2024/8/10 XiaoYc

0.实战演练

1. **启动类（Application.java）**

```
package com.example.demo;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
@SpringBootApplication
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

2. **实体类（User.java）**

```
package com.example.demo.entity;
import com.baomidou.mybatisplus.annotation.IdType;
import com.baomidou.mybatisplus.annotation.TableId;
import com.baomidou.mybatisplus.annotation.TableName;
import lombok.Data;
@Data
@TableName("user")
public class User {
    @TableId(type = IdType.AUTO)
    private Long id;
    private String name;
    private Integer age;
    private String email;
}
```

3. **Mapper层（UserMapper.java）**

```
package com.example.demo.mapper;
import com.baomidou.mybatisplus.core.mapper.BaseMapper;
import com.example.demo.entity.User;
import org.apache.ibatis.annotations.Mapper;
@Mapper
public interface UserMapper extends BaseMapper<User> {
    // 这里可以添加自定义的SQL方法
}
```

4. **Service层（UserService.java）**

```
package com.example.demo.service;
import com.baomidou.mybatisplus.extension.service.IService;
import com.example.demo.entity.User;
public interface UserService extends IService<User> {
    // 这里可以添加自定义的业务方法
}
```

Service实现类（UserServiceImpl.java）

```
package com.example.demo.service.impl;
import com.baomidou.mybatisplus.extension.service.impl.ServiceImpl;
import com.example.demo.entity.User;
import com.example.demo.mapper.UserMapper;
import com.example.demo.service.UserService;
import org.springframework.stereotype.Service;
@Service
public class UserServiceImpl extends ServiceImpl<UserMapper, User> implements
UserService {
    // 这里可以添加自定义的业务方法实现
}
```

5. **测试类（UserMapperTest.java）**

```
package com.example.demo;
import com.example.demo.entity.User;
import com.example.demo.mapper.UserMapper;
import com.example.demo.service.UserService;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
@SpringBootTest
public class UserMapperTest {
    @Autowired
    private UserMapper userMapper;
    @Autowired
    private UserService userService;
    @Test
    public void testSelect() {
        // 查询所有用户
        System.out.println(userMapper.selectList(null));

        // 根据ID查询用户
        System.out.println(userMapper.selectById(1L));

        // 使用Service层的save方法保存用户
        User user = new User();
        user.setName("Alice");
        user.setAge(18);
        user.setEmail("alice@example.com");
        userService.save(user);

        // 使用Service层的update方法更新用户
        user.setName("Alice Updated");
        userService.updateById(user);

        // 使用Service层的remove方法删除用户
        userService.removeById(1L);
    }
}
```

这个示例涵盖了基本的CRUD操作，使用了MyBatis-Plus提供的`BaseMapper`和`IService`接口。这些接口提供了大量的内置方法，比如`selectList`、`selectById`、`save`、`updateById`和`removeById`，这些方法大大简化了数据库操作。
请确保在你的`pom.xml`或`build.gradle`文件中添加了MyBatis-Plus和Spring Boot的依赖，并且正确配置了数据库连接信息。这个示例假设你已经有了对应的数据库和表结构。

1.Mp实现分页

在MyBatis-Plus中，分页查询非常简单，它提供了一个`Page`对象来简化分页操作。以下是如何在MyBatis-Plus中使用分页，并且对分页进行解析。

1. **修改启动类以启用分页插件**
   首先，你需要在启动类中添加`@MapperScan`注解来指定Mapper接口的扫描路径，并使用`@Bean`来注册MyBatis-Plus的分页插件。

```
package com.example.demo;
import com.baomidou.mybatisplus.extension.plugins.MybatisPlusInterceptor;
import
com.baomidou.mybatisplus.extension.plugins.inner.PaginationInnerInterceptor;
import org.mybatis.spring.annotation.MapperScan;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
@SpringBootApplication
@MapperScan("com.example.demo.mapper")
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
    @Bean
    public MybatisPlusInterceptor mybatisPlusInterceptor() {
        MybatisPlusInterceptor interceptor = new MybatisPlusInterceptor();
        interceptor.addInnerInterceptor(new PaginationInnerInterceptor());
        return interceptor;
    }
}
```

2. **Service层添加分页方法**
   在`UserService`接口中添加一个分页查询的方法。

```
package com.example.demo.service;
import com.baomidou.mybatisplus.extension.plugins.pagination.Page;
import com.baomidou.mybatisplus.extension.service.IService;
import com.example.demo.entity.User;
public interface UserService extends IService<User> {
    Page<User> selectPageWithCondition(Page<User> page);
}
```

在`UserServiceImpl`中实现该方法。

```
package com.example.demo.service.impl;
import com.baomidou.mybatisplus.extension.plugins.pagination.Page;
import com.baomidou.mybatisplus.extension.service.impl.ServiceImpl;
```

```
import com.example.demo.entity.User;
import com.example.demo.mapper.UserMapper;
import com.example.demo.service.UserService;
import org.springframework.stereotype.Service;
@Service
public class UserServiceImpl extends ServiceImpl<UserMapper, User> implements
UserService {
    @Override
    public Page<User> selectPageWithCondition(Page<User> page) {
        // 这里可以添加查询条件，比如：page.setRecords(userMapper.selectPageVo(page,
condition));
        // 这里假设没有其他查询条件，只是简单的分页查询
        return page.setRecords(baseMapper.selectPage(page, null).getRecords());
    }
}
```

3. **测试类中添加分页测试方法**
   在`UserMapperTest`测试类中添加一个测试分页的方法。

```
package com.example.demo;
import com.baomidou.mybatisplus.extension.plugins.pagination.Page;
import com.example.demo.entity.User;
import com.example.demo.service.UserService;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
@SpringBootTest
public class UserMapperTest {
    @Autowired
    private UserService userService;
    @Test
    public void testSelectPage() {
        // 创建分页对象，第一个参数是当前页码，第二个参数是每页显示条数
        Page<User> page = new Page<>(1, 5);
        // 执行分页查询
        Page<User> userPage = userService.selectPageWithCondition(page);
        // 输出查询结果
        System.out.println("总页数: " + userPage.getPages());
        System.out.println("总记录数: " + userPage.getTotal());
        userPage.getRecords().forEach(System.out::println);
    }
}
```

在上述代码中，`Page<User>`对象被用来执行分页查询。通过调用`userService.selectPageWithCondition(page)`，MyBatis-Plus会自动处理分页逻辑，并在SQL查询中添加`LIMIT`语句。

分页解析：

- `Page<User> page = new Page<>(1, 5);`创建了一个分页对象，其中`1`是当前页码，`5`是每页显示的记录数。
- `Page<User> userPage = userService.selectPageWithCondition(page);`调用服务层的分页方法，执行分页查询。
- `userPage.getPages();`获取总页数。

- `userPage.getTotal();` 获取总记录数。
- `userPage.getRecords();` 获取当前页的记录列表。
  MyBatis-Plus分页插件会在底层生成两个SQL查询：

1. 一个是用于计算总记录数的`COUNT`查询。
2. 另一个是获取当前页数据的`SELECT`查询，并带有`LIMIT`子句来限制返回的记录数。
   这种方式大大简化了分页逻辑，避免了手动编写复杂的分页SQL语句。

2.条件构造器（Wrapper）：

以下是一个基于MyBatis-Plus与Spring Boot的示例，其中将重点展示如何使用条件构造器（Wrapper）来实现复杂的查询条件。

1. **启动类（Application.java）**

