17 | 聚合的实现（下）：怎样用事务保护聚合？

2023-01-12 钟敬 来自北京

《手把手教你落地DDD》

课程介绍 >

讲述：钟敬

时长 14:44　大小 13.46M

你好，我是钟敬。

上节课我们完成了**添加员工**的功能，并且实现了关于**技能**和**工作经验**的**不变规则**。今天我们重点要做两件事。第一，是继续完成**修改员工**的功能。

另外，假如不考虑并发的情况，上节课的逻辑已经足以保证不变规则了。但是正如我们在第 14 节课讲聚合概念的时候讨论的，在并发环境下，这些规则仍然可能被破坏。所以今天的第二件事就是用事务来解决这一问题。

修改聚合对象

上节课，我们在**员工**实体（Emp）里只实现了**添加技能【addSkill()】的方法。如果要修改员工聚合，我们还要编写修改技能**和**删除技能**的方法。对于**工作经验**和**岗位**也是一样的。

我们先看看在领域层实现这些逻辑的代码。


```
package chapter17.unjuanable.domain.orgmng.emp;
// imports

public class Emp extends AuditableEntity {
    //属性、构造器、其他方法 ...

    public Optional<Skill> getSkill(Long skillTypeId) {
        return skills.stream()
                .filter(s -> s.getSkillTypeId() == skillTypeId)
                .findAny();
    }

    public void addSkill(Long skillTypeId, SkillLevel level
                         , int duration, Long userId) {
        // 上节课已经实现...
    }

    public Emp updateSkill(Long skillTypeId, SkillLevel level
                           , int duration, Long userId) {
        Skill theSkill = this.getSkill(skillTypeId)
                .orElseThrow(() ->
                new BusinessException("不存在要修改的skillTypeId!"));

        if (theSkill.getLevel() != level
            || theSkill.getDuration() != duration) {

            theSkill.setLevel(level)
                    .setDuration(duration)
                    .setLastUpdatedBy(userId)
                    .setLastUpdatedAt(LocalDateTime.now())
                    .toUpdate(); //设置修改状态
        }
        return this;
    }

    public Emp deleteSkill(Long skillTypeId) {
        this.getSkill(skillTypeId)
                .orElseThrow(() -> new BusinessException(
                            "不存在要删除的skillTypeId!"))
                .toDelete(); //设置修改状态
        return this;
    }

    public void addExperience(LocalDate startDate, LocalDate endDate, String co
        durationShouldNotOverlap(startDate, endDate);
        // 与Skill的处理类似...
    }

```

```
    public Emp updateExperience(LocalDate startDate, LocalDate endDate, String
        // 与Skill的处理类似...
    }

    public Emp deleteExperience(LocalDate startDate, LocalDate endDate) {
        // 与Skill的处理类似...
    }

    public Emp addEmpPost(String postCode, Long userId) {
        // 与Skill的处理类似...
    }

    public Emp deleteEmpPost(String postCode, Long useId) {
        // 与Skill的处理类似...
    }

}
```


我们看一下 updateSkill() 方法。之前说过，我们把**技能类型 ID**（SkillTypeId）当作**技能**的局部标识，所以程序里先通过这个 ID 找到相应的**技能**。

然后，我们会比较当前**技能**和输入参数中的各个属性值。如果都相同，证明事实上不需要改变，所以什么都不需要做。只有当至少一个值不同时，才对**技能**对象进行修改。修改属性值后，要用上节课写的 toUpdate() 方法来改变**修改状态**(ChangingStatus)。

用于修改聚合的应用服务

修改完领域对象，我们来完成应用服务。

复制代码

```
package chapter17.unjuanable.application.orgmng.empservice;
// imports ...

@Service
public class EmpService {
    private final EmpRepository empRepository;
    private final EmpAssembler assembler;
    private final EmpUpdator updator; //用于修改Emp聚合
    // 构造器、其他方法...

    @Transactional
    public EmpResponse updateEmp(Long empId, UpdateEmpRequest request
                                 , User user) {
        Emp emp = empRepository.findById(request.getTenantId(), empId)
                .orElseThrow(() -> new BusinessException(
```

```
                        "Emp id(" + empId + ") 不正确！"));

        updator.update(emp, request, user);

        empRepository.save(emp);
        return assembler.toResponse(emp);
    }

}
```


