# BMS系统设计文档

## 1. 系统模块设计

### 1. 电池信号上报

（1）支持通过接口 POST 方式上报信号数据

（2）支持按规则结合监控信号差值，输出预警信息

#### 1. 预警规则管理

（1）配置不同电池类型下的预警规则，包括规则编号、名称、详细规则、预警级别

### 3. 预警生成与查询

（1）定时扫描信号表，根据规则计算预警级别，并通过MQ发送和消费

（2）通过接口查询指定车辆预警记录

### 4. 缓存处理

（1）Redis：缓存车辆电池信号、预警信息

（2）Caffeine：本地内存缓存消耗较少的信息，如最新信号值

### 5. 缓存和数据库数据一致性

（1）修改数据时添加Redisson分布式锁

（2）短过期时间：设置缓存过期时间为10秒

（3）延迟双删：修改后删除缓存，延迟5秒再删

**使用技术：** `springboot + Mysql + redis + caffeine + mybatis + redisson + rocketmq`

**启动项目**

- 启动 `redis` 服务
- 启动 `rockmq` 服务

```
cd E:\rocketmq-all-5.3.3-bin-release
.\bin\mqnamesrv

```

```
$env:NAMESRV_ADDR='localhost:9876'
cd E:\rocketmq-all-5.3.3-bin-release
.\bin\mqbroker -n localhost:9876
```

# 二、数据库表设计

## 1. vehicleinfo (车辆信息表)

随机生成1000条数据传入数据库

| 列名 | 类型 | 说明 |
| --- | --- | --- |
| vid | varchar(16) | 唯一车辆编码，主键 |
| carId | int | 车架编号 |
| batteryType | varchar(10) | 电池类型 (三元电池/铁锭) |
| totalMileage | int | 总里程 |
| batteryHealthStatus | int | 电池健康状态 |

## 2. rule (预警规则表)

插入指定的四条规则

| 列名 | 类型 | 说明 |
| --- | --- | --- |
| ruleId | int | 序号，主键 |
| ruleNo | int | 规则编号 |
| name | varchar(20) | 规则名称 |
| batteryType | varchar(10) | 电池类型 |
| warningRule | varchar(255) | 预警级别规则描述 |

## 3. signalreport (电池信号报表)

用于存放汽车电池的各项参数

carId被用于检索，为其添加索引 `INDEX idx_carId (carId)`

| 列名 | 类型 | 说明 |
| --- | --- | --- |
| reportId | int | 报表ID，主键 |
| carId | int | 车架编号 |
| warnId | int | 规则编号 |
| signalString | varchar(255) | 信号 |
| batteryType | varchar(20) | 电池类型 |
| warnName | varchar(20) | 预警名称 |
| warnLevel | int | 预警级别 |

## 三、接口设计

### 1. 电池信号上报

- **URL**: `/api/warn`
- **Method**: `POST`
- **Body** (JSON 数组):

```
[
  {
    "carId": 1,
    "warnId": 1,
    "signal": "{\"Mx\":12.0,\"Mi\":0.6}"
  },
  {
    "carId": 2,
    "warnId": 2,
    "signal": "{\"Ix\":12.0,\"Ii\":11.7}"
  },
   {
    "carId": 3,
    "signal": "{\"Mx\":11.0,\"Mi\":9.6,\"Ix\":12.0,\"Ii\":11.7}"
  }
]
```

**Response**:

```
{
```

```
    "status": 200,
    "msg": "ok",
    "data":
        [
            {
                "车架编号": 1,
                "电池类型": "三元电池",
                "warnName": "电压差报警",
                "warnLevel": 0
            },
            {
                "车架编号": 2,
                "电池类型": "铁锂电池",
                "warnName": "电流差报警",
                "warnLevel": 2
            },
            {
                "车架编号": 3,
                "电池类型": "三元电池",
                "warnName": "电压差报警",
                "warnLevel": 2
            },
            {
                "车架编号": 3,
                "电池类型": "三元电池",
                "warnName": "电流差报警",
                "warnLevel": 2
            }
        ]
}
```


## 2. 电池信号状态查询

- **URL**: `/api/batterySignalStatus/{carId}`
- **Method**: `GET`
- **Response**:


```
{
  "status": 200,
  "msg": "ok",
  "data": "{\"Mx\":12.0,\"Mi\":0.6}"
}
```

