**第5章-产品详情-性能优化与数据脱敏**

**学习目标**

* 复习保险产品数据结构完成保险详情的功能开发
* 了解关键服务的性能优化方式，使用nginx和springcache对详情页优化
* 基于spring-aop自定义注解实现数据脱敏

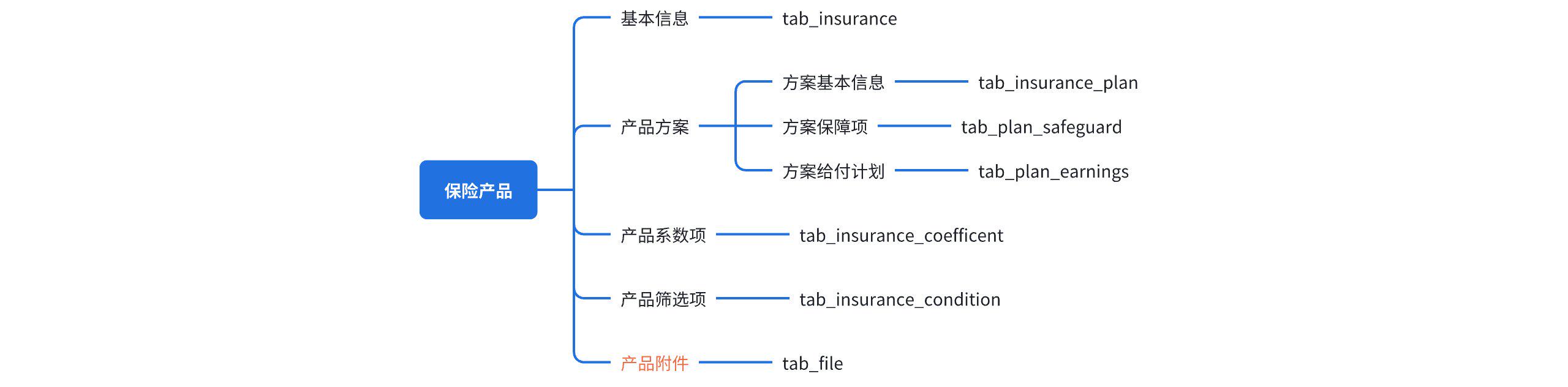
**1、保险详情页**

**1.1、需求分析**

登录 app 端；http://sf.app.itheima.net/#/pages/product/index；访问 产品；然后找选一个产品，查看详情如下图示：



根据我们前面的学习；了解到保险的整个组成结构：



基本上在app端用户在浏览保险产品时都需要记录；这样能够给系统提供以后的数据分析；如哪个保险产品更加吸引人等信息；所以在用户进入详情页面后还需要记录浏览记录。

|  |  |
| --- | --- |
| 功能点 | 功能说明 |
| 产品详情查询 | 就是根据产品ID查询出基本信息、产品方案、产品系数、产品附件等信息 |
| 产品浏览记录 | 在进入到详情页面后记录用户的浏览记录 |

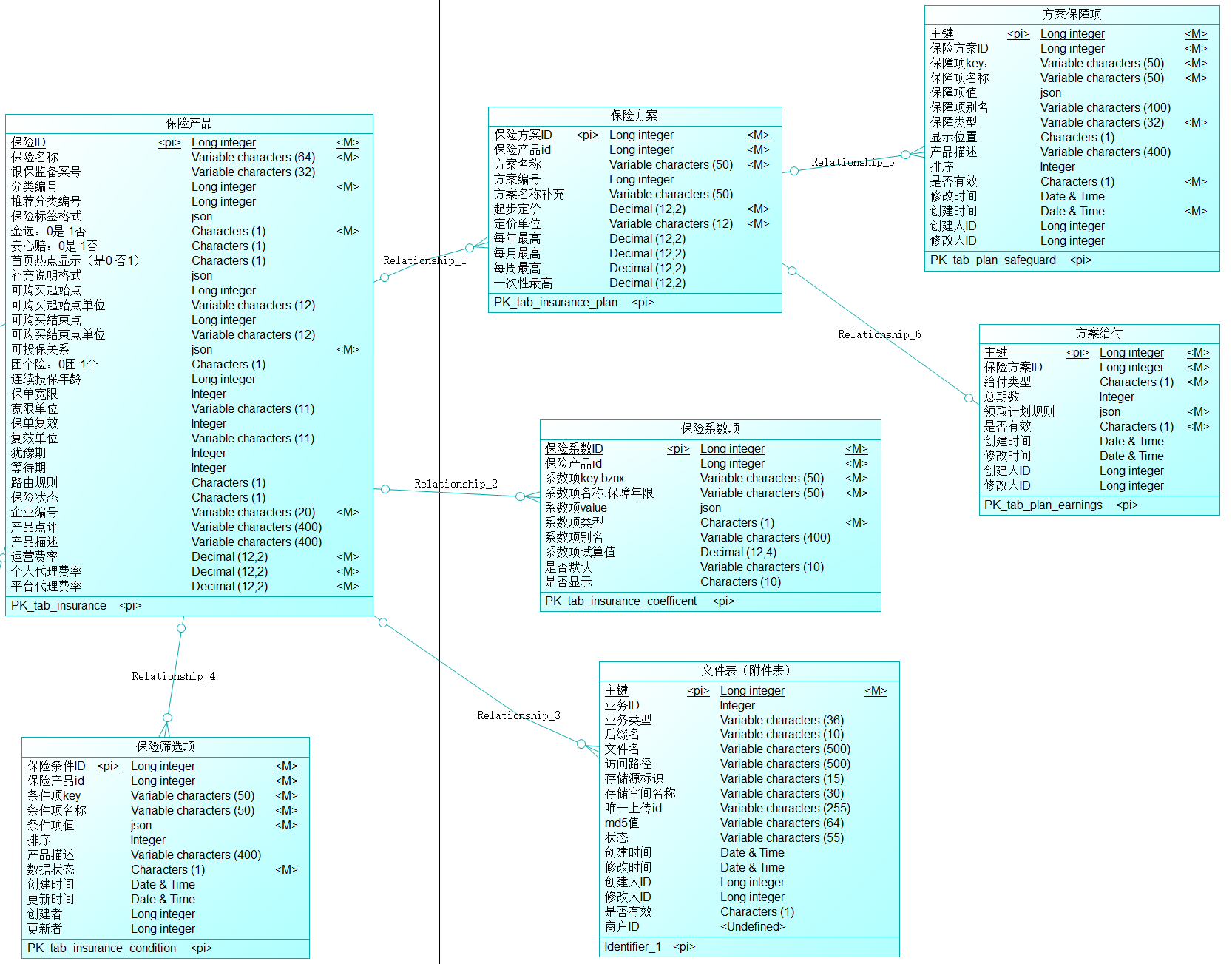
**1.2、库表设计需知（重要）**

（1）**如果是在管理后端开发功能**；那么在原型查看、需求分析之后一般都是进行数据库设计；设计对应的概念、物理模型，梳理清楚实体之间关系；再进行接口分析设计；再开发。

（2）**如果是在前台的页面、app等功能开发时**；那么在查看原型之后；并不是设计数据库表，因为数据库表已经设计好，这时需要做的是去复盘自己曾设计的或者去了解他人设计的表；再写分析编写接口；再开发。

在四方保险中；产品的数据库表设计已经做完，不需要再次设计，直接从这些表中查询出来展示即可。

它们的关系如下：



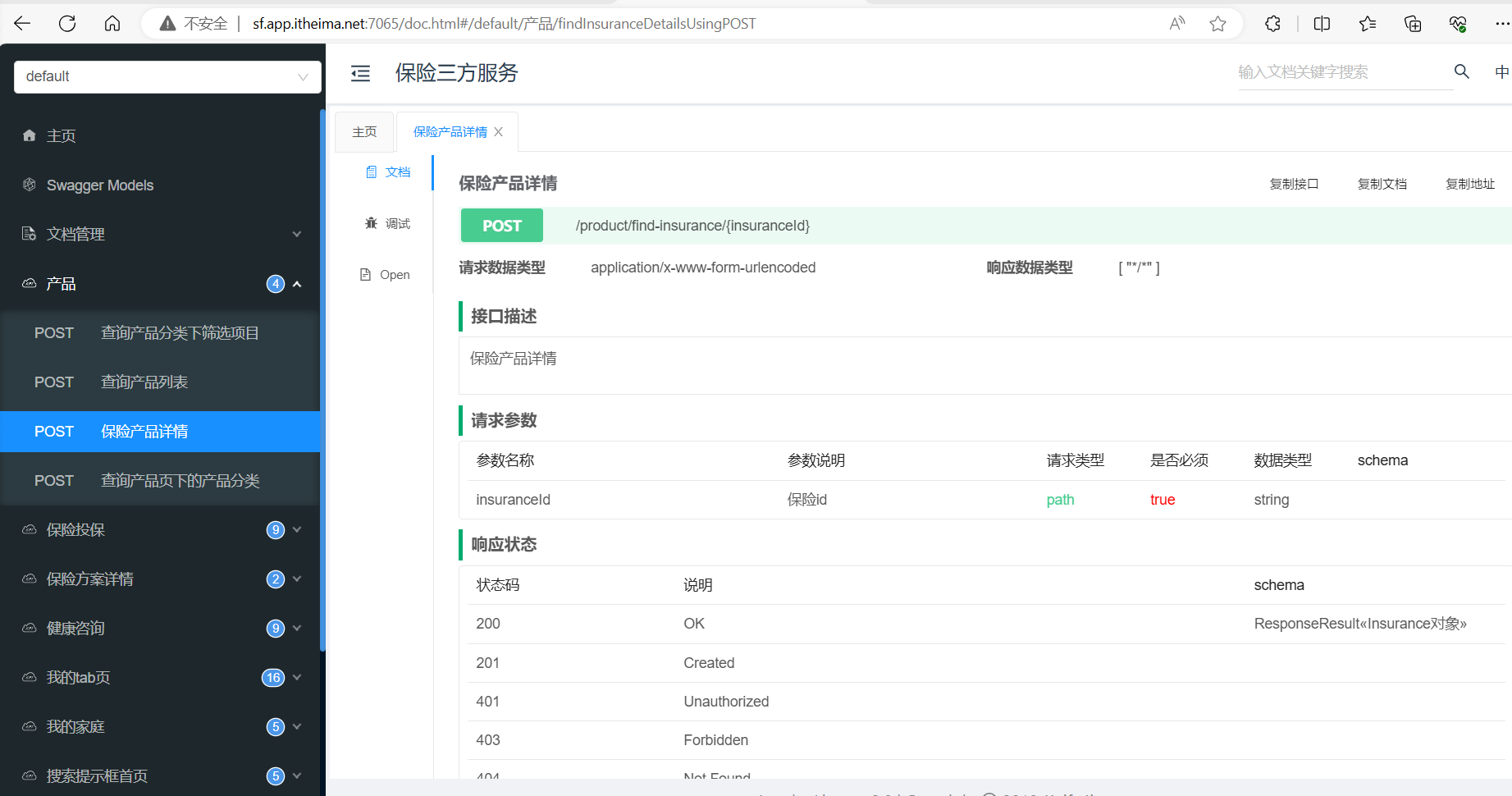
|  |
| --- |
| **课堂练习编写SQL语句**：  1、模拟app端进行筛选查询保险产品；查询 保多久 为 1天、2天 的保险产品信息（可使用 JSON\_CONTAINS(json字段, '"查询值"') ）——》保险产品列表  2、根据详情页面的分析；需要展示出保险产品的基本信息、方案（保障项、给付计划）、系数、附件、筛选项（可选）  也就是说：前端会传递一个保险产品id；然后再根据保险产品id查询基本信息、方案（保障项、给付计划）、系数、附件、筛选项（可选）——》从产品列表中点击了一个产品，进入详情页面 |

