那些年，我们见过的 Java 服务端乱象

原创：陈昌毅 阿里巴巴中间件 8月8日

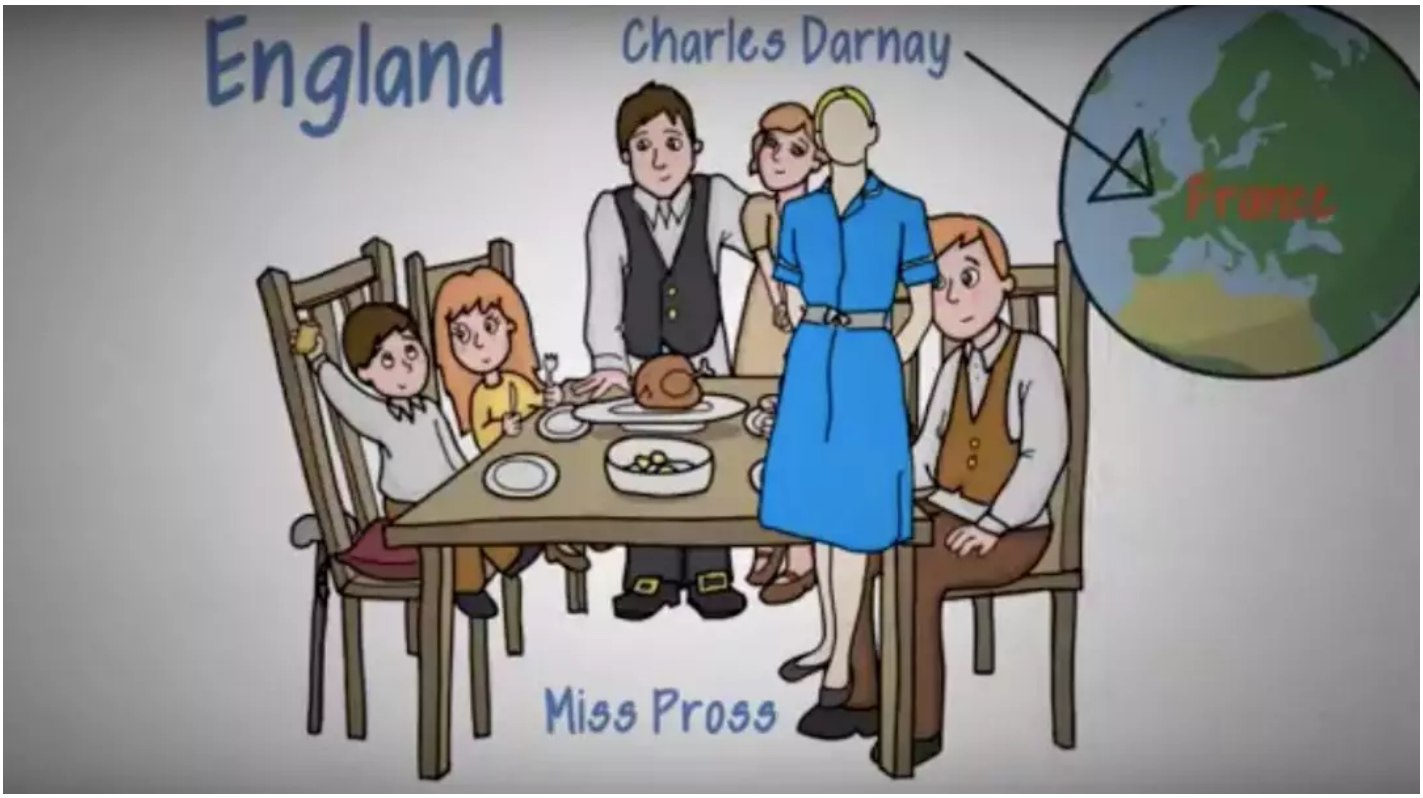


Photo by The Book Tutor @Youtube

文 | 陈昌毅

导读

查尔斯·狄更斯在《双城记》中写道：“这是一个最好的时代，也是一个最坏的时代。”

移动互联网的快速发展，出现了许多新机遇，很多创业者伺机而动；随着行业竞争加剧，互联网红利逐渐消失，很多创业公司九死一生。笔者在初创公司摸爬滚打数年，接触了各式各样的 Java 微服务架构，从中获得了一些优秀的理念，但也发现了一些不合理的现象。现在，笔者总结了一些创业公司存在的 Java 服务端乱象，并尝试性地给出了一些不成熟的建议。

