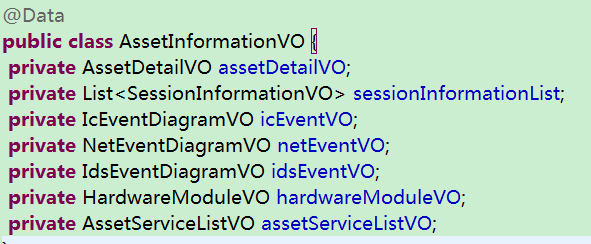
JAVA 接口设计文档

-----------------------------刘磊

# 设计返给前端对象VOs

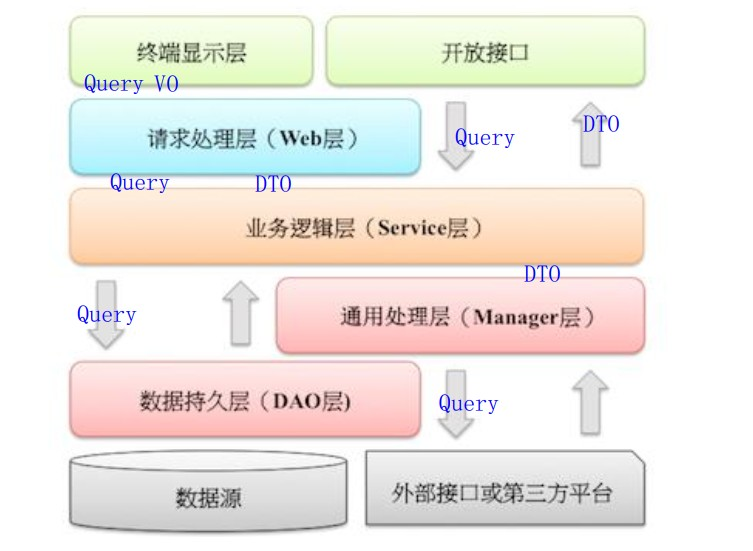
根据前端界面设计元素定义VO对象，前端界面中的数组在Java端为集合显示。如果返给前端的是一组字段，那么可以定义一个VO对象来返回；如果返给前端的是一组数据，可以返回list集合，集合中的每一组数据可以定义成VO对象，如果在返回该组数据的同时还有别的逻辑处理，那么可以定义一个大VO，里面定义一个小VO的list,再添加别的字段。

例子：



上面AssetInformationVO是返给前端的大VO，里面包含各种小VO以及其他处理。

# 公司项目实体对象分类

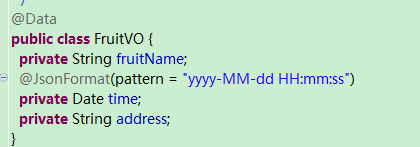
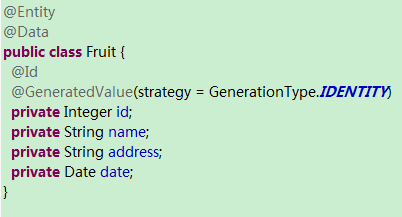


# JPA查询(Repository)

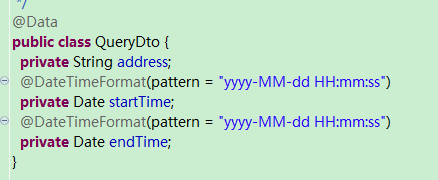
## 方法体接受Query对象

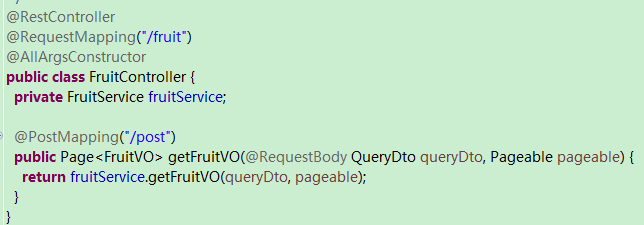
针对前端传的参数，设计一个Query对象去接受

1. 定义一个实体类与VO对象



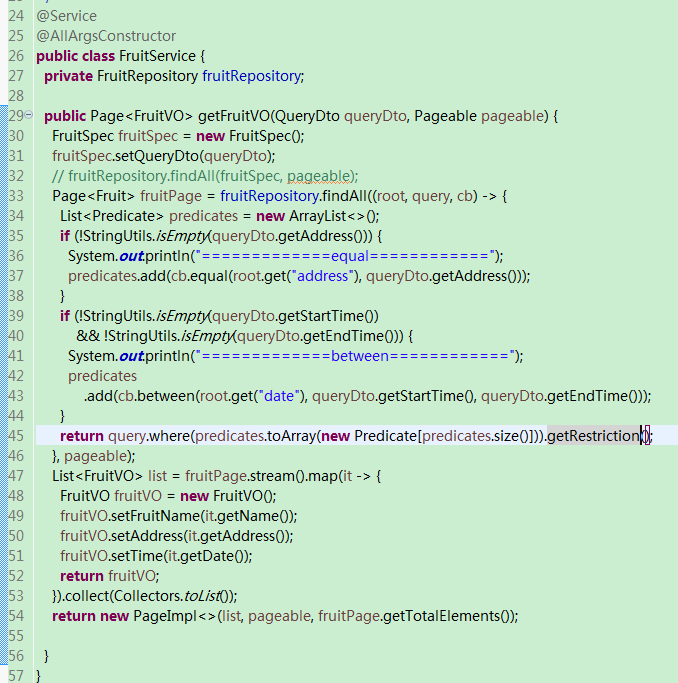
1. 定义控制层





控制层采用Post的形式接受请求参数封装的对象，由于是自定义封装请求参数，所以允许请求对象的字段与域对象(实体类)不一致，但是需要在service层进行转换，因此建议query对象的字段与实体类保持一致，针对例如前端请求参数是IP，格式是点分十进制，而数据库是Long类型，那么这个最终还是需要在service层对数据类型进行转换。

1. Service层



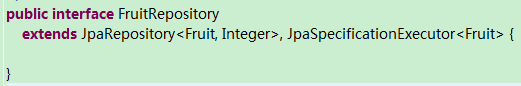
分析：

* 第33行，findAll()---findAll(@Nullable Specification<Fruit> spec, Pageable pageable，接受两个参数，一个是spec，它是一个接口的实现对象，该接口是函数式接口，因此采用lombda形式重写toPredicate()方法，当然针对多种请求对象的场景，可以自定义类去实现该接口，多态思想处理多种场景。例如：



* 如果要实现动态查询，需要人为对参数进行非null处理。

1. Dao层



## 方法体接受@QuerydslPredicate

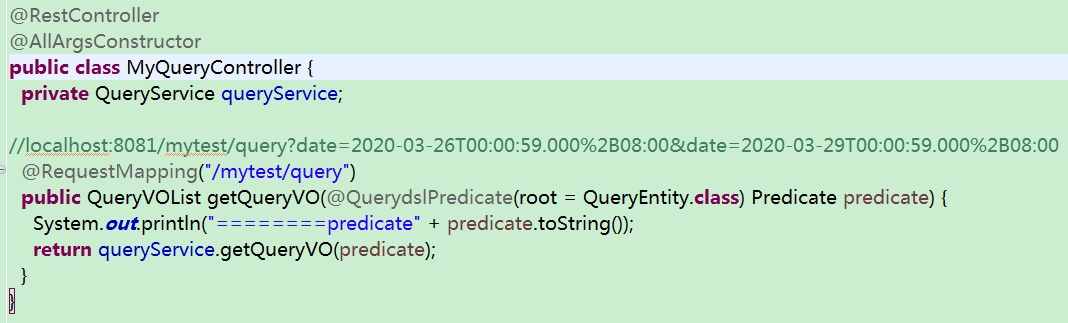
因为Controller层传入的参数是(@QuerydslPredicate(root = XXX.class) Predicate predicate),XXX表示请求参数所对应的字段所属的实体类，因此采用这种方式，前端传给后端的url中的请求参数字段必须与后端实体类字段名保持一致，因为这种方式无法做到字段转换

代码样例分析：

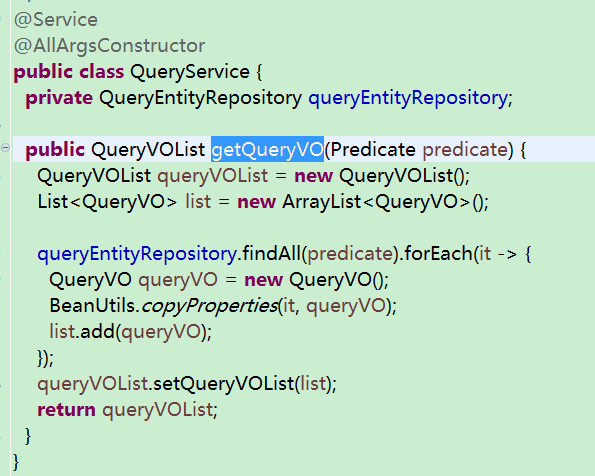
实体类 返回VO类

|  |  |
| --- | --- |
| C:\Users\lenovo\AppData\Roaming\feiq\RichOle\2959217548.bmp | C:\Users\lenovo\AppData\Roaming\feiq\RichOle\3713496725.bmp |

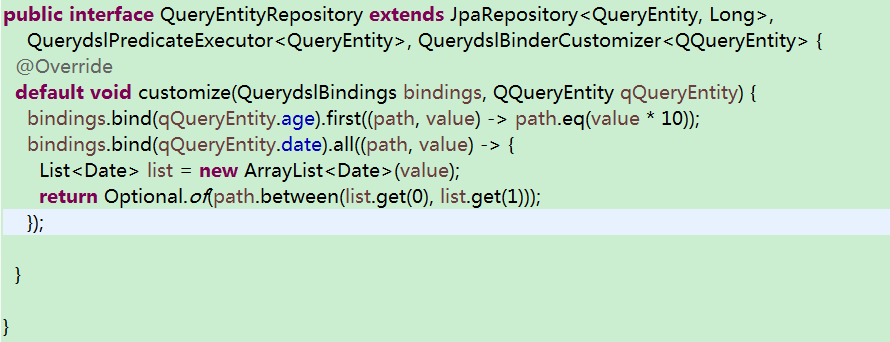
控制层：



服务层：



DAO层：



分析：

由于采用@QuerydslPredicate这种接受方式，前端输入的字段必须与Java实体类保持一致，并且在控制层已经生成predicate对象，那么就无法对请求参数的类型或者名称进行改变，但是可以对值的大小进行改变，比如说像上面的例子中请求参数age的输入值都会进行\*10后才与数据库中age字段的值进行比较，同时这种方式构建predicate默认对所有字段进行equals处理，如果进行的是betwee、or、contain等处理，都可以继承或实现QuerydslBinderCustomizer这个接口，在定制方法customize中进行处理，注意，这里无法实现一个场景，就是当前端传给一个请求参数，而后端需要用两个字段对它进行or处理，因为这里绑定的只能是一个参数bindings.bind(XXX)，XXX只能是一个字段。

同时注意两种方式：

1. customize定制方法中可以进行多项处理，如例子中所示，在动态查询中，也可以跳过定制方法中的请求参数。

2.如果一个实体类对应的Repository可能参与多种定制，那么就不可以将customize写在Repository的接口中，因此这种方式不适合单个Repository多种定制的情况。

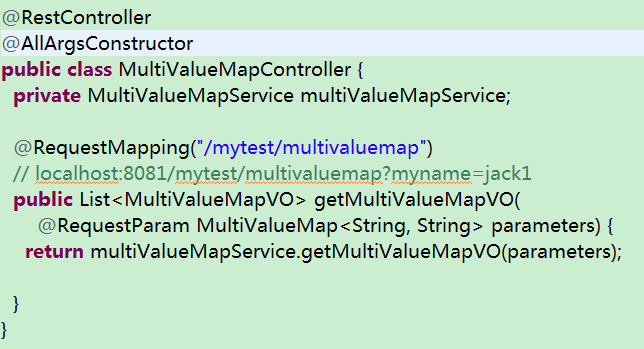
3.采用这种方式在控制层调用service层之前，就会先去执行实体类对应的Repository中的定制，最终生成predicate对象，接下来才是XXXRepository.find(predicate)等其他方法的调用。

## 方法体接受@RequestParam MultiValueMap

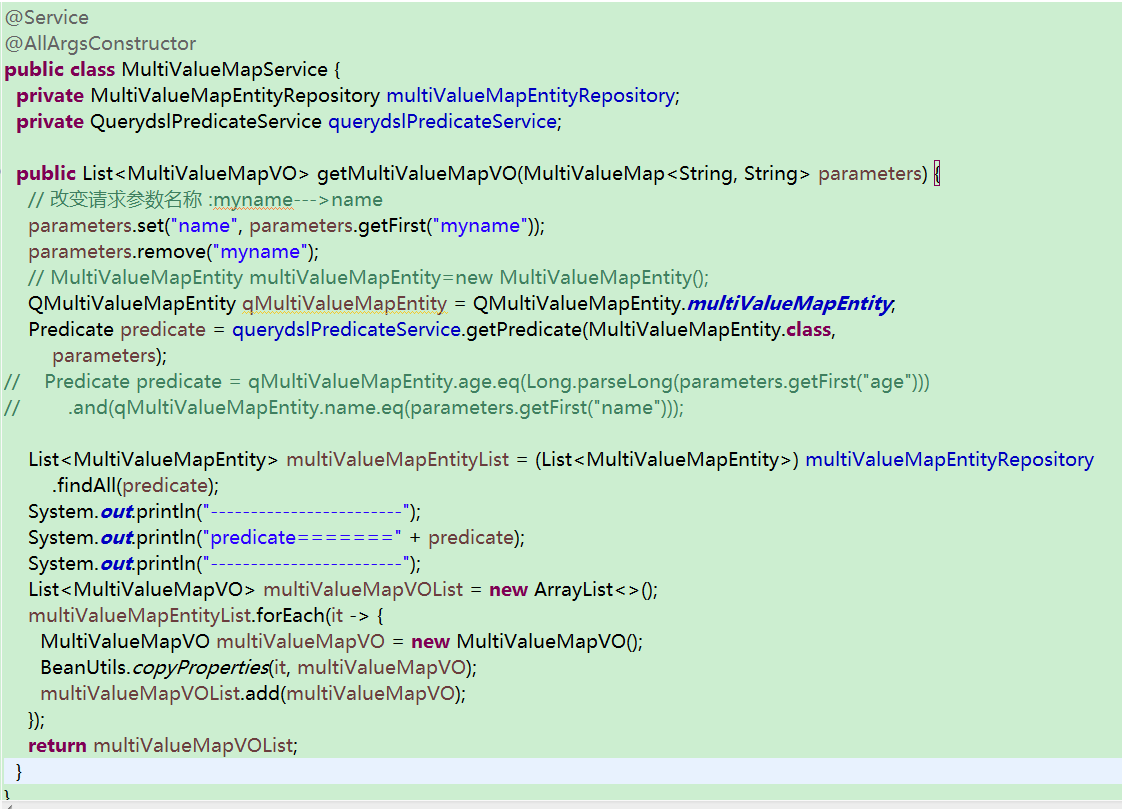
实体类 返回VO类

|  |  |
| --- | --- |
| C:\Users\lenovo\AppData\Roaming\feiq\RichOle\1228950365.bmp | C:\Users\lenovo\AppData\Roaming\feiq\RichOle\152444259.bmp |

