设计总结

# 系统需求

通过在项目实现或者设计过程中的一些用法总结

# 项目负责

# 项目前思考

# 项目前攻破

# 项目中使用的技术总结

# 项目后期思考

## 关于过度设计的总结

背景：在pms中需要接受第三方（支付宝）等付款完成后，然后去通知对应的订单后台管理系统和app后台服务器；

修改前的设计：



这里存在以下几个问题：

问题1:比如callbackService目前采用了3次回调的策略，那后续如果需要其他调用策略方式，需要重新定义一套CallbackService，覆盖重写callback方法，而且在callback方法需要使用PmsExecutor.execute(new Runnable(){

public void run(){

// 策略开始

if(job.run()){

// 跳出策略;

}

// 策略结束

job.process(策略执行结果)；

}

});

上面这一套模版是所有新建的CallbackService需要处理，那么代码根本就没有达到共用

问题2:CallbackJob中定义了模版方法，但是却要借助CallbackService来实现，这样模版方法就不具有一定有效性了。如果使用了其他CallbackService，这套方法又必须手动去执行，所以所用的策略方法失败

因此我想要将PmsExecutor.execute(…)等代码抽取，那么我就必须将callBackJob继承Runable，然后执行调用，因此对此设计我做了以下调整



## 关于接口设计的通用性



## 过度设计2：

如果一个方法，抽取了对另外一个类的封装，而且该方法并没有特殊处理，那么这样就是一种过度设计了，或者抽取的方法过于简单，那么也是过度设计的一种

或者一个方法抽取了代码不超过5行，我觉得也是一种过度封装

## 过度设计例子

### 例子1：过度类



修改前



修改后

修改后，采用了代理模式来消除了多余的接口，由于RemoteFileService及HttpFileService所做的事情其实是一致，抽成了两个类进行处理，显得冗余。原先的目的是想在原先已有的远程上传的基础上，增加一个本地上传的功能，无奈却多出了一个没必要的接口，可直接继承自原类，然后覆盖方法，也算代理的另一种体现吧。

### 例子2：违反单一职责



左侧的是未修改前的代码，可以看出SoftwareImportHttpOperate并没有起到效果，而只是封装了一个简单方法进行转发，若是这样，还不如直接放入SoftwareImportProxyService的类中，但是这样SoftwareImportProxyService的职责过分散，而SoftwareImportHttpOperate又过分简单了。

