

各软件设计原则在本项目中的应用

姜九鸣 201900302022

单一职责原则在本项目中的体现

- 该原则可以理解为：实现一个类，它的职责必须是单一的。类的职责要单一，不能将太多的职责放在一个类中。



- 在本项目中，我们让每个 controller 类只能有一个职责。当需要修改该职责时，不会导致其它问题。因为如果这个类有两个职责，那么修改器中一个职责时，可能导致另一个职责不能正常工作。
- 如图，每个 controller 负责相应一种前后端接口的实现。例如 CommentController（歌曲/歌单评论相关 API）的实现：

```
public class CommentController {
    @Autowired
    private CommentServiceImpl commentService;

    // 提交评论
    @ResponseBody
    @RequestMapping(value = "/comment/add", method = RequestMethod.POST)
    public Object addComment(HttpServletRequest req) {
        String user_id = req.getParameter("userId");
        String type = req.getParameter("type");
        String song_list_id = req.getParameter("songListId");
        String song_id = req.getParameter("songId");
        String content = req.getParameter("content").trim();

        Comment comment = new Comment();
        comment.setUserId(Integer.parseInt(user_id));
        comment.setType(new Byte(type));
        if (new Byte(type) == 0) {
            comment.setSongId(Integer.parseInt(song_id));
        } else if (new Byte(type) == 1) {
            comment.setSongListId(Integer.parseInt(song_list_id));
        }
        comment.setContent(content);
        comment.setCreateTime(new Date());
        boolean res = commentService.addComment(comment);
        if (res) {
            return new SuccessMessage<Null>(message: "评论成功").getMessage();
        } else {
            return new ErrorMessage(message: "评论失败").getMessage();
        }
    }
}
```

- 在这里，该类只负责实现与歌曲/歌单评论相关的功能，而与其他功能无关。例如它就不可能实现歌单创建、用户信息维护等功能。这是不符合单一职责原则的。
- 换言之，我们以这种方式来构建我们的项目，可以大大提升代码的可维护性，因为类的职责都是单一的，他们之间的耦合性几乎不存在。

开闭原则在本项目中的体现

- 可以认为，开闭原则是软件设计的总纲要。软件实体对扩展是开放的，但对修改是关闭的，即在不修改一个软件实体的基础上去扩展其功能。
- 软件实体类应当对扩展开放，对修改关闭。因此，要对软件功能进行升级，不要求修改写好的代码，而是利用原来的接口进行扩展。
- 以本项目中的登录页面为例，它需要一个登录表单，界面如下。



- 前端代码如下。

```
<div class="sign">
  <div class="sign-head">
    <span>帐号登录</span>
  </div>
  <el-form ref="signInForm" status-icon :model="registerForm" :rules="SignInRules">
    <el-form-item prop="username">
      <el-input placeholder="用户名" v-model="registerForm.username"></el-input>
    </el-form-item>
  </el-form>
</div>
```

- 注意到，`<el-form ref="signInForm" status-icon :model="registerForm" :rules="SignInRules">`定义了这一表单的 DOM 元素。
- 其中，`model="registerForm"` 指出了这个表单是什么格式的，它的定义如下。

```
// 登录用户名密码
const registerForm = reactive({
  username: "",
  password: "",
});
```

- 这是一个 Reactive 表单，由两个域构成，分别是 `username` 和 `password`。

- 开闭原则在这样的前端框架中有比较明显的体现。例如 el-form 元素需要属性 model，它是一个抽象的接口，需要传入一个实例化的表单，也就是下面我们定义的 registerForm。
- 于是，就可以通过定义新的具体表单，来修改 el-form 的行为了（或者说对他进行扩展了），而不需要修改 el-form 本身的代码，这就是开闭原则体现了。

里氏替换原则在本项目中的体现

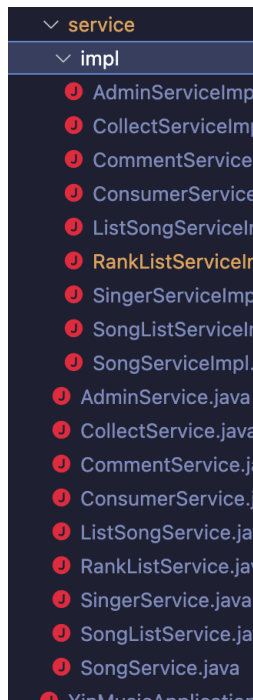
- 该原则可以理解为不要破坏继承体系。只要父类出现的地方子类就可以出现，换成子类也不会有什么异常，但是反之就不行了。子类可以增加自己的方法，反正就算替换了也不会被调用到。

```
@Override
public boolean addListSong(ListItem listSong)
{
    return listSongMapper.insertSelective(listSong) > 0;
}
```