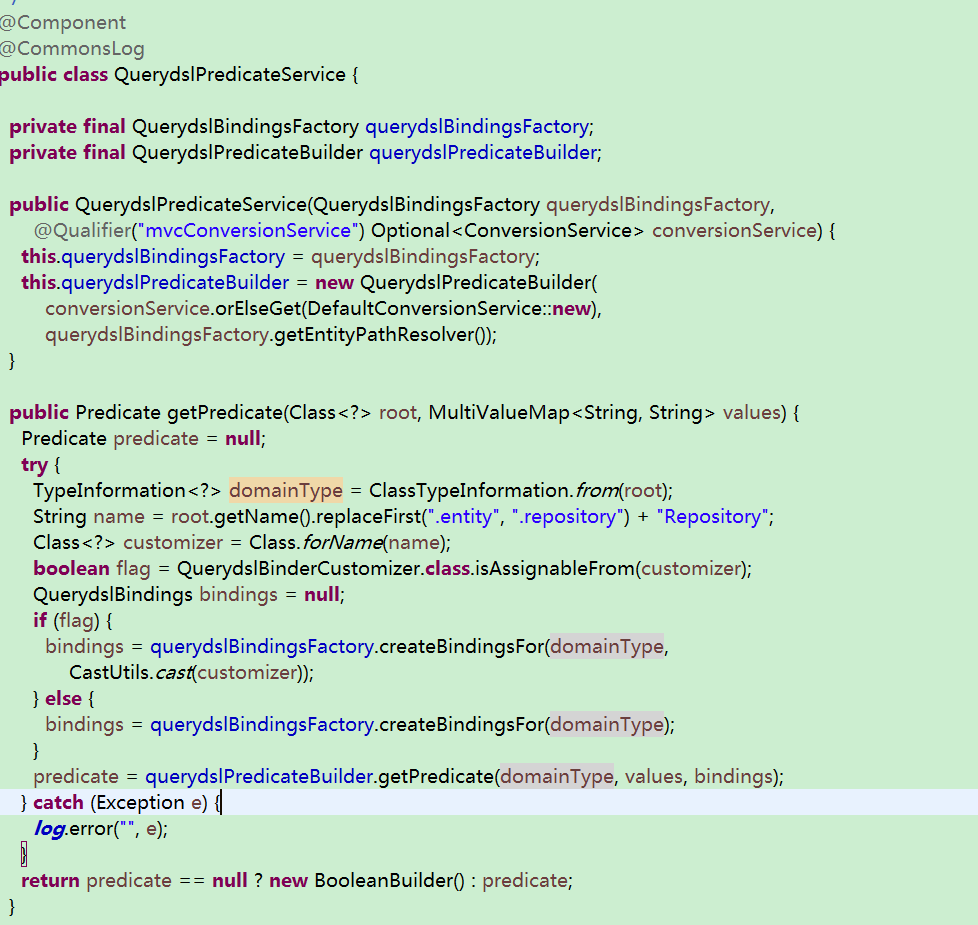
控制层：



服务层：



注意：上述代码中注解的方式是人为去构建predicate，那么为了支持动态查询，需要对查询条件进行非null判断，一般采用StringUtils.isNoneBlank()的方式。如果前后端决定所传请求参数不可以缺省，前端如果对该字段进行默认处理，也需要传一个默认值给后端，那么人为构建predicate的时候需要对该字段进行额外处理，如果接受到该字段的值是默认值，那么就对该字段不进行检索处理，即生成的predicate中不包含该条件。



**注意**：

-----------------------------------------------------------------------------------------------------------------

**roo**t:需要搜索的实体类对应的class文件

**values**: parameters

**domaintype**:包含该实体类中所有的字段名称与类型

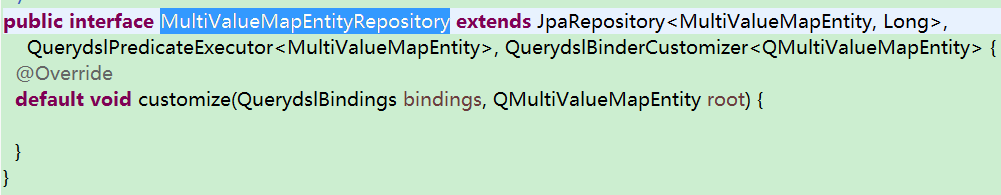
**customizer**:通过反射机制获取实体类对应的repository接口,主要是为了获取在repository定制默认方法。

**bingdings**：构建绑定

**getPredicate(domainType, values, bindings):**对请求参数的逻辑请求进行处理，包括eq、between、contain等，框架会根据字段的类型调整逻辑处理方法，默认采用eq处理，如果不需要采用定制，那么repository不需要继承QuerydslBinderCustomizer,同时在这里会对请求参数与实体类的字段类型进行绑定。

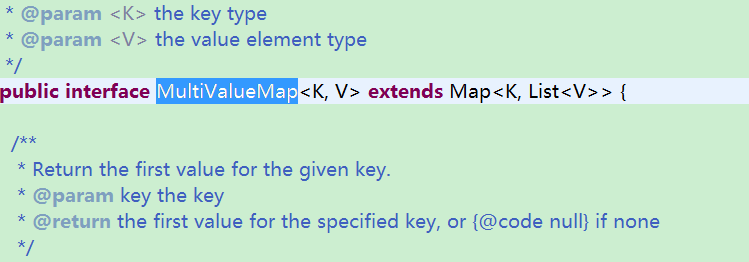
-----------------------------------------------------------------------------------------------------------------

DAO层：



分析：

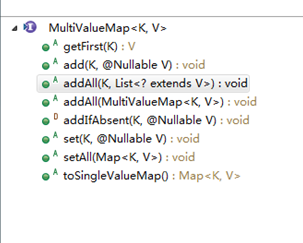
采用@RequestParam MultiValueMap<String, String> parameters方式获取请求参数，那么请求参数会以map的形式存储到parameters中，map的key是字段名称，map的值是List<String>的集合。(注意：这里采用的map的值所属类型定义为String)，下面通过map的源码来展示一下该过程：



分析前后端数据流转：

1. 前端在url中输入请求参数，用这种方式前端传给的请求参数的名称、类型以及值都可以与后端实体类(数据库字段)不一致。
2. 后端用map去接受请求参数，key是请求参数的名称，value是List集合。
3. 如果需要对请求参数进行转换，包括请求参数的名称，类型(例如前端请求参数IP是String类型的，数据库中是Long类型的，那么需要对请求参数的类型转为Long，不过这里只涉及形式的改变，最终存在map里的都是String格式的)以及值的改变。

Parameters的API:



1. 通过一系列改变操作，最后生成的Parameters包含的请求参数要与实体类字段一一对应，名称必须一致，否则无法完成最后的绑定操作，注意：最后生成的Parameters字段对应的都是List<String>,也就是说请求参数的值都是String类型，在步骤4中将完成Parameters字段类型与后端实体类的类型一一对应转换。

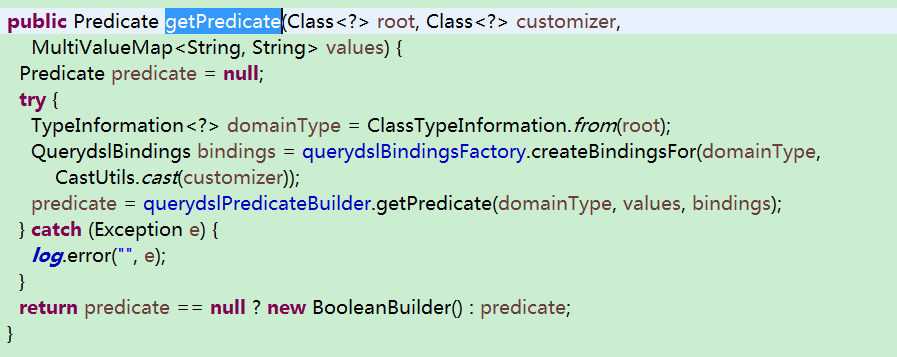
注意：这里如果是Date类型的值，那么从String—>Date的过程会涉及格式的转换，因为前端请求参数中日期字符串有格式，因此需要我们定义好转换格式，保证框架在帮你做字段一一映射的时候可以采用这种方式，如果后端不做额外处理，那么前端时间格式也是默认格式，需要按照Mon Mar 30 18:19:44 CST 2020类似这样的格式来定义，所以我们可以在实体类日期字段上添加类似@DateTimeFormat(pattern = "yyyy-MM-dd'T'HH:mm:ss")这样的注解或者在配置文件中添加spring.mvc.date-format=yyyy-MM-dd'T'HH:mm:ss，一般采用后面一种。

1. querydslPredicateService.getPredicate(MultiValueMapEntity.**class**, parameters)最终生成predicate对象，可以供jpa或者JPAQueryFactory作为过滤条件调用。

**注意**：

-----------------------------------------------------------------------------------------------------------------

* 在动态查询中，若某个请求参数缺省，那么map查询会跳过该字段，如果前端对url的请求参数进行统一，用户不填的请求参数也会返给后端一个默认值，如果默认值是“”,那么predicate也会跳过该项过滤条件。若前端传给后端的默认值是非“”,那么后端在接受该请求参数的时候需要判断该值，若equals==true，那么需要人为跳过该项判断，在parameters中去掉该项过滤条件，因此为了提高代码的简洁性，可以与前端协商处理该情况时默认采用“”。
* 为了解决在上一种方式中出现的一个Repository只能定义一个customize的问题,可以根据业务场景自己定义一个类，去实现QuerydslBinderCustomizer接口，然后在构建predicate的时候采用以下方式：



生成predicate的时候，传入自己定义的customize的字节码文件。

**缺点**：在面对如下场景时：前端请求参数为IP，需要后端去匹配源IP和目的IP对应的所有资源，那么这种方式包括上一种方式 (@QuerydslPredicate)都无法做到，因为在做逻辑处理的时候，bindings.bind(Qxxx.xxx)只能绑定一个参数，无法进行or的处理，因此这种场景只能自己去定义predicate，去进行or的处理

## 分页排序

使用Jpa自带的Repository.findAll(Predicate predicate,Pageable pageable)，可以直接接受由前端界面传过来的page参数进行分页，如果不采用该方式或者采用该方式，但是将生成的Page对象转为流对象进行额外操作，生成最后的VO对象的List集合，那么不可以直接用。

采用Jpa间接方式：



1

2

* 由于不是直接将fandAll的对象直接返给前端，而是进行了额外的逻辑处理，那么不可以直接将生成的List的size()赋给PageImpl，需要进行额外处理，如步骤1、2所示。

其实，对于Page分页，我们需要返给前端的参数有三个：

1. 当前页码
2. 每页元素个数
3. 元素总个数

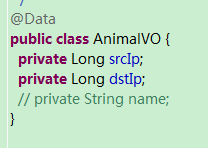
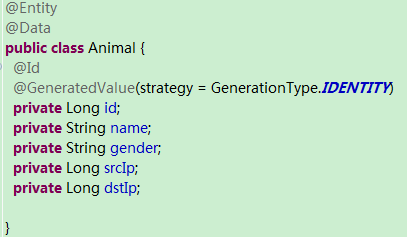
直接findAll会将这三个元素直接封装返给前端，其他处理需要自己单独组装。

* 不采用Repository.findAll(Predicate predicate,Pageable pageable):下一章节介绍。

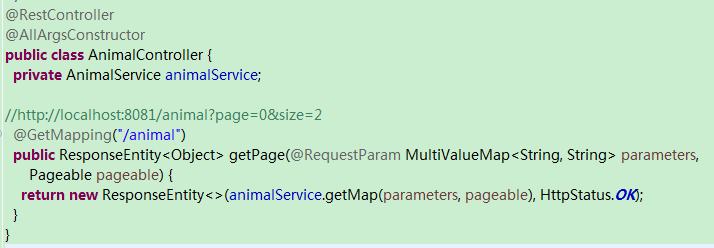
# JPAQueryFactory(QueryDsl)

QueryDsl仅仅是一个通用的查询框架，专注于通过JavaAPI构建类型安全的Sql查询，也可以说是QueryDsl是基于各种ORM框架以及Sql之上的通用的查询框架，QueryDsl的查询，类似于Sql查询。

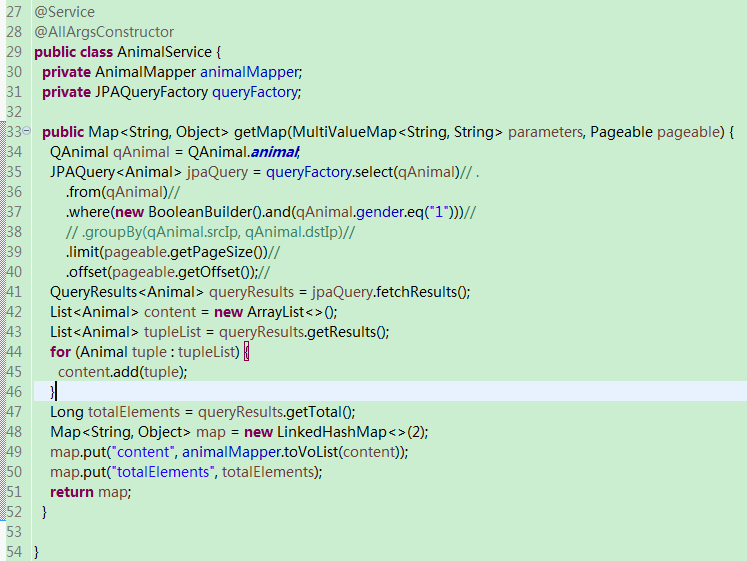
1. 实体类与Vo类



1. Controller层



1. Service层



上述展示了分页处理。

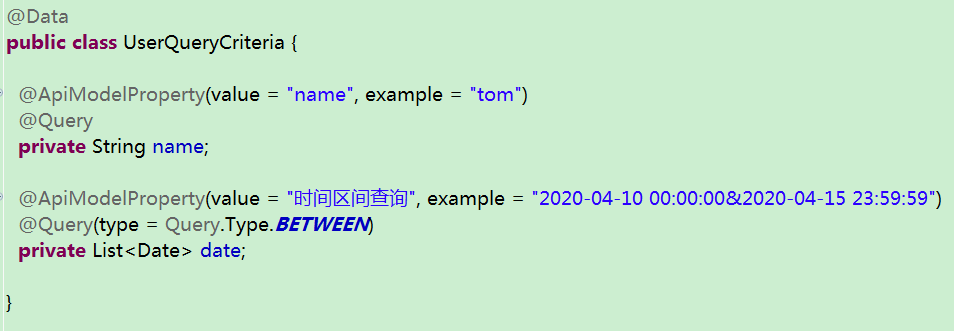
# 反射+自定义注解

## 思路

1. 自定义Query类，定义属性是前端获取的请求参数。
2. 自定义注解，注解类型元素根据场景自定义，一般包含基本逻辑处理，包含equals、between、or等基本处理。
3. 在第一步自定义的Query类中每一个属性上添加自定义注解，表示每一个字段需要进行什么逻辑处理。
4. 通过前端传参封装成Query对象，并通过反射获取该对象、该对象的所有字段以及该字段上的注解内容，进行逻辑处理，获取Specification对象(repository需要继承JpaSpecificationExecutor接口 )