**1.3、接口分析与设计**

根据原型界面的分析；在查看产品详情的时候；需要查看产品基本信息、产品系数、产品方案（方案保障项、方案给付计划）、产品筛选项、产品附件等信息即可。所以对应写一个查询产品详情的接口即可。

接口地址：

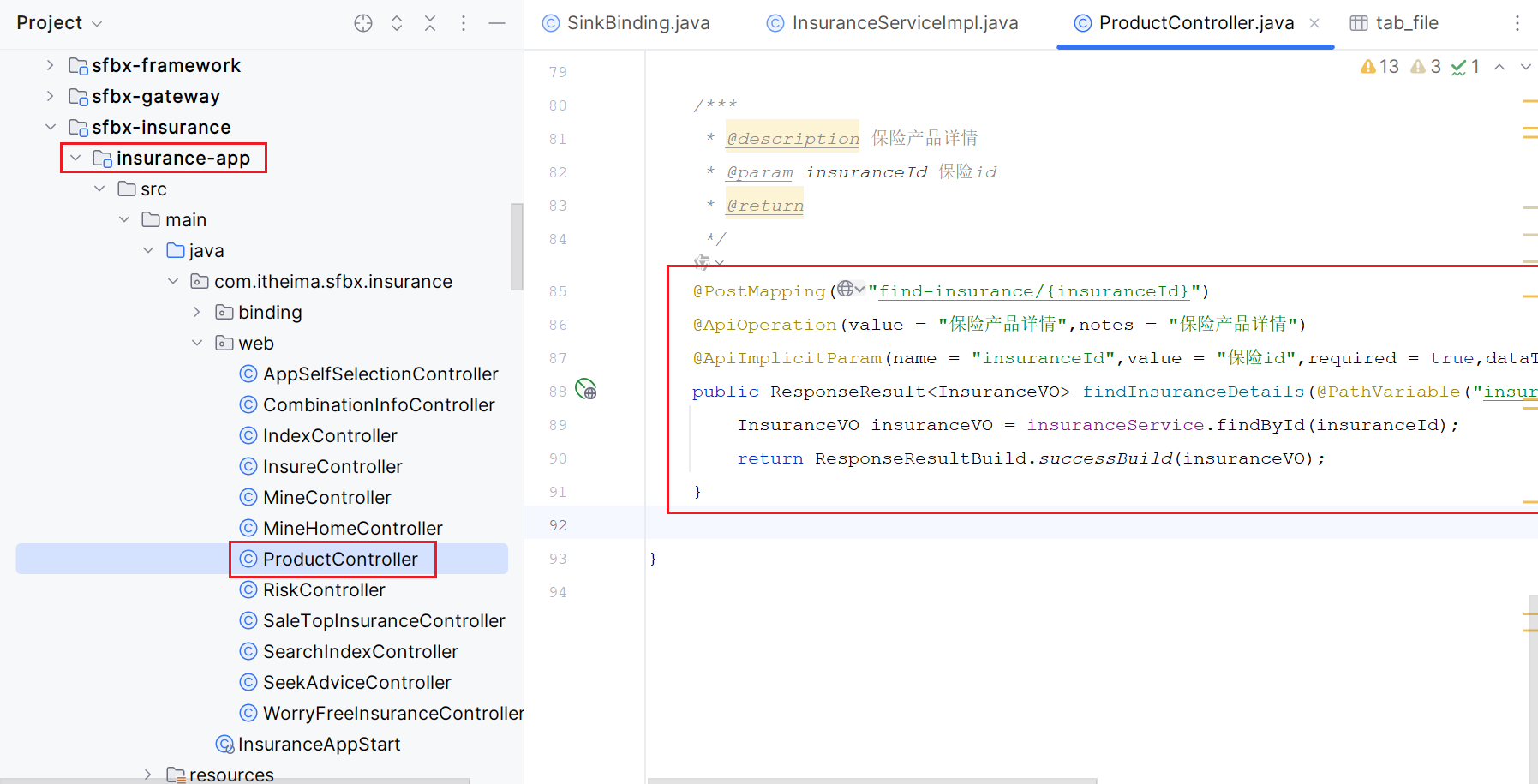
http://sf.app.itheima.net:7065/doc.html#/default/%E4%BA%A7%E5%93%81/findInsuranceDetailsUsingPOST



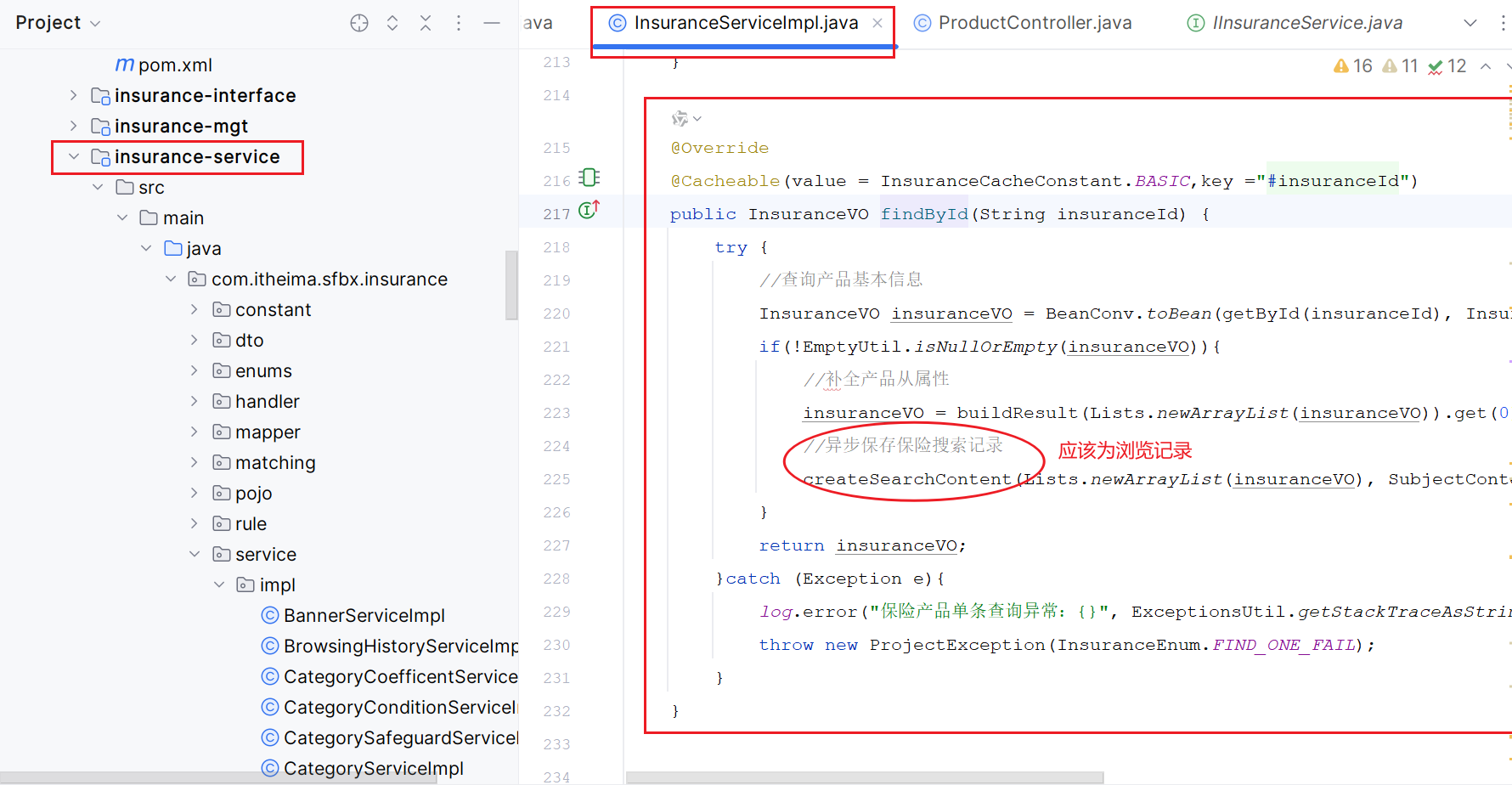
**1.4、接口实现**

需要注意；在实现根据产品id查询产品详情的时候；也需要记录产品被浏览的记录。

**处理器**：



**业务实现**：



|  |
| --- |
| 课后练习：  自行实现上述InsuranceServiceImpl中的findById()方法里面的buildResult方法重写（最好是自己写一个，只查询一个保险产品id对应的保险信息的方法） |