1.使用Controller基类和Service基类

1.1.现象描述

1.1.1.Controller 基类

常见的 Controller 基类如下：

```
/** 基础控制器类 */
public class BaseController {
    /** 注入服务相关 */
    /** 用户服务 */
    @Autowired
    protected UserService userService;
    ...

    /** 静态常量相关 */
    /** 手机号模式 */
    protected static final String PHONE_PATTERN = "/^[1]([3-9])[0-9]{9}$/";
    ...

    /** 静态函数相关 */
    /** 验证电话 */
    protected static vaildPhone(String phone) {...}
    ...
}
```

常见的 Controller 基类主要包含注入服务、静态常量和静态函数等，便于所有的Controller 继承它，并在函数中可以直接使用这些资源。

1.1.2. Service 基类

常见的 Service 基类如下：

```
/** 基础服务类 */
public class BaseService {
    /** 注入DAO相关 */
    /** 用户DAO */
    @Autowired
    protected UserDAO userDAO;
    ...

    /** 注入服务相关 */
    /** 短信服务 */
    @Autowired
    protected SmsService smsService;
    ...

    /** 注入参数相关 */
```

```
    /** 系统名称 */
    @Value("${example.systemName}")
    protected String systemName;
    ...

    /** 静态常量相关 */
    /** 超级用户标识 */
    protected static final long SUPPER_USER_ID = 0L;
    ...

    /** 服务函数相关 */
    /** 获取用户函数 */
    protected UserDO getUser(Long userId) {...}
    ...

    /** 静态函数相关 */
    /** 获取用户名称 */
    protected static String getUserName(UserDO user) {...}
    ...
}
```

常见的 Service 基类主要包括注入 DAO、注入服务、注入参数、静态常量、服务函数、静态函数等，便于所有的 Service 继承它，并在函数中可以直接使用这些资源。

1.2.论证基类必要性

首先，了解一下里氏替换原则：

> 里氏代换原则（Liskov Substitution Principle，简称LSP）：所有引用基类（父类）的地方必须能透明地使用其子类的对象。

其次，了解一下基类的优点：

- 子类拥有父类的所有方法和属性，从而减少了创建子类的工作量；
- 提高了代码的重用性，子类拥有父类的所有功能；
- 提高了代码的扩展性，子类可以添加自己的功能。

所以，我们可以得出以下结论：

- Controller 基类和 Service 基类在整个项目中并没有直接被使用，也就没有可使用其子类替换基类的场景，所以不满足里氏替换原则；

- Controller 基类和 Service 基类并没有抽象接口函数或虚函数，即所有继承基类的子类间没有相关共性，直接导致在项目中仍然使用的是子类；
- Controller 基类和 Service 基类只关注了重用性，即子类能够轻松使用基类的注入DAO、注入服务、注入参数、静态常量、服务函数、静态函数等资源。但是，忽略了这些资源的必要性，即这些资源并不是子类所必须的，反而给子类带来了加载时的性能损耗。

综上所述，Controller 基类和 Service 基类只是一个杂凑类，并不是一个真正意义上的基类，需要进行拆分。

1.3.拆分基类的方法

由于 Service 基类比 Controller 基类更典型，本文以 Service 基类举例说明如何来拆分“基类”。

1.3.1.把注入实例放入实现类

根据“使用即引入、无用则删除”原则，在需要使用的实现类中注入需要使用的DAO、服务和参数。

```
/** 用户服务类 */
@Service
public class UserService {
    /** 用户DAO */
    @Autowired
    private UserDAO userDAO;

    /** 短信服务 */
    @Autowired
    private SmsService smsService;

    /** 系统名称 */
    @Value("${example.systemName}")
    private String systemName;
    ...
}
```

1.3.2.把静态常量放入常量类

对于静态常量，可以把它们封装到对应的常量类中，在需要时直接使用即可。

```
/** 例子常量类 */
public class ExampleConstants {
    /** 超级用户标识 */
```

```
    public static final long SUPPER_USER_ID = 0L;
    ...
}
```

1.3.3.把服务函数放入服务类

对于服务函数，可以把它们封装到对应的服务类中。在别的服务类使用时，可以注入该服务类实例，然后通过实例调用服务函数。

```
/** 用户服务类 */
@Service
public class UserService {
    /** 获取用户函数 */
    public UserDO getUser(Long userId) {...}
    ...
}

/** 公司服务类 */
@Service
public class CompanyService {
    /** 用户服务 */
    @Autowired
    private UserService userService;

    /** 获取管理员 */
    public UserDO getManager(Long companyId) {
        CompanyDO company = ...;
        return userService.getUser(company.getManagerId());
    }
    ...
}
```

1.3.4.把静态函数放入工具类

对于静态函数，可以把它们封装到对应的工具类中，在需要时直接使用即可。

```
/** 用户辅助类 */
public class UserHelper {
    /** 获取用户名称 */
    public static String getUserName(UserDO user) {...}
    ...
}
```