### 3. 预警报告查询

- **URL**: `/api/signalReport/{carId}`
- **Method**: `GET`
- **Response**:

```
{
  "status": 200,
  "msg": "ok",
  "data": [
    {
        "车架编号": 1,
        "电池类型": "三元电池",
        "warnName": "电压差报警",
        "warnLevel": 0
     }
  ]
}
```

### 4. 预警报告修改

- **URL**: `/api/signalReport`
- **Method**: `PUT`
- **Body** (JSON):

```
{
    "carId": 2,
    "warnId": 2,
    "signal": "{\"Ix\":12.0,\"Ii\":10}"
}
```

- **Response**:

```
{
  "status": 200,
  "msg": "ok",
  "data": "Signal report updated successfully"
}
```

### 5. 预警报告删除

- **URL**: `/api/signalReport/{carId}`
- **Method**: `DELETE`
- **Response:**

```
{
  "status": 200,
  "msg": "ok",
  "data": "Signal report deleted successfully"
}
```

### 6. 获取所有预警报告

- **URL**: `/api/signalReports`
- **Method**: `GET`
- **Response:**

```
{
  "status": 200,
  "msg": "ok",
  "data": [
    {
        "车架编号": 1,
        "电池类型": "三元电池",
        "warnName": "电压差报警",
        "warnLevel": 0
     }
  ]
}
```

## 四、预警计算逻辑简要

1. 解析信号数据：通过请求数据中的signal字段获取信号，提取Mx（最高电压）,Mi（最小电压）、Ix（最高电流）,Ii（最小电流）。
2. 根据车架编号查询车辆信息表获取电池类型
3. 根据warnId和电池类型查询预警规则，如果没有warnId，则查询所有规则
4. 根据查询到的规则计算预警等级
5. 创建SignalReport对象并设置属性

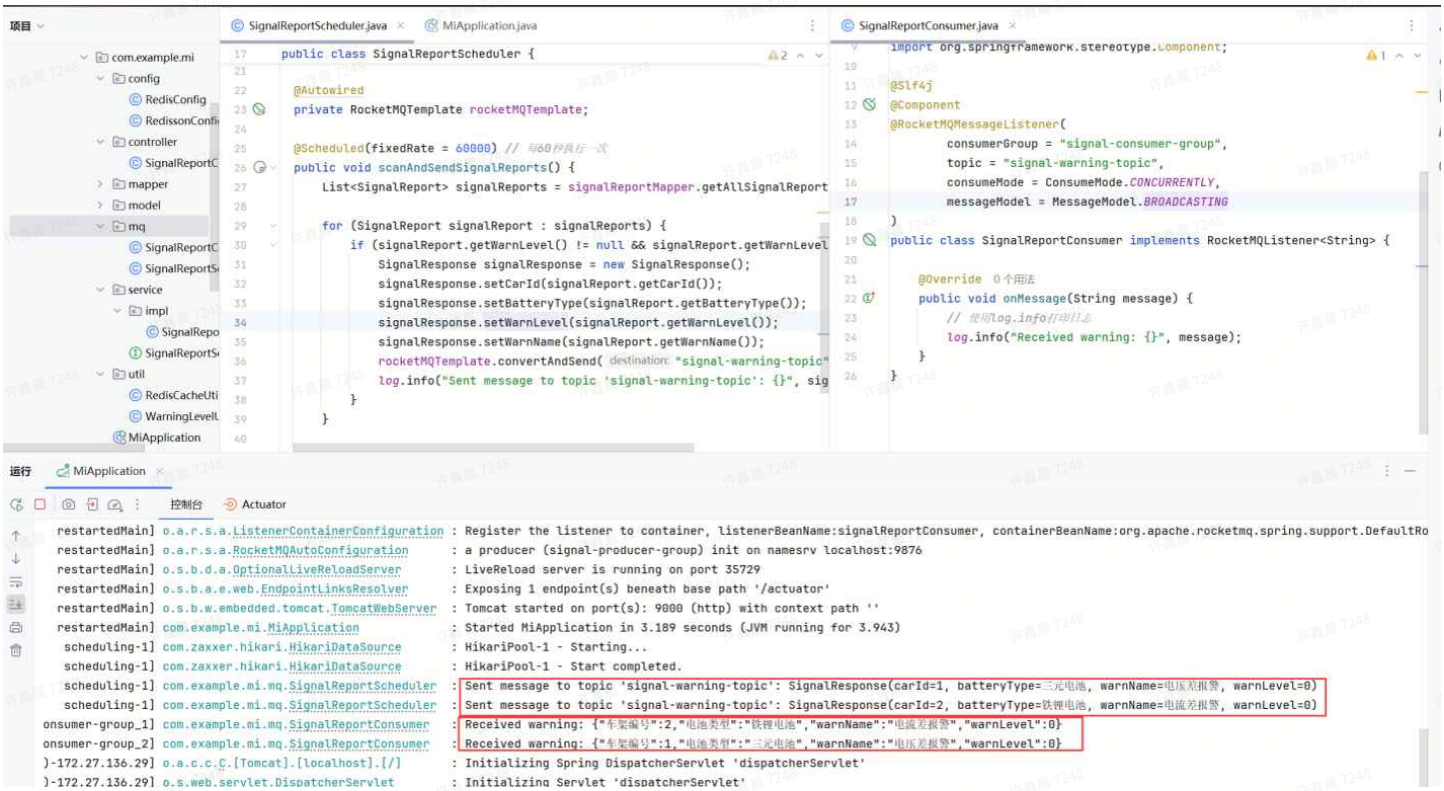
6. 将SignalReport对象插入数据库
7. 筛选出需要报警的信号，封装到返回数据SignalResponse中

