**第6章-投保试算-保障类**

**学习目标**

* 掌握并能阐述投保试算的整个业务处理流程
* 掌握策略模式和门面模式在投保试算功能中的应用

**1、投保试算分析**

**1.1、需求分析**

在app端的产品详情页面中，用户可以根据自身的实际需要选择适合自己的方案、系数。而投保试算就是用户按照自己的需求选择对应的保险产品后，系统给出本次需要缴纳的保险费用是多少。简单来说就是要**交多少钱**。

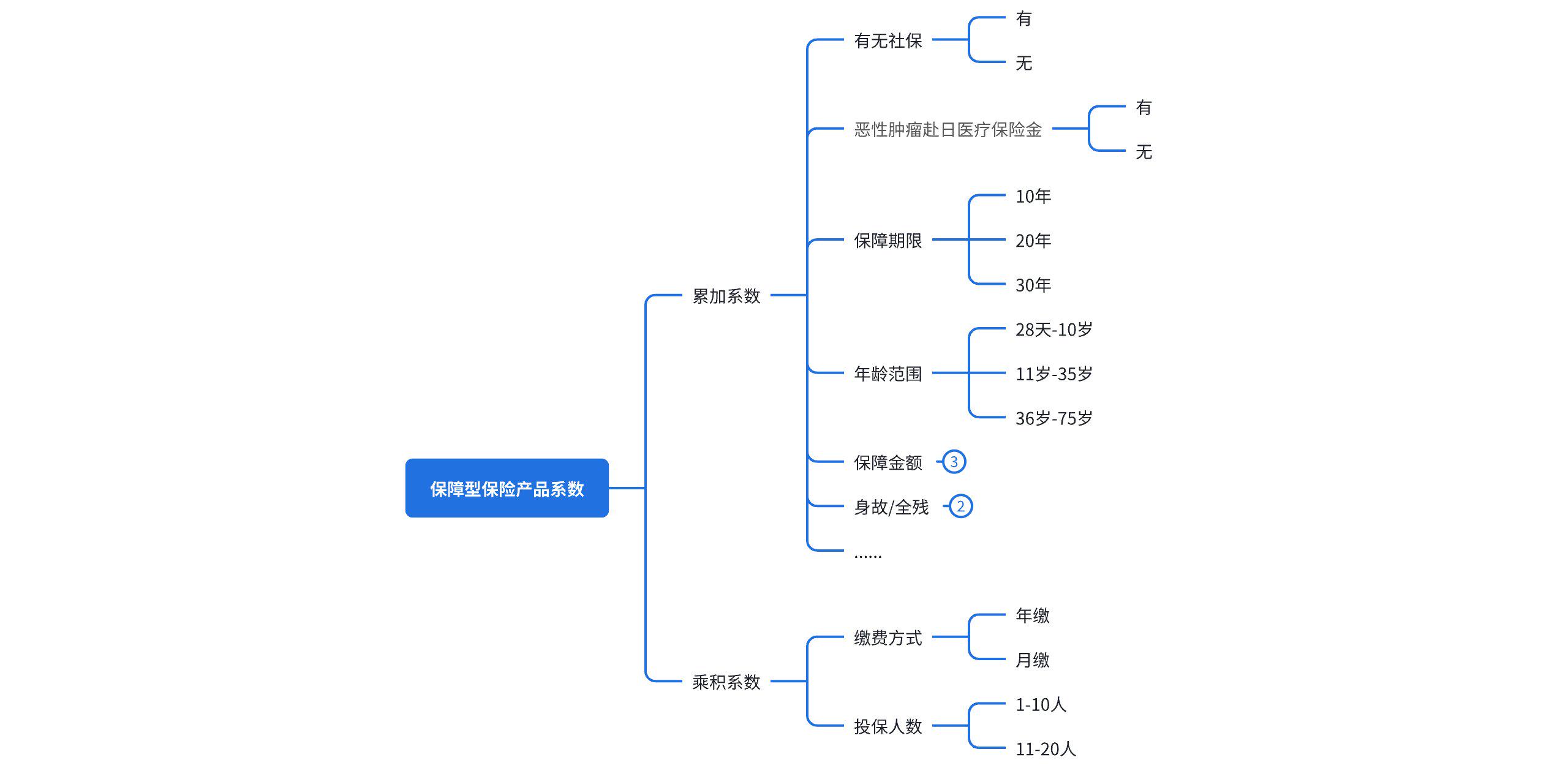
如下图：



观察发现：同一个保险选择不同的系数：有无保险得到的价格不同，也就是保险产品系数通常是根据保险产品中规定的条件和参数计算的，以确定最终的保险费。这些系数在保险发布时我们会定义各个系数计算值,如下图所示：



而这些系数在计算时也是有不同的处理方式，我们分为：累加系数、乘积系数（哪些是累加、乘积要遵循行业标准、产品要求），下图中罗列了一部分：

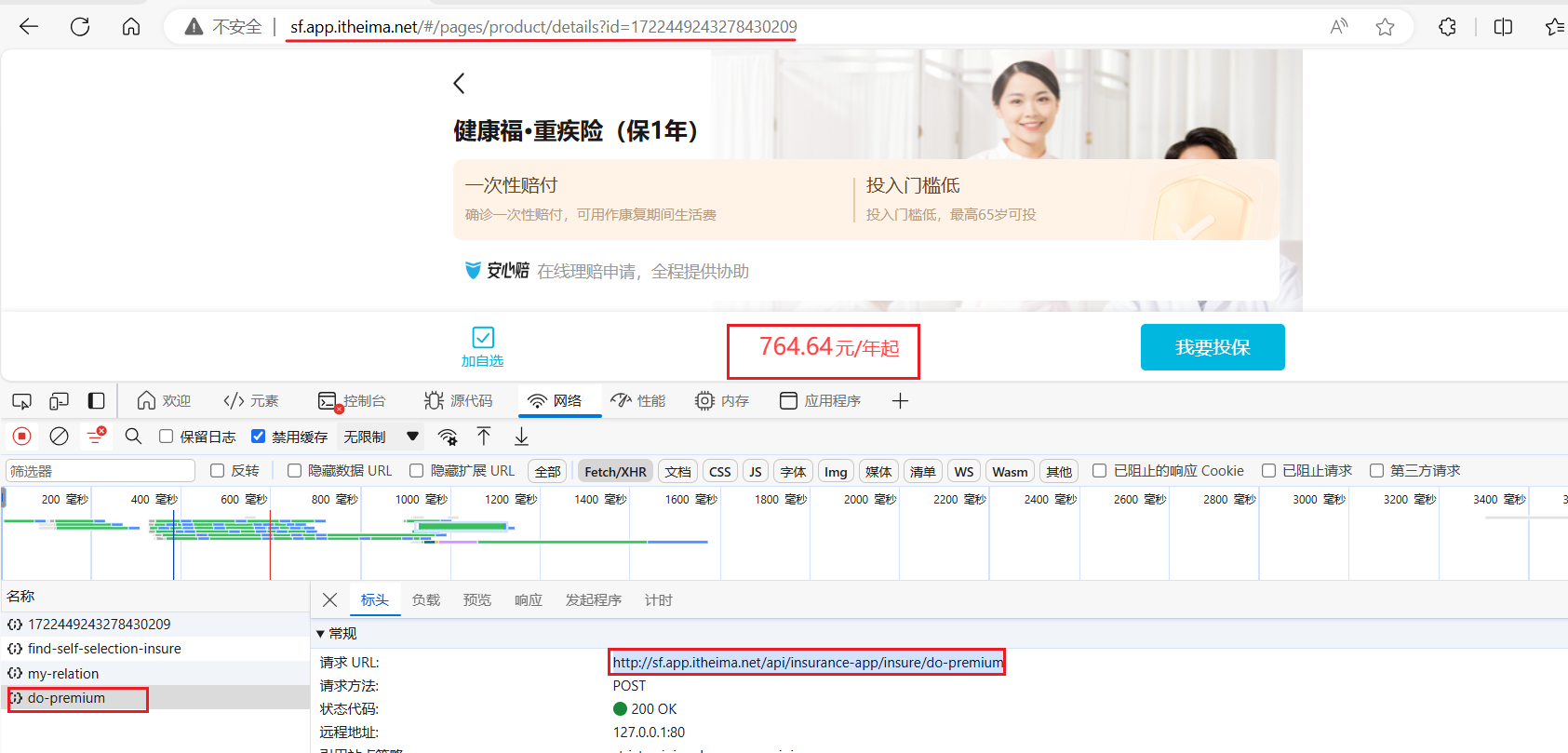


也正是因为这些系数的组合构建出了复杂的保险产品，那这些系数如何使用按照各个保险产品的特性；由产品经理或者需求方制定出如下两个公式：

|  |
| --- |
| 个险类保险保费：  最终投保价格=起步价格X乘积系数 X（1+累加系数合值【累加系数1+累加系数2+累加系数3+....+累加系数N】）  团险类保险保费：  最终投保价格=起步价格X乘积系数 X（1+累加系数合值【累加系数1+累加系数1+累加系数2+....+累加系数N】）X *投保人数* |

**1.2、接口分析**

在app端产品详情页面中，选择了不同的系数之后会发起保费价格的计算；如图：



Swagger接口地址：http://localhost:7065/doc.html#/default/%E4%BF%9D%E9%99%A9%E6%8A%95%E4%BF%9D/doPremiumUsingPOST