2.把业务代码写在 Controller 中

2.1.现象描述

我们会经常会在 Controller 类中看到这样的代码：

```
/** 用户控制器类 */
@Controller
@RequestMapping("/user")
public class UserController {
    /** 用户DAO */
    @Autowired
    private UserDAO userDAO;

    /** 获取用户函数 */
    @ResponseBody
    @RequestMapping(path = "/getUser", method = RequestMethod.GET)
    public Result<UserVO> getUser(@RequestParam(name = "userId", required = true) Long
        // 获取用户信息
        UserDO userDO = userDAO.getUser(userId);
        if (Objects.isNull(userDO)) {
            return null;
        }

        // 拷贝并返回用户
        UserVO userVO = new UserVO();
        BeanUtils.copyProperties(userDO, userVO);
        return Result.success(userVO);
    }
    ...
}
```

编写人员给出的理由是：一个简单的接口函数，这么写也能满足需求，没有必要去封装成一个服务函数。

2.2.一个特殊的案例

案例代码如下：

```
/** 测试控制器类 */
@Controller
@RequestMapping("/test")
public class TestController {
```

```
    /** 系统名称 */
    @Value("${example.systemName}")
    private String systemName;

    /** 访问函数 */
    @RequestMapping(path = "/access", method = RequestMethod.GET)
    public String access() {
        return String.format("系统(%s)欢迎您访问！", systemName);
    }
}
```

访问结果如下：

```
curl http://localhost:8080/test/access
系统(null)欢迎您访问！
```

为什么参数systemName（系统名称）没有被注入值？《Spring Documentation》给出的解释是：

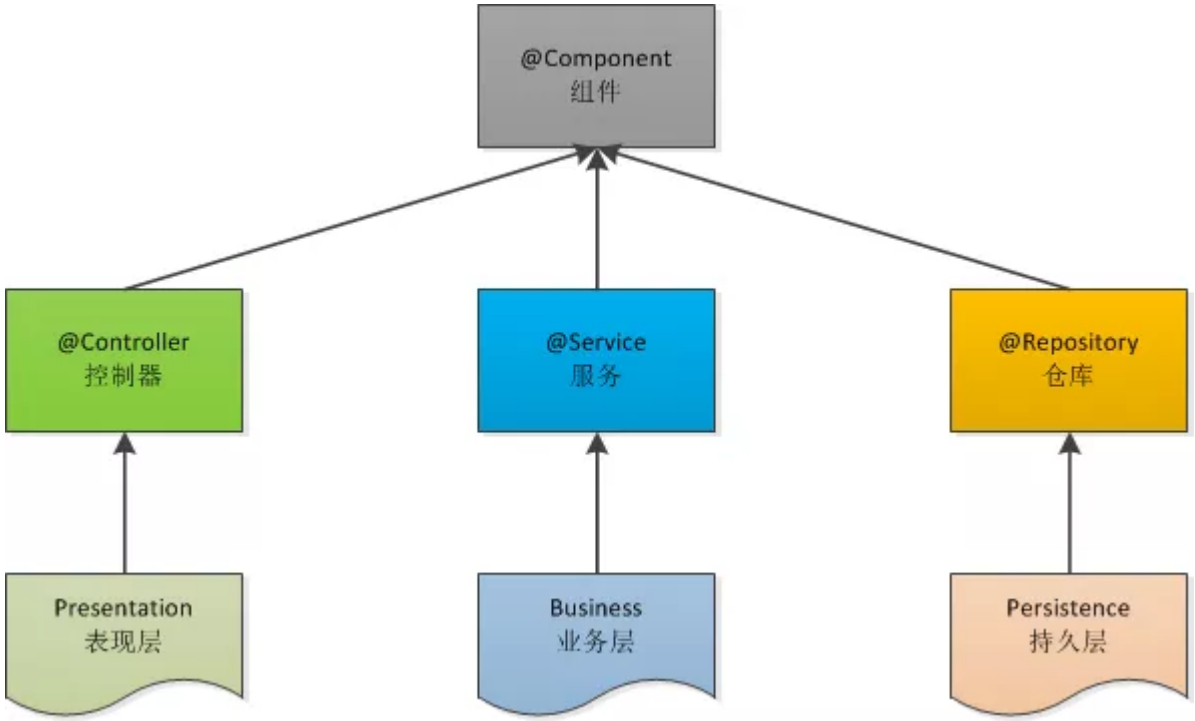
> Note that actual processing of the @Value annotation is performed by a BeanPostProcessor.
>
> BeanPostProcessor interfaces are scoped per-container. This is only relevant if you are using container hierarchies. If you define a BeanPostProcessor in one container, it will only do its work on the beans in that container. Beans that are defined in one container are not post-processed by a BeanPostProcessor in another container, even if both containers are part of the same hierarchy.