右侧中SoftwareImportHttpOperate的职责体现出来，主要负责对应的http请求了，而ProxyService也只取到了中间桥梁的作用

## 关于代理模式、外观模式、适配器模式理解

**代理模式：**代理与实际对象的目标是一样的，都是为了做成一件事情，但是由于实际对象可能不了解或者业务单一，这时候为了每个对象做成更多业务(日志业务)

主要区分于其它三个，就是代理对象可以完全替换目标对象，也不会造成程序错误，实际对象有的业务，代理对象必须等同拥有

**外观模式：**对于系统中，很多业务系统都互相有关系，例如股票与基金，如果散户直接与股票打交道，那么就必须深入了解股票的知识，而如果有了基金，那么为大家提供一个透明的股票投资机制。

区分于其它三个，就是外观模式就是在原有方法的基础上，组合封装出单个对象的调用，这样就不用关心具体的业务对象间的关系，只关心外层这个类的方法实现

**适配器模式：**对于系统A与系统B，这时候系统A已经发布并且无法再次修改，这时候系统B需要使用A的逻辑，但是需要A的接口数据又不大一样，这时候就可以采用适配器来适配出B的数据格式

**装饰模式：**通过一系列主键，来完成新事物的装饰出一套另一套东西，即通过不同主键实现，达到不同事物的效果。

**装饰模式与代理模式的不同点：**

装饰器相当于在类执行过程中能够被递归地被构造，而代理模式则是为了隐藏一个实际对象，在编译时一般就已经确认了代理的关系

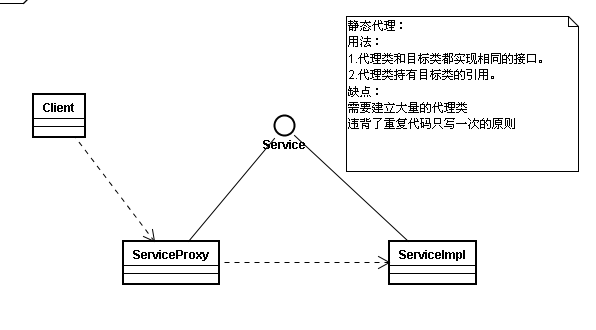
建造者模式：要求的是整个对象的建造过程是固定的，但是有不同的建造方式而已

1.装饰模式：对被装饰的对象增加额外的职责，满足里氏代换原则（子类可以代替父类）.如：杯子生产线，杯子必须可以装水，在生产线上可以给杯子涂颜色，加杯盖，但要保证杯子可以装水。

2.代理模式：对被代理的对象提供访问控制。如：客户订购商品，可以直接从厂家订购，也可以从网上商城订购，网上商城是厂家的代理，从网上商城订购商品，商城可以对商品进行控制，不交钱不给商品，人不在不给商品，也可以赠送……

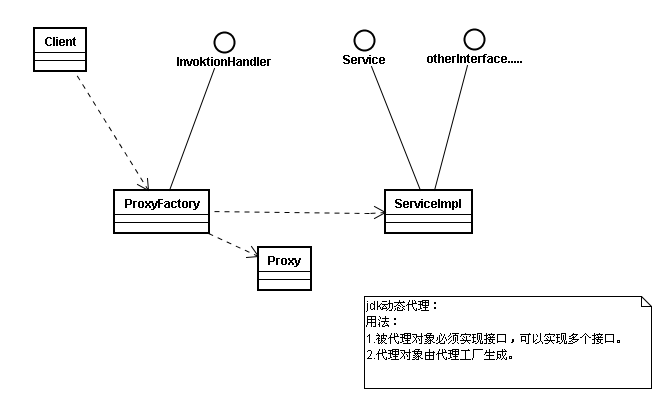
## 代理模式

### 静态代理



在编译时，已经决定了代理的对象

### 动态代理



### cglib代理

jdk给目标类提供动态要求目标类必须实现接口，当一个目标类不实现接口时，jdk是无法为其提供动态代理的。cglib 却能给这样的类提供动态代理。Spring在给某个类提供动态代理时会自动在jdk动态代理和cglib动态代理中动态的选择。

使用cglib为目标类提供动态代理：需要导入cglib.jar和asm.jar

## 关于Spring aop中的代理模式理解

### 静态代理

AspectJ，在编辑的时候，会直接将代码编译到原类中，然后生成的代码会与原类不一样，这样来说，就是一个代理类的方式

### 动态代理

jdk代理：

采用接口方式的编程，由jdk生成对应的代理类对象Proxy，然后由代理对象来完成加强处理+被代理对象的处理

cglib代理：

由代理工厂类，生成对应的代理对象EnhancerByCGLIB，由此类来完成加强处理+被代理对象的处理

## 关于数据库字段的设计

针对了支付渠道的表，由于微信、支付宝等支付方式所带的支付信息有不一致，所以如果采用了多字段存储，就达不到共用效果，因此采用了json格式的信息来存储该地段的信息

## 关于接口设计中参数命名的总结

这次在cvms中，接口设计中，定义了视频信息获取，其中用total\_count表示总集数和current\_count表示当前更新集数，但是从意义上来讲total即有总数的意思，再加上count有点画蛇添足。

1. 言简意赅
2. 参数名最好能够直接读出其含义

## 关于在使用第三方api时，对于参数的认知

这里写一下这个总结，是为了给自己脑子记上这一课；在调用微信支付的时候，有需要传递一个billcreate\_ip，开发者文档上面说，该字段是用于服务器创建微信订单的ip，当时也没有加以研究，直接就配上本地的ip。

