

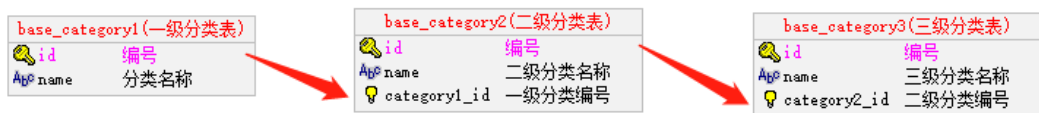
一、商品的基本知识

1.1 基本信息一分类

比如：家用电器是一级分类，电视是二级分类，那么超薄电视就是三级分类。



数据库结构



1.2 基本信息—平台属性

平台属性和平台属性值

屏幕尺寸：	70英寸及以上	65英寸	58-60英寸	55英寸	49-50英寸	45-48英寸	42-43英寸	39-40英寸	32英寸及以下
分辨率：	超高清	全高清	高清	属性值					
观看距离：	3.5米以上	3-3.5米	2.5-3米	2-2.5米	2米以内				

平台属性和平台属性值主要用于商品的检索，每个分类对应的属性都不同，分类包含一级分类、二级分类和三级分类，分类层级区分对应分类。



1.3 基本信息—销售属性与销售属性值

销售属性，就是商品详情页右边，可以通过销售属性来定位一组 spu 下的哪款 sku。可以让当前的商品详情页，跳转到自己的“兄弟”商品。

一般每种商品的销售属性不会太多，大约 1-4 种。整个平台的属性种类也不会太多，大概 10 种以内。比如：颜色、尺寸、版本、套装等等。



1.4 基本信息—spu 与 sku

SKU=Stock Keeping Unit (库存量单位)。即库存进出计量的基本单元，可以是以件，盒，托盘等为单位。SKU 这是对于大型连锁超市 DC（配送中心）物流管理的一个必要的方法。现在已经被引申为产品**统一编号**的简称，**每种产品均对应有唯一的 SKU 号**。

SPU(Standard Product Unit): 标准化产品单元。是商品信息**聚合**的最小单位，是一组**可复用、易检索**的**标准化信息的集合**，该集合描述了一个产品的**特性**。

首先通过检索搜索出来的商品列表中，每个商品都是一个 sku。每个 sku 都有自己独立的库存数。也就是说每一个商品详情展示都是一个 sku。

那 spu 又是干什么的呢？



如上图，一般的电商系统你点击进去以后，都能看到这个商品关联了其他好几个类似的商品，而且这些商品很多的信息都是共用的，比如商品图片，海报、销售属性等。

那么系统是靠什么把这些 sku 识别为一组的呢，那是这些 sku 都有一个公用的 spu 信息。而它们公共的信息，都放在 spu 信息下。

所以，sku 与 spu 的结构如下：

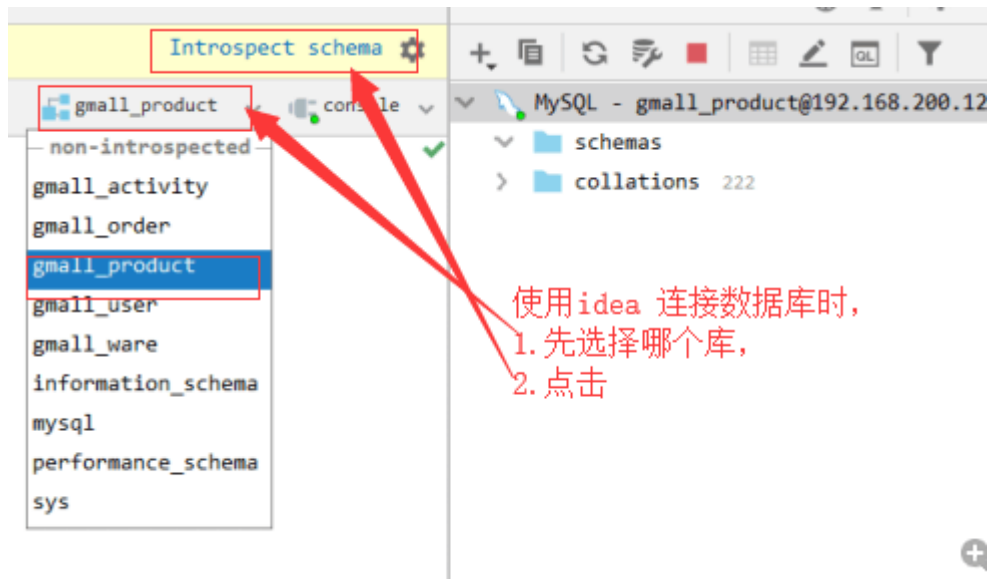


图中有两个图片信息表，其中 spu_image 表示整个 spu 相关下的所有图片信息，而 sku_image 表示这个 spu 下的某个 sku 使用的图片。sku_image 中的图片是从 spu_image 中选取的。