建议：从浏览器中跟踪接口信息如下：

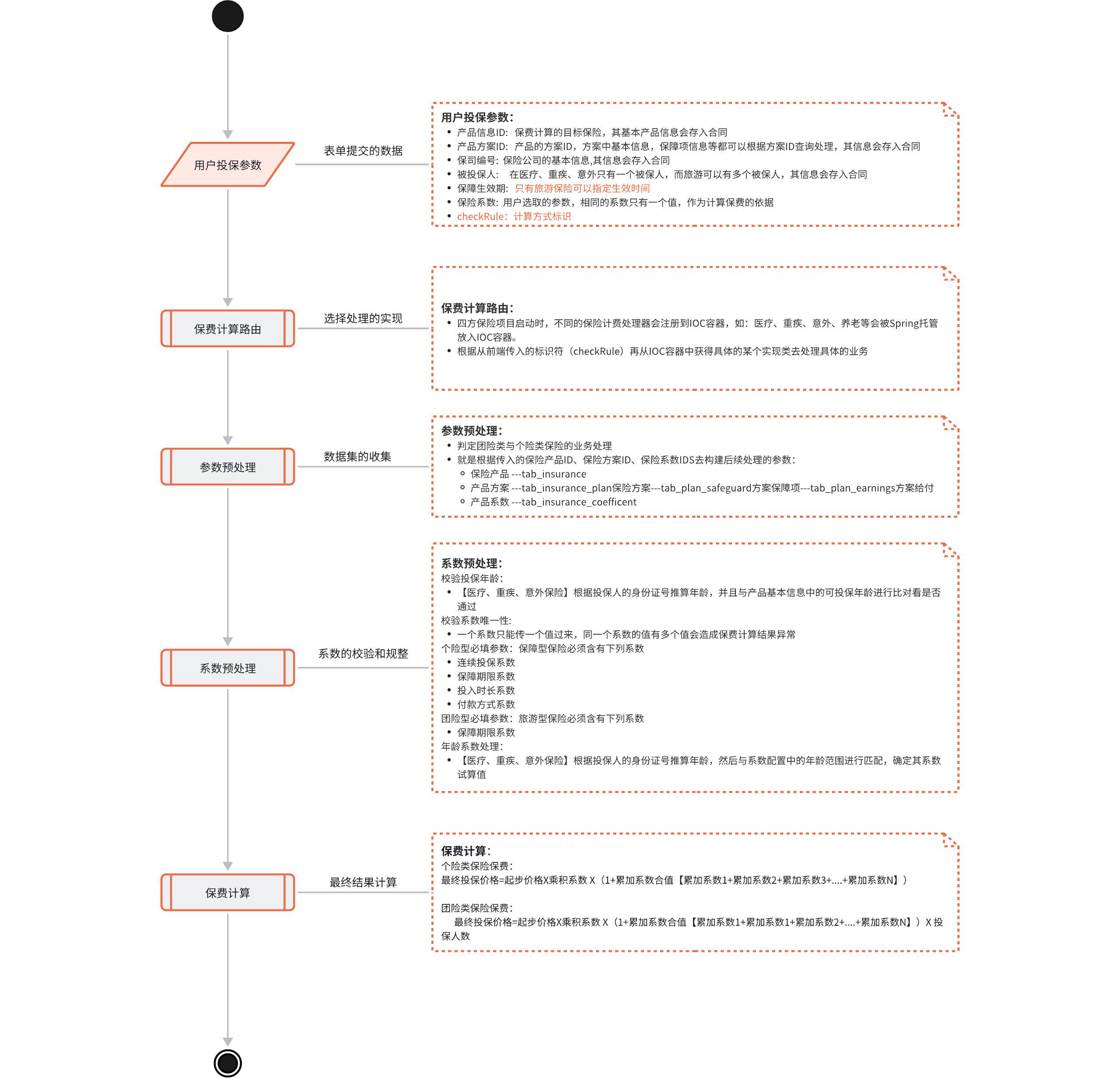
|  |  |
| --- | --- |
| 接口描述 | 说明 |
| **请求地址** | http://sf.app.itheima.net/api/insurance-app/insure/do-premium |
| **请求方式** | POST |
| **请求参数** | {  "checkRule": "1", # 路由key；*0医疗 1重疾 2意外 3养老 4储蓄 5旅游*  "companyNo": "100000001", # 企业编号  "insuranceId": "1722449243278430209", # 保险产品id  "insuranceCoefficentIds": [ # 保险系数项id集合  "1726792497633431553",  "1726792497666985985",  "1726792497650208770",  "1726792497658597381",  "1726792497658597383"  ],  "insurancePlanId": "1727575594796875778", #保险方案id  "customerRelationIds": [ # 被投保人id集合  "1225092536525656054"  ]  } |
| **响应结果** | {  "code": 200,  "msg": "操作成功",  "data": "764.64", # 保费金额；这也是投保试算出的价格  "operatorId": 1371500419615895666,  "operatorName": "cutomer@qq.com",  "operatorSex": "0",  "\_class": "java.lang.String",  "tip": "本站点所有接口仅用于教学演示，所有数据均非真实数据，请勿用作其它用途！",  "operationTime": "2024-05-23 11:03:50"  } |

**1.3、计算流程**

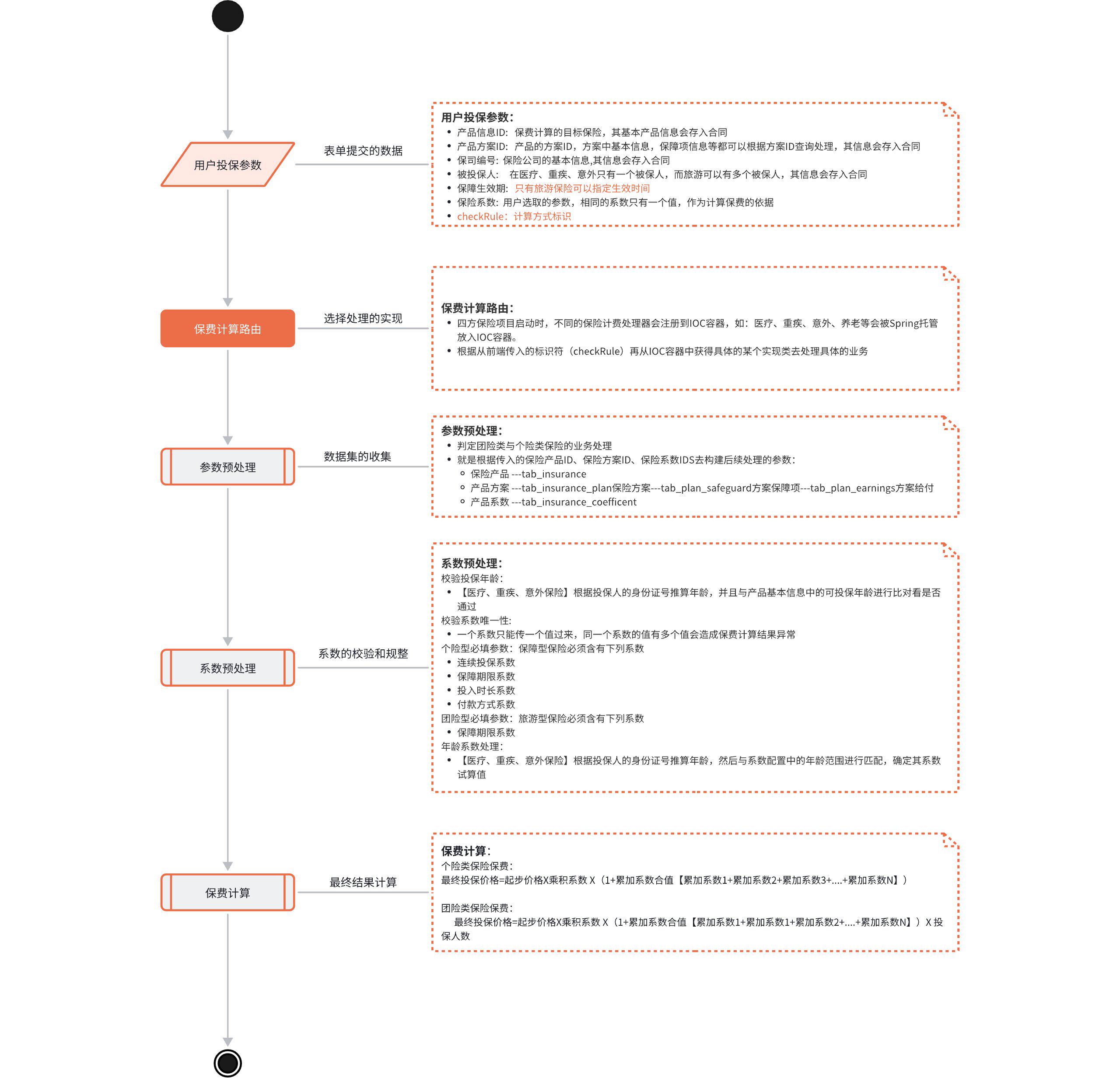
在分析了界面中保费的入口、影响费用的因素之后；那么计算出最终的费用的话；主要有如下步骤：

1. **保费计算策略**；不同的保险产品，它的要求、计算都可能不同；要根据前端传递的参数确定使用哪类计算方式来处理；
2. **参数预处理**；校验用户填写的参数是否合法；
3. **系数预处理**；校验系数，某些隐藏的系数获取；例如：投保人数系数、年龄范围系数处理；
4. **保费计算**；按照不同类型保费的计算公式，算出一个最终保险的保费

通过如下图比较完整描述一下整个保费计算的流程：



**2、保费试算策略**



**2.1、策略模式**

**2.1.1、介绍**

**策略模式**是一种行为设计模式，它能在运行时改变对象的行为。在策略模式中，一个类的行为或其算法可以在运行时更改。这种类型的设计模式属于行为模式。

**应用场景**

1. **多个相关类只区别在表现行为不同**：当一个系统需要动态地在几种算法或行为中选择一种时。
2. **避免使用多重条件选择语句（如if-else或switch-case）**：如果在一个系统中有很多条件判断语句，可以通过将这些判断分支封装成不同的策略类来避免代码的复杂性。
3. **可互换的算法**：程序需要支持多种算法，并且这些算法可以在运行时互换。

**为什么使用策略模式？**

* **提高灵活性**：通过将算法封装在独立的类中，可以轻松地添加、移除或修改算法，而无需修改使用它们的类。
* **易于扩展**：增加新的策略只需实现接口即可，符合开闭原则。
* **简化复杂逻辑**：将复杂的条件判断逻辑分散到各个策略类中，使得主逻辑更加清晰易懂。

**设计思想**

* 编写父类策略接口，定义行为方法
* 根据需要编写N个具体的策略接口的实现类
* 初始化 实现类到容器中（一般IOC容器）
* 在应用时根据标识符从容器中获取对应的处理器实例处理业务接口

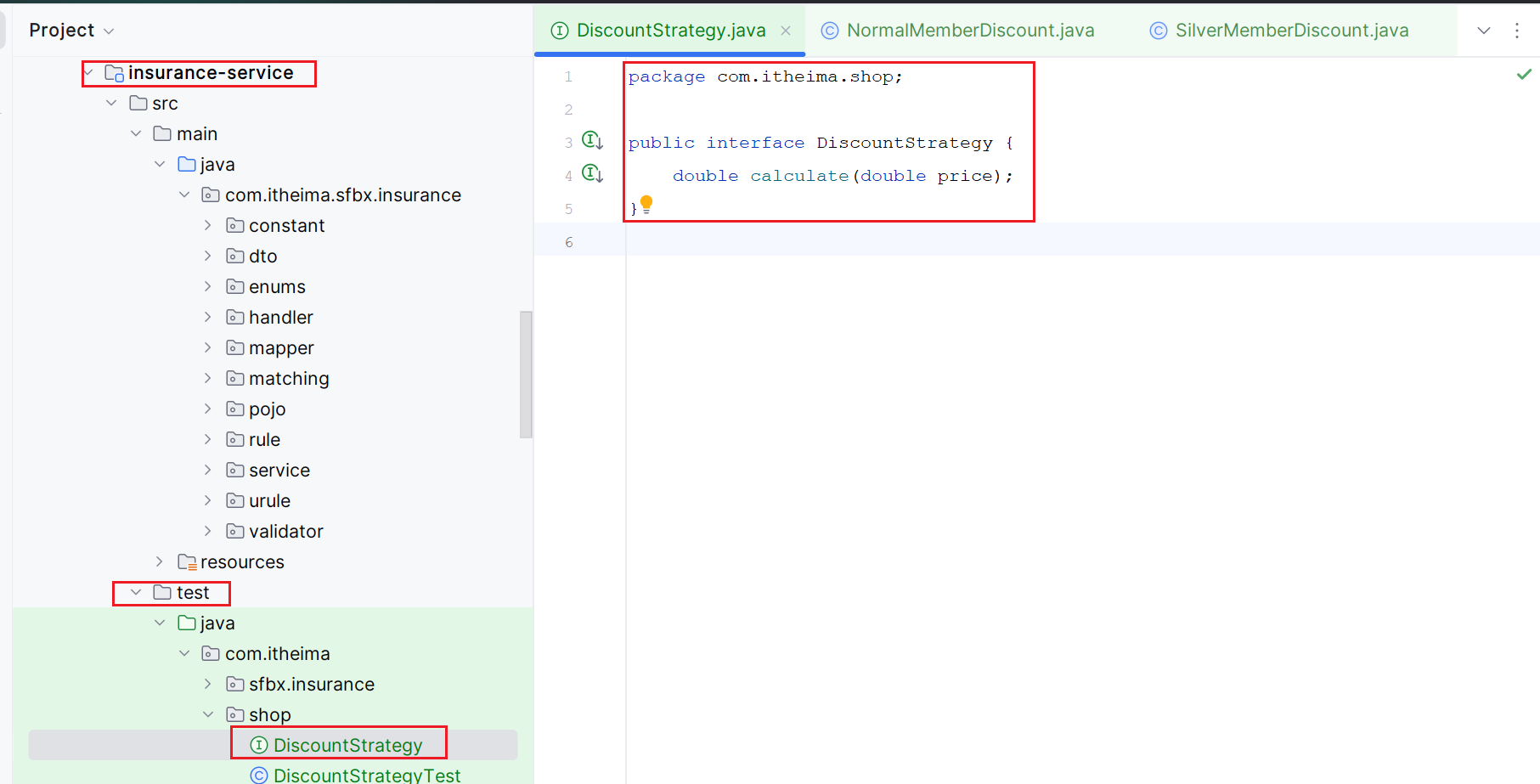
**2.1.2、入门**

假设在线购物平台，需要根据用户的不同会员等级提供不同的折扣策略。这里就可以使用策略模式来设计。

**1）定义策略接口**

编写 sfbx-cloud\sfbx-insurance\insurance-service\src\test\java\com\itheima\shop\DiscountStrategy.java

|  |
| --- |
| Java package com.itheima.shop;  public interface DiscountStrategy {  double calculate(double price); } |



