# 问题汇总

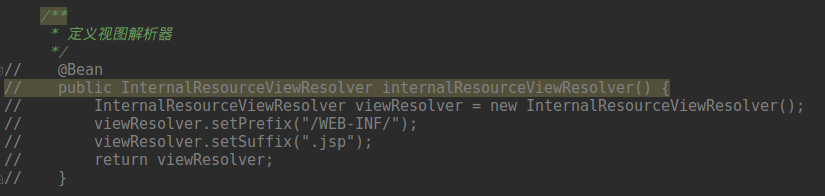
|  |  |
| --- | --- |
| 问题描述 | 参考案例 |
| 01.出现大段代码注释的情况 | 问题01 |
| 02.清理掉无用的引用 | 问题02 |
| 03.类必须加上注释内容 | 问题03 |
| 04.凡是属性需要使用文档签名 | 问题04 |
| 05.idea提示Java可以推断的,省略显性定义 | 问题05,可以不做追究 |
| 06.如果不是工具类,没有被使用,删除掉 | 问题06 |
| 07.不要出现idea出现的拼写错误提示 | 问题07 |
| 08.公共基础模块的jar包就是复用使用的,不是复制粘贴到每个项目中都保存一份 | 问题08 |
| service和controller中写业务的对比 | 说明09 |
| 10.复杂业务流程必须在方法注释中增加处理思路,实现步骤等内容 | 问题10 |
| 11.对于service和mapper接口,必须增加接口用途,接口参数含义,接口返回值含义 | 问题11 |
| 12.对于项目内容的重复的方法作必要的封装,如果不做封装一定有原因,没有原因必须封装,而不是复制一份代码 | 问题12,复制代码去开发,只能说明代码素养太差 |
| 13.给自己的代码增加点美感,不要一团乱 | 说明13 |
| 14.以后不要再出现狗屎一般的魔术 | 问题14 |
| 15.缓存的命名空间集中处理,而不是写散在业务代码中 | 问题15 |

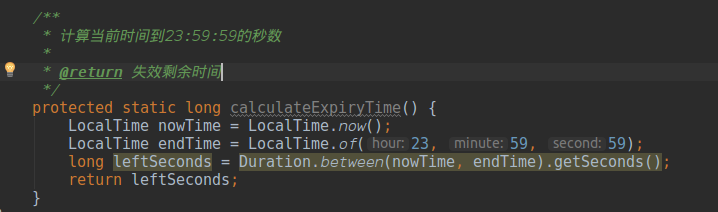
## 问题01:大段注释的情况的代码

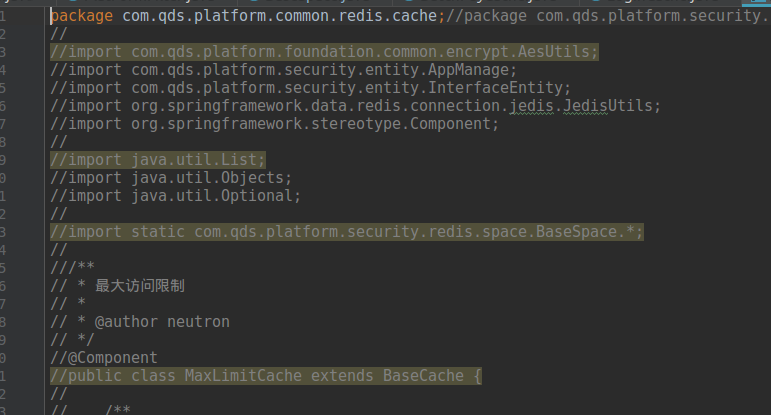
问题:不要以为注释的代码将来会用的到的,很多人都有这种想法,然后项目中很多代码都宁愿注册也不删的,这种情况就好比大街上捡到某个有点功能的破烂,总感觉将来会用的到,拿回家就一直堆着,占用家里的空间.大段注册的代码,类似,明明都不用的代码,还总想将来可能用到,将来可能用到的概率有,也只不过你注释留下来1%不到的代码.

解决:凡是注释掉的代码删除掉,可以做版本控制,将来有需要的时候根据需要把代码还原回来就好.

案例:







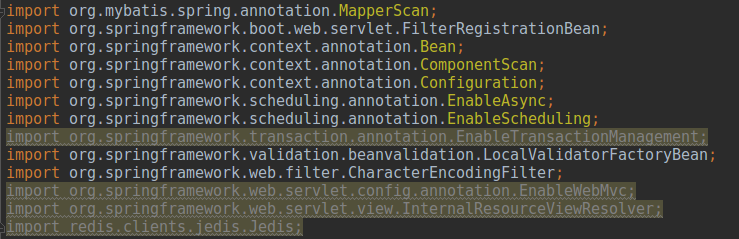
留着下崽吗?

## 问题02:对于类中出现大量的无用的引用

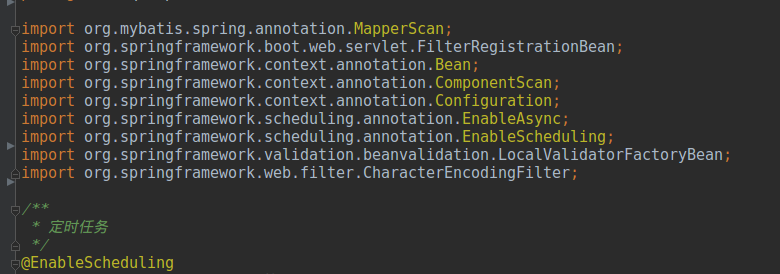
问题描述: 命名这个类就需要引用其他的内容,非得狗尾续貂地增加其他乱七八糟没用实际用途的代码,需要这个类在加载的时候加载一堆没有实际使用用途的代码,如果一个类加载3-5个不用的类,那么如果项目有个几千个类,那么就会有3-5000个根本不需要加载的类,浪费加载资源,消耗加载时间.如果不知道这块内容,去看看类加载机制去.

