

SpringCloud

xbZhong

2025-10-08

[本页PDF](#)

MybatisPlus

对于 Mybaitis 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 `MybaitisPlus` 提供的 `BaseMapper` 接口，指定要操作的实体类

```
1 public interface UserMapper extends BaseMapper<User> {  
2  
3 }
```

- `BaseMapper` 里面内置了很多单表CRUD的方法，可以直接使用
- `MyabaitisPlus` 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

```
1  @TableName("tb_user")
2  public class User{
3      @TableId(type=IdType.ASSIGN_ID)
4      private long id;
5
6      private String name;
7
8      private Boolean isMarried;
9      @TableField("`order`")
10     private Integer order;
11     @TableField(exist=false)
12     private String address;
13 }
```

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

```

1 // 自定义Mapper接口
2 public interface UserMapper extends BaseMapper<User>{
3
4 }
5
6 @Autowired
7 private UserMapper userMapper;
8
9 void testQueryMapper(){
10     // 1. 构建查询条件
11     QueryWrapper<User> wrapper = new QueryWrapper<>()
12         .select("id","username","info","balance")
13         .like("username","o")
14         .ge("balance",1000);
15
16     // 2. 查询
17     List<User> users = userMapper.selectList(wrapper);
18     users.forEach(user-> System.out.println(user));
19 }
20
21 void testUpdateByQueryWrapper(){
22     // 1. 准备要更新的数据
23     User user = new User();
24     user.setBalance(2000);
25     // 2. 要更新的条件
26     QueryWrapper<User> wrapper = new QueryWrapper<User>().eq("username","jack");
27     // 3. 执行更新
28     userMapper.update(user,wrapper);
29 }
30
31 void testUpdataWrapper(){
32     List<long> ids = List.of(1L,2L,4L);
33     UpdateWrapper<User> wrapper = new UpdateWrapper<User>()
34         .setSql("balance = balance - 200") //update里面set后面的
35         .in("id",ids);
36
37     // 不需要填更新参数
38     userMapper.update(null,wrapper);
39 }

```

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

```

1  void testLambdaQueryMapper(){
2      // 1.构建查询条件
3      LambdaQueryWrapper<User> wrapper = new LambdaQueryWrapper<User>()
4          .select(User::getId,User::getUsername,User::getInfo,User::getBalance) //实例方法的方法引用
5          .like(User::getUsername,"o")
6          .ge(User::getBalance,1000);
7
8      // 2.查询
9      List<User> users = userMapper.selectList(wrapper);
10     users.forEach(user-> System.out.println(user));
11 }

```

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

1. 在业务层基于 `Wrapper` 构建 `where` 条件

```

1  List <long> ids = List.of(1L,2L,4L);
2  int amount = 200;
3  // 1.构建查询条件
4  LambdaQueryWrapper<User> wrapper = new LambdaQueryWrapper<User>
5      ().in(User::getId,ids);
6
7  // 2.自定义SQL方法调用
8  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

```

1  <update id="updateBalanceByIds">
2      UPDATE tb_user
3      SET balance = balance - #{amount}
4      ${ew.customSqlSegment} <!-- 由Mp帮我们自动解析，自动拼接SQL语句 -->
5  </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**

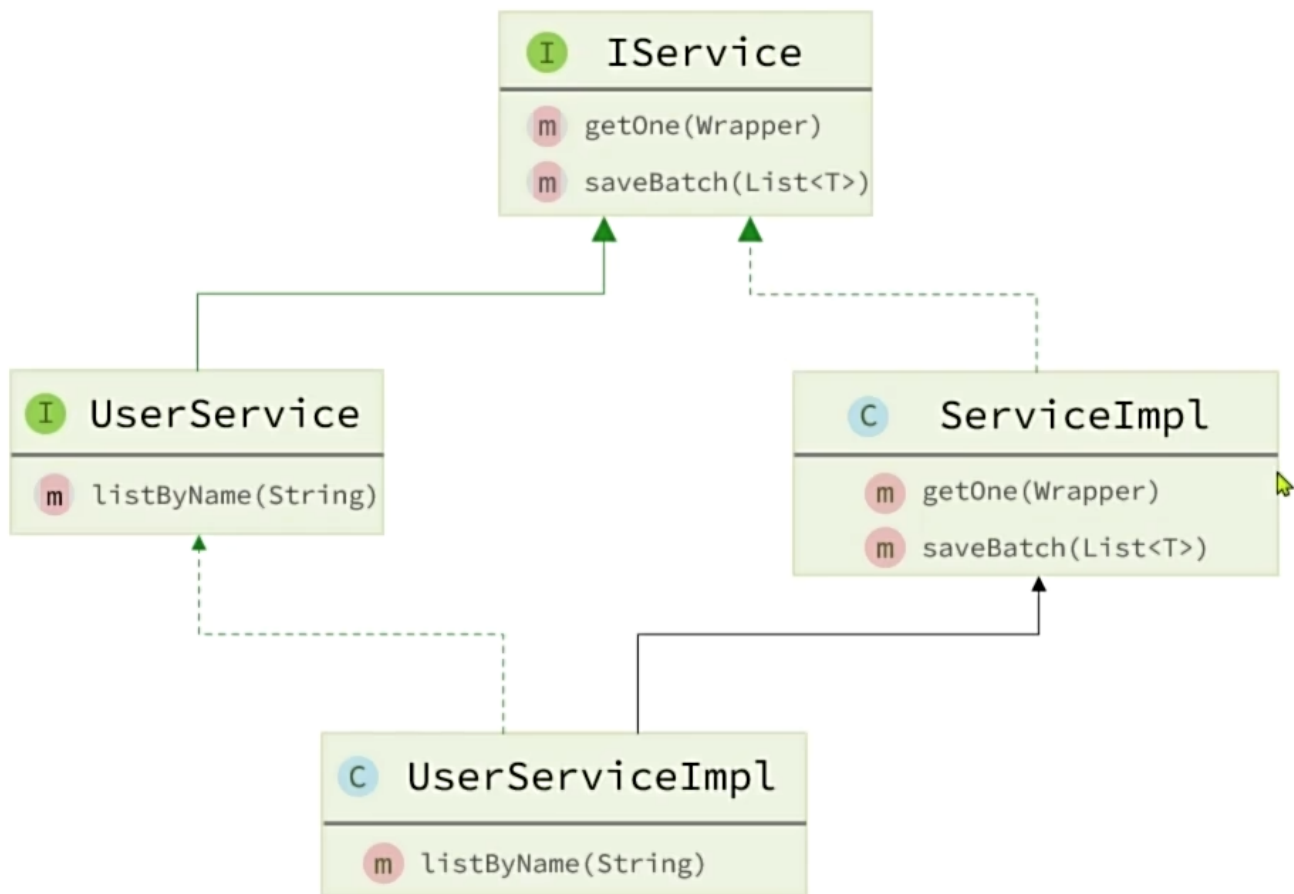


image-20250831160832309

```

1 // 自定义类
2 // 需要指定数据访问层的UserMapper和要操作的实体类User
3 @Service
4 public class UserServiceImpl extends ServiceImpl<UserMapper,User> implements
    UserService{
5
6 }
7 // 自定义接口
8 public interface UserService extends IService<User>{
9
10 }
11 // 使用
12 @Autowired
13 private UserService userService;
14 // 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
- 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

```

1 @RequestMapping("/users")
2 @RequiredArgsConstructor
3 public class UserController{
4     private final IUserService userService;
5
6     @DeleteMapping("/{id}")
7     public void deleteUserById(@PathVariable("id") Long id){
8         /*
9             方法体
10         */
11     }
12 }

```

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
- `lambdaUpadte()` ：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

```

1  // 查询
2  List<User> list = lambdaQuery()
3      .like(name != null,User::getNmae,name)
4      .eq(status != null,User::getStatus,status)
5      .ge(minBalance != null,User::getBalance,minBalance)
6      .le(maxBalance != null,User::getBalance,maxBalance)
7      .list();
8
9
10 // 更新
11 lambdaUpdate()
12     .set(User::getBalance,remainBalance)
13     .set(remainBalance == 0,User::getStatus,2)
14     .eq(User::getId,id)
15     .eq(User::getBalance,user.getBalance()) // 乐观锁
16     .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