在应用服务里，我们增加了 updateEmp() 方法，用来修改**员工**聚合。这个方法本身比较简单。首先从数据库中查出当前要修改的**员工**（Emp），然后调用 updator 来对聚合进行更新，最后调用仓库（empRepository）把聚合保存到数据库。

Updator 是我们新写的一个类，在地位上和 Assembler 是类似的，都是应用服务的 Helper。本来 Updator 的逻辑也可以写在 Assembler 里，但这样 Assembler 就过于庞大了，所以基于关注点分离的原则，我们单独写一个 Updator 来完成修改功能。

下面看看 Updator 的代码。

复制代码

```
package chapter17.unjuanable.application.orgmng.empservice;
// imports ...

@Component
public class EmpUpdator {
    public void update(Emp emp, UpdateEmpRequest request, User user) {
        emp.setNum(request.getNum())
                .setIdNum(request.getIdNum())
                .setDob(request.getDob())
                .setGender(Gender.ofCode(request.getGenderCode()))
                .setLastUpdatedAt(LocalDateTime.now())
                .setLastUpdatedBy(user.getId())
                .toUpdate();          // 设置修改状态

        updateSkills(emp, request, user.getId());
        updateExperiences(emp, request, user.getId());
    }

    //对技能的增删改
    private void updateSkills(Emp emp, UpdateEmpRequest request
                              , Long userId) {
        deleteAbsentSkills(emp, request);
        operatePresentSkills(emp, request, userId);
    }

```

```
    //删除目前聚合里有，但请求参数里没有的技能
    private void deleteAbsentSkills(Emp emp, UpdateEmpRequest request) {
        emp.getSkills().forEach(presentSkill -> {
            if (request.isSkillAbsent(presentSkill)) {
                emp.deleteSkill(presentSkill.getSkillTypeId());
            }
        });
    }

    //增加或修改技能
    private void operatePresentSkills(Emp emp
                          , UpdateEmpRequest request, Long userId) {
        for (SkillDto skill : request.getSkills()) {
            Optional<Skill> skillMaybe = emp.getSkill(
                                              skill.getSkillTypeId());
            if(skillMaybe.isPresent()) {
                emp.updateSkill(skill.getSkillTypeId()
                        , SkillLevel.ofCode(skill.getLevelCode())
                        , skill.getDuration()
                        , userId);
            } else {
                emp.addSkill(skill.getSkillTypeId()
                        , SkillLevel.ofCode(skill.getLevelCode())
                        , skill.getDuration()
                        , userId);
            }
        }

    }

    private void updateExperiences(Emp emp, UpdateEmpRequest request
                                   , Long userId) {
        // 与updateSkilL()类似...
    }

}
```


这里，程序逻辑的起点是 update() 方法。它首先修改员工对象的值，并调用 toUpdate() 方法设置**修改状态，**然后分别调用另外两个私有方法 updateSkills() 和 updateExperiences() 来修改技能和工作经验。我们假定按照业务需求，更改员工的岗位是单独的服务，所以这里没有修改岗位。

updateSkills() 方法用于修改技能，它包括两步。

首先是调用 deleteAbsentSkills() 来删除不存在的技能。逻辑是，比较请求参数（request）和当前员工聚合里的技能。

如果当前聚合有某个技能，但请求参数里没有，就认为用户希望删除这条技能，所以会调用 emp.deleteSkill() 方法来删除。这时并没有真的在内存里删除，只是修改了技能的修改状态，以便在持久化时在数据库里删除。对于技能是否存在，我们也是通过局部 ID（skillTypeId）来判断的。

第二步，调用 operatePresentSkills() 方法来处理请求参数里存在的技能。如果请求参数里的技能在当前聚合里存在，就更改，否则就增加。由于既可能是更改，也可能是增加，所以方法名用了 operate (操作）。

对于**工作经验**的修改是类似的，你可以参考前面的讲解自己试试。

聚合的查询

接下来我们来完成持久层。在 EmpService 里，有两处调用 empRepository 和持久层交互。一处是调用 empRepository.findById() 根据租户和员工 ID 查找要修改的员工，另一处是调用 empRepository.save() 来保存员工聚合。

咱们先看查询。由于聚合在逻辑上是一个整体，并且我们采用了在聚合内部用对象导航的策略，所以我们会把**员工**实体和从属于它的**技能**、**工作经验**和**岗位**都一次性取到内存。

乍一看，应该不太复杂，但这里会遇到一个问题。从数据库重建**员工**（Emp）聚合的过程中，当我们调用 Emp 的一些方法赋值的时候，会触发业务规则的校验。比如说，调用 addSkill() 增加技能的时候，会触发“技能类型不允许重复”的校验。

