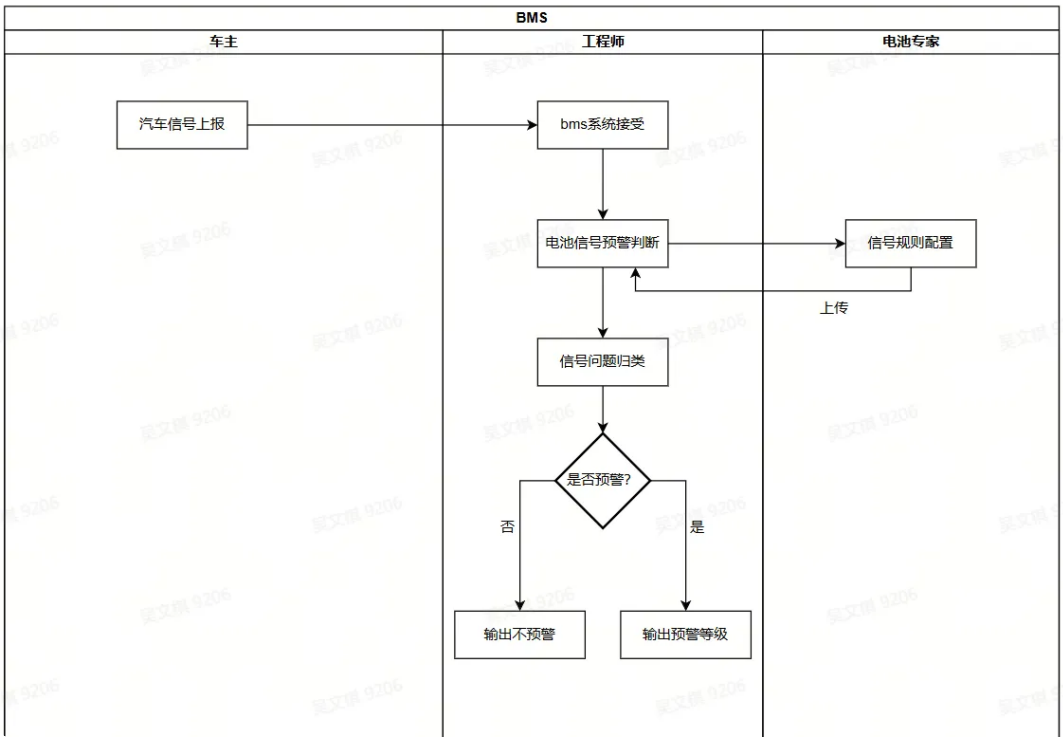
小米大作业-方案设计

1. 系统设计



1.1. 需求分析

1. 汽车信息模块，需要实现：
    a. 车辆信息的上传，车辆vid、车架编号、电池类型、总里程、电池健康度。
2. 电池模块，需要实现：
    a. 电池信息的上传，信息保存到数据库。
    b. 电池信息查询，数据量大。需要使用分页查询，缓存查询结果，并保持数据一致。
    c. 电池信息删除，根据信息记录 id 和车架 carId 进行删除，删除需判断信息所属的车辆。
3. 电池专家模块，需要实现：
    a. 配置电池信号的预警规则，考虑可拓展性。
4. 预警模块，需要实现：
    a. 定时扫描电池信号表，解析电池信号，转换为预警信息保存到数据库，需要注意：
        i. 电池信号数据量较大，需要控制对数据库的 IO 操作。
        ii. 定时任务的启动时间设定为系统压力较小的时间段。