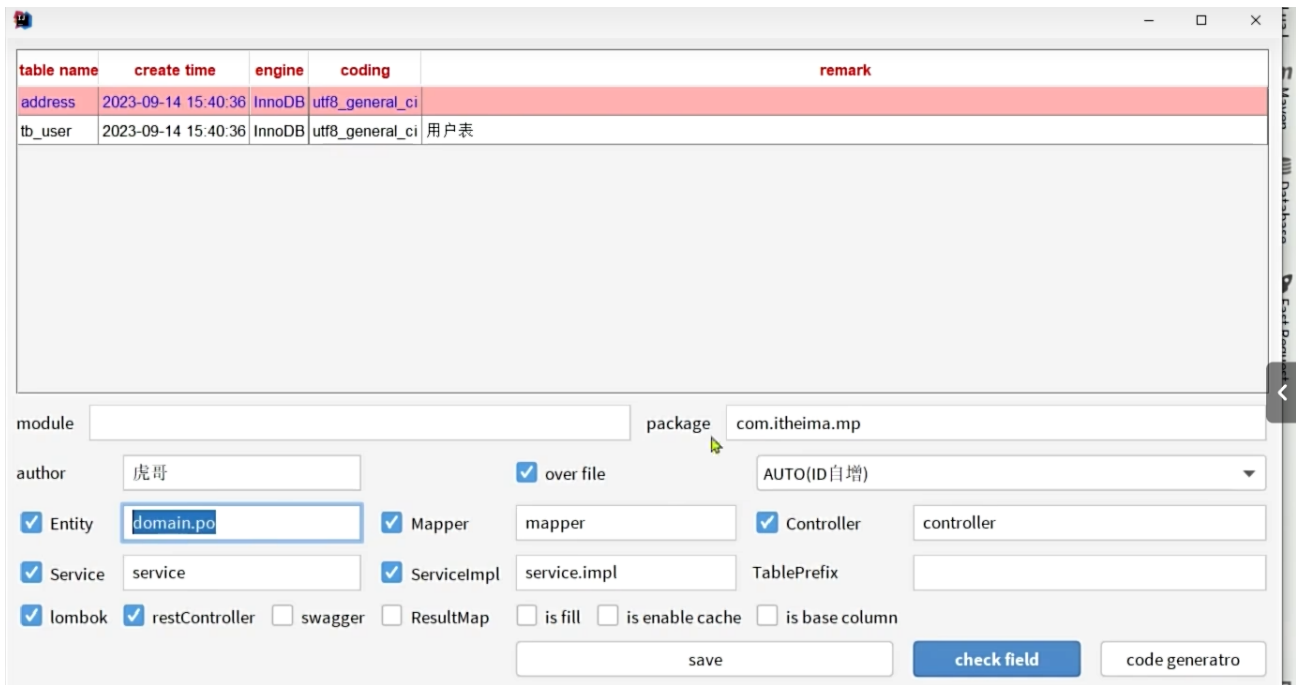


image-20250927144021258

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

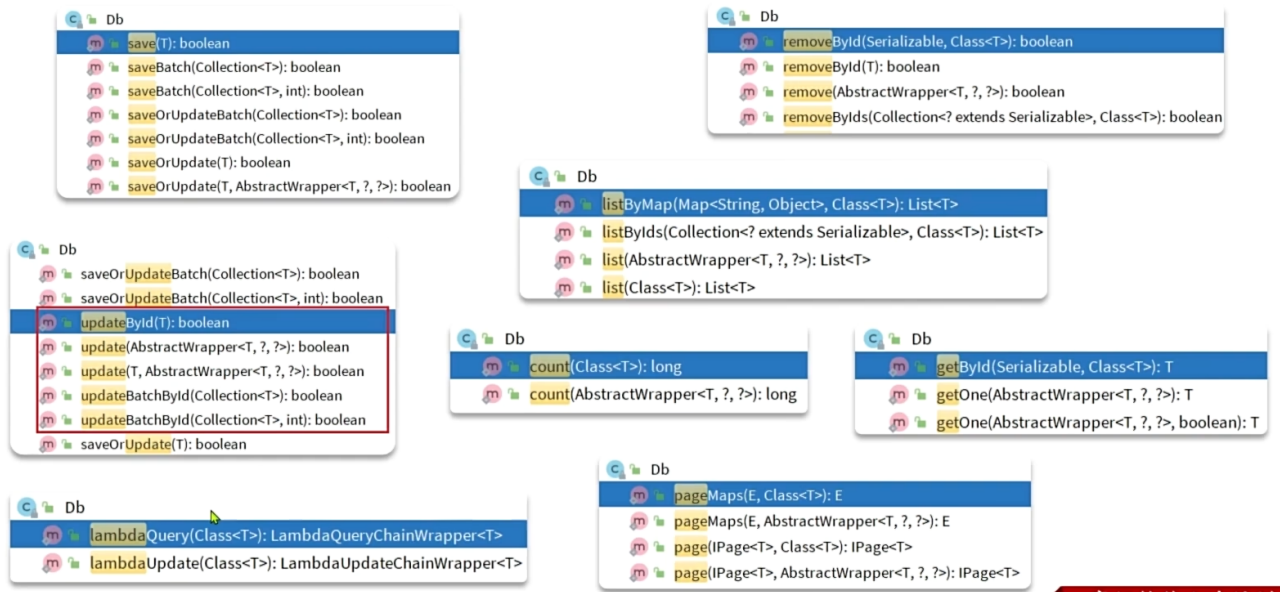


image-20250927144350529

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 application.yaml 文件中配置逻辑删除的字段名称和值即可

```
1 mybatis-plus:
2   global-config:
3     db-config:
4       logic-delete-field: flag # 全局逻辑删除的实体字段名
5       logic-delete-value: 1    # 逻辑已删除的值（默认1）
6       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

```
1 @Getter
2 public enum UserStatus{
3     NORMAL(1,"正常"),
4     FREEZE(2,"冻结")
5     ;
6     @EnumValue
7     @JsonValue
8     private final int value;
9     private final String desc;
10
11     UserStatus(int value,String desc){
12         this.value = value;
13         this.desc = desc;
14     }
15 }
```

同时还要在配置文件添加配置

```
1 mybatis-plus:
2   configuration:
3     default-enum-type-handler:
4       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`

```
1  @Data
2  @TableName(value = "user",autoResultMap = true)
3  public class User{
4      private long id;
5      private String name;
6
7      @TableField(typeHandler = JacksonTypeHandler.class)
8      private UserInfo info;
9  }
10
11 @Data
12 public class UserInfo{
13     private Integer age;
14     private String intro;
15     private String gender;
16 }
```

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：

```

1  @Configuration
2  public class MybatisConfig{
3
4      @Bean
5      public MybatisPlusInterceptor mybatisPlusInterceptor(){
6          // 初始化核心插件
7          MybatisPlusInterceptor interceptor = new MybatisPlusInterceptor();
8          // 添加分页插件
9          PaginationInnerInterceptor pageInterceptor = new
10         PaginationInnerInterceptor(DbType.MYSQL);
11         pageInterceptor.setMaxLimit(1000L); // 设置分页上限
12         interceptor.addInnerInterceptor(pageInterceptor);
13         return interceptor;
14     }
15 }

```

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

```

1  @Test
2  void testPageQuery() {
3      // 1. 准备分页参数
4      int pageNo = 1, pageSize = 5;
5
6      // 1.1 创建分页对象
7      Page<User> page = Page.of(pageNo, pageSize);
8
9      // 1.2 设置排序参数（按balance字段降序排列）
10     page.addOrder(new OrderItem("balance", false));
11
12     // 1.3 执行分页查询
13     Page<User> p = userService.page(page);
14
15     // 2. 输出总记录数
16     System.out.println("total=" + p.getTotal());
17
18     // 3. 输出总页数
19     System.out.println("pages=" + p.getPages());
20
21     // 4. 输出分页数据
22     List<User> records = p.getRecords();
23     records.forEach(System.out::println);
24 }

```

通用分页实体

查询类可以通过继承实现

```

1  @Data
2  public class PageQuery{
3      private Integer pageNo;
4      private Integer pageSize;
5      private String sortBy;
6      private Boolean isAsc;
7  }
8
9  @Data
10 public class UserQuery extends PageQuery{
11     private String name;
12     private Integer status;
13 }

```

分页结果

```

1 public class PageDTO<T>{
2     // 总条数
3     private Integer total;
4     // 总页数
5     private Integer pages;
6     // 分页结果
7     private List<T> list;
8 }

```

将 `PageQuery` 对象转为MP中的 `Page` 对象

```

1 @Data
2 public class PageQuery{
3     private Integer pageNo = 1;
4     private Integer pageSize = 5;
5     private String sortBy;
6     private Boolean isAsc = true;
7
8     public <T> Page<T> toMyPage(OrderItem ...items){
9         // 分页条件
10        Page<T> page = Page.of(pageNo,pageSize);
11        // 查询条件
12        if(StrUtil.isNotBlank(sortBy)){
13            page.addOrder(new OrderItem(sortBy,isAsc));
14        }else if(items != null){
15            page.addOrder(items);
16        }
17        return page;
18    }
19
20
21
22
23 }

```

将 `Page` 结果转换为 `PageDTO` 结果

```

1  public class PageDTO<T>{
2      // 总条数
3      private Integer total;
4      // 总页数
5      private Integer pages;
6      // 分页结果
7      private List<T> list;
8
9      public static <P0,V0> PageDTO<V0> of(Page<P0> page,Function<P0,V0> converter)
10     {
11         PageDTO<V0> dto = new PageDTO<>();
12         // 总条数
13         dto.setTotal(page.getTotal());
14         // 总页数
15         dto.setPages(page.getPages());
16         // 当前页数据
17         List<P0> records = page.getRecords();
18
19         if(CollUtil.isEmpty(records)){
20             dto.setList(Collections.emptyList());
21             return dto;
22         }
23         // 拷贝P0为V0
24         dto.setList(records.stream().map(converter).collect(Collectors.toList()));
25         return dto;
26     }
27 }

```

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

```

1  package com.hmall.cart;
2
3  import org.mybatis.spring.annotation.MapperScan;
4  import org.springframework.boot.SpringApplication;
5  import org.springframework.boot.autoconfigure.SpringBootApplication;
6  import org.springframework.context.annotation.Bean;
7  import org.springframework.web.client.RestTemplate;
8
9  @MapperScan("com.hmall.cart.mapper")
10 @SpringBootApplication
11 public class CartApplication {
12     public static void main(String[] args) {
13         SpringApplication.run(CartApplication.class, args);
14     }
15
16
17     @Bean
18     public RestTemplate restTemplate() {
19         return new RestTemplate();
20     }
21 }