那么重建聚合的时候，是否应该进行这种校验呢？

这取决于数据的“干净程度”。如果数据库中的数据比较“脏”，也就是说数据库里很多数据已经违反了业务规则，那么，可能在重建聚合时再校验一遍业务规则是可取的，这样可以找出脏数据错误。

不过多数情况下，数据库是比较干净的。这时候，如果每次从数据库取数据都要校验一遍，就会无谓地影响性能。

那么怎样绕过这些规则呢？有多种方法。我们的例子里采用这样的技巧：先把 Emp 中的属性都改成 protected 的，然后写一个 Emp 的子类，这个子类中的方法也可以设置 Emp 的值，但

是不调用业务规则，这样就达到了绕过业务规则的目的。

下面是这个子类的代码。

复制代码

```
//这个类位于适配器包
package chapter17.unjuanable.adapter.driving.persistence.orgmng;
//imports...

public class RebuiltEmp extends Emp {
    RebuiltEmp(Long tenantId, Long id, LocalDateTime create_at, long created_by
        super(tenantId, id, create_at, created_by);
        //由于是从数据库重建，所以状态默认为"不变"
        this.changingStatus = ChangingStatus.UNCHANGED;
    }

    //包级权限，并且用 resetXxx 命名
    RebuiltEmp resetOrgId(Long orgId) {
        this.orgId = orgId;
        return this;
    }

    RebuiltEmp resetNum(String num) {
        this.num = num;
        return this;
    }

    RebuiltEmp resetIdNum(String idNum) {
        this.idNum = idNum;
        return this;
    }

    RebuiltEmp resetName(String name) {
        this.name = name;
        return this;
    }

    RebuiltEmp resetGender(Gender gender) {
        this.gender = gender;
        return this;
    }

    RebuiltEmp resetDob(LocalDate dob) {
        this.dob = dob;
        return this;
    }

    RebuiltEmp resetStatus(EmpStatus status) {
        this.status = status;
        return this;
```

```
    }

    // 用 reAddXxx 命名
    public RebuiltEmp reAddSkill(Long id, Long skillTypeId, SkillLevel level, i

        RebuiltSkill newSkill = new RebuiltSkill(tenantId, id, skillTypeId, cre
                .resetLevel(level)
                .resetDuration(duration);

        skills.add(newSkill);
        return this;
    }

    public RebuiltEmp reAddExperience(LocalDate startDate, LocalDate endDate, S
        // ...
    }

    public RebuiltEmp reAddEmpPost(String postCode, Long userId) {
        // ...
    }

}
```


首先，这个子类和**员工仓库**的实现（EmpRepositoryJdbc）放在同一个包，类中的方法都是包级私有的，也就是说，只有**员工仓库**的实现类可以访问，从而避免了这个包外部的其他类绕过业务规则。

这个类的名字是 RebuiltEmp，也就是“重建的”**员工**。对应于父类（Emp）里的 setXxx() 方法，这里我们 setter 用 resetXxx() 来命名，以示区别。类似地，我们也用 reAddXxx() 来增加**技能**、**工作经验**和**岗位**。另外，这些方法都返回 RebuildEmp 对象本身，以便对这个对象进行链式操作。

