

SpringCloud

xbZhong

2025-10-08

Contents

MybatisPlus	1
微服务	12
Elasticsearch	33
微服务面试	51

本页 PDF

MybatisPlus

对于 Mybatis 的增强和升级，两者之间是合作的关系

如何使用：

- 自定义的 Mapper 接口要继承 MybatisPlus 提供的 BaseMapper 接口，指定要操作的实体类

```
public interface UserMapper extends BaseMapper<User> {  
}
```

- BaseMapper 里面内置了很多单表 CRUD 的方法，可以直接使用
- MybatisPlus 通过扫描实体类，基于反射获取实体类信息作为数据库表信息

约定：

- 类名驼峰转下划线，大写变小写会作为表名
 - User->user
 - UserInfo->user_info
- 名为 id 的字段作为主键
- 变量名驼峰转下划线，大写变小写作为表的字段名

常见注解

- @TableName：指定表名
- @TableId：用来指定表中的主键字段信息
 - 用 values 指定要替换的主键名称
 - 用 type 指定新增主键的手段，默认是 ASSIGN_ID
 - IdType.AUTO：数据库自增长
 - IdType.INPUT：程序员通过 set 方法输入
 - IdType.ASSIGN_ID：由程序帮我们自动生成
- @TableField：用来指定表中的普通字段信息

- 用 values 指定要替换的字段名称
- 需要使用 @TableField 的场景：
 - 成员变量名和数据库字段名不一致
 - 成员变量名以 is 开头，且是布尔值
 - 成员变量名与数据库关键字冲突，加 ‘包裹’
 - 成员变量不是数据库字段，加上 exist = false 即可

```
@TableName("tb_user")
public class User{
    @TableId(type=IdType.ASSIGN_ID)
    private long id;

    private String name;

    private Boolean isMarried;
    @TableField("`order`")
    private Integer order;
    @TableField(exist=false)
    private String address;
}
```

条件构造器

Mp 支持各种复杂的 where 条件

Wrapper：条件构造器，用于动态构建 SQL 查询条件，动态生成 where 语句!!!

- QueryWrapper：扩展查询相关的功能
 - .select()：里面填入要查询的字段
 - .like()：里面填入列和模糊查询的条件
 - .ge()：填入列和阈值，大于等于的意思
- UpdateWrapper：扩展更新相关的功能，动态生成 where 和 set 语句

例子

```
// 自定义 Mapper 接口
public interface UserMapper extends BaseMapper<User>{

}

@.Autowired
private UserMapper userMapper;

void testQueryMapper(){
    // 1. 构建查询条件
    QueryWrapper<User> wrapper = new QueryWrapper<>()
        .select("id", "username", "info", "balance")
        .like("username", "o")
        .ge("balance", 1000);

    // 2. 查询
}
```

```

List<User> users = userMapper.selectList(wrapper);
users.forEach(user-> System.out.println(user));
}

void testUpdateByQueryWrapper(){
    // 1. 准备要更新的数据
    User user = new User();
    user.setBalance(2000);
    // 2. 要更新的条件
    QueryWrapper<User> wrapper = new QueryWrapper<User>.eq("username","jack");
    // 3. 执行更新
    userMapper.update(user,wrapper);
}

void testUpdateWrapper(){
    List <long> ids = List.of(1L,2L,4L);
    UpdateWrapper<User> wrapper = new UpdateWrapper<User>()
        .setSql("balance = balance - 200") //update 里面 set 后面的
        .in("id",ids);

    // 不需要填更新参数
    userMapper.update(null,wrapper);
}

```

Lambda 例子

- 方法里面的参数填的是 function
 - User::getUsername：返回的是数据库字段名

```

void testLambdaQueryMapper(){
    // 1. 构建查询条件
    LambdaQueryWrapper<User> wrapper = new LambdaQueryWrapper<User>()
        .select(User::getId,User::getUsername,User:: getInfo,User::getBalance) //实例方法的方法引用
        .like(User::getUsername,"o")
        .ge(User::getBalance,1000);

    // 2. 查询
    List<User> users = userMapper.selectList(wrapper);
    users.forEach(user-> System.out.println(user));
}

```

自定义 SQL

利用 Mp 的 Wrapper 来构建复杂的 where 条件，自己写剩下的 SQL 语句

注意：需要在 Mapper 层完成 SQL 语句的组装，很多企业中都是不允许在 Service 层显式编写 SQL 代码

- 在业务层基于 Wrapper 构建 where 条件

```

List <Long> ids = List.of(1L,2L,4L);
int amount = 200;

```

```
// 1. 构建查询条件
LambdaQueryWrapper<User> wrapper = new LambdaQueryWrapper<User>().in(User::getId, ids);

// 2. 自定义 SQL 方法调用
userMapper.updateBalanceByIds(wrapper, amount);
```

2. 在 mapper 方法参数中用 Param 注解声明 wrapper 变量名称，必须是 ew

```
void updateBalanceByIds(@Param("ew") LambdaQueryWrapper<User> wrapper, @Param("amount") int amount);
```

3. 自定义 SQL，并使用 Wrapper 条件

```
<update id="updateBalanceByIds">
    UPDATE tb_user
    SET balance = balance - #{amount}
    ${ew.customSqlSegment} <!--由 Mp 帮我们自动解析，自动拼接 SQL 语句-->
</update>
```

Service 接口

接口名称：IService

- save
- remove
- update
- get

对于简单的增删改查逻辑可以用来替代 Service 层和 Mapper 层，是对 service 层的增强

使用流程：

- 自定义服务接口要继承 IService 接口
- 自定义类要继承 ServiceImpl 类，并且要实现自定义接口
- 使用自定义服务接口进行依赖注入并进行使用

```
// 自定义类
// 需要指定数据访问层的 UserMapper 和要操作的实体类 User
@Service
public class UserServiceImpl extends ServiceImpl<UserMapper, User> implements UserService{

}

// 自定义接口
public interface UserService extends IService<User>{

}

// 使用
@Autowired
private UserService userService;
// 调用 userService 来进行操作
```

几种常见实体

- DTO 实体：用于不同系统或层之间传输数据的中间对象

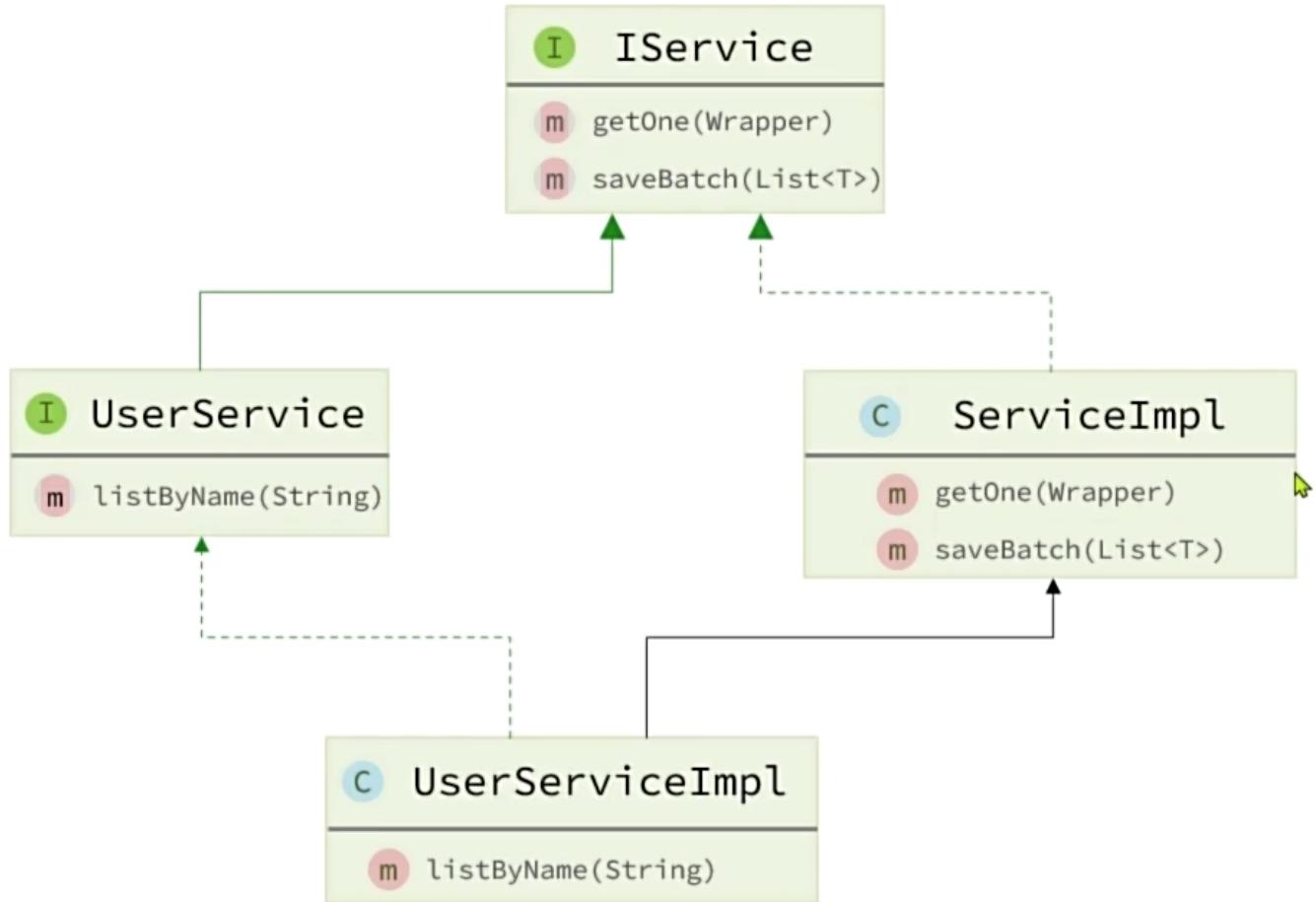


Figure 1: image-20250831160832309

- VO 实体：面向前端或者接口的数据封装对象
- PO 实体：与数据表直接映射的 java 对象

使用构造器注入

- 使用 @RequiredArgsConstructor 注解进行构造方法自动生成
- 使用 final 修饰接口

```
@RequestMapping("/users")
@RequiredArgsConstructor
public class UserController{
    private final IUserService userService;

    @DeleteMapping("{id}")
    public void deleteUserById(@PathVariable("id") Long id){
        /*
         方法体
        */
    }
}
```

IService 的 Lambda 查询 需要做条件查询等复杂操作的话可以用这个方法

由 IService 类提供

- lambdaQuery(): 可以做查询操作
 - 判断条件里面可以传一个 condition，用来判断字段是否为空
 - 最后可以调用 .list()、.one() 等方法来实现各种功能
 - 相对应的，其也有 query() 方法，可以构造查询条件
- LambdaUpdate(): 可以做更新操作
 - 也可以用 .update() 做更新操作
 - 最后使用 .update() 进行更新操作

```
// 查询
List<User> list = lambdaQuery()
    .like(name != null, User::getName, name)
    .eq(status != null, User::getStatus, status)
    .ge(minBalance != null, User::getBalance, minBalance)
    .le(maxBalance != null, User::getBalance, maxBalance)
    .list();

// 更新
LambdaUpdate()
    .set(User::getBalance, remainBalance)
    .set(remainBalance == 0, User::getStatus, 2)
    .eq(User::getId, id)
    .eq(User::getBalance, user.getBalance()) // 乐观锁
    .update();
```

IService 批量新增

- Mp 的批量新增，基于预编译实现
 - 如果要提交 1k 条数据，会生成 1k 条 sql，然后同时提交
- 基于 Mysql 驱动实现
 - 开启 `rewriteBatchedStatements=true` 参数
 - 将多条数据的提交重写成一条 sql 语句，**性能最好**