```

- 发起远程调用

```

1  ResponseEntity<List<ItemDTO>> response = restTemplate.exchange(
2      "http://localhost:8081/items?ids={ids}",
3      HttpMethod.GET,
4      null,
5      new ParameterizedTypeReference<List<ItemDTO>>() {
6      },
7      Map.of("ids", CollUtil.join(itemIds, ","))
8  );
9  if(!response.getStatusCode().is2xxSuccessful()){
10      return;
11  }
12
13  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

心原理

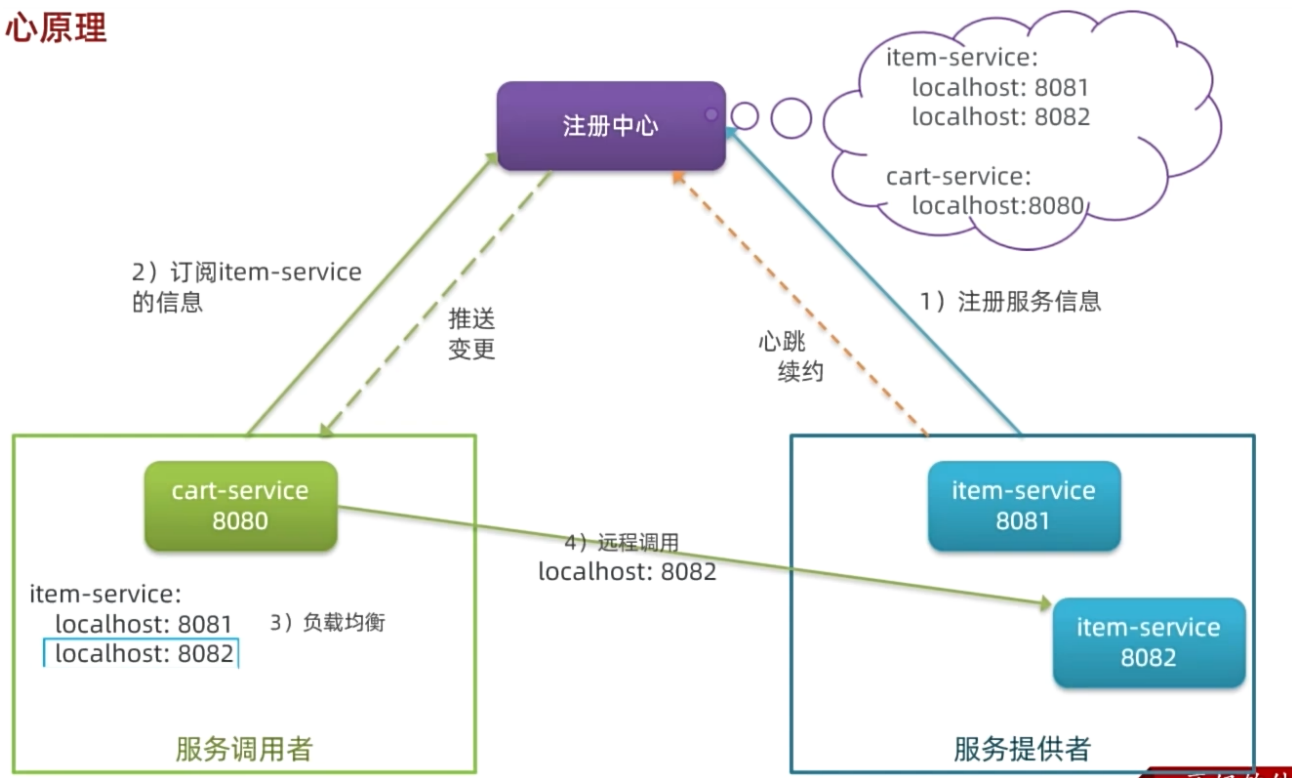


image-20251001133940462

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

```

1  <!--nacos 服务注册发现-->
2  <dependency>
3      <groupId>com.alibaba.cloud</groupId>
4      <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
5  </dependency>

```

```

1  spring:
2      application:
3          name: item-service # 服务名称
4      cloud:
5          nacos:
6              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

```
1 private final DiscoveryClient discoveryClient;  
2  
3 private void handleCartItems(List<CartV0> vos){  
4     // 1.根据服务名称，拉取服务的实例列表  
5     List<ServiceInstance> instances = discoveryClient.getInstances("item-  
service");  
6     // 2.负载均衡，挑选一个实例  
7     ServiceInstance instance =  
instances.get(RandomUtil.randomInt(instances.size()));  
8     // 3.获取实例的ip和端口  
9     URI uri = instance.getUri();  
10    // 4.略  
11 }
```

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`

```
1 <!--openFeign-->  
2 <dependency>  
3     <groupId>org.springframework.cloud</groupId>  
4     <artifactId>spring-cloud-starter-openfeign</artifactId>  
5 </dependency>  
6 <!--负载均衡器-->  
7 <dependency>  
8     <groupId>org.springframework.cloud</groupId>  
9     <artifactId>spring-cloud-starter-loadbalancer</artifactId>  
10 </dependency>
```

- 通过 `@EnableFeignClients` 注解，启用 `OpenFeign` 功能

```

1  @MapperScan("com.hmall.cart.mapper")
2  @EnableFeignClients
3  @SpringBootApplication
4  public class CartApplication {
5      public static void main(String[] args) {
6          SpringApplication.run(CartApplication.class, args);
7      }
8  }

```

- 编写 `FeignClient`

```

1  // 声明服务名
2  @FeignClient("item-service")
3  public interface ItemClient {
4      // 声明URL
5      @GetMapping("/items")
6      List<ItemDTO> queryItemByIds(@RequestParam("ids") Collection<Long> ids);
7  }

```

- 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

```

1  <!--OK http 的依赖 -->
2  <dependency>
3      <groupId>io.github.openfeign</groupId>
4      <artifactId>feign-okhttp</artifactId>
5  </dependency>

```

- 开启连接池

```
1 feign:
2   okhttp:
3     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

```
1 public class DefaultFeignConfig{
2     @Bean
3     public Logger.Level feignLogLevel(){
4         return Logger.Level.FULL;
5     }
6 }
```

- **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(defalutConfiguration = 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
- `Netfilx Zuul` ：阻塞式编程

网关

网关：就是网络的关口，负责请求的路由、转发、身份校验。

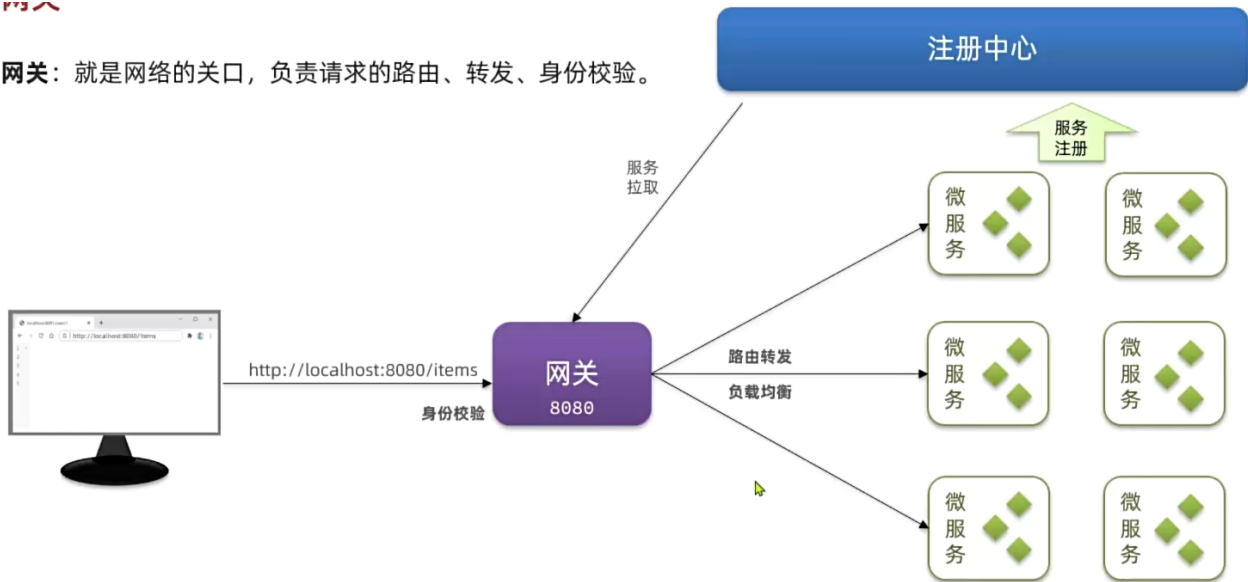


image-20251001152734999

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

```

1  spring:
2    application:
3      name: gateway
4    cloud:
5      nacos:
6        server-addr: 192.168.150.101:8848
7      gateway:
8        routes:
9          - id: item # 路由规则id, 自定义, 唯一
10            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
11            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
12              - Path=/items/**,/search/** # 这里是以请求路径作为判断规则
13          - id: cart
14            uri: lb://cart-service
15            predicates:
16              - 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

