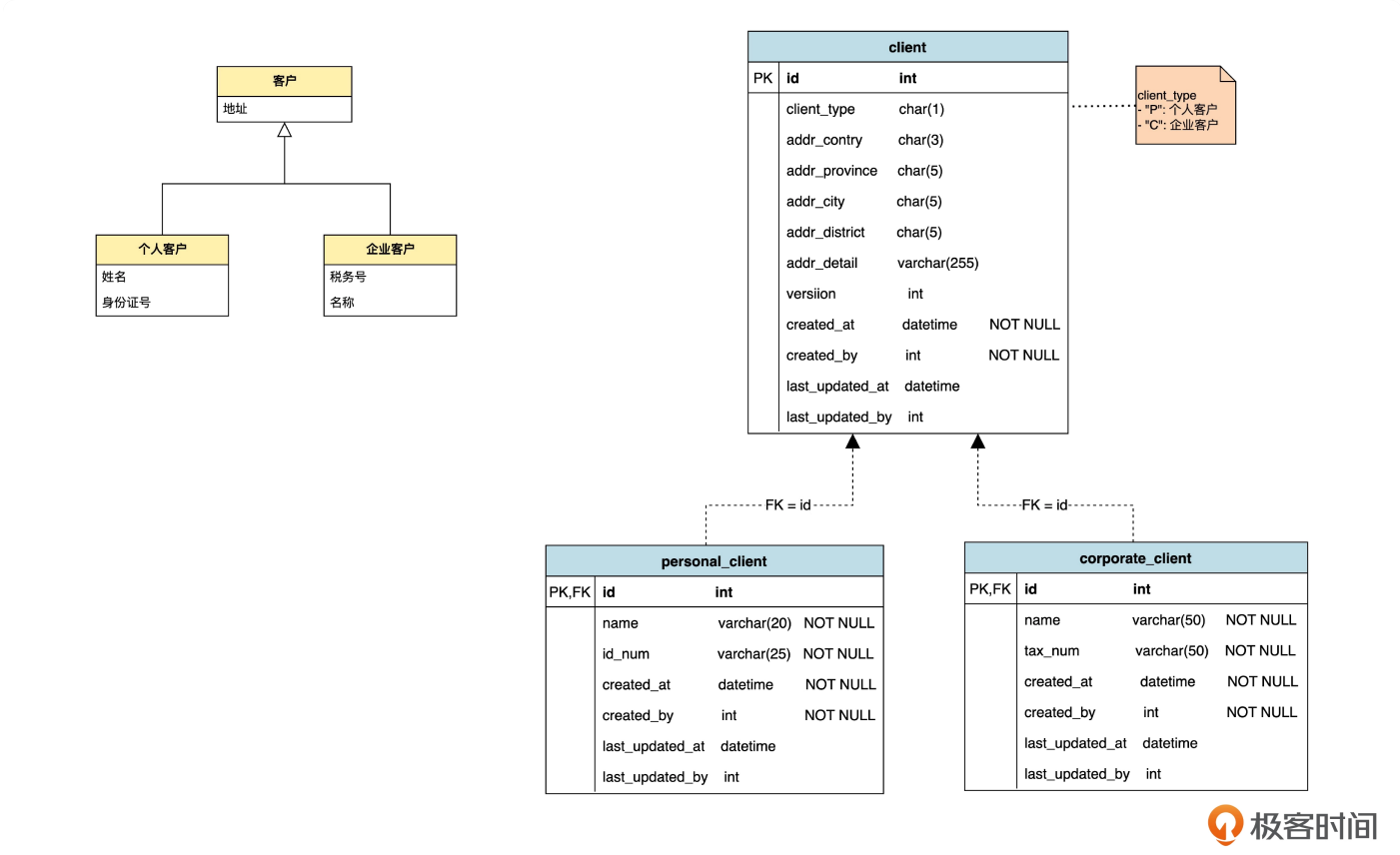

这样，我们就完成了关于工时项的例子。正如我们在上节课讲的，这种实现方式要查询三遍数据库，在性能方面还有改进余地。我们会在第三个迭代解决。

为“每个类一个表”编码

在工时项的例子里，我们采用的数据库设计策略是“每个子类一个表”。这种策略下，聚合的持久化和之前的做法变化并不大，所以仓库里有关增加和修改的代码我们就没有列出来了。