代码生成器

使用 **MybatisPlus** 插件可以实现代码自动生成

- 配置好数据库连接信息
- 配置好实体类、持久化层、服务层、控制层的存储路径
- 配置作者信息
- 配置数据库表明要去除的前缀

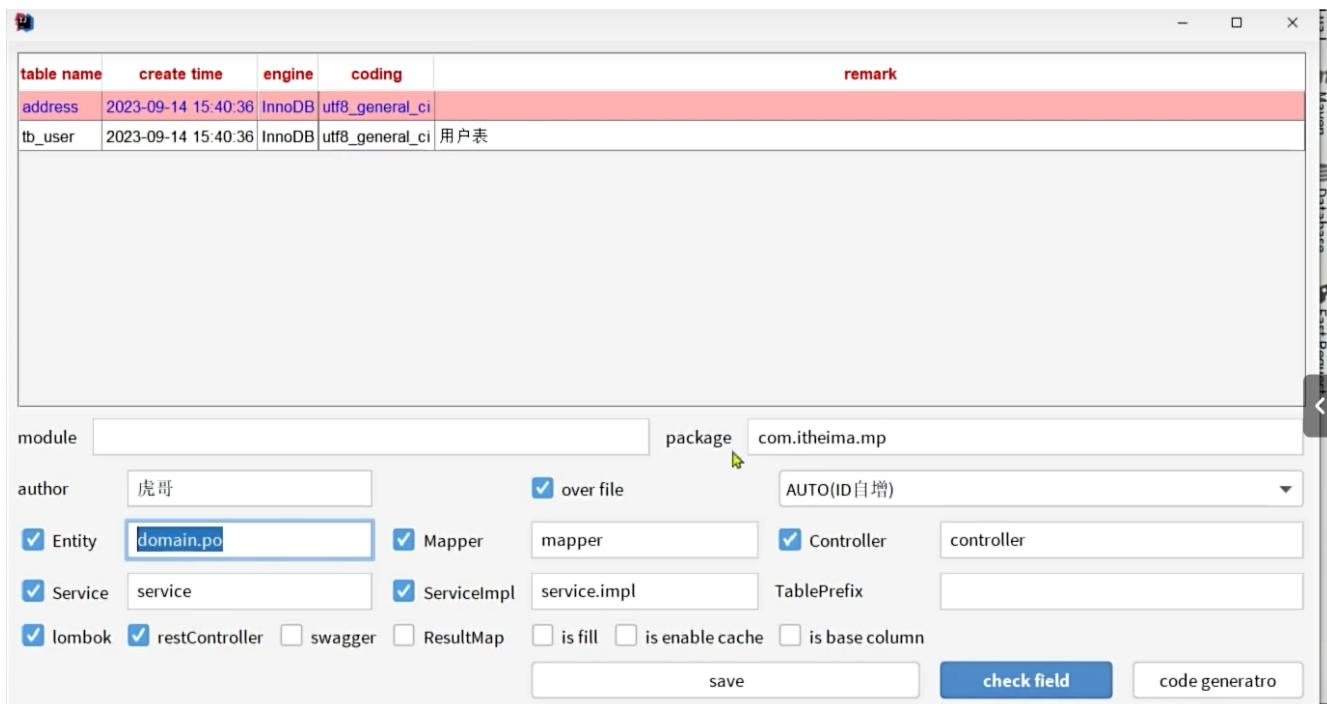


Figure 2: image-20250927144021258

静态工具

主要是为了解决循环依赖的问题

需要传入实体类的字节码

逻辑删除

MP 提供了逻辑删除的功能，**无需改变方法调用的方式**

我们要做的就是在 `application.yaml` 文件中配置逻辑删除的字段名称和值即可

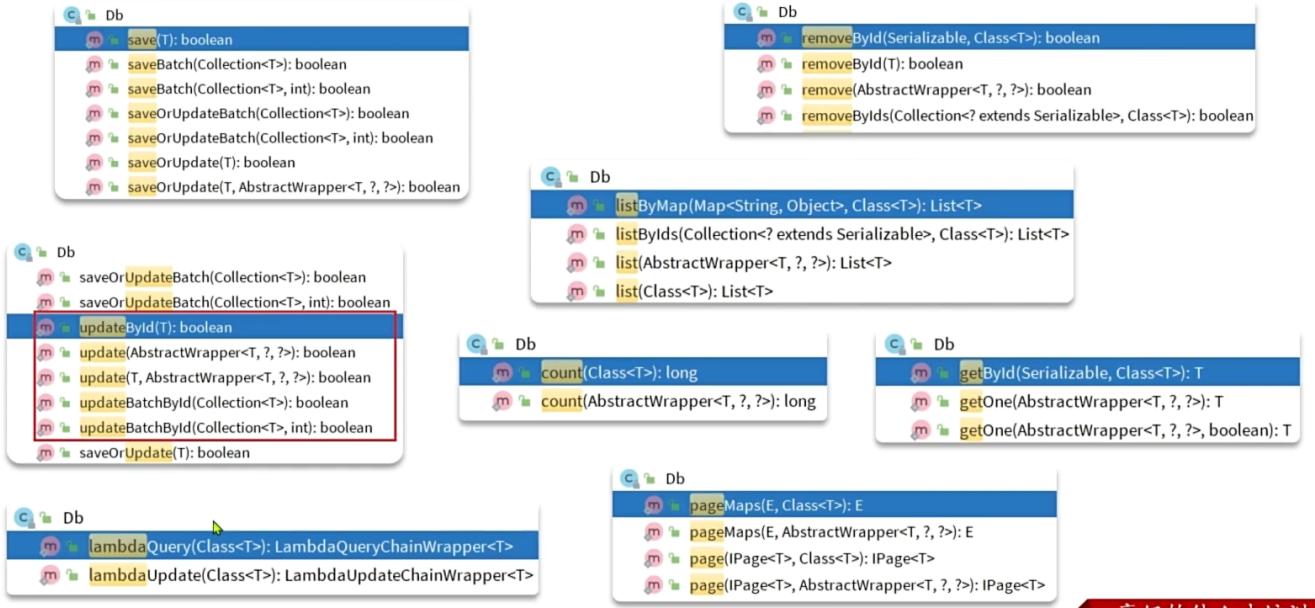


Figure 3: image-20250927144350529

```
mybatis-plus:
global-config:
db-config:
  logic-delete-field: flag # 全局逻辑删除的实体字段名
  logic-delete-value: 1     # 逻辑已删除的值（默认 1）
  logic-not-delete-value: 0 # 逻辑未删除的值（默认 0）
```

枚举处理器

User 类中有一个用户状态字段

```
private Integer status;
```

因为状态字段的值是有限的，可以用一个枚举类型去声明，增强代码可读性，但是枚举类型和数据库中的 Integer 类型相互转换会有问题

枚举类

- 使用 @EnumValue 告诉 MP 要把哪个字段存入数据库
- 使用 @JsonValue 告诉 SpringMVC 返回这个枚举类给前端的时候要返回哪个字段

```
@Getter
public enum UserStatus{
    NORMAL(1, "正常"),
    FREEZE(2, "冻结")
;
    @EnumValue
    @JsonValue
    private final int value;
```

```
private final String desc;

UserStatus(int value, String desc){
    this.value = value;
    this.desc = desc;
}
}
```

同时还要在配置文件添加配置

```
mybatis-plus:
  configuration:
    default-enum-type-handler: com.baomidou.mybatisplus.core.handlers.MybatisEnumTypeHandler
```

JSON 处理器

可以使存储在数据库的 json 字符串和 java 对象相互转换

使用方法

- 在大类上加上 @TableName 注解，并开启自动结果映射 autoResultMap = true
- 在存储数据库 json 字符串的字段上加上 @TableField 注解，并指定 JSON 处理器的字节码 typeHandler = JacksonTypeHandler.class

```
@Data
@TableName(value = "user", autoResultMap = true)
public class User{
    private long id;
    private String name;

    @TableField(typeHandler = JacksonTypeHandler.class)
    private UserInfo info;
}

@Data
public class UserInfo{
    private Integer age;
    private String intro;
    private String gender;
}
```

分页插件

首先要在配置类中注册 MP 的核心插件并添加分页插件：

```
@Configuration
public class MybatisConfig{

    @Bean
    public MybatisPlusInterceptor mybatisPlusInterceptor(){
        // 初始化核心插件
    }
}
```

```

MybatisPlusInterceptor interceptor = new MybatisPlusInterceptor();
// 添加分页插件
PaginationInnerInterceptor pageInterceptor = new PaginationInnerInterceptor(DbType.MYSQL);
pageInterceptor.setMaxLimit(1000L); // 设置分页上限
interceptor.addInnerInterceptor(pageInterceptor);
return interceptor;
}
}

```

分页查询流程：

- 准备分页参数：页码，每页查几条数据
- 封装成 **Page** 对象
 - MP 里用 Page 进行分页查询
- 添加排序参数.addOrder()
 - new OrderItem(" 排序字段", 排序方式)
 - true 为升序, false 为降序
- .getTotal(): 总条数
- .getPages(): 总页数
- .getRecords(): 分页查询的数据

```

@Test
void testPageQuery() {
    // 1. 准备分页参数
    int pageNo = 1, pageSize = 5;

    // 1.1 创建分页对象
    Page<User> page = Page.of(pageNo, pageSize);

    // 1.2 设置排序参数 (按 balance 字段降序排列)
    page.addOrder(new OrderItem("balance", false));

    // 1.3 执行分页查询
    Page<User> p = userService.page(page);

    // 2. 输出总记录数
    System.out.println("total=" + p.getTotal());

    // 3. 输出总页数
    System.out.println("pages=" + p.getPages());

    // 4. 输出分页数据
    List<User> records = p.getRecords();
    records.forEach(System.out::println);
}

```

通用分页实体 查询类可以通过**继承**实现

```

@Data
public class PageQuery{
    private Integer pageNo;
    private Integer pageSize;
    private String sortBy;
    private Boolean isAsc;
}

@Data
public class UserQuery extends PageQuery{
    private String name;
    private Integer status;
}

```

分页结果

```

public class PageDTO<T>{
    // 总条数
    private Integer total;
    // 总页数
    private Integer pages;
    // 分页结果
    private List<T> list;
}

```

将 PageQuery 对象转为 MP 中的 Page 对象

```

@Data
public class PageQuery{
    private Integer pageNo = 1;
    private Integer pageSize = 5;
    private String sortBy;
    private Boolean isAsc = true;

    public <T> Page<T> toMyPage(OrderItem ...items){
        // 分页条件
        Page<T> page = Page.of(pageNo,pageSize);
        // 查询条件
        if(StrUtil.isNotBlank(sortBy)){
            page.addOrder(new OrderItem(sortBy,isAsc));
        }else if(items != null){
            page.addOrder(items);
        }
        return page;
    }
}

```

将 Page 结果转换为 PageDTO 结果

```
public class PageDTO<T>{
    // 总条数
    private Integer total;
    // 总页数
    private Integer pages;
    // 分页结果
    private List<T> list;

    public static <P0,V0> PageDTO<V0> of(Page<P0> page,Function<P0,V0> converter){
        PageDTO<V0> dto = new PageDTO<>();
        // 总条数
        dto.setTotal(page.getTotal());
        // 总页数
        dto.setPages(page.getPages());
        // 当前页数据
        List<P0> records = page.getRecords();

        if(CollectionUtil.isEmpty(records)){
            dto.setList(Collections.emptyList());
            return dto;
        }
        // 拷贝 P0 为 V0
        dto.setList(records.stream().map(convertor).collect(Collectors.toList()));
        return dto;
    }
}
```

微服务

服务架构：

- 独立 project
- Maven 聚合

服务调用

Spring 提供了一个 RestTemplate 工具，可以方便的实现 Http 请求的发送，使用步骤如下：

- 注入 RestTemplate 到 Spring 容器

```
package com.hmall.cart;

import org.mybatis.spring.annotation.MapperScan;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
```

```

import org.springframework.web.client.RestTemplate;

@MapperScan("com.hmall.cart.mapper")
@SpringBootApplication
public class CartApplication {
    public static void main(String[] args) {
        SpringApplication.run(CartApplication.class, args);
    }

    @Bean
    public RestTemplate restTemplate() {
        return new RestTemplate();
    }
}

```

- 发起远程调用

```

ResponseEntity<List<ItemDTO>> response = restTemplate.exchange(
    "http://localhost:8081/items?ids={ids}",
    HttpMethod.GET,
    null,
    new ParameterizedTypeReference<List<ItemDTO>>() {
    },
    Map.of("ids", CollUtil.join(itemIds, ","))
);
if(!response.getStatusCode().is2xxSuccessful()){
    return;
}

List<ItemDTO> items = response.getBody();

```

服务治理

注册中心原理

- 服务提供者会去注册中心注册自己的服务信息
 - 服务提供者还需要向注册中心进行心跳续约，告诉服务中心自己还能工作
- 服务调用者会订阅注册中心的信息
 - 当服务提供者宕机，无法进行心跳续约，注册中心会进行信息的推送变更

Nacos 注册中心 是注册中心组件，阿里巴巴的产品

服务注册

- 使用 docker 部署，启动后可以输入 ip:8848/nacos 进行管理页面的访问，账号密码默认都是 nacos
- 需要在项目里的 xml 文件引入依赖并在 yaml 进行服务信息声明和

```

<!--nacos 服务注册发现-->
<dependency>

```

心原理

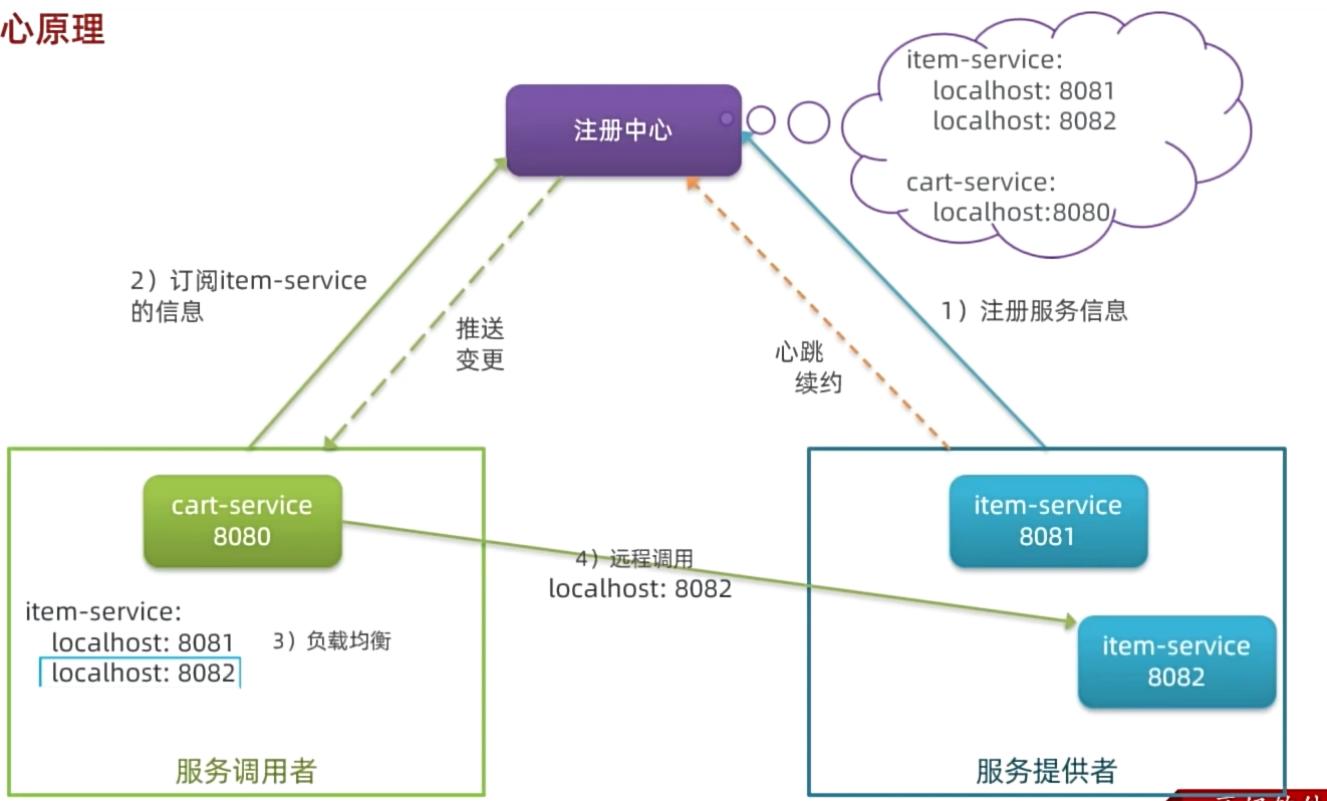


Figure 4: image-20251001133940462

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

```
spring:
  application:
    name: item-service # 服务名称
  cloud:
    nacos:
      server-addr: 192.168.150.101:8848 # nacos 地址
```