但是由于一个 spu 下的所有 sku 的海报都是一样，所以只存一份 spu_poster 就可以了。

二、商品管理模块开发

使用 idea 连接数据库时：

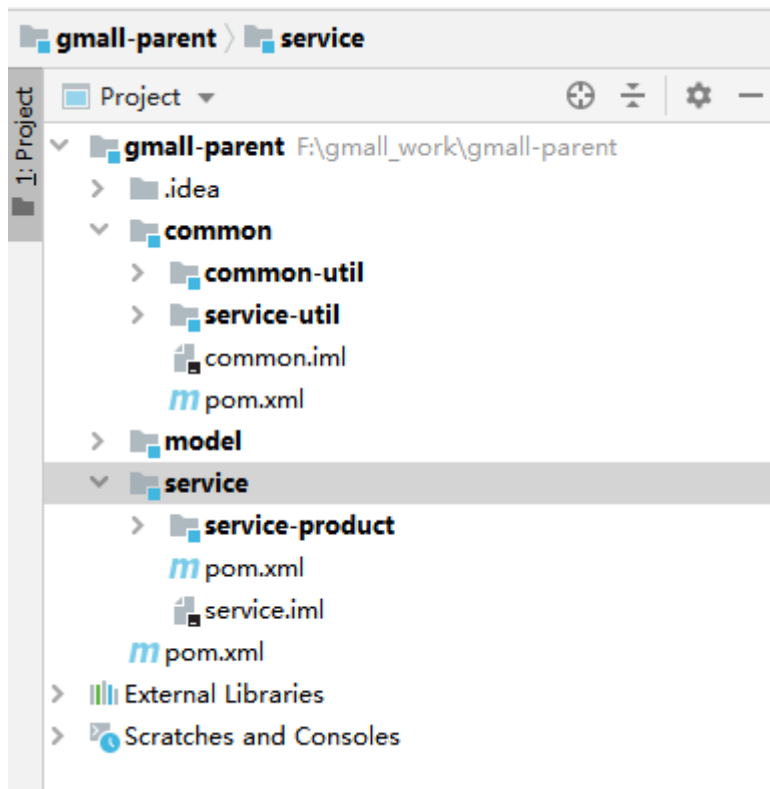


2.1 在 service 模块下搭建 service-product

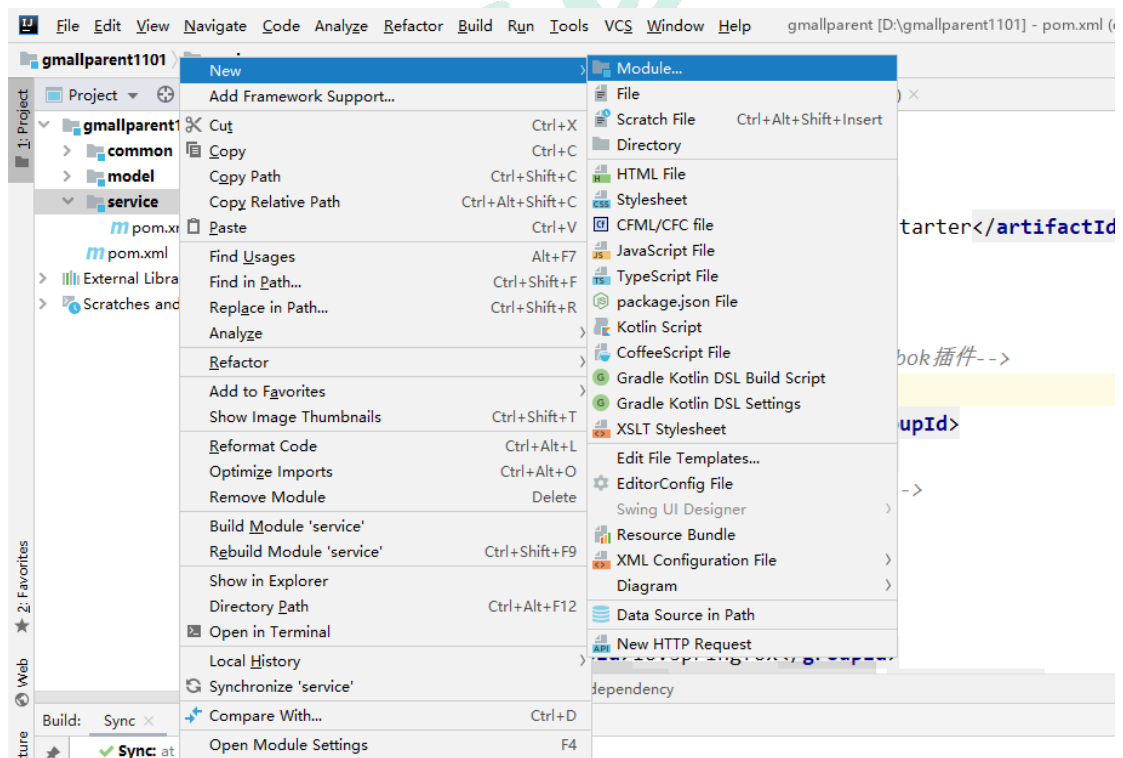
2.1.1 搭建 service-product

搭建过程同 common-util

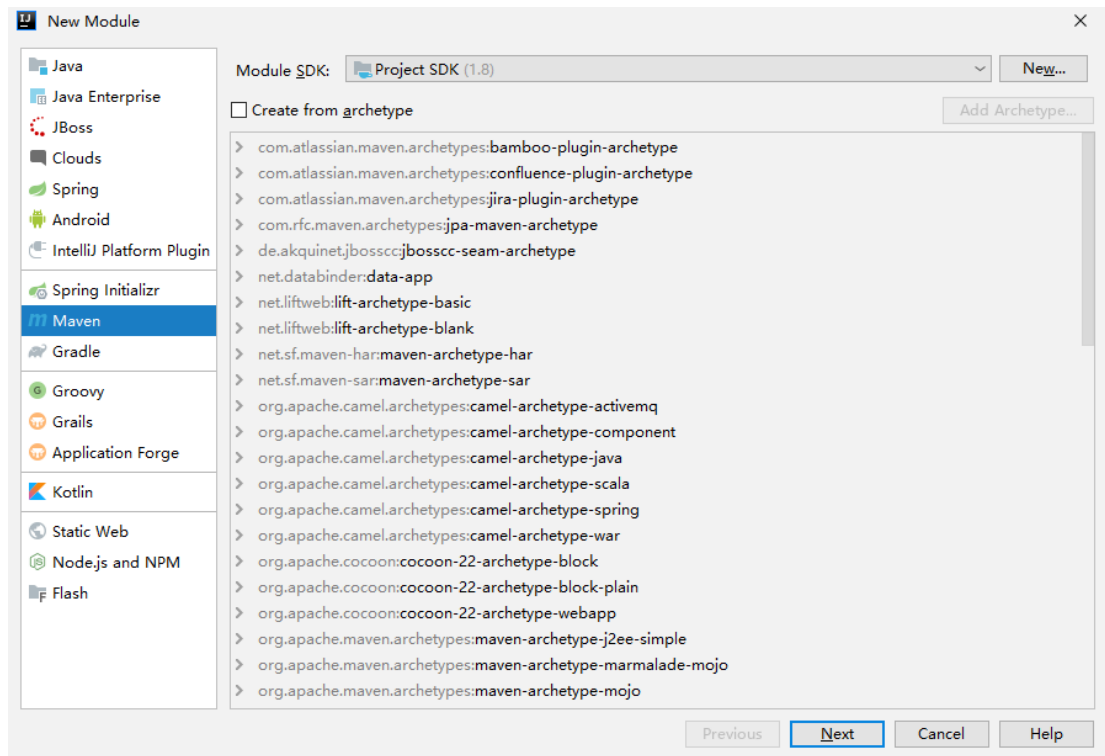
如图



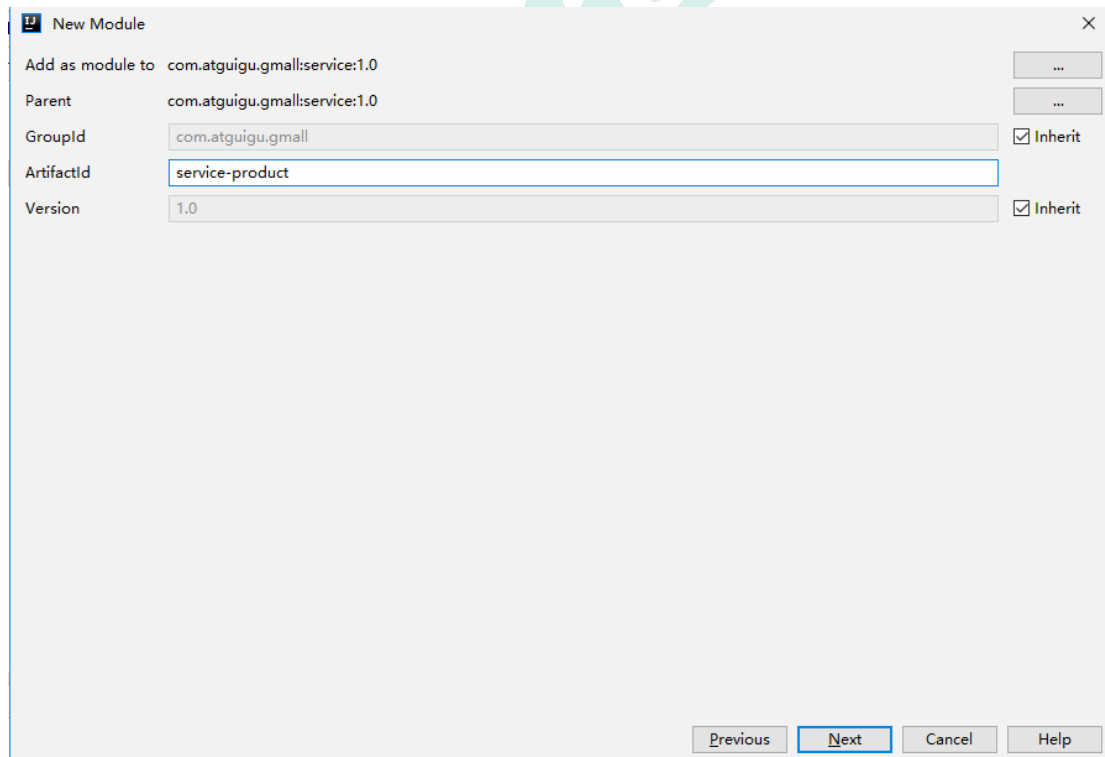
图一：



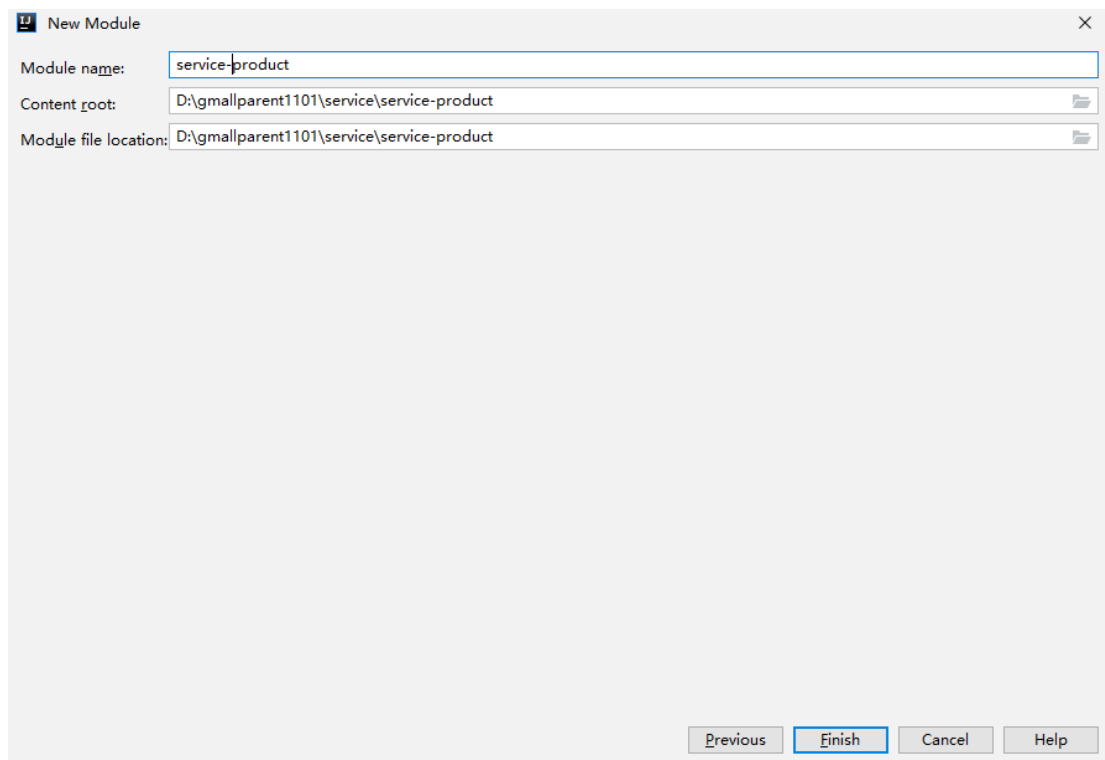
图二：



图三：



图四：



2.1.2 修改配置

修改 pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>com.atguigu.gmall</groupId>
    <artifactId>service</artifactId>
    <version>1.0</version>
  </parent>

  <artifactId>service-product</artifactId>
  <version>1.0</version>

  <packaging>jar</packaging>
  <name>service-product</name>
  <description>service-product</description>

</project>
```


添加配置文件 application.yml

```
spring:
  application:
    name: service-product
  profiles:
    active: dev
  cloud:
    nacos:
      discovery:
        server-addr: 127.0.0.1:8848
```

添加配置文件 application-dev.yml

```
server:
  port: 8206
mybatis-plus:
  mapper-locations: classpath:mapper/*Mapper.xml
  configuration:
    log-impl: org.apache.ibatis.logging.stdout.StdOutImpl
spring:
  datasource:
    driver-class-name: com.mysql.jdbc.Driver
    username: root
    password: root
    url: jdbc:mysql://192.168.200.129:3306/gmall_product
  redis:
    host: 192.168.200.129
    password:
    database: 0
    port: 6379
  zipkin:
    base-url: http://192.168.200.129:9411
  rabbitmq:
    host: 192.168.200.129
    port: 5672
    username: guest
    password: guest
```

2.1.3 创建启动类

包名: com.atguigu.gmall.product

```
@SpringBootApplication
@ComponentScan({"com.atguigu.gmall"})
@EnableDiscoveryClient
public class ServiceProductApplication {
    public static void main(String[] args) {
