4月读书笔记与小结

4月份提能内容：

1.《大话设计模式》第3-4章：

* 第三章：拍摄UFO—单一职责原则
* 第四章：考研求职两不误—开放-封闭原则

2.《重构 改善既有代码得设计》第1-2章：

* 第一章：重构，第一个示例
* 第二章：重构的原则

《大话设计模式》第3-4章读书笔记：

单一职责原则：

1. 单一职责原则(SRP—Single Responsibility Principle)，即有且仅有一个原因引起接口或类的变更，是面向对象设计六大原则之一。
2. 如果一个类承担的职责过多，就等于把这些职责耦合在一起，一个职责的变化可能会消弱或者抑制这个类完成其他的职责的能力，这种耦合会导致脆弱的设计，当变化发生时，设计就会遭受到意想不到的破坏。
3. 软件设计真正要做的很多内容，就是发现职责并把哪些职责相互分离，其实要去判断是否应该分离出类来，也不难，那就是如果你能够想到多于一个的动机去改变一个类，那么这个类就具有多于一个的职责，就应该考虑类的职责分离。

问题由来：

之所以会出现单一职责原则就是因为在软件设计时会出现以下类似场景：

T负责两个不同的职责：职责P1，职责P2。当由于职责P1需求发生改变而需要修改类T时，有可能会导致原本运行正常的职责P2功能发生故障。也就是说职责P1和P2被耦合在了一起。

产生原因：

没有任何的程序设计人员不清楚应该写出高内聚低耦合的程序，但是很多耦合常常发生在不经意之间，其原因就是，职责扩散-因为某种原因，某一职责被分化为颗粒度更细的多个职责了。

解决方法：

遵守单一职责原则，将不同的职责封装到不同的类或模块中。

单一职责，追求的是简单明确的设计，强调的是分工明确，职责简单。这种设计，不局限于类的设计，可以扩展到任何一个程序逻辑单元的设计，如函数的设计，模块的设计，工程的设计，项目的设计，集群的设计等。单一职责的核心可以理解为：化整为零，将整体分解为局部的逻辑单元，不同的逻辑单元之间尽可能的相对独立，并且职责明确。例如，一个项目，可以分解为多个工程；一个工程，可以分解为多个模块；一个模块，可以分解为多个包和类；一个类，可以分解为多个属性和方法行为。设计层面的分解，可以对应到具体实现层面的分离。

单一原则的好处：

* **组织代码**  
  让我们想象一个汽车修理工。 他使用很多工具​​一起工作。这些工具按类型分为：钳子，螺丝刀（十字/刀片），锤子，扳手（管/六角）等，他如何组织管理这些工具呢？他使用许多小抽屉将这些工具分门别类存放，这些抽屉其实类似模块作用，专门用来管理各种类。
* **减少脆弱**  
  当一个类有多个理由需要修改时，它变得脆弱，在一个地方的修改会导致其他地方不可预期的后果。.
* **更松耦合**  
  更多职责责任导致更高的耦合。  
  耦合也是一种责任  
  高度耦合导致高度依赖，意味着难以维护。
* **代码改变**  
  对于单一职责模块重构更容易.  
  如果你想获得猎枪的效果，就让你的类有更多职责。
* **维护性**  
  维护一个单一职责的类比维护一个铁板一块的类更容易。
* **易于测试**  
  测试单一目标的类只需要很少的测试类。让“用测试替代文档” “self documentation by tests”变得更加容易
* **易于调试**  
  在一个单一职责类找到问题是一件更容易的事情。

设计中哪些地方需要单一职责：

* 方法
* 类
* 包
* 模块

如何识别SRP被破坏？

* **类有太多依赖**  
  类的构造器有太多参数，意味着测试有太多依赖，需要制造mock太多测试输入参数，通常意味着已经破坏SRP了。
* **方法有太多参数**  
  类似类的构造器，方法参数意味着依赖。
* **测试类变得复杂**  
  如果测试有太多变量，意味着这个类有太多职责。
* **类或方法太长**  
  如果方法太长，意味着内容太多，职责过多。  
  一个类不超过 200-250
* **描述性名称**  
  如果你需要描述你的类 方法或包，比如使用"xxx和xxx"这种语句，意味着可能破坏了SRP.
* **低聚合Cohesion的类**  
  聚合Cohesion是一个很重要的概念，虽然聚合是有关结构概念，但是聚合和SRP非常相关，如前面论坛案例，如果一个类不代表一个高聚合，意味着低凝聚low Cohesion，它就可能意味破坏SRP。一个低凝聚的特点：  
  一个类有两个字段，其中一个字段被一些方法使用；另外一个字段被其他方法使用。
* **在一个地方改动影响另外一个地方**  
  如果在一个代码地方加入新功能或只是简单重构，却影响了其他不相关的地方，意味着这个地方代码可能破坏了SRP.
* **猎枪效果Shotgun Effect**  
  如果一个小的改变引起一发动全身，这意味SRP被破坏了。
* **不能够封装模块**  
  比如使用Spring框架，你使用@Configuration or XML 配置，如果你不能在一个配置中封装一个Bean。意味着它有太多职责，Spring配置应该隐藏内部bean，暴露最少接口，如果你因为多个原因需要改变Spring配置，可能破坏了SRP。