服务发现

消费者需要连接 nacos 拉取和订阅服务，步骤如下

- 引入 nacos 依赖
- 配置 nacos 地址
- 服务发现
 - 使用 `DiscoveryClient` (Spring 定义的标准) 进行服务列表的拉取

```
private final DiscoveryClient discoveryClient;

private void handleCartItems(List<CartVO> vos){
```

```

// 1. 根据服务名称，拉取服务的实例列表
List<ServiceInstance> instances = discoveryClient.getInstances("item-service");
// 2. 负载均衡，挑选一个实例
ServiceInstance instance = instances.get(RandomUtil.randomUUID(instances.size()));
// 3. 获取实例的 ip 和端口
URI uri = instance.getUri();
// 4. 略
}

```

OpenFeign 是一个声明式的 http 客户端，基于 SpringMVC 的常见注解帮我们优雅的实现 http 请求的发送
使用步骤：

- 引入依赖
 - 使用的负载均衡是 **Loadbalancer**

```

<!--openFeign-->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-openfeign</artifactId>
</dependency>
<!--负载均衡器-->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-loadbalancer</artifactId>
</dependency>

```

- 通过 `@EnableFeignClients` 注解，启用 OpenFeign 功能

```

@MapperScan("com.hmall.cart.mapper")
@EnableFeignClients
@SpringBootApplication
public class CartApplication {
    public static void main(String[] args) {
        SpringApplication.run(CartApplication.class, args);
    }
}

```

- 编写 FeignClient

```

// 声明服务名
@FeignClient("item-service")
public interface ItemClient {
    // 声明 URL
    @GetMapping("/items")
    List<ItemDTO> queryItemByIds(@RequestParam("ids") Collection<Long> ids);
}

```

- 使用 FeignClient，实现远程调用

```
List<ItemDTO> items = itemClient.queryItemByIds(List.of(1,2,3));
```

连接池 OpenFeign 对 Http 进行优雅封装，不过我们可以选择喜欢的框架去发起 Http 请求：

- HttpURLConnection：默认实现，不支持连接池
- Apache HttpClient：支持连接池
- OKHttp：支持连接池

使用连接池的话性能好一点!!!

整合 OKHttp 的步骤如下

- 引入依赖

```
<!--OK http 的依赖 -->
<dependency>
    <groupId>io.github.openfeign</groupId>
    <artifactId>feign-okhttp</artifactId>
</dependency>
```

- 开启连接池

```
feign:
  okhttp:
    enabled: true # 开启 OKHttp 功能
```

最佳实践 我们可以对微服务的服务治理这个功能进行划分，将其划分为一个新的模块

我们在新模块定义的 FeignClient 不在微服务模块的 SpringBootApplication 扫描范围内，会导致 FeignClient 无法使用，有两种方法解决：

- 指定 FeignClient 所在包

```
@EnableFeignClients(basePackages="com.hmall.api.clients")
```

- 指定 FeignClient 字节码

```
@EnableFeignClients(clients = UserClient.class)
```

日志输出 OpenFeign 只会在 FeignClient 所在包的日志级别为 DEBUG 时，才会输出日志。而且其日志级别有 4 级：

- **NONE**：不记录任何日志信息，这是默认值。
- **BASIC**：仅记录请求的方法，URL 以及响应状态码和执行时间。
- **HEADERS**：在 BASIC 的基础上，额外记录了请求和响应的头信息。
- **FULL**：记录所有请求和响应的明细，包括头信息、请求体、元数据。

由于 Feign 默认的日志级别就是 NONE，所以默认我们看不到请求日志

要自定义日志级别需要声明一个类型为 `Logger.Level` 的 Bean，在其中定义日志级别

```
public class DefaultFeignConfig{
    @Bean
    public Logger.Level feignLogLevel(){
```

```
        return Logger.Level.FULL;  
    }  
}
```

- 局部配置，在 @FeignClient 注解中声明

```
@FeignClient(value = "item-service", configuration = DefaultFeignConfig.class)
```

- 全局配置，在 @EnableFeignClients 注解中声明

```
@EnableFeignClients(defaultConfiguration = DefaultFeignConfig.class)
```

网关及配置管理

网关：网络的关口，负责请求的路由、转发、身份校验

- 网关还可以去注册中心拉取服务信息
- 网关本身也是一个微服务

在 SpringCloud 中网关的实现分为两种：

- SpringCloudGateWay：响应式编程
- Netflix Zuul：阻塞式编程

ITG

网关：就是网络的关口，负责请求的路由、转发、身份校验。

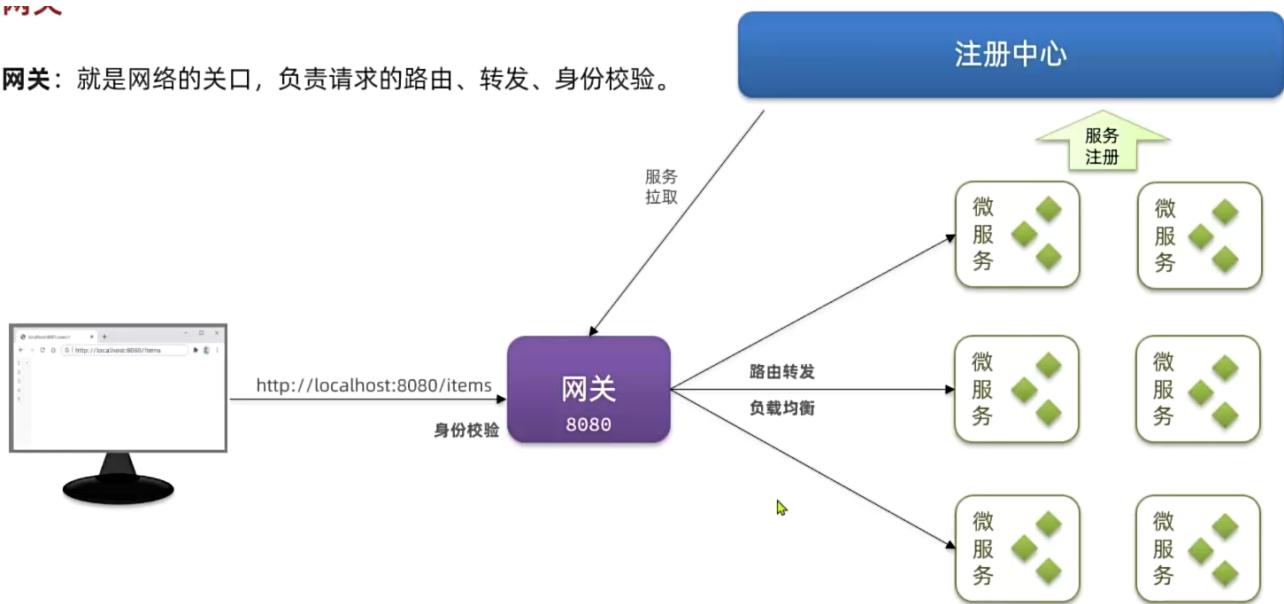


Figure 5: image-20251001152734999

配置路由规则 需要在 application.yaml 下进行配置

- 路由规则 id
- 路由目标服务，就是转发的目标路径
- 路由断言，定义匹配规则

```

spring:
  application:
    name: gateway
  cloud:
    nacos:
      server-addr: 192.168.150.101:8848
  gateway:
    routes:
      - id: item # 路由规则 id, 自定义, 唯一
        uri: lb://item-service # 路由的目标服务, lb 代表负载均衡, 会从注册中心拉取服务列表
        predicates: # 路由断言, 判断当前请求是否符合当前规则, 符合则路由到目标服务
          - Path=/items/**,/search/** # 这里是以请求路径作为判断规则
      - id: cart
        uri: lb://cart-service
        predicates:
          - Path=/carts/**

```

路由属性 网关路由对应的 Java 类型是 **RouteDefinition**, 其中常见属性有

- **id**: 路由唯一标识
- **uri**: 路由目标地址
- **predicates**: 路由断言, 判断请求是否符合当前路由
 - 有 12 种不同的路由断言
- **filters**: 路由过滤器, 对请求或响应做特殊处理
 - 有 33 种路由过滤器, 每种过滤器都有独特的作用
 - 如果想要对所有路由生效, 可以在和 routes 同级的目录下对 default-filters 进行配置

网关请求处理流程

- **路由映射器**: 根据路由断言找到匹配的路由, 把请求交给请求处理器处理
- **请求处理器**: 是一个过滤器处理器, 它会加载网关配置中的多个过滤器, 放入集合中并进行排序, **形成过滤器链**, 然后依次执行过滤器
 - PRE 过滤器: 请求转发到微服务之前执行, **顺序执行**
 - POST 过滤器: 请求转发到微服务之后执行, **倒序执行**
 - 最后会有个 **Netty 路由过滤器**, 负责将请求转发到微服务

要进行登录校验的话, 需要在网关自定义**过滤器**, 并且把用户信息保存到**请求头**, 在微服务之间传递用户信息, 也需要**把用户信息保存到请求头**

- 网关转发请求到微服务是基于 Http, 微服务之间传递信息是基于 OpenFeign

网关登录校验

自定义过滤器 网关过滤器:

- **GatewayFilter**: 路由过滤器, 作用于任意指定的路由, 默认不生效
- **GlobalFilter**: 全局过滤器, 作用范围是所有路由, 声明后自动生效

自定义全局过滤器实现步骤

- 实现 **GlobalFilter** 接口, 重写 filter 方法

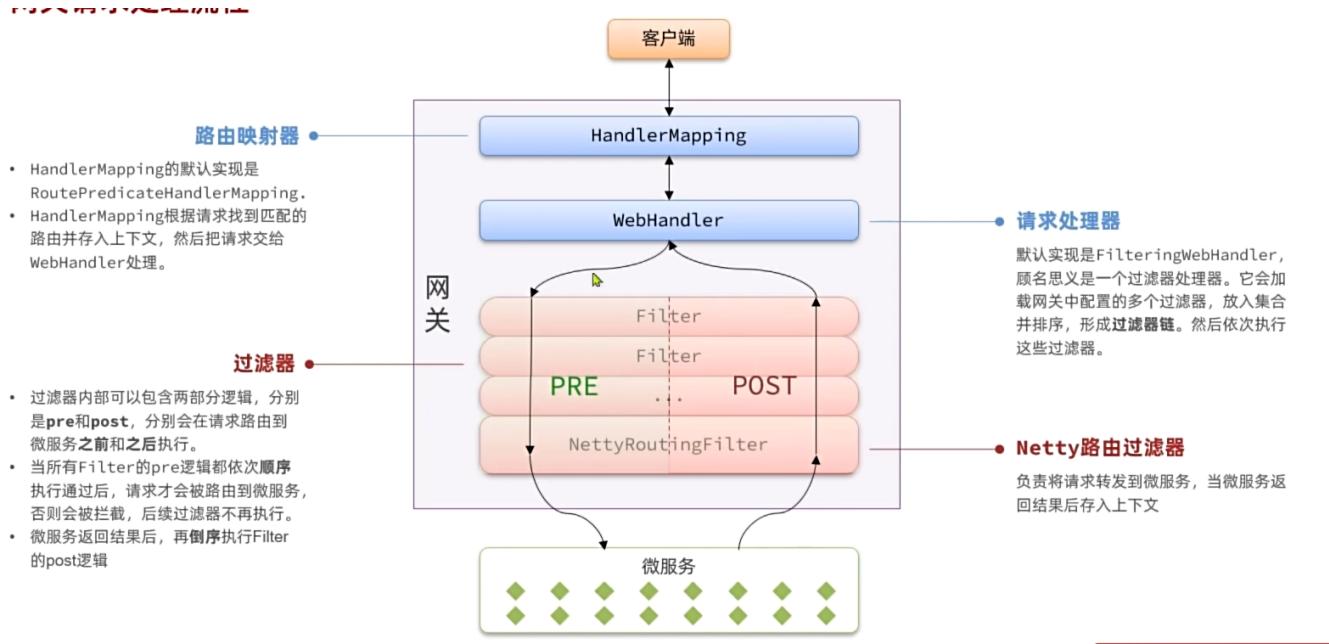


Figure 6: image-20251002135230563

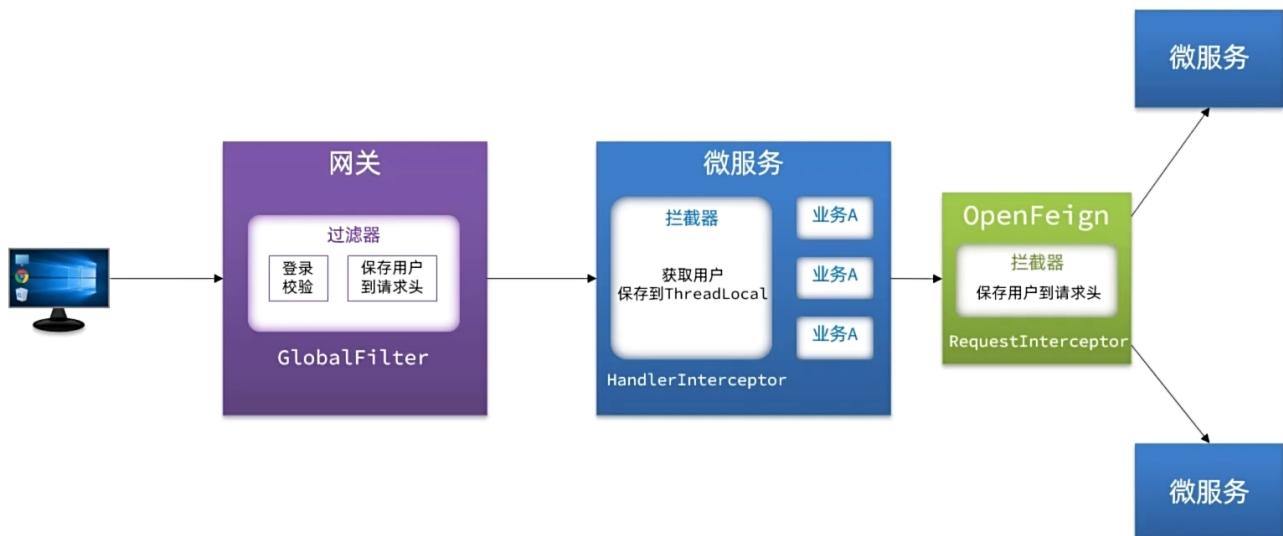


Figure 7: image-20251002152441612

- 实现 Ordered 接口，重写 getOrder 方法，实现过滤器排序
 - 值越小优先级越高
 - Netty 路由过滤器的值默认是 int 的最大值
- 注意：
 - 可以使用 AntPathMatcher 进行路径匹配
 - exchange 里面包含了上下文信息，可以直接获取并进行信息传递，也可以终止路由转发