解决: 这个问题的处理本身就是根本没有什么干净的意识,一个脏了衣服,染点颜色,沾了些垃圾,保暖作用还是有的.使用idea的工具, ctrl+alt+o来处理

案例:



结果:



## 问题03:类必须加注释

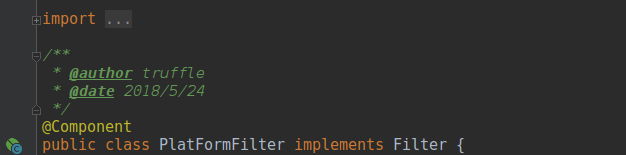
问题描述:类必须加上类注释说明,类注释的作用到底是什么,最重要的内容是说明这个类到底是做什么用的,然后增加作者,最后可以增加时间.

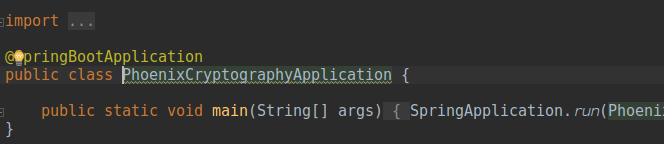
我们知道这个类是做什么的,方便后续开发者的理解;我们知道作者是谁,可以通过这个作者,如果还在职,可以去做咨询.所以类注释的内容必须有的是,这个类的说明,这个类是做什么的,作者是谁,这两个是最重要的,对于业务开发而言.

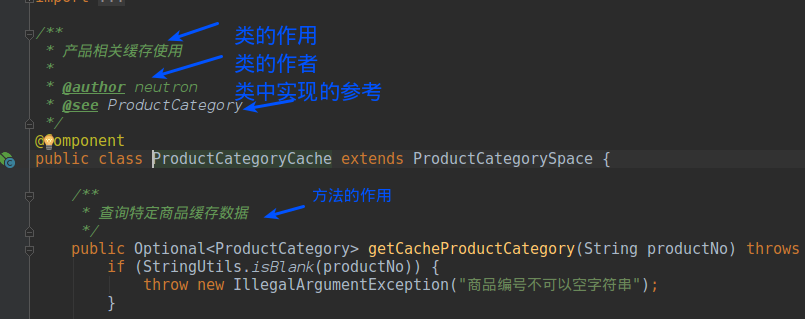
解决:这个就是同理心的问题,或者说是代码编写习惯问题,或者说换位思考问题.

案例:







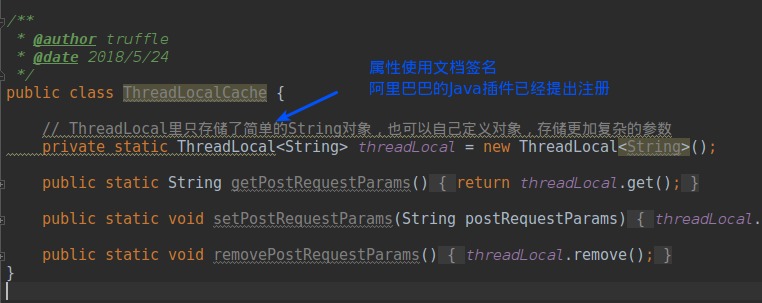


## 问题04:凡是属性需要使用文档签名

问题描述:类的属性需要使用文档注释内容,尤其JavaBean相关的属性,都必须使用文档注释,而不是使用单行或者多行注释.这个可以在断点调试查看属性作用,可以使用idea中直接直接查看属性,而不只在那一类中查看属性.

解决:这代码编写习惯问题,或者说换位思考问题.

案例:



## 问题05:如果在属性定义中,idea提示Java可以推断出来,那么可以省略掉显性定义.

问题描述:这类问题属于多此一举的问题,因为Java可以自动推断出来定义的数据到底是什么类型,这个是Java7以后才有的特性.

解决:Java可以自动推断出来的数据类型,可以不用显示填写上.

案例:

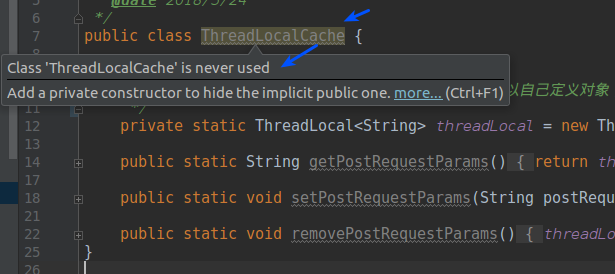


## 问题06:如果类不是工具类,项目中根本没有任何用途删除掉

问题描述:如果不是工具类,在项目中还没有什么用途使用的时候,那么删掉掉,不要以为将来会用,你不会给你的新房子中买一个印钞机,等待将来自己可以印钱使用的

解决:直接删除掉

案例:

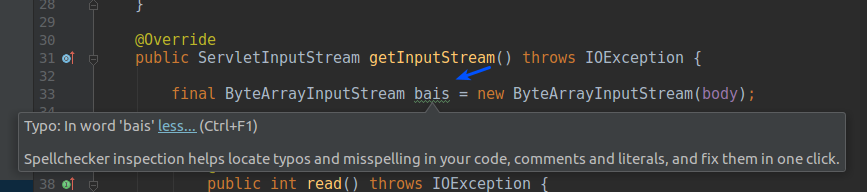


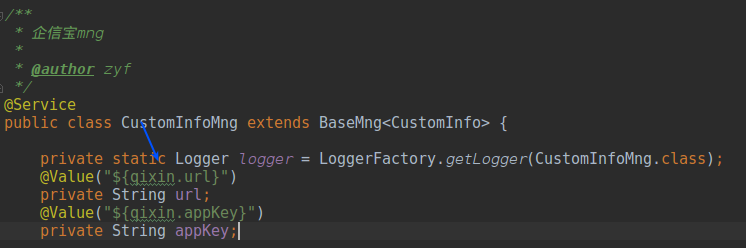
## 问题07:不要出现idea提醒的拼写错误问题

问题描述:不要出现idea提醒的语法错误,这东西和写作文类似,写一堆错别字,虽然大体上没有任何问题,但是如果一个类里面很多拼写错误,那就是小学生的作文水平,代码水平也只能说是初级而已,因为连命名都不在意的人,代码水平不会多么高