        SpringApplication.run(ServiceProductApplication.class, args);
    }
}
```

ServiceProductApplication 类

2.2 搭建后台页面

1. 拷贝资料中的前端项目页面，放入一个没有中文目录的文件下
2. 在 gmall-admin 当前目录下 cmd 回车
 - a) npm install [安装依赖 node_modules] 项目库中有 node_modules 就不需要执行 npm install
 - b) npm run dev
 - c) 直接访问浏览器
3. 将两个配置文件
 - a) dev.env.js <http://localhost>
 - b) index.js host: 'localhost', port: [8888](#)

注：第一个为网关地址，第二个为项目访问地址

注意：如果出现未找到 node-sass 模块，只需要在窗口中运行

npm install node-sass 即可，然后重新 npm install, npm run dev

运行项目的时候，没有提示 node-sass 模块为找到，需要看一下当前的 nodejs 版本

node -v : 建议 v10.15.3

2.3 属性管理功能

一级分类

手机

二级分类

手机通讯

三级分类

手机

+ 添加平台属性

序号	属性id	属性名称	操作
1	106	手机系统	
2	107	手机品牌	
3	23	运行内存	修改
4	24	机身内存	修改
5	111	颜色	修改
6	112	VV	修改

2.3.1 分类信息及属性的查询

2.3.1.1 创建 Mapper

包名: com.atguigu.gmall.product.mapper

BaseCategory1Mapper

```
@Mapper
public interface BaseCategory1Mapper extends
    BaseMapper<BaseCategory1> {
}
```

BaseCategory2Mapper

```
@Mapper
public interface BaseCategory2Mapper extends
    BaseMapper<BaseCategory2> {
}
```

BaseCategory3Mapper

```
@Mapper
public interface BaseCategory3Mapper extends
    BaseMapper<BaseCategory3> {
}
```

BaseAttrInfoMapper

```
@Mapper
public interface BaseAttrInfoMapper extends BaseMapper<BaseAttrInfo>
{
}
```

```
}
```

BaseAttrValueMapper

```
@Mapper
public interface BaseAttrValueMapper extends
BaseMapper<BaseAttrValue> {
}
```

2.3.1.2 创建 service 接口

包名: com.atguigu.gmall.product.service

接口方法是根据页面得来

```
public interface ManageService {

    /**
     * 查询所有的一级分类信息
     * @return
     */
    List<BaseCategory1> getCategory1();

    /**
     * 根据一级分类Id 查询二级分类数据
     * @param category1Id
     * @return
     */
    List<BaseCategory2> getCategory2(Long category1Id);

