

MybatisPlus

Mybatis 的增强版

特点

- 简化 Mapper 接口 MybatisPlus 的 BaseMapper 接口

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

- BaseMapper 提供了 CRUD 操作
- MybatisPlus 提供了分页插件

配置

- 配置数据源
 - User -> user
 - UserInfo -> user_info
- 配置 id 生成策略
- 配置分页插件

注解

- @TableName 表名
- @TableId 主键
 - values 值
 - type 类型 ASSIGN_ID
 - IdType.AUTO 自动
 - IdType.INPUT 手动 set
 - IdType.ASSIGN_ID 指定
- @TableField 字段
 - values 值
 - 配置 @TableField 属性
 - 是否必填
 - 是否 is 属性
 - 是否带下划线 `
 - 是否存在 exist = false

```
@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;
    }

```

测试

Mybatis 测试 where

Wrapper 是 Mybatis 中 SQL 语句的 where 子句

- QueryWrapper 是 Wrapper 的子类
 - .select() 指定要查询的列
 - .like() 指定要查询的列
 - .ge() 指定要查询的列
- UpdateWrapper 是 Wrapper 的子类 where 子句 set 子句

测试

```

// 测试Mapper
public interface UserMapper extends BaseMapper<User>{

}

@Autowired
private UserMapper userMapper;

void testQueryMapper(){
    // 1. 创建QueryWrapper
    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. 创建User
    User user = new User();
    user.setBalance(2000);
    // 2. 创建QueryWrapper
    QueryWrapper<User> wrapper = new QueryWrapper<User>.eq("username","jack");
    // 3. 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语句
}

```

```

        .in("id",ids);

        // 更新数据
        userMapper.update(null,wrapper);
    }

```

Lambda

- 使用 function
 - User::getUsername 使用

```

void testLambdaQueryMapper(){
    // 1. 创建 wrapper
    LambdaQueryWrapper<User> wrapper = new LambdaQueryWrapper<User>()
        .select(User::getId,User::getUsername,User::getInfo,User::getBalance) // 选择
        // 过滤
        .like(User::getUsername,"o")
        .ge(User::getBalance,1000);

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

```

SQL

在 Mp 中 Wrapper 使用 where 方法生成 SQL

使用 Mapper 生成 SQL 语句，通过 Service 生成 SQL

1. 使用 Wrapper 生成 where 语句

```

List<long> ids = List.of(1L,2L,4L);
int amount = 200;
// 1. 创建 wrapper
LambdaQueryWrapper<User> wrapper = new LambdaQueryWrapper<User>
().in(User::getId,ids);

// 2. 生成 SQL 语句
userMapper.updateBalanceByIds(wrapper,amount);

```

2. 在 mapper 中使用 Param 方法生成 wrapper 语句

```

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

```

3. 生成 SQL 语句 Wrapper

```

<update id="updateBalanceByIds">
    UPDATE tb_user

```

```
        SET balance = balance - #{amount}
        ${ew.customSqlSegment} <!--Mp自定义SQL-->
    </update>
```

Service

定义 IService

- save
- remove
- update
- get

实现 Service 和 Mapper 接口 service 接口

实现

- 实现 IService 接口
- 实现 ServiceImpl 接口
- 实现 Service 接口

image-20250831160832309

```
// 定义
// 定义UserMapper接口
@Service
public class UserServiceImpl extends ServiceImpl<UserMapper,User> implements
UserService{

}
// 实现
public interface UserService extends IService<User>{

}
// 注解
@Autowired
private UserService userService;
// 定义userService
```

实现

- DTO
- VO
- PO

实现

- @RequiredArgsConstructor
- final

```
@RequestMapping("/users")
@RequiredArgsConstructor
```

```

public class UserController{
    private final IUserService userService;

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

