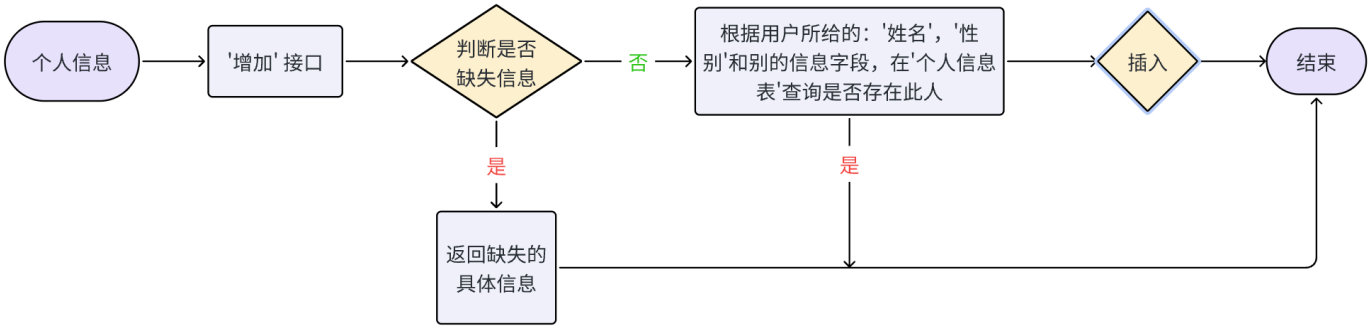
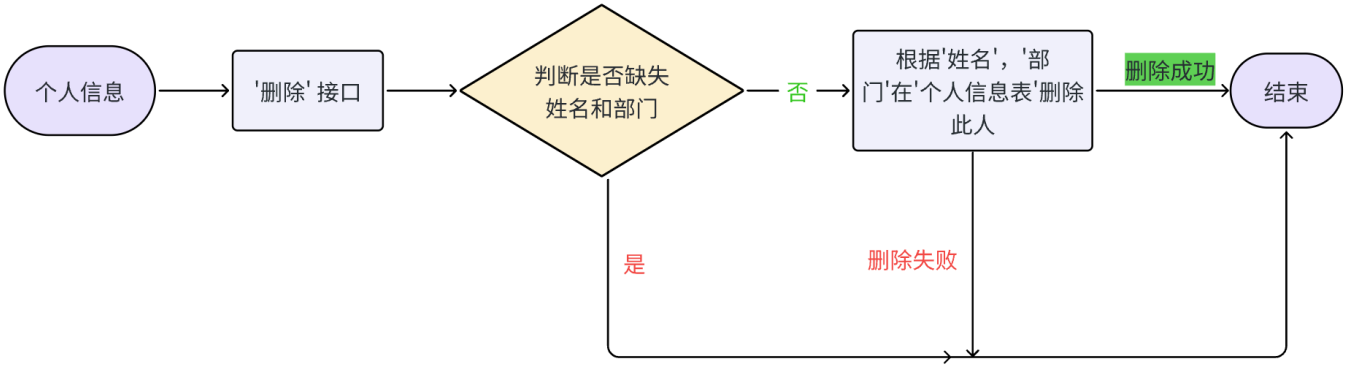
需求理解