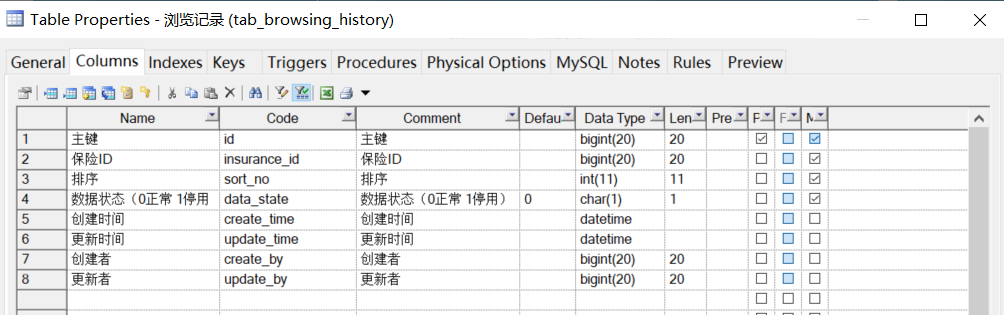
**1.5、修复问题-课堂练习**

在接口实现中查询产品详情时，需要记录的是浏览记录，而我们的开发人员记录为搜索记录是不对的；请各位同学将该问题修复。

要求：将每次查询产品时，当作一次浏览记录；记录到浏览记录表 tab\_browsing\_history

**1.5.1、浏览记录表**

保存浏览记录，只需要知道在什么时间是谁浏览了什么保险产品即可；所以浏览记录的数据库表结构如下；



**1.5.2、修复实现**

1）注册线程池

创建如下方法：

com.itheima.sfbx.framework.taskexecutor.config.TaskExecutorConfig#browsingHistoryExecutor

|  |
| --- |
| Java */\*\**  *\* 浏览记录程池*  *\* @return*  *\*/* @Bean("browsingHistoryExecutor") public Executor browsingHistoryExecutor() {  *log*.info("start browsingHistoryExecutor");  ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();  *//配置核心线程数* executor.setCorePoolSize(taskExecutorProperties.getCorePoolSize());  *//配置最大线程数* executor.setMaxPoolSize(taskExecutorProperties.getMaxPoolSize());  *//配置队列大小* executor.setQueueCapacity(taskExecutorProperties.getQueueCapacity());  *//配置线程池中的线程的名称前缀* executor.setThreadNamePrefix("browsingHistoryExecutor");  *// rejection-policy：当pool已经达到max size的时候，如何处理新任务*  *// CALLER\_RUNS：不在新线程中执行任务，而是由调用者所在的线程来执行* executor.setRejectedExecutionHandler(new ThreadPoolExecutor.CallerRunsPolicy());  *//执行初始化* executor.initialize();  *// 当关闭线程池的时候是否等待所有任务结束后再关闭* executor.setWaitForTasksToCompleteOnShutdown(true);  return executor; } |

2）创建保存浏览历史业务方法

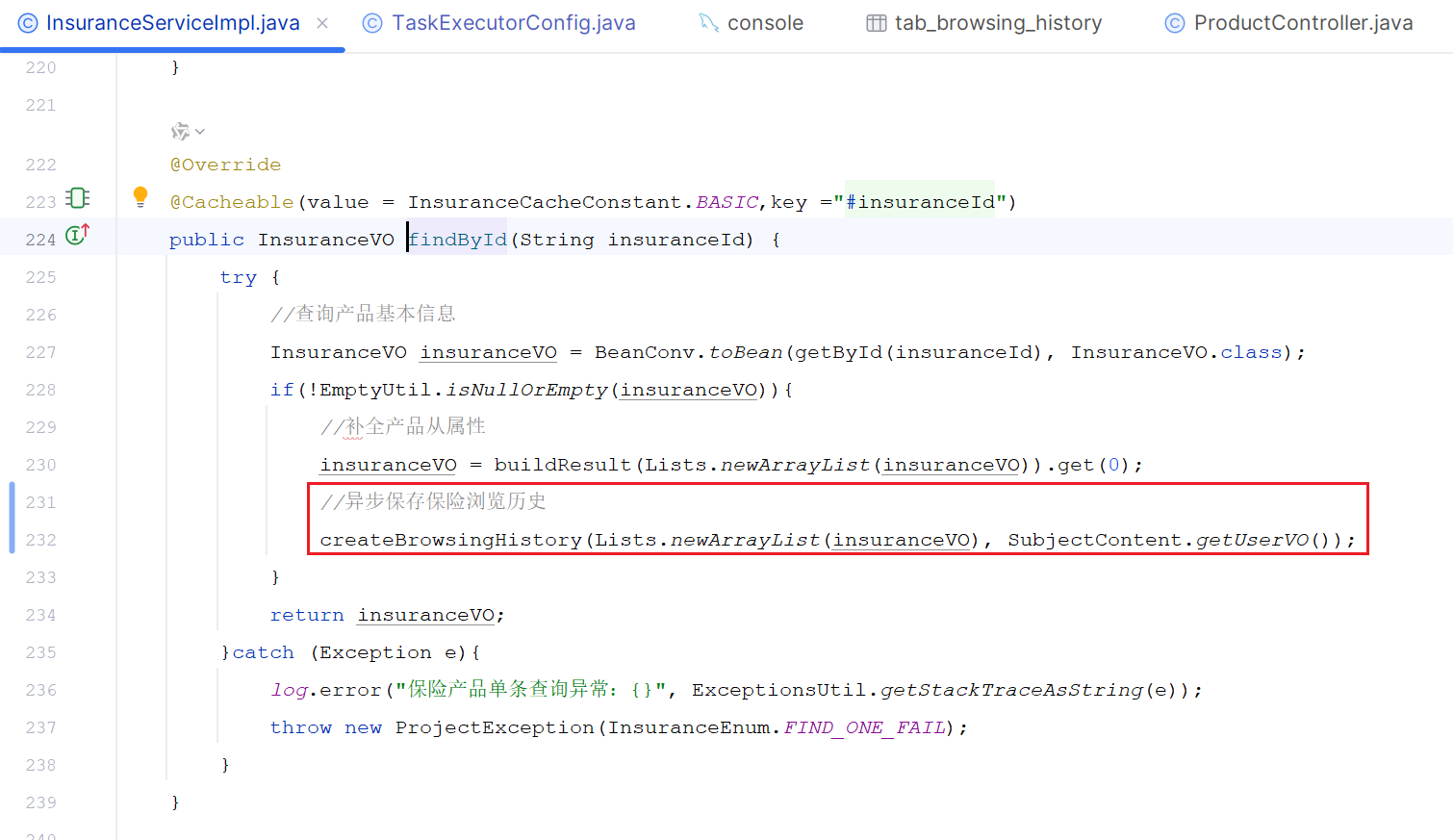
创建方法如下：

com.itheima.sfbx.insurance.service.impl.InsuranceServiceImpl#createBrowsingHistory

|  |
| --- |
| Java */\*\**  *\* 异步添加浏览记录*  *\* @param insuranceVOs*  *\* @return*  *\*/* private void createBrowsingHistory(List<InsuranceVO> insuranceVOs, UserVO userVO) {  browsingHistoryExecutor.execute(new Runnable() {  @Override  public void run() {  ArrayList<BrowsingHistory> browsingHistories = new ArrayList<>();  for (InsuranceVO insuranceVO: insuranceVOs) {  BrowsingHistory browsingHistory = new BrowsingHistory();  browsingHistory.setInsuranceId(insuranceVO.getId());  browsingHistory.setCreateBy(userVO.getId());  browsingHistory.setDataState(SuperConstant.*DATA\_STATE\_0*);  browsingHistory.setSortNo(0);  browsingHistories.add(browsingHistory);  }  browsingHistoryService.saveBatch(browsingHistories);  }  }); } |

3）修改查询产品方法

修复方法：com.itheima.sfbx.insurance.service.impl.InsuranceServiceImpl#findById



|  |
| --- |
| 常见面试题：  **通过ThreadPoolExecutor类创建线程池时7个核心参数：**   1. **corePoolSize**：线程池的基本大小，即即使线程池中的线程处于空闲状态，这些线程也不会被终止，除非设置了allowCoreThreadTimeOut为true。 2. **maximumPoolSize**：线程池允许的最大线程数。当线程池中的线程数达到corePoolSize后，如果继续提交任务，线程池会尝试复用已有的线程执行新任务。如果已有的线程都在执行任务，则会创建新的线程，直到总线程数达到maximumPoolSize。 3. **keepAliveTime**：当线程池中的线程数超过corePoolSize时，多余的空闲线程在等待新任务的时间超过keepAliveTime后会被终止。 4. **unit**：keepAliveTime参数的时间单位，如秒、分钟等，通常使用TimeUnit枚举值表示。 5. **workQueue**：用于保存待处理任务的阻塞队列。常见的阻塞队列实现包括ArrayBlockingQueue、LinkedBlockingQueue、SynchronousQueue等。 6. **threadFactory**：用于设置创建线程的工厂，可以通过自定义线程工厂来设置线程名称、优先级等属性。 7. **handler**：拒绝策略，当线程池和队列都满时，如何处理新提交的任务。常见的拒绝策略包括AbortPolicy（抛出异常；默认）、CallerRunsPolicy（由调用线程执行任务）、DiscardPolicy（丢弃任务）和DiscardOldestPolicy（丢弃队列中最老的任务）。 |