示例代码如下：

```

@Override
public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain) {

    // 1. 获取 request
    ServerHttpRequest request = exchange.getRequest();
    // 2. 判断是否需要做登录拦截
    if(isExclude(request.getPath().toString())){
        return chain.filter(exchange);
    }
    // 3. 获取登录 token
    String token = null;
    List<String> headers = request.getHeaders().get("authorization");
    if(headers != null && !headers.isEmpty()){
        token = headers.get(0);
    }
    // 4. 校验并解析 token
    Long userId = null;
    try{
        userId = jwtTool.parseToken(token);
    }catch (UnauthorizedException e){
        // 设置状态码为 401
        ServerHttpResponse response = exchange.getResponse();
        response.setStatusCode(HttpStatus.UNAUTHORIZED);
        return response.setComplete();
    }

    // TODO 5. 传递用户信息
    System.out.println("userId = " + userId);
    // 6. 放行
    return chain.filter(exchange);
}

```

网关传递用户信息给微服务

- 微服务定义拦截器，从请求头获取用户信息保存到 ThreadLocal 里面
- 在自定义拦截器中，可以使用 exchange.mutate() 对转发到微服务的请求进行修改
- 可以在 common 模块定义拦截器，保证每一个微服务都有自己的 ThreadLocal
 - 网关不是基于 SpringMvc 的，他是一个**非阻塞式的模块**，但网关引用了 common 模块，导致拦截器配置类的 WebMvcConfigurer 被网关引用，从而报错

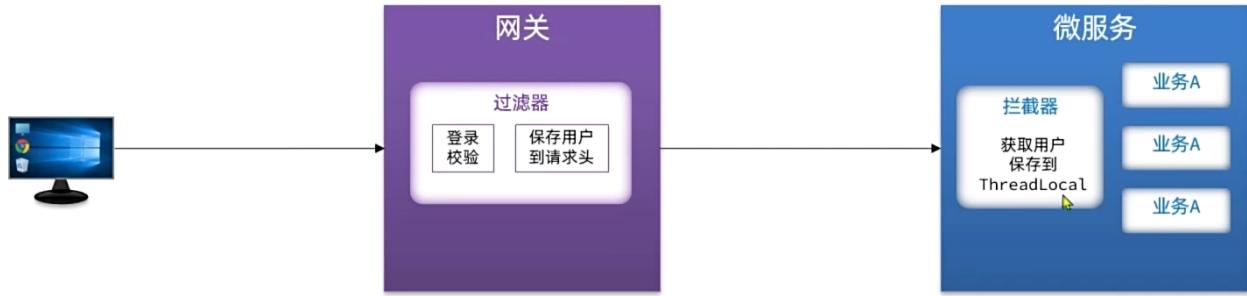


Figure 8: image-20251002143457177

```
exchange.mutate() // mutate就是对下游请求做更改
    .request(builder -> builder.header("user-info", userInfo))
    .build();
```

Figure 9: image-20251002143735906

- 因此需要使用 `@ConditionalOnClass` 注解，让拦截器配置类在网关里不生效，在其它微服务中生效

OpenFeign 传递用户信息 OpenFeign 中提供了一个拦截器接口 `RequestInterceptor`，所有由 OpenFeign 发起的请求发出前都会先调用拦截器处理请求

- 其中的 `RequestTemplate` 类提供了一些方法可以让我们修改请求头
- 这个拦截器接口需要在 **配置类** 使用 Bean 进行注册
- 配置类需要进行声明
 - 可在 `@FeignClient` 注解或者是在 `@EnableFeignClients` 注解中声明

配置管理 问题：微服务重复配置过多，维护成本高

- 可以实现一个**配置管理服务**，微服务启动时从配置管理服务读取配置
- 配置管理服务**可以监听配置的变更，当配置更新后可以把配置进行推送

配置共享 流程图如下：

使用 nacos 的配置列表进行配置文件的增加

- 可以使用 `${hm.db.port:3306}` 进行端口的声明，默认值为 3306

引入依赖、定义 `bootstrap.yaml`，对微服务拉取 nacos 共享配置进行配置

- 引入依赖

- 微服务重复配置过多，维护成本高
- 业务配置经常变动，每次修改都要重启服务
- 网关路由配置写死，如果变更要重启网关

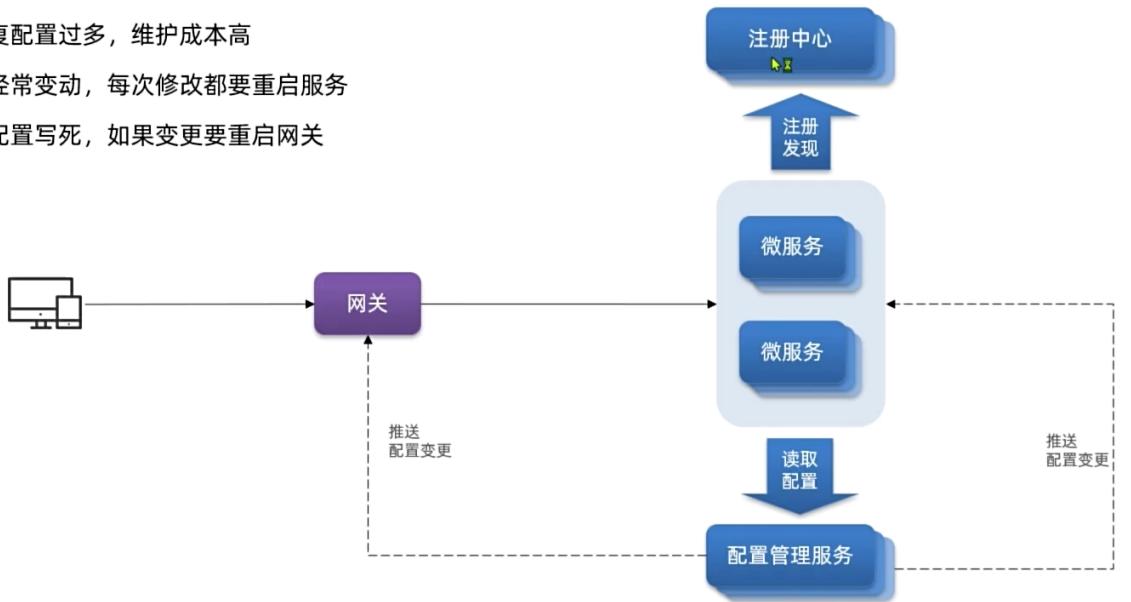


Figure 10: image-20251002153401193

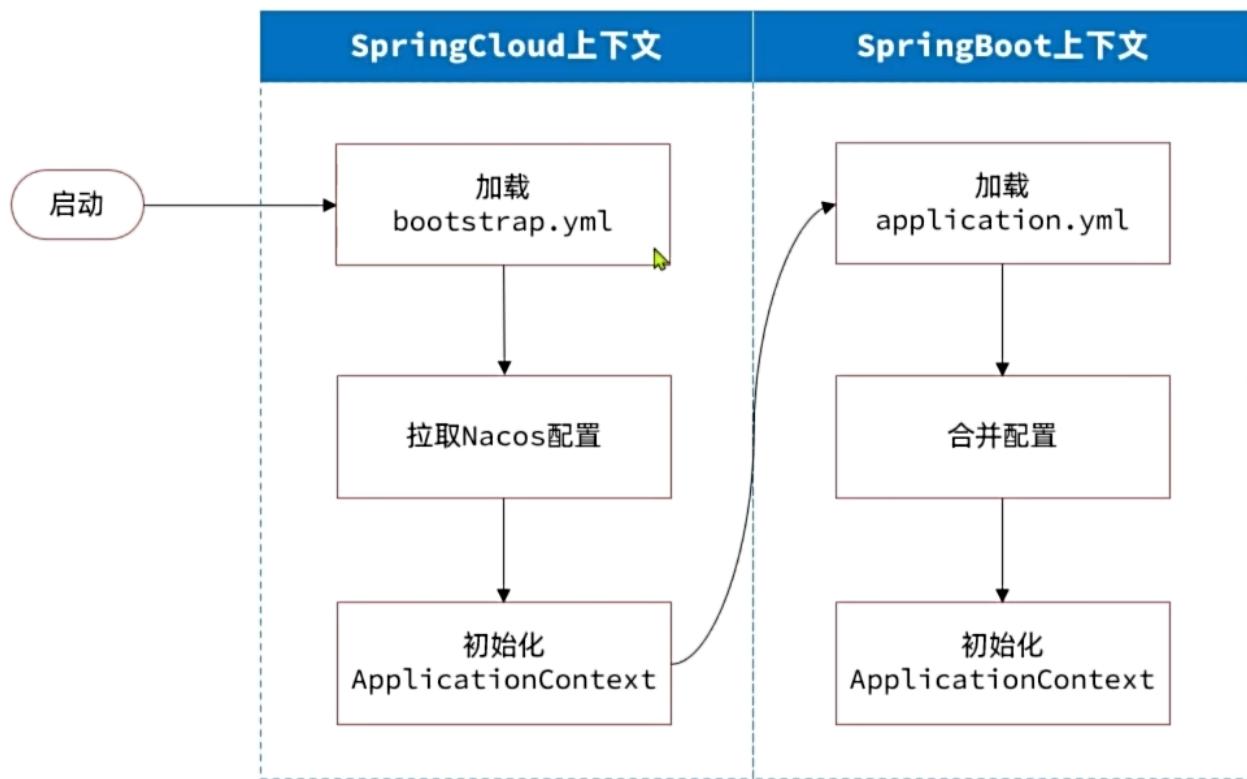


Figure 11: image-20251002154645476

```

<!--nacos 配置管理-->
<dependency>
    <groupId>com.alibaba.cloud</groupId>
    <artifactId>spring-cloud-starter-alibaba-nacos-config</artifactId>
</dependency>
<!--读取 bootstrap 文件-->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-bootstrap</artifactId>
</dependency>

```

- 创建 bootstrap.yaml 文件，下面是核心配置

```

config:
  file-extension: yaml # 文件后缀名
  shared-configs: # 共享配置
    - dataId: shared-jdbc.yaml # 共享 mybatis 配置
    - dataId: shared-log.yaml # 共享日志配置
    - dataId: shared-swagger.yaml # 共享日志配置

```

配置热更新 也就是当修改配置文件中的配置时，微服务**无需重启**即可使配置生效

前提条件：

- nacos 要有一个与微服务有关的配置文件
 - profile 可省略，那么就是对所有环境生效



Figure 12: image-20251003102041072

- 微服务中要以**特定方式**读取需要热更新的配置属性

动态路由 要实现**动态路由**首先要将路由配置保存到 Nacos，当 Nacos 中的路由配置变更时，推送最新配置到网关，实现更新网关的路由信息

步骤如下：

- 项目启动时先获取 Nacos 配置并添加监听器
- 当 Nacos 配置更新时会推送最新配置，并调用监听器的回调函数更新路由表
 - 查阅源码可知，可以使用 NacosConfigManager 进行依赖注入
- 监听到路由信息后，可以利用 RouteDefinitionWriter 更新路由表
 - 之前使用 yaml 配置路由，最后都是被 RouteDefinition 读取，配置信息都在这个类里面
 - 现在 Nacos 会推送更新的配置信息，以**字符串**的形式返回给我们，我们在 Nacos 使用 json 形式进行存储，可以很方便转换为 ‘RouteDefinition 对象’

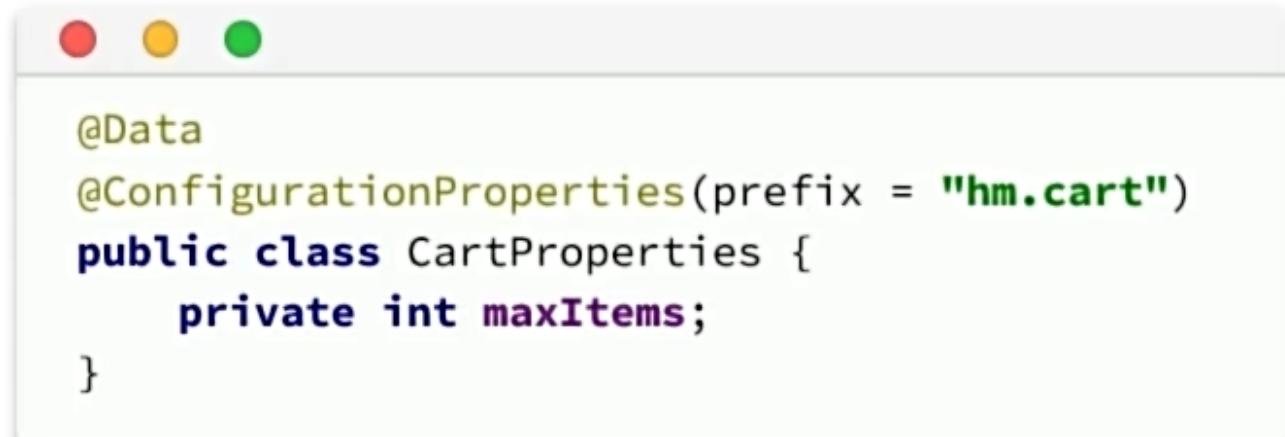


Figure 13: image-20251003102425170

```
private final NacosConfigManager nacosConfigManager;  
  
public void initRouteConfigListener() throws NacosException {  
    // 1. 注册监听器并首次拉取配置  
    String configInfo = nacosConfigManager.getConfigService()  
        .getConfigAndSignListener(dataId, group, 5000, new Listener() {  
            @Override  
            public Executor getExecutor() {  
                return null;  
            }  
            @Override  
            public void receiveConfigInfo(String configInfo) {  
                // TODO 监听到配置变更，更新一次配置  
            }  
        });  
    // TODO 2. 首次启动时，更新一次配置  
}
```