而对于“每个类一个表”，也就是为父类也建表的情况下，仓库的逻辑会更复杂一些。我们用上节课举过的个人客户和企业客户的例子来说明一下。

先回顾一下领域模型和表结构。



这里为父类和各个子类各建了一张表，并且采用“共享主键”的策略。

下面看看领域对象的代码。首先是父类 Client。

复制代码

```
package chapter26.partysample.domain.client;
// imports ...

public abstract class Client extends AggregateRoot {
    private Long id;
    private Address address;

```

```
8      // constructors ...
9
10     public abstract String getClientType();
11
12     // other getters and setters ...
13
14 }
```

关于这个父类，有三个要点需要留意。

首先，我们从第 4 行可以看到，Client（客户）类是一个抽象类，因为一个抽象的客户是不能实例化的，只有实例化具体的个人或企业客户才有意义。此外，Client 是聚合根的子类，这就意味着它的所有子类也是聚合根。

第二，从第 6 行看到，Address 是一个值对象。而对应的表是把各个地址的属性打散的，也就是我们之前提过的“嵌入”的方法。我们之前也说过，内存中的对象和数据库表的布局不一致的情况，称为“阻抗不匹配”，要通过仓库（repository）来进行转换。

第三，在数据库表里有一个 client_type 来区分是哪个子类。这个字段有两个可选值：“P” 代表个人客户，“C” 代表企业客户。那么这两个值在程序里定义在哪里呢？

一种方法是在 Client 父类里用枚举或字符串常量来定义。但这样的话，如果将来又多一个子类，就要改变 Client 或枚举的定义，这就违反了“开闭原则”。所以我们现在在父类里只定义了一个抽象方法，也就是第 10 行的 getClientType()，由子类去实现。

我们继续看子类 CorporateClient （企业客户）的代码。

复制代码

```
1  package chapter26.partysample.domain.corporateclient;
2  // import ...
3
4  public class CorporateClient extends Client {
5      public static final String CLIENT_TYPE_CORPORATE = "C";
6
7      private String name;
8      private String taxNum;
9
10     // constructors, setters and getters ...
11
12     @Override
13     public String getClientType() {
```

```
        return CLIENT_TYPE_CORPORATE;
    }
}
```

这个子类里包含企业客户独有的字段。另外，我们用一个常量写出了企业客户的 clientType 值，并通过 getClientType() 返回，这个常量在后面还会用到。

个人客户（PersonalClient）子类的实现方法也类似，就不列出来了。

下面我们重点看仓库的代码。CorporateClient 和 PersonalClient 各有一个对应的仓库。我们只看 CorporateClient 的仓库就可以了。

复制代码

```
package chapter26.partysample.adapter.driven.persistence;
// imports ...

@Repository
public class CorporateClientRepositoryJdbc
       implements CorporateClientRepository {

    final ClientDao clientDao;
    final CorporateClientDao corporateClientDao;

    // 用构造器注入 DAO
    @Autowired
    public CorporateClientRepositoryJdbc(ClientDao clientDao
            , CorporateClientDao corporateClientDao) {
        this.clientDao = clientDao;
        this.corporateClientDao = corporateClientDao;
    }

    @Override
    public boolean save(CorporateClient corporateClient) {
        switch (corporateClient.getChangingStatus()) {
            case NEW:
                addCorporateClient(corporateClient);
                break;
            case UPDATED:
                if (!updateCorporateClient(corporateClient)) {
                    return false;
                }
                break;
        }
        return true;
    }

    private void addCorporateClient(CorporateClient client) {
```

```
35          clientDao.insert(client);
36          corporateClientDao.insert(client);
37      }
38
39      private boolean updateCorporateClient(CorporateClient client) {
40          if (clientDao.update(client)) {
41              corporateClientDao.update(client);
42              return true;
43          } else {
44              return false;
45          }
46      }
47
48
49      @Override
50      public Optional<CorporateClient> findById(Long id) {
51          CorporateClient client = corporateClientDao.selectById(id);
52          return Optional.ofNullable(client);
53      }
54  }
```