```

IService Lambda

조건을 지정하여 쿼리 실행

IService 인터페이스

- `lambdaQuery()` 메서드
 - 조건을 지정하여 `condition` 메서드를 호출
 - `.list()` 또는 `.one()` 메서드를 호출
 - `query()` 메서드를 호출
- `lambdaUpadte()` 메서드
 - `.update()` 메서드를 호출
 - `.update()` 메서드를 호출

```

// 쿼리
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();

// 업데이트
lambdaUpdate()
    .set(User::getBalance, remainBalance)
    .set(remainBalance == 0, User::getStatus, 2)
    .eq(User::getId, id)
    .eq(User::getBalance, user.getBalance()) // 쿼리
    .update();

```

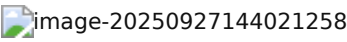
IService 구현

- `Mp` 메서드
 - `1k` 메서드를 호출
- `Mysql` 메서드
 - `rewriteBatchedStatements=true` 옵션
 - `sql` 메서드를 호출

테스트

1.1.2 MybatisPlus 1.0.0 版本

- 1.0.0 版本
- 1.0.0 版本
- 1.0.0 版本
- 1.0.0 版本



1.1.2

1.1.2

1.1.2



1.1.2

MP 1.0.0 版本

1.1.2 application.yaml 1.0.0 版本

```
mybatis-plus:
  global-config:
    db-config:
      logic-delete-field: flag # 逻辑删除标志
      logic-delete-value: 1 # 逻辑删除标志值1
      logic-not-delete-value: 0 # 逻辑删除标志值0
```

1.1.2

User 1.0.0 版本

```
private Integer status;
```

1.1.2 Integer 1.0.0 版本

1.1.2

- 1.1.2 @EnumValue 1.0.0 MP 1.0.0 版本
- 1.1.2 @JsonValue 1.0.0 SpringMVC 1.0.0 版本

```
@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;
    }
}

```

配置枚举处理器

```

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

```

JSON

使用mybatis-plus的json处理器java

配置

- 在配置类中配置 `autoResultMap = true`
- 在配置类中配置 `@TableField` 使用 `JacksonTypeHandler.class`

```

@Data
@Table(name = "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;
}

```

配置

配置mybatis-plus的MP配置类

```

@Configuration
public class MybatisConfig{

    @Bean
    public MybatisPlusInterceptor mybatisPlusInterceptor(){

```

```

        // 拦截器
        MybatisPlusInterceptor interceptor = new MybatisPlusInterceptor();
        // 分页拦截器
        PaginationInnerInterceptor pageInterceptor = new
        PaginationInnerInterceptor(DbType.MYSQL);
        pageInterceptor.setMaxLimit(1000L); // 设置最大限制
        interceptor.addInnerInterceptor(pageInterceptor);
        return interceptor;
    }
}

```

测试

- 测试分页功能
- 测试** Page **
 - MPPage
- 测试 .addOrder()
 - new OrderItem("测试", true)
 - true false
- .getTotal() 测试
- .getPages() 测试
- .getRecords() 测试

```

@Test
void testPageQuery() {
    // 1. 测试
    int pageNo = 1, pageSize = 5;

    // 1.1 测试
    Page<User> page = Page.of(pageNo, pageSize);

    // 1.2 测试balance
    page.addOrder(new OrderItem("balance", false));

    // 1.3 测试
    Page<User> p = userService.page(page);

    // 2. 测试
    System.out.println("total=" + p.getTotal());

    // 3. 测试
    System.out.println("pages=" + p.getPages());

    // 4. 测试
    List<User> records = p.getRecords();
    records.forEach(System.out::println);
}

```

测试

测试


```

@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

```

public class PageDTO<T>{
    // 总数
    private Integer total;
    // 页数
    private Integer pages;
    // 列表
    private List<T> list;
}

```

1 PageQuery 2 MP 3 Page 4

```

@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
        Page<T> page = Page.of(pageNo,pageSize);
        // 排序
        if(StrUtil.isNotBlank(sortBy)){
            page.addOrder(new OrderItem(sortBy,isAsc));
        }else if(items != null){
            page.addOrder(items);
        }
        return page;
    }
}

```