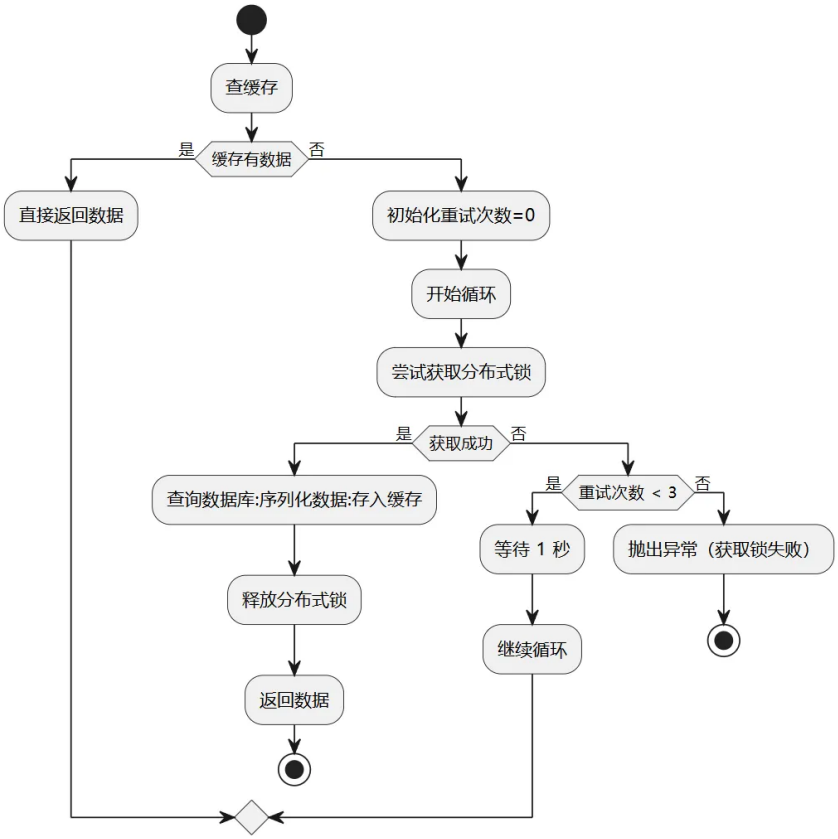
1.2. 具体思路

核心点分为：电池信息数据缓存、定时任务设计、预警规则动态解析

1.2.1. 电池信息数据缓存设计

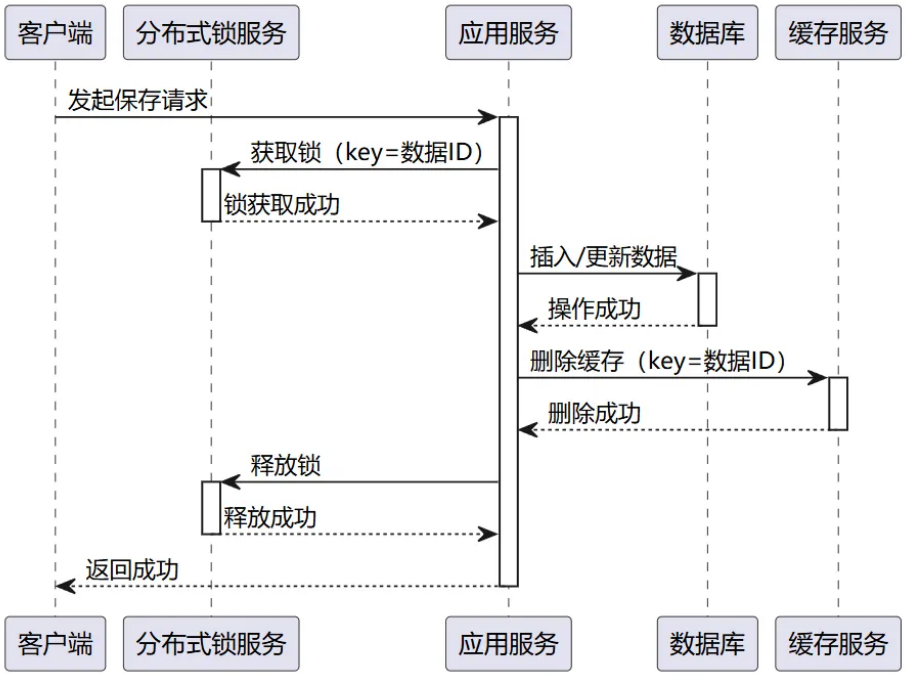
1.2.1.1. 查询流程：

1. 先查缓存，缓存有数据直接返回
2. 缓存没有，尝试获取分布式锁（失败则重复，重复三次，每一等待一秒）
3. 获取到分布式锁后，查询数据库，将数据进行序列化，存入缓存
4. 释放分布式锁