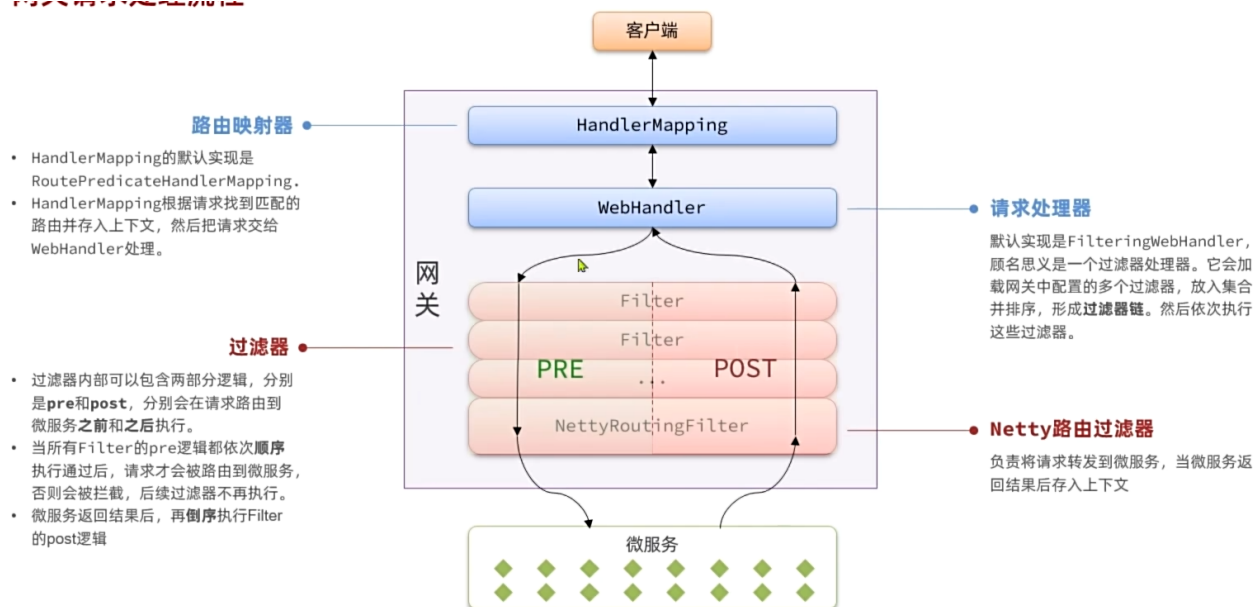


image-20251002135230563

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

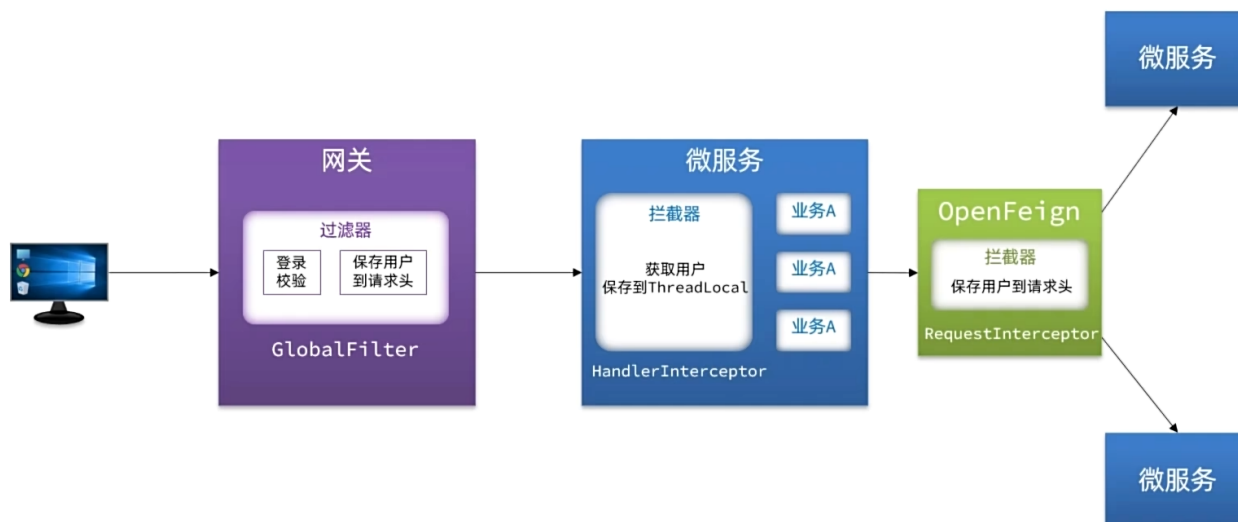


image-20251002152441612

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

- 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：

```
1  @Override
2  public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain
chain) {
3
4      // 1.获取request
5      ServerHttpRequest request = exchange.getRequest();
6      // 2.判断是否需要做登录拦截
7      if(isExclude(request.getPath().toString())){
8          return chain.filter(exchange);
9      }
10     // 3.获取登录token
11     String token = null;
12     List<String> headers = request.getHeaders().get("authorization");
13     if(headers != null && !headers.isEmpty()){
14         token = headers.get(0);
15     }
16     // 4.校验并解析token
17     Long userId = null;
18     try{
19         userId = jwtTool.parseToken(token);
20     }catch (UnauthorizedException e){
21         // 设置状态码为401
22         ServerHttpResponse response = exchange.getResponse();
23         response.setStatus(HttpStatus.UNAUTHORIZED);
24         return response.setComplete();
25     }
26
27     // TODO 5.传递用户信息
28     System.out.println("userId = " + userId);
29     // 6.放行
30     return chain.filter(exchange);
31 }
```

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

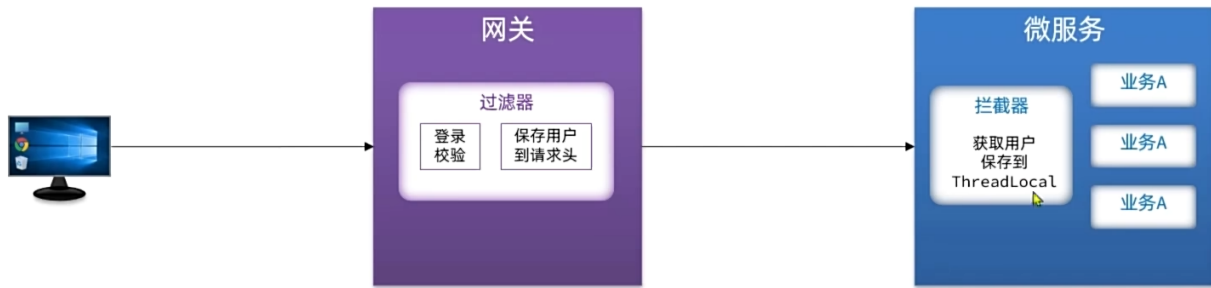


image-20251002143457177

- 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

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

image-20251002143735906

- 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
 - 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

- 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

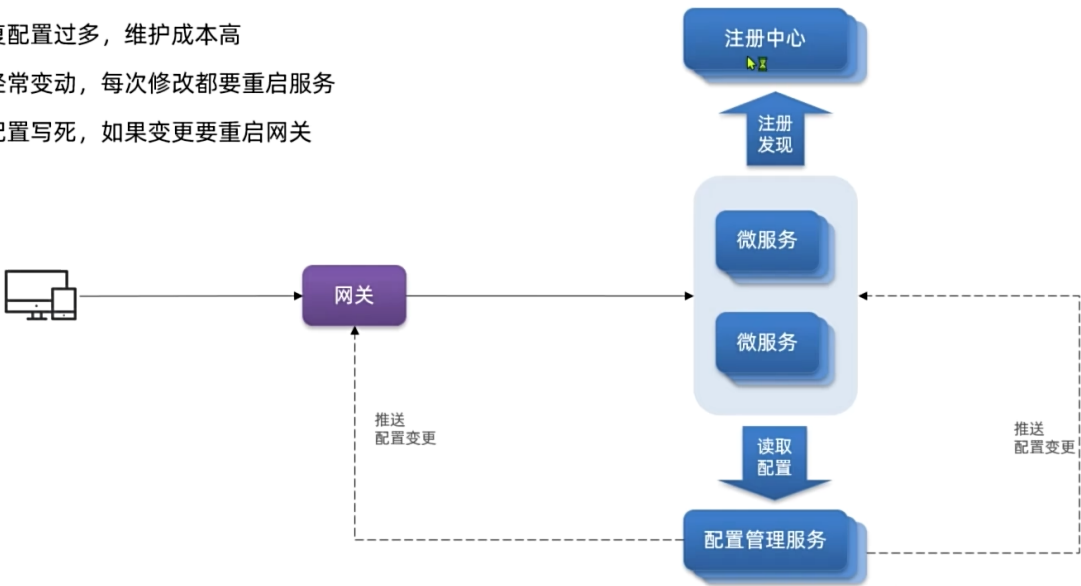


image-20251002153401193

配置共享

流程图如下：

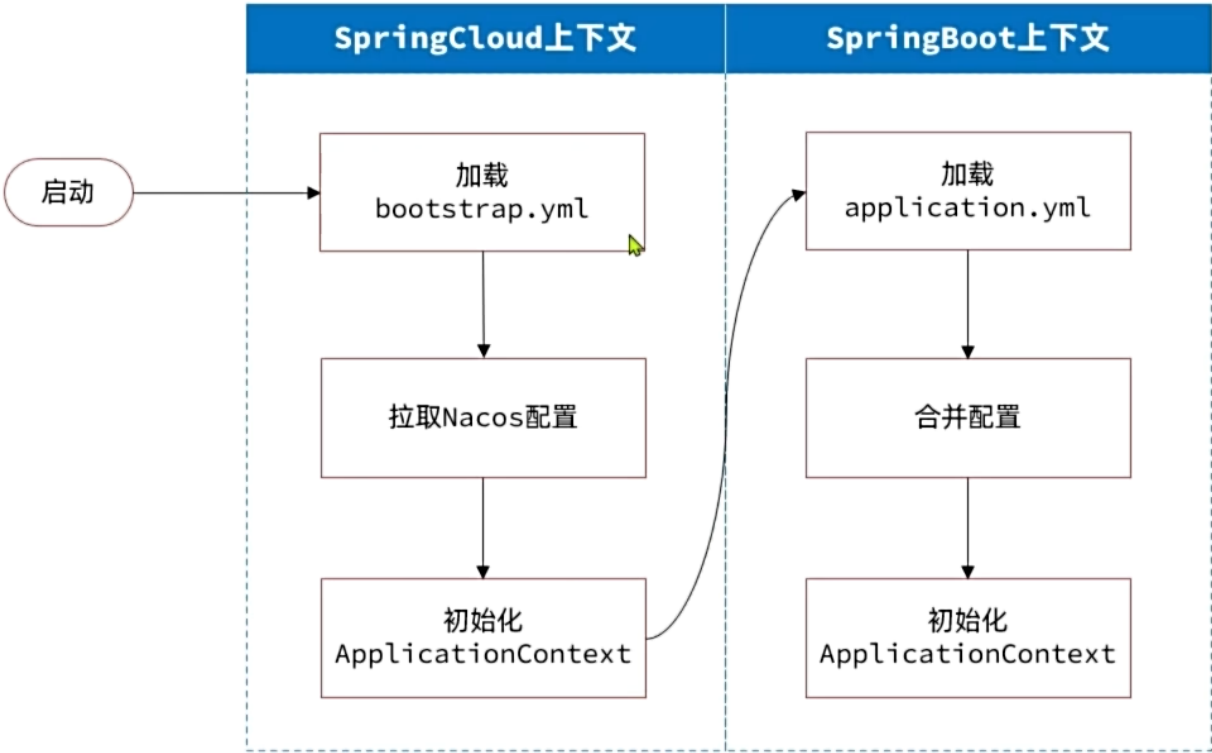


image-20251002154645476

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.yml` ，对微服务拉取nacos共享配置进行配置

- 引入依赖

```

1      <!--nacos配置管理-->
2      <dependency>
3          <groupId>com.alibaba.cloud</groupId>
4          <artifactId>spring-cloud-starter-alibaba-nacos-config</artifactId>
5      </dependency>
6      <!--读取bootstrap文件-->
7      <dependency>
8          <groupId>org.springframework.cloud</groupId>
9          <artifactId>spring-cloud-starter-bootstrap</artifactId>
10     </dependency>