## Demo

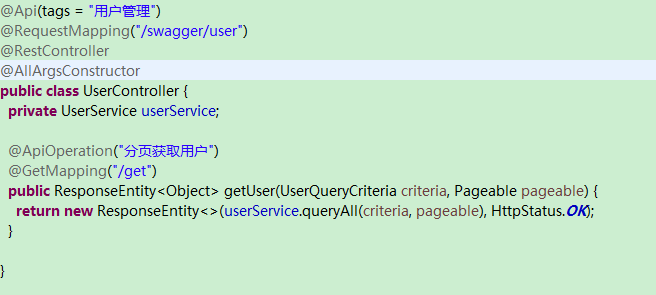
1. 自定义Query类：



1. 自定义注解：



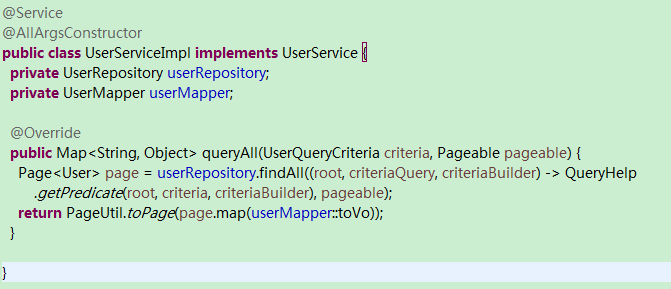
1. Controller层：



1. Service(抽离成接口)：



1. Service接口实现：



1. 请求对象解析：



解析：

1. 第30行if判断前端是否传参数，若没有传参，则query对象为null,过滤条件为空，直接返回所有数据。
2. 第34行根据query对象的字节码文件反射获取该对象所有字段，根据getAllFields(Class class,List<Field> fields)方法(后面代码180行)获取所有属性字段，包括public、private、protected等，该方法还采用递归的方式获取父类的属性，后面介绍。
3. 遍历所有字段，由于字段有些是非public修饰，所以采用：

第41行：field.setAccessible(**true**)

取消Java语言访问检查，可以接受非public的字段类型不报错。

1. 第43行：根据每一个字段获取它的@Query注解。
2. 第49行是判断该字段是否有数据库别名，如果有attributeName为该别名，如果没有，默认是空字符串，那么attributeName是该字段名。
3. 第50、51行获取该字段的类型与值。
4. **动态查询**过滤：如果没有值就说明该字段不需要参与构建过滤条件，直接continue，跳到下一个字段。



1. 第94行注释内容



1. 第140行as((Class<? extends Comparable>) between.get(0).getClass())：

是因为between中的是List,参与泛型限定的不是List类型，而是List中的元素类型，所以任取其一。



1. **第167行非空判断**(学习思想)：

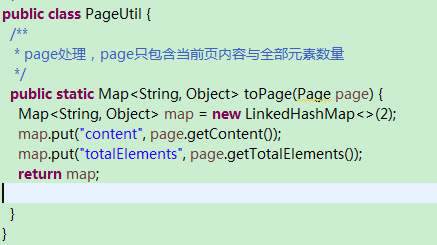
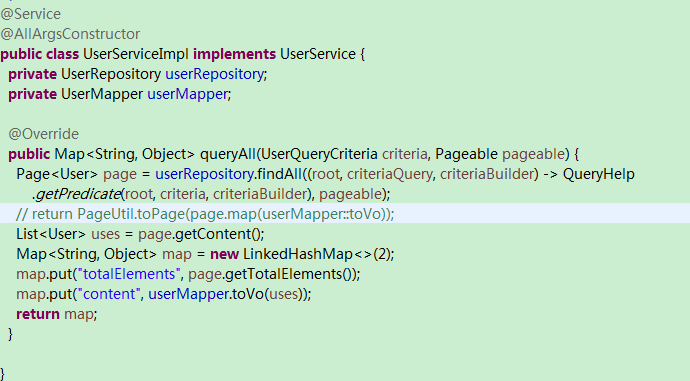
对字符串进行非空判断，String继承CharSequence接口：

* 如果String是null，直接返回true
* 如果不为null,在||第二个条件strLen = cs.length()获取长度，若长度是0，也直接返回true
* 若长度不是0，有两种情况，一是空字符串，二是有值，在第二步for循环中若任意一个字符不为空，则返回false。

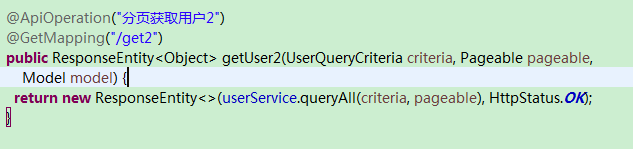
1. 获取到Specification对象后，serviceImpl获取page页，由于可能只是需要部分字段或者需要对字段进行转换，如数据库中IP Long类型转换为String。请看4.4小节。

## 返给前端

由于返给前端的数据字段一与数据库的字段有可能会存在类型或者值等差异，因此我们会定义Vo对象，且返给前端的数据(Get请求)要么是集合或者分页，以分页说明，前端要的数据就是分页的内容，以及元素的全部数量，因此我们可以定义一个工具类map，里面只有两个字段contenet与totalElements用于存放这两部分数据。

红框部分直接返给controller层。



## 实体映射工具(MapStruct)

---------------------------------------------------------------------------------------------------------------------------------

首先声明：

* Query ：用于接受前端请求参数。
* DO：业务实体对象。
* DTO：数据传输对象。
* VO:最后返给前端的对象

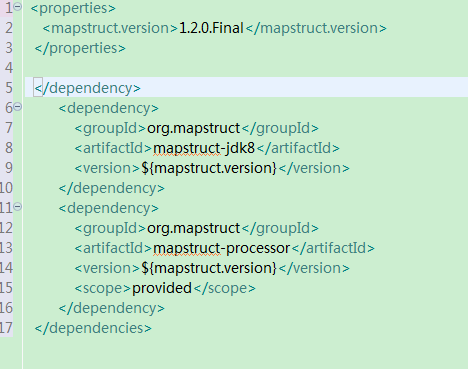
注意：如果逻辑处理中没有用于数据传输，可以将DTO和VO 合二为一。

---------------------------------------------------------------------------------------------------------------------------------

在一个成熟的工程中，尤其是现在的分布式系统中，应用与应用之间，还有单独的应用细分模块之后，DO一般不会让外部依赖，这时候需要在提供对外接口的模块里放入DTO用于传输对象，也就是DO对象对内，DTO对象对外，DTO可以根据业务的需求变更，并不需要映射DO的全部属性。

这种对象与对象之间的互相转换，就需要有一个专门用来解决咋转换问题的工具，那就不需要每一个字段都get/set。MapStruct就是这样的一个属性映射工具，只需要定义一个Mappper接口，MapStruct就会自动实现这个接口，避免了复杂繁琐的映射实现。

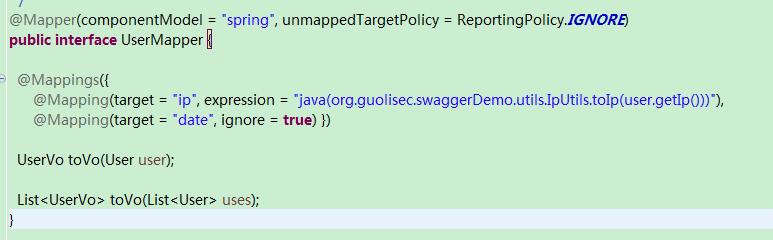
### 工程中引入maven依赖



### 定义DO、VO(DTO)

|  |  |
| --- | --- |
| C:\Users\lenovo\AppData\Roaming\feiq\RichOle\1590137254.bmp | C:\Users\lenovo\AppData\Roaming\feiq\RichOle\3125193827.bmp |

### 定义mapper接口



注意：接口里面有两个方法，一个是单体实体映射，另外一个是List映射，如果某个属性不想从数据库对象中直接获取，可以指定为ignore=true。



注意：

1. 该实现类是自动生成，手动编译或者启动IDE的时候IDE会自动生成，在target/classes下生成对应的实现类。
2. 第28行就是因为忽略了date属性，那么就可以在mapper的实现类中自定义，这里是默认存的是一个固定时间。

---------------------------------------------------------------------------------------------------------------------------------

重点：

1. MapStruct可以将多个对象映射我一个VO或者DTO

例如：UserVo toVo(User user，Person person)--------->将user和person映射到UserVo对象。

1. Spring注入：可以通过添加@Mapper(componentModel = "spring")将该mapper接口的实现类对象添加到容器中，也就是最后自动生成的MapperImpl会添加@Component注解。
2. 关键词
3. @Mapper只有在接口加上这个注解，MapStruct才会去实现该接口。
4. @Mapper里面有个componentModel属性，主要是指定实现类的类型，一般用到两个

default:默认，可以通过Mappers.getMapper(Class)方式获取实例对象。

spring:在接口的实现类上自动添加注解@Component ,可以通过@AutoWired方式注入。

1. @Mapping:属性映射，若原对象属性与目标对象属性名字一样，会自动映射对应属性

source:源属性

target:目标属性

dateFormat：String到Date日期之间互相转换。

Ignore:忽略这个字段。

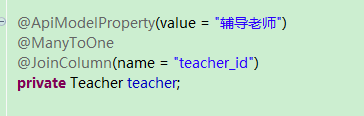
1. @Mappings:配置多个@Mapping。
2. @MappingTarget:用于更新已有对象。
3. @inheritConfiguration：用于继承配置。

---------------------------------------------------------------------------------------------------------------------------------

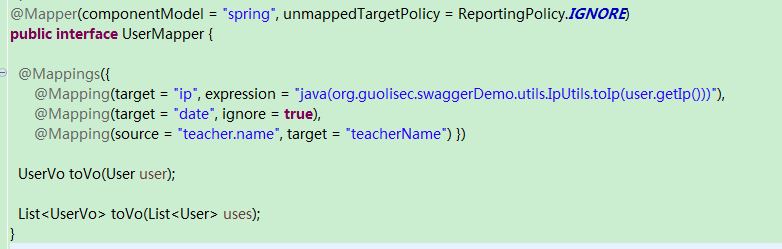
### VO存在多表关系

应用场景：如果表A中有一个字段关联表B，也可以说是A通过外键去关联B的主键。返回的VO对象中多数是A的字段，还有部分是B的字段，平时操作是先取出A的外键，然后拿着外键去找对应的B对象，然后取出B中的属性赋给返回对象VO，现在为了代码的优雅，可以在mapper中定义这些，但是底层还是采用上述逻辑(查两次)，实例如下。

* User类中添加关联表。



* UserMapper接口



* UserMapperImpl



注意：其实底层还是调用了两次数据库

C:\Users\lenovo\AppData\Roaming\feiq\RichOle\3680597417.bmp

但是如果不加上@ManyToOne注解，那么User表中只存了一个字段，值是Teacher表的主键，需要先查出Uesr实体，然后取到该实体对应对应的teacher\_id，然后调用repository.findById(),取出teacher中对应的字段赋给VO对象，这样代码比较冗余，不够优雅简洁，采用mapper方式代码优雅很多，底层还是采用一样的方式。

### 界面update和save(vo-->entity)

应用场景：前端界面update或者save的时候，更新或者新增对象包含了两张表表的数据，且该两张表的数据还有关联，那么可以用mapper。

|  |  |
| --- | --- |
| C:\Users\lenovo\AppData\Roaming\feiq\RichOle\176290031.bmp | C:\Users\lenovo\AppData\Roaming\feiq\RichOle\483485102.bmp |



分析：前端界面在更新或者新增对象的时候，如果该对象包含了数据库的两张表，比如User和Teacher表，那么根据mapper的映射，它会自动将数据存到这两张表，当然这里需要用到级联操作。

# 数据库操作

## 外键+外键约束+外键索引

首先给出定义：

* 外键：可以成为关联字段，与是否存在外键约束、外键索引无关
* 外键约束：外键约束是为了防止产生脏数据，若A表关联B表，不可以随意删除B表的数据，因为有A表引用，但是可以删除A表中的数据
* 外键索引：添加外键的时候默认会加上索引，该索引就是普通索引。
* JPA级联操作与数据库层面的外键索引无关，数据库外键索引去除，只要代码层面添加级联，去掉外键索引，那么JPA依然会对表间关系进行映射处理(外键字段还在)。

C:\Users\lenovo\AppData\Roaming\feiq\RichOle\1799789879.bmp

## Jpa关联注解

Jpa关联关系注解包括：@JoinColumn、@OneToOne、@OneToMany、@ManyToOne、@ManyToMany、@JoinTable、@OrderBy。

## 级联操作

Cascade是级联操作策略：

* CascadeType.PERSIST :级联新建，级联持久化(保存)操作
* CascadeType.REMOVE :级联删除，删除当前实体时，与它有关联的实体也会跟着被删除。
* CascadeType.REFRESH :级联刷新
* CascadeType.MERGE :级联更新
* CascadeType.DETACH :级联托管/游离操作，如果你要删除一个实体，但是它有外键无法删除，那就需要这个级联权限，它会撤销所有相关联的外键关联。
* CascadeType.ALL :全选

## 加载方式

可以在关联注解上标注加载方式，一般加载方式有两种：1是立即加载EAGER –2是懒加载LAZY。

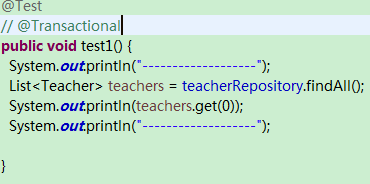
fetch=FetchType.LAZY(EAGER)

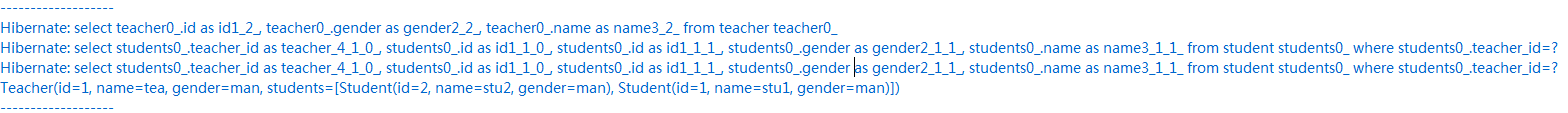
懒加载机制LAZY和实时加载EAGER的概念，在各种开发语言中都有广泛应用。其目的是实现关联数据的选择性加载，懒加载是在属性被引用时，才生成查询语句，抽取相关联的数据，而实时加载则是执行完主查询后，不管是否被引用，立马执行后续的关联数据查询。

注意：使用懒加载来调用关联数据，必须保证主查询session(数据库连接会话)的生命周期没有结束，否则你是无法抽取到数据的，解决方式是采用事务管理@Transactional来管理。

|  |  |
| --- | --- |
| C:\Users\lenovo\AppData\Roaming\feiq\RichOle\1833277364.bmp | C:\Users\lenovo\AppData\Roaming\feiq\RichOle\3185478400.bmp |

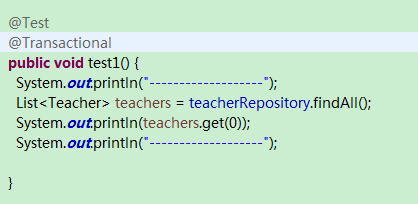
* FetchType.EAGER

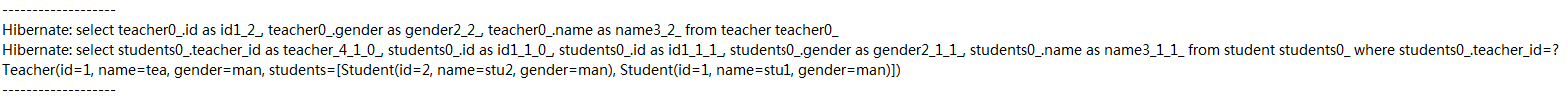




* FetchType.LAZY

|  |  |
| --- | --- |
| C:\Users\lenovo\AppData\Roaming\feiq\RichOle\1833277364.bmp | C:\Users\lenovo\AppData\Roaming\feiq\RichOle\705464897.bmp |





**@Transactional**注解：解决了懒加载session关闭的问题。

注意：从最后输出的sql语句来看，及时加载对输出了一条子查询，那是因为它在调用完主查询后就立即调用子查询，后来在调用第一条数据时再触发一次，而懒加载是在调用第一次查询才会触发子查询，但是即便对于懒加载而言还是会存在N+1的问题，下一小节给出Jpa解决方案。