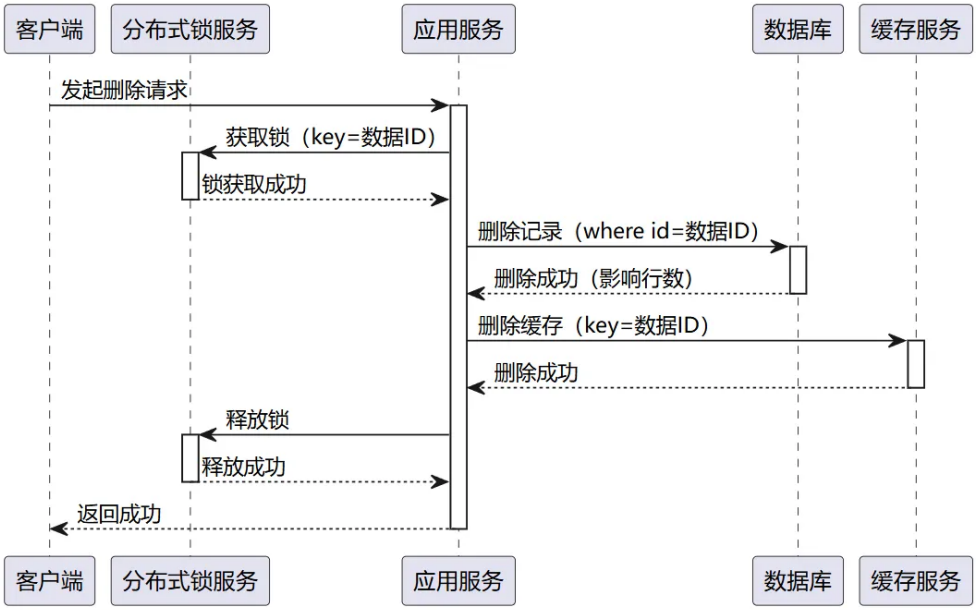
1.2.1.2. 保存流程：

1. 获取分布式锁
2. 将数据保存到数据库
3. 删除缓存
4. 释放分布式锁



1.2.1.3. 删除流程：

1. 获取分布式锁
2. 先删除 数据库，再删除缓存
3. 释放分布式锁



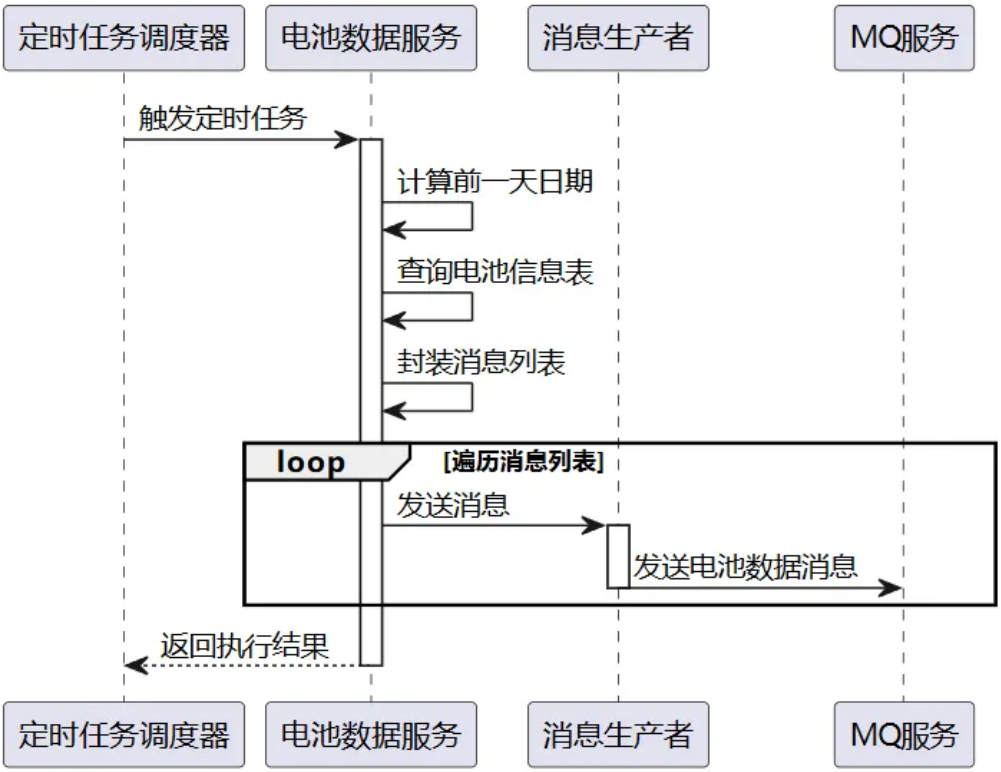
1.2.2. 定时任务设计

预警模块中定义一个定时任务，任务执行步骤：

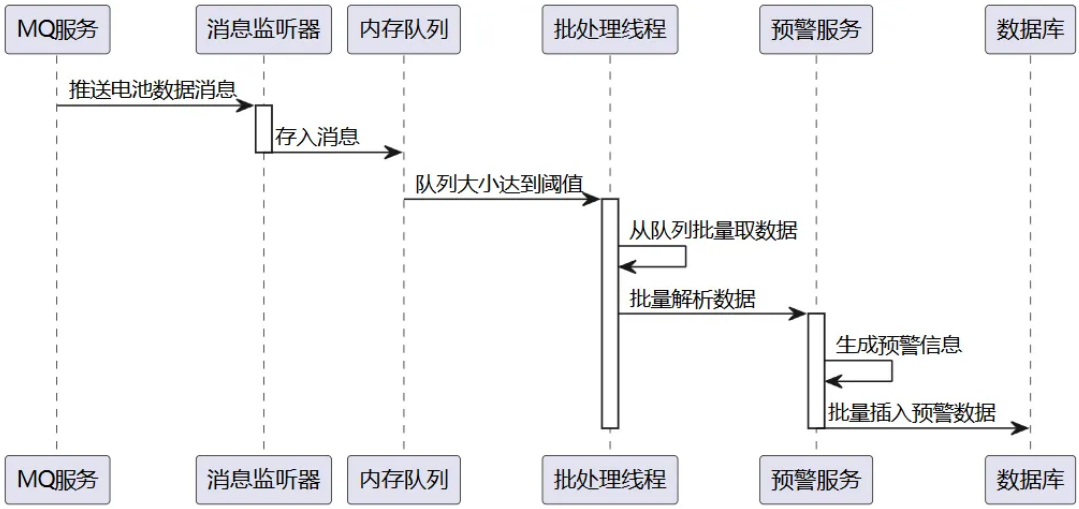
1. 根据当前日期查询前一天的电池信息表，获取电池信息
2. 通过 mq 进行推送。不考虑消息推送的顺序性，配置为 MessageListenerConcurrently 异步发送
3. 消费者端接受数据，设计一个队列存储接收到的数据，运行在另一个线程，确保消费者线程正常
4. 队列中的数据个数到达设定的批处理数量，即进行批处理
5. 批量解析电池数据获取预警信息
6. 将预警信息批量存储至数据库，减少对数据库的 IO 次数