意思是说： @Value 是通过 BeanPostProcessor 来处理的，而 WebApplicationContex 和 ApplicationContext是单独处理的，所以WebApplicationContex 不能使用父容器的属性值。

所以，Controller 不满足 Service 的需求，不要把业务代码写在 Controller 类中。

2.3.服务端三层架构

SpringMVC 服务端采用经典的三层架构，即表现层、业务层、持久层，分别采用@Controller、@Service、@Repository进行类注解。



表现层（Presentation）：又称控制层（Controller），负责接收客户端请求，并向客户端响应结果，通常采用HTTP协议。

业务层（Business）：又称服务层（Service），负责业务相关逻辑处理，按照功能分为服务、作业等。

持久层（Persistence）：又称仓库层（Repository），负责数据的持久化，用于业务层访问缓存和数据库。

所以，把业务代码写入到Controller类中，是不符合SpringMVC服务端三层架构规范的。

3.把持久层代码写在 Service 中

把持久层代码写在 Service 中，从功能上来看并没有什么问题，这也是很多人欣然接受的原因。

3.1.引起以下主要问题

- 业务层和持久层混杂在一起，不符合SpringMVC服务端三层架构规范；

- 在业务逻辑中组装语句、主键等，增加了业务逻辑的复杂度；
- 在业务逻辑中直接使用第三方中间件，不便于第三方持久化中间件的替换；
- 同一对象的持久层代码分散在各个业务逻辑中，背离了面对对象的编程思想；
- 在写单元测试用例时，无法对持久层接口函数直接测试。

3.2.把数据库代码写在Service中

这里以数据库持久化中间件 Hibernate 的直接查询为例。

现象描述：

```
/** 用户服务类 */
@Service
public class UserService {
    /** 会话工厂 */
    @Autowired
    private SessionFactory sessionFactory;

    /** 根据工号获取用户函数 */
    public UserVO getUserByEmpId(String empId) {
        // 组装HQL语句
        String hql = "from t_user where emp_id = '" + empId + "'";

        // 执行数据库查询
        Query query = sessionFactory.getCurrentSession().createQuery(hql);
        List<UserDO> userList = query.list();
        if (CollectionUtils.isEmpty(userList)) {
            return null;
        }

        // 转化并返回用户
        UserVO userVO = new UserVO();
        BeanUtils.copyProperties(userList.get(0), userVO);
        return userVO;
    }
}
```

建议方案：

```
/** 用户DAO类 */
@Repository
public class UserDAO {
     /** 会话工厂 */
    @Autowired
```

```
    private SessionFactory sessionFactory;

    /** 根据工号获取用户函数 */
    public UserDO getUserByEmpId(String empId) {
        // 组装HQL语句
        String hql = "from t_user where emp_id = '" + empId + "'";

        // 执行数据库查询
        Query query = sessionFactory.getCurrentSession().createQuery(hql);
        List<UserDO> userList = query.list();
        if (CollectionUtils.isEmpty(userList)) {
            return null;
        }

        // 返回用户信息
        return userList.get(0);
    }
}

/** 用户服务类 */
@Service
public class UserService {
    /** 用户DAO */
    @Autowired
    private UserDAO userDAO;

    /** 根据工号获取用户函数 */
    public UserVO getUserByEmpId(String empId) {
        // 根据工号查询用户
        UserDO userDO = userDAO.getUserByEmpId(empId);
        if (Objects.isNull(userDO)) {
            return null;
        }

        // 转化并返回用户
        UserVO userVO = new UserVO();
        BeanUtils.copyProperties(userDO, userVO);
        return userVO;
    }
}
```

关于插件：

阿里的 AliGenerator 是一款基于 MyBatis Generator 改造的 DAO 层代码自动生成工具。利用 AliGenerator 生成的代码，在执行复杂查询的时候，需要在业务代码中组装查询条件，使业务代码显得特别臃肿。

```
/** 用户服务类 */
@Service
public class UserService {
    /** 用户DAO */
    @Autowired
    private UserDAO userDAO;

    /** 获取用户函数 */
    public UserVO getUser(String companyId, String empId) {
        // 查询数据库
        UserParam userParam = new UserParam();
        userParam.createCriteria().andCompanyIdEqualTo(companyId)
            .andEmpIdEqualTo(empId)
            .andStatusEqualTo(UserStatus.ENABLE.getValue());
        List<UserDO> userList = userDAO.selectByParam(userParam);
        if (CollectionUtils.isEmpty(userList)) {
            return null;
        }

        // 转化并返回用户
        UserVO userVO = new UserVO();
        BeanUtils.copyProperties(userList.get(0), userVO);
        return userVO;
    }
}
```