1. 能够根据参数传递，成功创建
2. 对于参数的值，应该深入研究一下，对于不同的值，是否有不同限制
3. 如果参数的值都没问题，能够反问自己，如果是你来开发这个接口，你传递这个参数意义何在？为何传递该参数

## 关于数据库表设计的要点

对于表名的设计，习惯上会以\_info作为结尾，但是其实看起来较累赘，我觉得数据库的设计应该与接口设计一样，能用一个单词出来就用一个单词

## 对于工具类的封装

跟业务无关性。如果市面上已有，也不能封装。可复用

## 关于接口实现

关于接口实现上，有时候需要改掉旧接口，可是该接口可能已经被线上的其他版本给占用，这时候如果修改了，需要考虑到兼容，并做到兼容性测试。

## 关于之前在亿联用了IXXXService和XXXServiceImpl

虽然说这种方式是为了更好的扩展，但是在敏捷开发或者业务开发中，此种代码属于冗余，反而不容易进行开发。因为除了一个实现外，根本就没有其余多种实现。如果是针对组件式开发，就比较必要。因为要考虑更多的兼容。所以每一次开发并不能总是做得很好。如果不是一开始就有需求的话，代码还是越简洁越好

## MVC&MVP&MVVM

### MVC



mvc架构

1. view中的js及ftl通过请求，到controller处理或者获取数据
2. controller再到model中请求处理，根据结果返回数据
3. view获取内容，通过entity的属性来填充内容。所以view直接与model有交互
4. 由于web的controller无法控制view的展示，是由view直接控制view，所以也成为被动式mvc。

### MVP



mvp

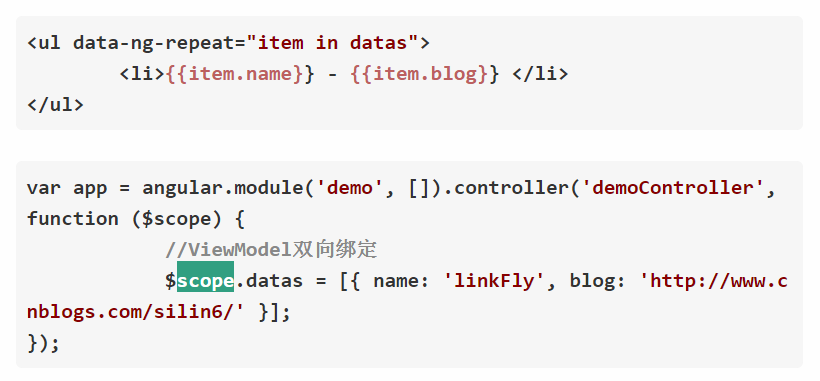
1. view将请求提交至presenter(类似于c)
2. presenter调用不同的model进行处理。
3. presenter对不同的view进行更新

### MVVM



mvvm

mvvm很像mvp，只不过这次不是由viewModel直接控制view，而是与view之间通过某个绑定来达到解耦过程。



### 对比：

|  |  |  |
| --- | --- | --- |
| 模式 | 优点 | 缺点 |
| MVC | 1.隔离业务逻辑和ui展示逻辑 | 1.view依赖于model，无法将view组件化  2. Controller测试困难。因为视图同步操作是由View自己执行，而View只能在有UI的环境下运行。在没有UI环境下对Controller进行单元测试的时候，应用逻辑正确性是无法验证的：Model更新的时候，无法对View的更新操作进行断言(无法理解) |
| MVP | 1.将view与model彻底隔开，view做到与业务无关，可组件化  2.便于测试 | 1.如果一个view页面复杂，会导致presenter越来越复杂，难于维护 |
| MVVM | 1.将presenter的工作简化,易于维护  2.便于测试 | 1.文件量更多，并且简单的业务可能复杂化 |