个人认为此场景不用考虑幂等消费问题，去掉幂等校验能减少数据库查询次数、简化业务逻辑

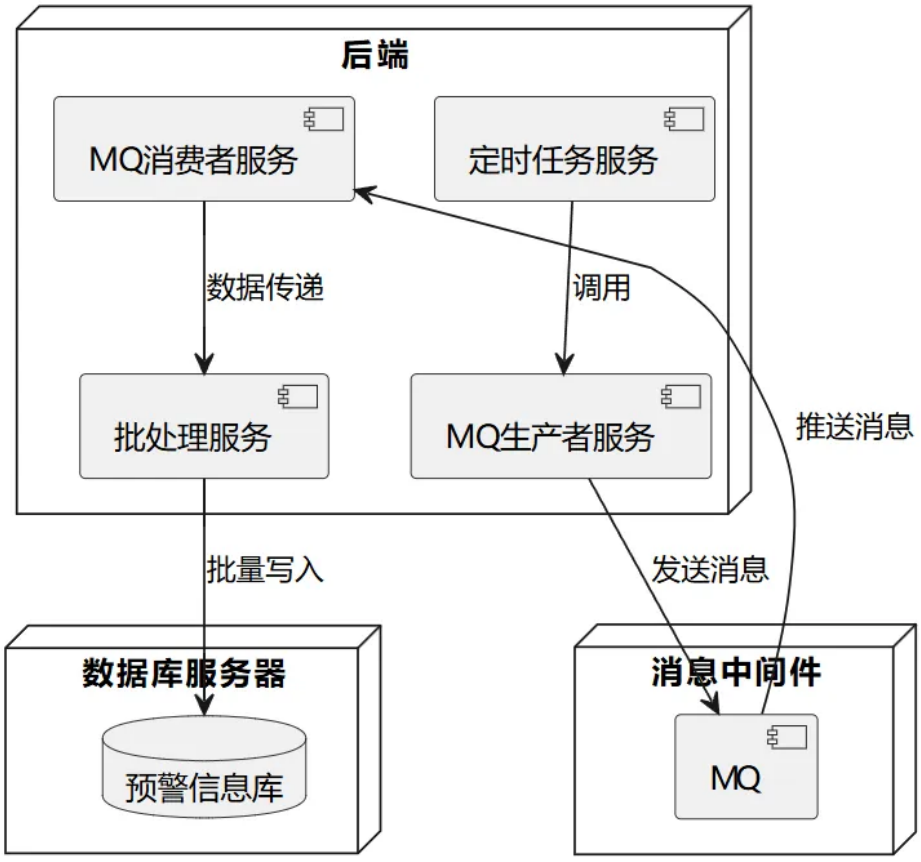
生产者端：



消费者端：

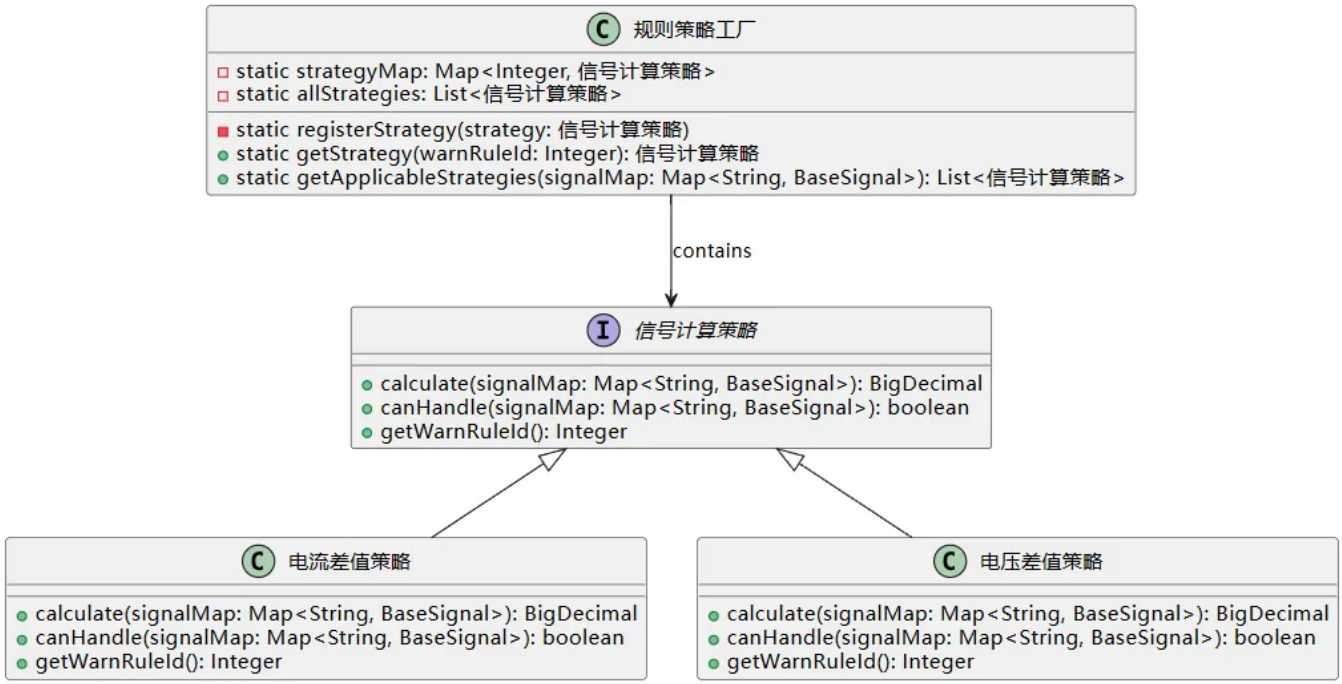


定时任务组件结构：

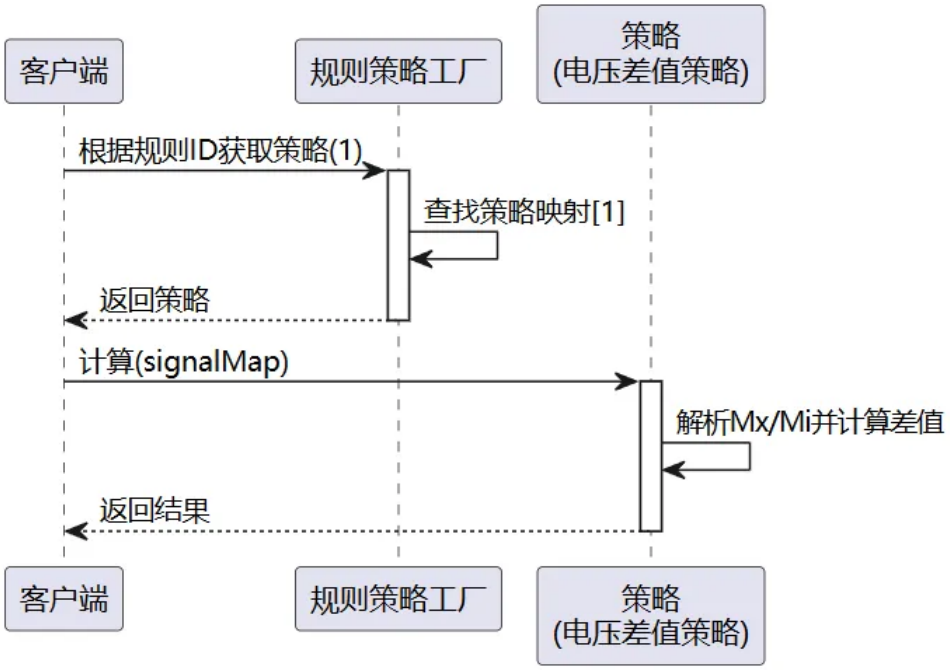


1.2.3. 预警规则解析设计

设计策略工厂模式，不同预警规则的解析实现放在具体的策略中。满足开闭原则，可拓展不同的预警规则。



规则解析时序图



2. 数据库表设计

总共设计四张表：车辆信息表、电池信号记录表、预警规则表、预警信息记录表

2.1. 车辆信息表

车辆信息表 SQL

```sql
CREATE TABLE `car_message` (
  `vid` BIGINT ( 20 ) NOT NULL COMMENT '车辆识别码',
  `frame_number` VARCHAR ( 255 ) DEFAULT NULL COMMENT '车架编号',
  `battery_type` VARCHAR ( 255 ) DEFAULT NULL COMMENT '电池类型',
  `total_mileage` VARCHAR ( 255 ) DEFAULT NULL COMMENT '总里程',
  `battery_health_percent` INT ( 11 ) DEFAULT NULL COMMENT '电池健康度百分比(%) ',
  `is_deleted` TINYINT ( 1 ) DEFAULT '0' COMMENT '是否删除',
  `create_time` DATETIME DEFAULT CURRENT_TIMESTAMP COMMENT '创建时间',
  `update_time` DATETIME DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP COMMENT '修改时间',
  PRIMARY KEY ( `vid` ) ,
  KEY idx_fNumber (frame_number)
) ENGINE = InnoDB DEFAULT CHARSET = utf8mb4 COMMENT = '车辆消息表';
```

2.2. 电池信号记录表

创建索引：1.idx_carId (car_id)　2.idx_cId_rId (car_id, warn_rule_id)　3.idx_cTime (create_time)

电池信号记录表 SQL

```sql
CREATE TABLE battery_signal_records (
    id BIGINT(20) UNSIGNED NOT NULL AUTO_INCREMENT COMMENT '主键ID',
    car_id BIGINT ( 20 ) NOT NULL COMMENT '车架编号',
    warn_rule_id INT NOT NULL COMMENT '预警规则ID',
    battery_signal TEXT NOT NULL COMMENT '电池信号json字符串',
    is_deleted TINYINT(1) UNSIGNED NOT NULL DEFAULT 0 COMMENT '是否删除(0-未删除,1-已删除)',
    create_time DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP COMMENT '记录创建时间',
    update_time DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP COMMENT '记录更新时间',
    PRIMARY KEY (id),