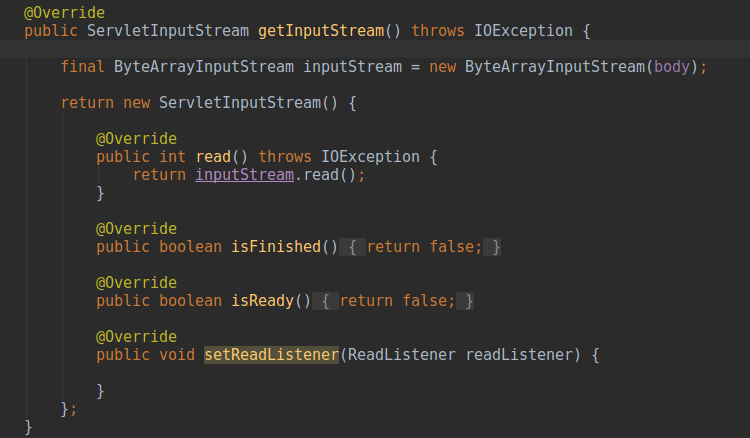
解决:改正掉

案例:





修改后:

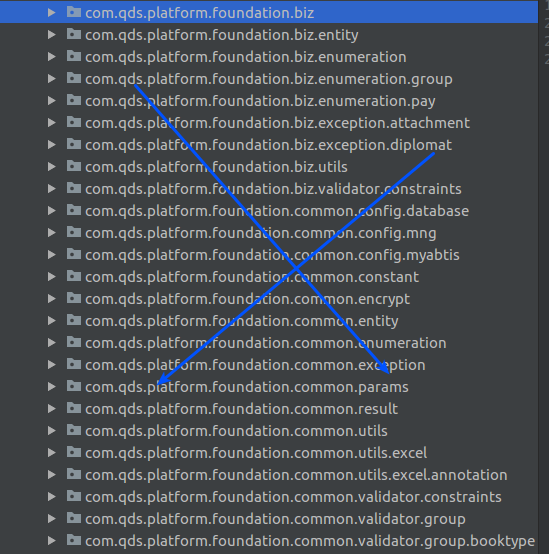


## 问题08:基础项目模块的jar就是为复用而使用,而不是哪个项目都要粘贴的

问题描述:foundation模块是就给其他模块复用使用的,而不是复制粘贴的每个项目中的,单一职责原则难道不明白吗?属性的单一职责,方法的单一职责,模块的单一职责,jar包的单一职责,微服务的单一职责.如果费率修改了,除了修改基础模块,修改依赖于基础模块的项目即可,难道谁还会记得把项目代码粘贴到其他模块中的代码吗?

凡是有大的业务变动修改,都需要修改两份,如果再出现复制代码的行为,就需要修改不知道多少份的项目和代码,我实现想不到有什么特殊理由,要这么处理代码.

解决:使用通用的jar,而不是重新复制粘贴一份.



## 说明09:具体业务在controller层和在service层处理的利弊.

情况1:很多人都喜欢所有的内容在service写具体的业务,然后service调用service或mapper.

情况1的不足:请参考qds中的内容,很多时候是为了实现具体业务,导致不得不写很多service和mapper,尤其mapper中的xml,在qds中基本上占用工作量的1/4-1/3的时间;为了减少写特定业务的xml,使用通用的增删改查的操作,引入tk.mybatis,这样可以减少xml的代码量;为了特定查询写个service中的方法,这个方法只有特定的实现,其他的方法基本上不可能复用的,为了功能写特定实现,就是一次性买卖,其他的操作很难进行复用,所以不推荐在service中为特定功能写特定的方法.service针对于业务来写,可复用的少,为特定功能写的多,需要维护的代码多,事实上qds中把这些问题体现的淋漓尽致.所以才会有情况2.

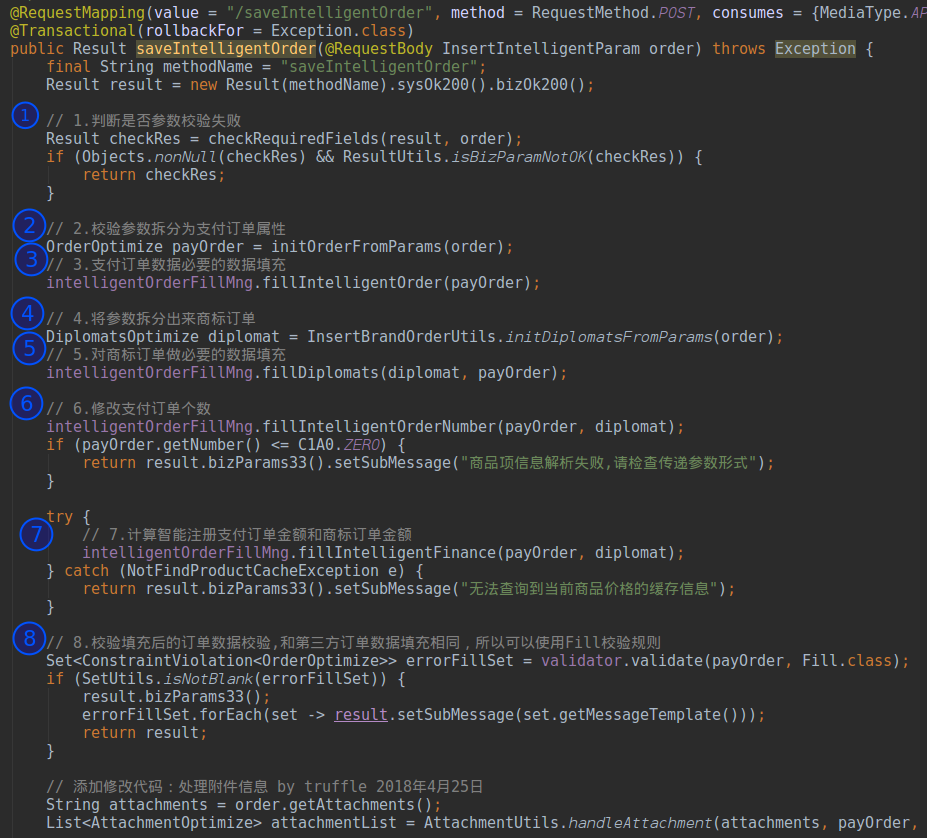
情况2:将具体的操作写在controller中,复用service和mapper,这样的话.