**2）实现策略类**

* 普通会员折扣类 com.itheima.shop.NormalMemberDiscount

|  |
| --- |
| Java package com.itheima.shop;  public class NormalMemberDiscount implements DiscountStrategy{  @Override  public double calculate(double price) {  *//普通会员折扣* return price \* 0.95;  } } |

* 银牌会员折扣类 com.itheima.shop.SilverMemberDiscount

|  |
| --- |
| Java package com.itheima.shop;  public class SilverMemberDiscount implements DiscountStrategy{  @Override  public double calculate(double price) {  *//普通会员折扣* return price \* 0.9;  } } |

* 金牌会员折扣类 com.itheima.shop.GoldMemberDiscount

|  |
| --- |
| Java package com.itheima.shop;  public class GoldMemberDiscount implements DiscountStrategy{  @Override  public double calculate(double price) {  *//普通会员折扣* return price \* 0.85;  } } |

**3）策略工厂（初始化）**



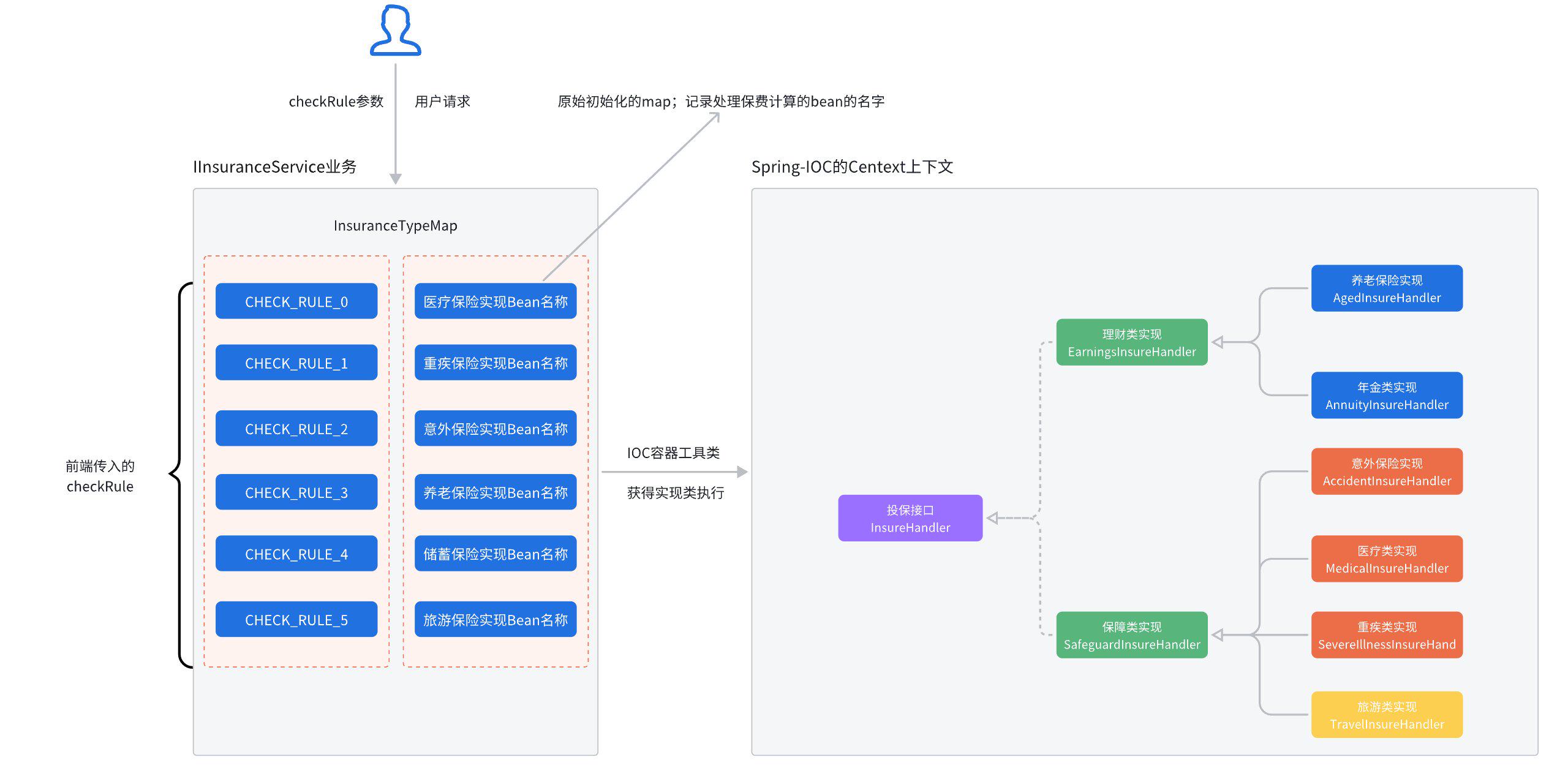
**4）应用策略**

com.itheima.shop.DiscountStrategyTest 内容如下：

|  |
| --- |
| Java package com.itheima.shop;  import cn.hutool.extra.mail.Mail;  import java.util.HashMap; import java.util.Map; import java.util.Scanner;  public class DiscountStrategyTest {   static Map<String, DiscountStrategy> *strategyMap* = new HashMap<String, DiscountStrategy>();  *//初始化策略map* static {  *strategyMap*.put("normal", new NormalMemberDiscount());  *strategyMap*.put("silver", new SilverMemberDiscount());  *strategyMap*.put("gold", new GoldMemberDiscount());  }   public static void main(String[] args) {  while(true) {  *//输入会员类型* System.*out*.println("请输入会员类型：normal/silver/gold");  *//获取从键盘输入的内容类型* Scanner scanner = new Scanner(System.*in*);  String memberType = scanner.nextLine();  System.*out*.println("请输入商品价格：");  double price = scanner.nextDouble();  DiscountStrategy discountStrategy = *strategyMap*.get(memberType);  System.*out*.println("折扣价为：" + discountStrategy.calculate(price));  }  } } |

**2.2、保费试算策略**

在四方保险中不同类型的产品，对应不同的保费计算；所以针对这些类型都设计了保费计算处理类InsureHandler；并应用策略模式初始化到map中；再根据前端传入的标识符从map中获取对应的处理器进行保费计算。如下图：



* **项目启动阶段：**

项目启动时，InsureHandler投保处理接口实现如：医疗、重疾、意外、旅游、养老、年金等会被Spring托管放入IOC容器。

静态变量InsuranceTypeMap被构建，并设置insuranceTypeMap中的内容：K：checkRule参数，V：投保实现类bean名称

* **业务请求阶段：**

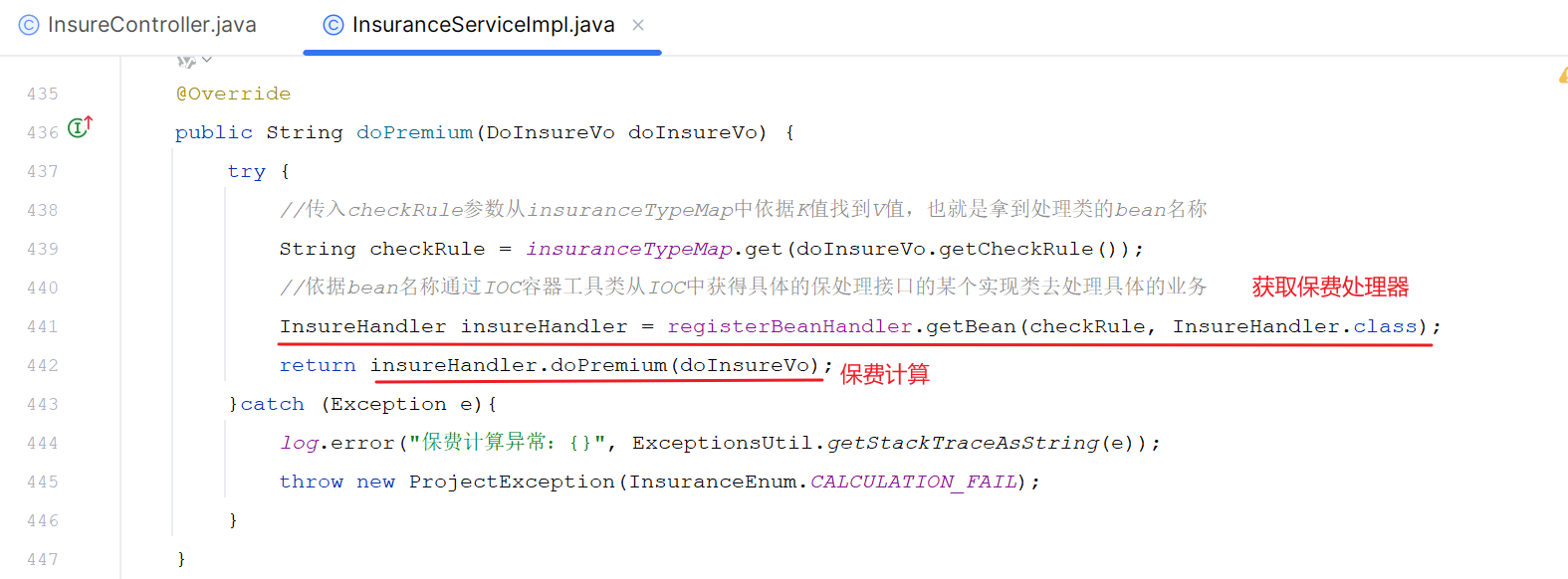
传入checkRule参数从insuranceTypeMap中依据K值找到V值，也就是拿到处理类的bean名称