    /**
     * 根据二级分类Id 查询三级分类数据
     * @param category2Id
     * @return
     */
    List<BaseCategory3> getCategory3(Long category2Id);

    /**
     * 根据分类Id 获取平台属性数据
     * 接口说明:
     * 1, 平台属性可以挂在一级分类、二级分类和三级分类
     * 2, 查询一级分类下面的平台属性, 传: category1Id, 0, 0; 取出该分类
    的平台属性
     * 3, 查询二级分类下面的平台属性, 传: category1Id, category2Id, 0;
     * 取出对应一级分类下面的平台属性与二级分类对应的平台属性
     */
}
```

```
*      4, 查询三级分类下面的平台属性, 传: category1Id, category2Id,
category3Id;
*      取出对应一级分类、二级分类与三级分类对应的平台属性
*
* @param category1Id
* @param category2Id
* @param category3Id
* @return
*/
List<BaseAttrInfo> getAttrInfoList(Long category1Id, Long
category2Id, Long category3Id);
}
```

2.3.1.3 创建实现类 ManageServiceImpl

包名: com.atguigu.gmall.product.service.impl

增加实现类

```
@Service
public class ManageServiceImpl implements ManageService {
    @Autowired
    private BaseCategory1Mapper baseCategory1Mapper;

    @Autowired
    private BaseCategory2Mapper baseCategory2Mapper;

    @Autowired
    private BaseCategory3Mapper baseCategory3Mapper;

    @Autowired
    private BaseAttrInfoMapper baseAttrInfoMapper;

    @Autowired
    private BaseAttrValueMapper baseAttrValueMapper;

    @Override
    public List<BaseCategory1> getCategory1() {
        return baseCategory1Mapper.selectList(null);
    }

    @Override
    public List<BaseCategory2> getCategory2(Long category1Id) {
        // select * from baseCategory2 where Category1Id = ?
        QueryWrapper queryWrapper = new
        QueryWrapper<BaseCategory2>();
        queryWrapper.eq("category1_id", category1Id);
        List<BaseCategory2> baseCategory2List =
}
```

```
baseCategory2Mapper.selectList(queryWrapper);
    return baseCategory2List;
}

@Override
public List<BaseCategory3> getCategory3(Long category2Id) {
    // select * from baseCategory3 where Category2Id = ?
    QueryWrapper queryWrapper = new
    QueryWrapper<BaseCategory3>();
    queryWrapper.eq("category2_id", category2Id);
    return baseCategory3Mapper.selectList(queryWrapper);
}

@Override
public List<BaseAttrInfo> getAttrInfoList(Long category1Id, Long
category2Id, Long category3Id) {
    // 调用 mapper:
    return baseAttrInfoMapper.selectBaseAttrInfoList(category1Id,
category2Id, category3Id);
}
}
```

2.3.1.4 在 BaseAttrInfoMapper 类添加方法

```
/**
 * 根据分类Id 查询平台属性集合对象 | 编写 xml 文件
 * @param category1Id
 * @param category2Id
 * @param category3Id
 * @return
 */
List<BaseAttrInfo> selectBaseAttrInfoList(@Param("category1Id")Long
category1Id, @Param("category2Id")Long category2Id,
@Param("category3Id")Long category3Id);
```

2.3.1.5 在 BaseAttrInfoMapper.xml 添加查询方法

在 resources 目录添加 mapper 文件夹，添加 BaseAttrInfoMapper.xml 文件

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper SYSTEM "http://mybatis.org/dtd/mybatis-3-
mapper.dtd" >
<!-- namespace 定义接口的全路径 -->
<mapper
```

```
namespace="com.atguigu.gmall.product.mapper.BaseAttrInfoMapper">
    <!--
        resultMap: 表示返回的映射结果集
        id : 表示唯一标识
        type: 表示返回结果集的数据类型
        autoMapping : 自动映射
    -->
    <resultMap id="baseAttrInfoMap"
type="com.atguigu.gmall.model.product.BaseAttrInfo"
autoMapping="true">
        <!--id: 表示主键 property: 表示实体类的属性名 column: 表示通过
sql 执行以后查询出来的字段名-->
        <id property="id" column="id"></id>
        <!--result : 表示映射普通字段-->
        <!--<result property="" column=""></result>-->
        <!--mybatis 如何配置一对多-->
        <!--ofType : 返回的数据类型-->
        <collection property="attrValueList"
ofType="com.atguigu.gmall.model.product.BaseAttrValue"
autoMapping="true">
            <!--如果有字段重复则起别名-->
            <id property="id" column="attr_value_id"></id>
        </collection>
    </resultMap>
    <!--id: 表示方法名-->
    <select id="selectBaseAttrInfoList" resultMap="baseAttrInfoMap">
        SELECT
            bai.id,
            bai.attr_name,
            bai.category_id,
            bai.category_level,
            bav.id attr_value_id,
            bav.value_name,
            bav.attr_id
        FROM
            base_attr_info bai
        INNER JOIN base_attr_value bav ON bai.id = bav.attr_id
        <where>
            <if test="category1Id != null and category1Id != 0">
                or (bai.category_id = #{category1Id} and
bai.category_level = 1)
            </if>
            <if test="category2Id != null and category2Id != 0">
                or (bai.category_id = #{category2Id} and
bai.category_level = 2)
            </if>
            <if test="category3Id != null and category3Id != 0">
                or (bai.category_id = #{category3Id} and
bai.category_level = 3)
            </if>
        </where>
        order by bai.category_level, bai.id
    </select>
</namespace>
```

```
</select>
```

```
</mapper>
```

2.3.1.6 创建 BaseManageController

针对平台属性的操作!

```
@Api(tags = "商品基础属性接口")
@RestController
@RequestMapping("admin/product")
public class BaseManageController {

    @Autowired
    private ManageService manageService;

    /**
     * 查询所有的一级分类信息
     * @return
     */
    @GetMapping("getCategory1")
    public Result<List<BaseCategory1>> getCategory1() {
        List<BaseCategory1> baseCategory1List =
manageService.getCategory1();
        return Result.ok(baseCategory1List);
    }

    /**
     * 根据一级分类Id 查询二级分类数据
     * @param category1Id
     * @return
     */
    @GetMapping("getCategory2/{category1Id}")
    public Result<List<BaseCategory2>>
getCategory2(@PathVariable("category1Id") Long category1Id) {
        List<BaseCategory2> baseCategory2List =
manageService.getCategory2(category1Id);
        return Result.ok(baseCategory2List);
    }

    /**
     * 根据二级分类Id 查询三级分类数据
     * @param category2Id
     * @return
     */
    @GetMapping("getCategory3/{category2Id}")
    public Result<List<BaseCategory3>>
getCategory3(@PathVariable("category2Id") Long category2Id) {
```



```

        List<BaseCategory3>                baseCategory3List                =
manageService.getCategory3(category2Id);
        return Result.ok(baseCategory3List);
    }

    /**
     * 根据分类Id 获取平台属性数据
     * @param category1Id
     * @param category2Id
     * @param category3Id
     * @return
     */

@GetMapping("attrInfoList/{category1Id}/{category2Id}/{category3Id}"
)
    public                                Result<List<BaseAttrInfo>>
attrInfoList(@PathVariable("category1Id") Long category1Id,

@PathVariable("category2Id") Long category2Id,

@PathVariable("category3Id") Long category3Id) {
        List<BaseAttrInfo>                baseAttrInfoList                =
manageService.getAttrInfoList(category1Id,                                category2Id,
category3Id);
        return Result.ok(baseAttrInfoList);
    }
}

```

<http://localhost:8206/swagger-ui.html> 测试数据接口!