## 五、缓存和数据库数据一致性

1. 获取电池信号状态时，先查本地缓存（Caffeine），再查Redis，最后查数据库
2. 上报信号时，涉及数据库写操作，填加事务管理
3. 获取车辆的预警报告时，使用Redis缓存热点数据
4. 更新指定车辆的预警报告时，使用分布式锁控制，并采用延迟双删缓存
5. 缓存设置10秒的短过期时间

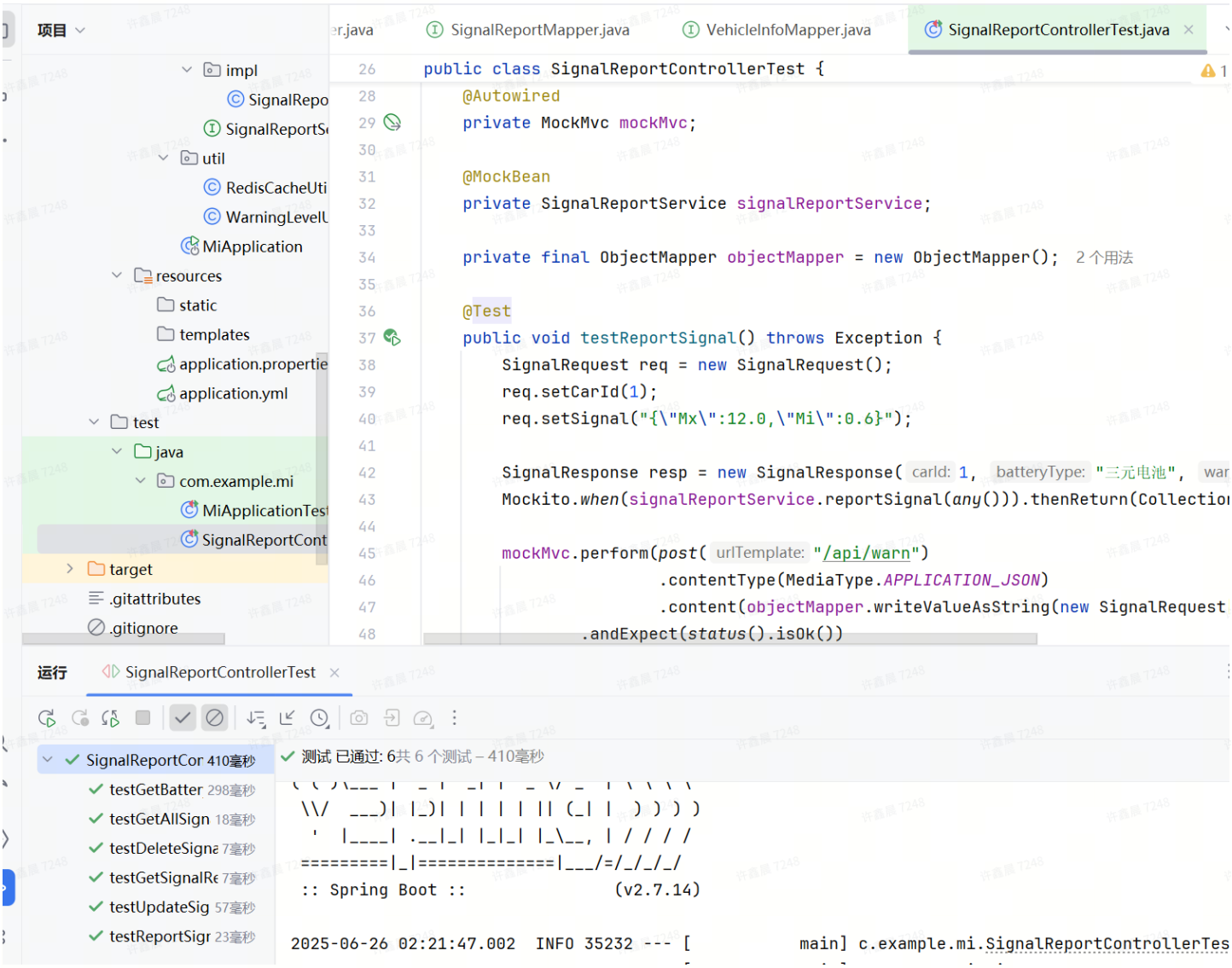
## 六、定时任务 & MQ

1. 每60分钟扫描 `signalReport` 表中数据
2. 根据预警等级字段 `warnLevel` （非null，非-1，-1表示不预警），筛选出预警信号
3. 生产者使用 `rocketMQTemplate` 发送 `signalResponse` 对象消息
4. 消费者监听队列，使用 `log.info()` 打印消息
5. 配置 `logback-spring.xml` 文件，将消息写入日志中