```
}
```

❑ Page 转换为 PageDTO

```
public class PageDTO<T>{
    // 总数
    private Integer total;
    // 页数
    private Integer pages;
    // 数据
    private List<T> list;

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

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

❑

❑

- 项目
- Maven

❑

Spring RestTemplate 使用 Http

- RestTemplate 使用

```
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();
    }
}
```

- 测试接口


```
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();
```

测试

测试接口

- 测试接口
 - 测试接口
- 测试接口
 - 测试接口

 image-20251001133940462

Nacos测试

测试接口

测试

- 测试docker测试接口 ip:8848/nacos 测试接口nacos

- 在 pom.xml 文件中添加 yml 配置

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

```
spring:
  application:
    name: item-service # 应用名
  cloud:
    nacos:
      server-addr: 192.168.150.101:8848 # nacos 地址
```

测试

测试 nacos 是否配置成功

- 在 nacos 中
- 在 nacos 中
- 测试
 - 在 DiscoveryClient 中 Spring 配置

```
private final DiscoveryClient discoveryClient;

private void handleCartItems(List<CartV0> vos){
    // 1. 获取实例
    List<ServiceInstance> instances = discoveryClient.getInstances("item-
service");
    // 2. 获取实例
    ServiceInstance instance =
instances.get(RandomUtil.randomInt(instances.size()));
    // 3. 获取 ip
    URI uri = instance.getUri();
    // 4.
}
```

OpenFeign

测试 http 接口 Spring MVC 测试 http 接口

测试

- 测试
 - 在 pom.xml 中 ** loadbalancer **

```

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

```

- 添加 @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);
    }
}

```

- 添加 FeignClient

```

// 接口定义
@FeignClient("item-service")
public interface ItemClient {
    // 接口URL
    @GetMapping("/items")
    List<ItemDTO> queryItemByIds(@RequestParam("ids") Collection<Long> ids);
}

```

- 添加 FeignClient 注解

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

接口

OpenFeign 支持 Http 客户端实现，目前支持以下 Http 客户端

- HttpURLConnection 原生实现
- Apache HttpClient 实现
- OKHttp 实现

目前默认使用 OKHttp

使用 OKHttp 实现

- 配置

```

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

```

- 配置

```

feign:
  okhttp:
    enabled: true # 开启OkHttp

```

配置

配置项名称与配置项值

配置项名称 FeignClient 配置项值 SpringBootApplication 配置项名称 FeignClient 配置项值

- 配置 FeignClient 配置

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

- 配置 FeignClient 配置

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

配置

OpenFeign 配置 FeignClient 配置项名称 DEBUG 配置项值 4

- **NONE** 配置项名称 配置项值
- **BASIC** 配置项名称 URL 配置项值
- **HEADERS** 配置项名称 BASIC 配置项值
- **FULL** 配置项名称 配置项值

配置 Feign 配置项名称 NONE 配置项值

配置项名称 Logger.Level 配置项值 Bean 配置项值

```

public class DefaultFeignConfig{
  @Bean
  public Logger.Level feignLogLevel(){
    return Logger.Level.FULL;
  }
}

```

- 配置 @FeignClient 配置

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

- `@EnableFeignClients` `@EnableFeignClients`

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

SpringCloud

SpringCloudGateway

- `SpringCloudGateway`
- `Netfilx Zuul`

SpringCloudGateway

- `SpringCloudGateway`
- `Netfilx Zuul`

image-20251001152734999

application.yaml

application.yaml

- `id`
- `uri`
- `predicates`