```
package com.example.demo;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
@SpringBootApplication
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

2. **实体类（User.java）**

```
package com.example.demo.entity;
import com.baomidou.mybatisplus.annotation.TableId;
import com.baomidou.mybatisplus.annotation.TableName;
import lombok.Data;
@Data
@TableName("user")
public class User {
    @TableId
    private Long id;
    private String name;
    private Integer age;
    private String email;
}
```

3. **Mapper层（UserMapper.java）**

```
package com.example.demo.mapper;
import com.baomidou.mybatisplus.core.mapper.BaseMapper;
import com.example.demo.entity.User;
import org.apache.ibatis.annotations.Mapper;
@Mapper
public interface UserMapper extends BaseMapper<User> {
    // 这里可以添加自定义的SQL方法
}
```

4. **Service层（UserService.java）**

```java
package com.example.demo.service;
import com.baomidou.mybatisplus.core.conditions.Wrapper;
import com.baomidou.mybatisplus.extension.service.IService;
import com.example.demo.entity.User;
public interface UserService extends IService<User> {
    // 使用Wrapper构建复杂查询
    List<User> findUsers(Wrapper<User> queryWrapper);
}
```

Service实现类（UserServiceImpl.java）

```java
package com.example.demo.service.impl;
import com.baomidou.mybatisplus.core.conditions.query.QueryWrapper;
import com.baomidou.mybatisplus.extension.service.impl.ServiceImpl;
import com.example.demo.entity.User;
import com.example.demo.mapper.UserMapper;
import com.example.demo.service.UserService;
import org.springframework.stereotype.Service;
import java.util.List;
@Service
public class UserServiceImpl extends ServiceImpl<UserMapper, User> implements
UserService {
    @Override
    public List<User> findUsers(Wrapper<User> queryWrapper) {
        return list(queryWrapper);
    }
}
```

5. **测试类（UserMapperTest.java）**

```java
package com.example.demo;
import com.baomidou.mybatisplus.core.conditions.query.QueryWrapper;
import com.example.demo.entity.User;
import com.example.demo.service.UserService;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import java.util.List;
@SpringBootTest
public class UserMapperTest {
    @Autowired
    private UserService userService;
    @Test
    public void testQueryWrapper() {
        // 创建QueryWrapper对象
        QueryWrapper<User> queryWrapper = new QueryWrapper<>();

        // 构建查询条件
        queryWrapper.eq("name", "Alice") // 名字等于Alice
                    .and(wrapper -> wrapper.gt("age", 20)) // 年龄大于20
                    .or() // 或者
                    .eq("email", "bob@example.com"); // 邮箱等于bob@example.com
        // 执行查询
        List<User> users = userService.findUsers(queryWrapper);
```

```
        // 输出结果
        users.forEach(System.out::println);
    }
}
```

在这个测试方法中，我们使用了`QueryWrapper`来构建一个查询条件，其中包含了多个条件逻辑：

- `eq` 方法用于添加等于的条件。
- `gt` 方法用于添加大于的条件。
- `and` 方法用于添加与（AND）的逻辑。
- `or` 方法用于添加或（OR）的逻辑。
  你可以根据需要添加更多的条件方法，如`like`、`in`、`between`等，以构建更复杂的查询条件。
  确保在`pom.xml`或`build.gradle`文件中添加了MyBatis-Plus和Spring Boot的依赖，并且正确配置了数据库连接信息。这个示例假设你已经有了对应的数据库和表结构。

3.代码生成器：

以下是一个基于MyBatis-Plus与Spring Boot的示例，其中将重点展示如何使用MyBatis-Plus的代码生成器。

1. **启动类（Application.java）**

```
package com.example.demo;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
@SpringBootApplication
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

2. **代码生成器配置和运行（CodeGenerator.java）**

```
package com.example.demo;
import com.baomidou.mybatisplus.annotation.DbType;
import com.baomidou.mybatisplus.annotation.IdType;
import com.baomidou.mybatisplus.generator.AutoGenerator;
import com.baomidou.mybatisplus.generator.config.DataSourceConfig;
import com.baomidou.mybatisplus.generator.config.GlobalConfig;
import com.baomidou.mybatisplus.generator.config.PackageConfig;
import com.baomidou.mybatisplus.generator.config.StrategyConfig;
import com.baomidou.mybatisplus.generator.config.rules.DateType;
import com.baomidou.mybatisplus.generator.config.rules.NamingStrategy;
public class CodeGenerator {
    public static void main(String[] args) {
        // 1. 全局配置
        GlobalConfig globalConfig = new GlobalConfig();
        globalConfig.setOutputDir(System.getProperty("user.dir") +
"/src/main/java");
        globalConfig.setAuthor("Author Name");
        globalConfig.setOpen(false);
```

```
        globalConfig.setFileOverride(true);
        globalConfig.setIdType(IdType.AUTO);
        globalConfig.setDateType(DateType.ONLY_DATE);
        globalConfig.setSwagger2(true);
        // 2. 数据源配置
        DataSourceConfig dataSourceConfig = new DataSourceConfig();
        dataSourceConfig.setUrl("jdbc:mysql://localhost:3306/your_database?
useUnicode=true&useSSL=false&characterEncoding=utf8");
        dataSourceConfig.setDriverName("com.mysql.cj.jdbc.Driver");
        dataSourceConfig.setUsername("root");
        dataSourceConfig.setPassword("password");
        dataSourceConfig.setDbType(DbType.MYSQL);
        // 3. 包配置
        PackageConfig packageConfig = new PackageConfig();
        packageConfig.setParent("com.example.demo");
        packageConfig.setEntity("entity");
        packageConfig.setMapper("mapper");
        packageConfig.setService("service");
        packageConfig.setServiceImpl("service.impl");
        packageConfig.setController("controller");
        // 4. 策略配置
        StrategyConfig strategyConfig = new StrategyConfig();
        strategyConfig.setInclude("table_name"); // 设置要映射的表名
        strategyConfig.setNaming(NamingStrategy.underline_to_camel);
        strategyConfig.setColumnNaming(NamingStrategy.underline_to_camel);
        strategyConfig.setEntityLombokModel(true);
        strategyConfig.setRestControllerStyle(true);
        strategyConfig.setControllerMappingHyphenStyle(true);
        // 5. 整合配置
        AutoGenerator autoGenerator = new AutoGenerator();
        autoGenerator.setGlobalConfig(globalConfig);
        autoGenerator.setDataSource(dataSourceConfig);
        autoGenerator.setPackageInfo(packageConfig);
        autoGenerator.setStrategy(strategyConfig);
        // 6. 执行
        autoGenerator.execute();
    }
}
```

在上面的代码中，你需要替换`dataSourceConfig`中的数据库连接信息，以及`strategyConfig`中的表名。运行`CodeGenerator`类将自动生成以下文件：

- 实体类（Entity）
- Mapper接口
- Mapper XML文件
- Service接口
- Service实现类
- Controller类

3. **Service层（示例：UserService.java）**