Figure 14: image-20251003104452368

- `RouteDefinitionWriter` 的 `save` 和 `delete` 方法需要传入 `Mono` 容器封装的参数，并且最后做`.subscribe()`订阅



```

/**
 * @author Spencer Gibb
 */
public interface RouteDefinitionWriter {
    /**
     * 更新路由到路由表，如果路由id重复，则会覆盖旧的路由
     */
    Mono<Void> save(Mono<RouteDefinition> route);
    /**
     * 根据路由id删除某个路由
     */
    Mono<Void> delete(Mono<String> routeId);
}

```

Figure 15: image-20251003105712632

服务保护

雪崩问题 微服务调用链路中的某个服务故障，导致整个链路中的所有微服务都不可用，这就是雪崩

解决方案：

- **请求限流**：限制访问微服务的请求的并发量，避免服务因流量激增出现故障
- **线程隔离**：也叫做舱壁模式，通过**限定每个业务能使用的线程数量**而将故障业务隔离，避免故障扩散
- **服务熔断**：由断路器统计请求的异常比例或者慢调用比例，如果超出阈值则**熔断该业务**，则拦截该接口的请求，避免无效资源浪费
 - 熔断期间，所有请求快速失败，全部走 **fallback** 逻辑

Sentinel 是阿里巴巴开源的一款**微服务流量控制组件**

簇点链路：

- 单机调用链路，是一次请求进入服务后经过的每一个被 **Sentinel** 监控的资源链
 - 默认 **Sentinel** 会监控 SpringMVC 的每一个接口（HTTP）
 - 限流、熔断都是**针对簇点链路中的资源设置的**
 - Restful 风格的 API 请求路径一般都相同，这会导致**簇点资源名称重复**
 - 因此要修改配置，把**请求方式 + 请求路径**作为簇点资源名称，在 `yaml` 文件配置
- ```
http-method-specify: true
```

## 请求限流

在簇点链路后面点击**流控按钮**，可对其做限流配置

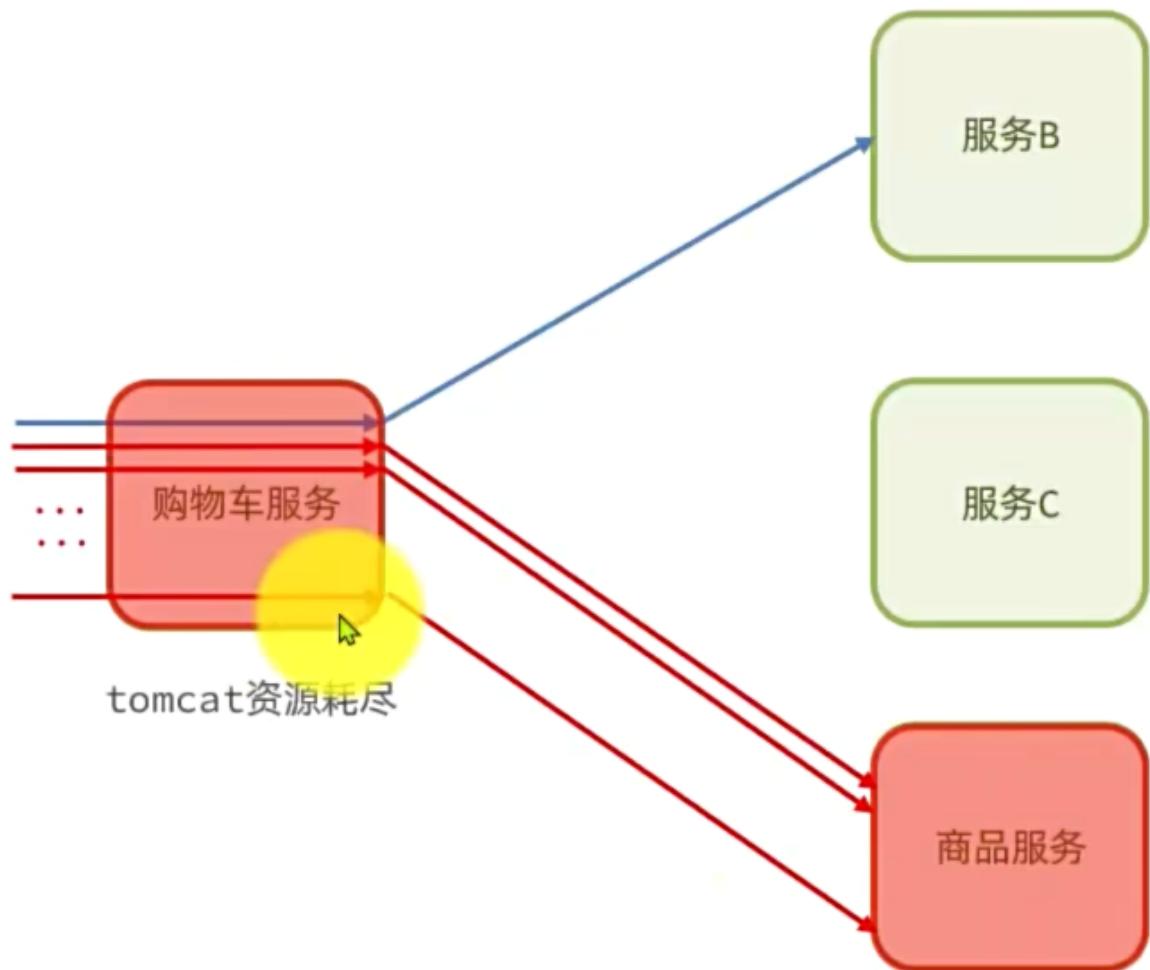


Figure 16: image-20251003113010570

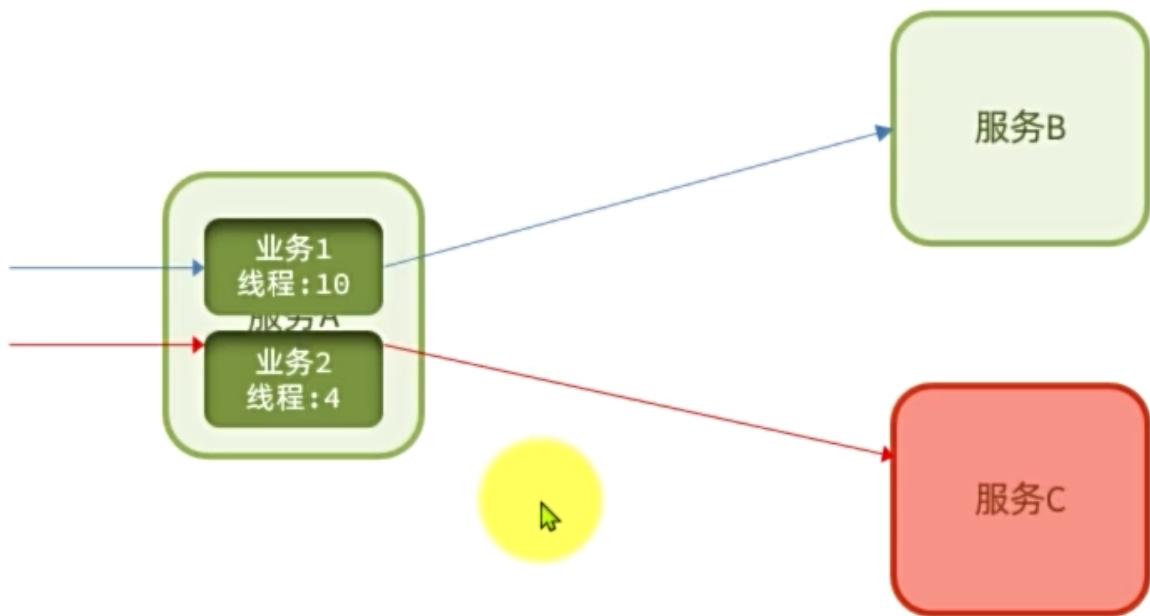


Figure 17: image-20251003114504227

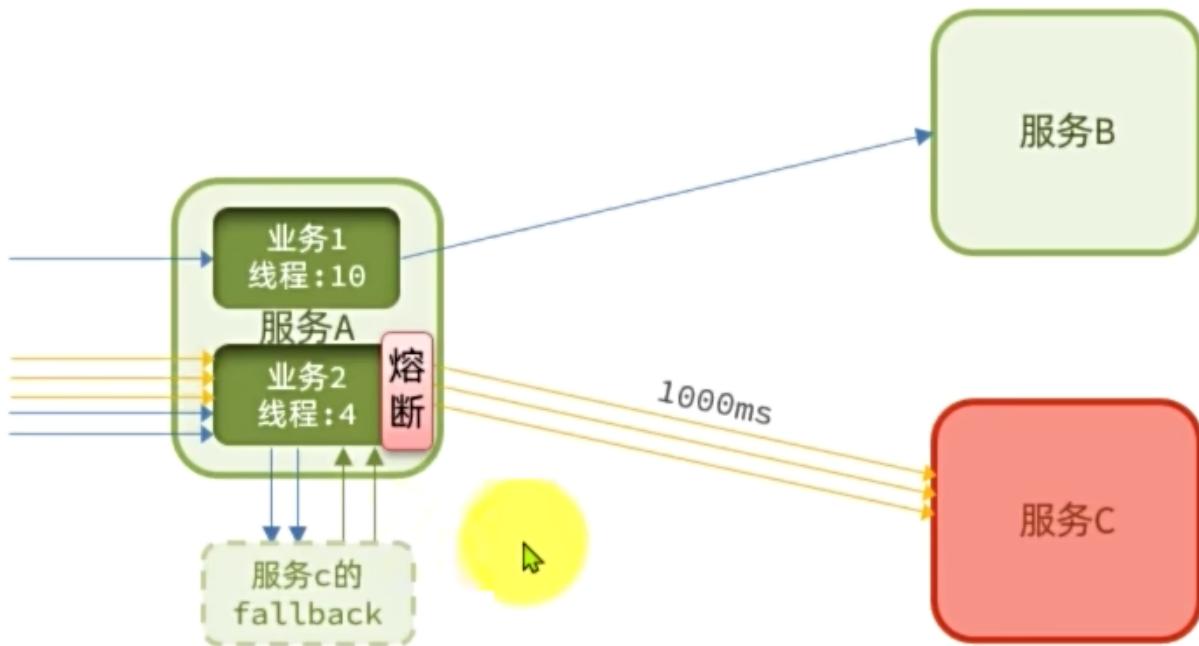


Figure 18: image-20251003115000482

## 线程隔离

在簇点链路后面点击**流控按钮**，可对其做线程隔离配置

## Fallback

- 将 FeignClient 作为 Sentinel 的簇点资源，需要在 yaml 进行配置

```
feign:
 sentinel:
 enabled: true
```

- Fallback 的配置方式：
  - FallbackClass，无法对远程调用的异常做处理
  - FallbackFactory，可以对**远程调用**的异常做处理
    - 自定义类**，实现 FallbackFactory 接口，泛型指定为对应的 Client 接口
    - 重写 create 方法，重新 new 一个 Client 对象，并实现里面的方法
    - 将自定义类注册为一个 Bean
    - 在对应的 Client 接口使用自定义类，在 @FeignClient 的 fallbackFactory 字段添加**自定义类的 class 文件**

## 服务熔断

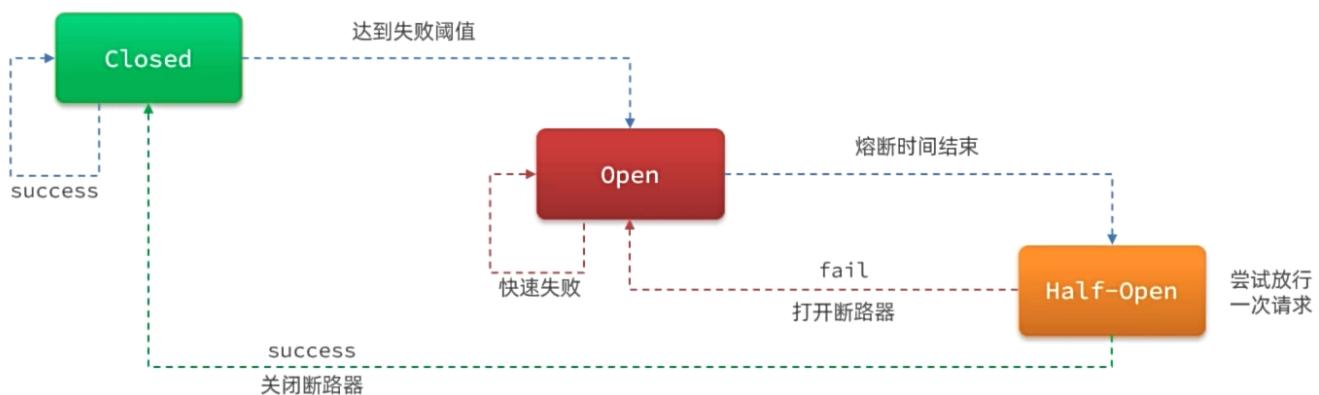


Figure 19: image-20251003150134978

## 分布式事务

在分布式系统中，如果一个业务需要**多个服务共同完成**，并且每一个服务都有事务，**多个事务必须同时成功或者失败**，这样的事务就是**分布式事务**

- 每个服务的事务是一个**分支事务**
- 整个业务称为**全局事务**

**Seata** 是阿里巴巴和蚂蚁金服共同开源的**分布式事务解决方案**

Seata 事务管理有三个重要角色：

- TC-事务协调者**：维护**全局和分支事务的状态**，协调全局事务提交或回滚
- TM-事务管理器**：定义**全局事务**的范围、开始全局事务、提交或回滚全局事务
- RM-资源管理器**：管理**分支事务**，与 TC 交谈以注册分支事务和报告分支事务的状态