针对于情况2:对于简单的内容,可以根据单表的操作,都是用tk.mybatis,对于业务操作只用通用的service,只有负责的业务内容实现,写几个service方法而已,其他内容可复用的话,都进行复用.基本上我们只维护controller即可,因为service和mapper基本上都是复用的.

对于复杂操作,不管是情况1,还是情况2,都需要在方法的方法注释中把处理思路写明白.针对于订单模块,我是在方法实现内部,写很多,比如1.xxx,2.yyy,3.zzz来描述处理思路.

其实不管是在controller和service中,只要复杂业务的处理,都必须增加具体业务的处理步骤.

比如保存智能注册订单时,我是以下处理的:

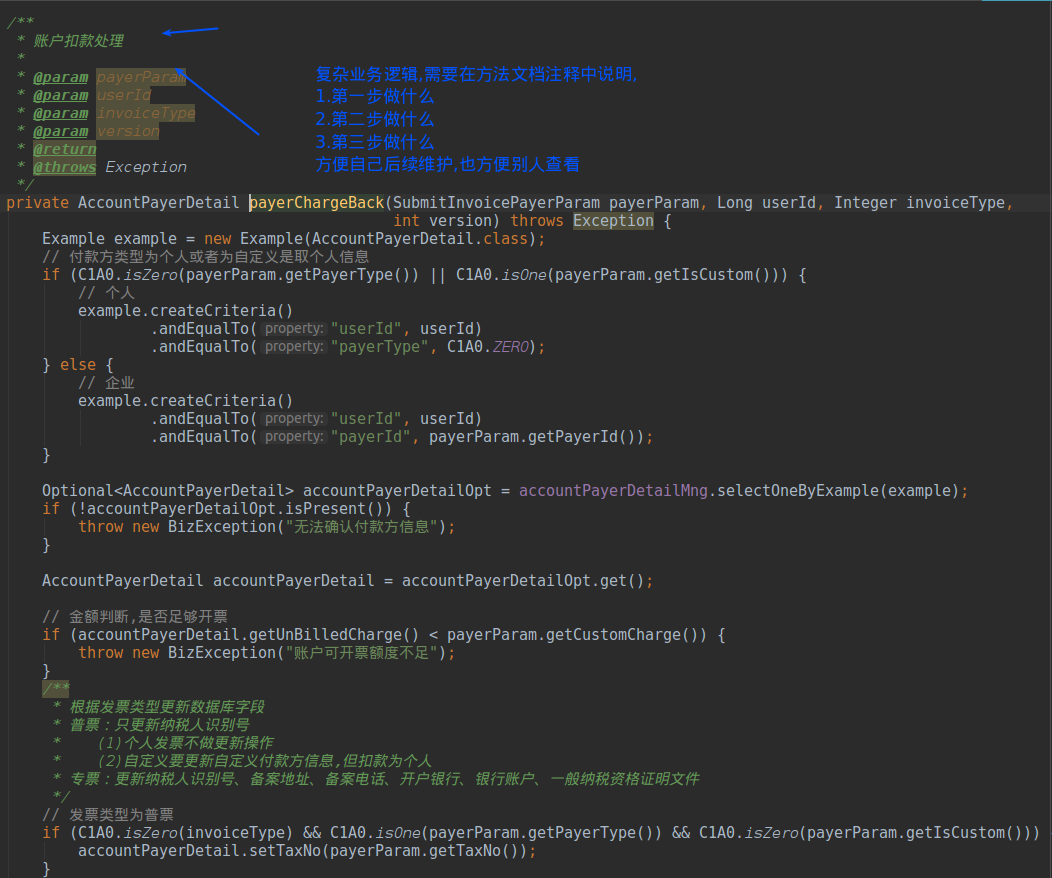


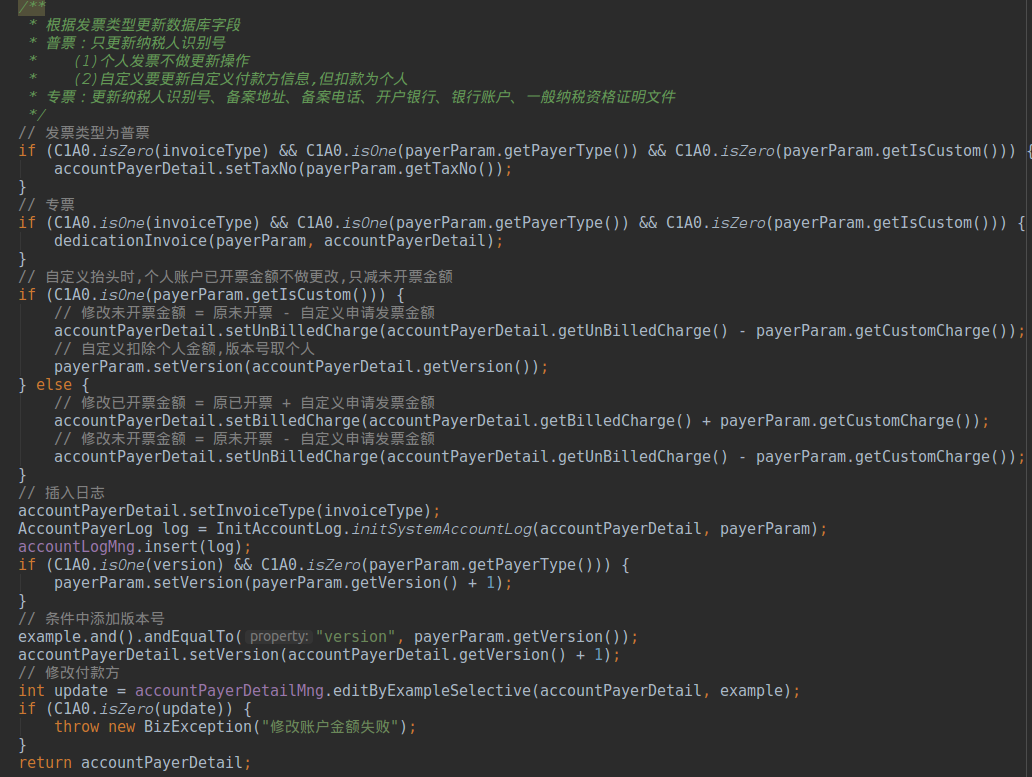
|  |  |  |
| --- | --- | --- |
|  | service写业务 | controller写业务 |
| 简单  业务 | 不足:需要在service多维护个方法;写的越多,需要维护的代码越多 | 无 |
| 无 | 优点:复用service和mapper;业务在controller编写不用去service和mapper,xml中去查找代码 |
| 复杂  业务 | 不足:无 | 不足:无法复用,但可以把controller中重复代码进行抽象到service中去复用 |
