告别面条代码，让代码一开始就简洁

> 编者按：本文作者是支付宝后端开发工程师言戒，提供了关于简洁代码的一些参考思路，欢迎查阅～

一、背景

在大部分业务领域的开发来说，主体代码基本分为【业务向】【数据向】，两者都是重业务逻辑的业务代码。这类代码通常复杂度来自业务，且变化比较频繁，如果没有良好的习惯和编码设计，很容易臃肿不堪，被人形容成“面条”。

如何写得易读，易懂、易于修改，即简洁代码。

本文提供一些简单的参考思路。

1.1 简洁的定义

极端的说，简洁的代码可以满足“我的代码一行都不能删”，例如教材版的HelloWorld。

然而现代工业化软件设计中，业务复杂，分工协作，变更/维护频繁且长期，势必要有结构分层，依赖/耦合关系抽象和再次组织。

那么在这样的客观环境下，简洁结构追求的是什么呢?

答：分层

1.满足业务需求

2.尽可能使逻辑简单，表述易于理解

3.尽可能最小化模块，易于变更维护

4.尽可能简化配置项，减少操作成本

5.尽可能没有相似/重复代码

6.尽可能用程序逻辑去替代人脑逻辑

1.2 简洁代码的类比

简洁代码的价值显而易见：【易读，易测，易修，易改】

应该时刻以简洁代码为目标，但也需要根据现实情况和操作成本以及能力，量力而为。

Q1：简洁代码Vs简单代码

A1：简洁代码是指逻辑、结构、表述条理清晰简洁，并不等同于代码简单，相反奖复杂代码写的简洁存在挑战，要有良好的设计。

简洁代码良好的结构和模块化拆分，正是为了更好的修改和扩展准备的，越合理的单元化，增量的功能涉及的影响面也越小，就越容易修改；即便不修改，也便于调试和测试；出了bug也很容易排查和改正。

Q2：简洁代码Vs低代码

A2：简洁代码并不一定是代码行数最少。

只论行数，所有逻辑写在一个方法内，总行数最少；但这样并不简洁，反而是臃肿。

简洁代码会因为合理的分层，实际在代码行数上有轻度增量，典型的增量部分

如：出入参模型、转化、模块内聚的校验等。

Q3：简洁代码Vs设计模式

A3：如果合理的使用设计模式，本质上并不会与简洁代码冲突。

设计模式的目的也是为了让代码可维护，可扩展。

但对设计模式的使用要充分理解，不能因为似是而非的理解胡乱选型，反而增加了复杂度；

有一些设计模式本身有一定复杂性，这部分仁者见仁，看coeder自身的水平。

总结来说，合理的设计模式，正是一种简洁代码。

二、分层框架
==========

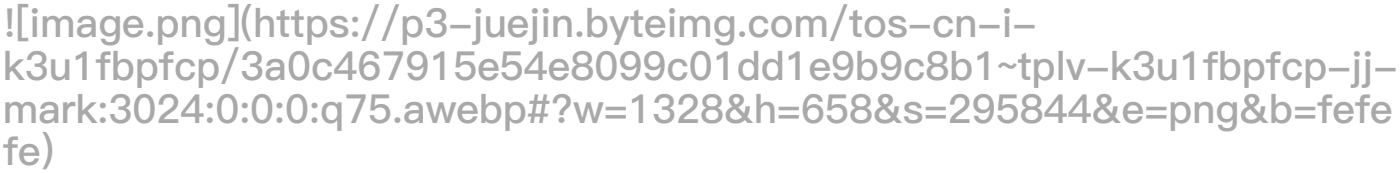
2.1 类结构分层

每个团队都有各自的分层，以我所在团队当前系统常见分层来举例。

2.1.1 标准分层

RPC/TR/Handler->service->manager->dao

对应的模型分别是Info、Model、Config/VO、DO



2.1.2 基本原则

代码分层的目的是为了将抽象分等级，是一种抽象方式的约定，模型对象的命名给出了约定的标准；

标准分层是一种基本思路，归根到底是抽象结构的定义，实操层面允许一定灵活性，适当的进行代码结构调整，以减少层次，减少模型，减少convertor等。

但是有一些原则：

1、DO对象不可替代，不可复用其他层次对象，必须单独一个模型。 DO对象一般不能直接出现在业务层，绝对不能出现在最上层。

2、 4层模型不是固定死的

如果两层模型结构非常接近，可以酌情合并或者跨级引用，减少一次模型定义和模型转化。

模型合并优先级：从上至下。（原因：a.上层模型通常更必要，比如rpc模型前端必然需要；b. 发生模型少量参差，通常上层模型可以兼容下层模型，即通常上层模型要更具象，字段要更多，下层模型更抽象，字段更少）