先看一下第 8 行和 9 行。有没发现这里的写法和之前的迭代（例如第 10 节课）不太一样。之前，我们是把 JdbcTemplate 和 SimpleJdbcInsert 直接注入到仓库。现在我们注入的是 DAO，也就是“数据访问对象”。每个 DAO 对应一个表。JdbcTemplate 和 SimpleJdbcInsert 注入到了 DAO，由 DAO 直接访问数据库。这种做法能使程序的关注点更加分离。

也有人喜欢把 XxxDAO 命名为 XxxTable，这样更能表明和表的一一对应关系。要是使用 MyBatis 的话，可以按习惯命名为 XxxMappter，前提是规定每个表对应一个 Mapper。如果用 JPA 则没有必要用 DAO 了，因为 DAO 做的事情都被底层框架自动化了。

第 20 行的 save() 方法和之前的做法没有区别，都是根据数据是否有变化，再决定是新增还是修改。

第 34 行 addCorporateClient() 是向数据库新增企业客户，调用 DAO 分别插入 client 和 coperate_client 两个表。这里发生了内存中的一个对象向数据库里两个表的转换。

第 39 行 updateCorporateClient() 用于修改企业客户。首先修改 client 表。这里实际上用了之前学过的乐观锁的判断，没有被别人并发地抢先修改，才继续修改 corporate_client 表。也就是说，在这个泛化体系中，是在父类的表上加乐观锁，同时就把子类也锁住了。这和之前工时项不同。

第 50 行 findById() 是查询，主要逻辑在 DAO 里。

接下来我们就看一下 DAO。首先是 ClientDao。

复制代码

```
package chapter26.partysample.adapter.driven.persistence;
// imports ...

@Component
public class ClientDao {
    final JdbcTemplate jdbc;
    final SimpleJdbcInsert insert;

    @Autowired
    public ClientDao(JdbcTemplate jdbc) {
        // 注入 JdbcTemplate, 初始化 SimpleJdbcInsert ...
    }

    public void insert(Client client) {
        Address address = client.getAddress();

        Map<String, Object> parms = Map.of(
                "client_type", client.getClientType()
                , "addr_country", address.getCountry()
                , "addr_province", address.getProvince()
                , "addr_city", address.getCity()
                , "addr_district", address.getDistrict()
                , "addr_detail", address.getDetail()
                , "version", 1L
                , "created_at", client.getCreatedAt()
                , "created_by", client.getCreatedBy()
        );

        Number createdId = insert.executeAndReturnKey(parms);
        forceSet(client, "id", createdId.longValue());
    }

    public boolean update(Client client) {
        Address address = client.getAddress();
        String sql = "update client "
                + " set version = version + 1 "
                + ", addr_country =? "
                + ", addr_province =? "
                + ", addr_city =? "
                + ", addr_district =? "
                + ", addr_detail =? "
                + ", last_updated_at =?"
                + ", last_updated_by =? "
                + " where id = ? and version = ?";

```

```
46          int affected = jdbc.update(sql
47                  , address.getCountry()
48                  , address.getProvince()
49                  , address.getCity()
50                  , address.getDistrict()
51                  , address.getDetail()
52                  , client.getLastUpdatedAt()
53                  , client.getLastUpdatedBy()
54                  , client.getId()
55                  , client.getVersion());
56
57          return affected == 1;
58      }
59  }
```

这段代码有几个地方可以注意一下。

在 14 行 insert() 方法里，我们可以看到值对象 address 是怎样以内嵌的方式转化成表数据的。在第 29 行插表的过程中取得 id。由于我们采用了**共享主键的策略**，所以只在这里取一次主键，插 corporate_client 和 personal_client 的时候就直接用这个 id 了。

第 18 行，调用 getClientType()，这里你可以再体会一下之前说的开闭原则。

第 35 行的 update() 方法中，可以看到对乐观锁的实现。

还有一点，实际上，ClientDao 不仅仅会被企业客户的 CorporateClientRepositoryJdbc 调用，也会被个人客户的 PersonalClientRepositoryJdbc 所调用。这说明，分离关注点提高了程序的可复用性。