| 优点:可复用service,给其他的controller使用 | 优点:无 |
| 优势 | 1.可复用service给其他controller复用 | 1.复用service和mapper;  2.少service,mapper,xml的维护 |
| 总结 | 1. 需要给其他controller复用,那么使用service写具体业务; 2. 如果不需要给其他复用,那么直接在controller中写业务即可,增加service,mapper,xml的复用. 3. 个人不推荐不鼓励在service写业务实现内容,除非controller中实现比service中问题大,否则推荐在controller中 | |

## 问题10:对于复杂的业务,需要在方法上增加业务处理流程

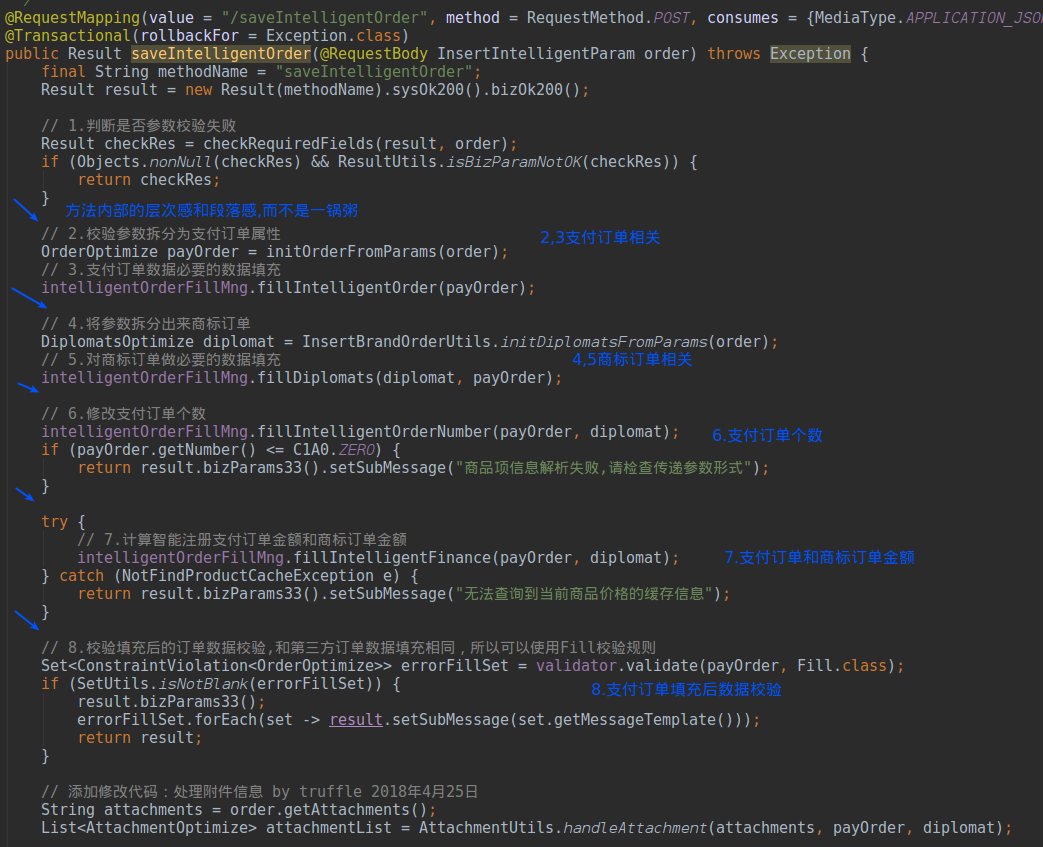
问题描述:对于复杂业务逻辑处理,参考说明09,不管是在service,还是在controller中,必须在方法的文档注释中增加业务处理步骤,即你的设计思路和你处理问题的步骤.目的就是方便你后续的维护,方便其他人参与进来维护代码.

解决:在方法的文档注释中增加方法设计步骤,或者实现思路,实现步骤等.





对于上面上面的代码,看不到具体的实现步骤思路说明.参考下我的代码,并不是我写的多么好,最起码有层次感,可读性更好一些.