依据bean名称通过IOC容器工具类从IOC中获得具体的保处理接口的某个实现类去处理具体的业务

具体代码中跟踪如下：



业务方法中；应用策略模式获取对应的保费处理器：



**2.3、补充需求**

既然已经了解和学习了策略模式；那么在已有的项目中如何对策略模式加以善用？请完成如下：

**需求**：在试算的时候；如果是 意外险 的话；那么保司对职业是 杀手 的人不进行保险，对应在《四方保险》中的话；判断被保人的月收入如果高于100万或者没有填写的人；不能投保意外险。

**分析**：

* 意外险的checkRule是2，对应的保费计算处理类是：accidentInsureHandler
* 在处理意外险保费计算的时候；提前对用户的收入进行判断
* 月收入 高于100万或者没有填写的人直接抛出异常
* 月收入 符合要求的则正常进行保费计算，保费计算直接沿用父类中方法即可

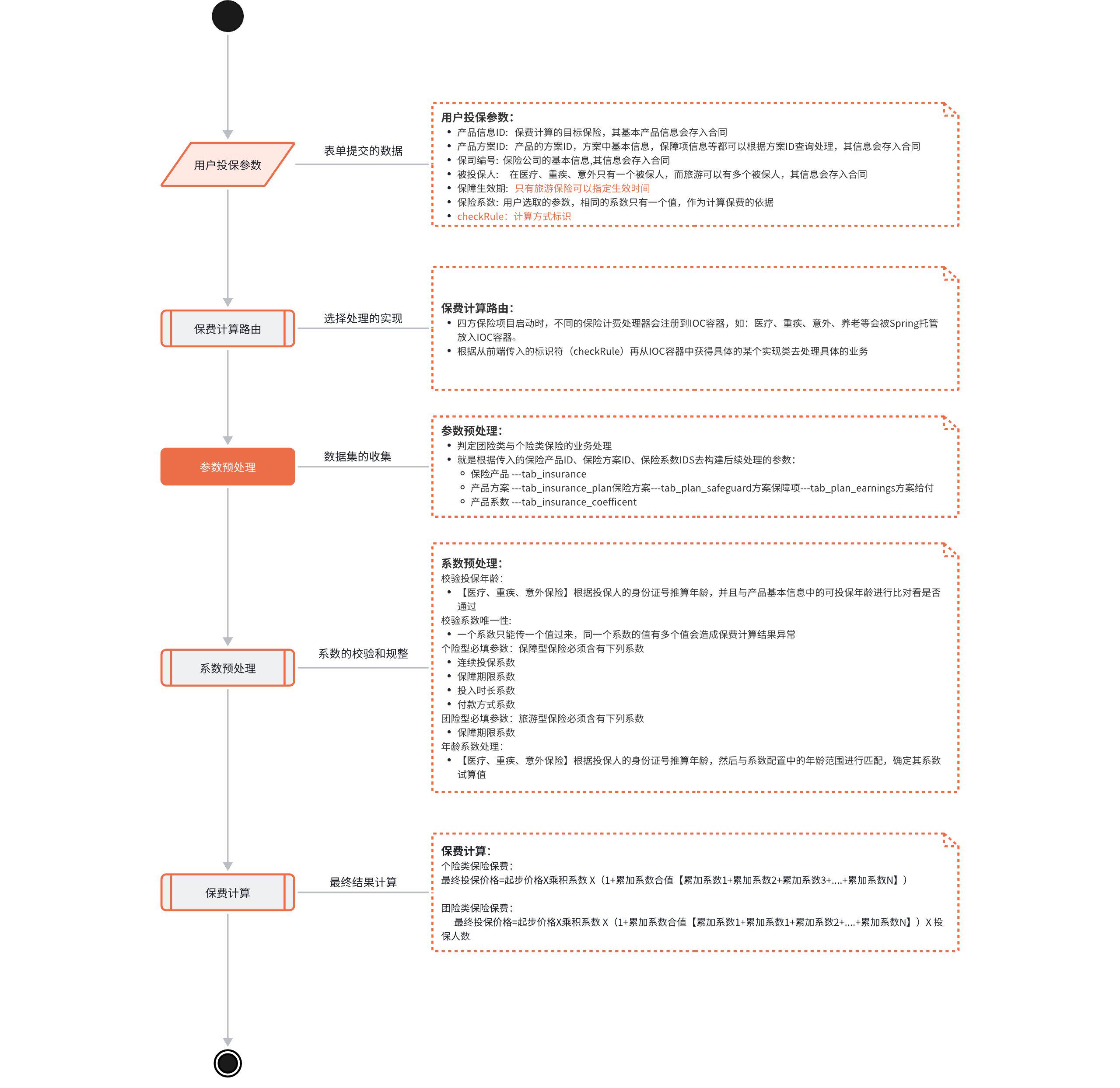
**实现**：

重写 sfbx-cloud\sfbx-insurance\insurance-service\src\main\java\com\itheima\sfbx\insurance\handler\impl\AccidentInsureHandler.java 父类中关于保费计算的方法：

|  |
| --- |
| Java @Override public String doPremium(DoInsureVo doInsureVo) {  *//查询被投保人* CustomerRelationVO relationVO = customerRelationService.findById(doInsureVo.getCustomerRelationIds().get(0));  if (relationVO != null) {  BigDecimal income = relationVO.getIncome();  *//高于100万或者没有填写的人直接抛出异常* if (income == null || income.compareTo(new BigDecimal(1000000)) > 0) {  throw new RuntimeException("杀手不能投保意外险！");  }  }   return super.doPremium(doInsureVo); } |



**3、参数预处理**



**3.1、门面模式**

**3.1.1、介绍**

**门面模式**（Facade Pattern）也被称为外观模式，是一种结构型设计模式。它的主要目的是为子系统中的一组接口提供一个统一的、高层次的接口，从而使得子系统更加容易使用。门面模式的核心思想是降低系统的复杂性，通过隐藏子系统的复杂性，只向客户端暴露一个简单、易用的接口。

**应用场景**

1. **系统有多个子系统或模块，且各模块间存在复杂的交互**：门面模式可以提供一个简单的接口，隐藏这些复杂性，使得客户端不需要了解这些内部细节。
2. **需要简化复杂的API接口**：当一个系统提供了大量的接口，而客户端仅需使用其中一小部分功能时，可以通过门面模式提供一个简化的接口。
3. **提高系统可维护性和可测试性**：通过减少外部与子系统间的耦合，使得修改子系统内部结构不会影响到客户端，同时便于对子系统进行单元测试。

**3.1.2、入门**

假定一个用户到银行开户并且立即开通信用卡为例；来说明门面模式。

如果不使用门面模式的话：

sfbx-cloud\sfbx-insurance\insurance-service\src\test\java\com\itheima\bank\NoFacadeTest.java

|  |
| --- |
| Java package com.itheima.bank;  public class NoFacadeTest {   public static void main(String[] args) {  *//开户* System.*out*.println("开户... 10000行代码。。。");  *//认证* System.*out*.println("开户需要认证... 流程多复杂。。。");  *//开信用卡* System.*out*.println("开信用卡各种手续，流程判断... 20000行代码");   } } |

**使用门面模式后：**

1. 定义门面类和其它相关业务处理类或方法

sfbx-cloud\sfbx-insurance\insurance-service\src\test\java\com\itheima\bank\BankFacade.java

|  |
| --- |
| Java package com.itheima.bank;  public class BankFacade {   *//组合操作* public void openAccountAndCreditCard() {  openAccount();  oauth();  openCreditCard();  }   *//开户* public void openAccount() {  *//开户* System.*out*.println("开户... 10000行代码。。。");  }  *//认证* public void oauth() {  *//认证* System.*out*.println("开户需要认证... 流程多复杂。。。");  }  *//开信用卡* public void openCreditCard() {  *//开信用卡各种手续，流程判断... 20000行代码* System.*out*.println("开信用卡各种手续，流程判断... 20000行代码");  } } |



1. 客户端调用门面类

sfbx-cloud\sfbx-insurance\insurance-service\src\test\java\com\itheima\bank\FacadeTest.java

|  |
| --- |
| Java package com.itheima.bank;  public class FacadeTest {   public static void main(String[] args) {  *//开户和开信用卡；使用门面模式* BankFacade bankFacade = new BankFacade();  bankFacade.openAccountAndCreditCard();  } } |