## 对于屏幕资源下的导入与导出

针对局点之间，布局的快速搬迁，将屏幕资源导出，并在另一方导入

### 设计

将屏幕元素关联的图片地址修改成绝对地址，然后在导入的时候，去下载文件

#### 优点：

逻辑简单，无需将文件下载打包，后面再匹配

#### 缺点：

在导入的局点中，如果与原先的局点不是同一网段，可能导致下载失败。

### 更好的设计：

解析屏幕元素关联图片，并下载，后打包上传

#### 优点：

将不同局点之间，可能的网络元素引起的问题，给避免了。脱离了联系

## 数据的一致性，行为的多态性

之前针对软件分类和子类的级联选择的设计：原先以为将整个select都一起抽取，但是由于select的id和名称属性可能会不一样，因此可以看出数据并没有多少一致性，还不如各自书写。而将改变大类的时候，级联刷新子类的行为作为该一致行为进行抽取。

## 动态分页

平常都是在find list的时候，会去count数据的条数，然后再分页。如果数据一多的话，count就会造成时间较长

解决方法：取pageSize+1条记录，如果获取的条数>=pageSize，则表示有下一页，然后再二次请求刷新页码

碰到问题：场景(由于当前列表已去请求页码，但是这时候又切换了条件。但是后来的页码先出来，结果前面请求的页码又出来，把之前的页码刷到第二次的请求了)

处理方式：通过后台去修改逻辑，而且又做了一个错误的方案。当没有记录的时候，还需要去二次请求页码。这实际是不合理的

更好的方法：应该将请求断绝到源头；就是如果当前又刷新列表，应该把旧的替换页码的操作给取消

## 对于封装力度

在使用了bootstrap的modal中，当主页上出现了滚动条，这时候modal弹出第一层，会将滚动条去掉，接着再弹出第二层。然后关掉第二层，就会再出现滚动条

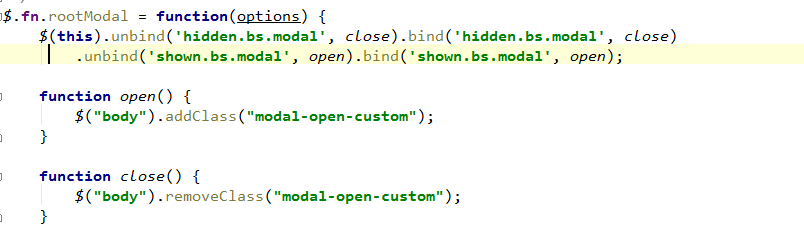
问题原因：

由于modal的时候，会在body添加了一个class，modal-open，这个是将over-flow变成hidden。再关闭的时候，再将该class删除。

解决方法：

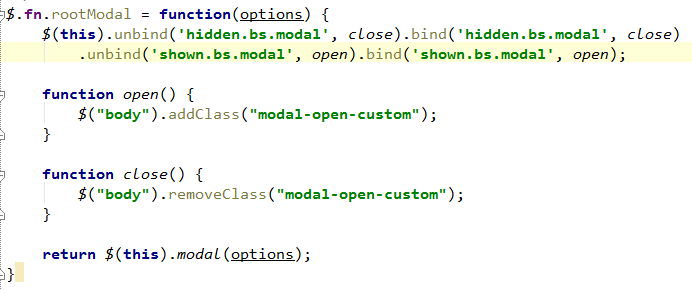
添加了一个新的class，只有在第一级弹窗的时候，才会新增，然后在第一级弹窗关闭后，才会删除。

抽取1：



在调用的时候$(…).rootModal().modal();

抽取2：



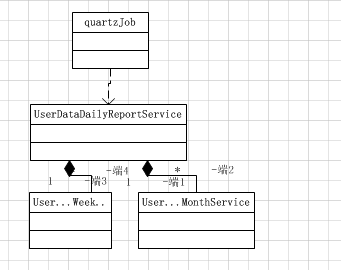
在调用的时候$(…).rootModal(..);