个人不喜欢用 DAO 层代码生成插件，更喜欢用原汁原味的 MyBatis XML 映射，主要原因如下：

- 会在项目中导入一些不符合规范的代码；
- 只需要进行一个简单查询，也需要导入一整套复杂代码；
- 进行复杂查询时，拼装条件的代码复杂且不直观，不如在XML中直接编写SQL语句；
- 变更表格后需要重新生成代码并进行覆盖，可能会不小心删除自定义函数。

当然，既然选择了使用 DAO 层代码生成插件，在享受便利的同时也应该接受插件的缺点。

3.3.把 Redis 代码写在 Service 中

现象描述：

```
/** 用户服务类 */
@Service
public class UserService {
    /** 用户DAO */
    @Autowired
    private UserDAO userDAO;
```

```
    /** Redis模板 */
    @Autowired
    private RedisTemplate<String, String> redisTemplate;
    /** 用户主键模式 */
    private static final String USER_KEY_PATTERN = "hash::user::%s";

    /** 保存用户函数 */
    public void saveUser(UserVO user) {
        // 转化用户信息
        UserDO userDO = transUser(user);

        // 保存Redis用户
        String userKey = MessageFormat.format(USER_KEY_PATTERN, userDO.getId());
        Map<String, String> fieldMap = new HashMap<>(8);
        fieldMap.put(UserDO.CONST_NAME, user.getName());
        fieldMap.put(UserDO.CONST_SEX, String.valueOf(user.getSex()));
        fieldMap.put(UserDO.CONST_AGE, String.valueOf(user.getAge()));
        redisTemplate.opsForHash().putAll(userKey, fieldMap);

        // 保存数据库用户
        userDAO.save(userDO);
    }
}
```

建议方案：

```
/** 用户Redis类 */
@Repository
public class UserRedis {
    /** Redis模板 */
    @Autowired
    private RedisTemplate<String, String> redisTemplate;
    /** 主键模式 */
    private static final String KEY_PATTERN = "hash::user::%s";

    /** 保存用户函数 */
    public UserDO save(UserDO user) {
        String key = MessageFormat.format(KEY_PATTERN, userDO.getId());
        Map<String, String> fieldMap = new HashMap<>(8);
        fieldMap.put(UserDO.CONST_NAME, user.getName());
        fieldMap.put(UserDO.CONST_SEX, String.valueOf(user.getSex()));
        fieldMap.put(UserDO.CONST_AGE, String.valueOf(user.getAge()));
        redisTemplate.opsForHash().putAll(key, fieldMap);
    }
}

/** 用户服务类 */
@Service
```

```
public class UserService {
    /** 用户DAO */
    @Autowired
    private UserDAO userDAO;
    /** 用户Redis */
    @Autowired
    private UserRedis userRedis;

    /** 保存用户函数 */
    public void saveUser(UserVO user) {
        // 转化用户信息
        UserDO userDO = transUser(user);

        // 保存Redis用户
        userRedis.save(userDO);

        // 保存数据库用户
        userDAO.save(userDO);
    }
}
```

把一个 Redis 对象相关操作接口封装为一个 DAO 类，符合面对对象的编程思想，也符合 SpringMVC 服务端三层架构规范，更便于代码的管理和维护。

4.把数据库模型类暴露给接口

4.1.现象描述

```
/** 用户DAO类 */
@Repository
public class UserDAO {
    /** 获取用户函数 */
    public UserDO getUser(Long userId) {...}
}

/** 用户服务类 */
@Service
public class UserService {
    /** 用户DAO */
    @Autowired
    private UserDAO userDAO;

    /** 获取用户函数 */
```

```
    public UserDO getUser(Long userId) {
        return userDAO.getUser(userId);
    }
}

/** 用户控制器类 */
@Controller
@RequestMapping("/user")
public class UserController {
    /** 用户服务 */
    @Autowired
    private UserService userService;

    /** 获取用户函数 */
    @RequestMapping(path = "/getUser", method = RequestMethod.GET)
    public Result<UserDO> getUser(@RequestParam(name = "userId", required = true) Long
        UserDO user = userService.getUser(userId);
        return Result.success(user);
    }
}
```

上面的代码，看上去是满足 SpringMVC 服务端三层架构的，唯一的问题就是把数据库模型类 UserDO 直接暴露给了外部接口。

4.2.存在问题及解决方案

存在问题：

- 间接暴露数据库表格设计，给竞争对手竞品分析带来方便；
- 如果数据库查询不做字段限制，会导致接口数据庞大，浪费用户的宝贵流量；
- 如果数据库查询不做字段限制，容易把敏感字段暴露给接口，导致出现数据的安全问题；
- 如果数据库模型类不能满足接口需求，需要在数据库模型类中添加别的字段，导致数据库模型类跟数据库字段不匹配问题；
- 如果没有维护好接口文档，通过阅读代码是无法分辨出数据库模型类中哪些字段是接口使用的，导致代码的可维护性变差。