通过门面模式，简化了接口调用；无需直接与复杂的子系统交互，使得整个银行服务系统的使用变得更加便捷和高效。

**3.2、参数预处理**

参数预处理就是根据产品id查询相关的所有数据；那产品关联的数据有哪些？我们通过概念模型直观的查看一下：

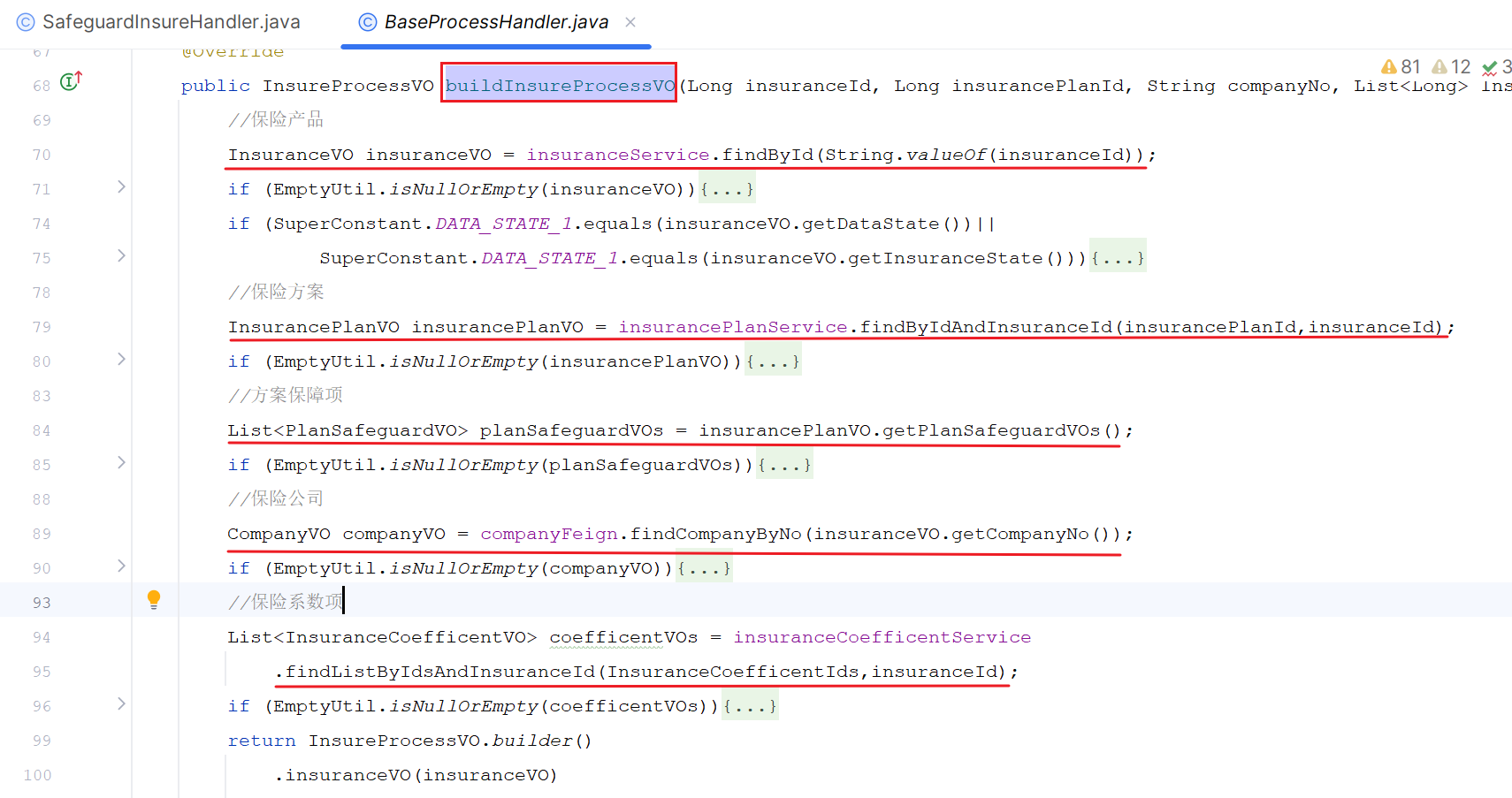


了解保险产品的关系之后；代码中查询如下：



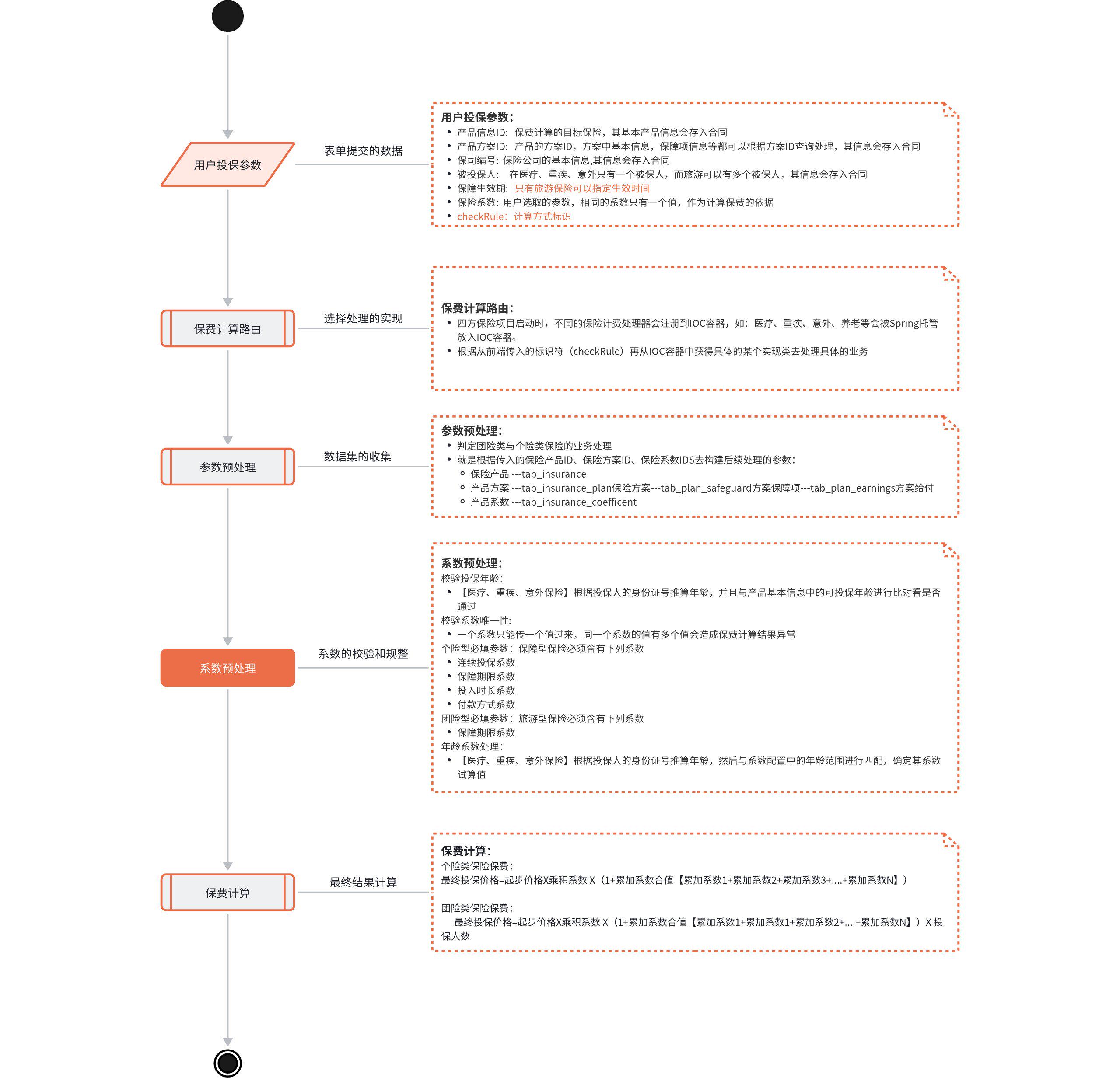
具体查询的代码：

com.itheima.sfbx.insurance.handler.impl.BaseProcessHandler#buildInsureProcessVO



从代码中我们可以看出；最终查询出来的结果全部都封装到了 InsureProcessVO 也就是；在实现过程中，查询到的数据都往这个返回结果中设置即可。

**4、系数预处理**

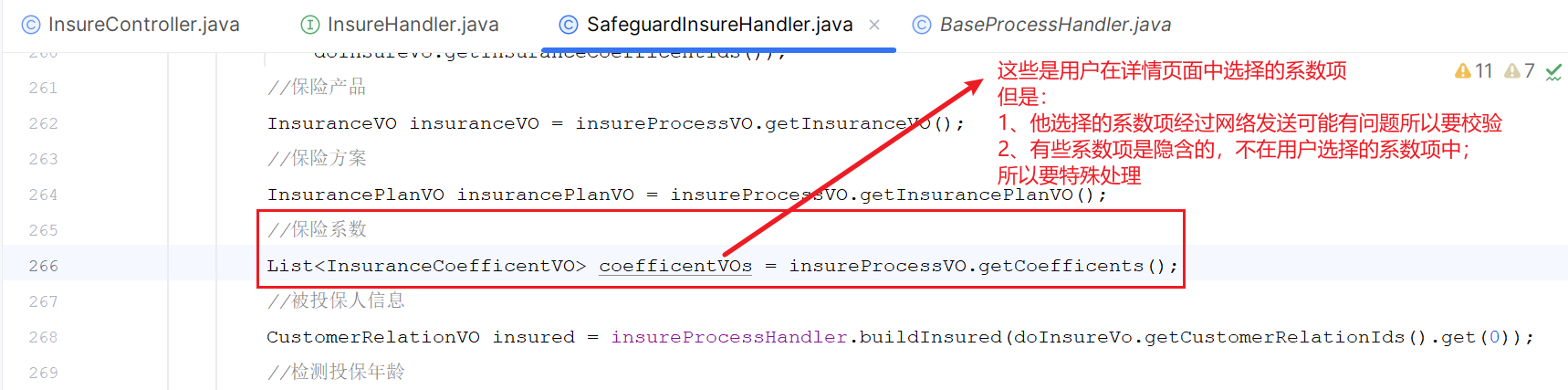


**4.1、功能分析**

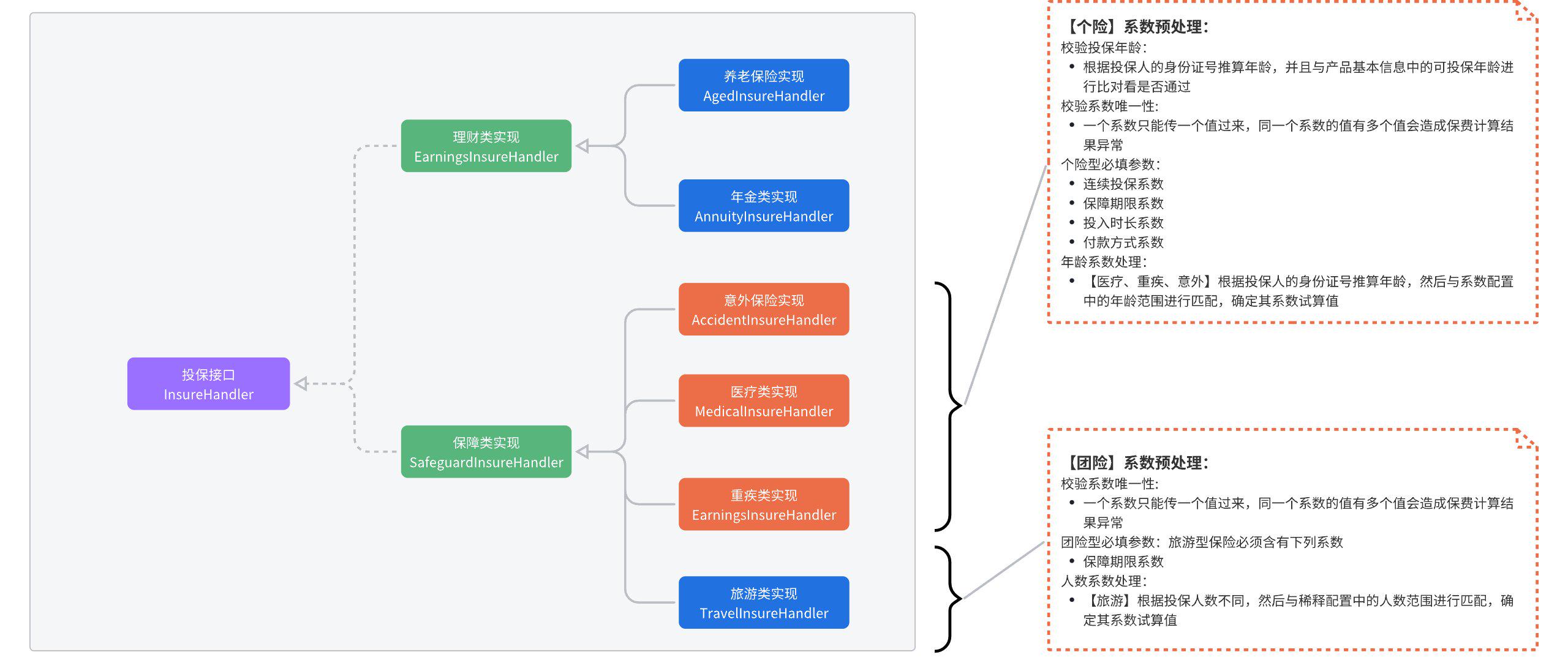
**目标**：

最终保费与各系数有关系；所以要确定下来详情页面中选择的系数项及隐含的一些系数获取并处理。

也就是说：



而不同类型的保险产品，它的系数校验与处理也有区别；以下图形方式说明一下：

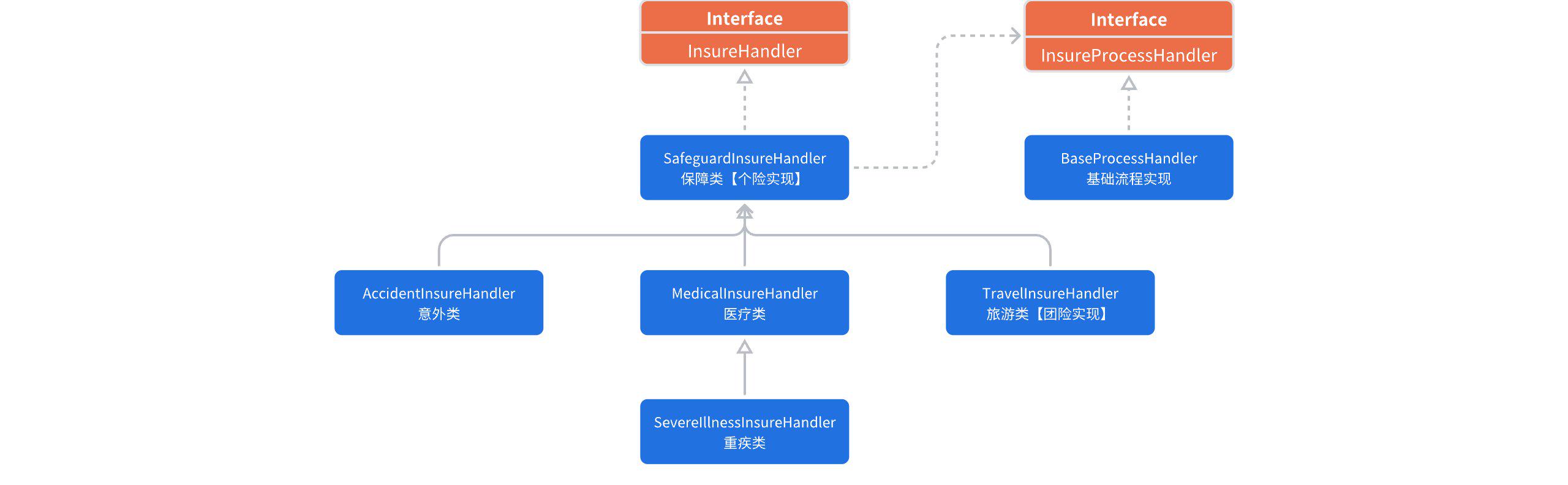


也就是说；每个类型的保险产品：

* 如果有公共校验、系数的获取、判断，那么可以在 父类（保障类）中进行；
* 如果有当前这个分类对应的保险特有的判断、系数的获取判断不同，那么可以在具体的保费子类处理器中实现