有了这个子类，我们就可以实现仓库了。

复制代码

```
package chapter17.unjuanable.adapter.driving.persistence.orgmng;
// imports...


@Repository
public class EmpRepositoryJdbc implements EmpRepository {
   //声明 JdbcTemplate 和各个 SimpleJdbcInsert ...
   // 构造器、其他方法 ...

    @Override
```

```
    public Optional<Emp> findById(Long tenantId, Long id) {
        Optional<RebuiltEmp> empMaybe = retrieveEmp(tenantId, id);
        if (empMaybe.isPresent()) {
            RebuiltEmp emp = empMaybe.get();
            retrieveSkills(emp);
            retrieveExperiences(emp);
            retrievePosts(emp);
            return Optional.of(emp);
        } else {
            return Optional.empty();
        }
    }

    private Optional<RebuiltEmp> retrieveEmp(Long tenantId, Long id) {
        String sql = " select org_id, num, id_num, name "
                + " , gender_code, dob, status_code "
                + " from emp "
                + " where id = ? and tenant_id = ? ";

        RebuiltEmp emp = jdbc.queryForObject(sql,
                (rs, rowNum) -> {
                    RebuiltEmp newEmp = new RebuiltEmp(tenantId
                            , id
                            , rs.getTimestamp("create_at").toLocalDateTime()
                            , rs.getLong("created_by"));

                    newEmp.resetOrgId(rs.getLong("org_id"))
                          .resetNum(rs.getString("num"))
                          .resetIdNum(rs.getString("id_num"))
                          .resetName(rs.getString("name"))
                          .resetGender(Gender.ofCode(
                                  rs.getString("gender_code")))
                          .resetDob(rs.getDate("dob").toLocalDate())
                          .resetStatus(EmpStatus.ofCode(
                                  rs.getString("status_code")));
                    return newEmp;
                },
                id, tenantId);

        return Optional.ofNullable(emp);
    }

    private void retrieveSkills(RebuiltEmp emp) {
        String sql = " select id, tenant_id, skill_type_id, level, duration "
                + " from skill "
                + " where tenant_id = ? and emp_id = ? ";

        List<Map<String, Object>> skills = jdbc.queryForList(
                            sql, emp.getTenantId(), emp.getId());

        skills.forEach(skill -> emp.reAddSkill(
                (Long) skill.get("id")
```

```
                    , (Long) skill.get("skill_type_id")
                    , SkillLevel.ofCode((String) skill.get("level_code"))
                    , (Integer) skill.get("duration")
                    , (Long) skill.get("created_by")
        });

    }

    private void retrieveExperiences(RebuiltEmp emp) {
        //与retrieveSkill 类似 ...
    }

    private void retrievePosts(RebuiltEmp emp) {
        //与retrieveSkill 类似 ...
    }
}
```


FindById() 方法首先会从数据库重建 Emp 对象本身，然后分别重建**技能**、**工作经验**和**岗位**。与数据库直接打交道的方法，用 retrieveXxx() 来命名，以便和更上层的 FindByXxx() 相区别。

对修改的聚合进行持久化

完成了查询功能，我们来看怎样把修改后的聚合存入数据库。无论新增还是修改聚合，我们都可以用同一个 empRepository.save() 方法，所以我们要对之前课程中的这个方法进行修改。代码如下。