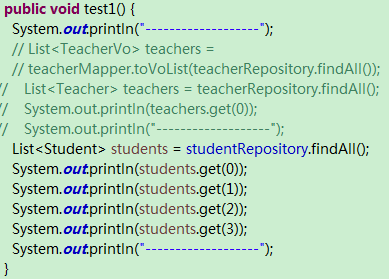
## N+1 sql问题

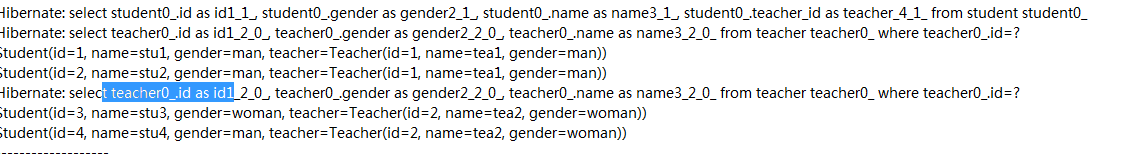
N+1的问题一般是出现在单向的多对一时触发，举例说明：Teacher表与Student是一对多的关系，且表关联方向是根据多的一方(Student)去查一的一方(Teacher)，那么只有在Student表上添加@ManyToOne注解，如果执行findAll()查询所有的学生，那么findAll()会出执行一条sql，如果采用懒加载的方式，后期在获取这些学生对象的时候回去执行子查询，这些学生对象的teachet\_id有几种类型，那么就会再去执行几条sql语句，所以所谓的N+1就是看多表findAll()后的对象中一的一方有N种，那就执行N次，如果多表只是findById()，那么就只会执行1+1次。

数据库：

|  |  |
| --- | --- |
| Student表(一) | Teacher表(多) |
| C:\Users\lenovo\AppData\Roaming\feiq\RichOle\455865127.bmp | C:\Users\lenovo\AppData\Roaming\feiq\RichOle\1484299680.bmp |
| C:\Users\lenovo\AppData\Roaming\feiq\RichOle\349037719.bmp | C:\Users\lenovo\AppData\Roaming\feiq\RichOle\3414395382.bmp |

测试：





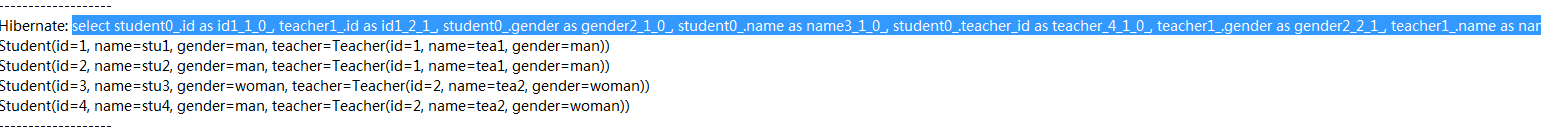
由于学生表中只有两个老师，所以一共执行3条sql，且Hibernate底层做了缓存，使得学生2和4不去执行子查询。

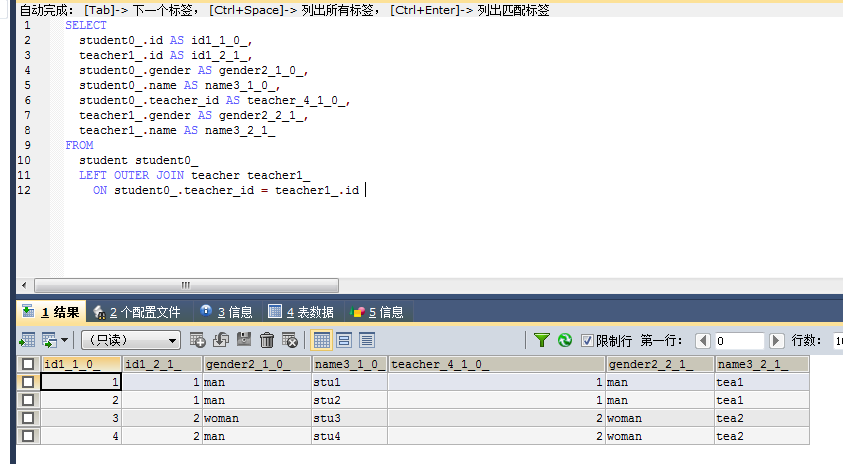
**解决方案：**

[Jpa2.1推出@EntityGraph、@NameEntityGraph用来提高查询效率，很好解决了N+1](mailto:Jpa2.1推出@EntityGraph、@NameEntityGraph用来提高查询效率，很好解决了N+1)条sql问题，两者需要配合使用，详情请看Spring Data Jpa。

|  |  |
| --- | --- |
| C:\Users\lenovo\AppData\Roaming\feiq\RichOle\2300618936.bmp | C:\Users\lenovo\AppData\Roaming\feiq\RichOle\3243786598.bmp |

执行上述一样的查询：





通过sql分析，这种方式只执行了一条sql，是通过做left join得到数据。

## @ManyToMany

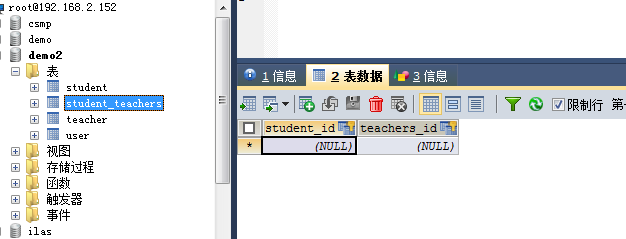
该注解表示多对多，和@ManyToOne、@OneToMany、@OneToOne一样都有单向、双向之分，单向与双向和注解没有关系，只看实体类之间是否互相引用。

下面的例子中Student表与Teacher表为多对多关系。

### 单向

|  |  |
| --- | --- |
| C:\Users\lenovo\AppData\Roaming\feiq\RichOle\315290936.bmp | C:\Users\lenovo\AppData\Roaming\feiq\RichOle\1911691021.bmp |

数据库表结构：



注意：采用@ManyToMany这样的关联注解默认都会添加外键约束与外键索引，如果不想添加这些，可以采取相应的配置，下文详解。

### 双向

|  |  |
| --- | --- |
| C:\Users\lenovo\AppData\Roaming\feiq\RichOle\315290936.bmp | C:\Users\lenovo\AppData\Roaming\feiq\RichOle\2200322716.bmp |

注意：添加mappedBy就是意味着可以通过Teacher表也可以关联到Student表，注意mappedBy后的value是主表(student)对应的字段(不是数据库字段)，并且标记了mappedBy的一方就是别维护端。

## @ManyToOne、@OneToMany

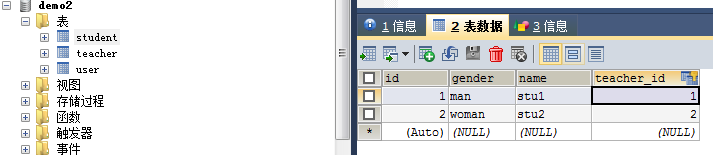
在这一组关系中，mappedBy属性只有@OneToMany才有，也就是说不管是单向还是双向，关系的维护端都是在多的那一侧，单双向不是来指定由谁来维护表间关系，只是看是否是可以从这张表查询到另外一张表，只看代码业务需求，至于维护方都是多的那一侧，外键都是在多的那一侧添加，即便是只有在一的一侧添加@OneToMany，只能代表只可以从一的一侧访问到多的一侧，但是外键还是加在多的那一边。

下面的例子中Student表与Teacher表为多对一关系。

### 单向

|  |  |
| --- | --- |
| C:\Users\lenovo\AppData\Roaming\feiq\RichOle\4083512145.bmp | C:\Users\lenovo\AppData\Roaming\feiq\RichOle\1911691021.bmp |

数据库表结构：



注意：由于主表Student表标记了级联操作，因此对于删除表Student中的记录，那么也会级联删除Teacher表中对应的记录。

### 双向

|  |  |
| --- | --- |
| C:\Users\lenovo\AppData\Roaming\feiq\RichOle\4083512145.bmp | C:\Users\lenovo\AppData\Roaming\feiq\RichOle\4167811398.bmp |

数据库表结构初始化：

|  |  |
| --- | --- |
| Student表(一) | Teacher表(多) |
| C:\Users\lenovo\AppData\Roaming\feiq\RichOle\434859587.bmp |  |

由于现在的表间关系是双向的，都添加了级联操作，那么对任意一张表的数据做改动均可以影响到另外一张表。

1. 删除teacher表中id=1的记录，那么对应还会删除student表中前两条数据。
2. 删除student表中id=3的记录，那么会级联删除teacher表中第2条记录(id=2)。

**不建议在多的一方做级联操作**，因为多的一方因为多条记录关联着一的方，比如A表是多表，B 是一表，由于A中的a、b、c记录都关联着B表中的d记录，那么无法删除a记录，因为删除a记录势必会去删除d记录，而d记录因为被b、c关联，所以无法删除就报错。

## 问题疑点

### 代码层面删除外键关联的数据

由于在添加@OneToMany类似关联注解的时候，Jpa会默认给我们加上外键约束，那么数据层面就无法删除被关联的表，因为它表里的数据被关联表在引用，如下图说明：

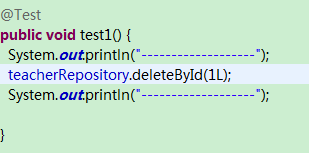
|  |  |
| --- | --- |
| C:\Users\lenovo\AppData\Roaming\feiq\RichOle\4237657540.bmp | C:\Users\lenovo\AppData\Roaming\feiq\RichOle\2325196384.bmp |

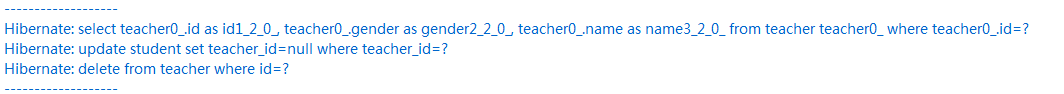
|  |  |
| --- | --- |
| 学生表 | 老师表 |
| C:\Users\lenovo\AppData\Roaming\feiq\RichOle\3009976901.bmp | C:\Users\lenovo\AppData\Roaming\feiq\RichOle\3544302017.bmp |

如果此时在数据库删除Teacher表中的数据会报错，因为此时Student表引用了该数据。



但是如果在代码层面删除，那么就可以，通过打印后的sql语句发现，Hibernate底层帮我们做了这样的改动。





Hibernate做了三步：1.取出id=1的老师数据；2.把学生表中的teacher\_id=1的数据改成**teacher\_id=null**；3.删除id=1的老师数据。

|  |  |
| --- | --- |
| 学生表 | 老师表 |
| C:\Users\lenovo\AppData\Roaming\feiq\RichOle\2567675584.bmp | C:\Users\lenovo\AppData\Roaming\feiq\RichOle\778048398.bmp |

### @OneToMany查询StackOverflowError问题

在进行一对多问题配置后，在测试方法中使用获取一方数据时，会出现堆栈溢出问题。

原因分析：

1. 为了方便看信息，在两类中分别重写了toString方法，导致查询加载时两类在互相调用对方的toString，形成递归，造成内存溢出。
2. 为了编写方便，代码更加优雅，使用了lombok插件中@Data以及@ToString来标注类，让lombok来代替生成getter/setter 以及toString，但是lombok在生成时会出现循环比较两类中的hashcode，导致内存溢出。

解决办法：不适用lombok

# 自定义注解

## 作用

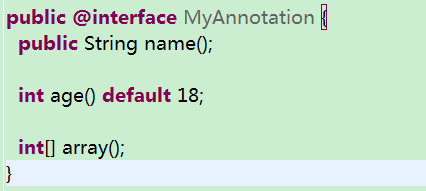
注解就是为容器提供原数据，例如@Controller注解就是标记了该Bean需要交给Spring容器进行管理。注解其实就是一种标记，可以在代码的关键节点(类、方法、变量、参数、包)上打上这些标记，然后程序在编译时或运行时可以检测到这些标记从而执行一特殊操作，因此可以得出自定义注解使用的基本流程：

* 第一步，定义注解——相当于定义标记
* 第二步，配置注解——把标记打在需要用到的程序代码中
* 第三步，解析注解——在编译器或运行时检测到标记，并进行特殊处理

## 格式

注解在Java中，与类、接口、枚举类型类似，因此声明语法基本一致，只是在关键字不同，自定义注解的关键字是@interface，在底层实现上，所有定义的注解都会自动继承Java.lang.annotation.Annotation接口。

根据我们自定义类的经验，在类的实现部分无非就是书写构造、属性或方法，但是在自定义注解中，其实现部分只能定义一个东西：注解类型元素(annotation type element)



定义注解类型元素需要注意如下几点：

* 访问修饰符必须是public，不写默认为public
* 该元素的类型只能是基本数据类型、String、Class、枚举类型、注解类型(体现了注解的嵌套效果)以及上述类型的数据。
* 该元素的名称一般定义为名词，如果注解中只有一个元素，请把名字命名为value(后面使用会带来便利操作)。
* ()不是定义方法参数的地方，也不能在括号中定义任何参数，这仅仅是一个特殊的语法。VVBV
* default代表默认值，值必须和第二点定义的类型一样。
* 如果没有默认值，代表后续使用注解时必须给该类型元素赋值。

## 常用的元注解

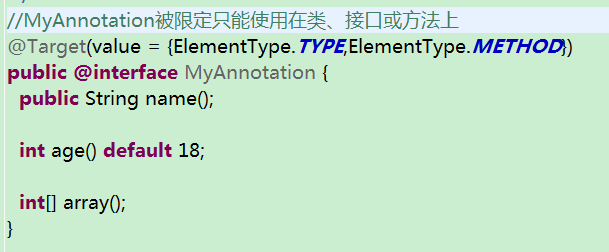
一个最基本的注解定义就只包含了上面两部分内容：1.注解的名字；2.注解包含的类型元素。但是我们在使用JDK自带注解的时候发现，有些注解只能写在方法上面(比如@Override);有些却可以写在类的上面(比如@Deprecated),当然除此之外还有很多细节性的定义，那么怎样去表示这些定义，就是采用元注解。

元注解：专门修饰注解的注解

* @Target

专门用来限定某个自定义注解能够被应用到哪些Java元素上

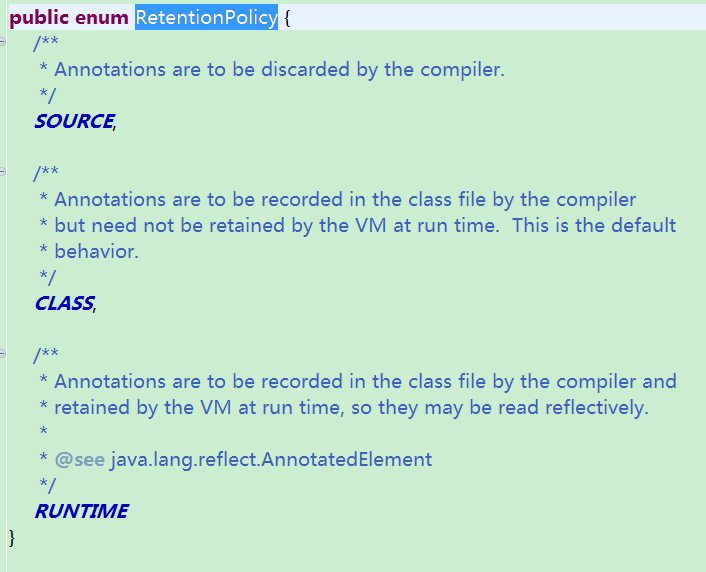




* @Retention

翻译为持久力、保持力，即用来修饰自定义注解的生命力。

注解的生命周期有三个阶段：1、Java源文件阶段；2.编译到class文件阶段；3.运行期阶段



详解如下：

1. 如果一个注解被定义为RetentionPolicy.SOURCE,则它将被限定在Java源文件中，那么这个注解既不会参与编译也不会在运行期起任何效果，这个注解就和一个注释是一样的效果，只能被阅读Java文件的人看到。
2. 如果一个注解被定义为RetentionPolicy.CLASS,则它将被编译到Class文件中，那么编译器可以在编译时根据注解做一些处理动作，但是运行时JVM(Java虚拟机)会忽略它，我们在运行期也不能读取到。
3. 如果一个注解被定义为RetentionPolicy.RUNTIME，那么这个注解可以在运行期的加载阶段被加载到class对象中，那么在程序运行阶段，我们可以通过反射得到这个注解，并通过判断是否有这个注解或这个注解中属性的值，从而执行不同的程序代码段，我们实际开发中的自定义注解几乎都是使用的**RetentionPolicy.RUNTIME**。