**1.5.3、测试**

重新启动 insurance-app ；然后访问app端的任意一个产品；到数据库中查询 tab\_browsing\_history 查看是否有数据即可。

**2、性能优化**

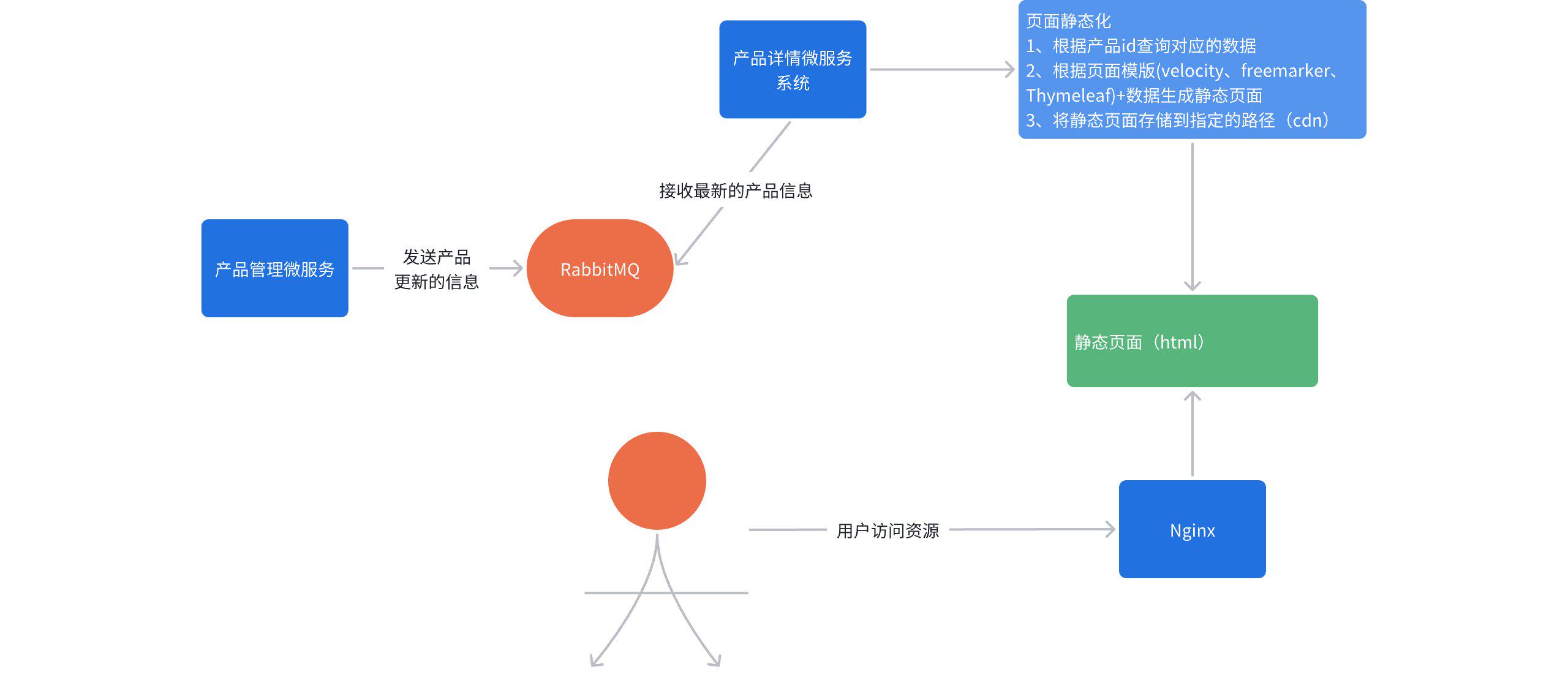
保险的产品并不像我们购买一般的商品那么多；但是它却是被查询非常频繁的；为了整体系统性能考虑；我们需要对系统中产品详情、各类查询做一些性能上的优化；为了提升数据加载展示速度，常见优化手段有：

* 页面采用CDN（内容分发节点）
* 限流（保障系统稳定性）
* 缓存

CDN一般是在不同的服务器节点中存储静态资源、页面；而我们的保险可能是app原生开发并不能缓存html等静态资源。所以暂时不讨论这种方式。下面我们基于Nginx限流和SpringCache缓存展开，对保险产品等的优化。

|  |
| --- |
| 存储静态资源——》页面静态化——》velocity、freemarker、Thymeleaf |

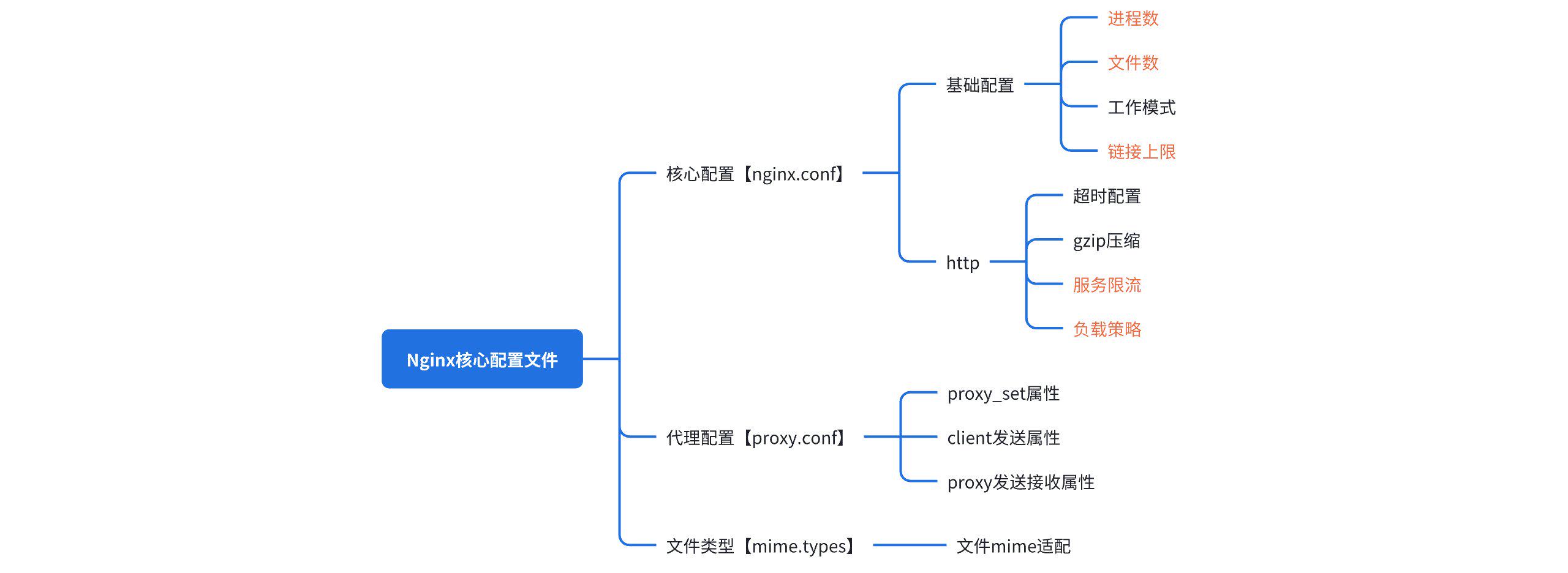
静态化技术方案参考如下：



**2.1、Nginx设置调优**

**2.1.1、配置文件说明**

Nginx中有几类配置文件，它们的作用不同；具体如下：



而我们一般对于Nginx的配置、调优都是在 nginx.conf文件中进行的。nginx.conf文件中有一些参数可以用于调优配置：

|  |  |
| --- | --- |
| 配置项 | 说明 |
| **worker\_processes** | Nginx 工作进程的数量。Nginx 使用基于事件的模型和异步 I/O 来处理多个连接，但每个连接都由一个工作进程来处理   * 如果 worker\_processes 设置为 1，则 Nginx 将只使用一个工作进程来处理所有连接。 * 如果设置为 auto，Nginx 将尝试自动检测并设置与 CPU 核心数量相等的进程数。 |
| **worker\_connections** | 每个工作进程可以处理的最大连接数。包括所有入站和出站连接。因此，Nginx 可以处理的总连接数是 worker\_processes 和 worker\_connections 的乘积。  例如：如果 worker\_processes 设置为 4，并且 worker\_connections 设置为 1024，则 Nginx 可以处理的总连接数是 4 \* 1024 = 4096。 |
| **worker\_rlimit\_nofile** | 每个工作进程可以打开的最大文件描述符数（操作系统文件、资源的最大数量）。worker\_rlimit\_nofile 的值应该足够大，以容纳 worker\_processes \* worker\_connections 所需的文件描述符。  例如：如果 worker\_processes 设置为 4，worker\_connections 设置为 1024，并且我们想要为每个工作进程分配足够的文件描述符，那么 worker\_rlimit\_nofile 可以设置为 4096 |

|  |
| --- |
| Nginx启动之后的两个进程：  master进程：主要负责管理和监控worker进程，不直接处理网络请求，而是负责加载配置、启动工作进程等任务，只有一个。master会监控worker进程的运行状态，重启、重载配置等操作  worker进程：负责处理实际的网络请求；可以配置多个，通常等于CPU核数，采用异步非阻塞的I/O模型处理网络请求。 |

**2.1.2、限流**

**1）基于IP限流**

基于IP限流可以限制特定IP地址在单位时间内的请求频率。这有助于防止单个IP地址对服务器发起过多的请求，导致服务器过载。

|  |  |
| --- | --- |
| 配置项 | 说明 |
| limit\_req\_zone | 用于定义限流区域和相关参数。这个区域用于存储客户端IP地址的请求计数信息，并根据定义的速率限制来处理请求。  例如，limit\_req\_zone $binary\_remote\_addr zone=mylimit:10m rate=1r/s; 这里，$binary\_remote\_addr是客户端的IP地址（二进制格式），mylimit是区域名称，10m是分配给该区域的共享内存大小，1r/s表示每秒只允许一个请求。 |
| limit\_req | 在需要限流的 nginx.conf中的http中的location 中使用，引用之前定义的限流区域。当客户端的请求频率超过指定的限制时，Nginx会采取相应的措施，如延迟请求或拒绝请求。  例如，limit\_req zone=mylimit burst=5 nodelay; 这里，mylimit是之前定义的限流区域名称，burst=5表示在超过速率限制时，还可以处理额外的5个请求，而nodelay表示立即拒绝超出限制的请求，而不是将它们放入队列中等待处理。 |