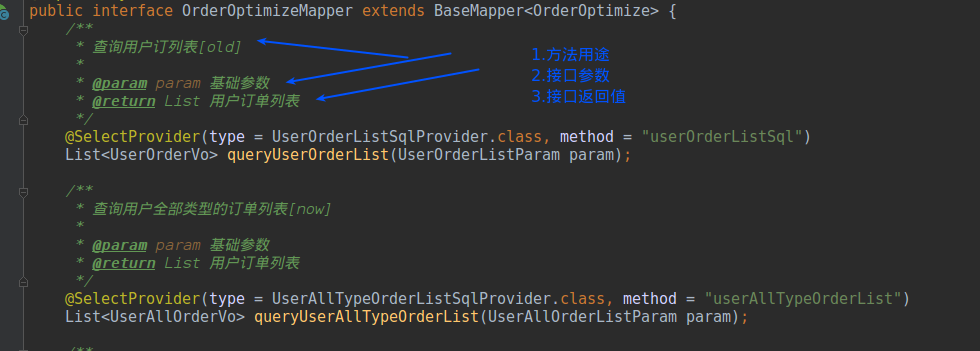


## 问题11:service和mapper中的接口,接口用途,接口参数含义,接口返回值都必须在文档注释中清除注明

问题描述:不要以为自己的写的可读性是多么的好,也不要有多么高的优越感,既然是做开发,那么就要方便自己和方便别人后续维护.凡是写的service和mapper中的方法,都必须增加接口用途,参数含义,接口返回值含义.

出发点是为自己以后的维护,别人维护提供方便,应付了事最可恶.

参考我的mapper和service代码



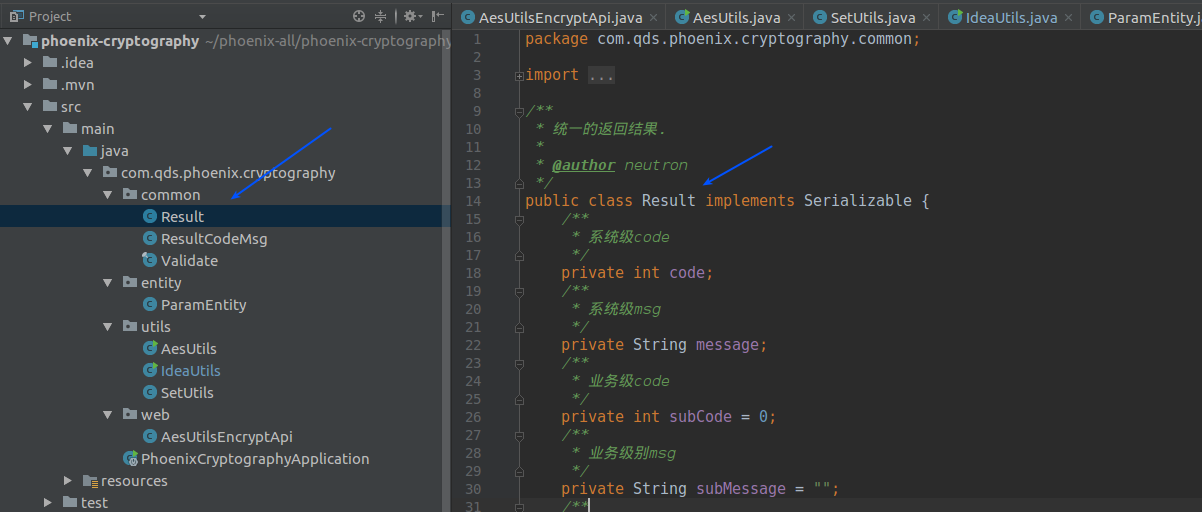


不要为了省点事,给后来维护者留下一堆问题,给自己积攒一些代码良心.

## 问题12:基础模块的作用就是为了复用,而不是为了复制代码

问题描述:程序界都说这么一句话,那就是don’t repeat yourself.不要重复代码,因为你每一次重复,只有做修改,重复的地方越多,你修改的地方越多.为什么要抽象,面向对象的目的就是为了复用,而不是复制.

我们做业务开发,spring的jar一定会重用,那么为什么我们的代码就要复制粘贴呢?就不能把jar包引入进行进行重用吗?



记得面向对象的原则:继承,封装,多态,本质上都是为了实现复用,用比较少的代码实现更多的功能,实现代码的可拓展性.复制,只是为了实现功能,最不可取的行为.

## 说明13:方法内部段落层次感说明

问题描述:记得曾经写作文是一逗号到底,然后有个句号,这就是一气呵成,没有停顿的感觉,当然这样的作文是被调侃的.那么我们的代码到底有要不要有层次感的,尤其方法内部.