复制代码

```
package chapter17.unjuanable.adapter.driving.persistence.orgmng;
// imports ...

@Repository
public class EmpRepositoryJdbc implements EmpRepository {

    final JdbcTemplate jdbc;
    final SimpleJdbcInsert empInsert;
    final SimpleJdbcInsert skillInsert;
    final SimpleJdbcInsert WorkExperienceInsert;
    final SimpleJdbcInsert empPostInsert;

    @Autowired
    public EmpRepositoryJdbc(JdbcTemplate jdbc) {
        this.jdbc = jdbc;
        this.empInsert = new SimpleJdbcInsert(jdbc)
                .withTableName("emp")
                .usingGeneratedKeyColumns("id");
        //初始化其他 SimpleJdbcInsert ...
```

```
    }

    @Override
    public void save(Emp emp) {
        saveEmp(emp);
        emp.getSkills().forEach(s -> saveSkill(emp, s));
        emp.getExperiences().forEach(e -> saveWorkExperience(emp, e));
        emp.getEmpPosts().forEach(p -> saveEmpPost(emp, p));
    }

    private void saveEmp(Emp emp) {
        switch (emp.getChangingStatus()) {
            case NEW:
                insertEmpRecord(emp);
                break;
            case UPDATED:
                updateEmpRecord(emp);
                break;
        }
    }

    private void insertEmpRecord(Emp emp) {
        Map<String, Object> parms = Map.of(
                "tenant_id", emp.getTenantId()
                , "org_id", emp.getOrgId()
                , "num", emp.getNum()
                , "id_num", emp.getIdNum()
                , "name", emp.getName()
                , "gender", emp.getGender().code()
                , "dob", emp.getDob()
                , "status", emp.getStatus().code()
                , "created_at", emp.getCreatedAt()
                , "created_by", emp.getCreatedBy()
        );

        Number createdId = empInsert.executeAndReturnKey(parms);

        forceSet(emp, "id", createdId.longValue());
    }

    private void updateEmpRecord(Emp emp) {
        String sql = "update emp " +
                " set org_id = ?" +
                ", num = ?" +
                ", id_num =? " +
                ", name = ?" +
                ", gender =?" +
                ", dob = ?" +
                ", status =?" +
                ", last_updated_at =?" +
                ", last_updated_by =? " +
                " where tenant_id = ? and id = ? ";
```

```
72          this.jdbc.update(sql
73                  , emp.getOrgId()
74                  , emp.getNum()
75                  , emp.getIdNum()
76                  , emp.getName()
77                  , emp.getGender().code()
78                  , emp.getDob()
79                  , emp.getStatus()
80                  , emp.getLastUpdatedAt()
81                  , emp.getLastUpdatedBy()
82                  , emp.getTenantId()
83                  , emp.getId());
84      }
85
86      private void saveSkill(Emp emp, Skill skill) {
87          switch (skill.getChangingStatus()) {
88              case NEW:
89                  insertSkillRecord(skill, emp.getId());
90                  break;
91              case UPDATED:
92                  updateSkillRecord(skill);
93                  break;
94              case DELETED:
95                  deleteSkillRecord(skill);
96                  break;
97
98          }
99      }
100
101     private void insertSkillRecord(Skill skill, Long empId) {
102         Map<String, Object> parms = Map.of(
103             "emp_id", empId,
104             "tenant_id", skill.getTenantId(),
105             "skill_type_id", skill.getSkillTypeId(),
106             "level_code", skill.getLevel().code(),
107             "duration", skill.getDuration(),
108             "created_at", skill.getCreatedAt(),
109             "created_by", skill.getCreatedBy()
110             );
111
112         Number createdId = skillInsert.executeAndReturnKey(parms);
113
114         forceSet(skill, "id", createdId.longValue());
115     }
116
117     private void updateSkillRecord(Skill skill) {
118         String sql = "update skill "
119                     + " set level_code = ?"
120                     + ", duration = ?"
121                     + ", last_updated_at = ?"
122                     + ", last_updated_by = ?"
123                     + " where tenant_id = ? and id = ? ";
```

```
        this.jdbc.update(sql
                , skill.getSkillTypeId()
                , skill.getDuration()
                , skill.getLastUpdatedAt()
                , skill.getLastUpdatedBy()
                , skill.getTenantId()
                , skill.getId());
    }


    private void deleteSkillRecord(Skill skill) {
        this.jdbc.update("delete from skll where tenant_id = ? "
                        + " and id = ?"
                , skill.getTenantId()
                , skill.getId());
    }

    private void saveWorkExperience(Emp emp, WorkExperience e) {
        // 与 saveSkill( ) 类似...
    }

    private void saveEmpPostRecord(Emp emp, EmpPost p) {
        // 与 saveSkill( ) 类似...
    }


```


save() 方法先调用 saveEmp() 方法，根据**员工**对象的修改状态（changingStatus），来插入或更新 emp 表，然后用同样的逻辑循环处理**技能**、**工作经验**和**岗位**。

我们假定将来会写专门的 removeEmp() 方法删除整个聚合，所以目前的 saveEmp() 中没有处理删除的情况。另外，对于直接操作数据库的类，我们用 insertXxxRecord() 的方式的命名，与更上一层的 saveXxx() 方法相区别。

用事务保证固定规则

完成了修改聚合的基本功能后，我们来考虑避免并发情况下破坏不变规则的问题。我们在第 14 节课已经讲过，需要把对聚合的修改封装到一个事务中去，这样，一个人修改完以后，另一个人才能修改，从而避免并发修改的问题。那么具体怎么做呢？

首先，我们要考虑一个问题，仅仅靠数据库事务，是无法完成这一任务的，需要自己编写一些代码来完成。这种比数据库事务“高一级”的事务，我们可以称为“业务事务”（Business Transaction）。业务事务一般要使用乐观锁或者悲观锁的机制。

悲观锁指的是，只要一个人开始修改操作，就为数据加锁，其他人根本不可能同步修改。乐观锁指的是，两个人可以同时操作，但最后保存到数据库的时候，先保存的那个人成功，后保存的那个人失败，只能重新进行操作。

我们这里选择乐观锁。对于聚合的情况而言，实际上是通过锁聚合根，来把整个聚合锁住。我们一步一步地看一看做法。

第一步，要在聚合根的代码和数据表里增加一个**版本**（version）字段，类型可以是长整型。由于多数聚合都要考虑加锁，所以我们为聚合根写一个父类，这个类又是 AuditableEntity 的子类。后面是具体代码。