## 七、单元测试

编写六个接口对应的单元测试方法，单元测试全都可以正常通过

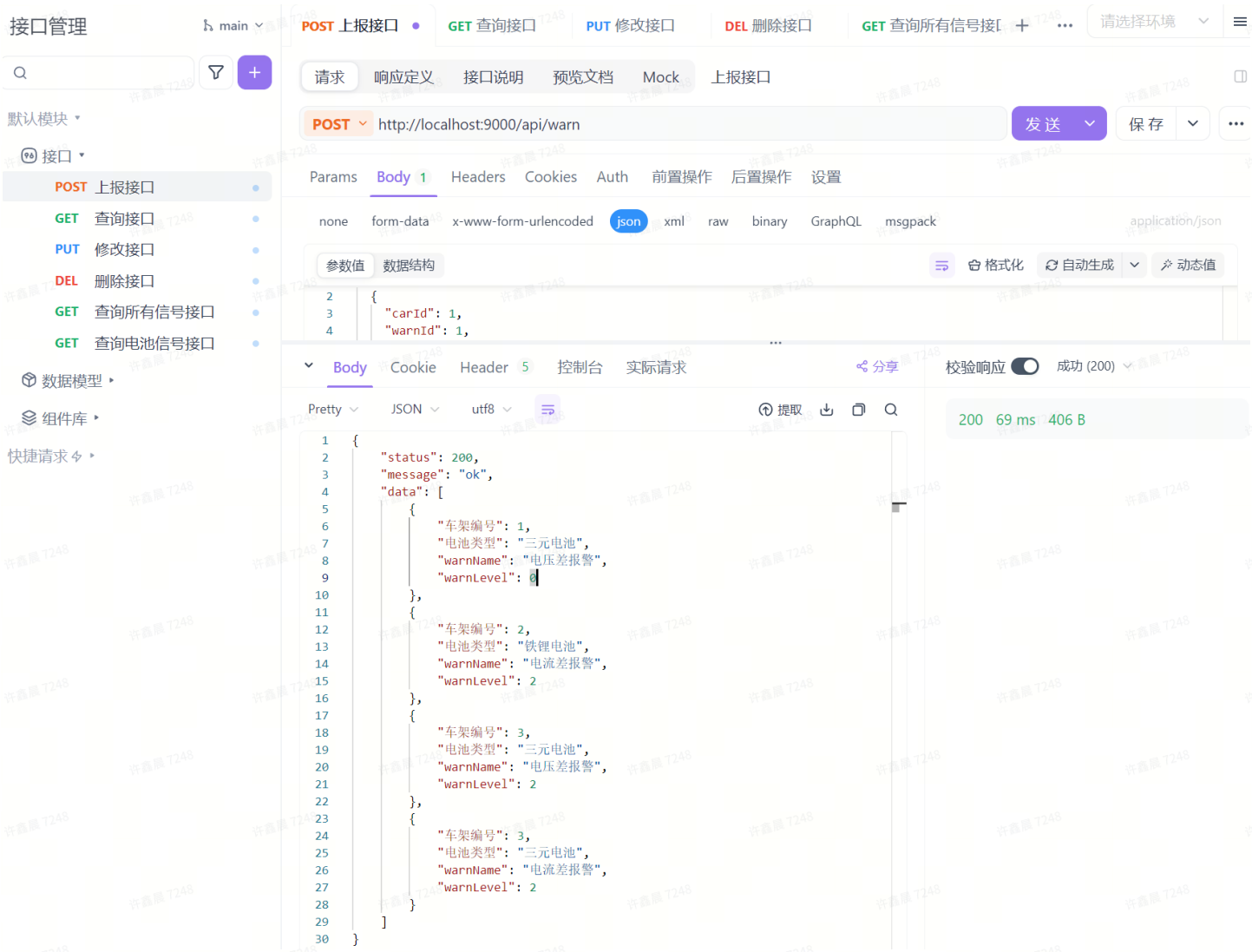



# 八、接口测试

使用apifox进行接口测试

## 1. 电池信号上报接口

- **URL**: `/api/warn`