开放-封闭原则：

1．开放-封闭原则（OCP-- Open-Closed Principle），是说软件实体（类、模块、函数等等）应该可以扩展，但不可修改。

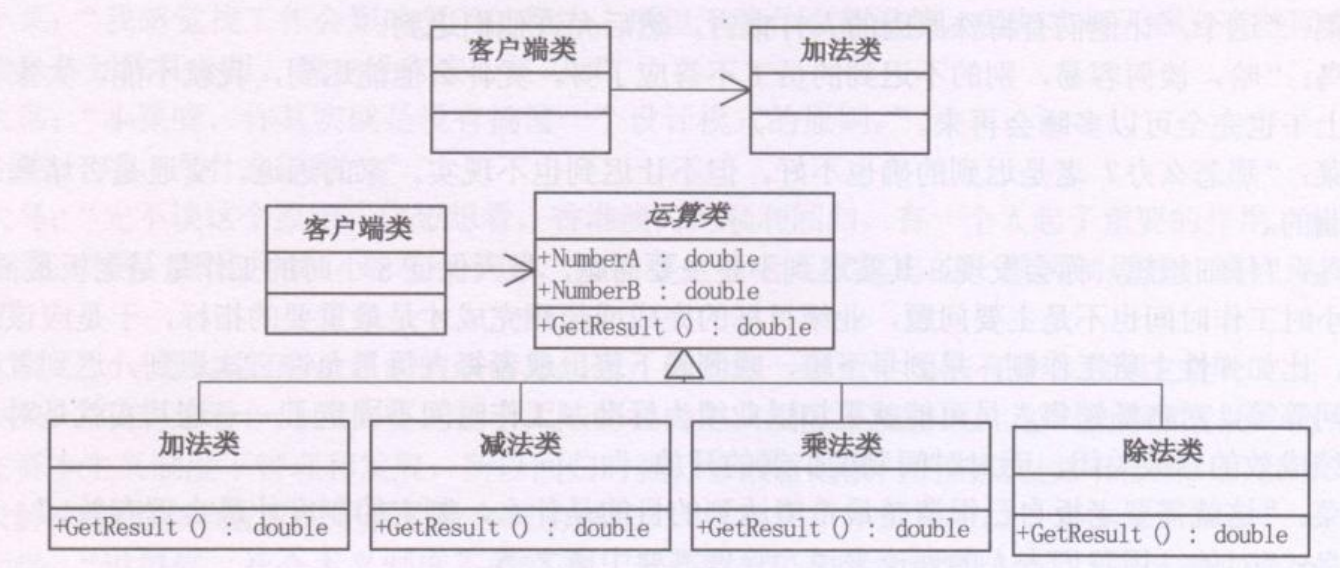
2．开放-封闭原则的两个特征：对于扩展时开放的；对于更改时封闭的。

3．在我们最初编写代码是，假设变化不会发生。当变化发生时，我们就创建抽象来隔离以后发生的同类变化。

4．开放-封闭原则的精神：面对需求，对程序的改动是通过增加新代码进行的，而不是更改现有的代码。

5．应该在开发工作展开不久就知道可能发生的变化。查明可能发生的变化所等待的时间越长，要创建正确的抽象就越困难。

6．开放-封闭原则是面向对象设计的核心所在，遵循这个原则可以带来面向对象技术所声称的巨大好处，也就是可维护、可扩展、可复用、灵活性好。开发人员应该仅对程序中呈现出频繁变化的那些部分做出抽象，然而，对于应用程序中的每个部分都刻意地进行抽象同样不是一个好主意。拒绝不成熟的抽象和抽象本身一样重要。



文中的这个例子就说明写一个加法程序很简单，在一个client类中很快就可以完成，但是一旦发生变化，比如增加减法，就要对这个类增加新代码，就会改变原有的结构，会让这个类变的庞大，但是，如果用一个运算类来继承client类，如果后期再添加有关运算类，就只需要对运算类做操作，这是就不会影响到client类，因为此时的client类不一定只有运算类相关关联，这样就是遵循开放-封闭原则。

开放封闭原则的优点

* 代码可读性高，可维护性强
* 帮助缩小逻辑粒度，以提高可复用性
* 可以使维护人员只扩展一个类，而非修改一个类，从而提高可维护性
* 在设计之初考虑所有可能变化的因素，留下接口，从而符合面向对象开发的要求

如何实现开放-封闭原则：

抽象，多态，继承，接口

思考：

1、没有扩展点的代码：如果一个类没有扩展点，那么当这个类需要添加新需求时，需要修改这个类中的方法，这样违背了封闭原则。为了满足新需求，需要创建一个新类来实现新需求的功能。

     2、虚方法：类中创建虚方法成员，这个虚方法就是一个扩展点。使用继承技术使子类可以修改虚方法。使用多态可以不需要修改客户端实现功能扩展。缺点：子类要么在基类方法上添加新功能，要么从新编写新功能，不存在中间状态（获取基类代码，在基类代码上进行修改）。与继承相同，子类对基类的私有成员不可见。

     3、抽象方法：创建抽象基类，基类中创建几个抽象方法。子类继承基类后实现抽象方法。各创建出的子类相互不影响，可以实现各自的功能。并且客户端只需要依赖抽象基类，任何实现的子类都满足要求。

     4、接口继承： 基于实现继承，所有子类（现有的和将来的）都是基类的客户端，子类都是依赖基类实现的。因此所有对实现的改动都会是客户端可能察觉到的改动。因此相对于继承，通常会建议优先选择组合，如果必须要使用继承，也要尽量使用只有少量分层的浅继承层次结构。（给继承图顶部节点添加新成员的改动会影响到该层次结构下的所有成员），接口继承可以根据不同的上下文给接口修饰丰富的功能（为接口添加修饰器）。接口比类灵活，具有强大的自适应性能力。

防止变异（protected variation）:识别可预见的变化点并围绕他们创建一个稳定的接口。

     1、可预见的变化（predicted variation）：单个类的需求应该直接与客户端的一个业务需求关联起来。冲刺过程中，开发人员可以提出预测将来潜在的相关需求，创建扩展点。

     2、一个稳定的接口：客户端依赖接口，如果接口发生变化，客户端也需要做响应的改变。用于表达扩展点的所有接口应该是稳定的，接口改变的可能性和频率应该很低，否则所有依赖的客户端都需要进行修改。