2.3.2 属性的添加

数据库中的表!



属性名称

VV

+ 添加属性值

序号	属性值id	属性值名称	操作
1	185	VV1	删除
2	186	VV2	删除
3	187	VV3	删除
4	188	测试中文	删除

2.3.2.1 创建 service 接口

在 ManageService 添加 service 接口

```

/**
 * 保存平台属性方法
 * @param baseAttrInfo
 */
void saveAttrInfo(BaseAttrInfo baseAttrInfo);

@Override
@Transactional(rollbackFor = Exception.class)
public void saveAttrInfo(BaseAttrInfo baseAttrInfo) {
    // 什么情况下是添加，什么情况下是更新，修改 根据 baseAttrInfo 的 Id
    // baseAttrInfo
    if (baseAttrInfo.getId() != null) {
        // 修改数据
        baseAttrInfoMapper.updateById(baseAttrInfo);
    } else {
        // 新增
        // baseAttrInfo 插入数据
        baseAttrInfoMapper.insert(baseAttrInfo);
    }

    // baseAttrValue 平台属性值
    // 修改：通过先删除{baseAttrValue}，在新增的方式！
    // 删除条件：baseAttrValue.attrId = baseAttrInfo.id
    QueryWrapper queryWrapper = new QueryWrapper<BaseAttrValue>();
    queryWrapper.eq("attr_id", baseAttrInfo.getId());
    baseAttrValueMapper.delete(queryWrapper);

    // 获取页面传递过来的所有平台属性值数据
    List<BaseAttrValue> attrValueList =
    baseAttrInfo.getAttrValueList();
    if (attrValueList != null && attrValueList.size() > 0) {
        // 循环遍历
        for (BaseAttrValue baseAttrValue : attrValueList) {
            // 获取平台属性 Id 给 attrId
            baseAttrValue.setAttrId(baseAttrInfo.getId()); // ?
            baseAttrValueMapper.insert(baseAttrValue);
        }
    }
}

```

```
}  
}
```

注意:

实现类添加: `@Transactional`

2.3.2.2 创建控制器

BaseManageController

```
/**  
 * 保存平台属性方法  
 * @param baseAttrInfo  
 * @return  
 */  
@PostMapping("saveAttrInfo")  
public Result saveAttrInfo(@RequestBody BaseAttrInfo baseAttrInfo) {  
    // 前台数据都被封装到该对象中 baseAttrInfo  
    manageService.saveAttrInfo(baseAttrInfo);  
    return Result.ok();  
}
```

2.3.3 回显平台属性

2.3.3.1 接口

接口

选中准修改数据, 根据该 attrId 去查找 AttrInfo, 该对象下 List<BaseAttrValue> !

所以在返回的时候, 需要返回 BaseAttrInfo

```
/**  
 * 根据 attrId 查询平台属性对象  
 * @param attrId  
 * @return  
 */  
BaseAttrInfo getAttrInfo(Long attrId);
```

2.3.3.2 实现类

实现类

`@Override`

```
public BaseAttrInfo getAttrInfo(Long attrId) {
    BaseAttrInfo baseAttrInfo = baseAttrInfoMapper.selectById(attrId);
    // 查询到最新的平台属性值集合数据放入平台属性中!
    baseAttrInfo.setAttrValueList(getAttrValueList(attrId));
    return baseAttrInfo;
}

/**
 * 根据属性 id 获取属性值
 * @param attrId
 * @return
 */
private List<BaseAttrValue> getAttrValueList(Long attrId) {
    // select * from baseAttrValue where attrId = ?
    QueryWrapper queryWrapper = new QueryWrapper<BaseAttrValue>();
    queryWrapper.eq("attr_id", attrId);
    List<BaseAttrValue> baseAttrValueList =
    baseAttrValueMapper.selectList(queryWrapper);
    return baseAttrValueList;
}
```

2.3.3.3 控制器

```
@GetMapping("getAttrValueList/{attrId}")
public Result<List<BaseAttrValue>> getAttrValueList(@PathVariable("attrId")
Long attrId) {
    BaseAttrInfo baseAttrInfo = manageService.getAttrInfo(attrId);
    List<BaseAttrValue> baseAttrValueList =
    baseAttrInfo.getAttrValueList();
    return Result.ok(baseAttrValueList);
}
```

问题：如果出现乱码问题

查看：

SHOW VARIABLES LIKE '%char%';

处理：

到 mysql 容器中执行

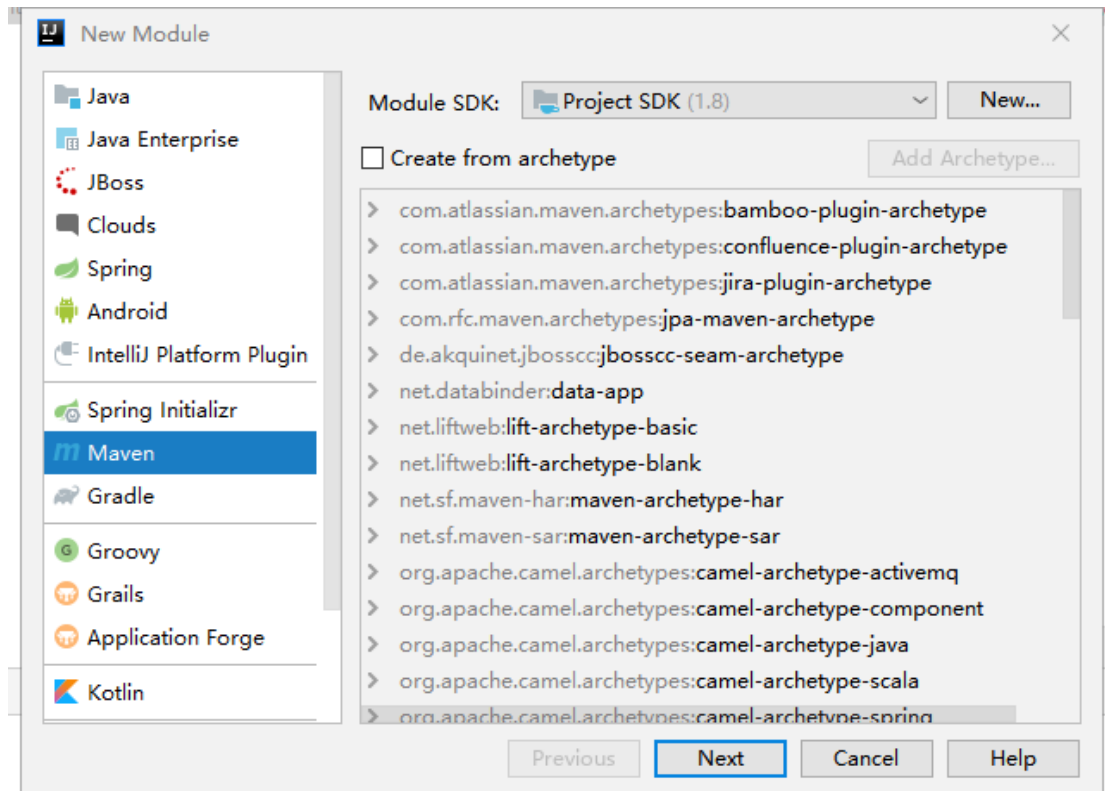
```
echo "character-set-server=utf8" >> /etc/mysql/mysql.conf.d/mysqld.cnf
```