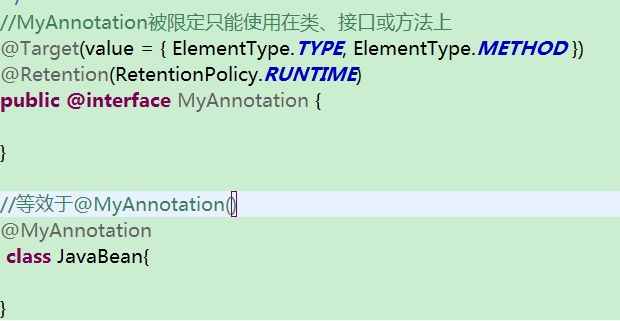
* @Documented

@Documented注解，是被用来指定自定义注解是否能跟随着被定义的Java文件生成到JavaDoc文档中。

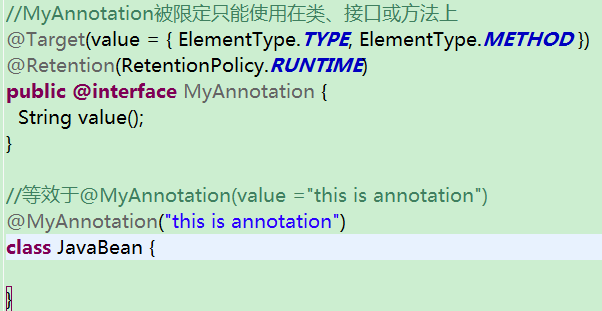
* @Inherited
* @Inherited注解是指定某个自定义注解如果写在了父类的声明部分，那么子类的声明部分也能自动拥有这注解，@Inherited注解只对那些@Target被定义为ElementType.TYPE的自定义注解起作用。

## 特殊语法

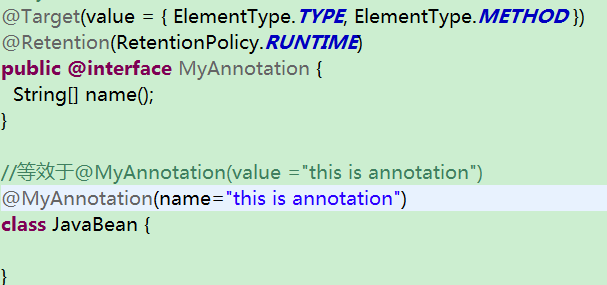
1. 如果注解本身没有注解类型元素，那么在使用注解的时候可以省略()，直接写为：@注解名，它和标准语法@注解名()等效。



1. 如果注解本身只有一个注解类型元素，并且命名为**value**，那么在使用注解的时候可以直接使用：@注解名(注解值)，其等效于@注解名(value=注解值)。



1. 如果注解中的某个注解类型元素是一个数组类型，在使用时又出现只需要填入一个值的情况：@注解名(类型名=类型值)，它和标准写法: @注解名(类型名={类型值})等效。

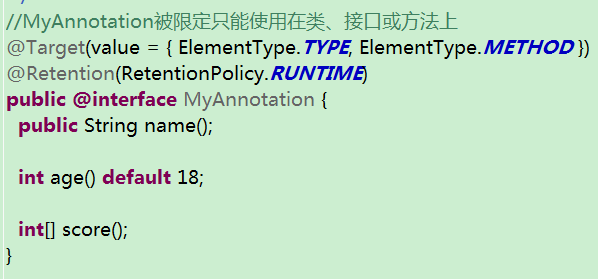


1. 如果一个注解的@Target是定义为Element.PACKAGE，那么这个注解时配置在package-info.java中的，而不能直接在某个类的package代码上面配置。

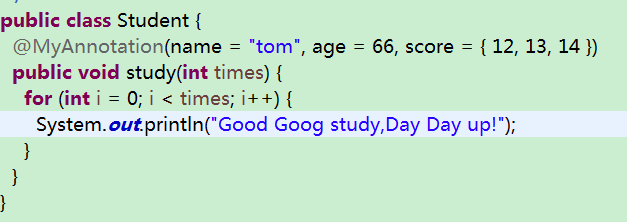
## 反射实例

下面通过反射实例来说明自定义注解：

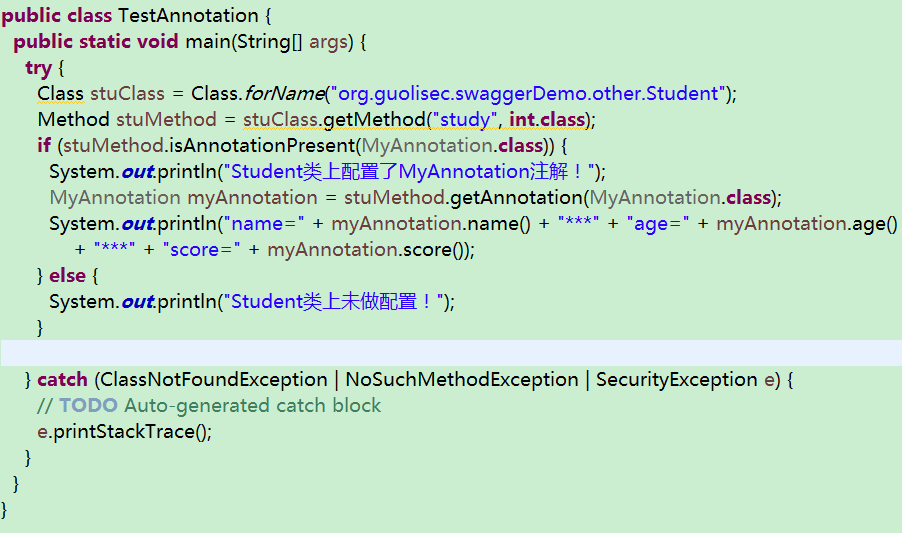
1. 定义自定义注解



1. 配置注解



1. 解析注解



输出=========C:\Users\lenovo\AppData\Roaming\feiq\RichOle\1587975510.bmp

---------------------------------------------------------------------------------------------------------------------------------

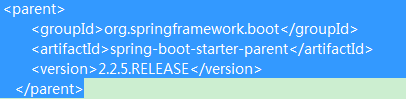
**备注**：

1. 如果我们要获取的注解配置在方法上，那么我们要从Method对象上获取；如果是配置在属性上，我们就需要从属性对应的Field对象上去获取，如果配置在类上，需要从Class对象上获取，总之在谁身上，就从谁身上获取。
2. isAnnotationPresent(Class<? extends Annotation> annotationClass)方法是专门判断该元素上是否配置有某个指定的注解。
3. getAnnotation(Class<? extends Annotation> annotationClass)方法是获取该元素上指定的注解，之后再调用该注解的注解类型元素方法就可以获得配置的数据了。
4. 反射对象上还有一个方法getAnnotations()，该方法可以获得该对象身上配置的所有的注解，它会返给我们一个注解数组，需要注意的是该数组的类型是Annotaion类型，这个Annotation是一个来自于java.lang.annotation包的接口。

---------------------------------------------------------------------------------------------------------------------------------

# Pom.xml文件配置

## Spring Boot的父级依赖

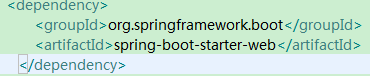


这是Spring Boot的父级依赖，这样当前项目就是Spring Boot项目了，spring-boot-starter-parent是一个特殊的starter，它用来提供相关的Maven默认依赖，使用它之后，常用的包依赖就可以省去version标签，因此可以用来作为版本统一。

## 基本依赖介绍

### Spring-boot-starter-web

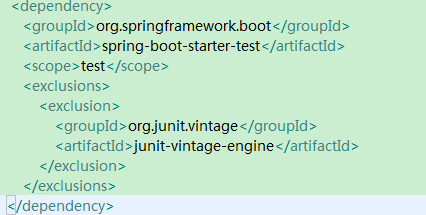
由于在父级依赖引入统一版本号，因此该依赖可以忽略版本号。



该依赖帮我们引入web模块开发需要的相关jar包。

### Spring-boot-starter-test

由于在父级依赖引入统一版本号，因此该依赖可以忽略版本号。



Springboot程序测试依赖，这里指定了junit的类型，如果需要在测试类中要启动spring容器，[那么需要引入org.junit.jupiter.api.Test下的@Test](mailto:那么需要引入org.junit.jupiter.api.Test下的@Test)，当然pom文件中也可以不指定junit,那么需要自己在测试类中选择正确的junit类型。

### Spring-boot-starter-data-jpa

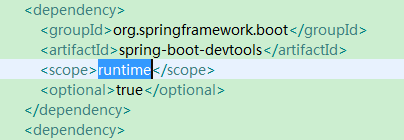
由于在父级依赖引入统一版本号，因此该依赖可以忽略版本号。

### spring-boot-starter-security

由于在父级依赖引入统一版本号，因此该依赖可以忽略版本号。

### spring-boot-devtools

由于在父级依赖引入统一版本号，因此该依赖可以忽略版本号。



### dependencies和dependencyManagement

dependencyManagement只是声明依赖，因此子项目中需要显示的声明需要用的依赖，如果不在子项目中声明依赖，是不会从父项目中继承下来，只有在子项目中的dependencies中写了该依赖，并且没有指定版本，才会从父项目中继承该项，并且version和scope都读取父项目的pom，如果子项目中指定了版本号，那么才会使用子项目中指定的jar。

## 基本插件介绍

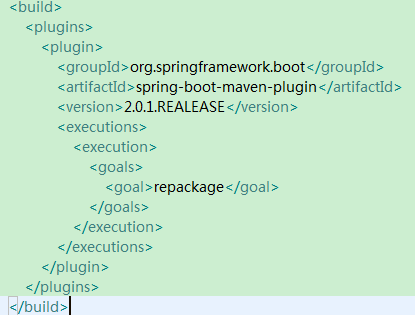
### spring-boot-maven-plugin

Spring Boot的Maven插件(Spring Boot Maven plugin)能够以Maven的方式为应用提供Spring Boot的支持，该插件能够将Spring Boot应用打包为可执行的jar或war文件。

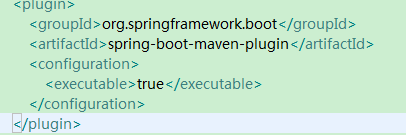
该插件有5个Goal:

* repackage(默认goal):maven packag之后，再次打包可执行的jar/war，同时保留mvn package生成的jar/war为.origin
* run:运行Spring Boot应用
* start:在maven integration-test阶段，进行Spring-Boot应用生命周期的管理。
* stop:在maven integration-test阶段，进行Spring-Boot应用生命周期的管理。
* build-info:生成Actuator使用的构建信息文件build-info.properties

该插件最主要的goal是repackag。

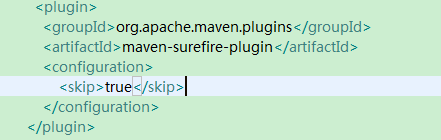


一般由于repackage是默认的，所以可以不写，并且我们的应用一般都是部署在Linux服务器上，所以我们需要指定一个额外的配置。



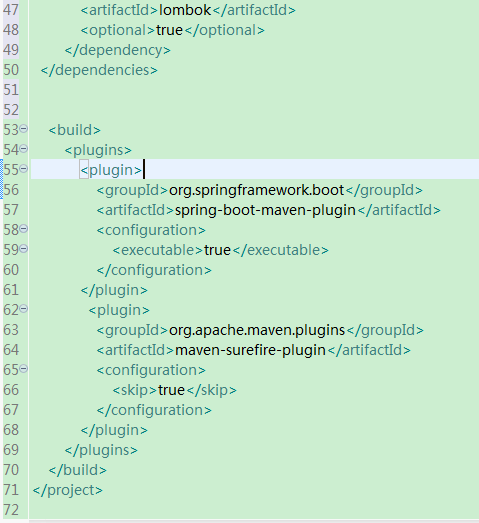
通过红框内的配置，生成的jar包会包含一个shell脚本，可以方便的将该应用设置为Unix/Linux的系统服务(init.d service)。

### 跳过测试



## 通用项目pom配置





# 定时作业@Scheduled

SpringBoot @Scheduled支持同步/异步同一任务及多任务并发执行。

1. 同一任务的同步执行(下次任务执行将在本次任务执行完毕后的下一次配置时间开始)。
2. 同一任务的异步执行(下次任务将在下一个配置时间开始，不等待当前任务执行完成)。

## 单/多任务并发执行

1. 首先，要想使用@Scheduled注解，首先要在启动类上添加注解@EnableScheduling。
2. 同一任务的同步执行(下次任务执行将在本次任务执行完毕后的下一次配置时间开始)，SpringBoot默认就是定时任务同步执行的，只要将@Scheduled添加到需要配置的任务方法上，下次任务执行将在本次任务执行完毕后开始。
3. 同一任务的异步执行(下次任务将在下一个配置时间开始，不等待当前任务执行完毕)，需要在方法上同时添加@Scheduled和@Async注解。
4. 多任务的并发执行：SpringBoot默认开启的线程数是1 ，如果针对多线程的并发问题，需要单独配置线程池。

# Swagger

# 单线程、多线程、异步、并行、并发

* 进程

正在运行的应用程序，包括运行中的程序和程序所使用的的内存和系统资源，包含堆与方法区。

* 线程

进程的执行单元，每一个线程都有自己的专有寄存器(栈指针、程序计数器等)，但代码是共享的，即不同的线程可以执行同样的函数。

程序计数器：是一块内存区域，用来记录线程当前要执行的指令地址。

栈指针：用来存储该线程的局部变量，这些局部变量是该线程私有的，除此之外还用来存放线程的调用栈帧。

堆：是一个进程中最大的一块内存，堆是被进程中的所有线程所共享。

由于线程是依赖于进程存在，而进程是由操作系统创建的，并且Java语言不能调用操作系统的功能，所以为了方便对多线程程序的使用，Java提供了线程的API对应的类---Thread。

## Java多线程对成员变量与局部变量的影响

关于成员变量与局部变量：

如果一个变量是成员变量，那么多线程对同一个对象的成员变量进行操作时，他们对该成员变量是彼此影响的(也就是说一个线程对成员变量的改变会影响到另一个线程)。

如果一个变量是局部变量，那么每一个线程都会有该局部变量的拷贝，一个线程对该局部变量的改变不会影响到其他的线程。

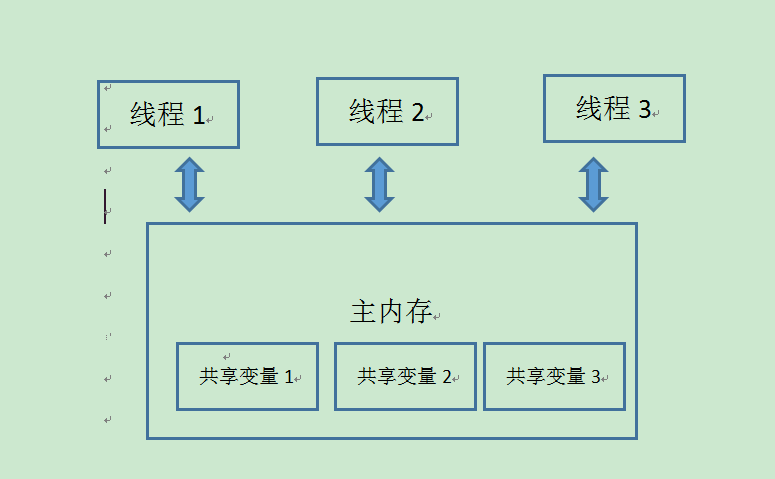
## 并行与并发

如果某个系统支持两个或者多个动作(Action)同时存在，那么这个系统就是一个并发系统，如果某个系统支持两个或者多个动作同时执行，那么这个系统就是并行系统，并发系统与并行系统这两个定义之间的关键差异在于“存在这个词”。