```java
package com.example.demo.service;
import com.baomidou.mybatisplus.extension.service.IService;
import com.example.demo.entity.User;
public interface UserService extends IService<User> {
    // 这里可以添加自定义的业务方法
}
```

4. **Mapper层（示例：UserMapper.java）**

```java
package com.example.demo.mapper;
import com.baomidou.mybatisplus.core.mapper.BaseMapper;
import com.example.demo.entity.User;
public interface UserMapper extends BaseMapper<User> {
    // 这里可以添加自定义的SQL方法
}
```

5. **测试类（示例：UserMapperTest.java）**

```java
package com.example.demo;
import com.example.demo.entity.User;
import com.example.demo.service.UserService;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
@SpringBootTest
public class UserMapperTest {
    @Autowired
    private UserService userService;
    @Test
    public void testService() {
        // 使用Service层的save方法保存用户
        User user = new User();
        user.setName("Alice");
        user.setAge(18);
        user.setEmail("alice@example.com");
        userService.save(user);
        // 使用Service层的list方法查询所有用户
        List<User> users = userService.list();
        users.forEach(System.out::println);
    }
}
```

确保在`pom.xml`或`build.gradle`文件中添加了MyBatis-Plus的依赖，以及数据库驱动的依赖。这个示例假设你已经有了对应的数据库和表结构。
在运行代码生成器之前，请确保已经正确配置了数据库

这段Java代码的作用是使用MyBatis-Plus的代码生成器（AutoGenerator）来生成一套完整的CRUD（创建、读取、更新、删除）代码。当你运行这个`CodeGenerator`类的`main`方法时，它会产生以下结果：

1. **实体类（Entity）**：
    - 在指定的包`com.example.demo.entity`下，会为数据库中的每个表生成一个对应的Java实体类。这个类将包含与数据库表字段相对应的属性，以及相应的getter和setter方法。
2. **Mapper接口**：

   - 在指定的包`com.example.demo.mapper`下，会为每个表生成一个Mapper接口。这个接口包含了基本的CRUD操作方法，以及你可以添加的自定义查询方法。
3. **Mapper XML文件**：
   - 在Mapper接口的同级目录下，会生成对应的XML文件，这些文件包含了Mapper接口中方法的SQL实现。
4. **Service接口**：
   - 在指定的包`com.example.demo.service`下，会为每个实体生成一个Service接口。这个接口定义了业务逻辑操作的方法。
5. **Service实现类**：
   - 在指定的包`com.example.demo.service.impl`下，会为每个Service接口生成一个实现类。这个类实现了Service接口中定义的业务逻辑。
6. **Controller类**：
   - 在指定的包`com.example.demo.controller`下，会为每个实体生成一个RESTful风格的Controller类。这个类处理HTTP请求，调用Service层的逻辑，并返回响应。
     以下是代码生成器的主要配置项：

- `globalConfig`：全局配置，包括输出目录、作者、是否打开输出目录、是否覆盖已有文件、主键生成策略、日期类型、是否生成Swagger注解等。
- `dataSourceConfig`：数据源配置，包括数据库URL、驱动名称、用户名、密码和数据源类型。
- `packageConfig`：包配置，定义了生成代码的包结构。
- `strategyConfig`：策略配置，包括要映射的表名、命名规则、是否使用Lombok、Controller风格等。
  运行这个代码生成器后，你将在指定的输出目录下得到一个完整的代码结构，可以立即开始进行业务逻辑的开发，而无需手动编写基本的CRUD代码。这大大提高了开发效率，并有助于保持代码的一致性和标准化。

4.逻辑删除：

逻辑删除是一种数据库设计中的概念，它允许在表中保留记录，但不实际从数据库中删除它们。这种方法在许多情况下都非常有用，尤其是当你需要保留历史数据或者在某些业务逻辑中需要暂时隐藏某些记录时。以下是一些逻辑删除的应用场景：

1. **保留历史数据**： 当你需要保留用户操作的历史记录时，逻辑删除非常有用。例如，如果你有一个帖子表，你可能会想要保留帖子被删除的历史记录，以便于审计和恢复。
2. **临时隐藏记录**： 在某些情况下，你可能需要暂时隐藏某些记录，例如，在用户账户被暂时禁用时。使用逻辑删除，你可以标记这些记录为已删除，而不需要实际从数据库中删除它们。
3. **数据迁移**： 在数据库迁移的过程中，你可能需要保留旧数据，以便于在新系统中进行验证或转换。逻辑删除可以帮助你保持旧数据，同时在新系统中添加新数据。
4. **批量操作**： 当你需要对大量记录执行删除操作时，逻辑删除可以避免因一次性删除大量记录而导致的性能问题。你可以先标记这些记录为已删除，然后在适当的时候执行物理删除。
5. **备份与恢复**： 在备份和恢复数据库时，逻辑删除可以帮助你保留原始数据，以便于在恢复时能够还原到特定的状态。
6. **业务逻辑处理**： 在某些业务逻辑中，你可能需要根据特定的条件隐藏或显示某些记录。逻辑删除可以让你在业务逻辑层面控制这些记录的可见性。

以下是一个基于MyBatis-Plus与Spring Boot的示例，其中将重点展示如何使用逻辑删除功能。

1. **启动类（Application.java）**

```java
package com.example.demo;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
@SpringBootApplication
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

2. **实体类（User.java）**

```java
package com.example.demo.entity;
import com.baomidou.mybatisplus.annotation.TableId;
import com.baomidou.mybatisplus.annotation.TableName;
import com.baomidou.mybatisplus.annotation.TableLogic;
import lombok.Data;
@Data
@TableName("user")
public class User {
    @TableId
    private Long id;
    private String name;
    private Integer age;
    private String email;
    @TableLogic
    private Integer deleted; // 逻辑删除字段，0表示未删除，1表示已删除
}
```

3. **Mapper层（UserMapper.java）**

```java
package com.example.demo.mapper;
import com.baomidou.mybatisplus.core.mapper.BaseMapper;
import com.example.demo.entity.User;
import org.apache.ibatis.annotations.Mapper;
@Mapper
public interface UserMapper extends BaseMapper<User> {
    // 这里可以添加自定义的SQL方法
}
```

4. **Service层（UserService.java）**

```java
package com.example.demo.service;
import com.baomidou.mybatisplus.extension.service.IService;
import com.example.demo.entity.User;
public interface UserService extends IService<User> {
    // 这里可以添加自定义的业务方法
}
```