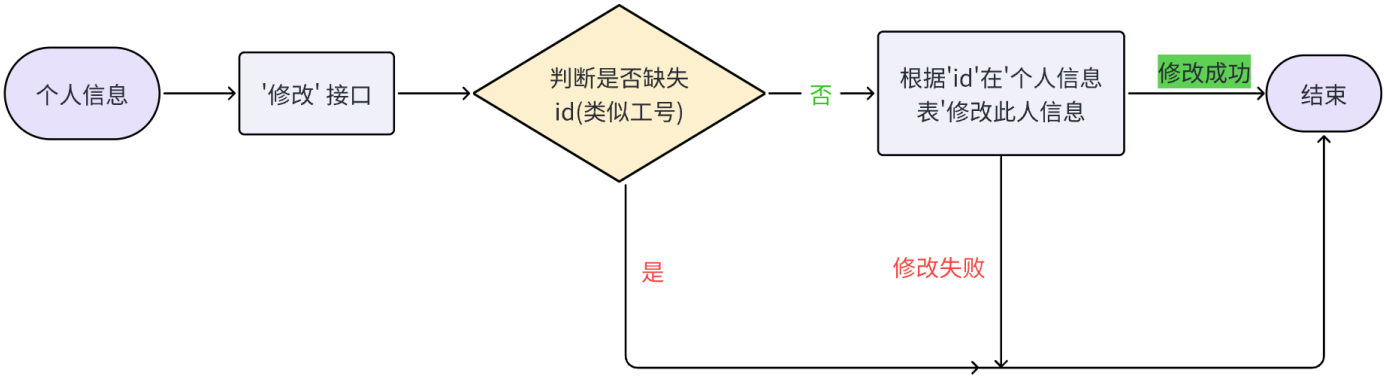
增加逻辑



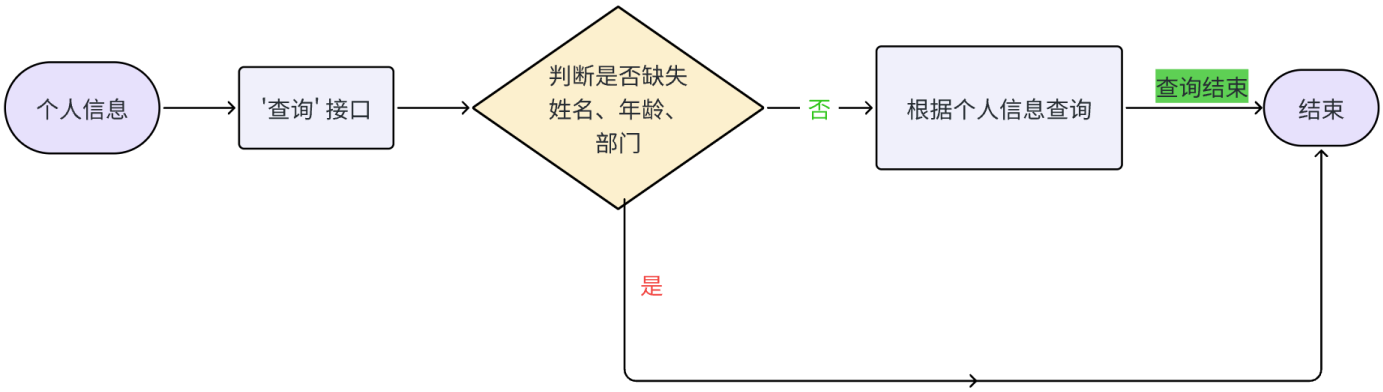
删除逻辑



修改逻辑



查询逻辑



demo说明文档

下面是实现个人信息增加、删除、修改、查询功能的SQL命令以及对应的MyBatis Mapper接口和XML配置文件。

首先，我们创建两个表：`person` 和 `department`。

1. 表 `person` 包含字段：`id`, `name`, `gender`, `age`, `occupation`。
2. 表 `department` 包含字段：`id`, `person_id`, `department_name`。

其中，`person` 表和 `department` 表通过 `person_id` 字段建立一对多的关系。

1. SQL命令：

```
-- 创建 person 表
CREATE TABLE person (
    id INT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(100) NOT NULL,
    gender VARCHAR(10) NOT NULL,
    age INT,
    occupation VARCHAR(100),
    is_deleted BOOLEAN DEFAULT false
);

-- 创建 department 表
CREATE TABLE department (
    id INT PRIMARY KEY AUTO_INCREMENT,
    person_id INT,
    department_name VARCHAR(100) NOT NULL,
    CONSTRAINT fk_person_id FOREIGN KEY (person_id) REFERENCES person(id)
);
```

2. Mapper接口和XML配置文件：

假设Mapper接口名为`PersonMapper`。

```
@Mapper
public interface PersonInfoMapper {
    // 新增个人信息
    boolean insertPerson(PersonInfo person);

    // 删除个人信息（逻辑删除）
    int softDeletePerson(PersonInfo person);

    // 修改个人信息
    int updatePerson(PersonInfo person);

    // 根据姓名、性别、年龄、部门（单个）查询个人信息
    List<PersonInfo> selectPersonsByConditions(String name, String gender, Integer age,
String department);

    // 根据姓名、性别、年龄、部门（单个）查询个人信（分页）
    List<PersonInfo> selectPersonsByConditionsLimited(String name, String gender,
Integer age, String department,
                                                      Integer offSet, Integer
pagesize);

}
```