在并发程序中可以同时拥有两个或者多个线程，这意味着，如果程序在单核处理器上运行，那么这两个程序将交替地换入或者换出内存。这些线程是同时“存在”的——每个线程都处于执行过程中的某个状态。如果程序能够并行执行，那么就一定是运行在多核处理器上，此时程序的每个线程都将分配到一个独立的处理器核上，因此可以同时执行。

结论：“并行”概念其实可以定义为“并发”概念的一个子集，也就是说，你可以编写一个拥有多个线程或者进程的并发程序，但是如果没有多核处理器来执行这个程序，那么就不能以并行方式来运行代码，因此，凡是在求解单个问题时涉及多个执行流程的编程模式或者执行行为，都属于并发编程的范畴。

## 并发中的线程安全



多个线程同时操作共享变量时，会出现线程1更新共享变量1的值，但是其他线程获取到的是共享变量没有被更新之前的值，就会导致数据不准确问题。

要跨越线程维护正确的可见性，只要在几个线程之间共享非final变量，就必须使用synchronized(或volatile)以确保一个线程可以看见另一个线程做的更改。

为了多线程并发访问共享资源的安全性，也为了互斥访问，同步时必须的，这是因为Java语言规范的内存模型，它规定了：一个线程所做的变化何时以及如何变成对其他线程可见，因此在多线程并发访问共享资源会采用同步安全机制，采用锁机制，锁提供了两种主要的特性：互斥(mutual exclusion)和可见性(visibility)。

互斥：即一次只允许一个线程持有某个特定的锁，因此可使用该特性实现对共享数据的协调访问协议，这样，一次就只有一个线程能够使用共享数据。

可见性：它必须确保释放锁之前对共享数据的修改对于随后获得该锁的另外一个线程是可见的——如果没有同步机制提供的这种可见性保证，线程看到的共享变量可能是修改前的值或者不一致的值。

## 同步与异步

首先在概念上，同步与异步是在执行顺序上，同步是按顺序执行，异步是开启另外一个线程执行，在Java代码层面上，添加@Async注解，相当于是开启了另外一个线程，对于同步而言，该方法只有执行完毕后才会返回一个HTTP的code码，可能是成功也可能是失败，但是对于异步代码块而言，不管最终结果是成功还是失败，执行最开始就会返回一个200的code码，然后最终代码如果报错，web界面不会显示错误的code码。那么我们如果需要让前端获取最终的结果，可以使用websocket与前端界面保持长连接，一旦出错，就返回给前端信息。

然而另外一个概念——多线程的同步与异步。此时的多线程指的是web访问后台，后台会生成一个单独的线程，多线程并发时，就是web同时访问后端同一个资源，必然导致此资源的数据不安全，A线程修改了B线程处理的数据，而B线程又修改了A线程处理的数据，显然这是由于全局资源造成的，为了解决该问题，优先使用局部变量，退而求其次考虑同步代码块，也就是用synchronized,处于这样的安全考虑就必须牺牲系统处理性能，加在多线程并发时资源争夺最激烈的地方，这就是所谓的同步机制，也就是锁机制，对应于数据库的是乐观锁与悲观锁。

## Java 4种同步机制

目的：都是为了解决多线程中的对同一变量的访问冲突。

* ThreadLocal

ThreadLocal保证不同线程拥有不同的实例，相同线程一定拥有相同的实例，即为每一个使用该变量的线程提供一个该变量值的副本，每一个线程都可以独立改变自己的副本，而不是与其他线程的副本冲突。

优势：提供了线程安全的对象共享。

与其他同步机制的区别：同步机制是为了同步多个线程对相同资源的并发访问，是为了多个线程之间进行通信；而ThreadLocal是隔离多个线程的数据共享，从根本上就不是多个线程之间的资源共享，这样当然不需要多个线程进行同步了。

* Volatile

volatile修饰的成员变量在每次被线程访问时，都强迫从共享内存中重读该成员变量的值，并且,当成员变量发生变化时，强迫线程将变化值写会共享内存。

优势：这样在任何时刻，两个不同的线程总是可以看到某个成员变量的同一个值。

缘由：Java语言规范中指出，为了获得最佳速度，允许线程保存共享成员变量的私有拷贝，并且只有当线程进入或者离开同步代码块的时候才与共享成员变量的原始值对比，这样当多个线程同时与某个对象交互时，就必须要注意到要让线程及时得到共享成员变化的值，而关键字volatile不能保存线程的私有拷贝，而是直接与共享成员变量交互。

使用技巧：在两个或者更多的线程访问的成员变量上使用volatile,当要访问的变量已经在所synchronized代码块中，或者为常量，那就不需要使用。

volatile变量具有synchronized的可见性，但是不具备原子特性(原子特性：互斥+可见性)，因此建议使用锁机制。

* sleep()、wait()

sleep是线程类Thread的方法，导致此线程暂停执行指定时间，把执行机会给其他线程，但是监控状态依然保存，到时会自动恢复，调用sleep() 不会释放锁对象。

wait()是object类的方法，调用wait()方法会导致本线程放弃对象锁，进入等待状态，只有针对此对象的其他线程使用notify方法(或者notifyAll)后本线程才进入唤醒状态(就绪)。

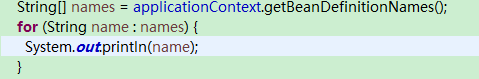
# 工具

## Spring获取容器内Bean对象

1. 注入ApplicationContext对象。

C:\Users\lenovo\AppData\Roaming\feiq\RichOle\3600328211.bmp

1. 查看bean对象。

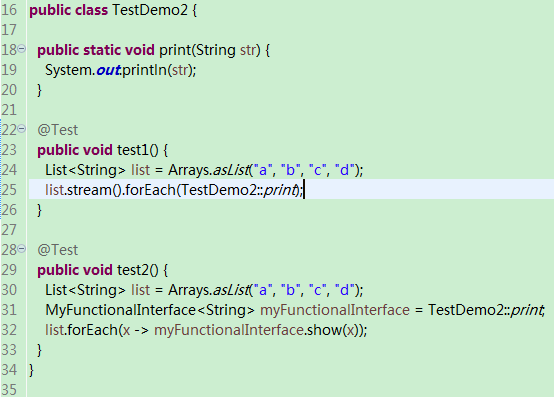


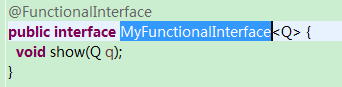
# Java8 新特性

## ：：关键字

* 用途一

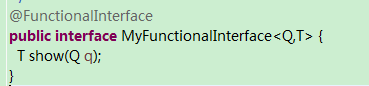
Java8中我们可以通过：：关键字来访问类的构造方法，对象方法以及静态方法。





注意：

1. 函数式接口有且只有一个抽象方法，可以有静态、默认方法，如果满足这些特性，jdk就识别该接口为函数式接口，采用@FunctionalInterface来识别，也可以不加，但是为了代码的易懂性，要求加上注解。
2. 函数式接口中的抽象方法结构与实体调用方法的结构保持一致，也就是说如果实体方法有返回值，那么需要

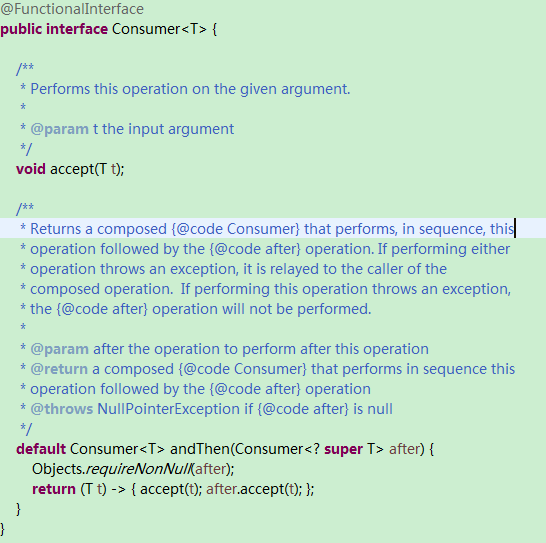


* 用途二

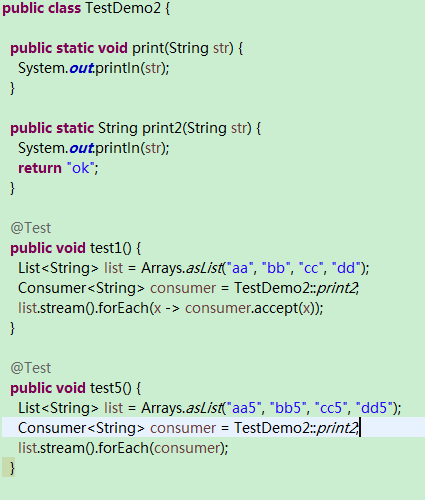
Jdk8使用：：，就是把方法当做参数传到stream内部，使stream的每一个元素都传到该方法里面执行一下。

### 疑点

Java提供了消费的函数式接口Consumer：



里面定义的抽象方法是accept,为void形式。



Test5中的print2是有返回值的，但是可以被consumer接受，到底是accept方法调用了TestDemo2.print2，还是TestDemo2.print2在实际调用中代替了accept?

## Lambda 包含局部变量

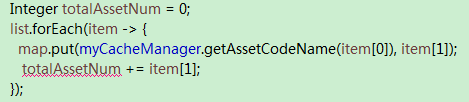
场景：通过stream处理集合时，可能需要操作一个中间变量或者局部变量，那么会提示：

Local variable xxx defined in an enclosing scope must be final or effectively final.

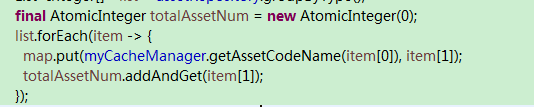
这是因为你使用局部变量在初始化后，又对这个变量进行了赋值，赋值后会认为这个变量不是final了，所以报错。

Lambda表达式可以访问给它传递的变量，访问自己内部定义的变量，同时也能访问它外部的变量，不过lambda表达式访问外部变量有一个非常重要的限制：变量不可以变(只是引用不可变，不是真正的不可变)

解决方法：



替换成：



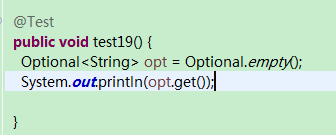
## Optional (解决空指针异常)

Optional类主要解决的问题是空指针问题(NullPointException)

本质上，这是一个包含有可选值的包装类，这意味着Optional类既可以含有对象也可以为空。

Optional是Java实现函数式编程重要一环，并且在范式中也有应用。

* empty(包装类不包含对象)



这样会导致NoSuchElementException

* of()
* ofNullable()

可以使用of()、ofNullable()方法创建包含值的Optional。两个方法的不同之处的是如果你把null值作为参数传递进去，of()会抛空指针异常。所以明确对象不为null的时候可以使用of()，如果对象既可以是null也可以是非null，你就应该使用ofNullable方法。

* get()

从Optional实例中取回实际对象。

* isPresent()

判断Optional对象是否有值。

* ifPresent()

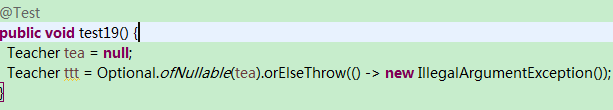
判断Optional对象是否有值，该方法除了执行判断是否为null，还接受一个Consumer函数，如果对象不是空的，就执行传入的Lambda表达式。

* orElse()

如果有值就返回该值，否则返回传递给它的参数值。

* orElseGet()
* 如果有值就返回该值，否则返回传递给它的参数值。与orElse()区别是：orElse()不管是否存在对象，都会去执行默认方法，orElseGet()只有在包装的对象为null的时候才会去执行后面的Lambda表达式。一般建议采用orElseGet()。
* orElseThrow()

它会在对象为空的时候抛出异常，而不是返回备选的值。



这个方法可以自定义决定抛出什么样的异常。

--------------------------------------------------------------------------------------------------------------------------------

转换值

* map()
* flatMap()

# 数据库

## 数据库的ACID

数据库管理系统(DBMS)中事务(Transaction)的四个特性：原子性(Atomicity)、一致性(Consistency)、隔离性(Isolation)、持久性(Durability)。

---------------------------------------------------------------------------------------------------------------------------------

隔离性:

多个事务并发访问时，事务之间是隔离的，一个事务不应该影响其他事务的运行效果。

在并发环境中，当不同的事务同时操纵相同的数据时，每一个事务都有各自的完整的数据空间。由并发事务所做的修改必须与任何其他并发事务所做的修改隔离。事务查看数据更新时，数据所处的状态要么是另一个事务修改它之前的状态，要么是另一事务修改它之后的状态，事务不会查看中间状态的数据。

事务最复杂问题都是由事务隔离性引起的。完全的隔离性是不现实的，完全的隔离性要求数据库同一时间只执行一条事务，这样会严重影响性能。

---------------------------------------------------------------------------------------------------------------------------------

## 隔离级别

隔离级别可以在不同程度上减少丢失更新，Spring或者Java只是按照SQL的规范定义的，隔离级别定义为脏读(dirty read)、不可重复度(readCommit)、幻读(repeatable read)以及序列化(Serializable)。

1. 脏读

脏读是最低的隔离等级，其含义是允许一个事务去读取另一个事务中未提交的数据。

1. 不可重复读

为了克服脏读，SQL采用第二个隔离级别——不可重复度，就是说一个事务只能读取到另一个事务已经提交的数据。

弊端:如果A事务与B事务同时操作相同数据，由于采用不可重复度的隔离机制，任何事务在未提交之前不会影响其他事务，A事务与B事务同时操作1000元。

步骤：

1. A事务查询余额，显示1000，取出100，显示900，同一时间，B事务已取出900，显示剩余100，由于AB事务未提交，所以不会读取相互的数据，数据库不可见。
2. B事务提交事务，那么此时A就可以读取B事务，第二次读取发现余额为100，上一次显示余额为900,因此对A而言，读取两次数据会变化。
3. 幻读

幻读也是两次读取结果不一致，但是与不可重复度不一样的是，不可重复读是对同一部分数据显示不同的读取效果，而幻读是当一个事务按条件查询时，行锁仅仅是加在了符合条件的记录行上，此时其他的写事务虽然不能操作加有行锁的记录行，但是完全可以操作其他行数据，甚至是新增一条符合条件的记录，所以如果在读取事务的过程中，写事务新增了一条符合记录的记录，那么读事务再次查询时，就会查询到上次没有的新记录，那么就出现了幻读。

1. 序列化

为了解决幻读的问题，仅仅在部分数据加行锁是不够的，需要对整个表加锁，也就是说事务挨个执行，不能并发，当一个读事务在读取数据时，无论读取的是什么数据，别的事务必须等待，整个表加锁，等待该事务执行完毕，再轮流下一个，可以解决所有问题，但是隔离级别太高，数据库并发效率低下，因此不建议设置数据库的隔离级别为最高等级的序列化。