Service实现类（UserServiceImpl.java）

```java
package com.example.demo.service.impl;
import com.baomidou.mybatisplus.extension.service.impl.ServiceImpl;
import com.example.demo.entity.User;
import com.example.demo.mapper.UserMapper;
import com.example.demo.service.UserService;
import org.springframework.stereotype.Service;
@Service
public class UserServiceImpl extends ServiceImpl<UserMapper, User> implements
UserService {
    // 这里可以添加自定义的业务方法实现
}
```

5. **测试类（UserMapperTest.java）**

```java
package com.example.demo;
import com.example.demo.entity.User;
import com.example.demo.service.UserService;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
@SpringBootTest
public class UserMapperTest {
    @Autowired
    private UserService userService;
    @Test
    public void testLogicDelete() {
        // 创建用户并保存
        User user = new User();
        user.setName("Alice");
        user.setAge(18);
        user.setEmail("alice@example.com");
        user.setDeleted(0); // 设置逻辑删除字段为0（未删除）
        userService.save(user);
        // 查询用户
        User foundUser = userService.getById(user.getId());
        System.out.println("User found: " + foundUser);
        // 逻辑删除用户
        userService.removeById(user.getId());
        // 再次查询用户，应该返回null，因为用户已被逻辑删除
        User deletedUser = userService.getById(user.getId());
        System.out.println("User after logic delete: " + deletedUser);
    }
}
```

在这个测试方法中，我们创建了一个用户，将其保存到数据库中，然后使用逻辑删除功能来删除该用户。逻辑删除并不会真正从数据库中删除记录，而是更新`deleted`字段的值来标记记录为已删除。在MyBatis-Plus中，你可以通过配置`@TableLogic`注解来启用逻辑删除功能。
确保在`application.properties`或`application.yml`中配置了逻辑删除的相关属性：

```properties
# 逻辑删除配置
mybatis-plus.global-config.db-config.logic-delete-value=1
mybatis-plus.global-config.db-config.logic-not-delete-value=0
```

这里的`logic-delete-value`和`logic-not-delete-value`分别表示逻辑删除字段为删除状态和未删除状态的值。通常情况下，我们使用`1`表示已删除，`0`表示未删除。

5.性能分析插件：

以下是一个基于MyBatis-Plus与Spring Boot的示例，其中将重点展示如何使用性能分析插件。

1. **启动类（Application.java）**

```
package com.example.demo;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
@SpringBootApplication
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

2. **实体类（User.java）**

```
package com.example.demo.entity;
import com.baomidou.mybatisplus.annotation.TableId;
import com.baomidou.mybatisplus.annotation.TableName;
import lombok.Data;
@Data
@TableName("user")
public class User {
    @TableId
    private Long id;
    private String name;
    private Integer age;
    private String email;
}
```

3. **Mapper层（UserMapper.java）**

```
package com.example.demo.mapper;
import com.baomidou.mybatisplus.core.mapper.BaseMapper;
import com.example.demo.entity.User;
import org.apache.ibatis.annotations.Mapper;
@Mapper
public interface UserMapper extends BaseMapper<User> {
    // 这里可以添加自定义的SQL方法
}
```

4. **Service层（UserService.java）**

```
package com.example.demo.service;
import com.baomidou.mybatisplus.extension.service.IService;
import com.example.demo.entity.User;
public interface UserService extends IService<User> {
    // 这里可以添加自定义的业务方法
}
```

Service实现类（UserServiceImpl.java）

```
package com.example.demo.service.impl;
import com.baomidou.mybatisplus.extension.service.impl.ServiceImpl;
import com.example.demo.entity.User;
import com.example.demo.mapper.UserMapper;
import com.example.demo.service.UserService;
import org.springframework.stereotype.Service;
@Service
public class UserServiceImpl extends ServiceImpl<UserMapper, User> implements
UserService {
    // 这里可以添加自定义的业务方法实现
}
```

5. **性能分析插件配置**
   在`application.properties`或`application.yml`中配置性能分析插件：

```
mybatis-plus.configuration.log-impl=org.apache.ibatis.logging.stdout.StdOutImpl
```

或者在`application.yml`中配置：

```
mybatis-plus:
  configuration:
    log-impl: org.apache.ibatis.logging.stdout.StdOutImpl
```

这个配置将日志输出重定向到控制台，以便于查看SQL语句及其执行时间。

6. **测试类（UserMapperTest.java）**

```
package com.example.demo;
import com.example.demo.entity.User;
import com.example.demo.service.UserService;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import java.util.List;
@SpringBootTest
public class UserMapperTest {
    @Autowired
    private UserService userService;
    @Test
    public void testPerformanceAnalysis() {
        // 这里可以添加自定义的业务方法实现
        List<User> users = userService.list();
        users.forEach(System.out::println);
    }
```

```
}
```

在这个测试方法中，我们调用`userService.list()`方法来获取所有用户，这会执行一个SQL查询。由于我们启由于我们启用了性能分析插件，SQL语句及其执行时间将被打印到控制台。这样，你就可以看到MyBatis-Plus执行SQL语句的速度，从而帮助你分析SQL语句的性能瓶颈。

请注意，性能分析插件可能会影响应用程序的性能，因为它需要记录每个SQL语句的执行时间。因此，通常只在开发和测试阶段使用它，而在生产环境中禁用。

确保在`application.properties`或`application.yml`中正确配置了性能分析插件，并且已经添加了MyBatis-Plus和Spring Boot的依赖。这个示例假设你已经有了对应的数据库和表结构。

在运行测试类后，你应该会在控制台看到SQL语句及其执行时间，这有助于你分析和优化SQL查询的性能。

在执行上述命令后，你应该会在控制台看到MyBatis-Plus执行SQL语句的性能分析结果。这些结果通常包括以下信息：

1. **SQL语句**：执行的SQL查询语句，包括参数绑定。
2. **执行时间**：SQL语句执行所花费的时间，以毫秒为单位。
3. **执行次数**：SQL语句执行的次数。
   例如，如果你有一个简单的查询所有用户的SQL语句，执行结果可能如下所示：

```
==>  Preparing: SELECT id, name, age, email, deleted FROM user WHERE deleted = 0
==> Parameters:
<==    Columns: id, name, age, email, deleted
<==        Row: 1000000
<==      Total: 1000000
```

在这个例子中：

- `==>  Preparing:` 表示MyBatis-Plus正在准备执行的SQL语句。
- `==> Parameters:` 表示传递给SQL语句的参数，如果有的话。
- `<== Columns:` 表示MyBatis-Plus从数据库中检索到的列名。
- `<==         Row:` 表示MyBatis-Plus从数据库中检索到的行数。
- `<==       Total:` 表示MyBatis-Plus从数据库中检索到的总行数。
  请注意，这些日志可能会根据MyBatis-Plus的版本和配置有所不同。在某些情况下，你可能需要调整日志级别或使用特定的日志框架来查看这些信息。
  性能分析插件对于调试和优化SQL查询非常有用，因为它可以帮助你识别执行缓慢的查询，并找出可能需要优化的地方。在生产环境中，这些日志可能会对性能产生影响，因此通常建议在开发和测试阶段使用，而在生产环境中禁用。

6.自动填充功能：

以下是一个基于MyBatis-Plus与Spring Boot的示例，其中将重点展示如何使用自动填充功能。

1. **启动类（Application.java）**