```
spring:
  application:
    name: gateway
  cloud:
    nacos:
      server-addr: 192.168.150.101:8848
    gateway:
      routes:
        - id: item # id
          uri: lb://item-service # lb
          predicates: # predicates
            - Path=/items/**,/search/** # Path
        - id: cart
          uri: lb://cart-service
          predicates:
            - Path=/carts/**
```

Java


Java `RouteDefinition`

- `id`
- `uri`
- `predicates`

- `filters` 过滤器链
- 33个过滤器
- 过滤器链的入口 `routes` 和 `default-filters`

过滤器链


- 过滤器链的入口
- 过滤器链的过滤器
 - `PRE` 过滤器
 - `POST` 过滤器
 - 过滤器 `** Netty` 过滤器 `**`

 image-20251002135230563

过滤器链的入口

- 过滤器链的入口 `Http` 过滤器 `OpenFeign`

过滤器链

 image-20251002152441612

过滤器链

过滤器链

- `GatewayFilter` 过滤器
- `GlobalFilter` 过滤器

过滤器链

- `GlobalFilter` 过滤器 `filter`
- `Ordered` 过滤器 `getOrder`
 - 过滤器
 - `Netty` 过滤器 `int`
- 过滤器
 - `AntPathMatcher` 过滤器
 - `exchange` 过滤器

过滤器链

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

    // 1. 请求
    ServerHttpRequest request = exchange.getRequest();
    // 2. 过滤器链
    if(isExclude(request.getPath().toString())){
        return chain.filter(exchange);
    }
    // 3. 过滤器token
```



```

String token = null;
List<String> headers = request.getHeaders().get("authorization");
if(headers != null && !headers.isEmpty()){
    token = headers.get(0);
}
// 4.解析token
Long userId = null;
try{
    userId = jwtTool.parseToken(token);
}catch (UnauthorizedException e){
    // 返回401
    ServerHttpResponse response = exchange.getResponse();
    response.setStatusCode(HttpStatus.UNAUTHORIZED);
    return response.setComplete();
}

// TODO 5.返回结果
System.out.println("userId = " + userId);
// 6.返回
return chain.filter(exchange);
}

```

线程安全问题

- 线程安全问题可以通过 ThreadLocal 解决



image-20251002143457177

- 线程安全问题可以通过 exchange.mutate() 解决



image-20251002143735906

- 通过 common 线程安全问题可以通过 ThreadLocal 解决
 - 通过 SpringMvc 线程安全问题可以通过 common 线程安全问题 WebMvcConfigurer 解决
 - 通过 @ConditionalOnClass 线程安全问题可以通过 ThreadLocal 解决

OpenFeign线程安全

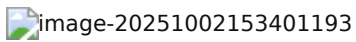
OpenFeign线程安全问题 RequestInterceptor 通过 OpenFeign线程安全问题

- 通过 RequestTemplate 线程安全问题
- 通过 Bean 线程安全问题
- 通过 @FeignClient 线程安全问题 @EnableFeignClients 线程安全问题

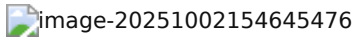
线程安全

线程安全问题

- 通过 RequestTemplate 线程安全问题
- 通过 Bean 线程安全问题



配置
数据库



配置nacos数据库

- 配置 `${hm.db.port:3306}` 数据库端口3306

配置bootstrap.yaml 配置nacos

- 配置

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

- 配置 bootstrap.yaml 配置

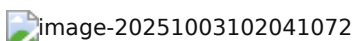
```
config:
  file-extension: yaml # 配置
  shared-configs: # 配置
    - dataId: shared-jdbc.yaml # mybatis配置
    - dataId: shared-log.yaml # 日志配置
    - dataId: shared-swagger.yaml # swagger配置
```

配置

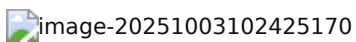
配置数据库

配置

- nacos配置
 - profile 配置



- 配置数据库




配置


配置中心是微服务架构中不可或缺的一部分，Nacos 是阿里巴巴开源的一个配置中心。

配置中心

- 配置中心是 Nacos 的核心功能
- Nacos 配置中心支持多种配置格式
 - 支持 YAML 格式 NacosConfigManager 配置

 image-20251003104452368


- 配置中心支持 RouteDefinitionWriter 接口
 - 支持 yaml 格式 RouteDefinition 配置
 - Nacos 配置中心支持 Nacos json 格式 RouteDefinition
 - RouteDefinitionWriter 的 save 和 delete 方法，Mono 返回类型，.subscribe() 方法