```

- 创建 `bootstrap.yaml` 文件，下面是核心配置

```

1      config:
2          file-extension: yaml # 文件后缀名
3          shared-configs: # 共享配置
4              - dataId: shared-jdbc.yaml # 共享mybatis配置
5              - dataId: shared-log.yaml # 共享日志配置
6              - 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



image-20251003102041072

- 微服务中要以**特定方式**读取需要热更新的配置属性

```
@Data
@ConfigurationProperties(prefix = "hm.cart")
public class CartProperties {
    private int maxItems;
}
```

image-20251003102425170

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

```
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
}
```

image-20251003104452368

- 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
 - `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);
}
```

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

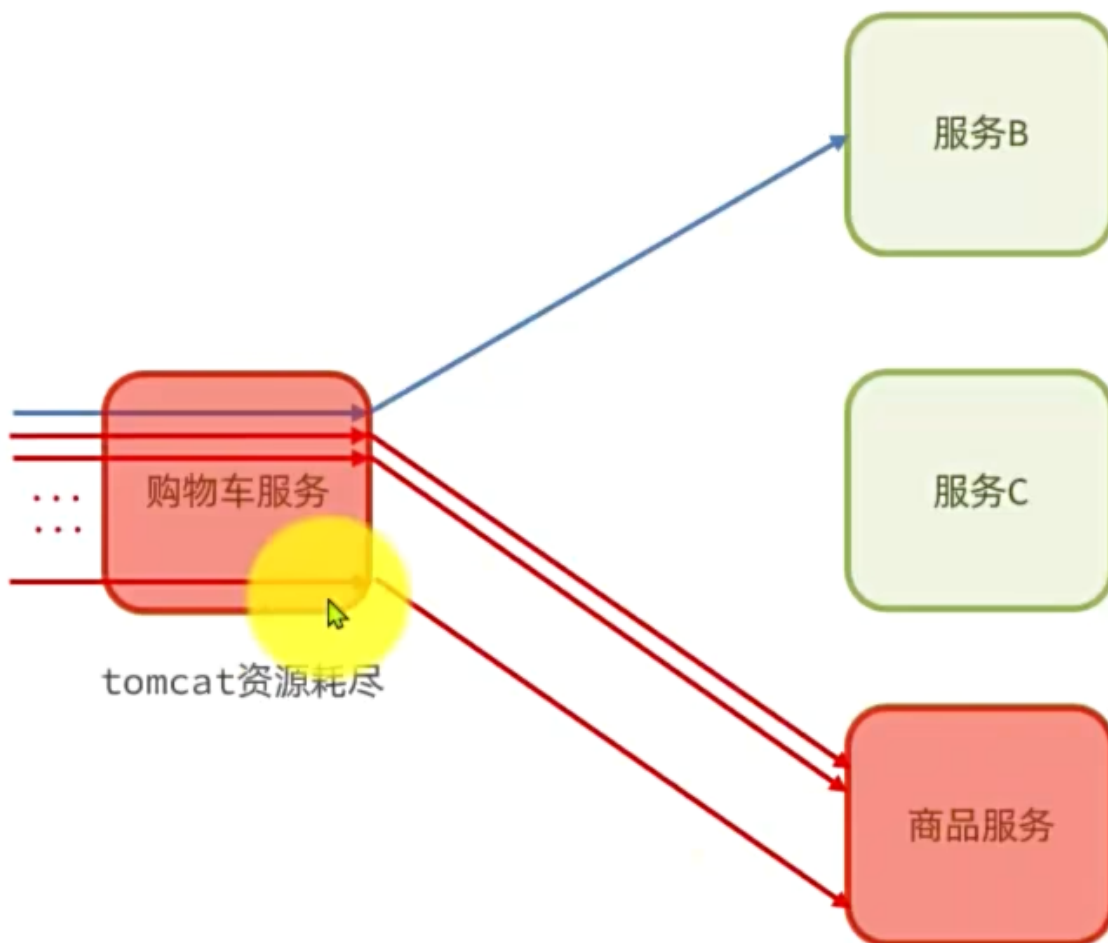


image-20251003113010570

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

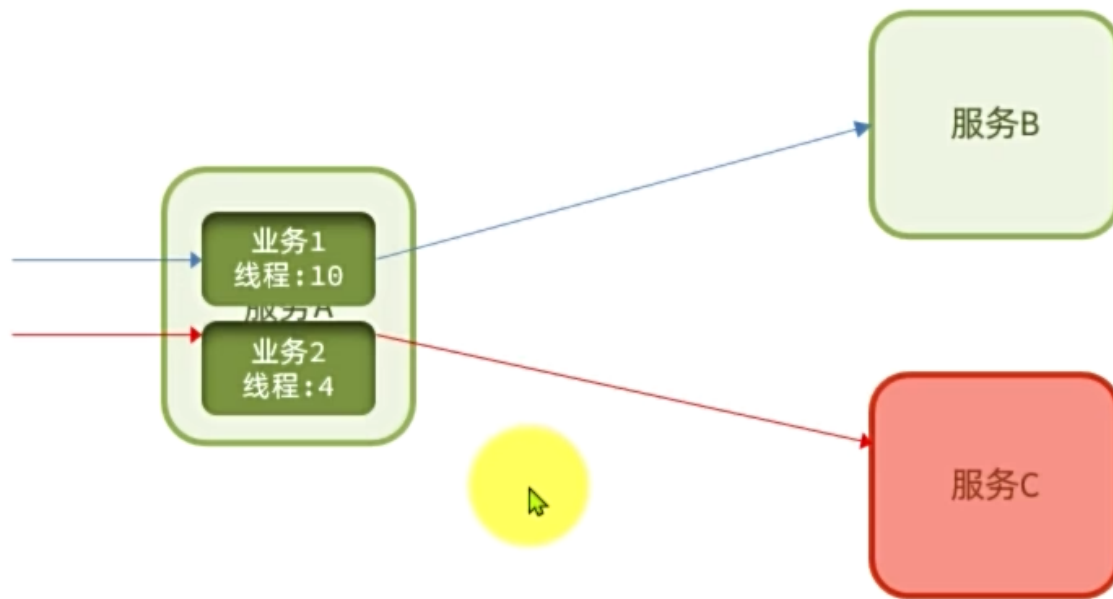


image-20251003114504227

- **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

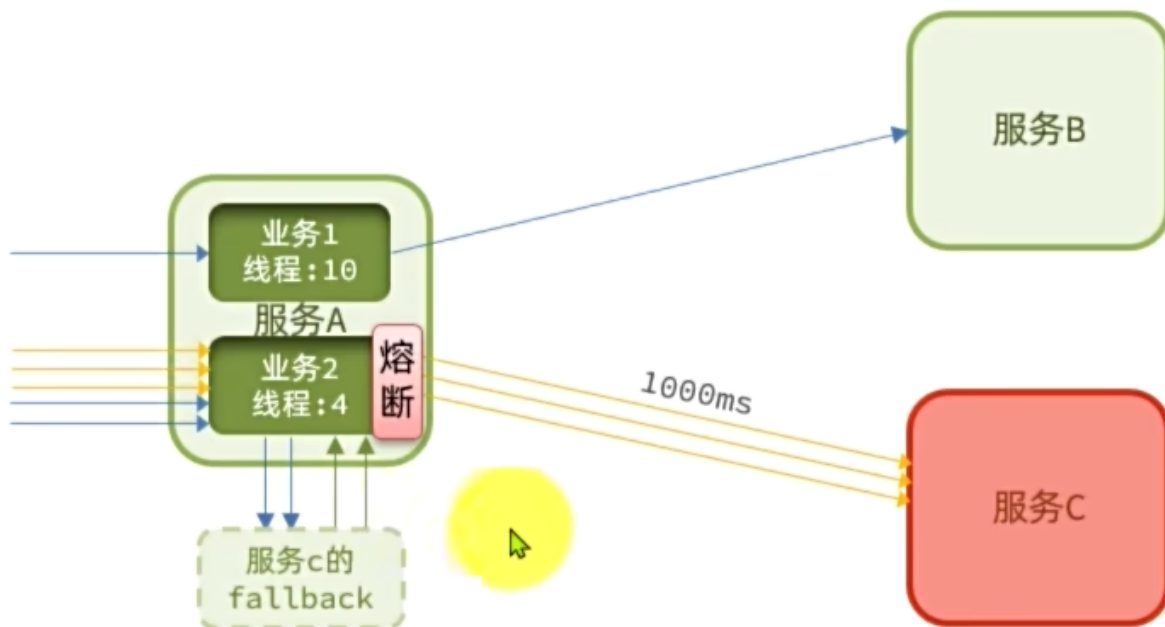


image-20251003115000482

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

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

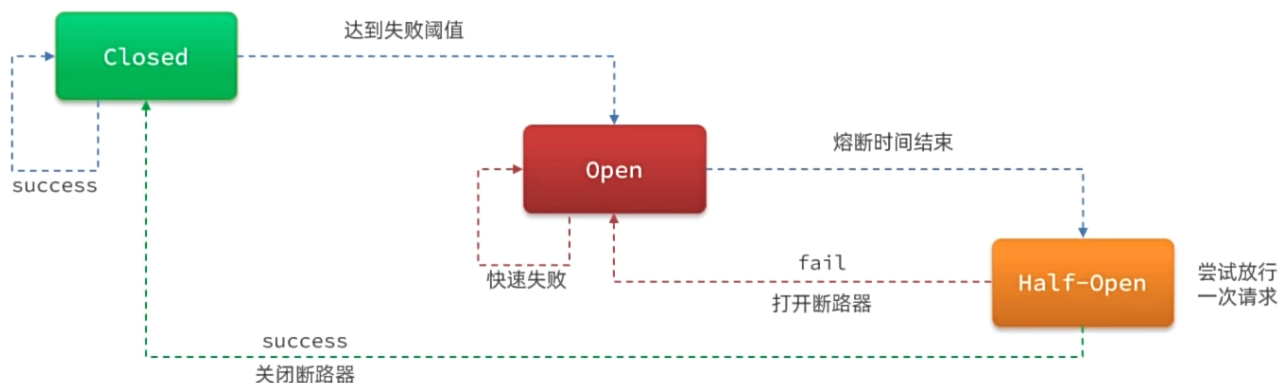
Fallback

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

```
1 feign:
2   sentinel:
3     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



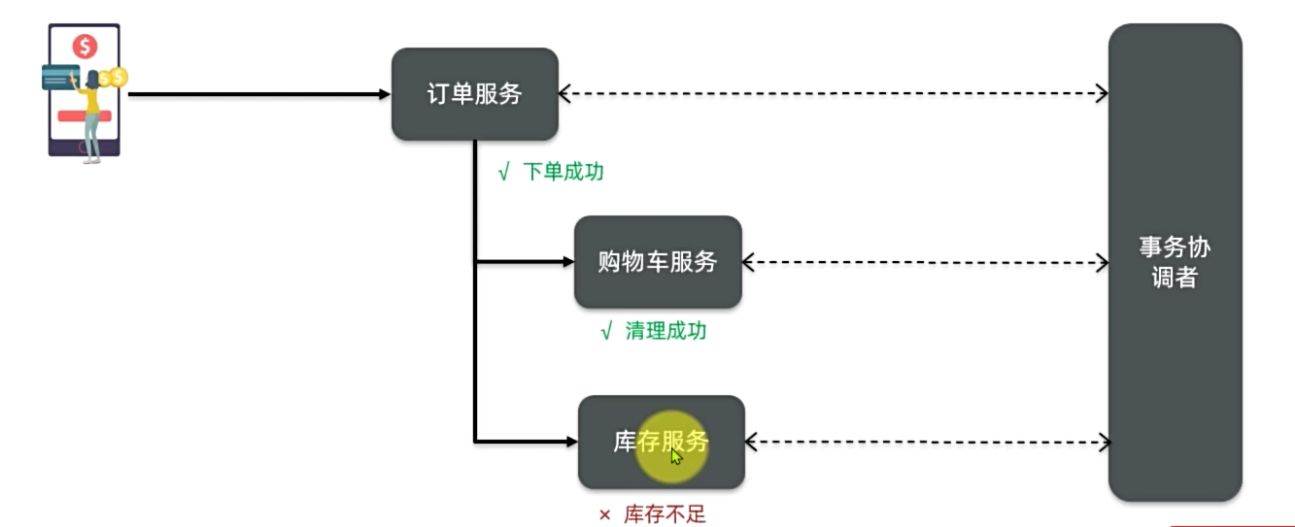
分布式事务

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**

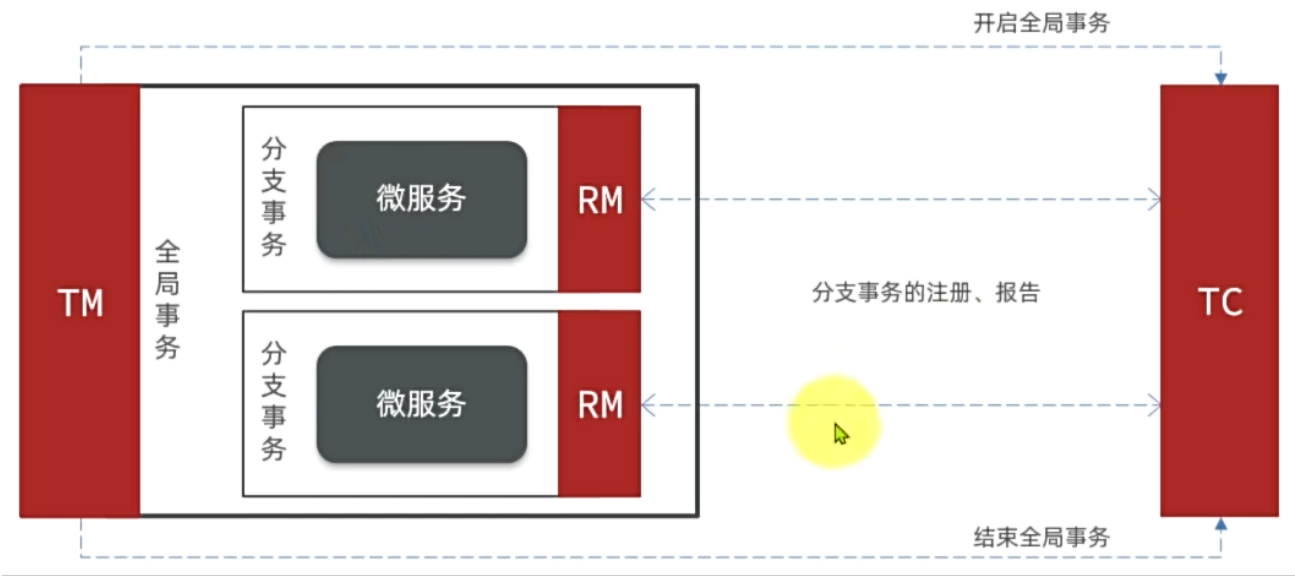
Seata

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



使用步骤

- 引入依赖

```
1  <!--统一配置管理-->
2  <dependency>