3、4层结构不是固定死的

Service和Manager都是业务层抽象，Manager粒度更细，Service更复合。

对于某些简单业务，可以合并成一层（个人偏好&建议，合并到Manager，这样如果后续有业务扩展增量写Service是纯新增逻辑，而不用动Manager；反之，就得从Service里extract，影响面变大）

4、Repository层的时机。

简单的单表操作，普通DAO层基本足够，但是在有些场景，操作数据/资源，有较为复杂的逻辑，比如批量读写，多路sql等，这时候可以用Repository来2次封装。但是要避免出现Repository里只做DAO的调用转发（因为这样除了增加代码量外没有任何意义）。

三、案例实战
==========

3.1 分层结构

3.1.1 分层不均

案例

```
    public MobileCreditIndexResult queryIndex(MobileCreditIndexRequest
request) {

        //【核心片段】
        MobileCreditResult queryRes =
mobileCreditManager.queryIndex(userInfo, wirelessCarrier, apdId, mobile,
request.getChargeList(), now);

        // ...省去部分片段对象初始化
    }
```

```

    @Override
    public MobileCreditResult queryIndex(UserInfo userInfo, String
wirelessCarrier, String apdId, String mobile,
                                         List<String> chargeList, Date now) {
        String userId = userInfo.getUserId();
        // 1.校验活动降级和活动时间
        activityStatusCheck(userId, now);

        // ...省去部分片段，对象初始化

        // 2.处理供应商金额信息
        SelectMoneyModel originSelectMoney =
resolveTargetPriceOfWirelessCarrier(userId, wirelessCarrier, chargeList,
            wirelessCarrierConfigByUser, activityConfigByUser);
```

```
        // 3.查询用户发奖事件信息
        List<BillFlowDO> sendPrizeBillFlows =
queryUserSendPrizeBillFlows(userId);
        List<UserEventFlowDO> userEventFlowDOS =
queryUserSendPrizeEventFlows(userId);
        // ...省去部分片段

        // 4.获取用户当前最近一次获得的券信息
        BillFlowDO recentBillFlow = resolveRecentBillFlow(userId,
sendPrizeBillFlows);
        // ...省去部分片段

        // 5.查询券的使用记录
        List<VoucherTransferDetail> directVoucherUseDetails =
queryVoucherUserDetail(userId, voucher);
        // ...省去部分片段

        // 6.校验用户是否有券，有则渲染
        MobileCreditResult prizeRes =
resolveAvailableVoucherResult(userInfo, mobile, voucher,
directVoucherUseDetails,
            prizeConfigByUser, originSelectMoney, activityConfigByUser,
now, bindCardFlag, autoChargeFlag);
        // ...省去部分片段

        // 7.准入规则校验
        checkUserCrowd(userId);

        //8.奖品决策
        //if 为中奖
        if (StringUtils.isBlank(userInfo.getBindedMobile()) ||
!userInfo.isCertified()) {
            // ...省去部分片段
        } else {
            // ...省去部分片段
            }
        }

        // 9.为空需要复原结果
        if (prizeRes == null) {
            // ...省去部分片段
        }

        return prizeRes;
    }