 image-20251003105712632

配置


配置

配置中心是微服务架构中不可或缺的一部分，Nacos 是阿里巴巴开源的一个配置中心。


 image-20251003113010570

配置

- 配置中心支持 RouteDefinitionWriter 接口
- 配置中心支持 Nacos json 格式 RouteDefinition

 image-20251003114504227

- 配置中心支持 RouteDefinitionWriter 接口
 - 配置中心支持 fallback 配置

 image-20251003115000482

Sentinel

配置中心

配置

- 配置中心支持 Sentinel 配置
- Sentinel 支持 Spring MVC 的 HTTP 接口
- 配置中心支持 fallback 配置

- RestfulAPI配置
- 配置+yaml

```
http-method-specify: true
```

配置

配置

配置

配置

Fallback

- FeignClient Sentinel yaml

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

- Fallback
 - FallbackClass
 - FallbackFactory
 - FallbackFactory Client
 - create new Client
 - Bean
 - Client @FeignClient fallbackFactory class

配置

image-20251003150134978

配置

配置

- 配置
- 配置

Seata

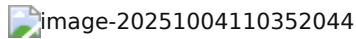
配置

image-20251003153535418

Seata配置

- TC-配置
- TM-配置

- **RM**-两阶段提交协议TC两阶段提交协议



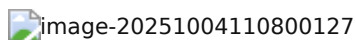
配置

- 配置

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

- Nacos配置TC配置

```
seata:
  registry: # TC配置
    type: nacos # 配置 nacos
    nacos:
      server-addr: 172.21.172.16:8848 # nacos
      namespace: "" # namespace
      group: DEFAULT_GROUP # 配置DEFAULT_GROUP
      application: seata-server # seata
      username: nacos
      password: nacos
  tx-service-group: hmall # 配置
  service:
    vgroup-mapping: # 配置tc
      hmall: "default"
```



Seata配置


- XA配置
- AT配置

XA

XA X/Open两阶段提交协议TM配置RM配置

配置

- 配置
 - TM配置项TC
 - 配置TM配置项RM
 - RM配置TC
 - RM配置sql
 - 配置RM配置项sql
 - RM配置TC
- 配置
 - TM配置TC
 - TC配置项
 - 配置TM配置项TC
 - 配置RM配置项TC

image-20251004112448684

配置

- 配置项

配置XA

- 配置 .yaml 配置XA

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

- 配置项** @GlobalTransactional **

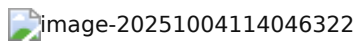
AT

Seata配置AT配置项XA配置项

配置

- 配置
 - TM配置项TC
 - 配置TM配置项RM
 - RM配置TC
 - RM配置项
 - RM配置sql配置项

- RMTC
- - TMT
 - TC
 -
 - TCRMR



- AT

AT

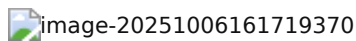
- .yaml AT

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

Elasticsearch

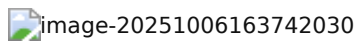
Lucene

-
- Restful
- kibana Logstash Beats ** ELK **



elasticsearch

- document
- term



IK

Elasticsearch Lucene ik_smart ik_max_word

-
- - config IkAnalyzer.cfg.xml (.dic)

- 数据类型

数据

elasticsearch 数据类型 json 数据类型 elasticsearch

index数据类型

mapping数据类型

image-20251007133546860

Mysql Elasticsearch

image-20251007133813213

数据

Kibana Dev Tools

Elasticsearch API Restful

- mapping数据类型

```
PUT /  
{  
  "mappings": {  
    "properties": {  
      "": {  
        "type": "text",  
        "analyzer": "ik_smart"  
      },  
      "2": {  
        "type": "keyword",  
        "index": "false"  
      },  
      "3": {  
        "properties": {  
          "": {  
            "type": "keyword"  
          }  
        }  
      },  
      // ...  
    }  
  }  
}
```