复制代码

```
package chapter17.unjuanable.common.framework.domain;

import java.time.LocalDateTime;

public class AggregateRoot extends AuditableEntity {
    protected Long version;

    public AggregateRoot(LocalDateTime createdAt, Long createdBy) {
        super(createdAt, createdBy);
    }

    public Long getVersion() {
        return version;
    }
}
```

Emp 原来继承的是 AuditableEntity, 现在改为继承 AggregateRoot，其他部分不需要修改。这样，Emp 就有了 version 属性。

复制代码

```
public class Emp extends AggregateRoot {
  //...
}
```

第二步，修改 EmpRepository 中的 findById() 方法，在取数据的时候，把 Emp 的 verion 值也取出来。逻辑比较简单，这里就不列代码了。

第三步，是在 update Emp 表的时候，修改 SQL 语句，这一步是最关键的，我们先看代码。


```
package chapter17.unjuanable.adapter.driving.persistence.orgmng;
// imports ...

@Repository
public class EmpRepositoryJdbc implements EmpRepository {

    // 声明 JdbcTemplate, SimpleJdbcInsert empInsert ...
    // 构造器，其他方法不变 ...

    @Override
    public boolean save(Emp emp) {
        if (saveEmp(emp)) {
            emp.getSkills().forEach(s -> saveSkill(emp, s));
            emp.getExperiences().forEach(e -> saveWorkExperience(emp, e));
            emp.getEmpPosts().forEach(p -> saveEmpPost(emp, p));
            return true;
        } else {
            return false;
        }
    }

    private boolean saveEmp(Emp emp) {
        switch (emp.getChangingStatus()) {
            case NEW:
                insertEmpRecord(emp);
                break;
            case UPDATED:
                if(!updateEmpRecord(emp)) {
                    return false;
                }
                break;
        }
        return true;
    }

    private void insertEmpRecord(Emp emp) {
       // 代码不变 ...
    }

    // 注意：SQL语句中增加了两处关于 version 的修改
    private boolean updateEmpRecord(Emp emp) {
        String sql = "update emp " +
                " set version = version + 1 " +
                ", org_id = ?" +
                ", num = ?" +
                ", id_num =? " +
                ", name = ?" +
                ", gender =?" +
```

```
                ", dob = ?" +
                ", status =?" +
                ", last_updated_at =?" +
                ", last_updated_by =? " +
                " where tenant_id = ? and id = ? and version = ?";
        int affected = this.jdbc.update(sql
                , emp.getOrgId()
                , emp.getNum()
                , emp.getIdNum()
                , emp.getName()
                , emp.getGender().code()
                , emp.getDob()
                , emp.getStatus()
                , emp.getLastUpdatedAt()
                , emp.getLastUpdatedBy()
                , emp.getTenantId()
                , emp.getId()
                , emp.getVersion());

        return affected == 1 ? true : false;
    }

    // 其他方法不变 ...
}
```


这里重点是 updateEmpRecord() 方法里 SQL 语句的变化。SQL 语句里增加了两处关于 version 的修改，其他部分不变。


```
update emp set version = version + 1
...
where  version = <当前Emp中的version值>
```

也就是说，根据当前 Emp 里的 version 值，找到记录，然后把 version 值加 1 。

我们想象一下，两个人几乎同时修改**员工**，但最后 update 语句的执行总有一个先后。

先 update 的人是可以根据原来的 version 值取到记录的，因为这时 version 值还没变。而后 update 的人，由于数据库里的 version 值已经被刚才的人加 1 了，所以无法通过原来的 version 找到记录，会导致更新失败，也就不会破坏业务规则。这就是乐观锁的诀窍。

我们再看回 updateEmpRecord() 方法，它的返回值由原来的 void 改成了 boolean，表示修改是否成功。update 语句执行后，会返回被 update 的记录数量。如果返回为 1，证明修改成功，则这个方法返回 true；如果返回 0 ，说明修改失败，也就是已经被别人抢先修改了，这时返回 false。

调用 updateEmpRecord() 的 saveEmp() 和再上层的 save() 的返回值也都改成了 boolean。updateEmpRecord() 的成功状态经由 saveEmp() 返回给 save() 。save() 方法只有在保存**员工**成功的时候才进一步保存**技能**、**工作经验**和**岗位**，否则，不会继续操作，而是返回 false。