```java
package com.example.demo;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
@SpringBootApplication
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

2. **实体类（User.java）**

```java
package com.example.demo.entity;
import com.baomidou.mybatisplus.annotation.TableId;
import com.baomidou.mybatisplus.annotation.TableName;
import com.baomidou.mybatisplus.annotation.TableField;
import lombok.Data;
@Data
@TableName("user")
public class User {
    @TableId
    private Long id;
    private String name;
    private Integer age;
    private String email;
    @TableField(fill = FieldFill.INSERT)
    private Date createTime; // 创建时间
    @TableField(fill = FieldFill.INSERT_UPDATE)
    private Date updateTime; // 更新时间
}
```

3. **Mapper层（UserMapper.java）**

```java
package com.example.demo.mapper;
import com.baomidou.mybatisplus.core.mapper.BaseMapper;
import com.example.demo.entity.User;
import org.apache.ibatis.annotations.Mapper;
@Mapper
public interface UserMapper extends BaseMapper<User> {
    // 这里可以添加自定义的SQL方法
}
```

4. **Service层（UserService.java）**

```java
package com.example.demo.service;
import com.baomidou.mybatisplus.extension.service.IService;
import com.example.demo.entity.User;
public interface UserService extends IService<User> {
    // 这里可以添加自定义的业务方法
}
```

Service实现类（UserServiceImpl.java）

```java
package com.example.demo.service.impl;
import com.baomidou.mybatisplus.extension.service.impl.ServiceImpl;
import com.example.demo.entity.User;
import com.example.demo.mapper.UserMapper;
import com.example.demo.service.UserService;
import org.springframework.stereotype.Service;
@Service
public class UserServiceImpl extends ServiceImpl<UserMapper, User> implements
UserService {
    // 这里可以添加自定义的业务方法实现
}
```

5. **测试类（UserMapperTest.java）**

```java
package com.example.demo;
import com.example.demo.entity.User;
import com.example.demo.service.UserService;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import java.util.Date;
@SpringBootTest
public class UserMapperTest {
    @Autowired
    private UserService userService;
    @Test
    public void testAutofill() {
        // 创建用户并保存
        User user = new User();
        user.setName("Alice");
        user.setAge(18);
        user.setEmail("alice@example.com");
        userService.save(user);
        // 输出创建时间
        System.out.println("Create Time: " + user.getCreateTime());
        // 更新用户
        user.setName("Alice Updated");
        userService.updateById(user);
        // 输出更新时间
        System.out.println("Update Time: " + user.getUpdateTime());
    }
}
```

在这个测试方法中，我们创建在这个测试方法中，我们创建了一个用户对象，并使用`UserService`的`save`方法将其保存到数据库中。由于我们为`createTime`字段设置了`@TableField(fill = FieldFill.INSERT)`注解，MyBatis-Plus会在保存操作时自动填充这个字段的值，即当前时间。

然后，我们更新了用户对象，并再次使用`UserService`的`updateById`方法来更新数据库中的记录。由于我们为`updateTime`字段设置了`@TableField(fill = FieldFill.INSERT_UPDATE)`注解，MyBatis-Plus会在更新操作时自动填充这个字段的值，即当前时间。

执行测试方法后，你应该会在控制台看到`createTime`和`updateTime`字段的值，它们应该分别是保存操作和更新操作时的时间戳。

确保在 `application.properties` 或 `application.yml` 中配置了MyBatis-Plus的相关属性，并且已经添加了MyBatis-Plus和Spring Boot的依赖。这个示例假设你已经有了对应的数据库和表结构。

在运行测试类后，你应该会在控制台看到创建时间和更新时间的输出，这证明了自动填充功能的工作原理。

> 在使用 MyBatis-Plus 的自动填充功能时，通常需要手动进行配置。这是因为自动填充不是 MyBatis-Plus 的默认行为，而是需要根据具体的业务需求来定制的。
> 具体来说，你需要进行以下几步手动配置：

1. **定义实体类字段**：在实体类中使用 `@TableField` 注解来标记需要进行自动填充的字段，并指定填充策略（如 `FieldFill.INSERT` 或 `FieldFill.INSERT_UPDATE`）。
2. **配置元数据处理器**：创建一个配置类，并定义一个 `MetaObjectHandler` 的 Bean，在这个 Bean 中实现 `insertFill` 和 `updateFill` 方法，这些方法指定了在插入和更新操作时如何填充这些字段。
   下面是一个简单的例子，展示如何配置自动填充：

```
import com.baomidou.mybatisplus.core.handlers.MetaObjectHandler;
import org.apache.ibatis.reflection.MetaObject;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import java.util.Date;
@Configuration
public class MyBatisPlusConfig {
    @Bean
    public MetaObjectHandler metaObjectHandler() {
        return new MetaObjectHandler() {
            @Override
            public void insertFill(MetaObject metaObject) {
                // 插入操作时填充的字段
                this.strictInsertFill(metaObject, "createTime", Date.class, new
Date());
                this.strictInsertFill(metaObject, "updateTime", Date.class, new
Date());
            }
            @Override
            public void updateFill(MetaObject metaObject) {
                // 更新操作时填充的字段
                this.strictUpdateFill(metaObject, "updateTime", Date.class, new
Date());
            }
        };
    }
}
```

在上面的配置中，`insertFill` 方法会在插入操作时填充 `createTime` 和 `updateTime` 字段，而 `updateFill` 方法会在更新操作时填充 `updateTime` 字段。

7.乐观锁（版本号机制）：

以下是一个基于 MyBatis-Plus 与 Spring Boot 的项目示例，其中包含启动类、Mapper 层、Service 层以及一个测试类。在这个示例中，我将重点展示如何使用乐观锁。
首先，你需要添加 MyBatis-Plus 的依赖到你的 `pom.xml` 文件中：

```xml
<!-- MyBatis-Plus 依赖 -->
<dependency>
    <groupId>com.baomidou</groupId>
    <artifactId>mybatis-plus-boot-starter</artifactId>
    <version>3.4.3.4</version>
</dependency>
```

然后，创建启动类：

```java
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
@SpringBootApplication
public class MybatisPlusApplication {
    public static void main(String[] args) {
        SpringApplication.run(MybatisPlusApplication.class, args);
    }
}
```