解决方案：

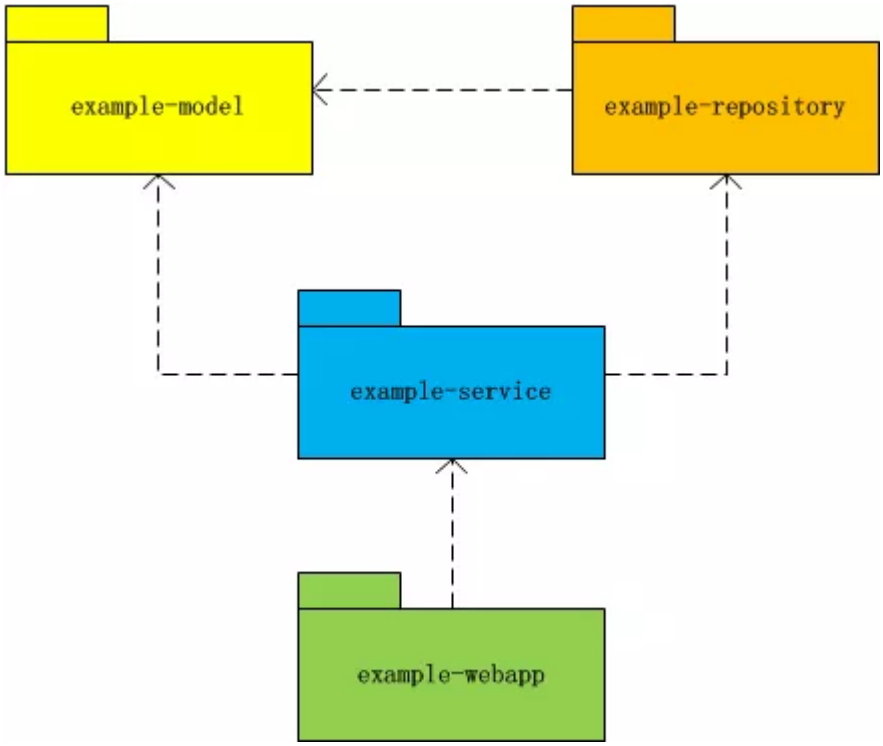
- 从管理制度上要求数据库和接口的模型类完全独立；
- 从项目结构上限制开发人员把数据库模型类暴露给接口。

4.3.项目搭建的三种方式

下面，将介绍如何更科学地搭建 Java 项目，有效地限制开发人员把数据库模型类暴露给接口。

第1种：共用模型的项目搭建

共用模型的项目搭建，把所有模型类放在一个模型项目（example-model）中，其它项目（example-repository、example-service、example-website）都依赖该模型项目，关系图如下：