     3、足够的自适应能力：只在合适的位置上包含恰当数目的扩展点的代码被称为代码的“宜居带”。他能适应代码的变更需求，同时也不会增加复杂度和过度设计方案。

**小结：**

单一原则的优点：降低了类的复杂性，提高了可读性，可维护性，变更引起的风险降低。难点在于职责的划分，但同时也得考虑实际情况，因项目而异，因环境而异。

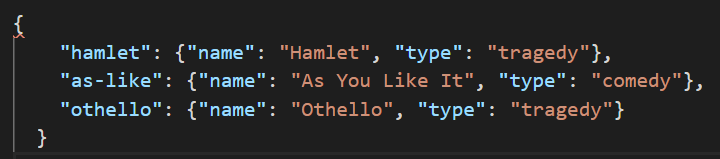
开放-封闭原则：软件实体（类，模块，函数等等）应该可以扩展，但是不可以修改。特性就是一个对于扩展时开放的，一个对于更改时封闭的。所以在实际的开放中需要更加需求选择开发的模式，java有很多的设计模式，各个模式都有特定的用途，开发过程中需要根据特定的场景，需求来选择，这样就可以做到“好钢用在刀刃上”，所以开发需要灵活，变通，这样会给开发带来便捷，也给后期的维护带来便捷。

《重构 改善既有代码得设计》第1-2章读书笔记：

这本书是以一个示例开始，避免了以往书籍先介绍理论的烦闷，总有弃书的冲动。

重构，第一个示例：

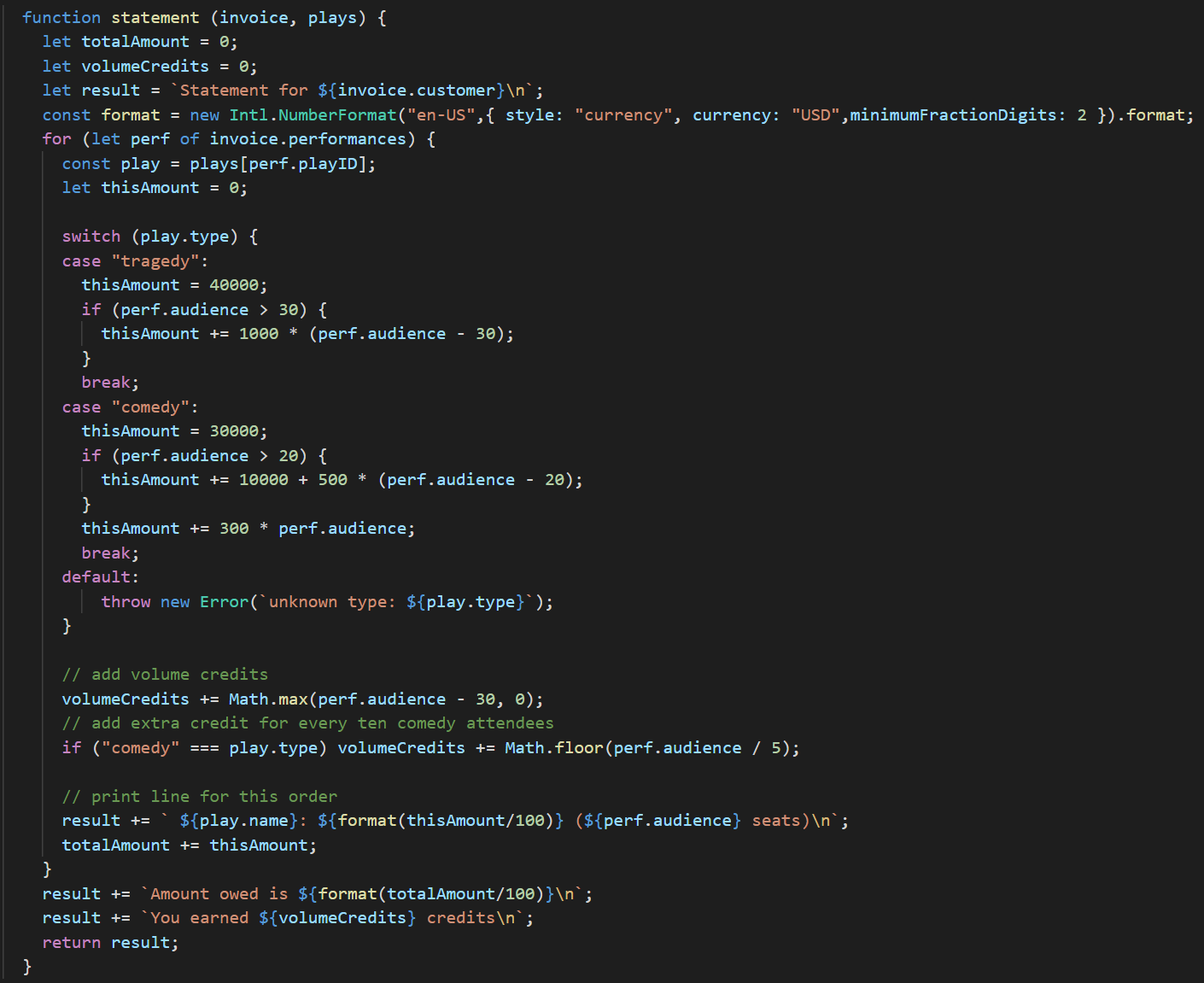
plays.json，剧目数据：



invoices.json 账单数据：



然后有了一个程序来输出这个账单的情况：



这个程序代码很少，简单，一个算账程序，但是，如果当你增加一个特性因为代码缺乏良好结构而导致比较难加这个特性的时候，你应该先重构这个代码，然后再加特性。

新增需求：

需求一：假设现在想要输出html的格式怎么办？

1. 部分程序员会采用if else 来判断输出为html还是输出为log，这样会使得代码变得非常麻烦。
2. 部分程序员会采用复制一遍，然后变成两个函数来输出。当然这个做法显然不可行，未来假设要改的时候，就要必须修改两个地方。