- 数据


```
GET /索引
```

- 索引

```
DELETE /索引
```

- 索引 **mapping** 配置

```
PUT /索引/_mapping
{
  "properties": {
    "索引": {
      "type": "integer"
    }
  }
}
```

Mapping 配置

mapping 配置

- type 配置
 - text 配置 keyword 配置
 - long integer short byte double float
 - boolean
 - date
 - object
- index 配置 **true**
 - 配置
- analyzer 配置
- properties 配置

索引

- 索引

```
POST /索引/_doc/索引id
{
  "索引1": "索引1",
  "索引2": "索引2",
  "索引3": {
    "索引1": "索引3",
    "索引2": "索引4"
  },
  // ...
}
```

- 取得

```
GET /{index}/_doc/{id}
```

- 削除

```
DELETE /{index}/_doc/{id}
```

- 更新

- 完全置き換え

```
PUT /{index}/_doc/{id}
{
  "field1": "value1",
  "field2": "value2",
  // ...
}
```

- 指定したidのフィールドのみ更新

```
POST /{index}/_update/{id}
{
  "doc": {
    "field": "value",
  }
}
```

- 一括操作

- index 一括操作
 - _index 一括操作
 - _id 一括操作
 - { "field1" : "value1" } 一括操作
- delete 一括操作
 - _index 一括操作
 - _id 一括操作
- update 一括操作
 - _index 一括操作
 - _id 一括操作
 - { "doc" : { "field2" : "value2" } } 一括操作

```
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

□□□□□□

- □□□□

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

- □□□ RestHighLevelClient
 - □□□ HttpHost.create □□□ip□□□□□□□□

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

□□□□□□

□□□□□□

- □□ CreateIndexRequest □□□ request □□
 - □□□□□□□□□□
- □□ request.source() □□□□□□□□□□
- □□ .indices() □□□□□□□□□□□□□□□□
 - □□ .create() □□□□□□□□□□
 - □□ request □□

```
@Test
void testCreateIndex() throws IOException {
    // 1.□□Request□□
    CreateIndexRequest request = new CreateIndexRequest("items");
    // 2.□□□□□□□□
    request.source(MAPPING_TEMPLATE, XContentType.JSON);
    // 3.□□□□□□
    client.indices().create(request, RequestOptions.DEFAULT);
}
```

□□□□□□

- □□ DeleteIndexRequest □□□ request □□
 - □□□□□□□□□□

- `client.indices()` 返回 `IndicesClient`
 - `client.indices().delete()` 删除索引
 - `request` 参数

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

测试通过

- `client.indices().get()` 返回 `GetIndexResponse`
 - `request` 参数
- `client.indices().exists()` 检查索引是否存在
 - `request` 参数

```
@Test
void testExistsIndex() throws IOException {
    // 1. 创建 Request
    GetIndexRequest request = new GetIndexRequest("items");
    // 2. 检查索引是否存在
    client.indices().exists(request, RequestOptions.DEFAULT);
}
```

测试通过

测试通过

- `client.index()` 创建索引
 - `request` 参数
- `client.index().source()` 设置索引源
- `client.index().id()` 设置索引 ID

```
@Test
void testIndexDoc() throws IOException {
    // 创建 Item
    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);
}
```


测试

- 创建 DeleteRequest 对象 Request 对象指定 id
- 调用 .delete() 删除文档

```
@Test
void testDeleteDocument() throws IOException {
    // 1. 创建 Request 对象指定 id
    DeleteRequest request = new DeleteRequest("item", "100002644680");
    // 2. 调用
    client.delete(request, RequestOptions.DEFAULT);
}
```

测试

- 创建 GetRequest 对象 Request 对象指定 id
- 调用 .get() 获取文档 GetResponse
- 调用 .getSourceAsString() 获取 _source 内容

 image-20251007151903429

```
@Test
void testGetDocumentById() throws IOException {
    // 1. 创建 Request 对象
    GetRequest request = new GetRequest("items").id("100002644680");
    // 2. 调用
    GetResponse response = client.get(request, RequestOptions.DEFAULT);
    // 3. 获取 _source
    String json = response.getSourceAsString();