    KEY idx_carId (car_id),
    KEY idx_cId_rId (car_id, warn_rule_id),
    KEY idx_cTime (create_time)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COMMENT='电池信号记录表';
```

2.3. 预警规则表

预警规则表 SQL

```sql
CREATE TABLE battery_warn_rules (
    id BIGINT(20) NOT NULL COMMENT '主键',
    warn_rule_id INT NOT NULL COMMENT '预警规则ID',
    warn_name VARCHAR(50) NOT NULL COMMENT '预警规则名称',
    warn_max_value DECIMAL(5,2) COMMENT '报警最大值',
    warn_min_value DECIMAL(5,2) COMMENT '报警最小值',
    warn_level INT NOT NULL COMMENT '报警等级',
    battery_type VARCHAR(20) NOT NULL COMMENT '电池类型',
    is_deleted TINYINT(1) DEFAULT '0' COMMENT '是否删除',
    create_time DATETIME DEFAULT CURRENT_TIMESTAMP COMMENT '创建时间',
    update_time DATETIME DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP COMMENT '修改时间',
    PRIMARY KEY (id)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COMMENT='电池预警规则表';
```

2.4. 预警信息记录表

创建索引：1.idx_carId (car_id)　2.KEY idx_cId_wlevel (car_id, warn_level)

预警信息记录表 SQL

```sql
CREATE TABLE battery_warning_records (
    id BIGINT(20) NOT NULL COMMENT '主键',
    car_id BIGINT ( 20 ) NOT NULL COMMENT '车架编号',
    battery_type VARCHAR(20) NOT NULL COMMENT '电池类型',
    warn_name VARCHAR(50) NOT NULL COMMENT '预警规则名称',
    warn_level INT NOT NULL COMMENT '报警等级(0-4)',
    is_deleted TINYINT(1) DEFAULT '0' COMMENT '是否删除(0-未删除,1-已删除)',
    create_time DATETIME DEFAULT CURRENT_TIMESTAMP COMMENT '创建时间',
    update_time DATETIME DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP COMMENT '修改时间',
    PRIMARY KEY (id),
    KEY idx_cid (car_id),
    KEY idx_cId_wlevel (car_id, warn_level)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COMMENT='电池预警信息记录表';
```