- 返回对应电池的预警信息




## 2. 电池信号状态查询

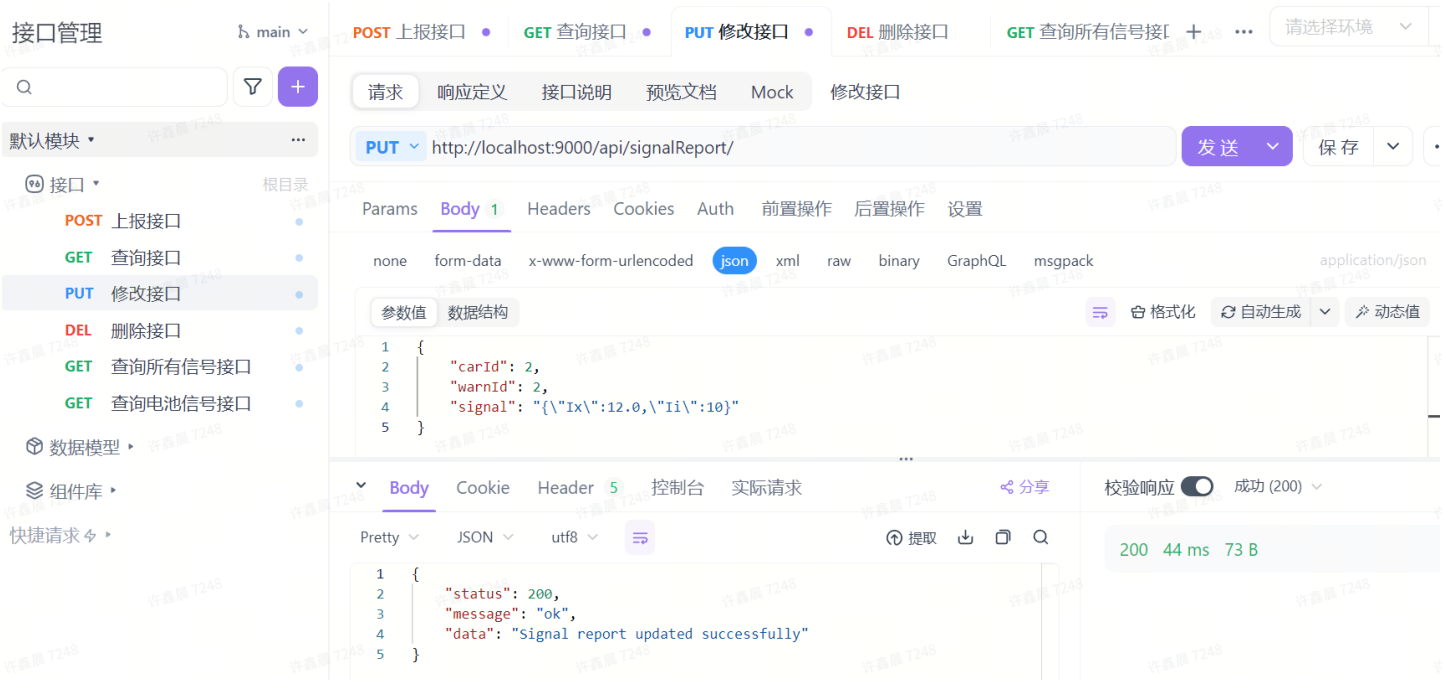
- **URL**: `/api/batterySignalStatus/{carId}`
- 返回对应carId的电池信号