3      <groupId>com.alibaba.cloud</groupId>
4      <artifactId>spring-cloud-starter-alibaba-nacos-config</artifactId>
5  </dependency>
6  <!--读取bootstrap文件-->
7  <dependency>
8      <groupId>org.springframework.cloud</groupId>
9      <artifactId>spring-cloud-starter-bootstrap</artifactId>
10 </dependency>
11 <!--seata-->
12 <dependency>
13     <groupId>com.alibaba.cloud</groupId>
14     <artifactId>spring-cloud-starter-alibaba-seata</artifactId>
15 </dependency>
```

- 在Nacos注册中心配置TC服务地址

```
1  seata:
2      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
3          type: nacos # 注册中心类型 nacos
4          nacos:
5              server-addr: 172.21.172.16:8848 # nacos地址
6              namespace: "" # namespace，默认为空
7              group: DEFAULT_GROUP # 分组，默认是DEFAULT_GROUP
8              application: seata-server # seata服务名称
9              username: nacos
10             password: nacos
11 tx-service-group: hmall # 事务组名称
12 service:
13     vgroup-mapping: # 事务组与tc集群的映射关系
14         hmall: "default"
```



image-20251004110800127

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
 - 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

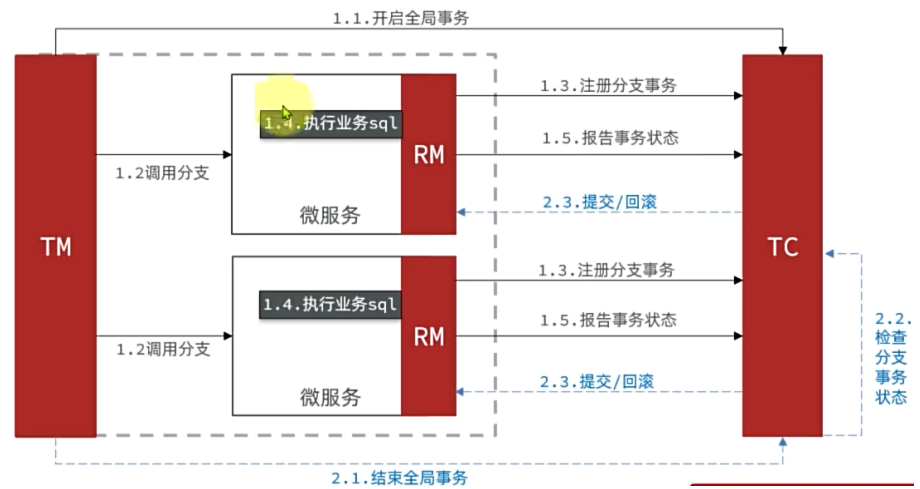


image-20251004112448684

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

```
1 seata:
2   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

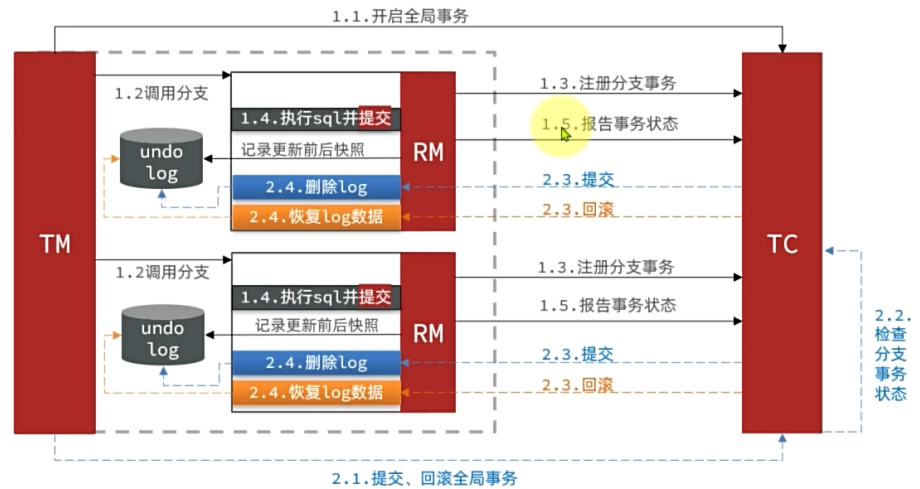


image-20251004114046322

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

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

Elasticsearch

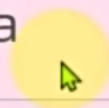
是基于Lucene开发的**高性能分布式搜索引擎**

- 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 `klibana`、`Logstash`、`Beats`，是一整套技术栈，被称为 **ELK**