**4.2、个险系数**

个险系数处理：主要是在SafeguardInsureHandler中实现，而AccidentInsureHandler【意外】、MedicalInsureHandler【医疗】、SevereIllnessInsureHandler【重疾】继承其实现逻辑。它们的关系如下图所示



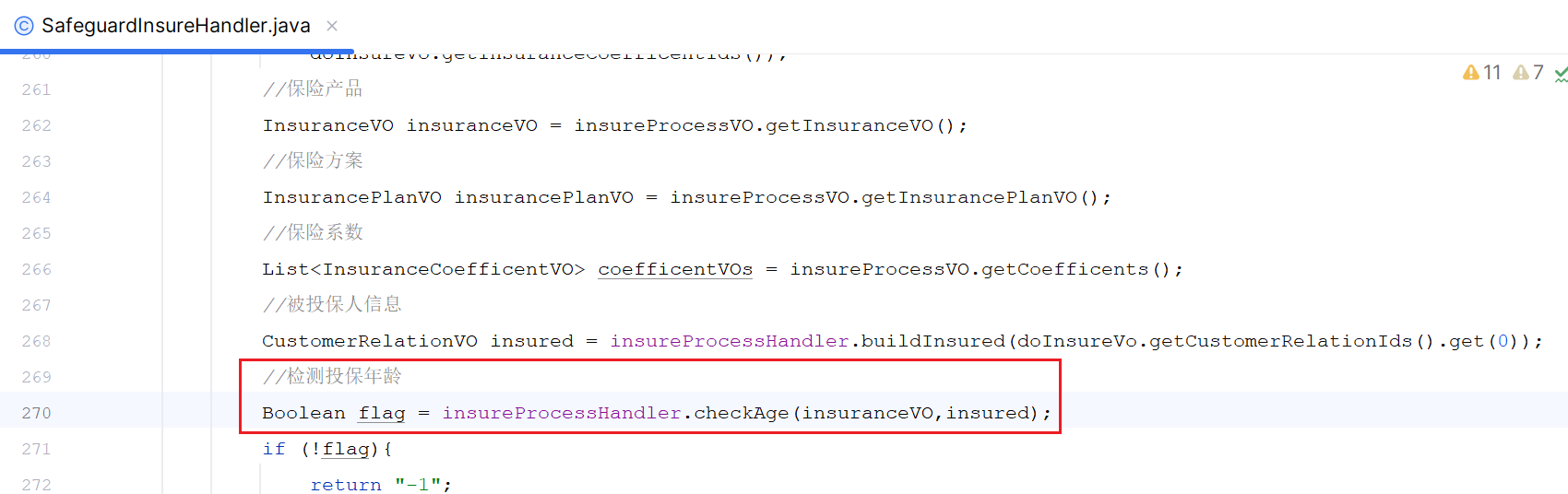
|  |
| --- |
| **【个险】系数预处理：对应的保险类型有意外、医疗、重疾**  **校验投保年龄**：   * 根据投保人的身份证号推算年龄，并且与产品基本信息中的可投保年龄进行比对看是否通过   **校验系数唯一性**:   * 一个系数只能传一个值过来，同一个系数的值有多个值会造成保费计算结果异常   个险型**必填参数**：   * 连续投保系数 * 保障期限系数 * 投入时长系数 * 付款方式系数   **年龄系数**处理：   * 【医疗、重疾、意外】根据投保人的身份证号推算年龄，然后与系数配置中的年龄范围进行匹配，确定其系数试算值 |