Figure 20: image-20251003153535418

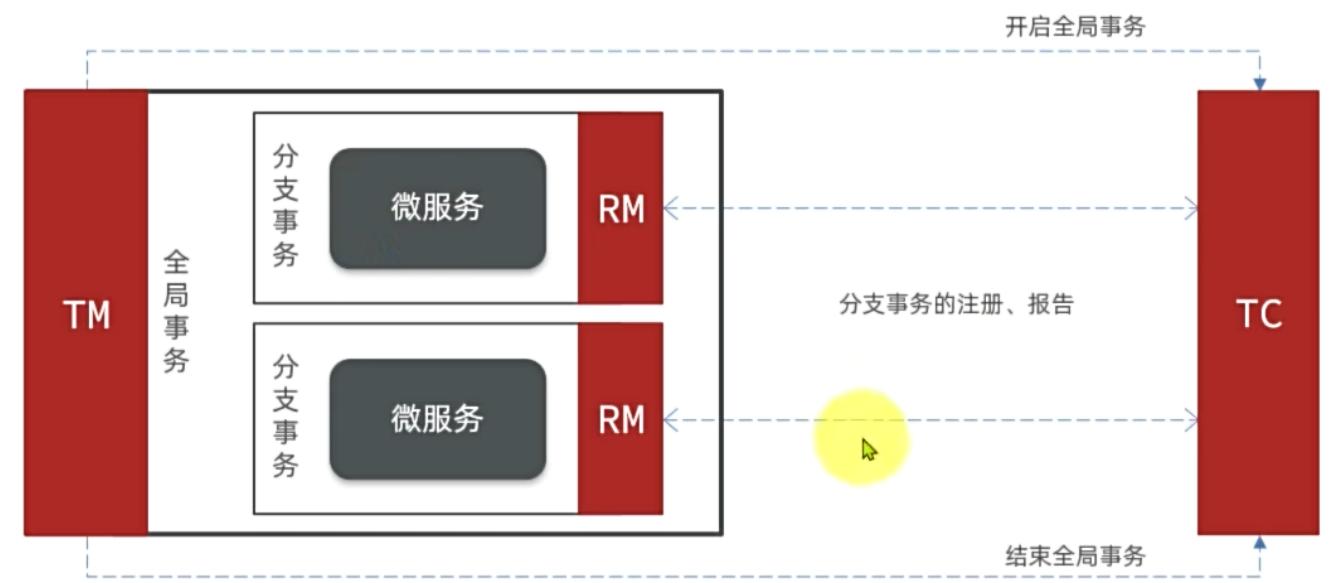


Figure 21: image-20251004110352044

## 使用步骤

- 引入依赖

```
<!--统一配置管理-->
<dependency>
 <groupId>com.alibaba.cloud</groupId>
 <artifactId>spring-cloud-starter-alibaba-nacos-config</artifactId>
</dependency>
<!--读取 bootstrap 文件-->
<dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-starter-bootstrap</artifactId>
</dependency>
<!--seata-->
<dependency>
 <groupId>com.alibaba.cloud</groupId>
 <artifactId>spring-cloud-starter-alibaba-seata</artifactId>
</dependency>
```

- 在 Nacos 注册中心配置 TC 服务地址

```
seata:
 registry: # TC 服务注册中心的配置，微服务根据这些信息去注册中心获取 tc 服务地址
 type: nacos # 注册中心类型 nacos
 nacos:
 server-addr: 172.21.172.16:8848 # nacos 地址
 namespace: "" # namespace，默认为空
 group: DEFAULT_GROUP # 分组，默认是 DEFAULT_GROUP
 application: seata-server # seata 服务名称
 username: nacos
 password: nacos
 tx-service-group: hmall # 事务组名称
 service:
 vgroup-mapping: # 事务组与 tc 集群的映射关系
 hmall: "default"
```

## Seata 不同的分布式事务模式

- XA 模式，强一致
- AT 模式，最终一致

**XA 模式** XA 模式是 X/Open 组织定义的**分布式事务管理标准**，描述了全局的 TM 与局部的 RM 之间的接口

### 执行流程

- 一阶段

- TM 开启全局事务，向 TC 报告
- 随即 TM 调用分支上的 RM
- RM 向 TC 注册分支事务
- RM 执行业务 sql



Figure 22: image-20251004110800127

- 为了保证所有分支事务的一致性，**执行完后不会立刻提交！**（锁定数据库资源）
- RM 向 TC 报告事务状态
- 二阶段**
  - TM 向 TC 报告结束全局事务
  - TC 检查分支事务状态
    - 分支事务状态**正常**，TC 告诉所有 RM 进行**提交**
    - 分支事务状态**异常**，TC 告诉所有 RM 进行**回滚**

一阶段的工作：

- ① RM注册分支事务到TC
- ② RM执行分支业务sql但不提交
- ③ RM报告执行状态到TC

二阶段的工作：

- TC检测各分支事务执行状态
  - a. 如果都成功，通知所有RM提交事务
  - b. 如果有失败，通知所有RM回滚事务
- RM接收TC指令，提交或回滚事务

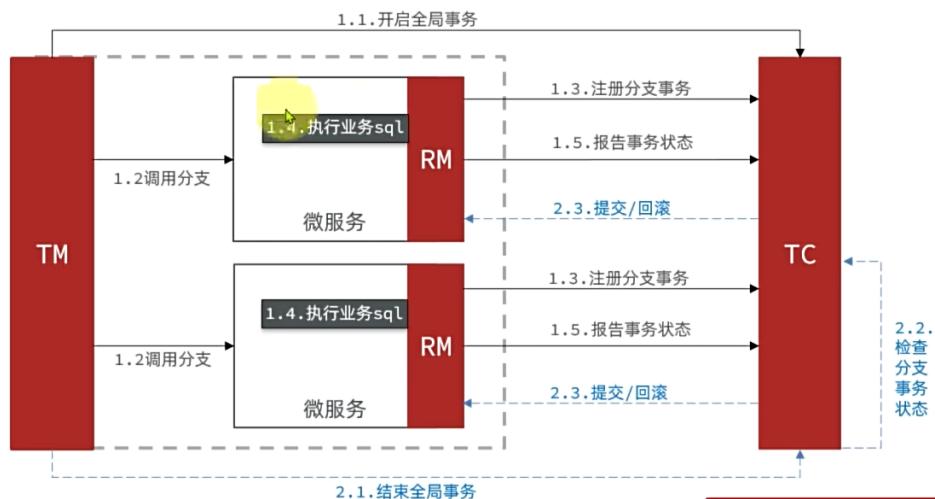


Figure 23: image-20251004112448684

**缺点：**

- 一阶段需要**锁定**数据库资源，二阶段结束才进行**释放**

## 实现 XA 模式

- 修改.yaml 文件，开启 XA 模式

```
seata:
 data-source-proxy-mode: XA
```

- 给发起全局事务的入口方法添加 `@GlobalTransactional` 注解

**AT 模式** Seata 主推的是 AT 模式，AT 模式同样是分阶段提交的事务模型，但弥补了 XA 模式中数据库资源锁定时间过长的缺陷

### 执行流程

#### • 一阶段

- TM 开启全局事务，向 TC 报告
- 随即 TM 调用分支上的 RM
- RM 向 TC 注册分支事务
- RM 记录更新前的快照
- RM 执行业务 sql 并立刻提交
- RM 向 TC 报告事务状态

#### • 二阶段

- TM 向 TC 报告结束全局事务
- TC 检查分支事务状态
  - 分支事务状态正常，删除快照
  - 分支事务状态异常，TC 告诉所有 RM 基于快照数据进行回滚

阶段一RM的工作：

- 注册分支事务
- 记录undo-log（数据快照）
- 执行业务sql并提交
- 报告事务状态

阶段二提交时RM的工作：

- 删除undo-log即可

阶段二回滚时RM的工作：

- 根据undo-log恢复数据到更新前



Figure 24: image-20251004114046322

### 缺点：

- AT 模式在中间可能出现短暂的不一致

## 实现 AT 模式

- 修改 .yaml 文件，开启 AT 模式

```
seata:
 data-source-proxy-mode: AT
```

## Elasticsearch

是基于 Lucene 开发的高性能分布式搜索引擎

- 支持分布式，可水平扩展
- 提供 Restful 接口，可被任何语言调用
- 结合 Kibana、Logstash、Beats，是一整套技术栈，被称为 ELK

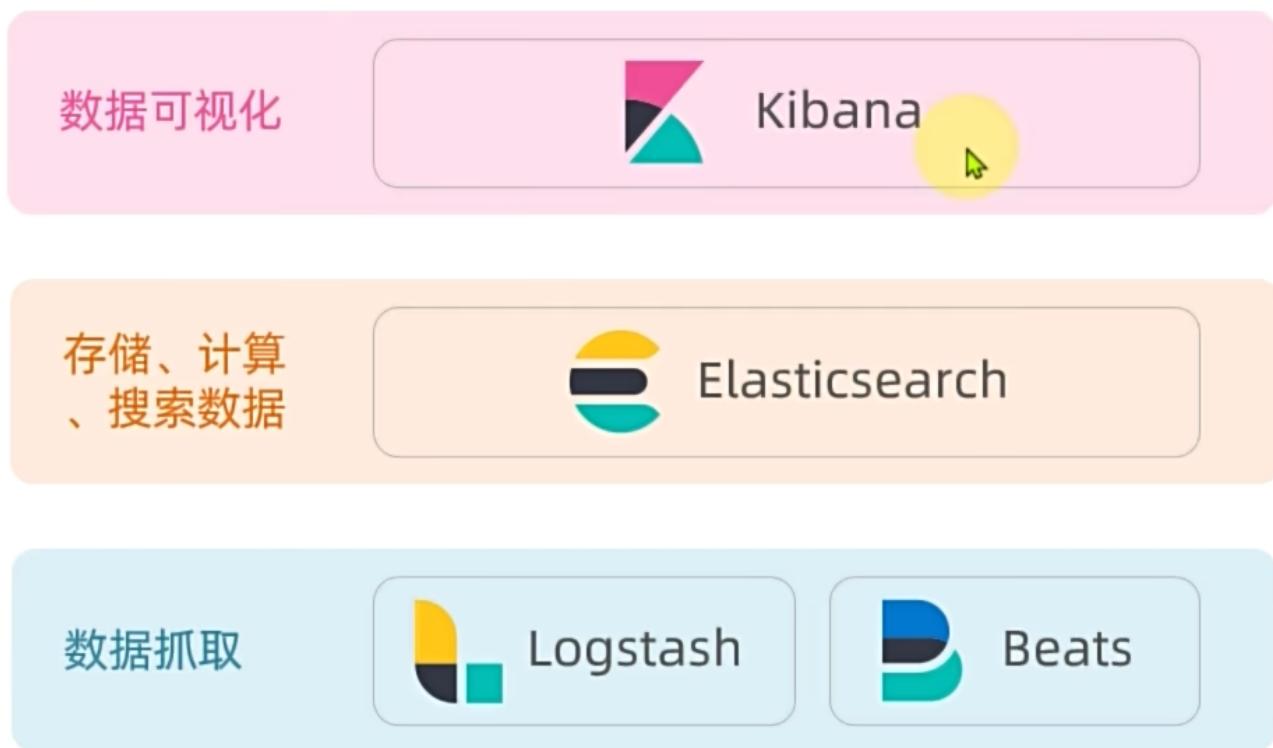


Figure 25: image-20251006161719370

## 倒排索引

传统数据库采用正向索引，而 Elasticsearch 采用倒排索引：

- 文档 (document)：每条数据就是一个文档
- 词条 (term)：文档按照语义分成的词语

The diagram illustrates the process of indexing documents. On the left, there is a table with four rows representing documents. The columns are labeled **id**, **title**, and **price**. The data is as follows:

<b>id</b>	<b>title</b>	<b>price</b>
1	小米手机	3499
2	华为手机	4999
3	华为小米充电器	49
4	小米手环	299

A large grey arrow points from the left table to the right, indicating the flow of data processing. On the right, there is another table representing the **词条 (term)** and **文档id (document id)**. The data is as follows:

<b>词条 (term)</b>	<b>文档id (document id)</b>
小米	1 , 3 , 4
手机	1 , 2
华为	2 , 3
充电器	3
手环	4

Figure 26: image-20251006163742030

## IK 分词器

是一款开源的中文分词器，专门为 Elasticsearch 和 Lucene 设计，用于对中文文本进行分词处理。它支持**细粒度切分 (ik\_smart)** 和**最细粒度切分 (ik\_max\_word)** 两种分词模式

- 内部会有一个词库，记录常见词语，会对输入字符串进行分词然后和词库的词语进行匹配
- 可以配置拓展词典来增加自定义词库，包括**扩展词**，**扩展停止词**
  - 利用 config 目录的 `IkAnalyzer.cfg.xml` 文件添加扩展词典 (`.dic`)
  - 在词典中添加扩展词