- 例如本项目中的歌曲列表相关实现。要实现这一点，子类不要重写父类的方法；如果真的要重载，那参数的要求只能低于父亲的；如果真的要重载，那返回值的要求只能高于父亲的。否则就不能完成原地替换了。
- 歌曲列表的每个 entry 是一个 ListItem 对象，实际上是“基类”的实例。我们还有 ListSongItem、ListSongListItem 等派生自这个类。基类可以认为是列表对象的每一个项目，而后两者分别可以理解为歌曲列表中的项目、歌单列表中的项目。他们只是在原有的 ListItem 对象内添加了相关属性，从而构成了更丰富的细节。
- 于是，只要 ListItem 对象出现的地方，其子类的对象 ListSongItem、ListSongListItem 等就可以出现，而且直接换成他们也不会有什么异常，但是反之就不行了，因为 ListSongItem、ListSongListItem 等添加了 ListItem 所不具有的细节。也就是子类可以增加自己的方法，反正就算替换了也不会被调用到。这就是里氏替换原则在本项目中的体现。

依赖倒置原则在本项目中的体现

- 依赖倒置原则要面向接口编程。高层不应依赖低层的细节，而是依赖低层的抽象。这就要求我们每个类一定都要继承自抽象类或接口。
- 以本项目后端的 service 层为例。如图。



- 在功能设计极端，我们首先定义一系列接口，然后每个 service 类都是其同名接口的实现，如下。

```
@Service
public class ListSongServiceImpl implements ListSongService {

    @Autowired
    private ListSongMapper listSongMapper;

    @Override
    public List<ListSong> allListSong()
    {
        return listSongMapper.allListSong();
    }


    @Override
    public boolean updateListSongMsg(ListSong listSong) {
        return listSongMapper.updateListSongMsg(listSong) > 0;
    }

    @Override
    public boolean deleteListSong(Integer songId) {
        return listSongMapper.deleteListSong(songId) > 0;
    }
}
```

- 这样一来，我们在逻辑设计的过程中就可以仅针对抽象层编程，而不要针对具体类编程。这极大地方便了我们的软件设计。设想如果在后端逻辑上需要做出调整，那么如果面向具体实现去做出调整，是及其困难的。
- 应当针对接口来编程。然后在后期再逐步实现这些接口，就能使程序条理更加清晰。

合成复用原则、迪米特原则在本项目中的体现

- 迪米特原则的要点在于降低代码的耦合度。如果两个软件实体无须直接通信，那么就不应当发生直接的相互调用，可以通过第三方转发该调用。因为耦合性越高，一个发生更改，对另一个的影响越强烈。
- 本项目的后端业务采用 MVC 架构，其中 Model 负责界面的显示，以及与用户的交互功能，例如表单、网页等。View 可以理解为一个分发器，用来决定对于视图发来的请求，需要用哪一个模型来处理，以及处理完后需要跳回到哪一个视图。Controller 对客户端的请求数据进行封装（如将 form 表单发来的若干个表单字段值，封装到一个实体对象中），然后调用某一个模型来处理此请求，最后再转发请求（或重定向）到视图（或另一个控制器）。



- > common
- > config
- > constant
- > **controller**
- > dao
- > domain
- > service

- 我们依照上述原则，在各层、各模块定义了 controller、service、dao 等类，可以将他们自上而下地组合在一起，我们在系统中尽量多使用组合和聚合关联关系，从而协作完成后端的某种特定业务功能，这就是这就是合成复用原则在本项目中的体现。
- 模型持有所有的数据、状态和程序逻辑。模型接受视图数据的请求，并返回最终的处理结果。

```
@ResponseBody
@RequestMapping(value = "/collection/add", method = RequestMethod.POST)
public Object addCollection(HttpServletRequest req) {
    String user_id = req.getParameter("userId");
    String type = req.getParameter("type");
    String song_id = req.getParameter("songId");
    String song_list_id = req.getParameter("songListId");

    if (collectService.existSongId(Integer.parseInt(user_id), Integer.parseInt(song_id))
        return new WarningMessage(message: "已收藏").getMessage();
    }

    Collect collect = new Collect();
    collect.setUserId(Integer.parseInt(user_id));
    collect.setType(new Byte(type));
    if (new Byte(type) == 0) {
        collect.setSongId(Integer.parseInt(song_id));
    } else if (new Byte(type) == 1) {
        collect.setSongListId(Integer.parseInt(song_list_id));
    }
}
```

- 在具体实现中，以增加歌单的前后端接口/collection/add 为例，controller 调用 service 层的对象，如图。

```
@Override
public boolean addCollection(Collect collect) {
    return collectMapper.insertSelective(collect) > 0 ? true : false;
}
```

- 然后，service 层的对象再调用 dao 层的对象，从而实现这一功能。
- 显然，在这种设计原则下，controller 对象并没有必要和 dao 对象进行直接通信，因此他们之间也并不存在直接调用关系，从而降低了彼此的耦合度。
- 根据迪米特原则，我们可以采用 service 充当中间人，把 dao 和 controller 对象之间的通信连接起来。