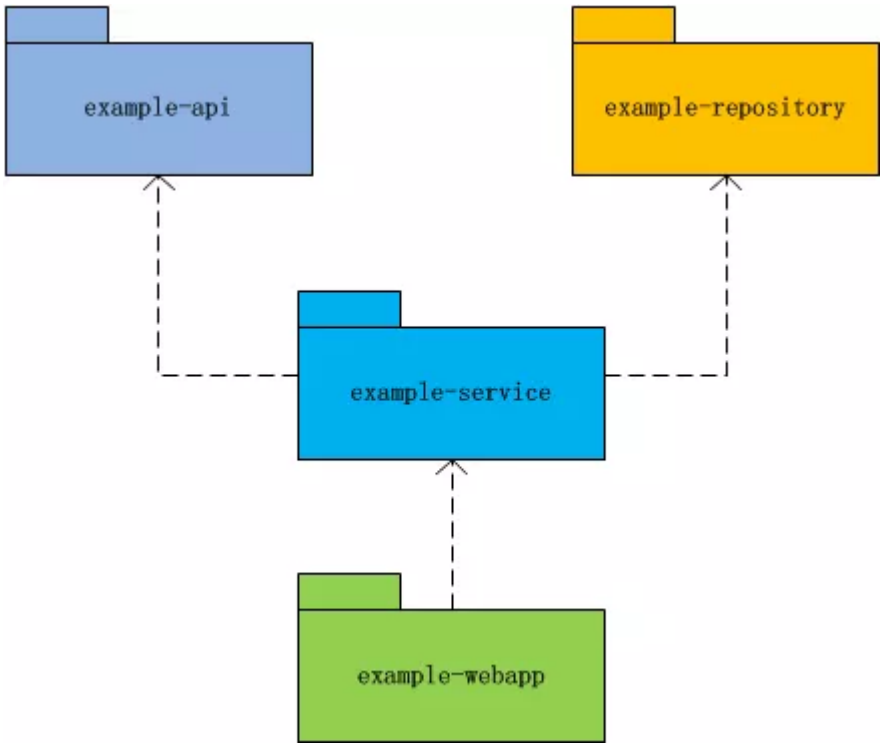


序号	项目名称	打包类型	项目功能
1	example-model	jar	定义了所有模型类，包括DO类和VO类等
2	example-repository	jar	对应持久层，实现了MySQL、Redis相关DAO等
3	example-service	jar	对应业务层，实现了Service、Job、Workflow等
4	example-webapp	war	对应表现层，实现了Controller、Interceptor、Filter等

风险：表现层项目（example-webapp）可以调用业务层项目（example-service）中的任意服务函数，甚至于越过业务层直接调用持久层项目（example-repository）的DAO函数。

第2种：模型分离的项目搭建

模型分离的项目搭建，单独搭建API项目（example-api），抽象出对外接口及其模型VO类。业务层项目（example-service）实现了这些接口，并向表现层项目（example-webapp）提供服务。表现层项目（example-webapp）只调用API项目（example-api）定义的服务接口。

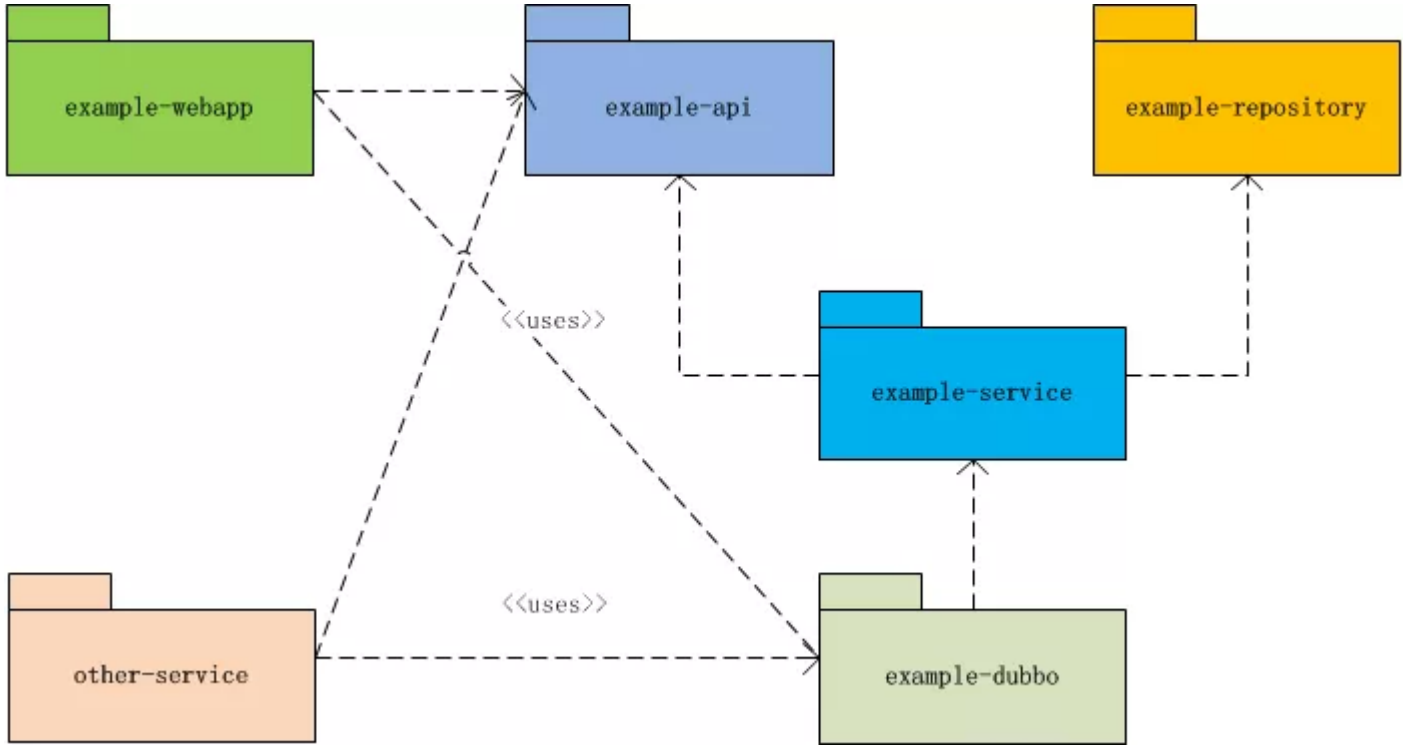


序号	项目名称	打包类型	项目功能
1	example-api	jar	业务层的表现层，定义了对外开放接口和VO类
2	example-repository	jar	对应持久层，定义了DO类并实现了MySQL、Redis相关DAO等
3	example-service	jar	对应业务层，实现了Service、Job、Workflow等
4	example-webapp	war	对应表现层，实现了Controller、Interceptor、Filter等