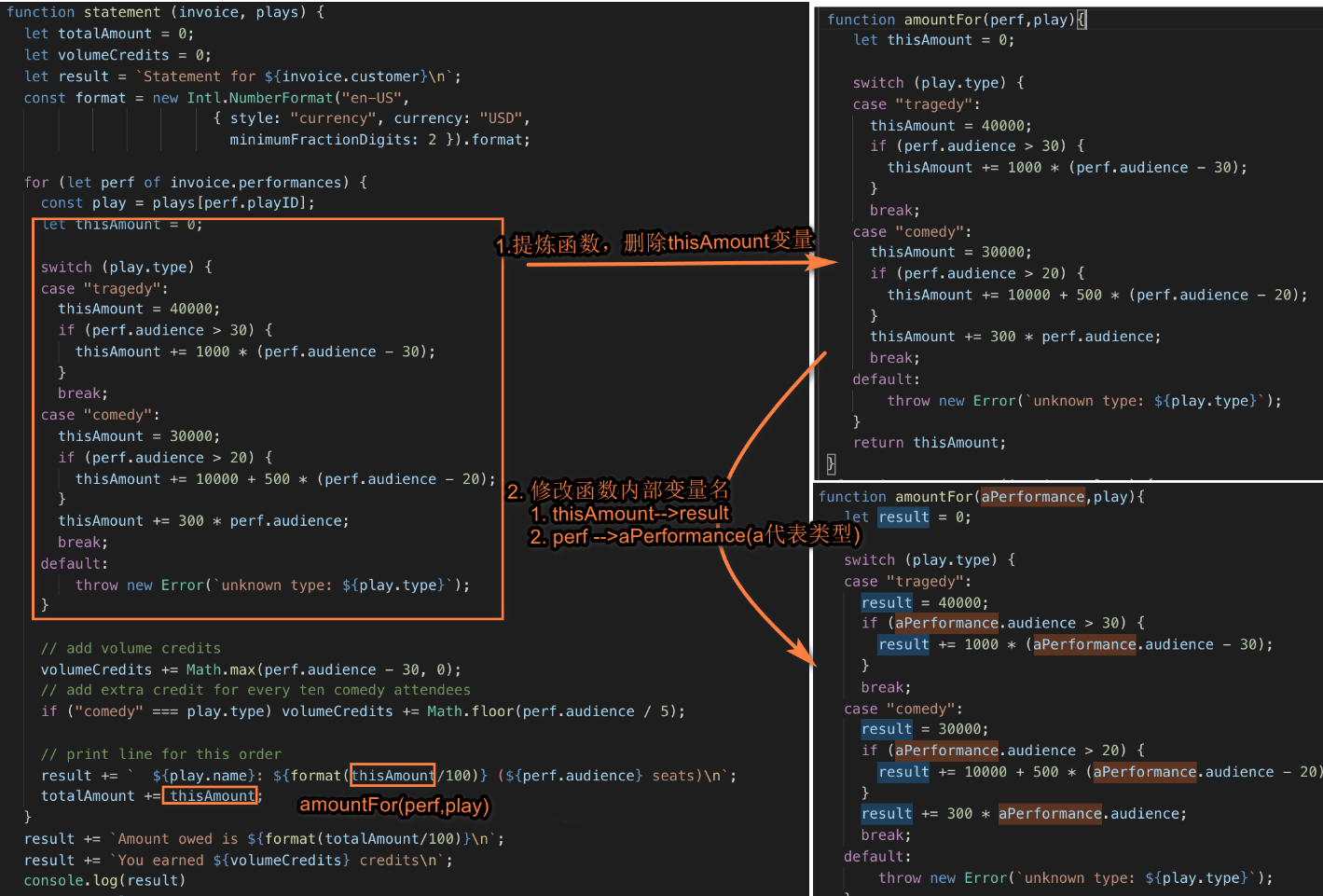
需求二：假设演员有更多的表演类型了怎么办？不同的表演类型肯定有不同的计费方式。

这样的新需求，就会造成对原有的程序改动大，并且不好改动，当然这样不重构原有框架，继续添加代码，就会造成后期的维护和别人来修改这段代码，都会造成困扰，而且维护的成本很大，但是如果重构原有的程序，就可以为未来的维护和修改带来效率。