3. 接口设计

统一响应参数：

```java
/**
 * 通用响应结果
 * @param <T>
 */
@Data
@NoArgsConstructor
public class Response<T> {

    private Integer code;
    private String message;
    private Boolean success;
    private T data;

    // 成功的构造方法
    public Response(Integer code, String message, Boolean success, T data) {
        this.code = code;
        this.message = message;
        this.success = success;
        this.data = data;
    }

    // 失败的构造方法
    public Response(Integer code, String message) {
        this.code = code;
        this.message = message;
        this.success = false;
    }

    // 成功-有数据响应
    public static <T> Response<T> success(T data) {
        return new Response<>(200,  "success", true, data);
    }

    // 成功-无数据响应
    public static <Void> Response<Void> success() {
        return new Response<>(200,  "success", true, null);
    }

    // 失败-带自定义状态码
    public static <T> Response<T> fail(Integer code, String message) {
        return new Response<>(code, message);
    }

    // 失败-默认400状态码
    public static <T> Response<T> fail(String message) {
        return new Response<>(400, message);
```

```
47         }
48     }
```

- **汽车信息上传接口**

请求路径：/carMessage/saveCarMessage

接口方法：POST

请求参数:

字段名	类型	是否必填	描述	约束条件
`vid`	`Long`	是	车辆唯一识别码	非空
`frameNumber`	`String`	是	车架编号	非空
`batteryType`	`String`	是	电池类型	不能为空；使用枚举值
`totalMileage`	`String`	否	总里程	
`batteryHealthPercent`	`Integer`	是	电池健康度百分比	`0 ≤ 值 ≤ 100`

响应数据示例:

```
{
    "code": 200,   // 状态码
    "message": "success",   // 状态码，成功success，失败fail
    "success": true,   //是否成功
    "data": null    // 返回数据，无数据时为空
}
```

- **电池信息上传接口**

请求路径 ： /battery/uploadBatteryMessage

接口方法：POST

请求参数:

字段名	类型	是否必填	描述	约束条件
`carId`	`Long`	是	关联车辆的唯一标识	不能为空；需对应数据库中存在的车辆记录
`warnId`	`Long`	否	预警规则 id	为空的时候遍历所有预警规则
`batterySignal`	`String`	是	电池信号 JSON 字符串	需为合法 JSON 格式

响应参数:

字段名	类型	描述
`code`	`Integer`	状态码（200 = 成功，非 200 = 失败）
`message`	`String`	状态描述信息（成功 / 失败原因）
`success`	`Boolean`	请求是否成功
`data`	`Object`	请求响应数据

- **电池状态信息查询接口（分页）**

请求路径：/battery/queryBatteryMessageByCarId

接口方法：POST

请求参数:

字段名	类型	是否必填	描述	约束条件

`carId`	`Long`	是	车架编号（关联车辆的唯一标识）	不能为空；需对应数据库中存在的车辆记录
`pageNum`	`Integer`	是	页码（从 1 开始）	必须 ≥ 1
`pageSize`	`Integer`	是	每页数据量	必须 ≥ 1

响应参数示例:

```
{
      "code": 200,   // 状态码
      "message": "success",
      "success": true,
      "data": {   // 返回数据
          "pageNum": 1,  // 页码
          "pageSize": 10,  // 请求的每页数据量
          "totalPage": 1,  // 总页数
          "carId": 10,   // 车架唯一id
          "batterySignalList": [  // 响应数据
              {
                  "id": 13,
                  "warnId": 2,
                  "batterySignal": "{\"Ix\": 12.0, \"Ii\": 11.7}",
                  "recordTime": "2025-05-19T13:26:25"
              },
              {
                  "id": 25,
                  "warnId": null,
                  "batterySignal": "{\"Mx\":11.0,\"Mi\":9.6,\"Ix\":12.0,\"Ii
    \":11.7}",
                  "recordTime": "2025-05-19T17:35:37"
              }
          ]
      }
  }
```

- **电池信息删除接口**

请求路径：/battery/deleteBatteryMsgById

接口方法：POST

请求参数:

字段名	类型	是否必填	描述	约束条件
`msgId`	`Long`	是	电池信息记录 id	需对应数据库中存在的记录
`carId`	`Long`	是	车架编号（关联车辆的唯一标识）	与 `id` 关联的车辆记录一致

响应参数:

字段名	类型	描述
`code`	`Integer`	状态码（200 = 成功，非 200 = 失败）
`message`	`String`	状态描述信息（成功 / 失败原因）
`success`	`Boolean`	请求是否成功
`data`	`Object`	null

- **预警信息记录查询接口**

请求路径：/warn/getWarnRecords

接口方法：POST

请求数据:

字段名	类型	是否必填	描述
`carId`	`Long`	是	车架编号（关联车辆的唯一标识）

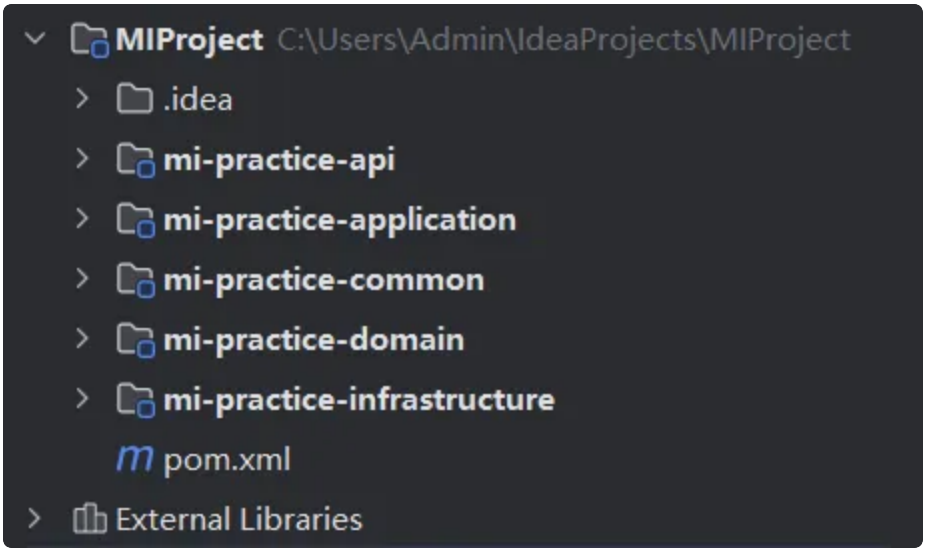
响应参数:

字段名	类型	描述
`code`	`Integer`	状态码（200 = 成功，非 200 = 失败）

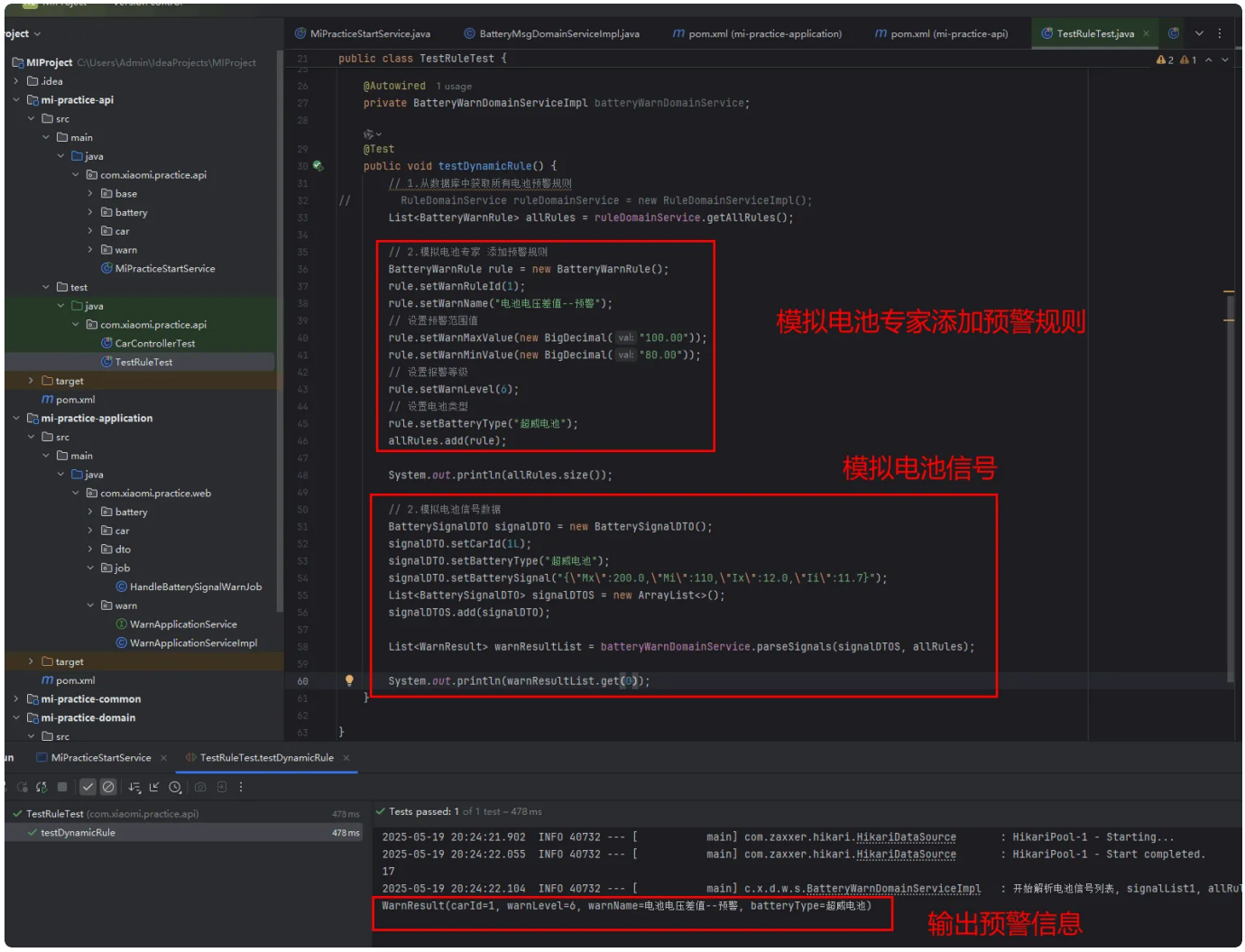
`message`	`String`	状态描述信息（成功 / 失败原因）
`success`	`Boolean`	请求是否成功
`data`	`Object`	null

4. 具体实现（截图 + 代码）

项目结构：DDD 思想的多模块项目架构



单元测试，模拟电池专家添加预警规则，并动态解析


模拟电池专家添加预警规则
模拟电池信号
输出预警信息

```java
    /**
     * 测试动态配置预警规则并解析
     */
    @Test
    public void testDynamicRule() {
        // 1.从数据库中获取所有电池预警规则
//        RuleDomainService ruleDomainService = new RuleDomainServiceImpl();
        List<BatteryWarnRule> allRules = ruleDomainService.getAllRules();

        // 2.模拟电池专家 添加预警规则
        BatteryWarnRule rule = new BatteryWarnRule();
        rule.setWarnRuleId(1);
        rule.setWarnName("电池电压差值--预警");
        // 设置预警范围值
        rule.setWarnMaxValue(new BigDecimal("100.00"));
        rule.setWarnMinValue(new BigDecimal("80.00"));
        // 设置报警等级
        rule.setWarnLevel(6);
        // 设置电池类型
        rule.setBatteryType("超威电池");
        allRules.add(rule);

        System.out.println(allRules.size());

        // 2.模拟电池信号数据
        BatterySignalDTO signalDTO = new BatterySignalDTO();
        signalDTO.setCarId(1L);
        signalDTO.setBatteryType("超威电池");
        signalDTO.setBatterySignal("{\"Mx\":200.0,\"Mi\":110,\"Ix\":12.0,\"Ii\":11.7}");
        List<BatterySignalDTO> signalDTOS = new ArrayList<>();
        signalDTOS.add(signalDTO);

        List<WarnResult> warnResultList = batteryWarnDomainService.parseSignals(signalDTOS, allRules);

        System.out.println(warnResultList.get(0));
    }
```