## 事务

事务管理是应用系统开发中必不可少的一部分，Spring为事务管理提供了丰富的功能支持。Spring事务管理分为编码式和声明式两种。编程式事务是通过编码方式实现事务；声明式事务是基于AOP,其本质是对方法前后进行拦截，然后在目标方法开始之前创建或者加入一个事务，在执行完目标方法之后根据执行情况提交或者回滚事务。将具体的业务逻辑与事务处理逻辑解耦。声明式事务管理使业务代码逻辑不受污染，因此在实际使用中声明式事务用的比较多。

声明式事务有两种：一种是在配置文件(XML)中做相关的事务规则说明，另外一种是基于@Transactional注解的方式。

### 自动提交(AutoCommit)与连接关闭时的是否自动提交

默认情况下，数据库处于自动提交模式，每一条sql语句处于一个单独的事务中，在这条语句执行完毕时，如果执行成功则隐式的提交事务，如果执行失败则隐式地回滚事务。因此我们单独的的一句sql程序不需要加@Transactional注解，因为自身就包含了事务处理机制。

对于正常的事务管理，是一组相关的操作处于一个事务之中，因此必须关闭数据库的自动提交模式，Spring的事务处理，不管是编码式还是声明式事务，都会在底层将自动提交的设置为false。

有些数据连接池提供了关闭事务自动提交的设置，最好在设置连接池时就将其关闭，但是C3P0没有提供这一特性，只能依靠spring来设置。因为JDBC规范规定，当连接对象建立时应该处于自动提交模式，这是跨DBMS(Database Management System，数据库管理系统)的缺省值，如果需要，就必须显示地关闭自动提交。C3P0遵守这一规范，让客户来显示的设需要的提交模式。

### Checked异常和unchecked异常

Spring使用声明式事务，默认情况下，如果被注解的数据库操作方法中发生了unchecked异常，所有的数据库操作将rollback，如果发生的异常是checked异常，默认情况下数据库操作还是会提交的。

* checked异常

check exception 继承自Exception，代码不能通过编译，需要用try…catch语句或者throws语句抛出去。

* unchecked异常

也称为RuntimeException，也是继承自Exception，代码可以通过编译，最常见的包含ArithmeticException(3/0)、ClassCastException、IndexOutOfBoundsException、IllegalArgumentException、NullPointerException等。

### @Transactional

1. 在需要事务管理的地方加@Transactional注解，@Transactional可以被应用到接口以及接口的方法、类定义以及类的public方法，注意，该注解只能对public方法生效，private、protected方法等添加该注解不会报错，但是也不会生效。
2. 在启动类上添加@EnableTransactionManagement注解，开启事务管理。
3. Service层的方法上如果涉及多条sql，那么添加@Transactional注解，在运行期异常时回滚，如果是针对非运行期异常，要么抛出throw new runtimeexception()，要么在try…catch的catch代码块里添加throw new runtimeexception()。

理解：@Transactional对应的数据库的事务的四大特性(ACID:原子性(Automicity)、一致性(consistency)、隔离性(Isolation)、持久性(Durability))。

## Mysql 与redis

1. mysql与redis的数据库类型

mysql是关系型数据库，主要用于存放持久化数据，将数据存在硬盘中，读取速度较慢。

reids是NOSQL，即非关系型数据库，也是缓存数据库，即将数据存储到缓存中，读取速度较快，能够大大提高运行效率，但是保存时间有限。

1. mysql的运行机制

mysql作为持久化存储的关系型数据库，相对薄弱的地方在于每一次请求访问数据库，都存在着I/O操作，如果反复频繁的访问数据库，第一：会在反复链接数据库上花费大量时间，从而导致运行速率过慢；第二：反复的访问数据库也会导致数据库的负载过高，那么此时缓存的概念就衍生出来了。

1. 缓存

缓存就是数据交换的缓冲区(cache)，当浏览器执行请求时，首先会对在缓存中进行查找，如果存在，就获取，否则就访问数据库。--------缓存的好处就是读取速度快。

1. redis数据库

redis数据库是一款缓存数据库，用于存储使用频繁的数据，这样减少访问数据库的次数，提高运行效率。

1. mysql与redis区别总结

* 类型上

从类型上来说，mysql是关系型数据库，redis是缓存数据库。

* 作用上

mysql用于持久化的存储数据到硬盘，功能强大，但是速度较慢。

redis用于存储使用较为频繁的数据到缓存中，读取速度快。

* 需求上

mysql和redis因为需求不用，可以配合使用。

# Spring 缓存

从spring3.1开始，spring引入了对cache的支持，其使用方法和原理都类似于spring对事务管理的支持。Spring cache 是作用在方法上或类上的，其核心思想是这样的：当我们在调用一个缓存方法时会把该方法参数和返回结果作为一个键值对存放在缓存中，等到下次利用同样的参数来调用该方法时将不再执行该方法，而是直接从缓存中获取结果进行返回，所以我们在使用spring cache 的时候我们要保证我们缓存的方法对于相同的方法参数要有相同的返回结果。

和spring对事务管理的支持一样，spring对cache的支持也有基于注解和基于XML配置两种方式，下文主要介绍基于注解的缓存机制。

## @Cacheable

Spring 为我们提供了几个注解来支持spring cache。其核心主要是@Cacheable和@CacheEvict。使用@Cacheable标记的方法在执行后spring cache将缓存返回其结果，而使用@CacheEvict标记的方法会在方法执行前或者执行后移除spring cache中的某些元素。

* @Cacheable

@Cacheable可以标记在一个方法上，也可以标记在一个类上。当标记在一个方法上是表示该方法是支持缓存的，当标记在一个类上时则表示该类的所有方法都是支持缓存的。对于一个支持缓存的方法，spring会在其调用后将其返回值缓存起来，以保证下次利用同样的参数来执行该方法时可以直接从缓存中获取结果，而不需要再次执行该方法。Spring在缓存方法的返回值是以键值对进行缓存的，值就是方法的返回结果，至于键的话，spring支持两种策略，默认策略和自定义策略

---------------------------------------------------------------------------------------------------------------------------------

注意：当一个支持缓存的方法在对象内部被调用时是不会触发缓存功能的。

---------------------------------------------------------------------------------------------------------------------------------

@Cacheable可以指定三个属性：value、key、condition

1. value属性是必须指定的，其表示当前方法的返回值是会被缓存到哪个cache上的，对于cache的名称，可以是一个cache也可以是多个cache，当需要指定多个cache时其是一个数组。

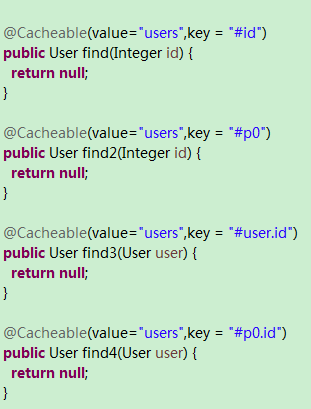
@Cacheable(“cache1”)

@Cacheable({“cache1”, “cache2”})

1. key属性是用来指定spring缓存方法的返回结果所对应的的key，该属性支持SpringEL表达式，当我们没有指定该属性时，spring将使用默认策略生成key.

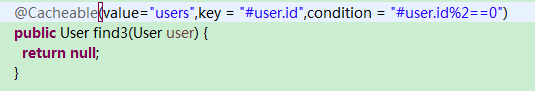
* 自定义策略

自定义策略是指我们可以通过spring的EL表达式来指定我们的key，这里的EL表达式可以使用方法参数及它们对应的属性。使用方法参数时我们可以直接使用“#参数名”或者“#p参数index”



1. Condition属性指定发生的条件

有的时候我们可能并不希望缓存一个方法所有的返回结果。通过condition属性可以实现这一功能，condition属性默认为空，表示会缓存所有的调用情形，也可以指定缓存的场景，如下例中：只有当user的id为偶数时才会进行缓存。



## @CachePut

在支持spring cache的环境下，对于使用@cacheable标记的方法，spring在每一次执行都会检查cache中是否存在相同key的缓存元素，如果存在就不再执行该方法，而是直接从缓存中获取结果进行返回，否则才会执行并将结果存入指定的缓存中。

@CachePut也可以声明一个方法支持缓存功能，与@cacheable不同的是使用@CachePut标注的方法在执行前不会去检查缓存中是否存在之前执行过的结果，而是每一次都会去执行该方法，并将执行结果以键值对的形式存入指定的缓存中。

## @CacheEvict

@CacheEvict是用来标注在需要清除缓存元素的方法或者类上的，当标记在一个类上时表示所有的方法都会触发缓存的清除操作，@CacheEvict可以指定的元素有value、key

、condition、allEntries和beforeInvocation。其中value、key、condition的语义与@cacheable对应的属性类似，即value表示清除操作时发生在哪些cache上(对应cache的名称)；key表示需要清除的是哪个key；condition表示清除操作发生的条件。

allEntries:它是boolean属性，表示是否需要清除缓存中的所有元素，默认为false，表示不需要，当指定了true时，spring cache将忽略指定的key，清除所有的缓存元素。

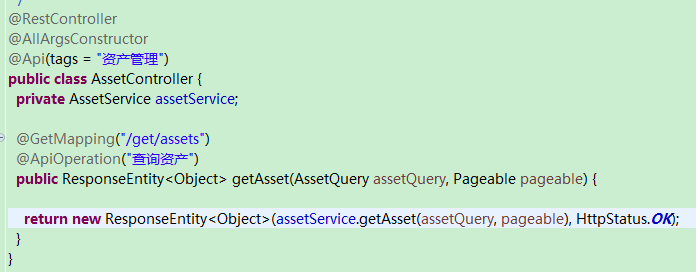
beforeInvocation:清除操作默认是在对应方法成功执行之后触发的，即方法因为抛出异常而未能成功返回时也不会触发清除操作，使用beforeInvocation可以改变触发清除操作的时间，当我们指定该属性的值是true时，spring会在调用该方法之前清除缓存的元素。

## Demo样例(添加Manager层)

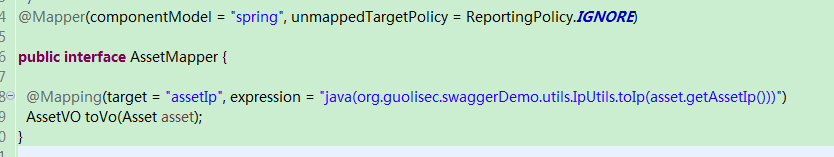
1. 实体类以及返回VO对象

|  |  |  |  |
| --- | --- | --- | --- |
| C:\Users\lenovo\AppData\Roaming\feiq\RichOle\3253488119.bmp   |  |  | | --- | --- | | C:\Users\lenovo\AppData\Roaming\feiq\RichOle\2902197101.bmp |  | | C:\Users\lenovo\AppData\Roaming\feiq\RichOle\2520738569.bmp |

1. Controller层



1. Mapper



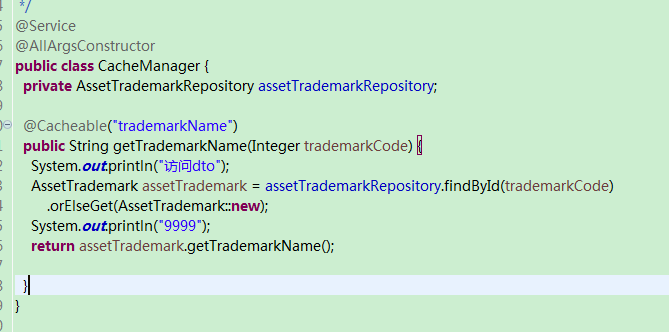
需求:

数据库存的IP是Long类型的，展示给起前端界面的是String。

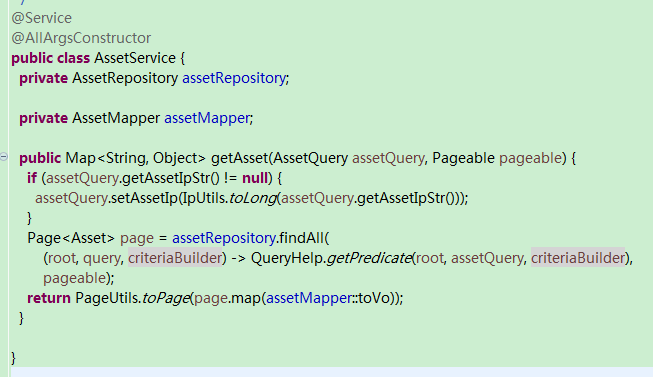
由于expression中的表达式一般定义为静态方法，那么由于展示给前端界面的品牌是名字，而数据库是存的是编号，所以我们需要建立缓存进制，减少访问数据库品牌的编码表，但是缓存方法不可以定义为静态，因为缓存方法中调用了仓储(静态不可以调用非静态)，因此我们需要在这个mapper的impl实现中编辑该功能。



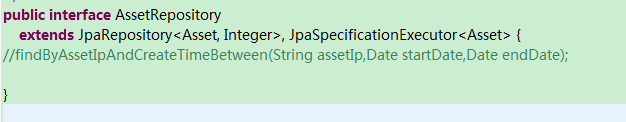
在Dao层与Service层之间添加Manager层(阿里规范)

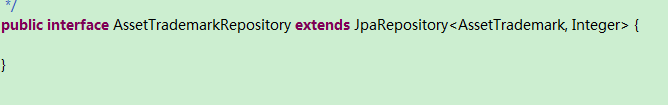


1. Service层



1. Dao层

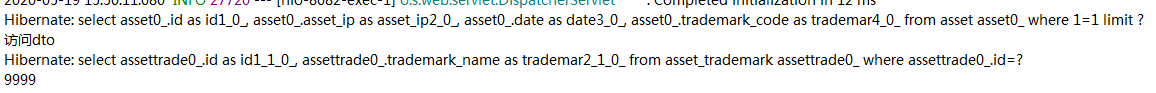




1. 查询结果

|  |  |
| --- | --- |
| asset | asset\_trademark |
| C:\Users\lenovo\AppData\Roaming\feiq\RichOle\1338136798.bmp | C:\Users\lenovo\AppData\Roaming\feiq\RichOle\1797327325.bmp |

第一次查询：



由于品牌都是1,所以第一次查询只查了一次，后面因为缓存不再查询。

第二次查询:

C:\Users\lenovo\AppData\Roaming\feiq\RichOle\1861333090.bmp

由于缓存机制，一次都不再查询。

# Spring 注解

## @postConstruct

Spring 中construct、@Autowired、@postConstruct加载顺序

Construct>@Autowired>@postConstruct

从依赖注入的字面意思就可以知道，要将对象P注入到对象a，那么首先就必须生成对象p与对象a，才能执行注入，所以一个类中有个成员变量被@Autowired注解，那么@Autowired注入是发生在A的构造函数之后的。

@postConstruct应用场景：如果想在生成对象的时候完成某些初始化操作，而这些操作又依赖于依赖注入，那么就无法在构造函数中实现，为此可以使用@postConstruct注解一个方法来完成初始化，@postConstruct注解的方法会在依赖注入后被自动调用。

## @DynamicInsert

@DynamicInsert注解设置为true时，insert一个对象的时候，会去动态判断实体对象的值，如果该值是null，就不会加入到sql语句。------------好处：这样做就可以给数据添加默认值，因为实体类的属性都是包装类，类似Integer这样的属性数据库默认值都是null，步骤：1.sql脚本建表的时候，给属性添加默认值。2，在实体类上添加@DynamicInsert注解，那么代码层面就不会给该包装属性字段赋值，在数据库层面赋值为0。

# ElasticSearch

## Elasticsearch、Logstash、Kibana

开源实时日志分析ELK，由elasticsearch、logstash、kibana三部分组成，ELK不是一款软件，而是一整套解决方案，也可以称为ELK协议栈。

其中三个部分分别是：

* Elasticsearch

开源分布式搜索引擎，它的特点是：分布式、零配置、自动发现、索引自动分片、索引副本机制、restful风格接口，多数据源，自动搜索负载等。

* Logstash

开源工具，支持对日志进行收集、分析，并存储到数据库。

* Kibana

Kabana可以为logstash和elasticsearch提供日志分析友好web界面。

Logstash通过管道进行操作，管道有两个必要的元素，输入和输出，还有一个可选的元素-过滤器(filter)，输入插件从数据源获取数据，过滤器过滤数据，输出到指定的数据库。