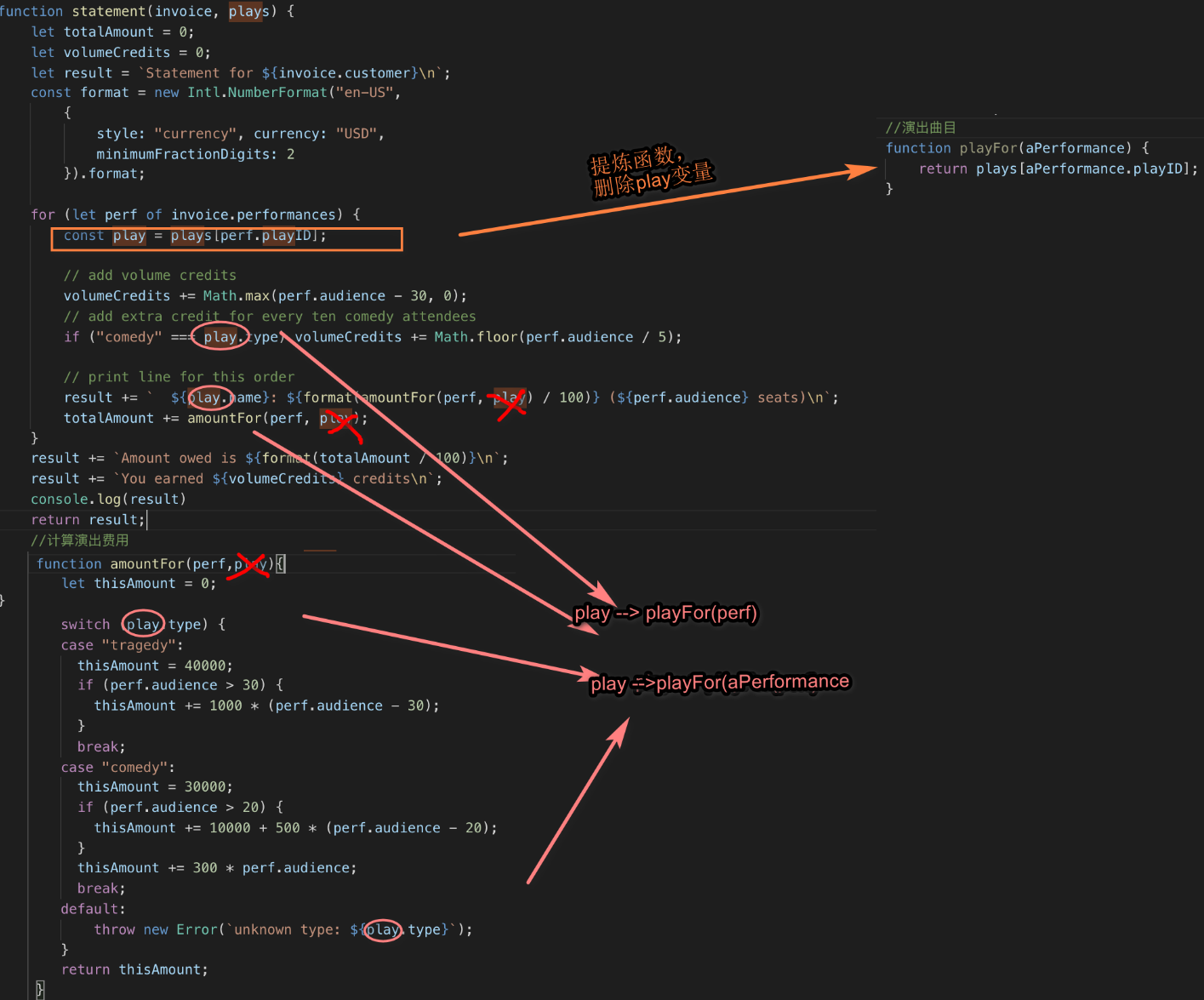
重构的第一步就要良好的测试，针对上面的程序，分解statement函数，因为该函数中间的很多计算函数，实际上时可以抽离出来的。

首先：

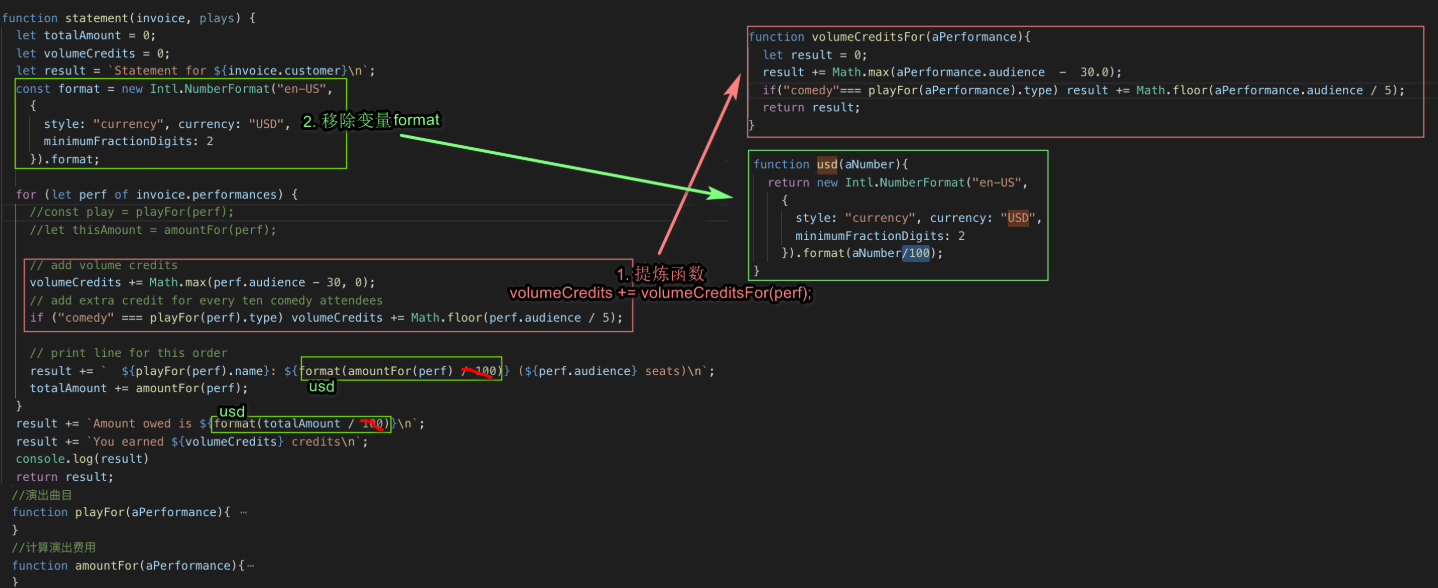
1. 提炼函数，删除thisAmount变量 thisAmount–>function amountFor(perf,play)
2. 修改函数内部变量名
   1. thisAmount–>result
   2. perf -->aPerformance(a代表类型)