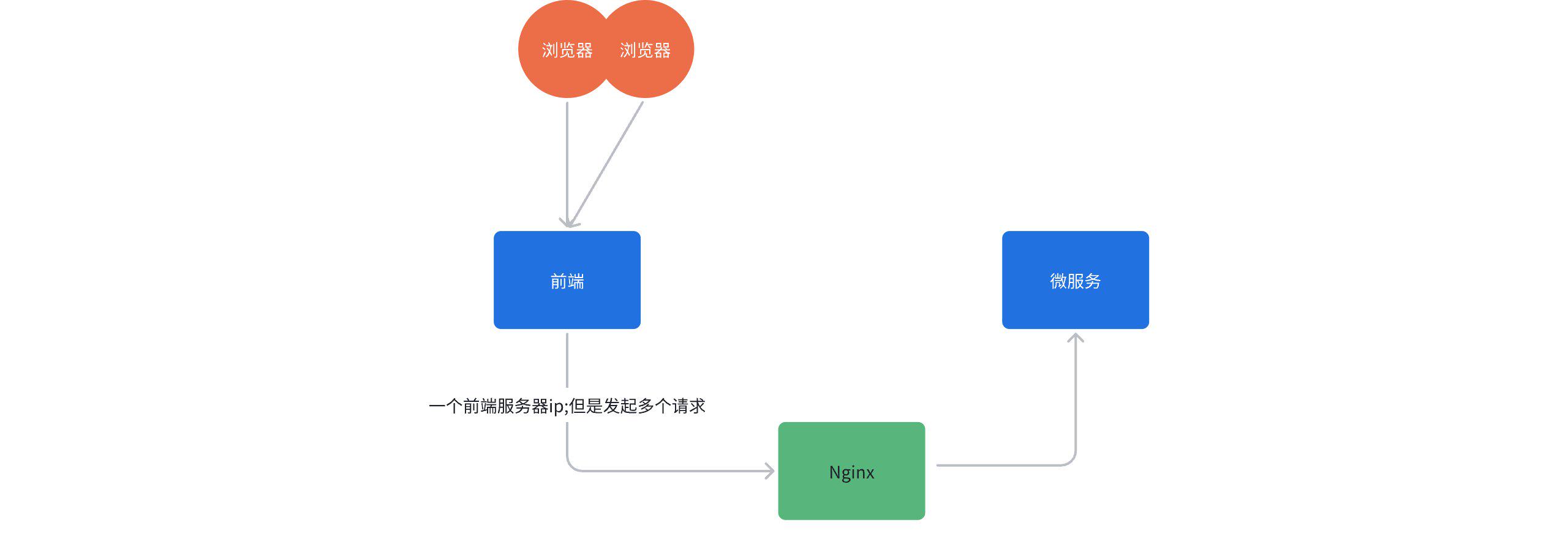
示例：如果限制每个IP地址每秒只能发送一个请求到/api路径的话；可以如下配置：

|  |
| --- |
| Nginx http {   limit\_req\_zone $binary\_remote\_addr zone=mylimit:10m rate=1r/s;     server {   listen 80;     location /api {   limit\_req zone=mylimit burst=5 nodelay;   # 其他配置...   }   }  } |

**说明**：如果某个IP地址在1秒内发送了超过一个请求到/api路径，Nginx将根据burst参数的值来决定如何处理这些额外的请求。上面的示例中，由于设置了burst=5，所以Nginx会允许额外的5个请求，但会立即拒绝超出这个限制的请求（由于nodelay参数的存在）。

**2）基于连接数限流**

连接数限流可以限制单个IP地址或特定key的**并发连接数**。这有助于防止单个IP地址或客户端建立过多的连接，耗尽服务器资源。



|  |  |
| --- | --- |
| 配置项 | 说明 |
| limit\_conn\_zone | 用于定义一个共享内存区域，这个区域用于存储每个客户端的连接数信息。  例如，limit\_conn\_zone $binary\_remote\_addr zone=addr:10m; 这行代码定义了一个名为addr、大小为10MB的共享内存区域，用于根据客户端的IP地址（$binary\_remote\_addr）来限制连接数。 |
| limit\_conn | 用于在指定的位置限制客户端的连接数。它引用之前通过limit\_conn\_zone定义的共享内存区域和key，当连接数达到限制时，新的连接将被拒绝。  例如，location / { limit\_conn addr 10; } 这行代码表示，对于IP地址为key的连接，最大并发连接数为10。如果某个IP地址的并发连接数超过了这个限制，新的连接请求将被Nginx拒绝。 |
|  |  |

**示例**：

|  |
| --- |
| Nginx http {   limit\_conn\_zone $binary\_remote\_addr zone=addr:10m;     server {   listen 80;   server\_name example.com;     location / {   limit\_conn addr 10;   # 其他配置...   }   }  } |

**说明**：在上述示例中，limit\_conn\_zone定义了一个名为addr的连接数限流区域，该区域使用IP地址作为key，并设置了10MB的共享内存区域来存储状态信息。在location块中，limit\_conn指令引用了该连接数限流区域，并设置了10作为单个IP地址的最大并发连接数。如果某个IP地址的并发连接数超过了这个限制，Nginx将拒绝新的连接请求。

**3）区别**

**一个连接中可以发起多个请求**。这就是最核心的区别。

* **关注点不同**：IP限流关注的是请求的频率，而连接数限流关注的是同时建立的连接数量。
* **应用场景不同**：IP限流更适用于防止密码暴力破解等场景，因为它关注的是请求的频率；而连接数限流更适用于防止资源耗尽的场景，因为它关注的是连接的数量。
* **效果不同**：IP限流可能会允许某个IP地址在短时间内建立多个连接，但会限制它的请求频率；而连接数限流则会限制某个IP地址同时建立的连接数量，不论请求频率如何。

**2.2、Spring Cache 缓存**

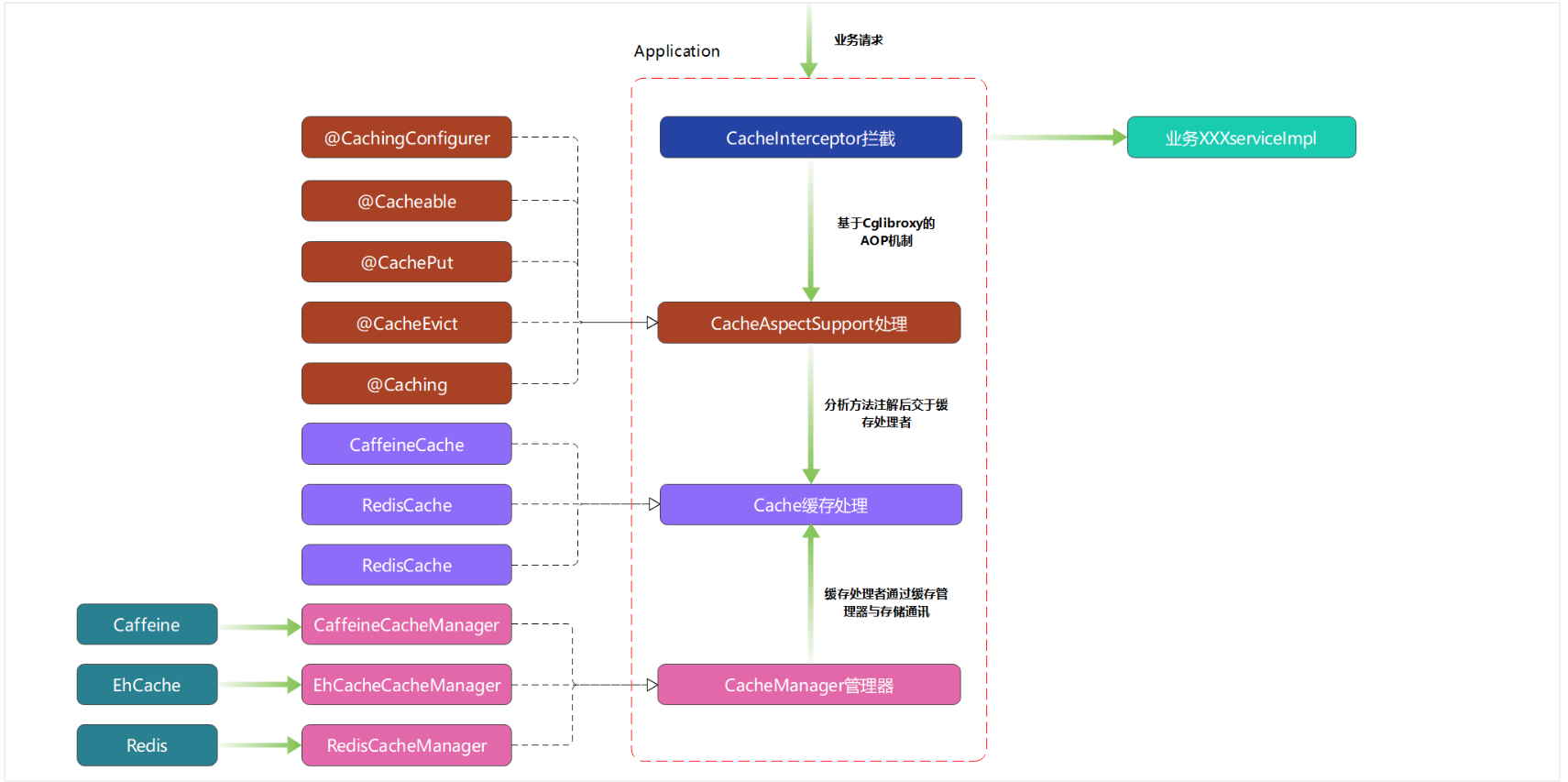
**2.2.1、简介**

Spring Cache是Spring框架提供的一种缓存抽象机制，旨在简化应用中的缓存操作。**它通过将方法的返回值缓存起来，当下次调用同一方法时，如果传入的参数与之前的调用相同，就可以直接从缓存中获取结果，而不需要再执行方法体中的代码**。这种方式可以显著提高系统的性能和响应速度。

Spring Cache的主要特点包括：

1. **声明式缓存**：通过在方法上添加注解，如@Cacheable、@CachePut、@CacheEvict等，来声明缓存的行为，无需手动编写缓存代码。这使得缓存功能更加易于集成和使用。
2. **多种缓存支持**：Spring Cache提供了对多种缓存框架的支持，包括**Redis**、**EhCache**、**Guava Cache**、**Caffeine**等。这意味着你可以根据项目的需要选择合适的缓存实现。
3. **基于AOP的缓存**：Spring Cache利用了面向切面编程（AOP）的技术，实现了基于注解的缓存功能。这使得业务代码无需关心底层使用了什么缓存框架，只需简单地添加注解即可实现缓存功能，降低了对代码的侵入性。
4. **动态指定缓存Key和条件**：Spring Cache还支持使用SpEL（Spring Expression Language）表达式来动态地指定缓存的Key和条件等。这使得缓存的配置更加灵活和强大。