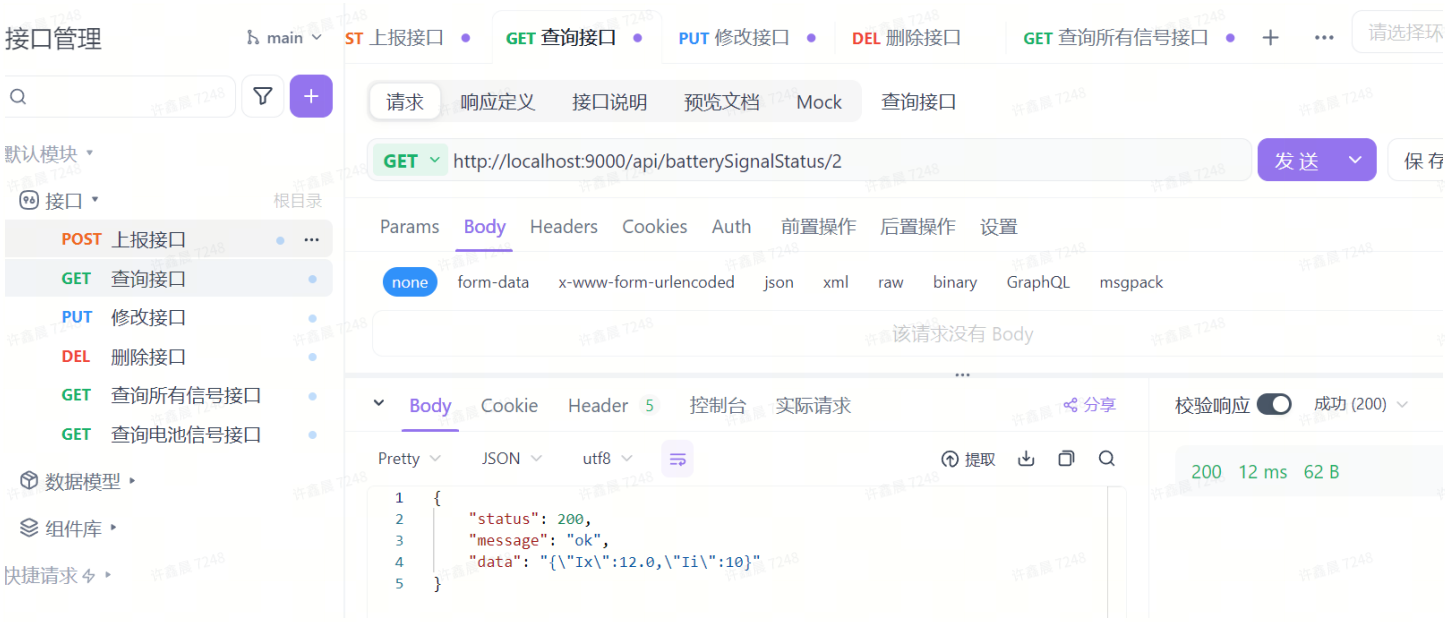

## 3. 预警报告查询

- URL: `/api/signalReport/{carId}`
- 查询carId为2的数据


## 4. 预警报告修改

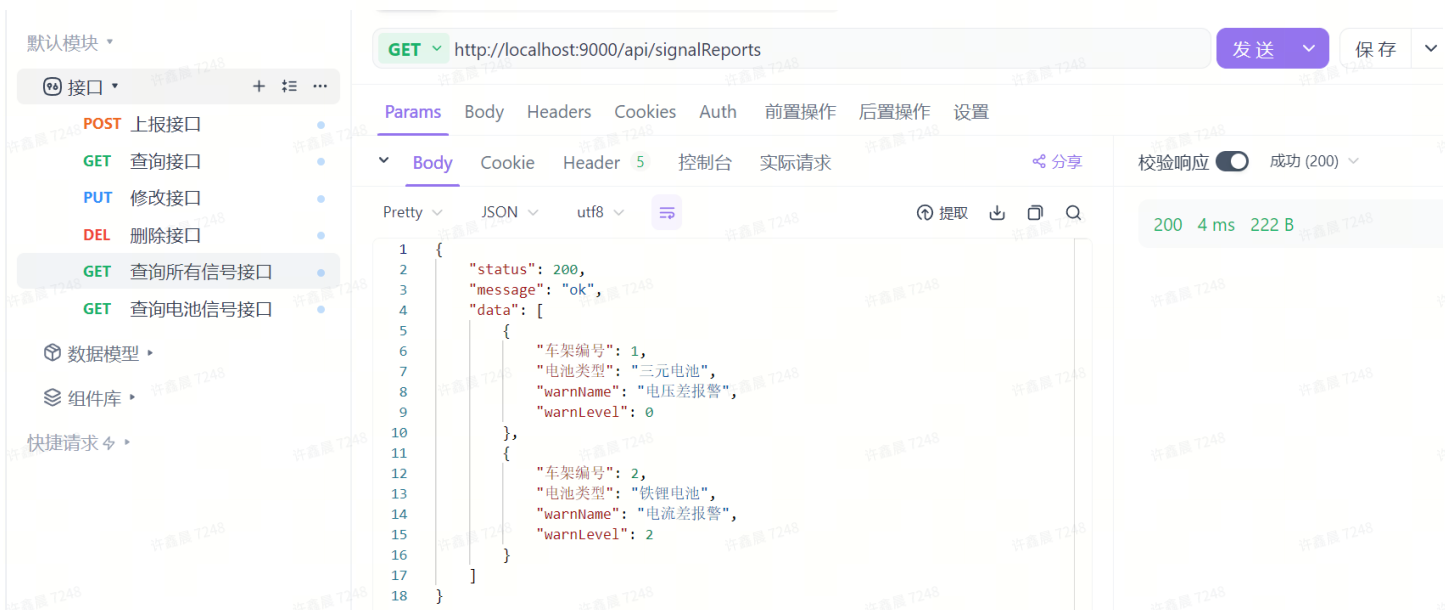
- URL: `/api/signalReport`
- 修改 `carId` 为2的数据， `signal` 修改为 `"{\"Ix\":12.0,\"Ii\":10}"` ，查询后修改成功
- 修改后同时会调用规则计算，查询计算预警等级 `warnLevel` 存入数据库