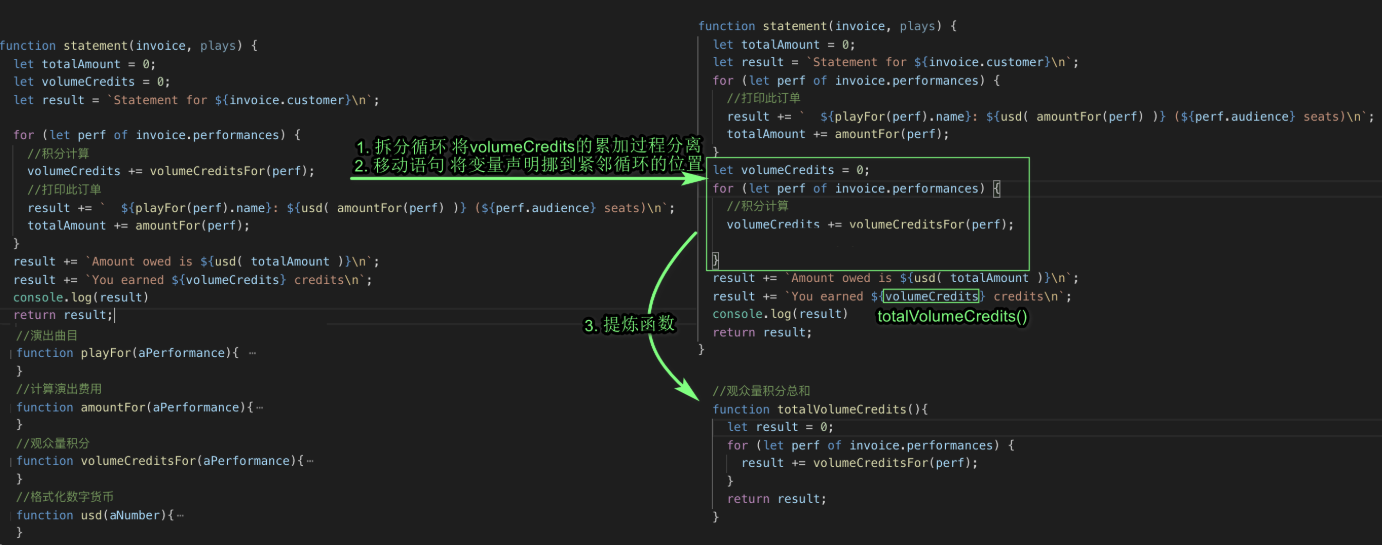
提炼函数，删除play变量 play–>function playFor(aPerformance)：



1. 提炼函数，volumeCredits–>function volumeCreditsFor(perf)
2. 提炼函数，删除format变量 format–>function usd(aNumber)