而 save() 方法又是由应用服务 EmpService() 调用的。EmpService() 的代码如下。

复制代码

```
package chapter17.unjuanable.application.orgmng.empservice;
// imports ...

@Service
public class EmpService {
    // 依赖注入、构造器、其他方法 ...

    @Transactional
    public EmpResponse updateEmp(Long empId, UpdateEmpRequest request
                                 , User user) {
        Emp emp = empRepository.findById(request.getTenantId(), empId)
                .orElseThrow(() -> new BusinessException(
                        "Emp id(" + empId + ") 不正确！"));

        updator.update(emp, request, user);

        // 这里增加了判断
        if(!empRepository.save(emp)) {
            throw new BusinessException(
                        "这个员工已经被其他人同时修改了，请重新修改！");
        };
        return assembler.toResponse(emp);
    }

}
```

EmpService 的 updateEmp() 方法会判断保存是否成功，如果不成功，则可推断出是其他人抢先修改了，于是抛出异常，提示当前用户重新修改。

单实体聚合

现在，我们已经完成了聚合代码的编写。最后再讨论一个问题：有些实体，既不是聚合根，也不从属于任何聚合，例如上个迭代讲过的组织（Org）实体，对于这些实体该怎么处理呢？

我们建议，把这种“游离”的实体看做一种“退化”的聚合，也就是说，它们也是聚合，只不过只有聚合根，没有“儿子”，可以称为“单实体聚合”。

比如说，**组织**实体就构成了一个单实体聚合，它本身就是聚合根，在代码层面可以和普通聚合一样处理。也就是说，这些实体也在自己单独的包内，这个包里面通常包括仓库的接口，有时还包括工厂和领域服务。事实上，上个迭代对**组织**的处理，就是这么做的。

但是在领域模型图里，如果把每个单实体聚合外面都套一个“包”的话，模型图就显得太凌乱了，所以在模型图上就没有必要为单独的实体加上包了。这时，模型和代码稍微有些不一致，算是一种妥协吧。

总结

好，这节课的主要内容就讲完了，下面我们来总结一下。今天主要解决的是聚合的修改，以及在并发环境下保护聚合不变规则的问题。

对于聚合的修改，有以下要点。

第一，在修改之前，要把聚合从数据库里取出来。为了这个目的，仓库要把聚合的数据整体装入内存，并重建聚合。这里我们还用了一个技巧，在仓库包里建立了聚合根的一个子类，从而绕过校验规则，避免不必要的性能损耗。

第二，要在领域层的聚合根里增加对技能、工作经验和岗位的更改和删除代码，并为这些对象设置合适的修改状态，从而把非聚合根对象的修改逻辑封装起来。

第三，在应用层把当前聚合与请求参数进行对比，确定对聚合里的各个对象应该进行增、删、改，还是保持不变。然后，调用聚合根来进行相应的操作。

最后，为了把聚合存入数据库，仓库要遍历聚合中的各个对象，根据对象的更改状态进行合适的数据库操作。

完成了聚合的修改以后，我们展示了怎样用乐观锁保护聚合的事务边界，避免并发操作对不变规则的破坏。此外，我们还讨论了单实体聚合的处理。

在介绍聚合概念的那节课里，我们讲了聚合的两大特征：一个是**概念上的整体性**；另一个是**维护不变规则的要求**。在这三节课，你应该能体会到怎样从代码层面实现这些聚合的特征了吧。

还有一点要注意，尽管我们目前选择的是偏过程式的编码风格，但是也会尽量实现封装、继承等面向对象编程的特征，这一点也是要着重体会的。

思考题

1. 我们在重建聚合时，采用了编写聚合子类的方式绕过业务规则的校验，你还能想到其他方法吗？

2. 如果用悲观锁的话，应该怎样实现？

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，我们开始讲解值对象和其他一些建模技巧。

上一篇　16丨聚合的实现（中）：怎样实现不变规则？

下一篇　18丨值对象（上）：到底什么是值对象？

精选留言 (8)

escray
2023-01-30 来自北京

UpdateEmpRequest 这个类的代码似乎之前没有出现过，感觉类似一个记录员工属性的”值对象“？

对于聚合的查询，本文使用了”重建“的方式，但是我的印象里面似乎 myBatis 里面自带了查询语句的生成？

对于思考题，

1. 在重建聚合的时候，是否可以使用类似于 myBatis 的默认方式？或者把所有的验证都抽取到独立的一个 empValidator 方法里面去，可以在查询的时候不调用。
2. 悲观锁应该就是在 update 的时候加锁，修改聚合根（员工）的 Update SQL 语句。