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

测试

- 测试
 - 创建 RestClient API 接口 ID
 - 创建 ID 接口
 - 创建 ID 接口
- 测试
 - 创建 UpdateRequest 对象 Request 对象指定 id

- `request.doc()` 返回文档的key value
- `request.update()` 更新文档

```
@Test
void testUpdateDocument() throws IOException {
    // 1. 创建Request
    UpdateRequest request = new UpdateRequest("items", "100002644680");
    // 2. 设置文档
    request.doc(
        "price", 58800,
        "commentCount", 1
    );
    // 3. 执行
    client.update(request, RequestOptions.DEFAULT);
}
```

批量操作

- `BulkRequest` 批量操作CRUD的请求
- `request.add()` 添加文档
 - 添加文档
- `request.bulk()` 批量操作

```
@Test
void testBulk() throws IOException {
    // 1. 创建Request
    BulkRequest request = new BulkRequest();
    // 2. 添加文档
    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. 执行
    client.bulk(request, RequestOptions.DEFAULT);
}
```

DSL

JSON

DSL

- 查询文档
- 更新文档

查询文档

- 查询1000条文档
- 查询from size MySQL
- 查询文档的元数据

- Elasticsearch 1.0.0 版本开始支持

DSL 查询

```
GET /{index}/_search
{
  "query": {
    "match": {
      // .. 查询
    }
  }
}
```

响应

- 响应
 - took 耗时
 - time_out 超时
 - _shards 分片
 - hits 命中
 - total 总命中数
 - max_score 最大得分
 - hits 命中记录

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

查询

- Elasticsearch 1.0.0 版本开始支持
 - match_query
 - multi_match_query
- Elasticsearch 1.0.0 版本开始支持 keyword 类型
 - ids 查询 id
 - range 范围查询
 - term 精确查询
- Elasticsearch 1.0.0 版本开始支持

match 查询

```
GET /{index}/_search
{
  "query": {
    "match": {

```

```
      "type": "type"
    }
  }
}
```

multi_match

multi_match 用于匹配多个字段的查询

```
GET /{index}/_search
{
  "query": {
    "multi_match": {
      "query": "keyword",
      "fields": ["field1", "field2"]
    }
  }
}
```

term

term 用于匹配单个字段的查询

```
GET /{index}/_search
{
  "query": {
    "term": {
      "type": {
        "value": "type"
      }
    }
  }
}
```

range

range 用于匹配字段的范围查询

- gte 大于等于 gt 大于
- lte 小于等于 lt 小于

```
GET /{index}/_search
{
  "query": {
    "range": {
      "type": {
        "gte": {value},
        "lte": {value}
      }
    }
  }
}
```

type

keyword

- `bool`
- `function_score`
- `dis_max`

Elasticsearch 的查询语言

- `must` 必须匹配
- `should` 应该匹配
- `must_not` 必须不匹配
- `filter` 过滤

```
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}}}
      ]
    }
  }
}
```

Elasticsearch 的查询语言

Elasticsearch

Elasticsearch

- `elasticsearch` 的 `_score` 字段** `keyword` 类型
- `function_score`

```
GET /indexName/_search
{
  "query": {
    "match_all": {}
  },
  "sort": [
```

```

{
  "category1": {
    "order": "categoryasc desc"
  },
  {
    "category2": {
      "order": "categoryasc desc"
    }
  }
}

```

注意

- Elasticsearch top10 排序时，默认是按倒序排序
- 从 from 开始 size 返回多少条
 - from 从 0 开始
 - size 返回多少条
 - from + size 不能超过 10000

```

GET /items/_search
{
  "query": {
    "match_all": {}
  },
  "from": 0, // 从 0 开始
  "size": 10, // 返回 10 条
  "sort": [
    {
      "price": {
        "order": "desc"
      }
    }
  ]
}

```

注意

- Elasticsearch 排序时，默认是按倒序排序
- Elasticsearch ** search after **
 - 从 search after 开始