Spring Cache主要是通过CacheManager接口来统一不同的缓存技术。CacheManager是Spring提供的各种缓存技术抽象接口；Spring Cache 的具体架构如下：



* **CacheInterceptor缓存拦截器**是Spring AOP（面向切面编程）中的一个拦截器，实现了MethodInterceptor接口。当一个方法被带有缓存注解（如@Cacheable、@CachePut、@CacheEvict等）标记时，CacheInterceptor会拦截这个方法的调用。拦截后，CacheInterceptor会根据注解的指示，检查缓存、从缓存中获取数据、将数据存入缓存或删除缓存中的数据。
* CacheAspectSupport**缓存切面支持** 是CacheInterceptor的基类，提供了缓存操作的核心实现。它封装了缓存操作的逻辑，如计算缓存键、检查缓存、从缓存中获取数据、将数据存入缓存或删除缓存中的数据等。当CacheInterceptor拦截到一个方法调用时，它会委托给CacheAspectSupport来处理实际的缓存操作。
* **CacheManager** **缓存管理器** 是Spring提供的缓存管理器接口，它维护着多个Cache。不同的缓存技术需要实现不同的Cache接口。当应用程序需要缓存数据时，它会先通过CacheManager获得某个Cache，然后使用该Cache的API进行数据的存取。

**2.2.2、常用注解**

在Spring Cache中，常见的缓存注解主要有@EnableCaching、@Cacheable、@CachePut、@CacheEvict和@CacheConfig。

|  |  |
| --- | --- |
| 注解 | 说明 |
| **@EnableCaching** | 启用缓存注解支持。通常放在启动类或配置类上。 |
| **@Cacheable** | 标记一个方法，该方法的结果是可以被缓存的。在方法执行前，Spring会先检查缓存中是否有之前执行过的结果。如果有，则直接返回缓存中的数据；否则，执行方法并将结果存入缓存。  **主要属性**：   * value：缓存的名称，用于指定将结果存储在哪个缓存中。 * key：缓存的key值，用于从缓存中获取数据。可以使用SpEL表达式进行定义，如#p0表示方法的第一个参数。 * condition：缓存的条件，使用SpEL表达式定义。只有满足条件的方法调用结果才会被缓存。 * unless：缓存的排除条件，使用SpEL表达式定义。当结果为true时，方法的结果不会被缓存。 * 其他属性还包括keyGenerator、cacheManager、cacheResolver等，用于更复杂的缓存配置。 |
| **@CachePut** | 标记一个方法，该方法的结果将被存入缓存，无论之前是否已有缓存数据。每次都会执行方法，并将结果存入指定的缓存中。  **主要属性**：与@Cacheable类似，包括value、key、condition等。但@CachePut通常用于更新缓存中的值。 |
| **@CacheEvict** | 标记一个方法，该方法执行后将从缓存中移除一个或多个数据。主要用于删除缓存数据。  **主要属性**：   * value：缓存的名称，用于指定从哪个缓存中删除数据。 * key：缓存的key值，用于指定要删除的数据。可以使用SpEL表达式进行定义。 * allEntries：当为true时，表示删除指定缓存中的所有数据。 * beforeInvocation：当为true时，表示在方法执行前就删除缓存数据（默认为false）。 * 其他属性还包括condition、cacheManager等。 |
| **@CacheConfig** | 在类级别提供共享的缓存配置。当类中的多个方法使用相同的缓存配置时，可以使用@CacheConfig进行统一配置。  **主要属性**：与@Cacheable、@CachePut、@CacheEvict中的value、keyGenerator、cacheManager等属性相同。这些属性将作为该类中所有方法的默认缓存配置。 |

在不同的业务操作时；一般的缓存操作大概如下：

* **新增**：删除分页、列表的缓存可以删除，再将当前这条数据存缓存
* **修改**：删除分页、列表和当前这条数据的缓存
* **删除**：删除分页、列表和id对应的那些缓存
* **查询**：不会修改数据；所以只做存储缓存。将本次查询条件或者分页条件对应的查询结果缓存到缓存中

|  |
| --- |
| 若不在上述的操作中且缓存逻辑不定，可以全部缓存删除，保持数据一致 |

**2.2.3、项目整合缓存**

将在基础数据管理中 保障项管理 的操作都利用spring cache进行缓存操作。

**1）引入依赖**

在项目中一般需要引入如下的依赖；

|  |
| --- |
| XML <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-cache</artifactId> </dependency> <!--cache底层的实现不同引入不一样--> <dependency>  <groupId>org.redisson</groupId>  <artifactId>redisson-spring-boot-starter</artifactId> </dependency> |

而 保障项管理 属于 insurance-mgt里面的功能；引入了insurance-service；在其里面又引入了 framework-redis 在这个工程中有引入上述的依赖。

**2）配置缓存**

在 framework-redis 中的如下类；

sfbx-cloud\sfbx-framework\framework-redis\src\main\java\com\itheima\sfbx\framework\redis\config\RedisCacheConfig.java

|  |
| --- |
| Java package com.itheima.sfbx.framework.redis.config;  @Configuration *//开启spring cache缓存* @EnableCaching @EnableConfigurationProperties({TenantCacheProperties.class,DataSecurityCacheProperties.class}) public class RedisCacheConfig {   @Autowired  TenantCacheProperties tenantCacheProperties;   @Autowired  DataSecurityCacheProperties dataSecurityCacheProperties;   */\*\**  *\* 修改spring-cache默认前缀为:*  *\*/* @Bean  public CacheKeyPrefix cacheKeyPrefix() {  return cacheName -> cacheName + ":";  }   */\*\**  *\* 申明缓存管理器，会创建一个切面（aspect）并触发Spring缓存注解的切点（pointcut）*  *\* 根据类或者方法所使用的注解以及缓存的状态，这个切面会从缓存中获取数据，*  *\* 将数据添加到缓存之中或者从缓存中移除某个值*  *\*/* @Bean  public CacheManager cacheManager(RedisConnectionFactory redisConnectionFactory) {  *//对key的序列化操作：String* RedisSerializer<String> keySerializer = new StringRedisSerializer();  *//对value的序列化操作：json* GenericJackson2JsonRedisSerializer valuesSerializer = new GenericJackson2JsonRedisSerializer();  *//配置config,指定超时时间记得key val 序列化处理* RedisCacheConfiguration config = RedisCacheConfiguration.*defaultCacheConfig*()  .computePrefixWith(cacheKeyPrefix())  *//指定全局默认超时时间【600S】；测试方便，设置为1s，相当于不用缓存；测试缓存时可以设置多一些时间* .entryTtl(Duration.*ofSeconds*(1))  *//配置key的序列化方式* .serializeKeysWith(RedisSerializationContext.SerializationPair.*fromSerializer*(keySerializer))  *//配置value的序列化方式* .serializeValuesWith(RedisSerializationContext.SerializationPair.*fromSerializer*(valuesSerializer));  *//关闭空值的存储*  *//.disableCachingNullValues();*  *//使用CustomRedisCacheManager进行初始化* return new CustomRedisCacheManager(redisConnectionFactory, config, tenantCacheProperties,dataSecurityCacheProperties);  } } |

按照上述直接的配置了redis类型的 CacheManager的话；那么接下来在项目中就可以使用了。

**3）使用**

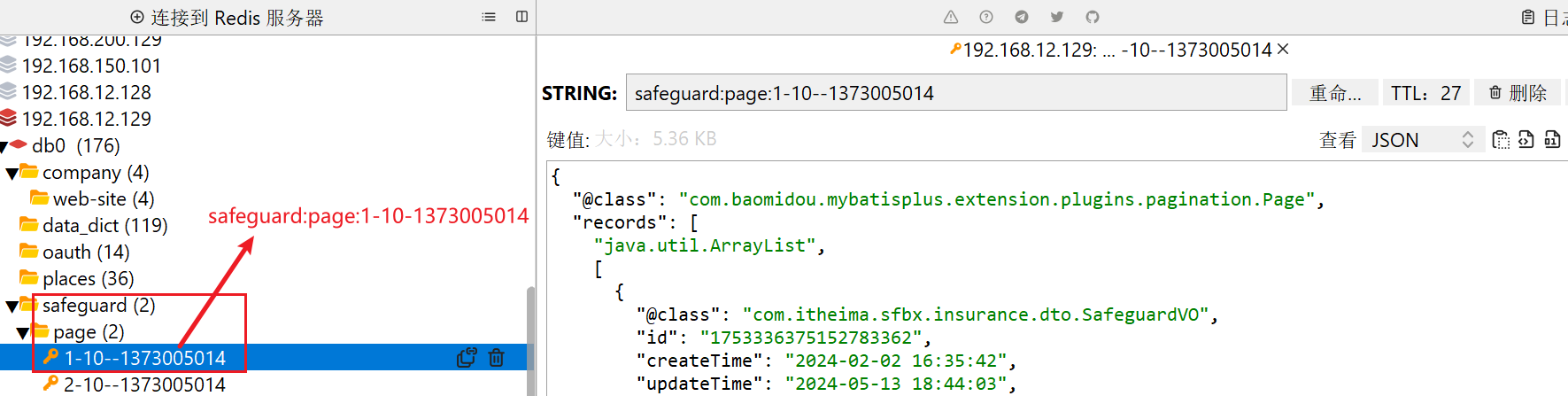
缓存的设置一般都是在业务层类中的各个方法上进行；那么对应保障项的话就是在 com.itheima.sfbx.insurance.service.impl.SafeguardServiceImpl 类中的方法中设置即可。