Johar
2023-01-25 来自重庆

1. 我们在重建聚合时，采用了编写聚合子类的方式绕过业务规则的校验，你还能想到其他方法吗？
重建聚合时，主要是查重的逻辑和新建聚合校验逻辑不一致，可以考虑把校验放在db层，设置唯一所以来解决，查询数据的方式，在高并发下，还是有可能重复。
2. 如果用悲观锁的话，应该怎样实现？
一种，可以使用redis的全局锁，A进入某员工编辑界面，就不允许其他人进入该员工编辑页面；另外一种，也是使用全局锁，后端收到更新用户信息的请求时，使用key+租户id+员工id作为key锁定资源。


iam593
2023-01-24 来自湖南

继承于AuditableEntity的对象，在数据库中对应的表都有创建者、创建时间、修改者、修改时间等字段？从数据库层面看，这样会不会有点繁琐？


aoe
2023-01-17 来自浙江

思考题
1. 将之前创建聚合对象的方法提取到新的类中（例如 EmpCreater)，这里只负责创建工作，不进行规则校验。提取后的方法可以被有规则校验与无规则校验的方法共同使用。
2. 编辑的一开始，在数据库里插入一条记录（例如使用“员工 id”做唯一标识，设置为唯一索引），插入成功后再进行修改操作，否则就拒绝编辑；当编辑完成后，再删除使用“员工 id”做唯一标识的这条数据。

读后感

1. Updator 单独写一个类挺好
2. RebuiltEmp，从数据库加载数据，不调用业务规则，绕过业务规则创建对象的方式对性能提升确实有帮助
3. 当读到下列 3 段代码时，已经不记得是如何实现的，不能完全理解代码，感觉是时候跟着钟老师实现一遍代码了。

```
emp.deleteSkill(presentSkill.getSkillTypeId());
emp.updateSkill(skill.getSkillTypeId() , SkillLevel.ofCode(skill.getLevelCode()) , skill.getDuration() , userId);
emp.addSkill(skill.getSkillTypeId() , SkillLevel.ofCode(skill.getLevelCode()) , skill.getDuration() , userId);
```


张强
2023-01-13 来自北京

老师您好：针对以下代码有个疑问？

```
@Override
public boolean save(Emp emp) {
    if (saveEmp(emp)) {
        emp.getSkills().forEach(s -> saveSkill(emp, s));
        emp.getExperiences().forEach(e -> saveWorkExperience(emp, e));
        emp.getEmpPosts().forEach(p -> saveEmpPost(emp, p));
        return true;
    } else {
        return false;
    }
}
```

1. 如果saveEmp 成功了， 在保存saveEmpPost 时，saveEmpPost 方法有没有可能被其他并发修改？
改成以下 是否能解决1问题？也就后保存聚合根。

```
@Override
public boolean save(Emp emp) {

 emp.getSkills().forEach(s -> saveSkill(emp, s));
        emp.getExperiences().forEach(e -> saveWorkExperience(emp, e));
        emp.getEmpPosts().forEach(p -> saveEmpPost(emp, p));

    if (saveEmp(emp)) {
```

```
            return true;
        } else {
            return false;
        }
    }
```


作者回复: 你好呀，假定我们用的数据库隔离方式是常见的read commited,在新增整个聚合时，saveemp成功后，由于事务没有提交，所以其他线程无法读到新的emp,也就不会发生并发修改错误。对于修改聚合的情况，在乐观锁的保护下，也不会出错。


张逃逃

2023-01-12 来自北京

有个疑问想请教老师，为什么EmpRepository在查找Emp的时候不把对应Emp的所有状态(包括技能，工作经验...)全部查出来，然后通过Emp的构造参数来实例化对象，而是先实例化对象再调用addSkill()等方法来初始化，如果用构造方法来实例化对象，好像就不需要RebuiltEmp了。

作者回复: 你说的也是一种可行的做法。Evans认为，如果构造器太复杂，就掩盖了对象的主要职责，所以这时候倾向于把构造的职责抽出来。


南山

2023-01-12 来自江苏

1.能直接从数据库中查询值构造聚合对象，不做任何检查或者校验可行吗？
2.查询emp就加写锁，语句使用forUpdate

PS：这种方式的修改聚合很有启发性

作者回复: 第一点，关键是从数据库查到值以后，怎么构建领域对象。第二点，确实是悲观锁的可行做法。


Karson

2023-01-12 来自辽宁

是否有课程源码呢？

作者回复: 有，不过要等课程结束后整理一下再放出来