移除volumeCredits变量：

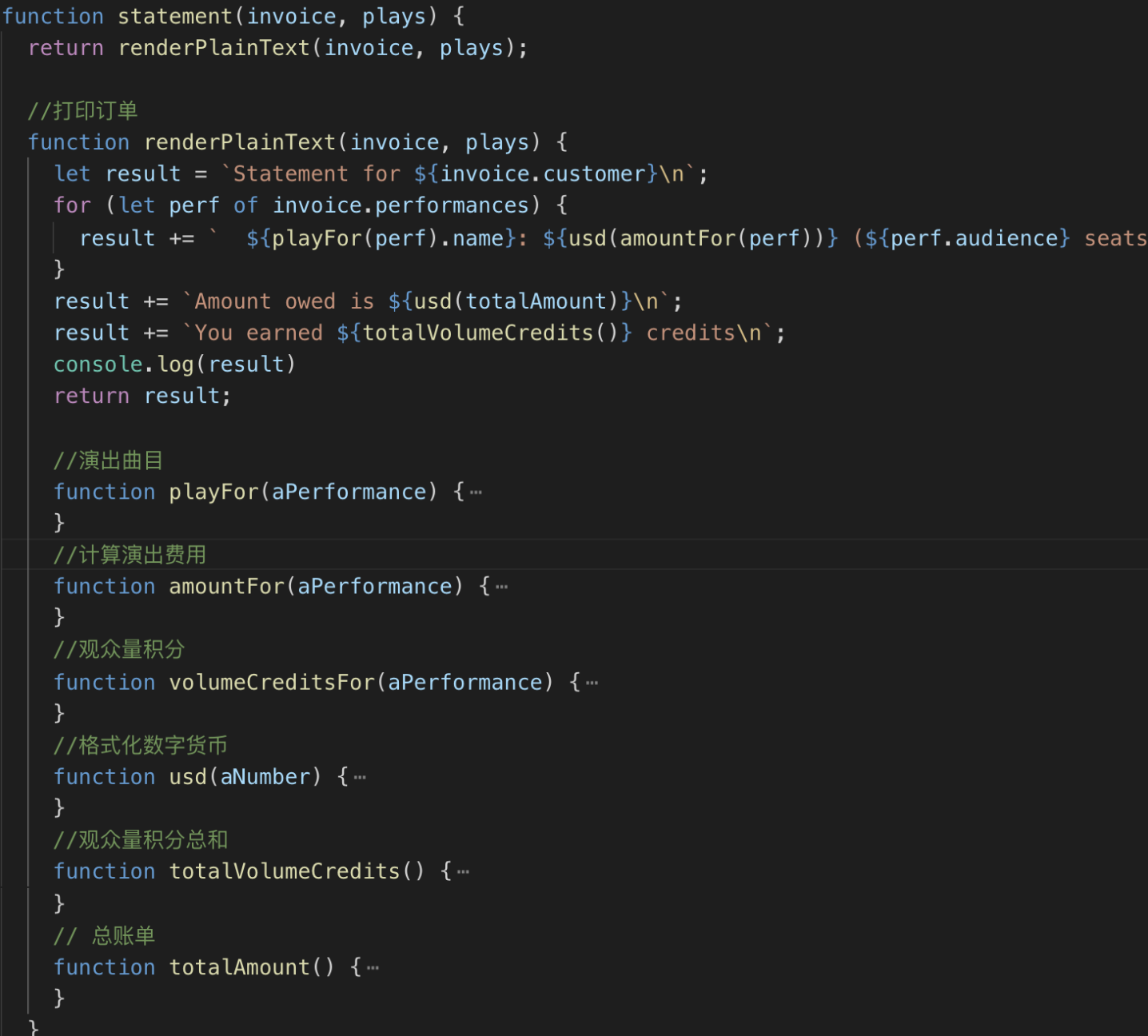


4. 拆分循环  
5. 移动语句  
6. 提炼函数  
7. 移除内联变量

同样的过程，删除totalAmount，result



重构后的代码：



重构后的代码可以清晰的看到：曲目，费用，积分，货币，总和和总账单等操作，简单明了，各个方法在各自实现，这样减少了一个方法的代码量，并让代码功能明确清晰，不管是维护还是新人读代码，都很容易理解代码的含义，重构的手法包括提炼函数，内联变量，搬移函数和以多态取代条件表达式等，重构的3个重要的节点，讲原函数分解成一组嵌套的函数，应用拆分阶段分离计算逻辑与输出格式化逻辑，以及为计算器引入多态性来处理计算逻辑，每一步都给代码添加了更多的结构，以便我能更好地表达代码的意图，重构的用途不仅是让代码能够工作，还能更轻松地被别人理解和修改。

好代码的校验标准就是人们是否能轻而易举地修改它。

重构的原则：

重构是对软件内部结构的一种调整，目的是在不改变软件可观察行为的前提下，提高其可理解性，降低其修改成本。重构与性能优化有很多相似之处：两者都需要修改代码，并且两者都不会改变程序的整体功能，两者差别在于其目的，重构是为了让代码“更容易理解，更易于修改”，这可能使程序运行得更快，也可能使程序运行得更慢，在性能优化时，我只关心让程序运行得更快，最终得到得代码有可能更难理解和维护，对此我有心理准备。

开发时需要遵守的准则：

* 添加新功能时，不应该修改既有代码，只管添加新功能并通过测试。
* 重构时不再添加新功能，只管改进程序结构，并通过已有测试。

重构目标：

* 持续纠偏和改进软件设计
* 帮助发现隐藏的代码缺陷
* 提高编程效率
* 提高代码可读性
* 规范数据的统一性
* 提高调试程序的效率

重构时机：

* 三次法则—事不过三，三则重构
* 添加功能时重构
* 修补错误时重构
* 复审代码时重构

重构禁忌：

* 既有代码太混乱，且不能正常工作，需要重写而不是重构
* 项目接近最后期限后，应该避免重构

重构技巧：

* 提炼重复代码
* 分割冗长方法
* 优化嵌套条件分支
* 提取继承类中重复的属性与方法到父类
* 清理冗赘类及巨类

重构思想：

* 架构
* 界面
* 实现
* 数据

重构中的间接层----间接层的作用：

* 允许逻辑共享（避免重复代码）
* 分开解释意图和实现（方法越短小，越容易起好名字揭示意图，单一职责）
* 隔离变化（软件需要根据需求的变化不断修改，隔离缩小修改的范围）
* 封装条件逻辑（多态消息）

**小结：**

目前项目正在进行重构，重构的目标也是让以前框架变的更加容易理解，架构更清晰，条理更明了，VM层重构，是解决之前书籍get，set方法较多，数据格式混乱，导致经常会出现数据格式问题，导致的异常。

以上便是单值控件的重构，本次重构，也涉及到协议的抽离，协议抽离目的：

1、统一并分类之前众多繁杂的取值赋值方法 比如七八个的setViewData / getViewData。

2、充分利用interface的default能力去执行默认行为，提高抽象。

3、控件根据控件遵循的协议即可了解该控件具备什么能力。

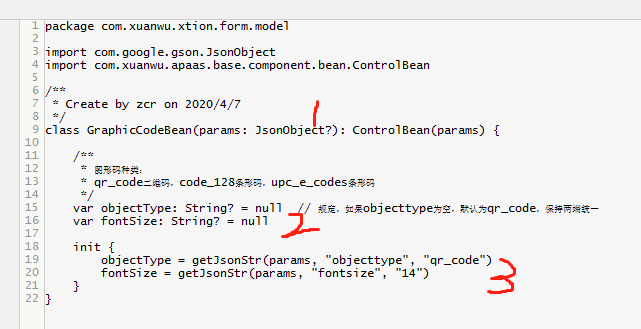
4、引擎层对控件的操作，不再通过instanceOf XXViewModel 来进去直接处理，而是判断控件是否遵循XX interface，方便于后续拓展。

5、降低基类庞大臃肿的各类函数，目的是让基类成为稳定不变的代码，新增的功能不再是通过在基类增加而是让控件遵循协议。

此外项目重构，也是对代码规范的一种习惯的养成，也是自身学习和提高的重要途经，因为在重构的同时，需要阅读别人的代码，这也是学习的好手段。学习别人的良好代码，编程思想，对自己也是一种启发，平时我们学习编程思想，其实别人的编程思想就是对自己编程习惯的一种总结，好的编程习惯，就是一种指引。当然也是需要不断地在平时编程中去总结和思考的，能够形成自己的一套正确的，规范的思想。在项目中能够学习大佬们的优秀代码编写技巧，能够得到大佬们的指导，并能够从中学到知识，也是一种收获。在大佬给我的代码review中，让我更能体会代码的规范的重要性。

以下便是第一次代码review时遇到的问题：

1.