```

点评：

1.业务抽象直接等同成了rpc接口抽象，而所有逻辑都在业务层里面实现，导致rpc层单薄（核心代码只有一行方法调用），而业务层臃肿（原始方法接近200行）

2.接口层做的逻辑太多，没有任何逻辑拆分，只是单纯的运算罗列堆积，导致臃肿

3.原本应该在上层做的通用逻辑下放到了业务层：比如活动降级校验、配置校验。进一步增加了业务层的臃肿

建议：

1.rpc抽象：首页接口，业务层抽象多个子业务：【面额处理】、【券周期处理】、【发券处理】

2.子业务模块在首页接口里组装衔接

3.跟子业务本身无关的辅助逻辑、或者通用逻辑（比如活动校验、全上下文要用的配置校验）在rpc层编写

3.1.2 分层冗余/无效

案例（代码略）

1.Service内所有代码仅简单调用Manager层；

2.Repository内所有代码仅简单调用DAO层；

建议：

如果只是简单转发，按需合并或简化层次，如合并掉Manager/Repository

3.2 内聚 & 耦合

3.2.1 传参冗长

```
/**
     * 获取准入的奖品规则
     *
     * @param apdId
     * @param bizDate
     * @param userId
     * @param prizeConfigByUser
     * @param ruleConfigByUser
     * @param prizeConfigMap
     * @param userEventFlowDOS
     * @param recentBenefitFlow
     * @param recentEventFlow
     * @param voucher
     * @param directVoucherUseDetails
     * @param autoChargeFlag
     * @return
     */
    private AdmitConfig consultAdmitConfig(String apdId, Date bizDate,
String userId,
                                           List<PrizeConfig> prizeConfigByUser,
                                           List<RuleConfig> ruleConfigByUser,
Map<String, PrizeConfig> prizeConfigMap,
                                           List<UserEventFlowDO>
userEventFlowDOS, UserBenefitFlow recentBenefitFlow,
                                           UserEventFlowDO recentEventFlow,
Voucher voucher,
                                           List<VoucherTransferDetail>
directVoucherUseDetails,
                                           boolean autoChargeFlag) {
        // ...省去部分片段
         return admitConfig;
    }
```

点评：

案例中传参多达12个，同时包含map，list复杂结构
传参过多的时候问自己两个问题：

1、这些参数都是必要的吗？有简化可能吗

2、这些参数所运算的逻辑，都必须在这个方法内吗？

建议：

1.【内聚】传参都是必要的， 传参的处理逻辑也是必需在本方法内完成的。

2.【内聚】传参最小化。只用到useId就不要传UserInfo，只用某一个value，就不要传入整个集合。

3.【内聚】自给自足。能用本方法内通过工具类获得的（如Util.get(userId)），就不要在上文获取后当做入参传递

4.【重构】合理抽象依赖关系，简化方法内部逻辑，少做点事。

3.2.2 出参、入参的误用

代码案例

```
public Result process(Param param);

public void process(Param param, Result result);
```

以上两个方法执行逻辑一致。

区别在于方法2把result当做入参，返回值改成void。

这是出入参误用的典型特征，虽然语义能够执行，但是后续加上维护变更后

，容易导致：

1.书写代码栈深变长，代码变的冗长。

2.result的修改发生在方法中，读码容易忽略，修bug或者需求调整时，容易遗漏。

建议：

【入参】是程序计算因子，一般只读，不做变更。

【出参】是程序计算结果，一般只在赋值时候做变更。

1.合理抽象，明确输入输出

2.严格定义出入参（入参只读，出参只在赋值的时候写）

3.如果特殊场景，必须要传参修改，方法内部要标明，例如：fillConfig，processModel等

3.2.3 耦合关系扩散

```
/**
* 获取彩蛋状态
*/
public CatSportsEggModel getEggStatus(String userId, LocalDate bizDate, CatSportsConfig actConfig) {
        CatSportsEggModel model = new CatSportsEggModel();

        // ...省去部分片段

        // 彩蛋处于有效期 且 在发奖期 且 当日已领奖 且 非当日有效 且 非6.18
        else if (prizeIsEnd && bizDate.
                isBefore(TimeUtil.toLocalDate(actConfig.getActEndTime()))) {
model.setLotteryPhase(LotteryPhaseEnum.AWARD_PERIOD.getCode());
model.setLotteryStatus(LotteryStatusEnum.AWARDED_NOT_DAY_618.ge
```

```
tCode());
            // ...省去部分片段
        }
        // 彩蛋处于有效期 且 在发奖期 且 当日已领奖 且 非当日有效 且 6.18
        else if (prizeIsEnd && bizDate.plusDays(1).
                isAfter(TimeUtil.toLocalDate(actConfig.getActEndTime()))) {
model.setLotteryPhase(LotteryPhaseEnum.AWARD_PERIOD.getCode());
model.setLotteryStatus(LotteryStatusEnum.AWARDED_NOT_DAY.getCod
e());
            // ...省去部分片段
        }

        else if... // ...省去部分片段

        return model;
    }
```