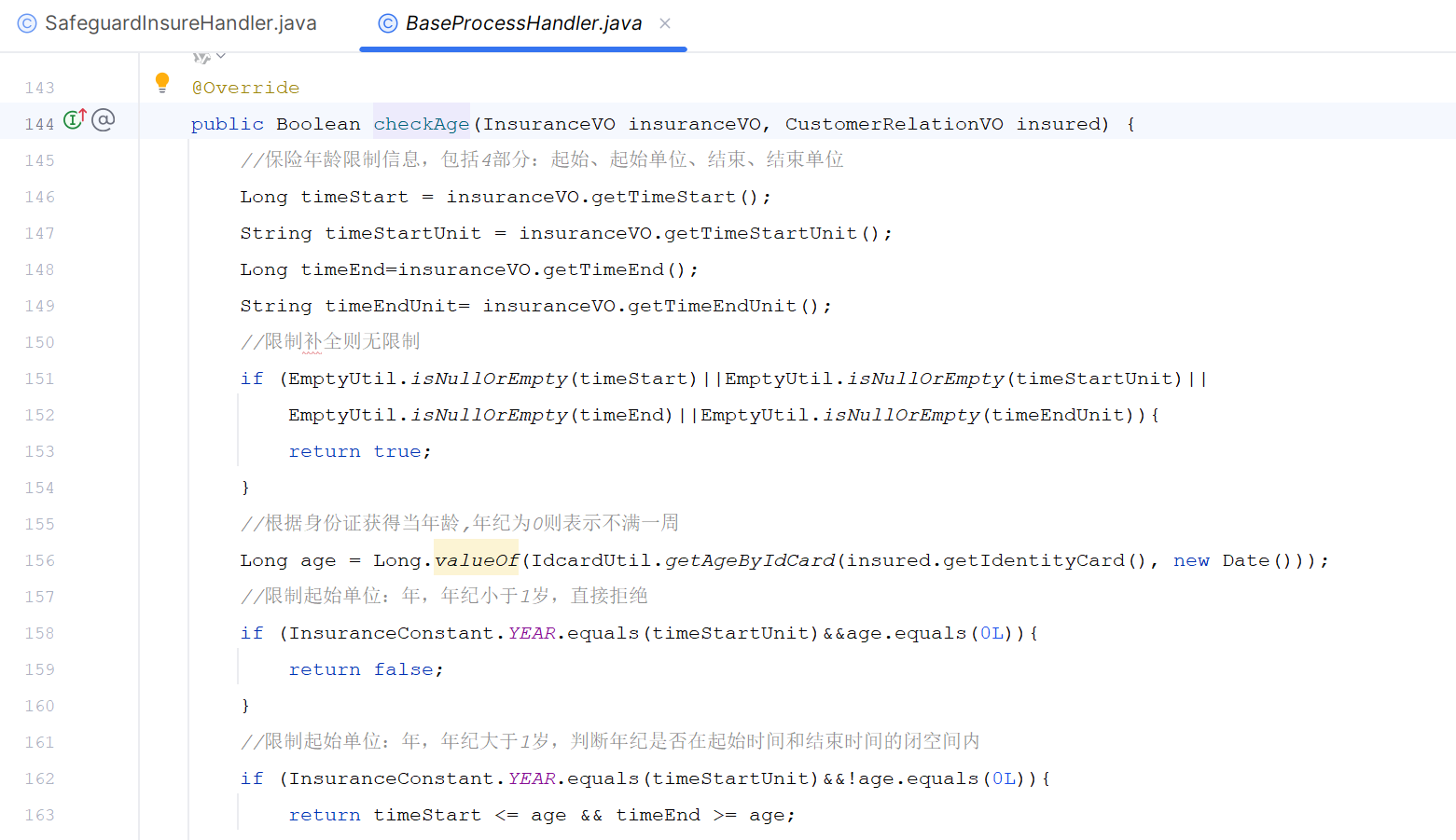
**1）校验投保年龄**

在添加保险产品的时候指定产品可以被投保的年龄范围。界面如下：



如下的校验皆是为了校验当前保险是否在年限范围内：





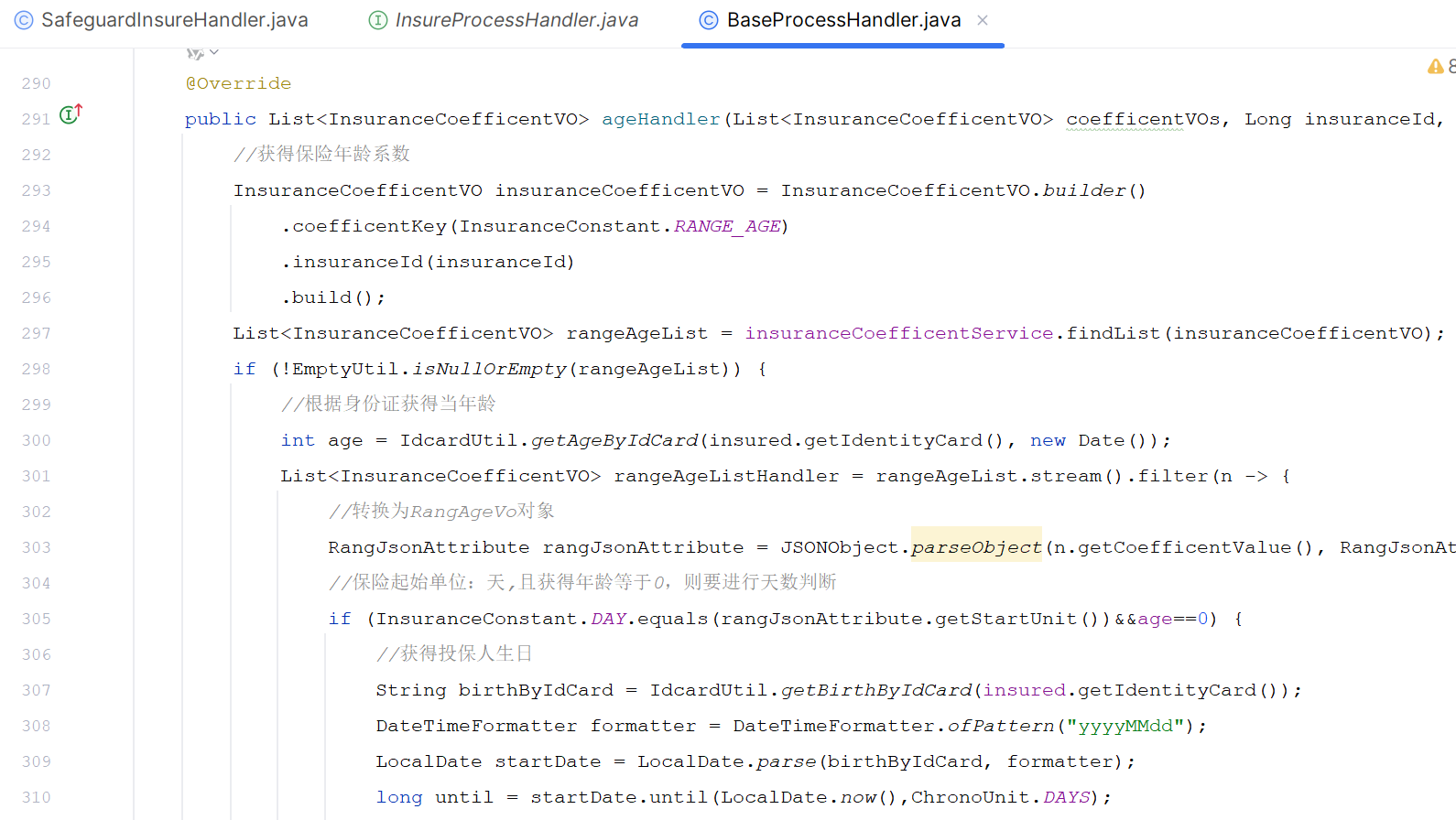
**2）投保年龄系数处理**

在保险产品添加的时候；可以设置产品系数项，里面有一个 年龄范围 可以设置不同的年龄范围对应的系数值不同：



如下的代码就是为了根据当前投保人的年龄来确定他使用哪个系数：



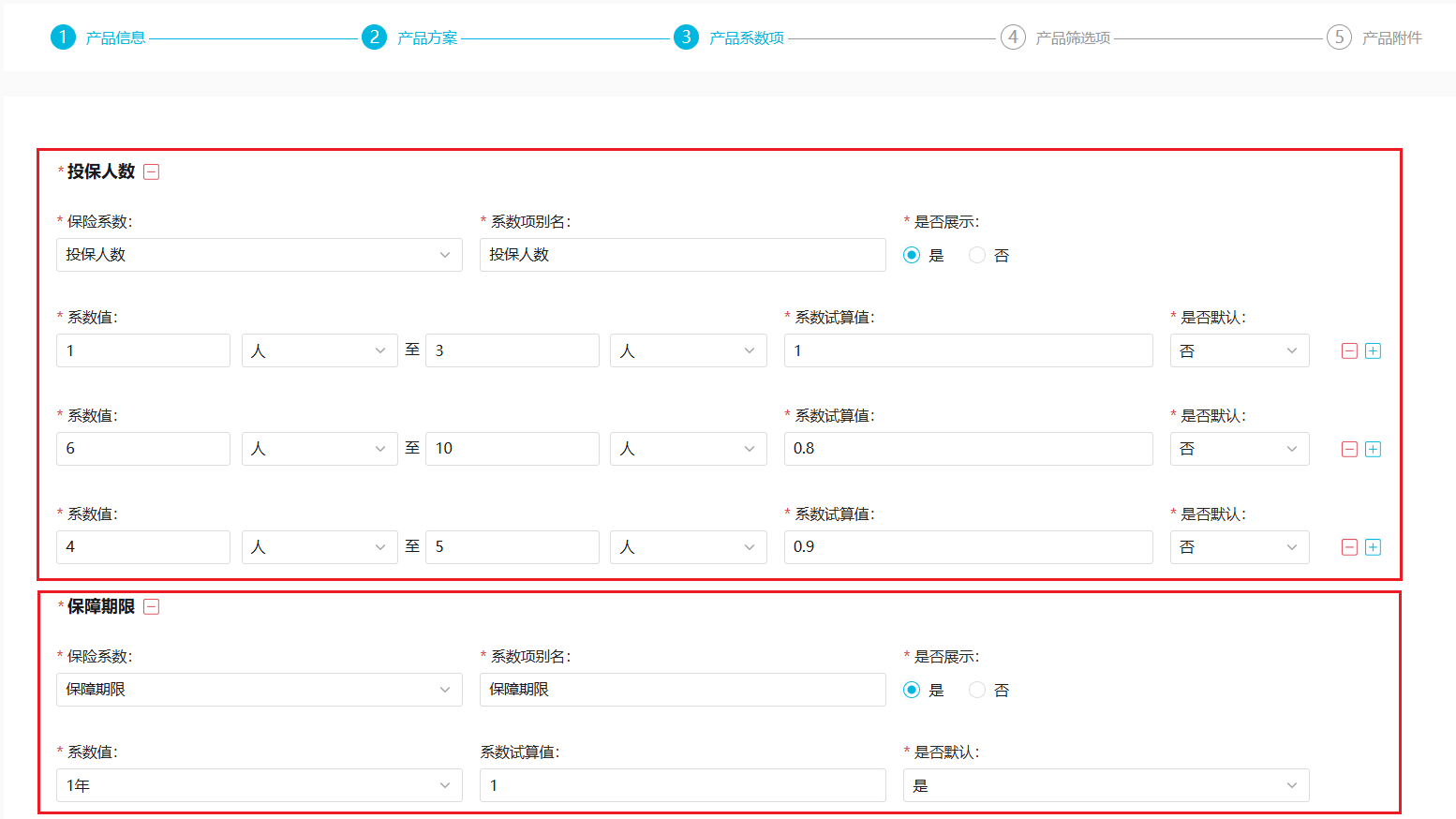


**4.3、团险系数**

团险系数属于旅游类保险；它比一般的保险多了如下的校验和系数：

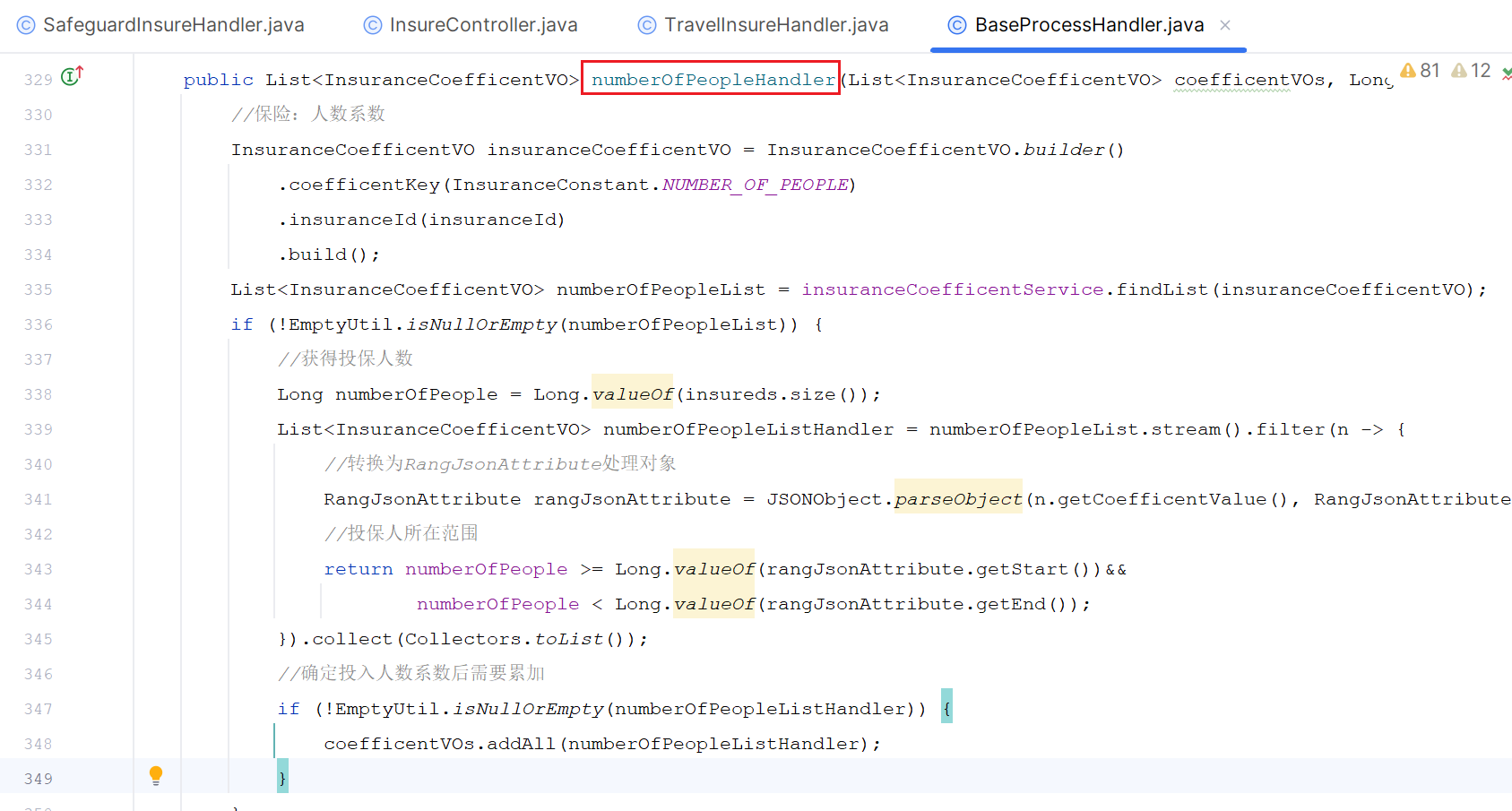
|  |
| --- |
| **【团险】系数预处理：对应的类型有旅游**  校验系数唯一性:   * 一个系数只能传一个值过来，同一个系数的值有多个值会造成保费计算结果异常   团险型必填参数：旅游型保险必须含有下列系数   * 保障期限系数   人数系数处理：   * 【旅游】根据投保人数不同，然后与系数配置中的人数范围进行匹配，确定其系数试算值 |