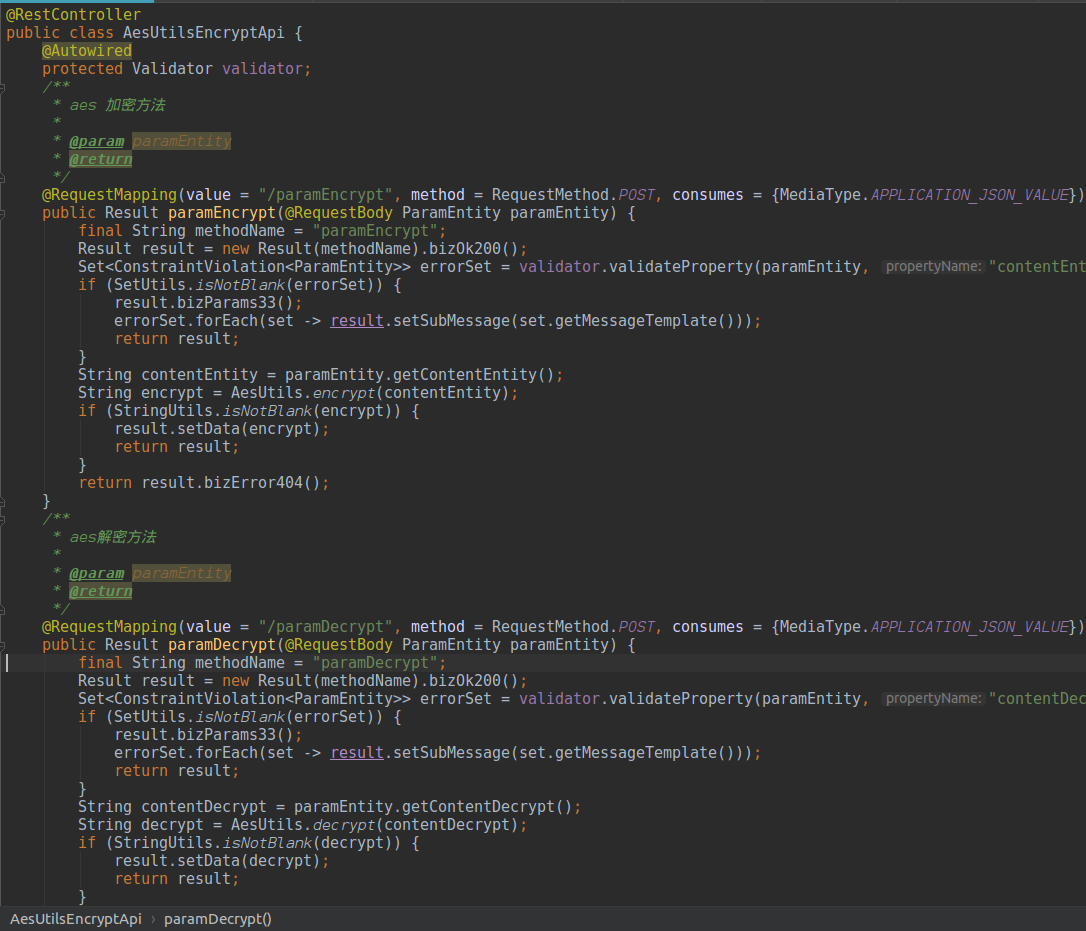
任何一个方法可能内部调用1个方法,或者调用多个方法,调用一个方法可能不需要什么层次感,就好比叙述一件事情,但是调用多个方法,比如二次世界大战,里面那么多内容,那么一气呵成写完成吗?或者按照年月,或者按照参战方来写,总之是有维度和层次的.

案例:

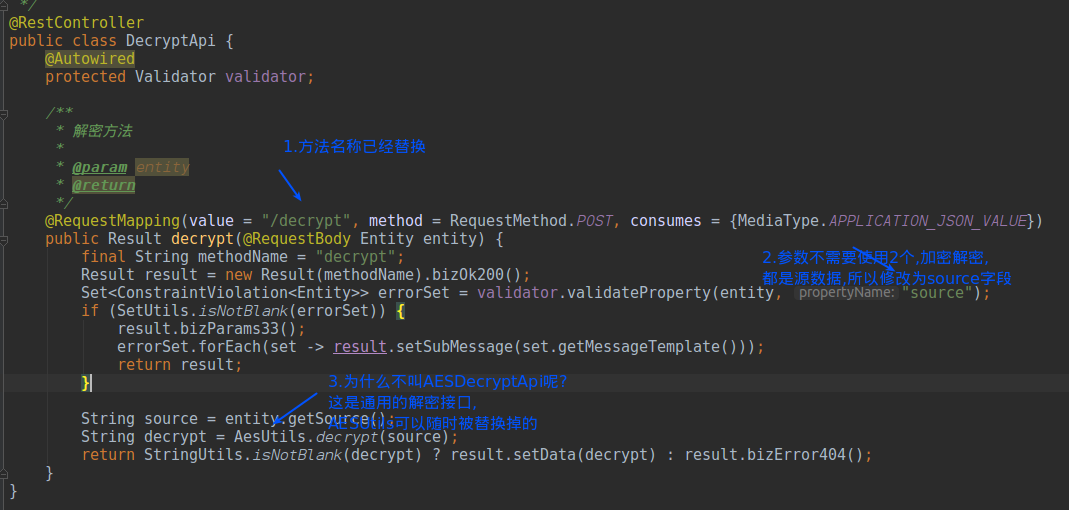


这段代码有什么层次感上的问题呢?这就是没有停顿的流水账,和做事情差不多,做了1/3是个里程碑,可以停止休息,都没有,一直到底完成,没有任何终止,和写文章一逗到底没有任何区别.其次,根本不明白空格在编程中到底有什么作用?举个极端例子,属性,变量,方法之间都没有空格,你要怎么看,怎么读,你的内心会怎么样.

比如处理如下:



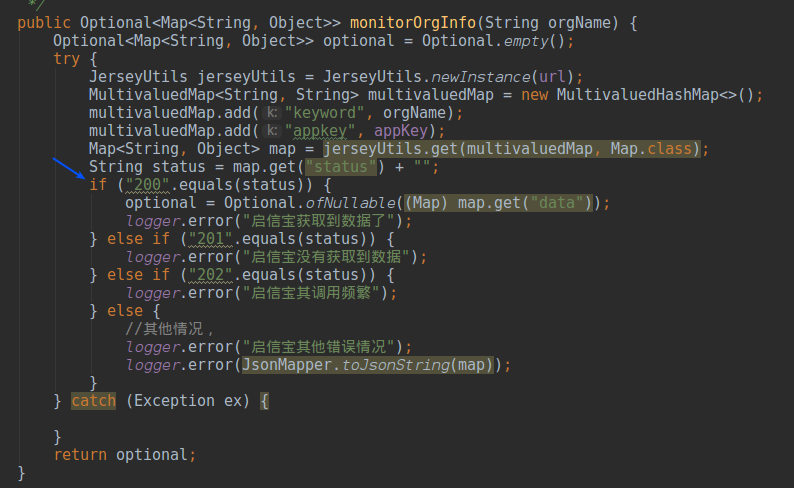
最根本的行为就是单一职责,做好单一职责,用坏单一职责,所说的过犹不及,不怕过,因为很多人根本没有到过的地步呢,只是不及而已,不断行动才能不断思考适合不适合.