点评：

原始需求“活动的最后一天，发奖状态特殊处理，前端展示【已开奖】，否则展示【明日再来】”。

代码案例中，直观的从prd翻译，自然的就耦合活动时间相关的判断了，导致if条件复杂冗长。

但挖掘需求条件，隐含了条件：

【活动最后一天】=【发奖最后一天】=【6.18当天】
根据需求只管理解是活动最后一天，但根本关联因素条是发奖时间。

因此耦合关系应该只依赖发奖时间配置，而不是活动时间配置。

判断条件即可简化成“是否发奖结束”，而这个值可以从配置参数中获取。

建议：

深度需求分析，找到真正的关联关系，而不是业务语义直接翻译，以此定义出逻辑、变量的耦合关系。

3.3 程序流分支

3.3.1 条件判断层次复杂

```
//这个taobaoRedPacketDowngrade判断字段

public void process() {
    if (!alipayEggDowngrade) {
        if (!taobaoRedPacketDowngrade ||
buildingQueryRequest.isForceToQuery()) {
            BuildingMainPageInfo mainPageInfo =
copartnerManager.queryBuildingMainPage(userId, securityInfo);
             //...省略部分片段
            if (mainPageInfo != null) {
                BuildingRedPacket buildingRedPacket =
mainPageInfo.getRedPacket();
                 //...省略部分片段
                if (buildingRedPacket != null) {
                    //...省略部分片段
                }
            }
        } else {
            //...省略部分片段
        }
    }

    if (!taobaoRedPacketDowngrade ||
buildingQueryRequest.isForceToQuery()) {
        //...省略部分片段
    }

    //...省略部分片段

    if (!alipayEggDowngrade && !taobaoRedPacketDowngrade) {
        //...省略部分片段
    }

    //...省略部分片段
}

```

点评

1、判断条件内含多层嵌套，分支链路复杂

2、同一个判断条件，在多条分支中重复判断，本案中taobaoRedPacketDowngrade条件，有3处判断，加大了阅读和维护难度，后续业务调整很容易修改遗漏
建议

1.分支的走向路径要一致，一个分支尽量把逻辑做完，避免重复开启分支导致的复杂度增加

2.简化逻辑，定义出子逻辑，提取成子方法

3.用程序逻辑思考，把逻辑理顺，而不是简单粗暴的对业务语义直接翻译

四、总结
========

定义（WHAT）

【功能正确】【结构简单】【条理清晰】

价值（WHY）：

【易读，易测，易修，易改】

方向（HOW）

合理抽象，合理分层

高内聚，松耦合

用程序逻辑去思考，而非需求翻译

适当的编写技巧

重构时机（WHEN）

单个方法太长时（建议不超过40行）

参数太多时（建议不超过4个）

单元测试非常难编写时（写一个单测，需要造非常多的前置数据）

配置变复杂时（改一个逻辑，需要改多个配置联动，或者要求配置里有逻辑限定）

“这段代码好像刚刚写过”的感受时（DRY原则）

后记

java中抽象、封装、继承、多态，哪一个是最关键的?

刚入行时我觉得是多态。因为多态的特性复杂，以此能够衍生出很多编程技巧。

工作经验越是往后积累，越能明白抽象的重要。

小到代码的参数抽象，方法抽象；中到业务的模型抽象，逻辑抽象；大到架构抽象，模式抽象。

抽象是最考验抓本质的本事，只有把抽象思路理清楚，才能在地基之上建高楼。

如何能把业务抽象的更简单，是写成简洁代码的根。

推荐学习
========

《代码大全》

《重构》

《代码整洁之道》

《编程珠玑》

《effective java》

《java8实战》

原文链接: https://juejin.cn/post/7379960023407804428