接下来，定义实体类 `User` 并添加乐观锁字段：

```java
import com.baomidou.mybatisplus.annotation.*;
import lombok.Data;
import java.util.Date;
@Data
@TableName("user")
public class User {
    @TableId
    private Long id;
    private String name;
    private Integer age;
    private String email;
    @TableField(fill = FieldFill.INSERT)
    private Date createTime;
    @TableField(fill = FieldFill.INSERT_UPDATE)
    private Date updateTime;

    // 乐观锁字段
    @Version
    private Integer version;
}
```

定义 Mapper 层：

```java
import com.baomidou.mybatisplus.core.mapper.BaseMapper;
import com.example.demo.entity.User;
public interface UserMapper extends BaseMapper<User> {
    // 这里可以添加自定义的数据库操作方法
}
```

定义 Service 层：

```
import com.baomidou.mybatisplus.extension.service.IService;
import com.baomidou.mybatisplus.extension.service.impl.ServiceImpl;
import com.example.demo.entity.User;
import com.example.demo.mapper.UserMapper;
public interface UserService extends IService<User> {
    // 这里可以添加自定义的业务方法
}
@Service
public class UserServiceImpl extends ServiceImpl<UserMapper, User> implements
UserService {
    // 这里可以添加业务逻辑的实现
}
```

最后，创建一个测试类来演示乐观锁的使用：

```
import com.example.demo.entity.User;
import com.example.demo.service.UserService;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
@SpringBootTest
public class UserServiceTest {
    @Autowired
    private UserService userService;
    @Test
    public void testOptimisticLock() {
        // 假设我们从数据库中查询到一个用户对象
        User user = userService.getById(1L);
        user.setName("New Name");

        // 在业务逻辑中修改了用户信息，准备更新到数据库
        // 由于使用了乐观锁，这里会检查 version 字段
        boolean result = userService.updateById(user);
        if (result) {
            System.out.println("Update successfully.");
        } else {
            System.out.println("Update failed, data has been modified by
others.");
        }
    }
}
```

在这个测试类中，我们通过 `getById` 方法查询出一个用户对象，然后修改其 `name` 属性。当调用 `updateById` 方法更新用户信息时，MyBatis-Plus 会自动检查乐观锁字段 `version`。如果 `version` 字段与数据库中的值相同，更新操作将成功，并且 `version` 字段的值会增加。如果 `version` 字段的值在读取和更新之间被其他事务修改，更新操作将失败。

请注意，为了使乐观锁功能正常工作，数据库表 `user` 需要有一个 `version` 字段，并且该字段的类型应该支持自增操作，通常为 `INTEGER` 类型。

以上代码仅为示例，实际项目中可能需要根据具体需求进行调整。

8.序列化器：

以下是一个基于 MyBatis-Plus 与 Spring Boot 的项目示例，其中包含启动类、Mapper 层、Service 层以及一个测试类。在这个示例中，我将重点展示如何使用 MyBatis-Plus 的序列化器。
首先，添加 MyBatis-Plus 的依赖到你的 `pom.xml` 文件中：

```
<!-- MyBatis-Plus 依赖 -->
<dependency>
    <groupId>com.baomidou</groupId>
    <artifactId>mybatis-plus-boot-starter</artifactId>
    <version>3.4.3.4</version>
</dependency>
```

然后，创建启动类：

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
@SpringBootApplication
public class MybatisPlusApplication {
    public static void main(String[] args) {
        SpringApplication.run(MybatisPlusApplication.class, args);
    }
}
```

定义实体类 `User`：

```
import com.baomidou.mybatisplus.annotation.*;
import com.baomidou.mybatisplus.extension.handlers.FastjsonTypeHandler;
import com.alibaba.fastjson.JSON;
import com.alibaba.fastjson.serializer.SerializeConfig;
import lombok.Data;
import java.io.Serializable;
import java.util.Date;
@Data
@TableName("user")
public class User implements Serializable {
    @TableId
    private Long id;
    private String name;
    private Integer age;
    private String email;
    @TableField(fill = FieldFill.INSERT)
    private Date createTime;
    @TableField(fill = FieldFill.INSERT_UPDATE)
    private Date updateTime;
    // 序列化器字段，假设这里是一个复杂的类型
    @TableField(typeHandler = FastjsonTypeHandler.class)
    private SerializeConfig serializeConfig;
}
```

定义 Mapper 层：

```
import com.baomidou.mybatisplus.core.mapper.BaseMapper;
import com.example.demo.entity.User;
public interface UserMapper extends BaseMapper<User> {
    // 这里可以添加自定义的数据库操作方法
}
```

定义 Service 层：

```
import com.baomidou.mybatisplus.extension.service.IService;
import com.baomidou.mybatisplus.extension.service.impl.ServiceImpl;
import com.example.demo.entity.User;
import com.example.demo.mapper.UserMapper;
public interface UserService extends IService<User> {
    // 这里可以添加自定义的业务方法
}
@Service
public class UserServiceImpl extends ServiceImpl<UserMapper, User> implements
UserService {
    // 这里可以添加业务逻辑的实现
}
```

最后，创建一个测试类来演示序列化器的使用：

```
import com.alibaba.fastjson.JSON;
import com.example.demo.entity.User;
import com.example.demo.service.UserService;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
@SpringBootTest
public class UserServiceTest {
    @Autowired
    private UserService userService;
    @Test
    public void testSerialization() {
        // 创建一个用户对象
        User user = new User();
        user.setName("Test User");
        user.setAge(30);
        user.setEmail("test@example.com");
        user.setCreateTime(new Date());
        user.setUpdateTime(new Date());

        // 假设 SerializeConfig 是一个复杂的类型需要序列化
        SerializeConfig serializeConfig = new SerializeConfig();
        user.setSerializeConfig(serializeConfig);
        // 保存用户信息到数据库
        userService.save(user);
        // 查询用户信息
        User savedUser = userService.getById(user.getId());
        // 输出序列化后的信息
        System.out.println(JSON.toJSONString(savedUser.getSerializeConfig()));
    }
}
```

在这个测试类中，我们创建了一个 `User` 对象，并为其设置了 `SerializeConfig` 字段，该字段使用了 `FastjsonTypeHandler` 来处理序列化。在保存用户信息到数据库后，我们通过 `getById` 方法查询出用户信息，并使用 Fastjson 的 `JSON.toJSONString` 方法来序列化 `SerializeConfig` 字段。
请注意，为了使用 `FastjsonTypeHandler`，你需要确保你的项目中已经添加了 Fastjson 的依赖，并且在实体类中正确地使用了 `@TableField(typeHandler = FastjsonTypeHandler.class)` 注解。
以上代码仅为示例，实际项目中可能需要根据具体需求进行调整。

9.数据权限：

以下是一个基于 MyBatis-Plus 与 Spring Boot 的项目示例，其中包含启动类、Mapper 层、Service 层以及一个测试类。在这个示例中，我将重点展示如何使用 MyBatis-Plus 的数据权限功能。
首先，添加 MyBatis-Plus 的依赖到你的 `pom.xml` 文件中：