## 基本概念

elasticsearch 中的文档数据会被序列化成 json 格式后存储在 elasticsearch 中

**索引 (index)**：相同类型的文档的集合

**映射 (mapping)**：索引中文档的字段约束信息，类似表的**结构约束**

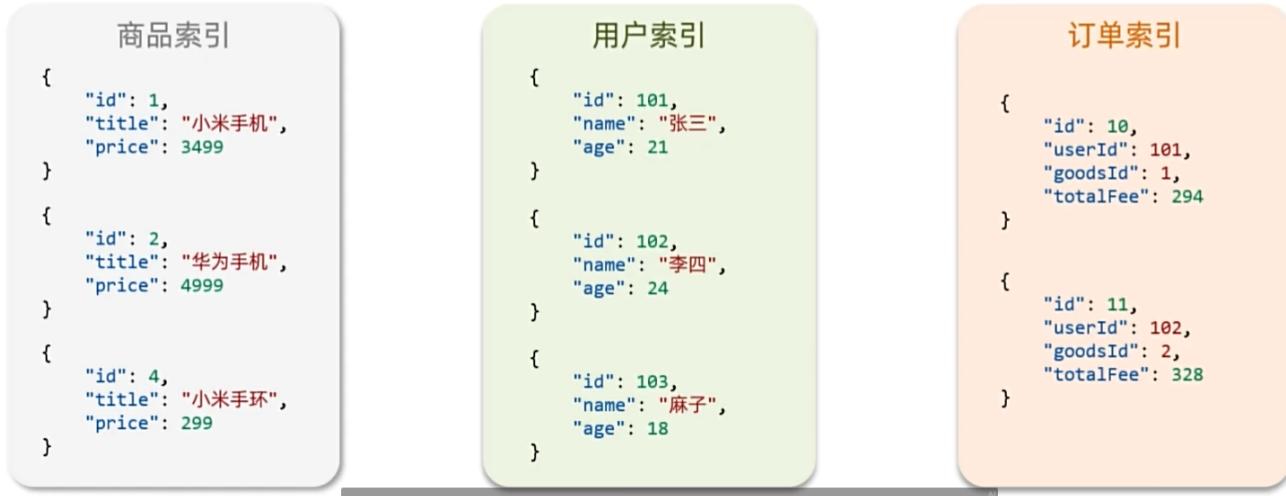


Figure 27: image-20251007133546860

## Mysql 和 elasticsearch 对应关系

MySQL	Elasticsearch	说明
Table	Index	索引(index)，就是文档的集合，类似数据库的表(table)
Row	Document	文档(Document)，就是一条条的数据，类似数据库中的行(Row)，文档都是JSON格式
Column	Field	字段(Field)，就是JSON文档中的字段，类似数据库中的列(Column)
Schema	Mapping	Mapping(映射)是索引中文档的约束，例如字段类型约束。类似数据库的表结构(Schema)
SQL	DSL	DSL是elasticsearch提供的JSON风格的请求语句，用来定义搜索条件

Figure 28: image-20251007133813213

## 索引库的操作

在 Kibana 的 Dev Tools 进行操作

Elasticsearch 提供的所有 API 都是 Restful 的接口

- 创建索引库和 mapping 的请求语法如下：

**PUT /索引库名称**

```
{
 "mappings": {
 "properties": {
 "字段名":{
 "type": "text",
 "analyzer": "ik_smart"
 },
 }
 }
}
```

```

 " 字段名 2": {
 "type": "keyword",
 "index": "false"
 },
 " 字段名 3": {
 "properties": {
 " 子字段": {
 "type": "keyword"
 }
 },
 // ... 略
 }
}

```

- 查询索引库

**GET /索引库名**

- 删除索引库

**DELETE /索引库名**

- 索引库和 mapping 一旦创建无法修改，但是可以添加新的字段

**PUT /索引库名/\_mapping**

```
{
 "properties": {
 " 新字段名": {
 "type": "integer"
 }
 }
}
```

## Mapping 映射属性

mapping 是对索引库中**文档的约束**，常见 mapping 属性包括：

- type: **字段数据类型**
  - 字符串: text (可分词的文本)、keyword (精确值)
  - 数值: long、integer、short、byte、double、float
  - 布尔: boolean
  - 日期: date
  - 对象: object
- index: 是否创建索引，**默认为 true**
  - **创建索引表明当前文档参与搜索**
- analyzer: 使用哪种分词器
- properties: 该字段的**子字段**

## 文档操作

- 新增文档的请求格式如下：

```
POST /索引库名/_doc/文档 id
```

```
{
 "字段 1": "值 1",
 "字段 2": "值 2",
 "字段 3": {
 "子属性 1": "值 3",
 "子属性 2": "值 4"
 },
 // ...
}
```

- 查询文档

```
GET /{索引库名称}/_doc/{id}
```

- 删除文档

```
DELETE /{索引库名}/_doc/id 值
```

- 修改文档

- 全量修改：删除旧文档，添加新文档

```
PUT /{索引库名}/_doc/文档 id
{
 "字段 1": "值 1",
 "字段 2": "值 2",
 // ... 略
}
```

- 局部修改：只修改指定 id 匹配的文档中的部分字段

```
POST /{索引库名}/_update/文档 id
{
 "doc": {
 "字段名": "新的值",
 }
}
```

- 批量处理

- index 代表新增操作
  - \_index：指定索引库名
  - \_id 指定要操作的文档 id
  - { "field1" : "value1" }：则是要新增的文档内容
- delete 代表删除操作
  - \_index：指定索引库名
  - \_id 指定要操作的文档 id
- update 代表更新操作
  - \_index：指定索引库名

- `_id` 指定要操作的文档 id
- `{ "doc" : { "field2" : "value2" } }`: 要更新的文档字段

### POST \_bulk

```
{
 "index" : { "_index" : "test", "_id" : "1" } }
{ "field1" : "value1" }
{ "delete" : { "_index" : "test", "_id" : "2" } }
{ "create" : { "_index" : "test", "_id" : "3" } }
{ "field1" : "value3" }
{ "update" : { "_id" : "1", "_index" : "test" } }
{ "doc" : { "field2" : "value2" } }
```

## JavaRestClient

### 初始化步骤：

- 引入依赖

```
<dependency>
 <groupId>org.elasticsearch.client</groupId>
 <artifactId>elasticsearch-rest-high-level-client</artifactId>
</dependency>
```

- 初始化 `RestHighLevelClient`
  - 需要用 `HttpHost.create` 方法指定 ip 地址和端口号

```
RestHighLevelClient restHighLevelClient = new RestHighLevelClient(RestClient.builder(
 HttpHost.create("172.21.172.16:9200")
));
```

## 索引库的操作 创建索引库

- 使用 `CreateIndexRequest` 类创建 `request` 对象
  - 里面传入索引库名称
- 使用 `request.source()` 方法设置请求参数
- 使用 `.indices()` 方法获取索引库的所有操作方法
  - 使用 `.create()` 方法创建索引库
    - 传入 `request` 对象

```
@Test
void testCreateIndex() throws IOException {
 // 1. 创建 Request 对象
 CreateIndexRequest request = new CreateIndexRequest("items");
 // 2. 准备请求参数
 request.source(MAPPING_TEMPLATE, XContentType.JSON);
 // 3. 发送请求
 client.indices().create(request, RequestOptions.DEFAULT);
}
```

## 删除索引库

- 使用 `DeleteIndexRequest` 类创建 `request` 对象

- 里面传入索引库名称
- 使用`.indices()`方法获取索引库的**所有操作方法**
  - 使用`.delete()`方法删除索引库
    - 传入 request 对象

```
@Test
void testDeleteIndex() throws IOException {
 // 1. 创建 Request 对象
 DeleteIndexRequest request = new DeleteIndexRequest("items");
 // 2. 发送请求
 client.indices().delete(request, RequestOptions.DEFAULT);
}
```

## 查询索引库信息

- 使用`GetIndexRequest`类创建`request`对象
  - 里面传入索引库名称
- 使用`.indices()`方法获取索引库的**所有操作方法**
  - 使用`.get()`方法查询索引库，也可以调用`.exists()`方法查看索引库是否存在
    - 传入`request`对象

```
@Test
void testExistsIndex() throws IOException {
 // 1. 创建 Request 对象
 GetIndexRequest request = new GetIndexRequest("items");
 // 2. 发送请求
 client.indices().get(request, RequestOptions.DEFAULT);
}
```

## 文档操作 新增文档

- 使用`IndexRequest`类创建`Request`对象，并使用`.id()`方法指定文档`id`
- 使用`.source()`方法准备请求参数
- 使用`.index()`方法发送请求

```
@Test
void testIndexDoc() throws IOException {
 // 准备文档数据
 Item item = iItemService.getById(100000011127L);
 ItemDoc itemDoc = BeanUtil.copyProperties(item, ItemDoc.class);

 IndexRequest request = new IndexRequest("items").id(itemDoc.getId());
 request.source(JSONUtil.toJsonStr(itemDoc), XContentType.JSON);
 restHighLevelClient.index(request, RequestOptions.DEFAULT);
}
```

## 删除文档

- 使用`DeleteRequest`类创建`Request`对象，指定索引和文档`id`
- 使用`.delete()`方法发送请求

```

@Test
void testDeleteDocument() throws IOException {
 // 1. 准备 Request, 两个参数, 第一个是索引库名, 第二个是文档 id
 DeleteRequest request = new DeleteRequest("item", "100002644680");
 // 2. 发送请求
 client.delete(request, RequestOptions.DEFAULT);
}

```

## 查询文档

- 使用 GetRequest 类创建 Request 对象, 指定索引和文档 id
- 使用 .get() 方法发送请求获取响应结果, 返回类型为 GetResponse
- 使用 .getSourceAsString() 方法获取 \_source 字段的信息



```

{
 "_index": "users",
 "_type": "_doc",
 "_id": "1",
 "_version": 1,
 "_seq_no": 0,
 "_primary_term": 1,
 "found": true,
 "_source": {
 "name": "Jack",
 "age": 21
 }
}

```

Figure 29: image-20251007151903429

```

@Test
void testGetDocumentById() throws IOException {
 // 1. 准备 Request 对象
 GetRequest request = new GetRequest("items").id("100002644680");
 // 2. 发送请求
}

```

```

GetResponse response = client.get(request, RequestOptions.DEFAULT);
// 3. 获取响应结果中的 source
String json = response.getSourceAsString();

ItemDoc itemDoc = JSONUtil.toBean(json, ItemDoc.class);
System.out.println("itemDoc= " + itemDoc);
}

```

## 修改文档

- 全量修改

- 在 RestClient 的 API 中，全量修改与新增的 API 完全一致，判断依据是 ID：
  - 如果新增时，ID 已经存在，则修改
  - 如果新增时，ID 不存在，则新增

- 局部修改

- 使用 UpdateRequest 类创建 Request 对象，指定索引和文档 id
- 使用 request.doc() 方法准备请求参数，每两个参数为一对 key value
- 使用 .update() 方法更新文档

```

@Test
void testUpdateDocument() throws IOException {
 // 1. 准备 Request
 UpdateRequest request = new UpdateRequest("items", "100002644680");
 // 2. 准备请求参数
 request.doc(
 "price", 58800,
 "commentCount", 1
);
 // 3. 发送请求
 client.update(request, RequestOptions.DEFAULT);
}

```

## 批量处理

- 使用 BulkRequest 类来封装普通的 CRUD 请求，获得 request 对象
- 使用 .add() 方法添加批量提交的请求
  - 使用链式语法声明参数
- 使用 .bulk() 方法发送 bulk 请求

```

@Test
void testBulk() throws IOException {
 // 1. 创建 Request
 BulkRequest request = new BulkRequest();
 // 2. 准备请求参数
 request.add(new IndexRequest("items").id("1").source("json doc1", XContentType.JSON));
 request.add(new IndexRequest("items").id("2").source("json doc2", XContentType.JSON));
 // 3. 发送请求
 client.bulk(request, RequestOptions.DEFAULT);
}

```

## DSL 查询

### 以 JSON 格式来定义查询条件

DSL 查询可以分为两大类：

- **叶子查询**：一般是在特定的字段里查询特定值，**简单查询**，很少使用
- **复合查询**：以逻辑方式组合多个叶子查询或者更改叶子查询的行为方式

查询以后还可以对查询的结果做处理：

- **排序**：按照 1 个或多个字段值做排序
- **分页**：根据 from 和 size 做分页，类似 MySQL
- **高亮**：对搜索结果中的关键字添加**特殊样式**，使其更加醒目
- **聚合**：对搜索结果做数据统计以形成报表

### 基于 DSL 的查询语法

```
GET /{索引库名}/_search
{
 "query": {
 "查询类型": {
 // .. 查询条件
 }
 }
}
```

### 无条件查询

- 返回结果
  - took: 查询耗时
  - time\_out: 是否超时
  - \_shards: 分片信息
  - hits: 命中结果
    - total: 匹配文档总数
    - max\_score: 最高相关性分数
    - hits: 文档详情数组

```
GET /items/_search
{
 "query": {
 "match_all": {
 }
 }
}
```

### 叶子查询

- **全文检索查询**：利用分词器对用户输入内容分词，然后去词条列表匹配
  - match\_query
  - multi\_match\_query
- **精确查询**：不对用户输入内容分词，直接精确匹配，一般是查找 keyword、数值、日期、布尔等类型
  - ids: 按文档 id 查询

- range: 给定范围去进行查询
- term: 给定词条直接匹配
- 地理查询: 用于搜索地理位置

**match 查询:** 全文检索查询的一种, 对用于输入内容分词, 然后去倒排索引库检索

```
GET /{索引库名}/_search
{
 "query": {
 "match": {
 "字段名": "搜索条件"
 }
 }
}
```

**multi\_match 查询:** 与 match 查询类似, 不过允许同时查询多个字段

```
GET /{索引库名}/_search
{
 "query": {
 "multi_match": {
 "query": "搜索条件",
 "fields": ["字段 1", "字段 2"]
 }
 }
}
```

### term 查询

```
GET /{索引库名}/_search
{
 "query": {
 "term": {
 "字段名": {
 "value": "搜索条件"
 }
 }
 }
}
```

### range 查询

- gte 是包含最小值, gt 是不包含最小值
- lte 是包含最大值, lt 是不包含最大值

```
GET /{索引库名}/_search
{
 "query": {
 "range": {
 "字段名": {
 "gte": {最小值},
 "lte": {最大值}
 }
 }
 }
}
```

```
 }
}
}
```

## 复合查询 可以分为两大类

- 基于逻辑运算组合叶子查询，实现组合条件
  - bool
- 基于某种算法修改查询时的文档相关性算分，改变文档排名
  - function\_score
  - dis\_max

**布尔查询：**是一个或多个查询子句的组合，组合方式有：

- must：必须匹配每个子查询，类似”与”
- should：选择性匹配子查询，类似”或”
- must\_not：必须不匹配，不参与算分，类似”非”
- filter：必须匹配，不参与算分