风险：表现层项目（example-webapp）仍然可以调用业务层项目（example-service）提供的内部服务函数和持久层项目（example-repository）的DAO函数。为了避免这种情况，只好管理制度上要求表现层项目（example-webapp）只能调用API项目（example-api）定义的服务接口函数。

第3种：服务化的项目搭建

服务化的项目搭，就是把业务层项目（example-service）和持久层项目（example-repository）通过 Dubbo 项目（example-dubbo）打包成一个服务，向业务层项目（example-webapp）或其它业务项目（other-service）提供API项目（example-api）中定义的接口函数。



序号	项目名称	打包类型	项目功能
1	example-api	jar	对应业务层的表现层，定义了对外开放接口和VO类
2	example-repository	jar	对应持久层，定义了DO类并实现了MySQL、Redis相关DAO等
3	example-service	jar	对应业务层，实现了Service、Job、Workflow等
4	example-dubbo	war	对应业务层的表现层，通过Dubbo提供服务
5	example-webapp	war	对应表现层，实现了Controller等，通过Dubbo调用服务
6	other-service	jar	对应其它项目的业务层，通过Dubbo调用服务

说明：Dubbo 项目（example-dubbo）只发布 API 项目（example-api）中定义的服务接口，保证了数据库模型无法暴露。业务层项目（example-webapp）或其它业务项目（other-service）只依赖了 API 项目（example-api），只能调用该项目中定义的服务接口。

4.4.一条不太建议的建议

有人会问：接口模型和持久层模型分离，接口定义了一个查询数据模型VO类，持久层也需要定义一个查询数据模型DO类；接口定义了一个返回数据模型VO类，持久层也需要定义一个返回数据模型DO类……这样，对于项目早期快速迭代开发非常不利。能不能只让接口不暴露持久层数据模型，而能够让持久层使用接口的数据模型？

如果从SpringMVC服务端三层架构来说，这是不允许的，因为它会影响三层架构的独立性。但是，如果从快速迭代开发来说，这是允许的，因为它并不会暴露数据库模型类。所以，这是一条不太建议的建议。

```
/** 用户DAO类 */
@Repository
public class UserDAO {
    /** 统计用户函数 */
    public Long countByParameter(QueryUserParameterVO parameter) {...}
    /** 查询用户函数 */
    public List<UserVO> queryByParameter(QueryUserParameterVO parameter) {...}
}

/** 用户服务类 */
@Service
public class UserService {
    /** 用户DAO */
    @Autowired
    private UserDAO userDAO;

    /** 查询用户函数 */
    public PageData<UserVO> queryUser(QueryUserParameterVO parameter) {
        Long totalCount = userDAO.countByParameter(parameter);
        List<UserVO> userList = null;
        if (Objects.nonNull(totalCount) && totalCount.compareTo(0L) > 0) {
            userList = userDAO.queryByParameter(parameter);
        }
        return new PageData<>(totalCount, userList);
    }
}

/** 用户控制器类 */
@Controller
@RequestMapping("/user")
public class UserController {
```

```
    /** 用户服务 */
    @Autowired
    private UserService userService;

    /** 查询用户函数(parameter中包括分页参数startIndex和pageSize) */
    @RequestMapping(path = "/queryUser", method = RequestMethod.POST)
    public Result<PageData<UserVO>> queryUser(@Valid @RequestBody QueryUserParameterVO
        PageData<UserVO> pageData = userService.queryUser(parameter);
        return Result.success(pageData);
    }
}
```

后记

“仁者见仁、智者见智”，每个人都有自己的想法，而文章的内容也只是我的一家之言。

谨以此文献给那些我工作过的创业公司，是您们曾经放手让我去整改乱象，让我从中受益颇深并得以技术成长。

本文作者：

陈昌毅，花名常意，高德地图技术专家，2018年加入阿里巴巴，一直从事地图数据采集的相关工作。

本文缩略图：icon by 是一只啊Y

/ 点击下方图片，报名参加 /

Tips：

点下“在看” 💔

然后，公众号对话框内发送“**U你不困**”，试试手气？ 😆

本期奖品是来自淘宝心选的天然乳胶颗粒 U 型枕。