最后，我们看看用于企业客户表的 CorporateClientDao。


```
1  package chapter26.partysample.adapter.driven.persistence;
2  // imports ...
3
4  @Component
5  public class CorporateClientDao {
6
7      final JdbcTemplate jdbc;
8      final SimpleJdbcInsert insert;
9
10     @Autowired
```

```
    public CorporateClientDao(JdbcTemplate jdbc) {
        // 注入 JdbcTemplate 并初始化 SimpleJdbcInsert ...
    }

    void insert(CorporateClient client) {
       // 插入 corporate_client 表 ...
    }

    void update(CorporateClient client) {
        // 插入 corporate_client 表 ...
    }

    CorporateClient selectById(Long id) {
        String sql = " select c.version"
                + ", c.addr_country"
                + ", c.addr_province"
                + ", c.addr_city"
                + ", c.addr_district"
                + ", c.addr_detail"
                + ", cc.name"
                + ", cc.tax_num"
                + ", cc.created_at"
                + ", cc.created_by"
                + ", cc.last_update_at"
                + ", cc.last_updated_by  "
                + " from client as c"
                + "   left join corporate_client  as cc"
                + "   on c.id = cc.id "
                + " where c.id = ? and c.client_type = ? ";


        CorporateClient client = jdbc.queryForObject(sql,
                (rs, rowNum) -> {
                    Address address = new Address(
                              rs.getString("addr_country")
                            , rs.getString("addr_province")
                            , rs.getString("addr_city")
                            , rs.getString("addr_district")
                            , rs.getString("addr_detail")
                    );
                    return new CorporateClient(id
                            , rs.getString("name")
                            , rs.getString("tax_num")
                            , address
                            , rs.getTimestamp("created_at").toLocalDateTime()
                            , rs.getLong("created_by")
                            , rs.getLong("last_updated_by")
                            , rs.getTimestamp("last_updated_at").toLocalDateTim
                },
                id, CLIENT_TYPE_CORPORATE);
        return client;
    }
```

```
63
```

这里 23 行 selectById() 方法值得讲一下，用于从数据库里查询出 CorporateClient 对象。

首先，从 24 行开始，我们用了一个连表查询，同时查 client 和 corporate_client 表，因为 CorporateClient 对象的内容整体上来自于这两个表。

那么既然是查询两个表，这个逻辑应该放在 ClientDao 还是放在 CorporateClientDao 呢？

我们可以从两个角度来思考。第一个角度是，从 selectById() 的返回值可以看到，这个方法目的就是返回 CorporateClient ，那么放在 CorporateClientDao 里，在含义上更顺畅，或者说，程序员更容易凭常理推断出这个逻辑放在哪里。

第二个角度是，如果放在 ClientDao 的话，那么当我们增加关于 PersonalClient（个人客户）的逻辑时，也要类似地改 ClientDao 这个类。而如果放在 CorporateClientDao 的话，就意味着增加 PersonalClient 逻辑时，只需要把连表查询逻辑写在 PersonalClientDao 里面，而不需要修改 ClientDao 类。也就是说，这种方法更符合**开闭原则**。

所以，最终这个连表查询的逻辑，我们写在了 CorporateClientDao 里。

第 43 行开始的数据库数据向内存对象的转换逻辑里，包含了内嵌在数据表里的地址数据向 address 值对象的转换。

第 60 行的 CLIENT_TYPE_CORPORATE 实际上是定义在 CorporateClient 里的。由于这时候 CorporateClient 对象还不存在，不能用对象层面的 getClientType() 方法，只能使用在类的层面定义的常量。

顺便说一下，在 Repository 里，我们用 save 和 findByXxx 这样的方式为方法命名。而在 DAO 中用 insert、 update、 selectByXxx 这样的方式命名，目的是**更接近 SQL 语句中的命名**。这样也把两个层面的代码更好地区分开。

总结

好，这节课的主要内容就讲到这，我们来总结一下。

今天我们讨论的是泛化的代码实现。主要抓住两个点：**一是领域对象的代码采用类的继承或接口的实现；二是用仓库实现内存中的对象和数据库表中的数据之间的双向转换。**

由于在领域建模时，虽然仍然反映的是业务概念，但架构师已经刻意使模型更容易和代码设计保持一致了，所以代码直接用继承或接口实现就可以。

但是，到底用类继承还是接口实现，则要根据具体情况而定。今天工时项的例子用的就是接口实现，而客户的例子用的则是类的实现。而如果我们在写代码的时候，发现用继承或接口实现都不合适，就应该反过来修改领域模型。