数据可视化



Kibana



存储、计算
、搜索数据



Elasticsearch

数据抓取



Logstash



Beats

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）：文档安装语义分成的词语

id	title	price
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



词条 (term)	文档id
小米	1 , 3 , 4
手机	1 , 2
华为	2 , 3
充电器	3
手环	4

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为 `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

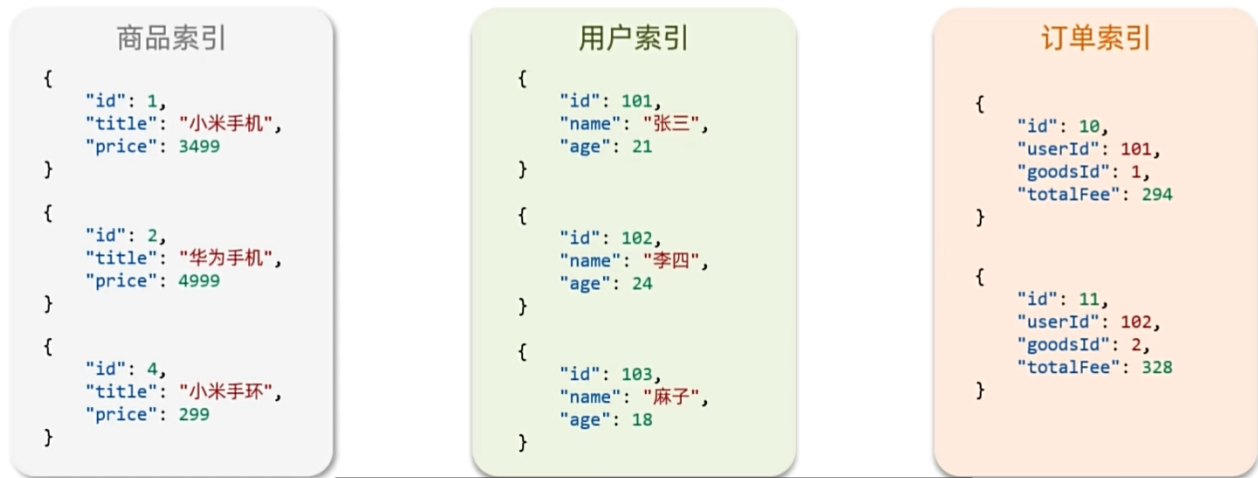


image-20251007133546860

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风格请求语句, 用来定义搜索条件

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：

```
1 PUT /索引库名称
2 {
3   "mappings": {
4     "properties": {
5       "字段名": {
6         "type": "text",
7         "analyzer": "ik_smart"
8       },
9       "字段名2": {
10        "type": "keyword",
11        "index": "false"
12      },
13      "字段名3": {
14        "properties": {
15          "子字段": {
16            "type": "keyword"
17          }
18        }
19      },
20      // ...略
21    }
22  }
23 }
```

- 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

```
1 PUT /索引库名/_mapping
2 {
3   "properties": {
4     "新字段名": {
5       "type": "integer"
6     }
7   }
8 }
```

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：

```
1  POST /索引库名/_doc/文档id
2  {
3      "字段1": "值1",
4      "字段2": "值2",
5      "字段3": {
6          "子属性1": "值3",
7          "子属性2": "值4"
8      },
9      // ...
10 }
```

- 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

```

1 PUT /{索引库名}/_doc/文档id
2 {
3     "字段1": "值1",
4     "字段2": "值2",
5     // ... 略
6 }

```

- **局部修改**：只修改指定id匹配的文档中的部分字段

```

1 POST /{索引库名}/_update/文档id
2 {
3     "doc": {
4         "字段名": "新的值",
5     }
6 }

```

• 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

```

1 POST _bulk
2 { "index" : { "_index" : "test", "_id" : "1" } }
3 { "field1" : "value1" }
4 { "delete" : { "_index" : "test", "_id" : "2" } }
5 { "create" : { "_index" : "test", "_id" : "3" } }
6 { "field1" : "value3" }
7 { "update" : { "_id" : "1", "_index" : "test" } }
8 { "doc" : { "field2" : "value2" } }

```

JavaRestClient

初始化步骤：

- 引入依赖

```
1 <dependency>
2   <groupId>org.elasticsearch.client</groupId>
3   <artifactId>elasticsearch-rest-high-level-client</artifactId>
4 </dependency>
```

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

```
1 RestHighLevelClient restHighLevelClient = new
  RestHighLevelClient(RestClient.builder(
2     HttpHost.create("172.21.172.16:9200")
3     ));
```

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

```
1 @Test
2 void testCreateIndex() throws IOException {
3     // 1.创建Request对象
4     CreateIndexRequest request = new CreateIndexRequest("items");
5     // 2.准备请求参数
6     request.source(MAPPING_TEMPLATE, XContentType.JSON);
7     // 3.发送请求
8     client.indices().create(request, RequestOptions.DEFAULT);
9 }
```

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

```

1  @Test
2  void testDeleteIndex() throws IOException {
3      // 1.创建Request对象
4      DeleteIndexRequest request = new DeleteIndexRequest("items");
5      // 2.发送请求
6      client.indices().delete(request, RequestOptions.DEFAULT);
7  }

```

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

```

1  @Test
2  void testExistsIndex() throws IOException {
3      // 1.创建Request对象
4      GetIndexRequest request = new GetIndexRequest("items");
5      // 2.发送请求
6      client.indices().get(request, RequestOptions.DEFAULT);
7  }

```

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**

```

1  @Test
2  void testIndexDoc() throws IOException {
3      // 准备文档数据
4      Item item = iItemService.getById(100000011127L);
5      ItemDoc itemDoc = BeanUtil.copyProperties(item, ItemDoc.class);
6
7
8      IndexRequest request = new IndexRequest("items").id(itemDoc.getId());
9      request.source(JSONUtil.toJsonStr(itemDoc), XContentType.JSON);
10     restHighLevelClient.index(request, RequestOptions.DEFAULT);
11 }

```

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

```
1  @Test
2  void testDeleteDocument() throws IOException {
3      // 1.准备Request，两个参数，第一个是索引库名，第二个是文档id
4      DeleteRequest request = new DeleteRequest("item", "100002644680");
5      // 2.发送请求
6      client.delete(request, RequestOptions.DEFAULT);
7  }
```

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
  }
}
```

image-20251007151903429

```

1  @Test
2  void testGetDocumentById() throws IOException {
3      // 1.准备Request对象
4      GetRequest request = new GetRequest("items").id("100002644680");
5      // 2.发送请求
6      GetResponse response = client.get(request, RequestOptions.DEFAULT);
7      // 3.获取响应结果中的source
8      String json = response.getSourceAsString();
9
10     ItemDoc itemDoc = JSONUtil.toBean(json, ItemDoc.class);
11     System.out.println("itemDoc= " + ItemDoc);
12 }

```

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

```

1  @Test
2  void testUpdateDocument() throws IOException {
3      // 1.准备Request
4      UpdateRequest request = new UpdateRequest("items", "100002644680");
5      // 2.准备请求参数
6      request.doc(
7          "price", 58800,
8          "commentCount", 1
9      );
10     // 3.发送请求
11     client.update(request, RequestOptions.DEFAULT);
12 }

```

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

```

1  @Test
2  void testBulk() throws IOException {
3      // 1.创建Request
4      BulkRequest request = new BulkRequest();
5      // 2.准备请求参数
6      request.add(new IndexRequest("items").id("1").source("json doc1",
XContentType.JSON));
7      request.add(new IndexRequest("items").id("2").source("json doc2",
XContentType.JSON));
8      // 3.发送请求
9      client.bulk(request, RequestOptions.DEFAULT);
10 }

```

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

```

1  GET /{索引库名}/_search
2  {
3      "query": {
4          "查询类型": {
5              // .. 查询条件
6          }
7      }
8  }

```

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

```

1 GET /items/_search
2 {
3   "query": {
4     "match_all": {
5
6     }
7   }
8 }
```

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

```

1 GET /{索引库名}/_search
2 {
3   "query": {
4     "match": {
5       "字段名": "搜索条件"
6     }
7   }
8 }
```

multi_match查询：与match查询类似，不过允许同时查询多个字段

```
1 GET /{索引库名}/_search
2 {
3   "query": {
4     "multi_match": {
5       "query": "搜索条件",
6       "fields": ["字段1", "字段2"]
7     }
8   }
9 }
```

term查询

```
1 GET /{索引库名}/_search
2 {
3   "query": {
4     "term": {
5       "字段名": {
6         "value": "搜索条件"
7       }
8     }
9   }
10 }
```

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

```
1 GET /{索引库名}/_search
2 {
3   "query": {
4     "range": {
5       "字段名": {
6         "gte": {最小值},
7         "lte": {最大值}
8       }
9     }
10   }
11 }
```

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

```

1  GET /items/_search
2  {
3    "query": {
4      "bool": {
5        "must": [
6          {"match": {"name": "手机"}}
7        ],
8        "should": [
9          {"term": {"brand": { "value": "vivo" }}}},
10         {"term": {"brand": { "value": "小米" }}}
11       ],
12       "must_not": [
13         {"range": {"price": {"gte": 2500}}}
14       ],
15       "filter": [
16         {"range": {"price": {"lte": 1000}}}
17       ]
18     }
19   }
20 }
```

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