## Elasticsearch

---------------------------------------------------------------------------------------------------------------------------------

* 集群(cluster):由一个或多个节点组成，并通过集群名称与其他集群进行区分。
* 节点(node)：单个ElasticSearch实例，通常一个节点运行在一个隔离的容器或虚拟机中。
* 索引(index):在ES中，索引是一组文档的集合。
* 分片(shard):因为ES是个分布式的搜索引擎，所以索引通常都会分解成不同部分，而这些分布在不同节点的数据就是分片，ES自动管理和组织分片，并在必要的时候对分片数据进行再平衡分配，所以用户基本上不用担心分片的处理细节。
* 副本(replica)：ES默认为一个索引创建5个主分片，并分别为其创建一个副本分片，也就是说每个索引都由5个主分片成本，而每一个主分片都相应的有一个copy。对于分布式搜索引擎来说，分片以及副本的分配将是高可用及快速搜索响应的设计核心，主分片与副本都能处理查询请求，它们的唯一区别是只有主分片才能处理索引请求，副本对搜索性能非常重要，同时用户也可以在任何时候添加或删除副本。额外的副本能带来更大的容量，更高的吞吐能力以及更强的故障恢复能力。

---------------------------------------------------------------------------------------------------------------------------------

分片是ElasticSearch集群分发数据的单元。ElasticSearch在重新平衡数据时可以移动分片的速度，例如发生故障后，将取决于分片的大小和数量以及网络和磁盘性能。

注意：

1.避免使用非常大的分片，因为这会对集群从故障中恢复的能力产生负面影响，对分片的大小没有固定的限制，但是通常情况下很多场景限制在50GB的分片大小以内。

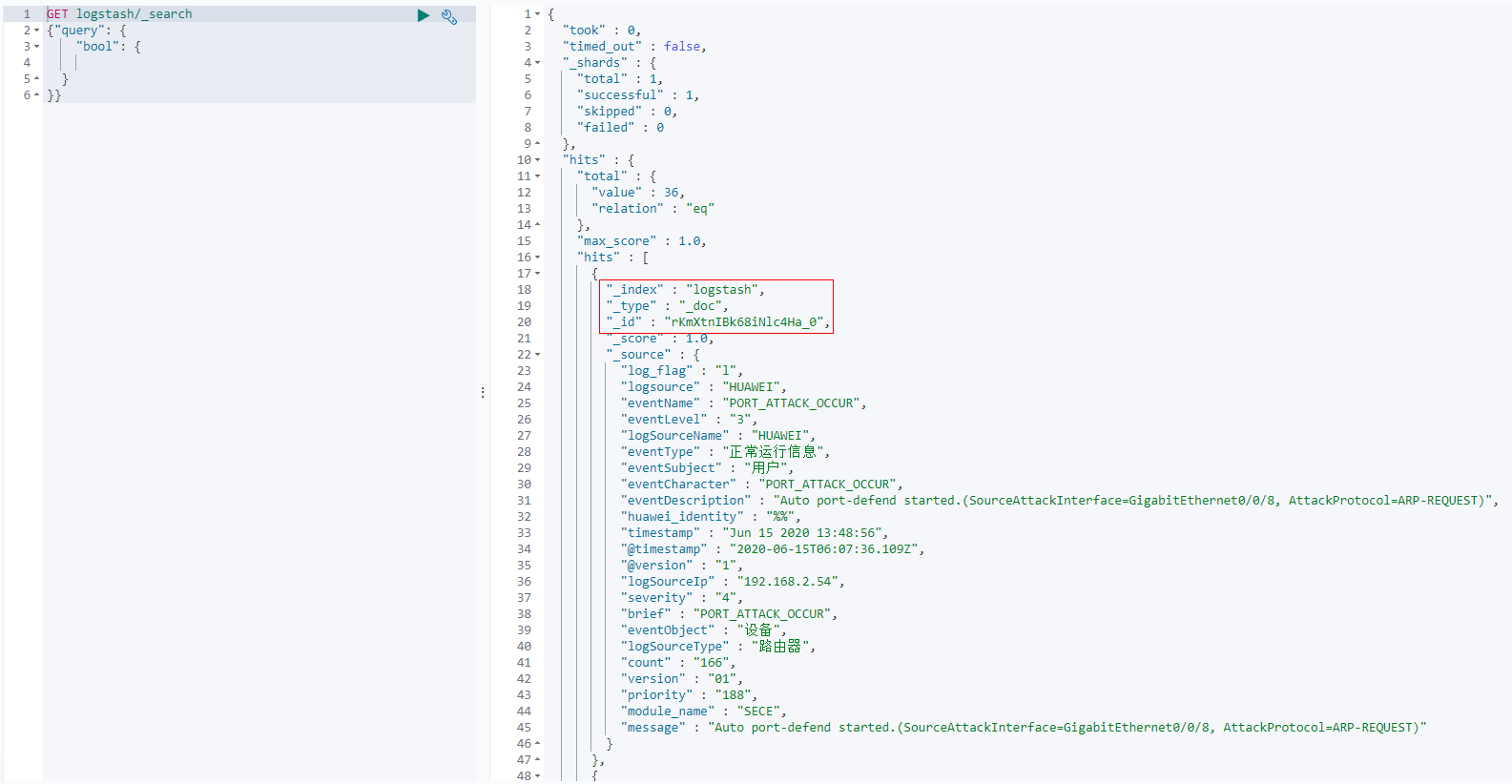
2.当在ElasticSearch集群配置好你的索引后，你要明白在集群运行中你无法调整分片设置，即便你以后发现需要调整分片数量，你也只能新建创建并对数据进行重新索引(reindex)，虽然reindex会比较耗时，但是至少能保证你不会停机。

3.尽可能使用基于时间的索引来管理数据保留期，根据保留期将数据分组到索引中，基于时间的索引可以轻松地随时间改变主分片和副本的数量，因为可以更改下一个要生成的索引。

### 索引Index

#### 索引、类型、文档

* 索引是ElasticSearch存放数据的地方，可以理解为关系型数据库中的一个数据库。事实上，我们的数据被存储和索引在分片(shards)中，索引只是一个把一个或多个分片组在一起的逻辑空间，然而，这只是一些内部细节——我们的程序完全不用关心分片。对于我们的程序而言，文档存储在索引(index)中，剩下的细节由ElasticSearch关心(索引的名字必须是全部小写，不能以下划线开头，不能包含逗号)。
* 类型(type)用于区分同一个索引下不同的数据类型，相当于关系型数据库中的表。在ElasticSearch中，我们使用相同类型(type)的文档表示相同的“事物”，因为它们的数据结构是相同的，每个类型(type)都有自己的映射(mapping)或者结构定义，就像传统数据库中的列一样。所有类型下的文档被存储在同一个索引下，但是类型的映射(mapping)会告诉ElasticSearch不同的文档如何被索引。
* 文档是ElasticSearch中存储的实体，类比关系型数据库，每一个文档相当于数据库表中的一行数据。在ElasticSearch中，文档(document)这个术语有着特殊的含义，它特指最顶层结构或者根对象(root object)序列化成的JSON数据(以唯一ID标识并存储于ElasticSearch中)。
* 文档由字段组成，相当于关系数据库中列的属性，不同的是，ElasticSearch的不同文档可以具有不同的字段集合。



文档元数据：

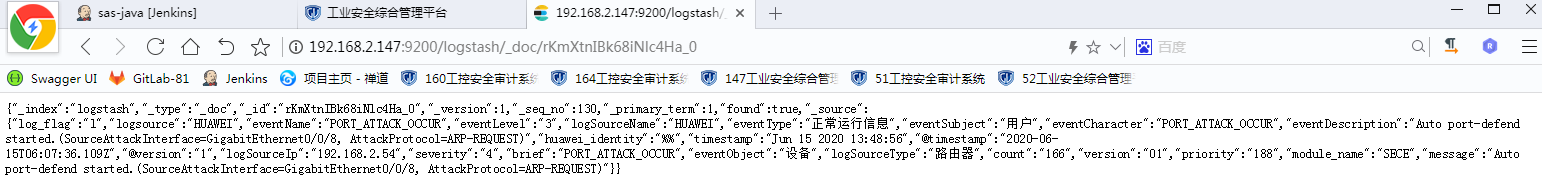
一个文档不只有数据，它还包含了元数据(metadata)——关于文档的信息。三个必须的元数据节点是：

\_index:文档存储的地方。即索引。

\_type:文档代表的对象的类。即类型。

\_id:文档的唯一标识。id仅仅是一个字符串，它与\_index和\_type组合时，就可以在ElasticSearch中唯一标识一个文档。当创建一个文档，你可以自定义\_id，也可以让ElasticSearch帮你自动生成。

通过这三元素可以唯一确定ES数据库中的文档，对于restful而言，则是一个资源，通过以下方式可以访问：



---------------------------------------------------------------------------------------------------------------------------------

1. 第2行：took表示ElasticSearch执行搜索所用的时间，单位是毫秒。
2. 第3行：time\_out:表示搜索是否超时。
3. 第4行：\_shards指示搜索了多少分片，以及搜索成功和失败的分片的计数。
4. 第10行：hits用来实际搜索结果集。
5. 第11行：hits.total是包含与搜索条件匹配的文档总数信息的对象。
6. 第12行：hits.total.value表示总命中计数的值(必须在hits.total.relation上下文中解释)。
7. 第13行：确切来说默认情况下，hits.total.value是不确切的命中计数，在这种情况下，当hits.total.relation的值是eq时，hits.total.value的值是准确计数，hits.total.relation的值是gte时，hits.total.value的值是不准确的。
8. 第16行：hits.hits是存储搜索结果的实际数据(默认为前10个文档)。

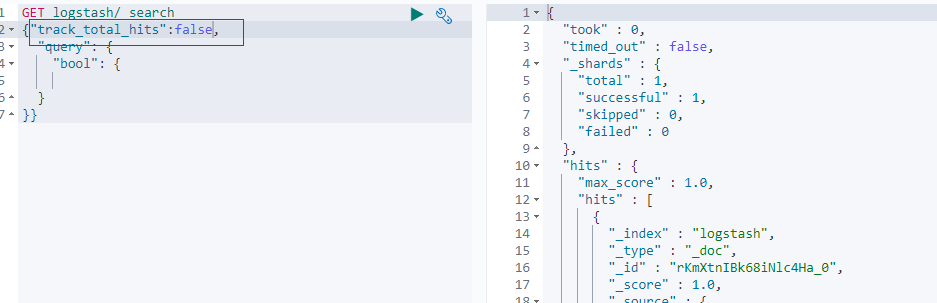
---------------------------------------------------------------------------------------------------------------------------------

注意：

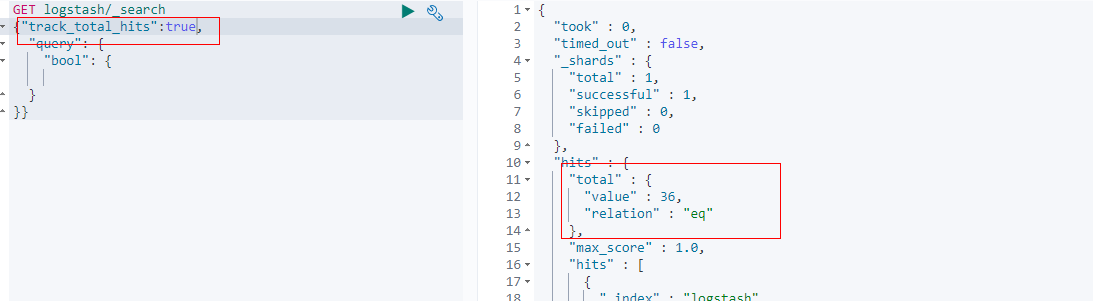
---------------------------------------------------------------------------------------------------------------------------------

ElasticSearch 7.x在查询时，必须加上“track\_total\_hits”:true，不然就只显示10000。

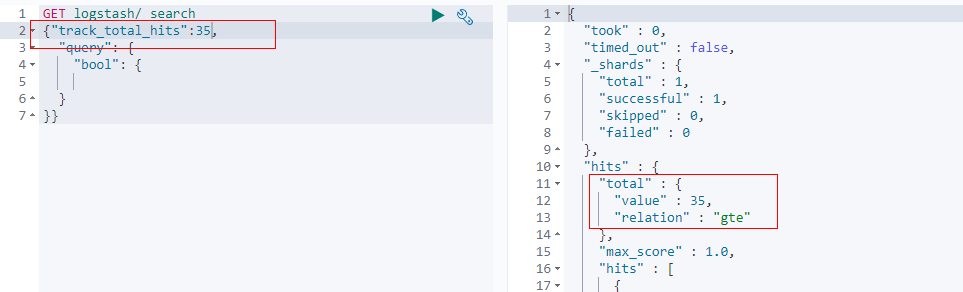
track\_total\_hits可以控制hits.total的准确性，可以设置为false、true或正整数，当为false时，不再返回total信息。



当是true时，表示精确的命中，



当时正整数时，如果是小于36，则显示：



大于等于36，则显示



那么track\_total\_hits这个参数怎么设置是最合理？

1. 保持默认值：10000，这足以满足一般的业务需求，就算是淘宝、京东这样的大型电商网站，一页展示40个结果，10000个结果可以展示250页，相信没有用户会看到250页后的商品，大多数情况下用户基本上都是浏览前10页数据。
2. 如果需要精确知道命中的玩的文档数量，此时应该把track\_total\_hits设置为true，但是用户应该明白，如果命中的文档数量很大，会影响查询性能，而且会消耗大量的内存，甚至存在内存风险的异常
3. 如果你确切知道不需要知道命中的结果数，则把track\_total\_hits设置为false，这会提升查询性能。

---------------------------------------------------------------------------------------------------------------------------------

#### 索引创建原则

1. 类似的数据放在一个索引，非类似的数据放不同索引：product index(包含了所有的商品)，sales index(包含了所有的商品销售数据)，inventory index(包含了所有库存相关的数据)。如果你把product、sales、human resources(employee)，全部都放在一个大的index里面，比如说company index，这是不合理的
2. index中包含了很多类似的document：类似是什么意思？其实指的是，这些document的fields很大一部分是相同的，如果2个document，它们的每个字段都完全不一样，这就不是类似了，就不适合放到一个index里面。
3. 索引名称必须是小写的，不能用下划线开头，不能包含逗号。

#### 索引Index 别名

在Es里面index aliases就像软连接一样，它可以映射一个或多个索引，提供了非常灵活的特性，使用它可以做到：

1. 在一个运行中的es集群中可以无缝的切换一个索引到另外一个索引上。
2. 分组多个索引，比如按月创建的索引，我们可以通过别名构造出一个最近三个月的索引。
3. 查询一个索引里面的部分数据构成一个类似数据库的视图。

## 聚合

聚合框架有助于根据搜索查询提供聚合数据。聚合查询时数据库中重要的功能特性，ES作为搜索引擎兼数据库，提供了强大的聚合分析能力，它基于查询条件来对数据进行分桶、计算的方法，有点类似于SQL中的group by再加一些函数方法的操作。聚合可以嵌套，由此可以组成复杂的操作(Bucketing聚合可以包含sub-aggregation).



--------------------------------------------------------------------------------------------------------------------------

第10行：aggregations，也可以简写为aggs。

第11行：聚合的名字。

第12行：聚合的类型。

第13行：聚合的字段。

第27行：是第一个聚合的子聚合。

--------------------------------------------------------------------------------------------------------------------------