校验与其它保险类似；而保障期限、投保人数系数是团险独有；需要根据在产品添加时的系数设置来确定当前投保的系数：

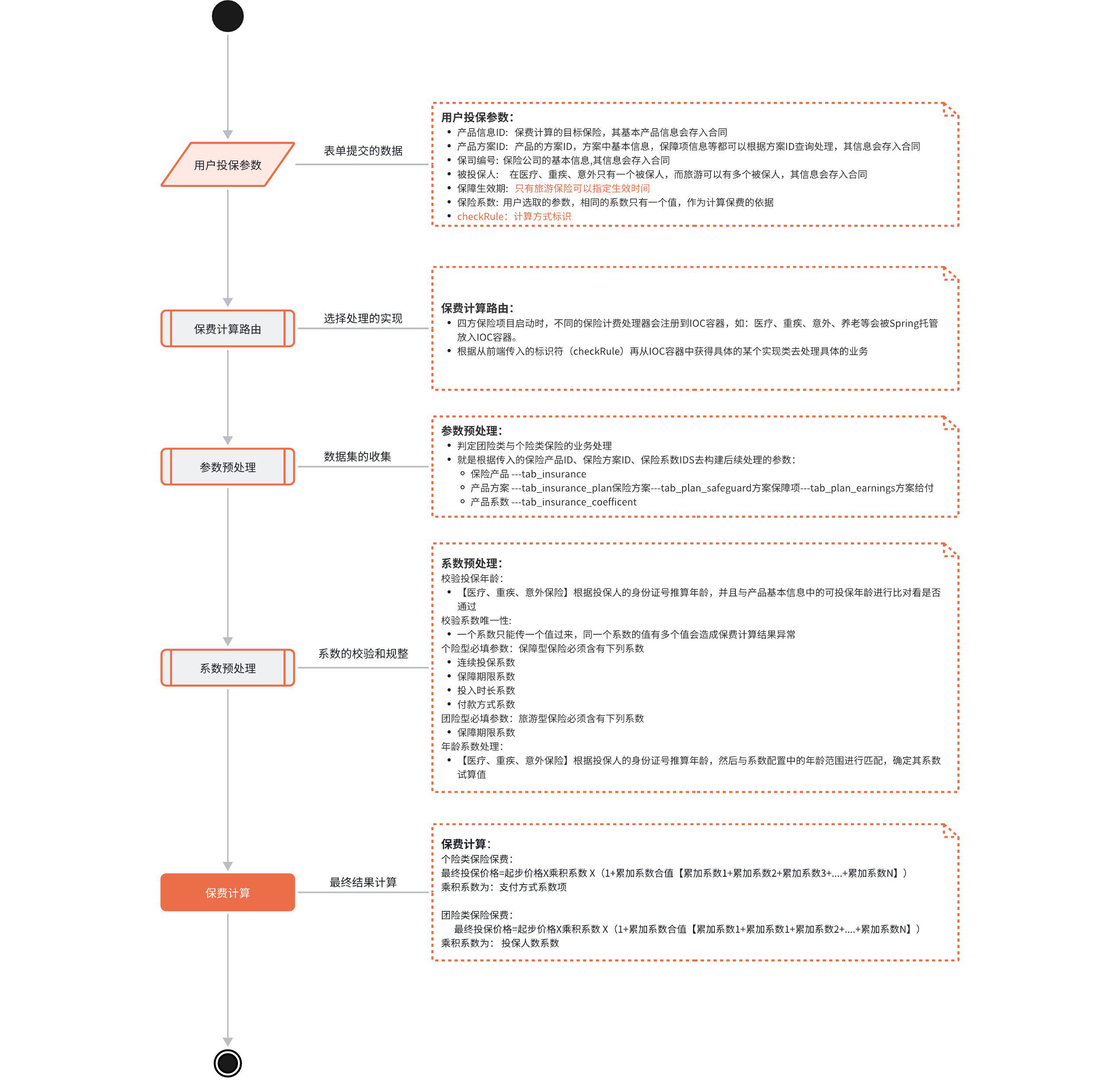


根据上图添加产品时设置的系数；如下代码在保险投保时根据当前用户填写的投保信息确定具体的系数：



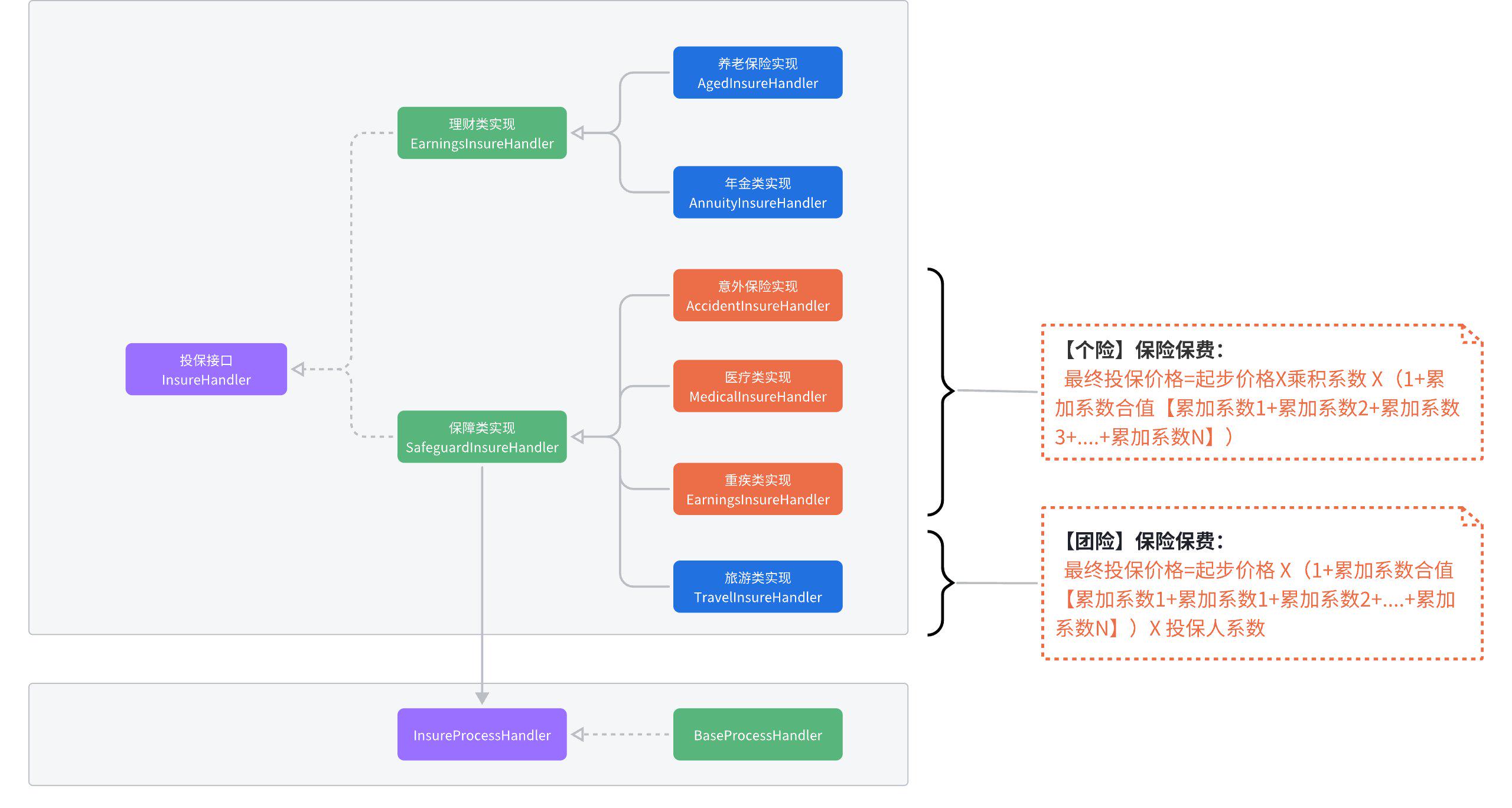


**5、保费试算**



**5.1、计算公式**

保费计算：分为个险类保险保费、团险类保险保费；它们的计算公式有区别，如下图；



**5.2、个险保费计算**

个险保费计算：主要是在SafeguardInsureHandler中实现，而AccidentInsureHandler【意外】、MedicalInsureHandler【医疗】、SevereIllnessInsureHandler【重疾】继承其实现逻辑。

|  |
| --- |
| **【个险】保险保费：**  最终投保价格=起步价格X乘积系数（支付方式） X（1+累加系数合值【累加系数1+累加系数2+累加系数3+....+累加系数N】） |

实现保费计算：



在经过了前面的参数、系数等预处理；已经在上图中得到本次保险所有要参加计算的系数；那么只需要按照计算公式计算即可；具体：

com.itheima.sfbx.insurance.handler.impl.BaseProcessHandler#premiumComputeSafeguard

|  |
| --- |
| Java public String premiumComputeSafeguard(InsurancePlanVO insurancePlanVO, List<InsuranceCoefficentVO> coefficentVOs) {  *//保费 = 方案起步价格 \* （1+各个系数项之和）\* 支付方式系数*  *//1、起步价格* BigDecimal price = insurancePlanVO.getPrice();   *//2、从系数集合中排除支付方式系数之后；统计出：（1+各个系数项之和）* BigDecimal sumCoefficient = coefficentVOs.stream().filter(cef -> {  return !cef.getCoefficentKey().equals(InsuranceConstant.*PAY\_MENT*);  })  .map(cef -> {  return !EmptyUtil.*isNullOrEmpty*(cef.getScore()) ? cef.getScore() : BigDecimal.*ZERO*;  })  *//累加：参数一：初始值，参数二：二元操作函数* .reduce(BigDecimal.*ZERO*, BigDecimal::add).add(new BigDecimal(1));*//2、从系数集合中排除支付方式系数之后；统计出：（1+各个系数项之和）*   *//3、支付方式系数从系数集合中挑出来* InsuranceCoefficentVO paymentCoefficient = coefficentVOs.stream().filter(cef -> {  return cef.getCoefficentKey().equals(InsuranceConstant.*PAY\_MENT*);  })  .findFirst().get();  BigDecimal paymentScore = paymentCoefficient.getScore();   *//4、计算价格并返回* DecimalFormat decimalFormat = new DecimalFormat("0.00");  return decimalFormat.format(price.multiply(sumCoefficient).multiply(paymentScore)); } |

**5.3、团险保费计算**

团险保费计算：主要是在TravelInsureHandler中实现，是对SafeguardInsureHandler实现的重写。

|  |
| --- |
| **【团险】保险保费：**  最终投保价格=起步价格 X 乘积系数（投保人系数） X (1+累加系数合值【累加系数1+累加系数1+累加系数2+....+累加系数N】） |

也就是说 团险 与个险的计算区别就是；团险要把人数算进去计算。具体如下：



在计算方法中，实现如下代码：

com.itheima.sfbx.insurance.handler.impl.BaseProcessHandler#premiumComputeTravel

|  |
| --- |
| Java public String premiumComputeTravel(InsurancePlanVO insurancePlanVO, List<InsuranceCoefficentVO> coefficentVOs) {  *//保费 = 起步价格 \* （1+其它系数项）\* 投保人系数*  *//1、起步价格* BigDecimal price = insurancePlanVO.getPrice();   *//2、从系数项集合中排除投保人数系数之后；统计出：（1+各个系数项之和）* BigDecimal sumCoefficient = coefficentVOs.stream().filter(cef -> {  return !cef.getCoefficentKey().equals(InsuranceConstant.*NUMBER\_OF\_PEOPLE*);  }).map(cef -> {  return !EmptyUtil.*isNullOrEmpty*(cef.getScore())?cef.getScore():BigDecimal.*ZERO*;  }).reduce(BigDecimal.*ZERO*, BigDecimal::add).add(BigDecimal.*ONE*);   *//3、获取本次投保人数系数* InsuranceCoefficentVO numberPeopleCoefficient = coefficentVOs.stream().filter(cef -> {  return cef.getCoefficentKey().equals(InsuranceConstant.*NUMBER\_OF\_PEOPLE*);  }).findFirst().get();  BigDecimal numberScore = numberPeopleCoefficient.getScore();   *//4、计算保费 = 起步价格 \* （1+其它系数项）\* 投保人系数* DecimalFormat decimalFormat = new DecimalFormat("0.00");  return decimalFormat.format(price.multiply(sumCoefficient).multiply(numberScore)); } |

**6、作业&问题**

* 说出：你熟悉Java中的那些设计模式？在哪里用到了，举例说明一下
* 保障类的保险产品有什么？
* **简历职责-候选**：改造app端用户保费试算的原条件控制语句的计算处理为策略模式，提高不同保险类型进行保费计算时的灵活性和开发效率；
* **简历职责-候选**：采用门面模式重构在保费试算代码中对于参与计算的系数累加因子、乘积因子的安全校验，数值校验的代码；
* **简历职责-候选**：独立完成在app端中用户选择保险方案及系数项之后的投保试算功能；