* **分页查询保障项**

需要按页号、页大小和本次的查询条件进行缓存，下次以同样的条件查询的时候才可以命中。



访问保障管理后：



能看到redis中已经有缓存数据；如果再次访问同一个页那么不会再次查询数据库数据；说明缓存成功。

* **保存保障项**

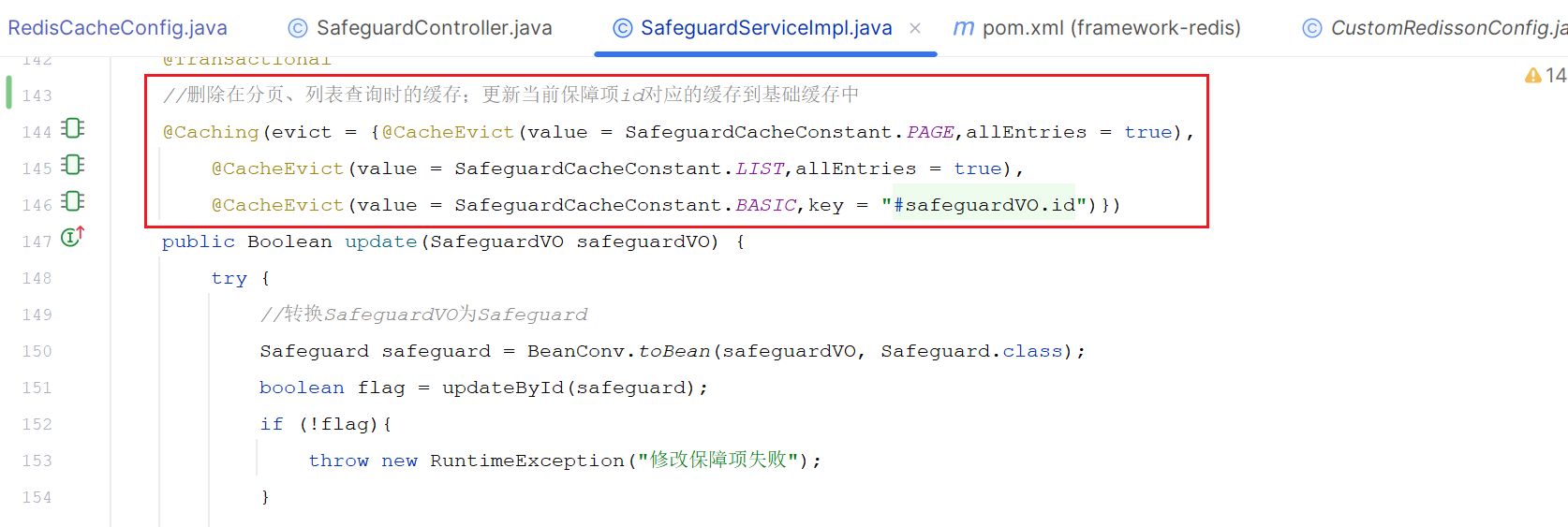
需要将分页、列表等查询中缓存了的数据清理掉；然后再将新保存的数据更新到缓存



可以新增一个保障项进行测试；原来在redis中的分页缓存数据已经被清空；但是由于页面又马上加载分页列表数据；所以又会再次执行分页查询后缓存了最新的分页数据到缓存中。

* **更新保障项**

需要将分页、列表等查询中缓存了的数据清理掉；然后再将新保存的数据更新到缓存



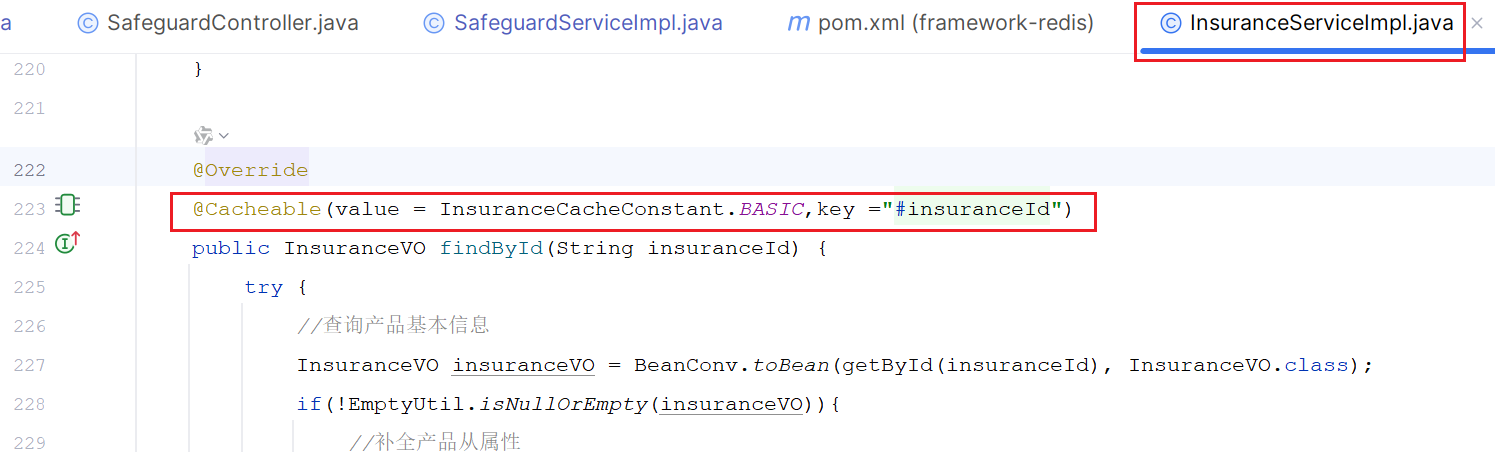
* **删除保障项**

需要将分页、列表，以及当前删除的数据等缓存了的数据清理掉

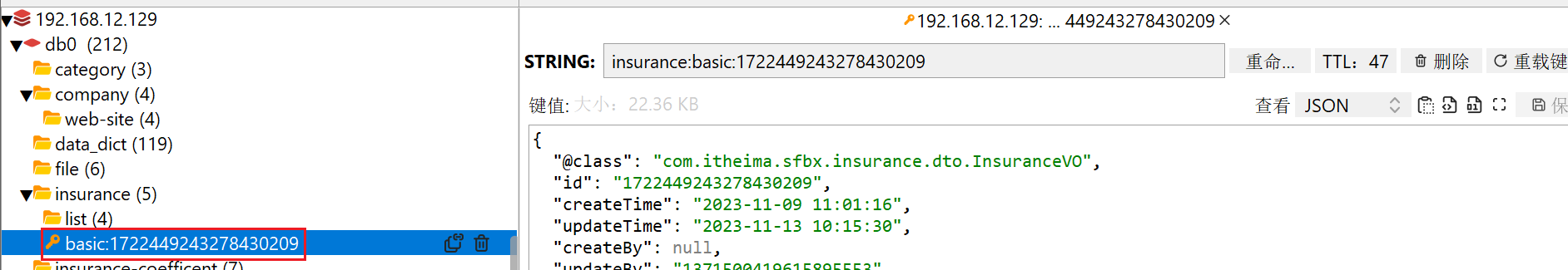


**2.2.4、保险产品缓存**

我们回到产品详情缓存优化这块；产品详情是由 insurance-app模块中的 ；其中 com.itheima.sfbx.insurance.web.ProductController#findInsuranceDetails 方法是处理根据产品id查询的处理器方法；它里面调用了 insurance-service 里面的 com.itheima.sfbx.insurance.service.impl.InsuranceServiceImpl#findById 方法；所以只需要在这个方法中添加缓存逻辑即可。



在app端访问任意一个产品之后；查看redis中存在对应缓存；说明产品缓存成功：



|  |
| --- |
| 注意：  四方保险中主要是在 frame-redis 模块中配置了基于redis的spring cache缓存。所以保存产品的缓存配置不再需要额外配置。只需要在有需要使用缓存的业务方法上方添加对应的缓存注解实现缓存即可。 |

**3、数据脱敏**

**3.1、需求分析**

数据脱敏（Data Masking），也被称为数据遮蔽或数据混淆，是一种数据安全技术，用于保护敏感信息免受未经授权的访问。数据脱敏是在互联网和金融等项目中广泛采用的一种安全措施。

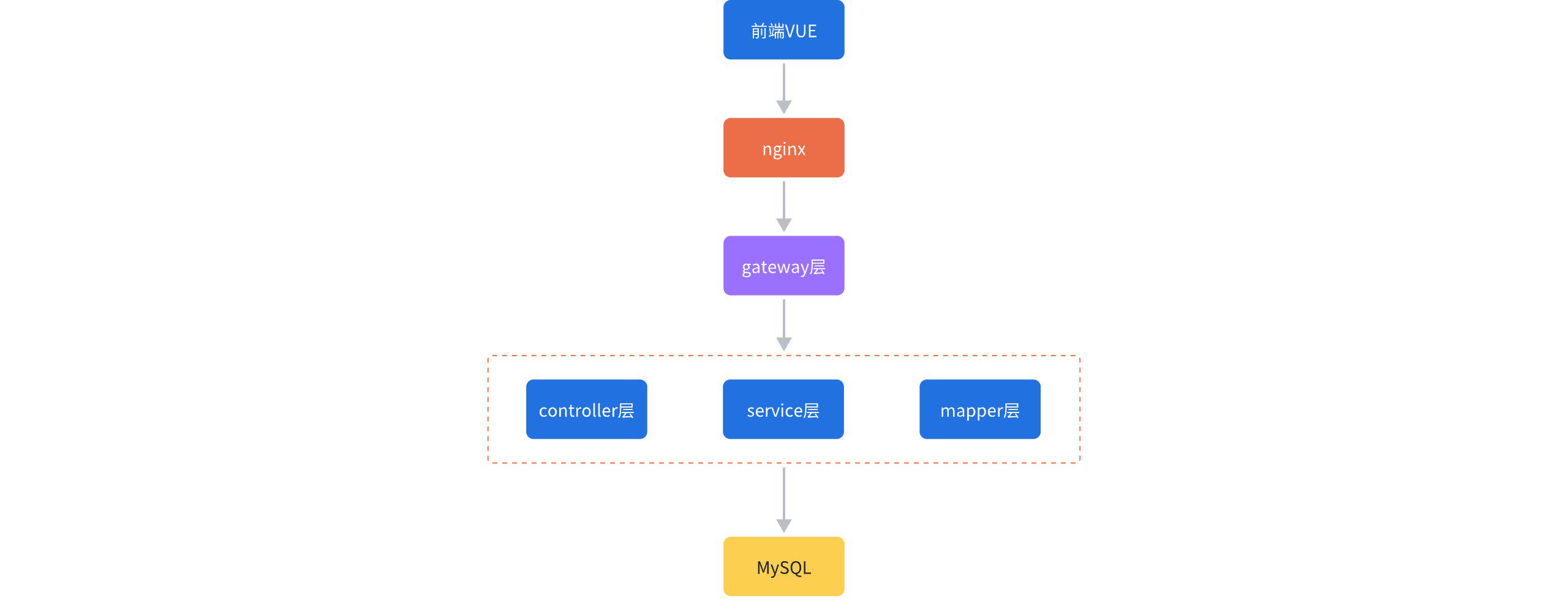
数据脱敏目的是保护敏感信息，防止数据泄露和滥用。脱敏是通过一系列的处理手段，将原始数据中的敏感信息替换为不具有敏感特征的数据，以降低数据的敏感程度。在《四方保险》中数据脱敏效果如下：