在数据库设计上，工时项的例子用的是“一个子类一个表”的策略，这种策略的仓库实现起来相对简单。

而客户的例子用的是“每个类一个表”的策略，由于每个实体都牵涉到两张表，所以实现相对要复杂一些。但是，这种复杂性被仓库屏蔽掉了，除了仓库以外，代码的其他部分看不到这些复杂性。从另一个角度来说，如果数据库设计的策略改了，比如由“每个子类一个表”改成了“每个类一个表”，那么，理论上只需要修改仓库就可以了。

我们今天还讲了用仓库对嵌入式的值对象进行转换的方法。同时在代码设计上，还考虑了开闭原则，也就是“对增加打开，对修改关闭”。

思考题

我给你准备了两道思考题。

1. 在工时项的例子里，子类共用的字段只有一个**工时项 ID**，这时用接口实现问题不大。但是，如果共用的字段比较多，今天的做法就会导致较多的代码重复，在 Java 这种单继承语言的限制下，有什么更好的办法呢？

2. 在最后一段代码的 51 行创建 CorporateClient 的时候，构造器字段比较多，不是太整洁，有什么更好的办法改进呢？

好，今天的课程结束了，有什么问题欢迎在评论区留言。下节课，我们开始第三个迭代，敬请期待。

上一篇　25丨泛化的实现（上）：怎样为泛化设计数据库？

下一篇　27丨迭代三概述：怎样处理规模更大的系统？

精选留言 (7)

子衿
2023-02-09 来自上海

1. 公共字段比较多，那么首先从上节课表设计的角度，就不应该每个子类一个表了，先将表的设计改成每个类一个表，此时由于子项目仍然不能是聚合根，因此依然不能使用继承的方式，由于EffortItem中新增了属性值，又不适合作为接口，所以此时考虑将整个EffortItem作为一个属性放入到项目、子项目、普通工时项中，也就是组合替代继承，最终仍然通过Resposit

ory消除这种不匹配
2. 可以考虑为CorporateClient创建Builder

作者回复: 嗯，两个问题的思路都不错。尤其是第1题的组合思路。

5

aoe
2023-02-25 来自浙江

原来在敏捷实战中可以忽略「详细的设计图」，确实比传统的面向对象方法学要快很多

学到了在父类中使用抽象方法 getClientType() 代替 枚举类实现「开闭原则」的技巧

1

杰
2023-04-09 来自广东

“此外，Client 是聚合根的子类，这就意味着它的所有子类也是聚合根。”
老师，一个聚合不是只能有一个聚合根吗？这样的话，个人客户也是聚合根，企业客户也是聚合根，那不是冲突了吗？

作者回复: 注意一点：聚合是对象实例和实例之间的关系，不是类和类之间的关系。从类的角度有三个类：客户、个人客户、企业客户，但具体到对象，比如某个个人客户，就只有一个对象了。关键要理解这句话“聚合是对象实例和实例之间的关系，不是类和类之间的关系”

赵晏龙
2023-02-24 来自湖南

1 abstract
2 builder，另外，我一般只在构造函数中放【键】，其他不放。

作者回复: 1 关键是 abstract 也不能解决 Java 只能单继承的问题
2 没问题

请叫我和尚
2023-02-21 来自北京

在代码里的 addCorporateClient、updateCorporateClient 方法应该加事务控制吧，看文中没有加

作者回复: 建议把事务控制加在应用层，而不是Repository上面

共 2 条评论>

tt

2023-02-18 来自北京

1、按照这里的场景，因为考虑到聚合根和工时项两大特性，只能把工时项作为接口，如果共用字段比较多，那可以写一个默认实现，真正的子类在派生自它，只重写必要的方法。

2、使用builder模式。

作者回复: 嗯，是一种办法

子衿

2023-02-09 来自上海

老师这边有个问题想问一下，就是下层肯定是不应该调用上层，那么同层之间可不可以互相调用呢，看示例中，Handler和Repository都是领域层，他们间就进行了互相调用，但如果不同的两个模块的应用服务间，是不是可以互相调用呢，互相调用时，是不是就可能产生循环依赖，这种问题一般怎么解决，也是通过在领域服务层加接口，然后在适配器层实现，从而解决吗，还是有什么最佳实践

作者回复: 理论上可以互相调用。
尽量避免循环依赖。解决方式有多种，要看具体情况。抽接口也是一种常见的做法。

共 2 条评论>