2.4 搭建 server-gateway 模块

服务网关

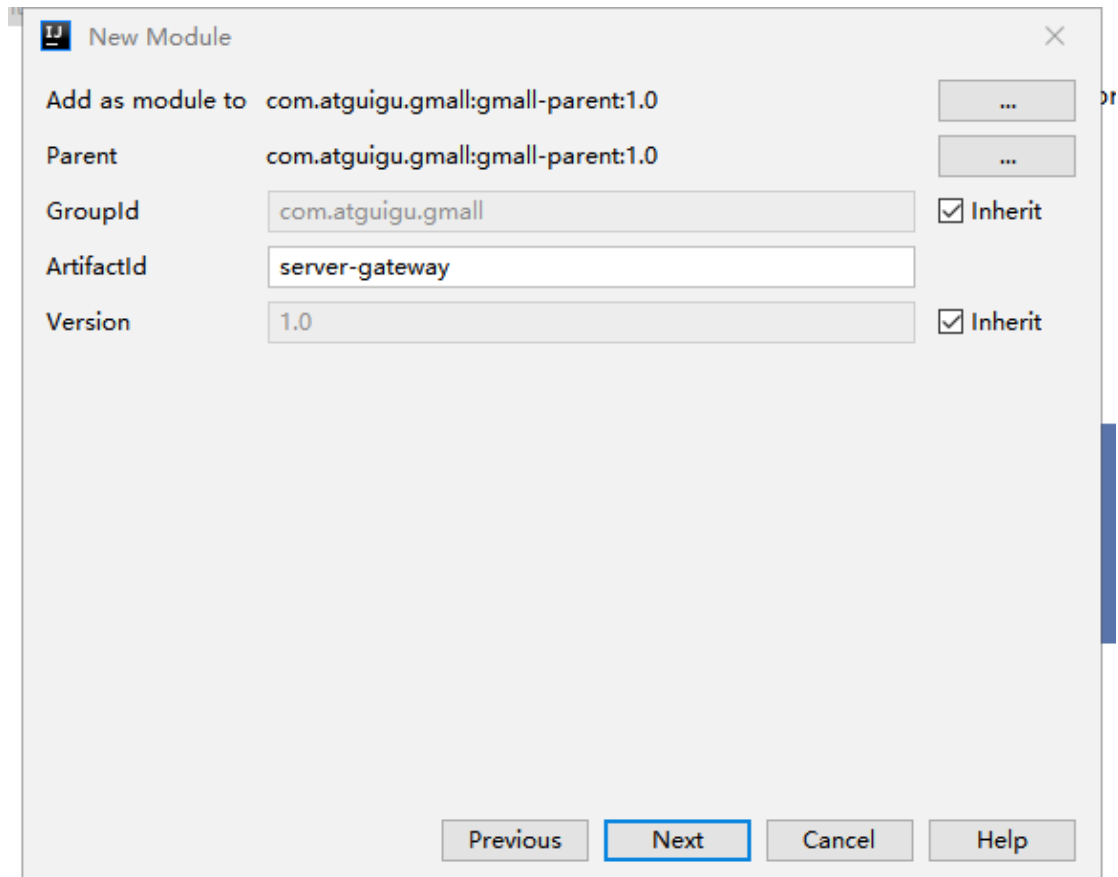
2.4.1 搭建 server-gateway

点击 gmall-parent, 选择 New->Module,操作如下



点击下一步





New Module

Add as module to: com.atguigu.gmall:gmall-parent:1.0

Parent: com.atguigu.gmall:gmall-parent:1.0

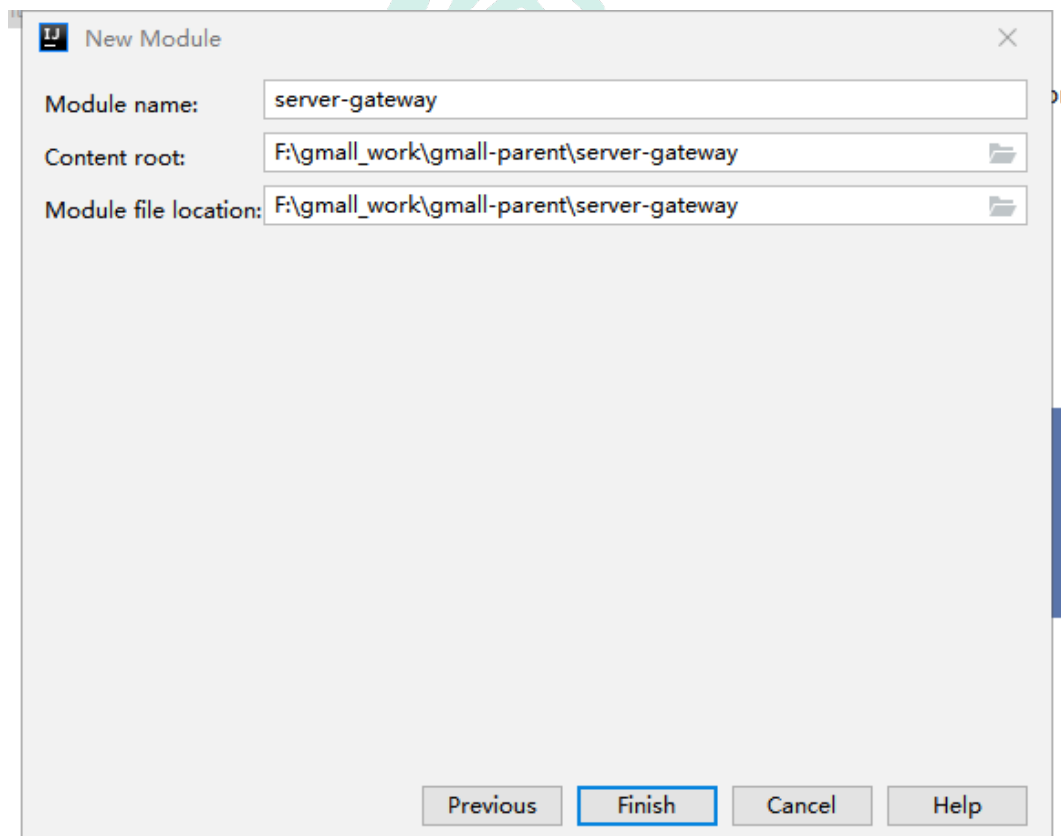
GroupId: com.atguigu.gmall ☒ Inherit

ArtifactId: server-gateway

Version: 1.0 ☒ Inherit

Previous Next Cancel Help

点击下一步



New Module

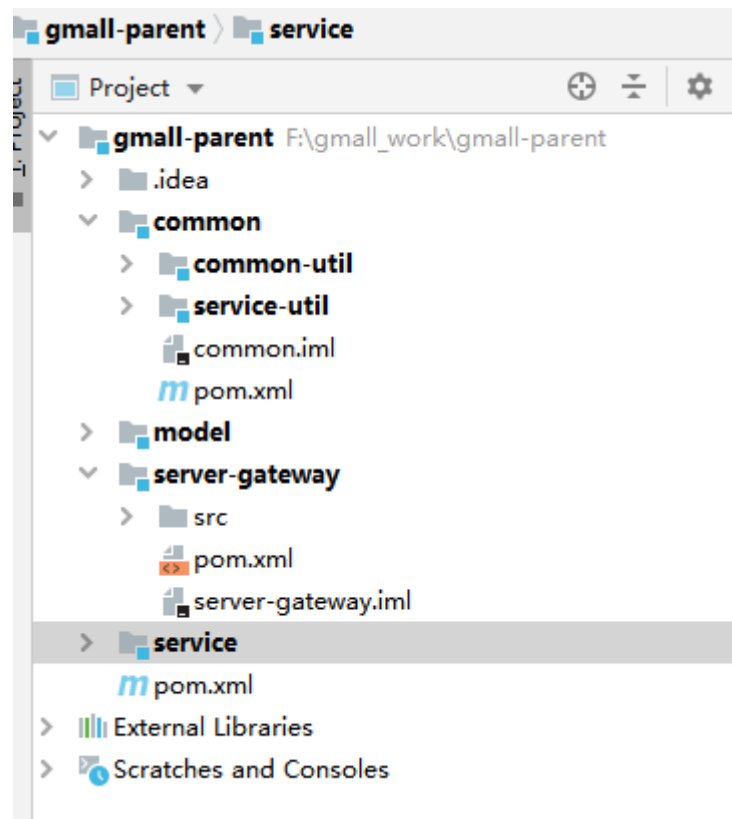
Module name: server-gateway

Content root: F:\gmall_work\gmall-parent\server-gateway

Module file location: F:\gmall_work\gmall-parent\server-gateway

Previous Finish Cancel Help

点击完成



2.4.2 修改配置 pom.xml

修改 pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <parent>
    <groupId>com.atguigu.gmall</groupId>
    <artifactId>gmall-parent</artifactId>
    <version>1.0</version>
  </parent>

  <version>1.0</version>
  <artifactId>server-gateway</artifactId>

  <packaging>jar</packaging>
  <name>server-gateway</name>

  <dependencies>
```

```
<dependency>
  <groupId>com.atguigu.gmall</groupId>
  <artifactId>common-util</artifactId>
  <version>1.0</version>
</dependency>

<!-- 服务注册 -->
<dependency>
  <groupId>com.alibaba.cloud</groupId>
  <artifactId>spring-cloud-starter-alibaba-nacos-
discovery</artifactId>
</dependency>

<!-- 服务配置-->

<!--
  <dependency>
    <groupId>com.alibaba.cloud</groupId>
    <artifactId>spring-cloud-starter-alibaba-nacos-
config</artifactId>
  </dependency>
-->

  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-gateway</artifactId>
  </dependency>
</dependencies>
</project>
```

2.4.3 在 resources 下添加配置文件

启动类:

```
package com.atguigu.gmall.gateway;

@SpringBootApplication
public class ServerGatewayApplication {
    public static void main(String[] args) {

        SpringApplication.run(ServerGatewayApplication.class,args);
    }
}
```

application.yml


```
server:
  port: 80
spring:
  application:
    name: api-gateway
  cloud:
    nacos:
      discovery:
        server-addr: localhost:8848
    gateway:
      discovery:      #是否与服务发现组件进行结合，通过 serviceId(必须设置成大写) 转发到具体的服务实例。默认为 false，设为 true 便开启通过服务中心的自动根据 serviceId 创建路由的功能。