注意

Elasticsearch 排序时，默认是按倒序排序

注意

- Elasticsearch 排序时，默认是按倒序排序

```
GET /{index}/_search
{
  "query": {
    "match": {
      "type": "text"
    }
  },
  "highlight": {
    "fields": {
      "text": {
        "pre_tags": "<em>",
        "post_tags": "</em>"
      }
    }
  }
}
```

agg

agg 是 aggregation 的缩写

- 聚合类型
 - TermAggregation 词频聚合
 - Date Histogram 日期直方图
- 聚合函数
 - Avg 平均值
 - Max 最大值
 - Min 最小值
 - Status 统计 max min avg sum
- 聚合桶

agg

- 聚合
 - 通过 aggs 聚合

```
GET /items/_search
{
  "query": {
    "match_all": {}
  }, // 匹配所有
  "size": 0, // size 0 表示不返回数据
  "aggs": { // 聚合
    "cateAgg": { // 聚合桶
      "terms": { // 聚合桶 term
        "field": "category", // 聚合桶
        "size": 20 // 聚合桶大小
      }
    }
  }
}
```

```
}  
}
```

- `SearchRequest`
 - `SearchRequestBuilder`

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

JavaRestClient

예제

- `SearchRequest` 객체 request 생성
 - `SearchRequestBuilder`
- `request.source()` 쿼리 조건
- `request.query()` 쿼리 조건

- `QueryBuilders` `matchAllQuery()`
- `.search()`
 - `request`

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

QueryBuilders

QueryBuilders

- `QueryBuilders` API
 - `QueryBuilders.MatchQuery`
 - `QueryBuilders.multiMatchQuery`

```
//
QueryBuilders.MatchQuery("name", " ");
//
QueryBuilders.multiMatchQuery(" ", "name", "catrgory");
```



image-20251008134512732

- `QueryBuilders` API
 - `QueryBuilders.termQuery`
 - `QueryBuilders.rangeQuery`

```
//
QueryBuilders.termQuery("category", " ");
//
QueryBuilders.rangeQuery("price").gte(100).lte(150);
```



image-20251008134635397

- `QueryBuilders` API
 - `BoolQueryBuilder` `QueryBuilders.boolQuery`
 - `.must()` `must`

```
// 布尔查询
BoolQueryBuilder boolQuery = QueryBuilders.boolQuery();
// 必须
boolQuery.must(
    QueryBuilders.termQuery("brand", "小米");
// 应该
boolQuery.should(
    QueryBuilders.rangeQuery("price").lte(2500));
```

查询

request.source() 查询

- .from() .size() 查询
- .sort() 排序
 - SortOrder 排序
 - 排序

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

查询

request.source() 查询

- .highlighter() 高亮
- SearchSourceBuilder.highlight() 高亮

```
request.source().highlighter(
    SearchSourceBuilder.highlight()
        .field("name")
        .preTags("<em>")
        .postTags("</em>")
);
```

- 查询

```
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();

        // 查询
    }
}
```

```

        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);
    }
}

```

□□□□

□□ request.source() □□

- □□ .aggregation() □□□□□□□□□□
- □□ AggregationBuilders □□□□□□

```

request.source().aggregation(
    AggregationBuilders
        .terms("brand_agg")    // □□□□□□
        .field("brand")        // □□□□□□
        .size(20)              // □□□□□□□□
);

```

- □□□□□□□□□□□□□□□□

@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());  
}  
}
```

分布式数据库

分布式数据库

分布式数据库的组成


- **CP** 分布式数据库的组成
 - **XA**
- **AP** 分布式数据库的组成
 - **AT**

CAP

分布式数据库的组成

- Consistency 一致性
- Availability 可用性
- Partition tolerance 分区容忍性

分布式数据库的组成

 image-20251008145844622

Base

BASE 分布式数据库的组成

- **Basically Available** 基本可用
- **Soft State** 软状态
- **Eventually Consistent** 最终一致性

AT