以下是一些常见需要进行数据脱敏的数据类型和情况：

* 个人身份信息：包括**姓名**、地址、**电话号码**、**电子邮件地址**、**身份证号码**等。
* 财务数据：信用卡号、银行账号、交易记录、薪资信息等。
* 医疗记录：患者姓名、病例、医疗历史、诊断、病人病例号等。
* 个人偏好信息：包括性别、性取向、宗教信仰、政治立场等。
* 客户数据：客户姓名、联系信息、购买历史、账户信息等。

下面我们从系统的不同层级，来分析下如何进行数据脱敏的设计处理，下图为一个微服务的调用层级：





**点击图片可查看完整电子表格**

这四方保险中我们选用自定义注解+AOP处理的方式来完成数据脱敏的实现，其主要的实现步骤：

* 自定义@SensitiveResponse注解，用于在需要进行数据脱敏的方法上标记切点
* 自定义SensitiveAspect切面处理被@SensitiveResponse标记的业务进数据脱敏
* 在业务的service方法上添加@SensitiveResponse注解

**3.2、功能实现**

**3.2.1、自定义注解**

查看 sfbx-cloud\sfbx-framework\framework-web\src\main\java\com\itheima\sfbx\framework\anno\SensitiveResponse.java 注解类内容如下：

|  |
| --- |
| Java package com.itheima.sfbx.framework.anno;  import java.lang.annotation.ElementType; import java.lang.annotation.Retention; import java.lang.annotation.RetentionPolicy; import java.lang.annotation.Target; *//注解只能用于方法声明，而不能用于其他地方，如类、字段等。* @Target(ElementType.*METHOD*) *//注解在运行时可见。这表示该注解的生命周期在运行时，可以通过反射在运行时获取到该注解的信息。* @Retention(RetentionPolicy.*RUNTIME*) public @interface SensitiveResponse {  } |

* @Target(ElementType.METHOD)：该注解用于标注在方法上。这表示该注解只能用于方法声明，而不能用于其他地方，如类、字段等。
* @Retention(RetentionPolicy.RUNTIME)：该注解在运行时可见。这表示该注解的生命周期在运行时，可以通过反射在运行时获取到该注解的信息。

**3.2.2、编写切面**

切面中主要处理的业务逻辑：

1. 得到执行方法之后结果
2. 对返回的结果进行解析为字符串
3. 再对该字符串中特殊的值进行替换
4. 将替换后字符转换为最终要返回的数据结果

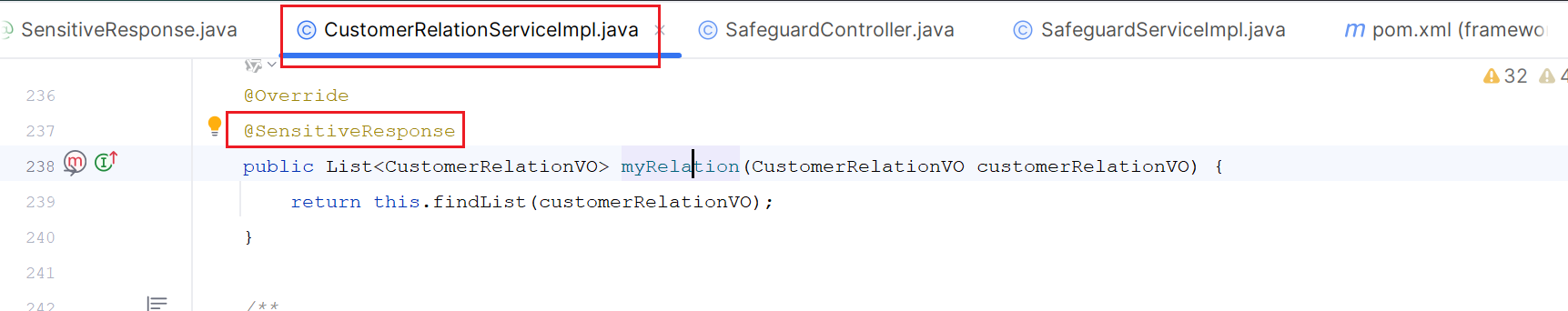
编写 sfbx-cloud\sfbx-framework\framework-web\src\main\java\com\itheima\sfbx\framework\aspect\SensitiveAspect.java 内容如下：

|  |
| --- |
| Java package com.itheima.sfbx.framework.aspect;  import com.alibaba.fastjson.JSONObject; import com.itheima.sfbx.framework.anno.SensitiveResponse; import com.itheima.sfbx.framework.commons.utils.EmptyUtil; import org.aspectj.lang.ProceedingJoinPoint; import org.aspectj.lang.annotation.Around; import org.aspectj.lang.annotation.Aspect; import org.aspectj.lang.reflect.MethodSignature; import org.springframework.stereotype.Component;  import java.lang.reflect.Type;  */\*\**  *\* 敏感数据处理切面*  *\*/* @Aspect @Component public class SensitiveAspect {   */\*\**  *\* 针对查询方法中使用了 @SensitiveResponse注解的返回结果中敏感数据处理*  *\*/* @Around("@annotation(sensitiveResponse)")  public Object sensitiveResponse(ProceedingJoinPoint joinPoint, SensitiveResponse sensitiveResponse) throws Throwable {  *//1、执行原方法；获取执行结果* Object result = joinPoint.proceed();  *//2、获取执行的原始方法；并获取其返回类型* MethodSignature methodSignature = (MethodSignature) (joinPoint.getSignature());  Type returnType = methodSignature.getMethod().getGenericReturnType();  if (!EmptyUtil.*isNullOrEmpty*(result)) {  *//3、将返回结果转换为json字符串* String jsonString = JSONObject.*toJSONString*(result);  *//4、在处理敏感字符的方法中对返回结果处理并返回处理后字符串* String resultJsonStr = maskSensitiveData(jsonString);  *//5、将处理后的字符串转换为返回类型* return JSONObject.*parseObject*(resultJsonStr, returnType);  }  return result;  }   */\*\**  *\* 敏感数据处理 \"(.).+?\" 表示匹配双引号内的第一个字符 $1也就是表示这第一个字符*  *\* @param jsonData json字符串*  *\* @return 处理过后的json字符串*  *\*/* private String maskSensitiveData(String jsonData) {  return jsonData  .replaceAll("\"identityCard\"\\s\*:\\s\*\"(\\d{1})(\\d{15})(\\d{2})\"", "\"identityCard\":\"$1\*\*\*\*\*\*\*\*\*\*\*\*\*\*$3\"")  .replaceAll("\"name\"\\s\*:\\s\*\"(.).+?\"", "\"name\":\"$1\*\"")  .replaceAll("\"bankCardNo\"\\s\*:\\s\*\"(.).+?\"", "\"bankCardNo\":\"$1\*\"")  .replaceAll("\"bankName\"\\s\*:\\s\*\"(.).+?\"", "\"bankName\":\"$1\*\"")  .replaceAll("\"bankReservedPhoneNum\"\\s\*:\\s\*\"(.).+?\"", "\"bankReservedPhoneNum\":\"$1\*\"");  } } |

**3.2.3、应用**

可以找app端中的我的家人数据进行应用测试；那么需要在对应的查询方法上添加上述注解；如下：

找到 sfbx-cloud\sfbx-insurance\insurance-service\src\main\java\com\itheima\sfbx\insurance\service\impl\CustomerRelationServiceImpl.java 我的家人的查询方法上添加注解：



测试：app端中；进入 我的——》我的家人 查看信息：



**4、作业&问题**

回答并总结下述问题：

* 你在项目中做过优化吗？哪些优化，具体如何做的？
* 说一个你在项目中做过的业务
* 什么叫 详情页面静态化？怎么做？
* 常见的缓存组件有哪些？
* 四方保险中哪些场景的数据做了数据脱敏，怎么做的
* 如果数据库被人盗查会泄露用户的隐私信息；怎么有效防止隐私泄露？在哪些数据上做了脱敏？
* **简历职责-候选**：解决APP端保险产品详情页面数据加载过慢问题，从原详情页面查询响应时间700ms，优化后查询响应时间在50ms内，查询效率提升93%+；
* **简历职责-候选**：负责APP端中保险详情页面被投保人（本人、配偶、父母、子女）及个人中心的数据脱敏

课堂练习答案参考：

|  |
| --- |
| SQL  -- SQL练习题 -- 1、模拟app端进行筛选查询保险产品；查询 为谁买 为 本人、配偶 的保险产品信息（可使用 JSON\_CONTAINS(json字段, '"查询值"') ）——》保险产品列表 select t1.\* from tab\_insurance t1 left join tab\_insurance\_condition t2 on t2.insurance\_id = t1.id where  JSON\_CONTAINS(t2.condiction\_val, '"BZ"') AND JSON\_CONTAINS(t2.condiction\_val, '"PO"') and condiction\_key\_name in('为谁买')  --#根据产品id查询保险产品基本信息 Select \* from tab\_insurance where id = ?;  --根据产品id查询保险方案 Select \* from tab\_insurance\_plan where insurance\_id = ?;  --根据产品id查询保障项 Select s.\* from tab\_plan\_safeguard s, tab\_insurance\_plan p where p.id = s.plan\_id and p.insurance\_id = ?;  --根据产品id查询给付方案 Select e.\* from tab\_plan\_earnings e, tab\_insurance\_plan p where p.id = e.plan\_id and p.insurance\_id = ?;  --根据产品id查询保险系数 Select \* from tab\_insurance\_coefficent where insurance\_id = ?;  --根据产品id查询附件 Select \* from tab\_file where business\_id = ?;  --根据产品id查询筛选项 Select \* from tab\_insurance\_condition where insurance\_id = ?; |