`Person` 类为Java对象，包含个人信息的字段。

```
public class Person {
    private int id;
    private String name;
    private String gender;
    private Integer age;
    private String occupation;
    // Getters and setters
}
```

3. XML配置文件 `PersonMapper.xml`：

```
   <insert id="insertPerson" parameterType="com.example.nowcoder.entity.PersonInfo">
        INSERT INTO person_info (name, gender, age, occupation, department_id)
        VALUES (#{name}, #{gender}, #{age}, #{occupation}, #{departmentId})
    </insert>

    <!-- 删除个人信息（逻辑删除） -->
    <update id="softDeletePerson"
parameterType="com.example.nowcoder.entity.PersonInfo">
        UPDATE person_info
        SET is_deleted = 1
        WHERE name = #{name} and gender = #{gender}
    </update>
```

```
    <!-- 修改个人信息 -->
    <update id="updatePerson" parameterType="com.example.nowcoder.entity.PersonInfo">
        UPDATE person_info
        SET
        <if test="name != null">name = #{name},</if>
        <if test="gender != null">gender = #{gender},</if>
        <if test="age != null">age = #{age},</if>
        <if test="occupation != null">occupation = #{occupation},</if>
        <if test="departmentId != null">department_id = #{departmentId}</if>
        WHERE id = #{id}
    </update>

    <!-- 根据姓名、性别、年龄、部门（单个）查询个人信息 -->
    <select id="selectPersonsByConditions"
resultType="com.example.nowcoder.entity.PersonInfo">
        SELECT id, name, gender, age, occupation, department_id, is_deleted
        FROM person_info
        WHERE
        <if test="name != null">name = #{name}</if>
        <if test="gender != null">AND gender = #{gender}</if>
        <if test="age != null">AND age = #{age}</if>
        <if test="department != null">AND department_id IN (SELECT id FROM
department_info WHERE department_name = #{department})</if>
    </select>

    <select id="selectPersonsByConditionsLimited"
resultType="com.example.nowcoder.entity.PersonInfo">
        SELECT p.id, p.name, p.gender, p.age, p.occupation, d.department_name
        FROM person_info p
        LEFT JOIN department_info d ON p.department_id = d.id
        WHERE 1=1
        <if test="name != null and name != ''">AND p.name = #{name}</if>
        <if test="gender != null and gender != ''">AND p.gender = #{gender}</if>
        <if test="age != null">AND p.age = #{age}</if>
        <if test="department != null and department != ''">AND d.department_name = #
{department}</if>
        order by p.id
        LIMIT #{offSet}, #{pagesize}
    </select>
```

请注意，为了实现逻辑删除，我们在`person`表中添加了一个`is_deleted`字段，用于标记是否已删除。默认值为0（未删除），当执行删除操作时，将该字段的值设为1。

当进行查询时，我们可以在Mapper接口中对输入参数进行非空校验，**并在XML配置文件中使用MyBatis的动态SQL来处理条件**。下面是对查询接口进行非空校验的示例代码：

在上述代码中，我们在查询接口中对输入参数`name`、`gender`、`age`和`department`进行了非空校验。在XML配置文件中，我们使用MyBatis的动态SQL来根据条件动态拼接SQL查询语句。

以上就是实现个人信息的增加、删除、修改、查询功能的SQL命令以及对应的MyBatis Mapper接口和XML配置文件的示例。在实际应用中，你需要根据具体的数据库和业务需求进行适当的调整和扩展。

为了提高查询效率，添加唯一索引以及在插入时检查重复性。

修改 `person` 表：

```
-- 添加唯一索引
CREATE UNIQUE INDEX idx_person_name ON person (name);
```

接口的单元测试

为了编写单元测试，你需要一个单元测试框架（例如JUnit）和一个Mock框架（例如Mockito）来模拟数据库的行为。在这个示例中，我将使用JUnit和Mockito来演示单元测试。