```
<!-- MyBatis-Plus 依赖 -->
<dependency>
    <groupId>com.baomidou</groupId>
    <artifactId>mybatis-plus-boot-starter</artifactId>
    <version>3.4.3.4</version>
</dependency>
```

然后，创建启动类：

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
@SpringBootApplication
public class MybatisPlusApplication {
    public static void main(String[] args) {
        SpringApplication.run(MybatisPlusApplication.class, args);
    }
}
```

定义实体类 `User`：

```
import com.baomidou.mybatisplus.annotation.*;
import lombok.Data;
import java.io.Serializable;
@Data
@TableName("user")
public class User implements Serializable {
    @TableId
    private Long id;
    private String name;
    private Integer age;
    private String email;
    @TableField(fill = FieldFill.INSERT)
    private Date createTime;
    @TableField(fill = FieldFill.INSERT_UPDATE)
    private Date updateTime;
    // 假设有一个部门字段，用于数据权限控制
    private Long deptId;
}
```

定义 Mapper 层：

```
import com.baomidou.mybatisplus.core.mapper.BaseMapper;
import com.example.demo.entity.User;
public interface UserMapper extends BaseMapper<User> {
    // 这里可以添加自定义的数据库操作方法
}
```

定义 Service 层：

```
import com.baomidou.mybatisplus.extension.service.IService;
import com.baomidou.mybatisplus.extension.service.impl.ServiceImpl;
import com.example.demo.entity.User;
import com.example.demo.mapper.UserMapper;
import org.springframework.stereotype.Service;
@Service
public class UserService extends ServiceImpl<UserMapper, User> implements
IService<User> {
    // 这里可以添加自定义的业务方法
}
```

为了实现数据权限，我们需要自定义一个数据权限处理器，并注册到 MyBatis-Plus 的配置中：

```
import
com.baomidou.mybatisplus.extension.plugins.inner.DataPermissionInterceptor;
import com.baomidou.mybatisplus.extension.plugins.inner.InnerInterceptor;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
@Configuration
public class MybatisPlusConfig {
    @Bean
    public InnerInterceptor dataPermissionInterceptor() {
        return new DataPermissionInterceptor((sql, tableName) -> {
            // 假设当前登录用户属于部门ID为1，则只查询部门ID为1的数据
            Long deptId = 1L; // 应该从当前登录用户信息中获取
            if ("user".equals(tableName)) {
                sql.WHERE("dept_id = #{deptId}");
            }
        });
    }
}
```

最后，创建一个测试类来演示数据权限的使用：

```
import com.example.demo.service.UserService;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
@SpringBootTest
public class UserServiceTest {
    @Autowired
    private UserService userService;
    @Test
    public void testDataPermission() {
        // 假设当前用户属于部门ID为1
```

```
        Long deptId = 1L;
        // 查询当前部门下的所有用户
        List<User> users = userService.lambdaQuery()
                .eq(User::getDeptId, deptId)
                .list();
        // 输出查询结果
        users.forEach(user -> System.out.println(user.getName()));
    }
}
```

在这个测试类中，我们通过 `lambdaQuery` 方法构建了一个查询条件，只查询 `deptId` 等于当前用户所在部门的用户信息。由于我们已经在 MyBatis-Plus 配置中添加了数据权限处理器，这个查询会自动应用数据权限过滤。
请注意，上面的数据权限处理器示例非常简单，并且假设了一个固定的部门ID。在实际应用中，你需要根据实际业务逻辑和登录用户信息来动态构建数据权限过滤条件。
以上代码仅为示例，实际项目中可能需要根据具体需求进行调整。

10.多数据源：

以下是一个基于 MyBatis-Plus 与 Spring Boot 的项目示例，其中包含启动类、Mapper 层、Service 层以及一个测试类。在这个示例中，我将重点展示如何配置和使用 MyBatis-Plus 的多数据源功能。
首先，添加 MyBatis-Plus 的依赖到你的 `pom.xml` 文件中：

```
<!-- MyBatis-Plus 依赖 -->
<dependency>
    <groupId>com.baomidou</groupId>
    <artifactId>mybatis-plus-boot-starter</artifactId>
    <version>3.4.3.4</version>
</dependency>
<!-- Druid 数据源依赖 -->
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid-spring-boot-starter</artifactId>
    <version>1.2.8</version>
</dependency>
```

然后，创建启动类：

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
@SpringBootApplication
public class MybatisPlusApplication {
    public static void main(String[] args) {
        SpringApplication.run(MybatisPlusApplication.class, args);
    }
}
```

接下来，定义实体类 `User`：

```
import com.baomidou.mybatisplus.annotation.*;
import lombok.Data;
@Data
@TableName("user")
```

```
public class User {
    @TableId
    private Long id;
    private String name;
    private Integer age;
    private String email;
    @TableField(fill = FieldFill.INSERT)
    private Date createTime;
    @TableField(fill = FieldFill.INSERT_UPDATE)
    private Date updateTime;
}
```

定义 Mapper 层:

```
import com.baomidou.mybatisplus.core.mapper.BaseMapper;
import com.example.demo.entity.User;
public interface UserMapper extends BaseMapper<User> {
    // 这里可以添加自定义的数据库操作方法
}
```

定义 Service 层:

```
import com.baomidou.mybatisplus.extension.service.IService;
import com.baomidou.mybatisplus.extension.service.impl.ServiceImpl;
import com.example.demo.entity.User;
import com.example.demo.mapper.UserMapper;
public interface UserService extends IService<User> {
    // 这里可以添加自定义的业务方法
}
@Service
public class UserServiceImpl extends ServiceImpl<UserMapper, User> implements
UserService {
    // 这里可以添加业务逻辑的实现
}
```

配置多数据源，创建一个配置类 `MybatisPlusConfig`:

```
import com.baomidou.mybatisplus.extension.spring.MybatisSqlSessionFactoryBean;
import org.apache.ibatis.session.SqlSessionFactory;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Primary;
import javax.sql.DataSource;
@Configuration
public class MybatisPlusConfig {
    @Bean(name = "primaryDataSource")
    @Primary
    public DataSource dataSource1() {
        // 配置第一个数据源
        return // Druid数据源配置...
    }
    @Bean(name = "secondaryDataSource")
    public DataSource dataSource2() {
```

```
        // 配置第二个数据源
        return // Druid数据源配置...
    }
    @Bean(name = "primarySqlSessionFactory")
    @Primary
    public SqlSessionFactory sqlSessionFactory1(@Qualifier("primaryDataSource")
DataSource dataSource) throws Exception {
        MybatisSqlSessionFactoryBean sqlSessionFactoryBean = new
MybatisSqlSessionFactoryBean();
        sqlSessionFactoryBean.setDataSource(dataSource);
        // 其他配置...
        return sqlSessionFactoryBean.getObject();
    }
    @Bean(name = "secondarySqlSessionFactory")
    public SqlSessionFactory sqlSessionFactory2(@Qualifier("secondaryDataSource")
DataSource dataSource) throws Exception {
        MybatisSqlSessionFactoryBean sqlSessionFactoryBean = new
MybatisSqlSessionFactoryBean();
        sqlSessionFactoryBean.setDataSource(dataSource);
        // 其他配置...
        return sqlSessionFactoryBean.getObject();
    }
}
```

最后，创建一个测试类来演示多数据源的使用：