      locator:        #路由访问方式：http://Gateway_HOST:Gateway_PORT/ 大写的 serviceId/**，其中微服务应用名默认大写访问。
      enabled: true
      routes:
        - id: service-product
          uri: lb://service-product
          predicates:
            - Path=/*/product/** # 路径匹配
```

2.4.4 跨域处理

跨域：浏览器对于 javascript 的同源策略的限制。

以下情况都属于跨域：

跨域原因说明	示例
域名不同	<code>www.jd.com</code> 与 <code>www.taobao.com</code>
域名相同，端口不同	<code>www.jd.com:8080</code> 与 <code>www.jd.com:8081</code>
二级域名不同	<code>item.jd.com</code> 与 <code>miaosha.jd.com</code>

如果域名和端口都相同，但是请求路径不同，不属于跨域，如：

`www.jd.com/item`

`www.jd.com/goods`

http 和 https 也属于跨域

而我们刚才就是从 `localhost:1000` 去访问 `localhost:8888`，这属于端口不同，跨域了。

2.4.4.1 为什么有跨域问题？

跨域不一定都会有跨域问题。

因为跨域问题是浏览器对于 ajax 请求的一种安全限制：一个页面发起的 ajax 请求，只能是与当前页域名相同的路径，这能有效阻止跨站攻击。

因此：跨域问题 是针对 ajax 的一种限制。

但是这却给我们的开发带来了不便，而且在实际生产环境中，肯定会有很多台服务器之间交互，地址和端口都可能不同，怎么办？

2.4.4.2 解决跨域问题的方案

目前比较常用的跨域解决方案有 3 种：

- Jsonp

最早的解决方案，利用 script 标签可以跨域的原理实现。

<https://www.w3cschool.cn/json/json-jsonp.html>

限制：

- 需要服务的支持
- 只能发起 GET 请求

- nginx 反向代理

思路是：利用 nginx 把跨域反向代理为不跨域，支持各种请求方式

缺点：需要在 nginx 进行额外配置，语义不清晰

- CORS

规范化的跨域请求解决方案，安全可靠。

优势：

- 在服务端进行控制是否允许跨域，可自定义规则
- 支持各种请求方式

缺点：

- 会产生额外的请求

我们这里会采用 cors 的跨域方案。

2.4.4.3 什么是 cors

CORS 是一个 W3C 标准，全称是"跨域资源共享"（Cross-origin resource sharing）。

它允许浏览器向跨源服务器，发出 XMLHttpRequest 请求，从而克服了 AJAX 只能同源使用的限制。

CORS 需要浏览器和服务器同时支持。目前，所有浏览器都支持该功能，IE 浏览器不能低于 IE10。

- 浏览器端：不用考虑

目前，所有浏览器都支持该功能（IE10 以下不行）。整个 CORS 通信过程，都是浏览器自动完成，不需要用户参与。

- 服务端：进行相关设置

CORS 通信与 AJAX 没有任何差别，因此你不需要改变以前的业务逻辑。只不过，浏览器会在请求中携带一些头信息，我们需要以此判断是否允许其跨域，然后在响应头中加入一些信息即可。这一般通过过滤器完成即可。

2.4.4.4 原理有点复杂

预检请求

跨域请求会在正式通信之前，增加一次 HTTP 查询请求，称为"预检"请求（preflight）。

浏览器先询问服务器，当前网页所在的域名是否在服务器的许可名单之中，以及可以使用哪些 HTTP 动词和头信息字段。只有得到肯定答复，浏览器才会发出正式的 XMLHttpRequest 请求，否则就报错。

一个“预检”请求的样板：

```
OPTIONS /cors HTTP/1.1
```

```
Origin: http://localhost:1000
```

```
Access-Control-Request-Method: GET
```

```
Access-Control-Request-Headers: X-Custom-Header
```

```
User-Agent: Mozilla/5.0...
```

- Origin：会指出当前请求属于哪个域（协议+域名+端口）。服务会根据这个值决定是否允许其跨域。
- Access-Control-Request-Method：接下来会用到的请求方式，比如 PUT
- Access-Control-Request-Headers：会额外用到的头信息

预检请求的响应

服务的收到预检请求，如果许可跨域，会发出响应：