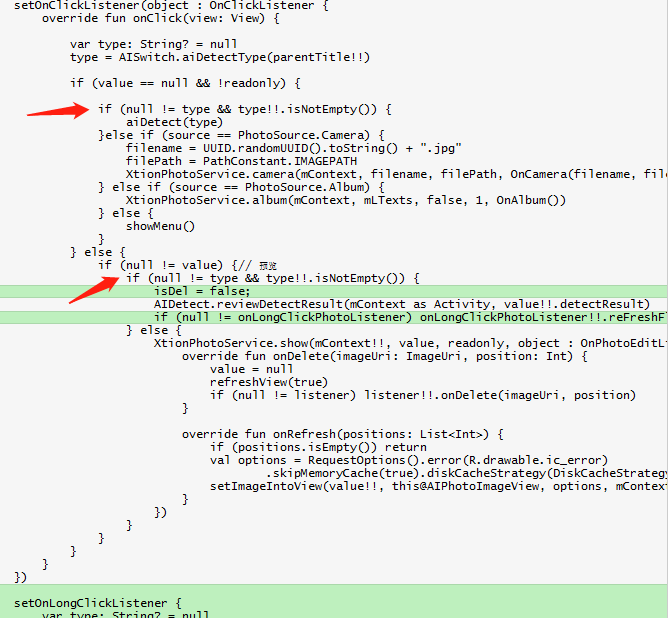
问题点？

1.params必定不为空，？不需要

2 、3已经明确对变量初始化，var和？都是不需要

分析: 方法内的参数可空与否，需要根据上下文来判断，java转kotlin时，默认会加上?，转完之后，需要分析上下文。

2.

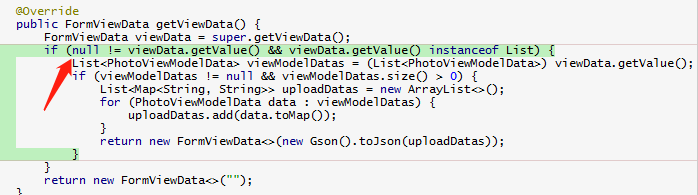


问题：

1.使用value?.let{  }   对value进行拆包，即可判空，又可减少{}嵌套

分析：如果变量可能为空，尽量做到对变量进行分包，简化代码

3.



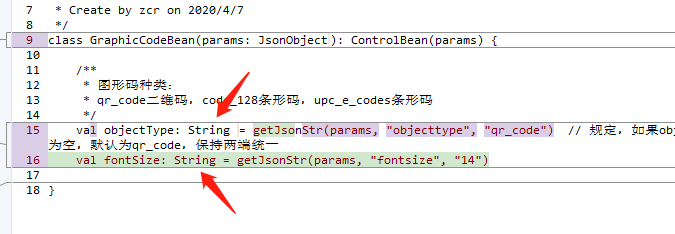
问题：

1.if(null == viewData.getValue() || !(viewData.getValue() instanceof List)) return

逆向判断，尽早返回

分析：如果涉及变量判空，可以对空变量做尽早退出的操作，但多线程中需要注意这样的操作，可能再多线程中，存在某一个时间点，即走了判空，又走了非空执行逻辑，导致走了非空逻辑报异常，可以通过在方法中加锁

4.



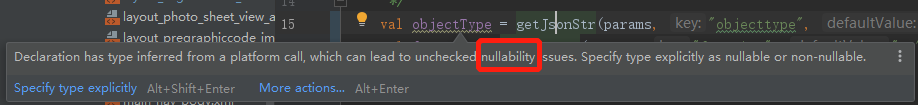
问题点？

这里String不用写了 编译器能根据getJsonStr做出推断

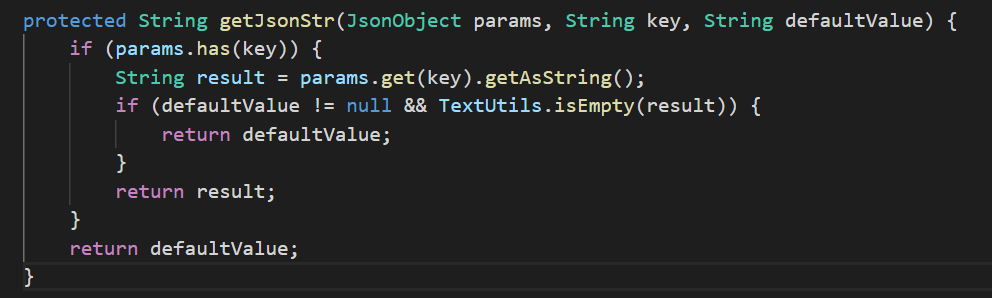
分析：这里getJsonStr()返回类型为String，这里初始化

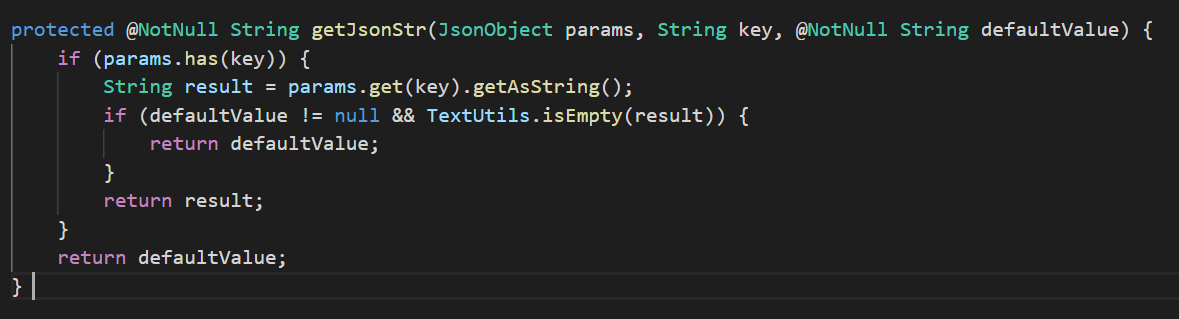
val objectType = getJsonStr(params, "objecttype", "qr\_code") val fontSize = getJsonStr(params, "fontsize", "14")

但是如果改为如下，会报提示，原因是调用getJsonStr()方法时，getJsonStr()方法返回值并没有判空



getJsonStr()方法源码如下：

修改getJsonStr()方法如下，加入@NotNull标识，标志改方法不会返回空值



最后：代码重构过程是一个不断学习，思考和总结的过程，也是自身提高的历程，在平时的重构中，多总结，多思考，多学习，多阅读别人优秀的代码。