```
import com.example.demo.service.UserService;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.boot.test.context.SpringBootTest;
@SpringBootTest
public class UserServiceTest {
    @Autowired
    @Qualifier("primaryUserService") // 假设你定义了两个UserService实例，分别对应不同的
数据源
    private UserService primaryUserService;
    @Autowired
    @Qualifier("secondaryUserService")
    private UserService secondaryUserService;
    @Test
    public void testMultipleDataSources() {
        // 使用第一个数据源
        primaryUserService.save(new User());
        // 使用第二个数据源
        secondaryUserService.save(new User());
    }
}
```

在这个测试类中，我们通过 `@Qualifier` 注解指定了两个不同的 `UserService` 实例，分别对应不同的数据源。这样，在测试方法中，我们可以分别调用这两个服务实例来操作不同的数据源。
请注意，为了实现多数据源，你需要确保每个数据源都有相应的配置，并且创建了对应的 `SqlSessionFactory`。在实际情况中，你可能还需要配置事务管理器、动态数据源切换等。
以上代码仅为示例，

11.SQL 注入器：

以下是基于Mybatis-plus与SpringBoot的启动类、Mapper层、Service层以及测试类的示例代码。我将重点展示如何使用Mybatis-plus的SQL注入器。

1. 启动类（Application.java）

```
package com.example.demo;
import org.mybatis.spring.annotation.MapperScan;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
@SpringBootApplication
@MapperScan("com.example.demo.mapper")
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

2. Mapper层（UserMapper.java）

```
package com.example.demo.mapper;
import com.baomidou.mybatisplus.core.mapper.BaseMapper;
import com.example.demo.entity.User;
public interface UserMapper extends BaseMapper<User> {
    // 这里可以自定义方法，Mybatis-plus会自动注入SQL
}
```

3. SQL注入器（MySqlInjector.java）

```
package com.example.demo.config;
import com.baomidou.mybatisplus.core.injector.AbstractMethod;
import com.baomidou.mybatisplus.core.injector.DefaultSqlInjector;
import com.baomidou.mybatisplus.core.metadata.TableInfo;
import java.util.List;
public class MySqlInjector extends DefaultSqlInjector {
    @Override
    public List<AbstractMethod> getMethodList(Class<?> mapperClass, TableInfo
tableInfo) {
        List<AbstractMethod> methodList = super.getMethodList(mapperClass,
tableInfo);
        // 这里可以添加自定义方法
        methodList.add(new FindAllMethod());
        return methodList;
    }
}
```

自定义方法示例（FindAllMethod.java）：

```
package com.example.demo.config;
import com.baomidou.mybatisplus.core.injector.AbstractMethod;
import org.apache.ibatis.mapping.MappedStatement;
import org.apache.ibatis.mapping.SqlSource;
public class FindAllMethod extends AbstractMethod {
    @Override
    public MappedStatement injectMappedStatement(Class<?> mapperClass, Class<?>
modelClass, TableInfo tableInfo) {
        String sql = "SELECT * FROM " + tableInfo.getTableName();
        SqlSource sqlSource = languageDriver.createSqlSource(configuration, sql,
modelClass);
        return this.addSelectMappedStatementForTable(mapperClass, "findAll",
sqlSource, tableInfo);
    }
}
```

4. Service层（UserService.java）

```
package com.example.demo.service;
import com.baomidou.mybatisplus.extension.service.IService;
import com.example.demo.entity.User;
public interface UserService extends IService<User> {
    // 这里可以添加自定义方法
}
```

5. Service实现类（UserServiceImpl.java）

```
package com.example.demo.service.impl;
import com.baomidou.mybatisplus.extension.service.impl.ServiceImpl;
import com.example.demo.entity.User;
import com.example.demo.mapper.UserMapper;
import com.example.demo.service.UserService;
import org.springframework.stereotype.Service;
@Service
public class UserServiceImpl extends ServiceImpl<UserMapper, User> implements
UserService {
    // 这里可以重写或添加自定义方法
}
```

6. 测试类（UserMapperTest.java）

```
package com.example.demo;
import com.example.demo.entity.User;
import com.example.demo.mapper.UserMapper;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
@SpringBootTest
public class UserMapperTest {
    @Autowired
    private UserMapper userMapper;
    @Test
```

```
    public void testFindAll() {
        List<User> userList = userMapper.findAll();
        userList.forEach(System.out::println);
    }
}
```

在上述代码中，我们自定义了一个SQL注入器`MySqlInjector`，并在其中添加了一个`FindAllMethod`方法。这样，我们就可以在Mapper层使用`findAll`方法，而无需编写XML或注解。这个示例展示了Mybatis-plus的强大功能，可以方便地进行SQL注入。

7.SQL注入器在Mybatis-plus框架中有以下应用价值：

1. **扩展性**：SQL注入器允许开发者在不修改现有Mapper接口和XML文件的情况下，动态添加自定义的SQL方法。这大大提高了代码的扩展性，使得在项目迭代过程中可以更加灵活地处理新的需求。
2. **简化代码**：通过SQL注入器，可以将通用的SQL操作封装成方法，减少重复代码的编写。比如，对于分页查询、批量插入、数据统计等操作，可以一次编写，多次使用。
3. **提高开发效率**：开发者可以利用SQL注入器快速实现一些常用的数据库操作，而不需要每次都编写详细的SQL语句和对应的映射，这样可以显著提高开发效率。
4. **减少错误**：手动编写SQL语句容易出错，尤其是在复杂的业务逻辑中。SQL注入器通过封装常用的SQL操作，减少了直接编写SQL语句的机会，从而降低了出错的可能性。
5. **统一管理**：通过SQL注入器，可以将所有自定义的SQL方法集中管理，便于维护和更新。当数据库结构发生变化时，只需要修改注入器中的相关方法即可。
6. **功能增强**：Mybatis-plus提供的默认方法可能无法满足所有业务需求，SQL注入器允许开发者根据业务需求定制特殊的SQL方法，增强了框架的功能。
7. **安全防护**：虽然SQL注入器本身不是用来防止SQL注入攻击的，但它通过预定义和封装SQL语句，减少了直接拼接SQL语句的风险，从而在某种程度上提高了应用程序的安全性。
8. **易于测试**：由于SQL方法被封装，因此在单元测试时，可以更容易地模拟和测试这些方法，而不需要直接操作数据库。
   总之，SQL注入器为Mybatis-plus框架带来了更大的灵活性和便捷性，使得数据库操作更加高效和安全，同时也为代码的维护和扩展提供了便利。