下面是对HomeController的单元测试代码，并添加了注释说明：

```
@RunWith(MockitoJUnitRunner.class)
public class HomeControllerTest {

    @Mock
    private PersonalInfoService personalInfoService;

    @InjectMocks
    private HomeController homeController;

    @Before
    public void setUp() {
        // 可以在这里初始化一些Mock的行为
    }

    @Test
    public void testInsert_NoName() {
        PersonInfo personInfo = new PersonInfo();
        // 没有姓名和性别
        String expectedResult = "No name entered.";
        String result = homeController.insert(personInfo);
        assertEquals(expectedResult, result);
    }

    @Test
    public void testInsert_NoGender() {
        PersonInfo personInfo = new PersonInfo();
        personInfo.setName("John");
```

```
        // 没有性别
        String expectedResult = "No gender entered";
        String result = homeController.insert(personInfo);
        assertEquals(expectedResult, result);
    }

    @Test
    public void testInsert_Success() {
        PersonInfo personInfo = new PersonInfo();
        personInfo.setName("John");
        personInfo.setGender("Male");
        // 成功插入

when(personalInfoService.insertPersonInfo(any(PersonInfo.class))).thenReturn(true);
        String expectedResult = "true";
        String result = homeController.insert(personInfo);
        assertEquals(expectedResult, result);
    }

    @Test
    public void testDelete() {
        PersonInfo personInfo = new PersonInfo();
        // 模拟Service层的返回结果

when(personalInfoService.softDeletePerson(any(PersonInfo.class))).thenReturn(1);

        int expectedResult = 1;
        int result = homeController.delete(personInfo);
        assertEquals(expectedResult, result);

        // 验证Controller是否正确调用了Service层的方法
        verify(personalInfoService, times(1)).softDeletePerson(personInfo);
    }

    @Test
    public void testUpdate() {
        PersonInfo personInfo = new PersonInfo();
        // 模拟Service层的返回结果

when(personalInfoService.updatePersonInfo(any(PersonInfo.class))).thenReturn(1);

        int expectedResult = 1;
        int result = homeController.update(personInfo);
        assertEquals(expectedResult, result);

        // 验证Controller是否正确调用了Service层的方法
        verify(personalInfoService, times(1)).updatePersonInfo(personInfo);
    }
```

```
    @Test
    public void testSelectPersonsByConditions() {
        String name = "John";
        String gender = "Male";
        int age = 30;
        String department = "Engineering";

        // 模拟Service层的返回结果
        List<PersonInfo> mockResult = new ArrayList<>();
        mockResult.add(new PersonInfo("John", "Male", 30, "Engineer", 2));
        when(personalInfoService.selectPersonsByConditions(name, gender, age,
department)).thenReturn(mockResult);

        List<PersonInfo> result = homeController.selectPersonsByConditions(name,
gender, age, department);
        assertEquals(mockResult, result);

        // 验证Controller是否正确调用了Service层的方法
        verify(personalInfoService, times(1)).selectPersonsByConditions(name, gender,
age, department);
    }

    @Test
    public void testSelectPersonsByConditionsLimited() {
        String name = "John";
        String gender = "Male";
        int age = 30;
        String department = "Engineering";
        int pageNum = 1;
        int pageSize = 10;

        // 模拟Service层的返回结果
        List<PersonInfo> mockResult = new ArrayList<>();
        mockResult.add(new PersonInfo("John", "Male", 30, "Engineer", 2));
        when(personalInfoService.selectPersonsByConditionsLimited(name, gender, age,
department, pageNum, pageSize)).thenReturn(mockResult);

        List<PersonInfo> result = homeController.selectPersonsByConditionsLimited(name,
gender, age, department, pageNum, pageSize);
        assertEquals(mockResult, result);

        // 验证Controller是否正确调用了Service层的方法
        verify(personalInfoService, times(1)).selectPersonsByConditionsLimited(name,
gender, age, department, pageNum, pageSize);
    }
}
```

在这个示例中，使用了Mockito来模拟PersonalInfoService的行为。在每个测试方法中，通过`when()`和`thenReturn()`方法来设置PersonalInfoService的模拟对象的行为，然后调用HomeController的方法进行测试，并使用断言来验证结果是否正确。

注意，这里的单元测试主要测试HomeController的逻辑，不涉及PersonalInfoService的实际连接和数据库操作。通过Mockito的模拟对象，我们可以更加专注于HomeController的行为和逻辑的测试，而无需真正依赖外部服务。在实际项目中，可以根据具体需求编写更全面的测试用例，以覆盖更多的场景和逻辑分支。