**GET /items/\_search**

```
{
 "query": {
 "bool": {
 "must": [
 {"match": {"name": "手机"}}
],
 "should": [
 {"term": {"brand": { "value": "vivo" }}},
 {"term": {"brand": { "value": "小米" }}}
],
 "must_not": [
 {"range": {"price": {"gte": 2500}}}
],
 "filter": [
 {"range": {"price": {"lte": 1000}}}
]
 }
 }
}
```

因此，用来做过滤筛选的**不参与算分**，用户输入框输入的搜索条件**要参与算分**

## 排序和分页 排序语法

- elasticsearch 支持对搜索结果排序，默认是按照相关度算分 (\_score) 来排序，也可以**按指定字段排序**，可排序类型有：**keyword 类型、数值类型、地理坐标类型、日期类型等**
- 可以指定多个排序字段

**GET /indexName/\_search**

```
{
```

```

"query": {
 "match_all": {}
},
"sort": [
 {
 "排序字段 1": {
 "order": "排序方式 asc 和 desc"
 }
 },
 {
 "排序字段 2": {
 "order": "排序方式 asc 和 desc"
 }
 }
]
}

```

## 分页语法

- 默认情况只返回 top10 的数据，要查询更多数据就需要修改分页参数了
- 通过 from、size 参数来控制要返回的分页结果
  - from: 从第几个文档开始
  - size: 查询几个文档
  - from + size 的值不得超过 10000

```

GET /items/_search
{
 "query": {
 "match_all": {}
 },
 "from": 0, // 分页开始的位置，默认为 0
 "size": 10, // 每页文档数量，默认 10
 "sort": [
 {
 "price": {
 "order": "desc"
 }
 }
]
}

```

## 深度分页

- es 的数据一般会采用分片存储，也就是把一个索引中的数据分成 N 份，存储到不同节点上，查询数据时需要 **汇总各个分片的数据**
- 深度分页的解决方法：**search after**
  - 分页时需要排序，原理是从上一次的排序值开始，查询下一页数据

**高亮显示** 把搜索关键字突出显示，使用 em 标签对需要高亮的字段进行包裹

## 语法

- 也可以不加标签，默认就是 em

```
GET /{索引库名}/_search
{
 "query": {
 "match": {
 "搜索字段": "搜索关键字"
 }
 },
 "highlight": {
 "fields": {
 "高亮字段名称": {
 "pre_tags": "",
 "post_tags": ""
 }
 }
 }
}
```

**数据聚合** 可以实现对文档数据的统计、分析、运算，常见的有三类：

- **桶聚合**：对文档做分组
  - TermAggregation：按照文档字段值分组
  - Date Histogram：按照日期阶梯分组
- **度量聚合**：计算一些值，如最大值、最小值等
  - Avg：求平均值
  - Max：求最大值
  - Min：求最小值
  - Status：同时求 max、min、avg、sum 等
- **管道聚合**：其它聚合的结果为基础做聚合

### 聚合语法

- **桶聚合**
  - 使用 aggs 定义聚合，可以定义多个聚合

```
GET /items/_search
{
 "query": {
 "match_all": {}
 }, //可以省略
 "size": 0, //设置 size 为 0，结果中不包含文档，只包含聚合结果
 "aggs": { //定义聚合
 "cateAgg": { //给聚合起个名字
 "terms": { //聚合的类型，按照品牌值聚合，所以选择 term
 "field": "category", //参与聚合的字段
 "size": 20 // 希望获取的聚合结果数量
 }
 }0
 }
}
```

- 度量聚合
  - 使用嵌套聚合获取每个品牌的度量信息

```
GET /items/_search
{
 "query": {
 "bool": {
 "filter": [
 {
 "term": {
 "category": "手机"
 }
 },
 {
 "range": {
 "price": {
 "gte": 300000
 }
 }
 }
]
 }
 },
 "size": 0,
 "aggs": {
 "brand_agg": {
 "terms": {
 "field": "brand",
 "size": 20
 },
 "aggs": { // 嵌套聚合
 "stats_meric": {
 "stats": {
 "field": "price"
 }
 }
 }
 }
 }
}
```

## JavaRestClient 查询

### 搜索

- 使用 SearchRequest 类创建 request 对象
  - 里面传入索引库名称
- 使用 request.source() 方法设置请求参数
- 使用 request.query() 方法设置查询方式

- 传入 QueryBuilders 的请求方式
  - matchAllQuery() 方法
- 使用 .search() 方法进行查询
  - 传入 request 对象

```
@Test
void testMatchAll() throws IOException{
 // 1. 准备 request
 SearchRequest request = new SearchRequest();
 // 2. 构建 DSL 参数
 request.source()
 .query(QueryBuilder.matchAllQuery());
 // 3. 发送请求
 client.search(request,Req)
}
```

**构建查询条件** 所有类型的 Query 查询条件都是基于 QueryBuilders 构建的

- 全文检索构建查询 API
  - QueryBuilders.MatchQuery 构建单字段查询
  - QueryBuilders.multiMatchQuery 构建多字段查询

```
// 单字段查询
QueryBuilders.MatchQuery("name", " 脱脂牛奶");
// 多字段查询
QueryBuilders.multiMatchQuery(" 脱脂牛奶", "name", "catrgory");
```



Figure 30: image-20251008134512732

- 精确查询构建 API
  - 使用 QueryBuilders.termQuery 构建词条查询
  - 使用 QueryBuilders.rangeQuery 构建范围查询

```
// 词条查询
QueryBuilders.termQuery("category", "牛奶");
// 范围查询
QueryBuilders.rangeQuery("price").gte(100).lte(150);
```

The image shows two terminal windows side-by-side. The left window contains Java code for creating term and range queries:

```
// 词条查询
QueryBuilders.termQuery("category", "牛奶");
// 范围查询
QueryBuilders.rangeQuery("price").gte(100).lte(150);
```

The right window shows the generated Elasticsearch search requests:

```
GET /items/_search
{
 "query": {
 "term": {
 "category": "牛奶"
 }
 }
}
GET /items/_search
{
 "query": {
 "range": {
 "price": { "gte": 100, "lte": 150 }
 }
 }
}
```

Figure 31: image-20251008134635397

- 布尔查询构建 API
  - 使用 `BoolQueryBuilder` 接收 `QueryBuilders.boolQuery()` 创建的布尔查询
  - 使用 `.must()` 添加 must 条件，里面传入叶子查询的构建

```
// 创建布尔查询
BoolQueryBuilder boolQuery = QueryBuilders.boolQuery();
// 添加 must 条件
boolQuery.must(
 QueryBuilders.termQuery("brand", "华为"));
// 添加 should 条件
boolQuery.should(
 QueryBuilders.rangeQuery("price").lte(2500));
```

### 排序和分页 排序和分页的参数是基于 `request.source()` 设置

- 使用 `.from()` 和 `.size()` 方法指定分页参数
- 使用 `.sort()` 方法进行排序
  - 使用 `SortOrder` 指定排序方式
  - 可以加多个排序条件

```
// 分页
request.source().from(0).size(5);
// 价格排序
request.source().sort("price", SortOrder.ASC);
```

### 高亮显示 基于 `request.source()` 设置

- 使用 `.highlighter()` 方法传入高亮参数

- 使用 SearchSourceBuilder.highlight() 构建高亮参数

```
request.source().highlighter(
 SearchSourceBuilder.highlight()
 .field("name")
 .preTags("")
 .postTags("")
);
```

- 解析代码如下，上面的查询，排序，分页都可以用这套解析代码

```
private static void parseResponseResult(SearchResponse search) {
 SearchHits hits = search.getHits();

 long total = hits.getTotalHits().value;

 SearchHit[] searchHits = hits.getHits();

 for (SearchHit searchHit : searchHits) {
 String json = searchHit.getSourceAsString();

 // 处理高亮
 Map<String, HighlightField> hfs = searchHit.getHighlightFields();
 if(hfs!=null && !hfs.isEmpty()){
 HighlightField hf = hfs.get("name");
 json = hf.getFragments()[0].string();
 }
 System.out.println("json"+json);
 }
}
```

## 数据聚合 基于 request.source() 设置

- 使用.aggregation() 方法传入聚合参数构造器
- 使用 AggregationBuilders 指定聚合参数

```
request.source().aggregation(
 AggregationBuilders
 .terms("brand_agg") // 指定聚合名称
 .field("brand") // 指定聚合字段
 .size(20) // 指定返回的类别数
);
```

- 解析的时候根据查询结果一点一点推

```
@Test
void testAgg() throws IOException {

 SearchRequest request = new SearchRequest("items");
```

```

request.source().size(0);

String brandAggName = "brandAgg";
request.source().aggregation(
 AggregationBuilders
 .terms(brandAggName)
 .field("brand")
 .size(10)
);
SearchResponse search = restHighLevelClient.search(request, RequestOptions.DEFAULT);
System.out.println("search results:" + search);

Aggregations aggregations = search.getAggregations();
Terms brandTerms = aggregations.get(brandAggName);
List<? extends Terms.Bucket> buckets = brandTerms.getBuckets();

for (Terms.Bucket bucket : buckets) {
 System.out.println("brand:" + bucket.getKey());
 System.out.println("count:" + bucket.getDocCount());
}
}

```

## 微服务面试

### 分布式事务

分布式事务最大的问题就是各个子事务的一致性问题

- **CP 模式：**各个子事务执行后互相等待，同时提交和回滚，**达成强一致**，但事务等待过程中处于**弱可用状态**
  - 对应 XA 模式
- **AP 模式：**各子事务分别执行和提交，**允许出现结果不一致**，然后采用弥补措施恢复数据，**实现最终一致**
  - 对应 AT 模式

**CAP 定理** 分布式系统的三个指标：

- Consistency (一致性)
- Availability (可用性)
- Partition tolerance (分区容错性)

在一个分布式系统里无法同时满足这三者

**Base 理论** BASE 理论是对 CAP 的一种解决思路，包含三个思想：

- **Basically Available (基本可用)：**分布式系统在出现故障时，允许损失部分可用性，即保证核心可用。
- **Soft State (软状态)：**在一定时间内，允许出现中间状态，比如临时的不一致状态。
- **Eventually Consistent (最终一致性)：**虽然无法保证强一致性，但是在软状态结束后，最终达到数据一致。

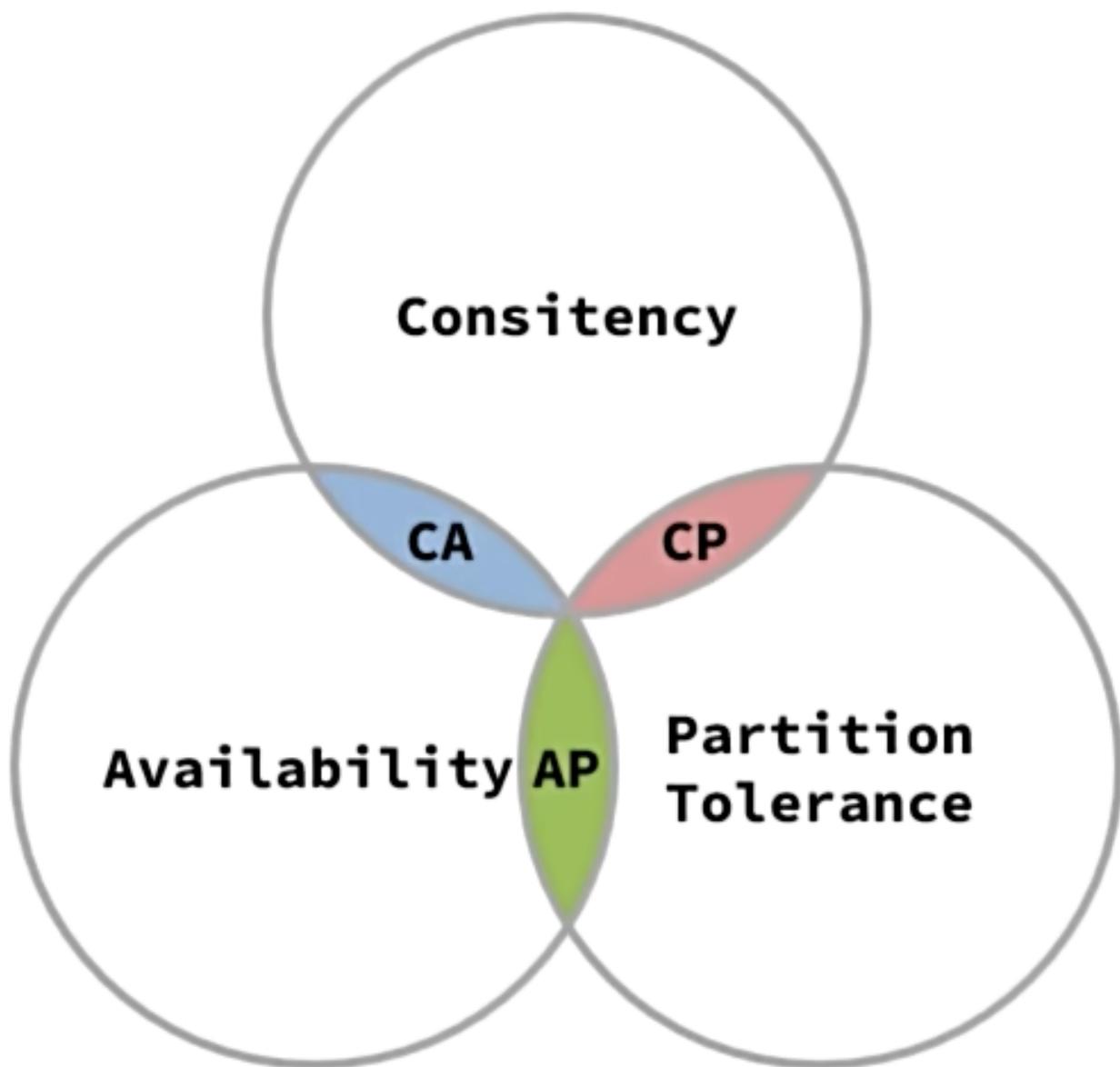


Figure 32: image-20251008145844622

## AT 模式的脏写问题