```
1 GET /indexName/_search
2 {
3   "query": {
4     "match_all": {}
5   },
6   "sort": [
7     {
8       "排序字段1": {
9         "order": "排序方式asc和desc"
10      }
11    },
12    {
13      "排序字段2": {
14        "order": "排序方式asc和desc"
15      }
16    }
17  ]
18 }
```

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

```
1 GET /items/_search
2 {
3   "query": {
4     "match_all": {}
5   },
6   "from": 0, // 分页开始的位置，默认为0
7   "size": 10, // 每页文档数量，默认10
8   "sort": [
9     {
10      "price": {
11        "order": "desc"
12      }
13    }
14  ]
15 }
```

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`

```
1  GET /{索引库名}/_search
2  {
3    "query": {
4      "match": {
5        "搜索字段": "搜索关键字"
6      }
7    },
8    "highlight": {
9      "fields": {
10        "高亮字段名称": {
11          "pre_tags": "<em>",
12          "post_tags": "</em>"
13        }
14      }
15    }
16  }
```

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

```
1 GET /items/_search
2 {
3   "query": {
4     "match_all": {}
5   }, //可以省略
6   "size": 0, //设置size为0, 结果中不包含文档, 只包含聚合结果
7   "aggs": { //定义聚合
8     "cateAgg": { //给聚合起个名字
9       "terms": { //聚合的类型, 按照品牌值聚合, 所以选择term
10         "field": "category", //参与聚合的字段
11         "size": 20 // 希望获取的聚合结果数量
12       }
13     }0
14   }
15 }
```

- **度量聚合**

- 使用嵌套聚合获取每个品牌的信息

```

1  GET /items/_search
2  {
3      "query": {
4          "bool": {
5              "filter": [
6                  {
7                      "term": {
8                          "category": "手机"
9                      }
10                 },
11                 {
12                     "range": {
13                         "price": {
14                             "gte": 300000
15                         }
16                     }
17                 }
18             ]
19         }
20     },
21     "size": 0,
22     "aggs": {
23         "brand_agg": {
24             "terms": {
25                 "field": "brand",
26                 "size": 20
27             },
28             "aggs": {                // 嵌套聚合
29                 "stats_meric": {
30                     "stats": {
31                         "field": "price"
32                     }
33                 }
34             }
35         }
36     }
37 }

```

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

```

1  @Test
2  void testMatchAll() throws IOException{
3      // 1.准备request
4      SearchRequest request = new SearchRequest();
5      // 2.构建DSL参数
6      request.source()
7          .query(QueryBuilder().matchAllQuery());
8      // 3.发送请求
9      client.search(request,Req)
10 }

```

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

```

1  // 单字段查询
2  QueryBuilders.MatchQuery("name", "脱脂牛奶");
3  // 多字段查询
4  QueryBuilders.multiMatchQuery("脱脂牛奶", "name", "category");

```

```

// 单字段查询
QueryBuilders.matchQuery("name", "脱脂牛奶");
// 多字段查询
QueryBuilders.multiMatchQuery("脱脂牛奶", "name", "category");

```

```

GET /items/_search
{
  "query": {
    "match": {
      "name": "脱脂牛奶"
    }
  }
}
GET /items/_search
{
  "query": {
    "multi_match": {
      "query": "脱脂牛奶",
      "fields": ["category", "name"]
    }
  }
}

```

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

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

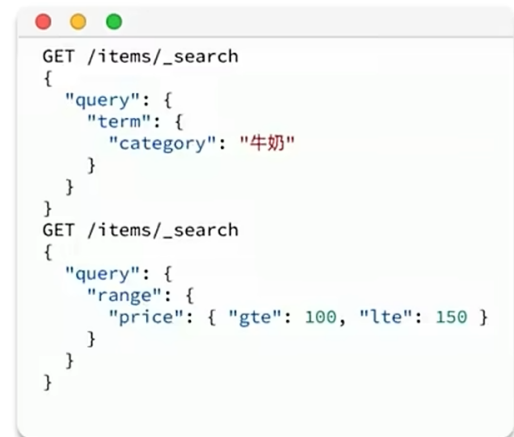


image-20251008134635397

- 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

```
1 // 创建布尔查询
2 BoolQueryBuilder boolQuery = QueryBuilders.boolQuery();
3 // 添加must条件
4 boolQuery.must(
5     QueryBuilders.termQuery("brand", "华为"));
6 // 添加should条件
7 boolQuery.should(
8     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

```
1 // 分页
2 request.source().from(0).size(5);
3 // 价格排序
4 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

```
1 request.source().highlighter(
2     SearchSourceBuilder.highlight()
3         .field("name")
4         .preTags("<em>")
5         .postTags("</em>")
6 );
```

- 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

```
1 private static void parseResponseResult(SearchResponse search) {
2     SearchHits hits = search.getHits();
3
4     long total = hits.getTotalHits().value;
5
6     SearchHit[] searchHits = hits.getHits();
7
8     for (SearchHit searchHit : searchHits) {
9         String json = searchHit.getSourceAsString();
10
11
12         // 处理高亮
13         Map<String, HighlightField> hfs = searchHit.getHighlightFields();
14         if(hfs!=null && !hfs.isEmpty()){
15             HighlightField hf = hfs.get("name");
16             json = hf.getFragments()[0].string();
17         }
18         System.out.println("json"+json);
19     }
20 }
```

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

```
1 request.source().aggregation(  
2     AggregationBuilders  
3     .terms("brand_agg")    // 指定聚合名称  
4     .field("brand")        // 指定聚合字段  
5     .size(20)              // 指定返回的类别数  
6 );
```

- 解析的时候根据查询结果一点一点推

```
1 @Test  
2 void testAgg() throws IOException {  
3  
4     SearchRequest request = new SearchRequest("items");  
5  
6  
7     request.source().size(0);  
8  
9     String brandAggName = "brandAgg";  
10    request.source().aggregation(  
11        AggregationBuilders  
12            .terms(brandAggName)  
13            .field("brand")  
14            .size(10)  
15    );  
16    SearchResponse search = restHighLevelClient.search(request,  
RequestOptions.DEFAULT);  
17    System.out.println("search results:"+search);  
18  
19  
20    Aggregations aggregations = search.getAggregations();  
21  
22    Terms brandTerms = aggregations.get(brandAggName);  
23  
24    List<? extends Terms.Bucket> buckets = brandTerms.getBuckets();  
25  
26    for (Terms.Bucket bucket : buckets) {  
27        System.out.println("brand:"+bucket.getKey());  
28        System.out.println("count:"+bucket.getDocCount());  
29    }  
30 }
```

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**

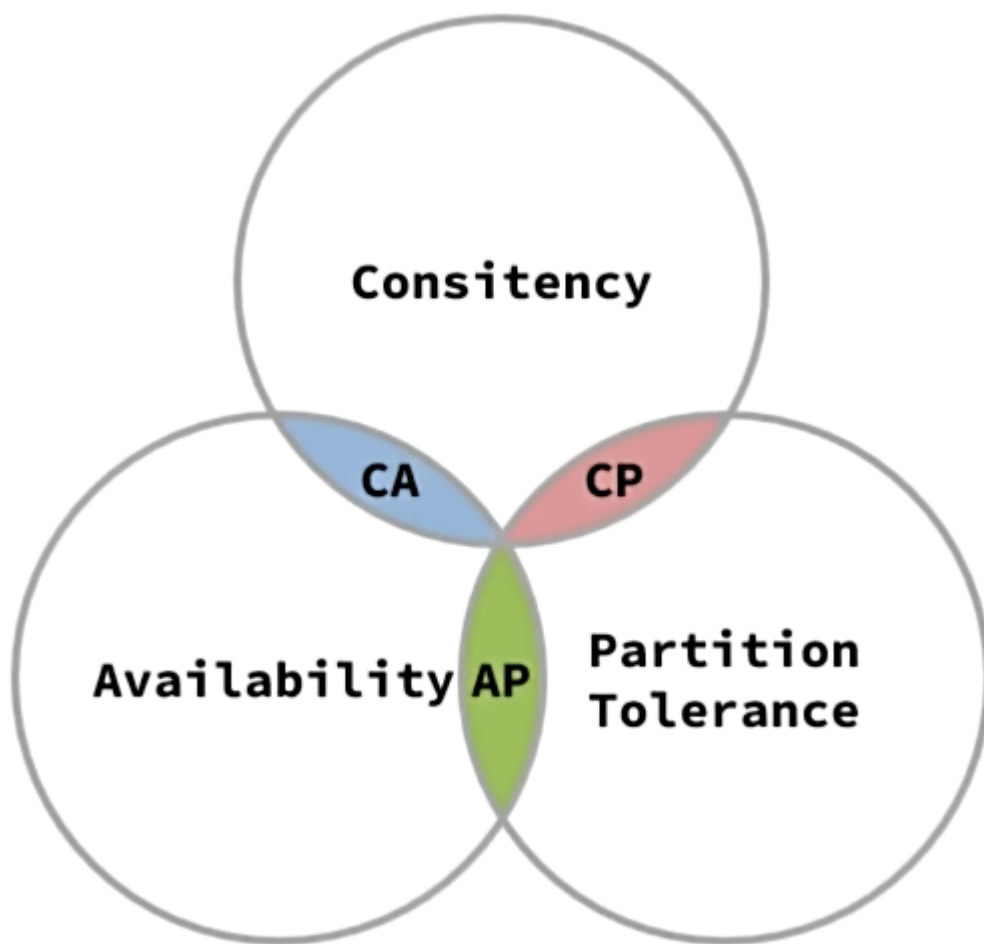


image-20251008145844622

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。

AT模式的脏写问题