```
HTTP/1.1 200 OK
```

```
Date: Mon, 01 Dec 2008 01:15:39 GMT
```

```
Server: Apache/2.0.61 (Unix)
```

```
Access-Control-Allow-Origin: http://localhost:1000
```

```
Access-Control-Allow-Credentials: true
```

```
Access-Control-Allow-Methods: GET, POST, PUT
```

```
Access-Control-Allow-Headers: X-Custom-Header
```

```
Access-Control-Max-Age: 1728000
```

```
Content-Type: text/html; charset=utf-8
```

```
Content-Encoding: gzip
```

Content-Length: 0

Keep-Alive: timeout=2, max=100

Connection: Keep-Alive

Content-Type: text/plain

如果服务器允许跨域，需要在返回的响应头中携带下面信息：

- Access-Control-Allow-Origin：可接受的域，是一个具体域名或者*（代表任意域名）
- Access-Control-Allow-Credentials：是否允许携带 cookie，默认情况下，cors 不会携带 cookie，除非这个值是 true
- Access-Control-Allow-Methods：允许访问的方式
- Access-Control-Allow-Headers：允许携带的头
- Access-Control-Max-Age：本次许可的有效时长，单位是秒，过期之前的 ajax 请求就无需再次进行预检了

有关 cookie：

要想操作 cookie，需要满足以下条件：

- 服务的响应头中需要携带 Access-Control-Allow-Credentials 并且为 true。
- 浏览器发起 ajax 需要指定 withCredentials 为 true

2.4.4.5 在网关中实现跨域

全局配置类实现

包名：com.atguigu.gmall.gateway.config

CorsConfig 类

```
@Configuration
public class CorsConfig {
    @Bean
```

```
public CorsWebFilter corsWebFilter(){

    // cors 跨域配置对象
    CorsConfiguration configuration = new CorsConfiguration();
    configuration.addAllowedOrigin("*"); //设置允许访问的网络
    configuration.setAllowCredentials(true); // 设置是否从服务器获取 cookie
    configuration.addAllowedMethod("*"); // 设置请求方法 * 表示任意
    configuration.addAllowedHeader("*"); // 所有请求头信息 * 表示任意

    // 配置源对象
    UrlBasedCorsConfigurationSource configurationSource = new
    UrlBasedCorsConfigurationSource();
    configurationSource.registerCorsConfiguration("/**",
    configuration);
    // cors 过滤器对象
    return new CorsWebFilter(configurationSource);
}
```

2.5 配置文件迁移 nacos

2.5.1 安装 nacos

- 1, 重新安装 nacos, nacos 数据保存至 mysql, 先删除已安装的 nacos, 再安装
- 2, 资源库获取 nacos 数据库表结构并且导入数据库
- 3, 更改 nacos 启动配置参数

```
docker run -d \
-e MODE=standalone \
-e PREFER_HOST_MODE=hostname \
-e SPRING_DATASOURCE_PLATFORM=mysql \
-e MYSQL_SERVICE_HOST=192.168.200.129 \
-e MYSQL_SERVICE_PORT=3306 \
-e MYSQL_SERVICE_USER=root \
-e MYSQL_SERVICE_PASSWORD=root \
-e MYSQL_SERVICE_DB_NAME=nacos \
```

```
-p 8848:8848 \  
--name nacos \  
--restart=always \  
nacos/nacos-server:1.4.1
```

2.5.2 添加依赖

在 service，还有 gateway 父模块添加依赖

```
<dependency>  
    <groupId>com.alibaba.cloud</groupId>  
    <artifactId>spring-cloud-starter-alibaba-nacos-  
config</artifactId>  
</dependency>
```

说明：搭建环境是我们注释了的，现在打开

2.5.3 改造 service-product

删除之前的配置文件

1，添加配置文件 bootstrap.properties

```
#server.port = 8206  
spring.application.name=service-product  
spring.profiles.active=dev  
spring.cloud.nacos.discovery.server-addr=192.168.200.128:8848  
spring.cloud.nacos.config.server-addr=192.168.200.128:8848  
spring.cloud.nacos.config.prefix=${spring.application.name}  
spring.cloud.nacos.config.file-extension=yaml  
spring.cloud.nacos.config.shared-configs[0].data-id=common.yaml
```

说明：

1，配置文件统一配置到 nacos 配置中心

2，common.yaml 为公共配置，后续有需要的 service 模块都可直接引用，避免重复配置

2，common.yaml 配置文件如下：

```
mybatis-plus:
  configuration:
    log-impl: org.apache.ibatis.logging.stdout.StdOutImpl
    mapper-locations: classpath:mapper/*Mapper.xml
feign:
  sentinel:
    enabled: true
  client:
    config:
      default:
        readTimeout: 3000
        connectTimeout: 1000
spring:
  cloud:
    sentinel:
      transport:
        dashboard: http://192.168.200.128:8080
  rabbitmq:
    host: 47.93.148.192
    port: 5672
    username: guest
    password: guest
    publisher-confirm-type: correlated
    publisher-returns: true
    listener:
      simple:
        acknowledge-mode: manual #默认情况下消息消费者是自动确认消息的，如
        果要手动确认消息则需要修改确认模式为 manual
        prefetch: 1 # 消费者每次从队列获取的消息数量。此属性当不设置时为：
        轮询分发，设置为 1 为：公平分发
  redis:
    host: localhost
    port: 6379
    database: 0
    timeout: 1800000
    password:
    lettuce:
      pool:
        max-active: 20 #最大连接数
        max-wait: -1 #最大阻塞等待时间(负数表示没限制)
        max-idle: 5 #最大空闲
        min-idle: 0 #最小空闲
  jackson:
    date-format: yyyy-MM-dd HH:mm:ss
    time-zone: GMT+8
```


3, 商品模块配置文件 service-product-dev.yaml

```
server:
  port: 8206
spring:
  datasource:
    type: com.zaxxer.hikari.HikariDataSource
    driver-class-name: com.mysql.jdbc.Driver
    url: jdbc:mysql://localhost:3306/gmall_product?characterEncoding=utf-8&useSSL=false
    username: root
    password: root
    hikari:
      connection-test-query: SELECT 1 # 自动检测连接
      connection-timeout: 60000 #数据库连接超时时间, 默认 30 秒
      idle-timeout: 500000 #空闲连接存活最大时间, 默认 600000 (10 分钟)
      max-lifetime: 540000 #此属性控制池中连接的最长生命周期, 值 0 表示无限生命周期, 默认 1800000 即 30 分钟
      maximum-pool-size: 12 #连接池最大连接数, 默认是 10
      minimum-idle: 10 #最小空闲连接数量
      pool-name: SPHHikariPool # 连接池名称
  minio:
    endpointUrl: http://47.93.148.192:9000
    accessKey: admin
    secreKey: admin123456
    bucketName: gmall
```

说明：其实配置属性还是以前项目的配置属性，只是变化了文件命名规则，配置项不变。

2.5.3 改造 server-gateway

删除之前的配置文件

1, 添加配置文件 bootstrap.properties

```
spring.application.name=server-gateway
spring.profiles.active=dev
spring.cloud.nacos.discovery.server-addr=192.168.200.129:8848
spring.cloud.nacos.config.server-addr=192.168.200.129:8848
spring.cloud.nacos.config.prefix=${spring.application.name}
spring.cloud.nacos.config.file-extension=yaml
```

说明：其他服务配置文件统一提供，资源库获取，上传 nacos