## 问题14:不要出现的魔数

问题描述:谁知道突然冒出来个数字,是什么鬼东西的,现在开发的人也许知道,过了一个月呢,过了半年呢,后来维护的人会知道吗?

如何解决:定义个变量,定义个有含义的变量,必要的时候加上写注释,说明它代表什么含义.

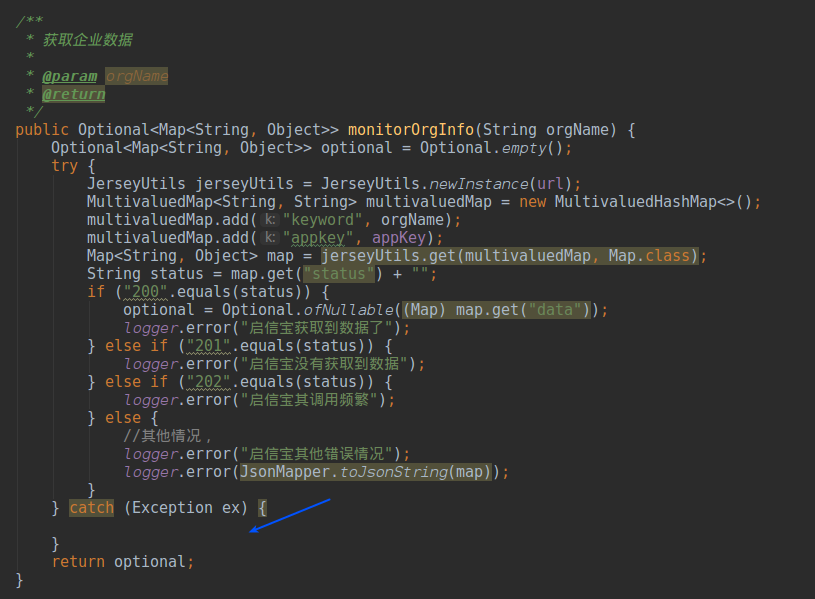


问题15:凡是异常不做处理,比作出名和增加日志说明

问题描述:很多情况,做try ...catch,那么捕获到了异常,却不做任何处理,这会让任何查看这段代码的人发蒙的,第一想法就是为什么不做异常处理呢?

如何解决:1.在catch后面增加注释内容,说明不做处理的理由; 2.增加logger.error内容,最起码告诉服务器发生异常了,可以不做处理,但是谁不知道后续是否真的不做处理,如果连日志都没有,发生问题都不知道,那就是大问题.

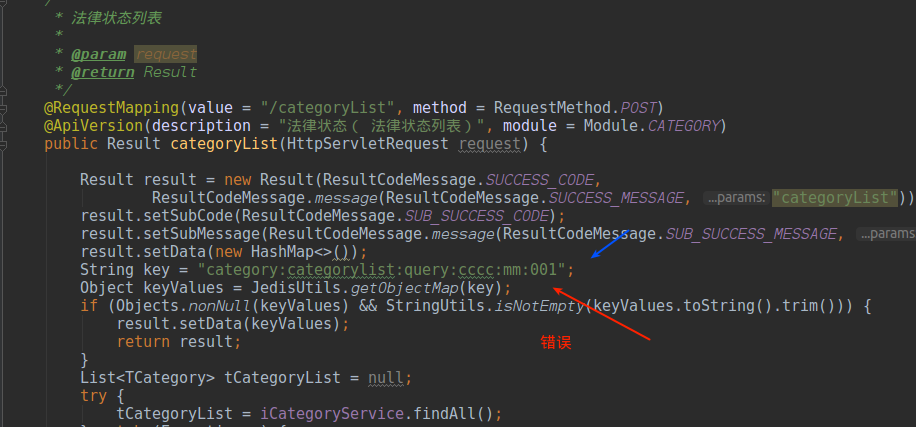
案例:

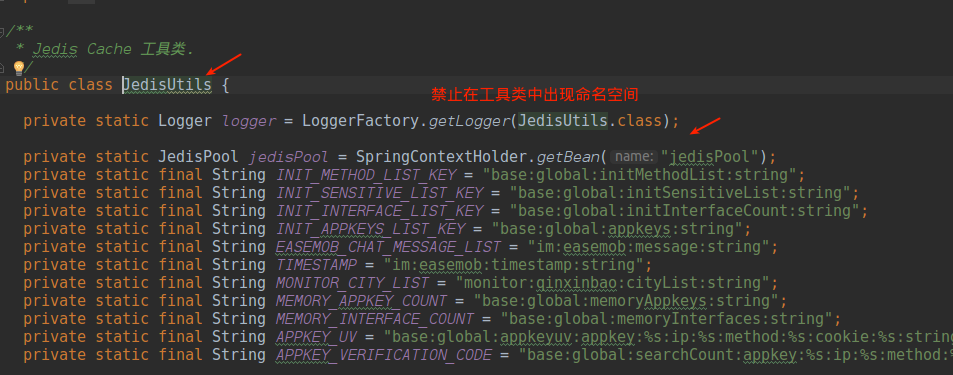


## 问题15:缓存命名空间必须集中放在单独包下的单独类中,不准四处分散

问题描述:以后要处理缓存的问题,还得根据请求去四处去找缓存的命名空间,如果约定俗称在某个模块的某个包下,这不是很好处理吗? 为什么要约定大于配置呢,那是因为约定的东西知道哪里可以找得到,不用根据配置四处去找,第一次浪费一堆时间,第二次许久又是浪费一堆时间去查询缓存是什么,然后采取处理. 如何解决:集中放在模块下面的cache.space包下面.满足单一职责原则,同时方便拓展和修改.约定大于配置.

错误的处理方式:禁止在业务代码中出现缓存命名空间





应该如何处理,参考深度优化中订单模块的处理方式,集中处理