预警模块解析电池信号，消费者端

阻塞队列 + 批量处理

```java
@Slf4j
@Component
@RocketMQMessageListener(consumerGroup = "MyConsumerGroup", topic = "TestTopic",consumeMode= ConsumeMode.CONCURRENTLY,messageModel= MessageModel.BROADCASTING)
public class BatteryMsgConsumer implements RocketMQListener<String> {

    // 阻塞队列存储dto
    private final BlockingQueue<BatterySignalDTO> dtoQueue = new LinkedBlockingQueue<>();

    // 批量处理数量
    private static final int BATCH_SIZE = 2;

    // 预警规则列表
    List<BatteryWarnRule> allRules;

    ......

    /**
     * 消费者实例化后，启动后台线程，处理队列中的dto
     */
    public BatteryMsgConsumer() {  new Thread(this::batchProcessLoop).start();}

    /**
     * 接收消息
     */
    @Override
    public void onMessage(String message) {
        try {
            BatterySignalDTO dto = JSON.parseObject(message, BatterySignalDTO.class);
            // 将消息放入队列中，后台异步线程批量处理,避免阻塞消费线程
            dtoQueue.put(dto);
        } catch (Exception e) {
            log.error("消息消费失败: {}", message, e);
        }
    }

    /**
     * 后台线程循环批量处理队列中的dto,
     */
    private void batchProcessLoop() {
        while (true) {
            try {

```

```
                List<BatterySignalDTO> dtoList = new ArrayList<>(BATCH_SIZE);
                // 从队列中取出BATCH_SIZE个dto
                for (int i = 0; i < BATCH_SIZE; i++) {
                    BatterySignalDTO dto = dtoQueue.take();

                    // 根据carId获取电池类型
                    String batteryType = getBatteryTypeByCarId(dto.getCarId());

                    dto.setBatteryType(batteryType);
                    dtoList.add(dto);
                }
                batchConsumerBatteryMessage(dtoList);
            } catch (InterruptedException e) {
                log.error("批量处理线程被中断", e);
                // 恢复中断状态
                Thread.currentThread().interrupt();
            } catch (Exception e) {
                log.error("批量处理失败", e);
            }
        }
    }
    ......
}
```

电池信息相关操作

redis缓存 + 缓存时间设置为区间范围内随机

```java
    /**
     * 分页查询电池信息，并缓存
     */
    @Override
    public BatteryMsgDTO queryBatteryMsgListByCarId(QueryBatteryMsgDTO queryBatteryMsgDTO) {
        // 1.查缓存
        Long carId = queryBatteryMsgDTO.getCarId();
        Integer pageNum = queryBatteryMsgDTO.getPageNum();
        Integer pageSize = queryBatteryMsgDTO.getPageSize();
        // 1.1 获取缓存key，根据carId、pageNum、pageSize生成
        String batteryMsgPageKey = RedisKeyUtil.getBatteryMsgPageKey(carId, pageNum, pageSize);
        log.info("查询缓存{}", batteryMsgPageKey);
        String batteryMsgPageCache = redisTemplateUtil.get(batteryMsgPageKey);
        if (Objects.nonNull(batteryMsgPageCache)) {
            log.info("缓存命中， carId{}， pageNum{}， pageSize{}", carId, pageNum, pageSize);

            return JSON.parseObject(batteryMsgPageCache, BatteryMsgDTO.class);
        }

        // 2.缓存没有从数据库查询
        String lockKey = RedisKeyUtil.getBatteryMsgLockKey(carId);
        String lockValue = UUID.randomUUID().toString();
        try {
            // 尝试获取锁，失败则等待一秒，尝试三次
            boolean isLock = redisTemplateUtil.tryLock(lockKey, lockValue, 2, TimeUnit.SECONDS, 3, 1);
            if (isLock) {
                BatteryMsgDTO batteryMsgDTOList = batteryMsgRepository.queryBatteryMsgListByCarId(queryBatteryMsgDTO);
                log.info("从数据库查询， carId{}， pageNum{}， pageSize{}", carId, pageNum, pageSize);

                // 序列化后存入Redis
                String cacheValue = JSON.toJSONString(batteryMsgDTOList);
                // 缓存时间设置为区间随机值
                int randomExpiry = ThreadLocalRandom.current().nextInt(30, 46);
                redisTemplateUtil.set(batteryMsgPageKey, cacheValue, randomExpiry, TimeUnit.SECONDS);
                return batteryMsgDTOList;
            }
        } catch (Exception e) {
```

```
38              log.error("获取锁失败， carId{}， pageNum{}， pageSize{}", carId
    , pageNum, pageSize);
39          } finally {
40
41              // 释放锁时校验锁值
42              if (lockValue.equals(redisTemplateUtil.get(lockKey))) {
43                  redisTemplateUtil.delete(lockKey);
44              }
45              log.info("释放锁成功");
46          }
47          return new BatteryMsgDTO();
48      }
```