在设计模式上，2类似于覆盖了，而1却什么都不是。

# 项目中觉得亮点

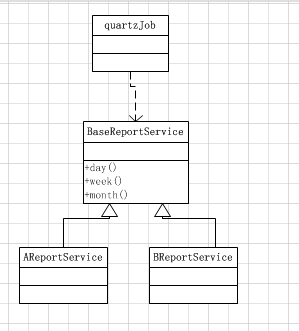
## 对于统计任务的设计演变

第一版：每一个统计指标+每一个统计维度，都是一个单独的服务类，如设计1



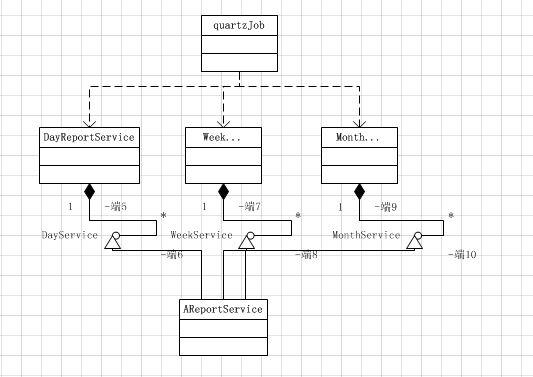
图设计1

第二版：每一个统计指标都有不同维度的统计行为，如图设计2



图设计2

第三版：每一个统计维度都是单独类，但是每一个指标实现不同的统计维度，来表示不同的行为，如图设计3.还可考虑将DayReportService、Weeek...、Month等转化为quartz的行为



设计3

|  |  |  |
| --- | --- | --- |
| 设计方式 | 优点 | 缺点 |
| 设计1 | 统计指标和统计维度的关系清晰明了  不同指标与时间统计维度相独立 | 类越来越庞大 |
| 设计2 | 统计指标清晰明了  类数量足够少 | 统计指标与时间统计维度无法直观查看  时间统计维度无法独立共用 |
| 设计3 | 统计指标清晰明了  类数量足够少  时间统计独立共用 | 虽无法看出统计指标与时间维度之间的关系，但是可以通过实现接口查看。一般第一视觉也是根据统计指标维度查看。所以这个缺点可以忽略 |

## 屏幕资源的设计

不同资源类型，采用策略的方式来实现，实现对外扩展，对内封装



对于下发数据的设计

### 设计1：



### 设计2：



### 对比：

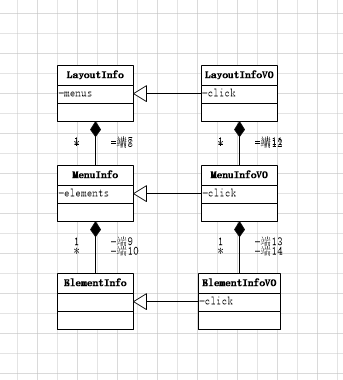
两个设计关键的差异性在于对于待机图的设计；

设计1：把待机图当成了菜单数据的一种特种数据，虽然数据结构没问题。但是在下发终端时，由于数据的行为不一致，导致在数据库中做了不同的兼容处理

设计2：将待机图当成了一种特殊的数据，也从行为上可以和菜单做区分。也不需要进行兼容。所以设计2比较好

## 热力图设计

### 设计1：





### 设计2：





### 对比：

从扩展性来讲；设计2会比设计1来得更容易扩展。因为有些扩展性菜单没有写死，对于可变的数据保留了扩展性和封装性。设计2做到了数据不变，行为扩展。不像设计1，从数据来决定行为

## 保存修改用户的设计

### 设计1：



### 设计2：



使用了spel的表达式，来设置参数值。参数上的注解，可使用methoc.getParameterAnnotation的方式来获取。这里对于注解类的使用中@Inherited，表示这个这个在类的注解可以被继承，但是却无法被用于方法或者属性的继承

### 对比：

设计1会在代码上比设计2多出更多无畏的setModifyRecord的方法；之前以为通过SecurityContext的方式来获取登录名不好，想自己在搞一层中间层。发现其实有点多余，因为本身SecurityContext就是针对三方又自己封装的。设计1将SecurityContext放入了RecordEntity，这就让RecordEntity的职责溢出，不太合理。所以从代码美观和代码层次上来说，更好