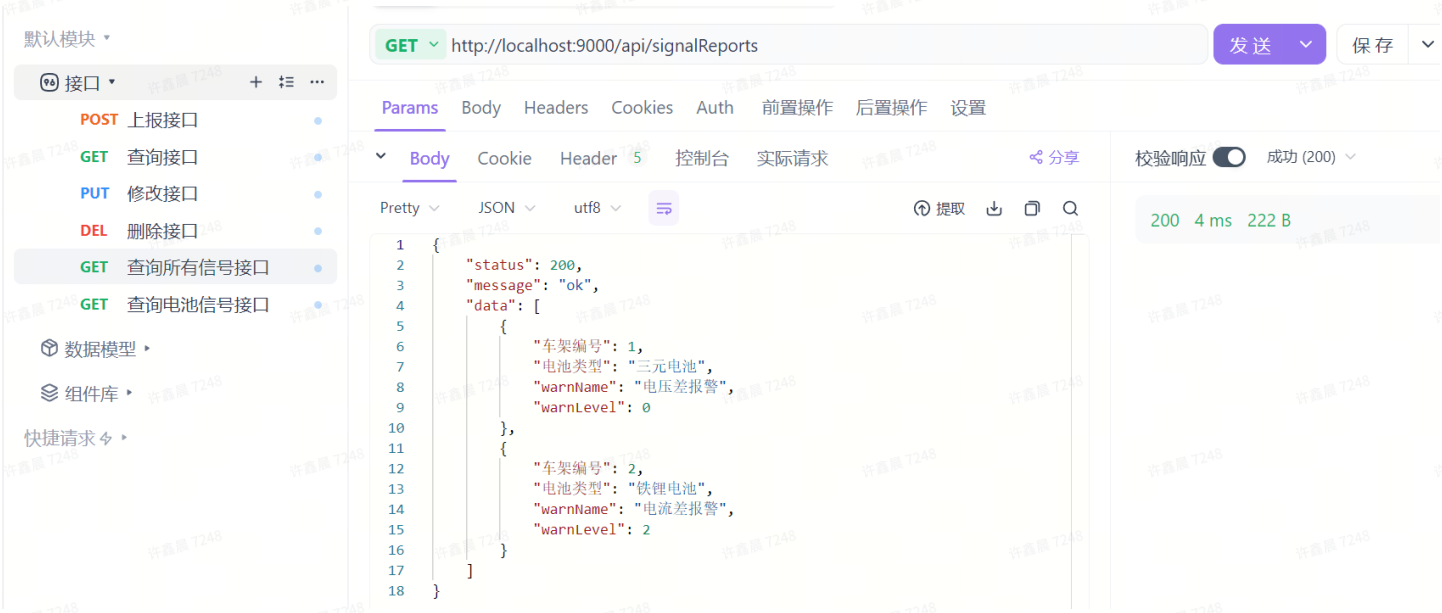

## 5. 预警报告删除

- **URL**: `/api/signalReport/{carId}`
- 删除carId为3的数据，carId为3的两条数据删除成功

### 6. 获取所有预警报告

- **URL**: `/api/signalReports`




## 九、加分技术点

### 1. 规则解析不是写成固定在代码里面，而是根据规则编号获取预警规则然后解析

编写工具类`WarningLevelUtil`，方法为`calculateWarningLevel(String signal, String ruleString)`

- **传入参数：**

signal：JSON字符串，如 {"Mx": 3.5, "Mi": 2.1, "Ix": 4.2, "Ii": 1.2}

ruleString：规则字符串

- **返回参数：**

Integer：报警等级，-1 表示不报警

- **步骤**

1. **解析 JSON 数据**

使用 Jackson 的`ObjectMapper`解析`signal`字符串。

提取差值指标：

`voltageDiff = Mx - Mi`

`currentDiff = Ix - Ii`

### 2. 预处理规则字符串

替换可能出现的中文符号为英文符号（如中文括号、冒号、比较符）。

使用 `;` 拆分每一条独立的规则。

`ruleString.replace("（", "(").replace("）", ")")`
`ruleString.split(";")`

### 3. 遍历每一条规则

每条规则格式如：`0.2<=(Mx-Mi)<0.6，报警等级：4`

对每条规则：

- 拆分为条件和报警等级描述：`"0.2<=(Mx-Mi)<0.6"` 与 `"报警等级：4"`
- 缓存规则到 `ConcurrentHashMap` 中，重复规则直接读取
- 调用表达式匹配函数 `evaluateCondition()` 判断当前差值是否满足该条件。
- 若满足，则提取报警等级并返回。
- 若包含“不报警”，直接返回 -1。

### 4. 表达式匹配与判断 evaluateCondition()

- 替换表达式中的 `(Mx-Mi)` 和 `(Ix-Ii)` 为 `x`，统一处理。
- 根据 `x` 代表的变量选择差值值（如 `voltageDiff` 或 `currentDiff`）。
- 使用正则表达式支持格式：
  - 单条件：`x<0.2`、`x>=1`
  - 双条件：`0.2<=x<0.6`
- 对每个条件（最多两个）：
  - 将字符串拆分为 [左值, 比较符, 右值]
  - 调用 `compare()` 方法进行浮点数比较判断。
  - 所有条件均满足则返回 `true`

### 5. 比较运算函数 compare()和符号方向转换 flipOperator()

- compare()实现通用的数值比较函数
- 为了解析 `0.2 <= x`，使用flipOperator()转换为 `x >= 0.2`

6. **最终结果返回**

- 遍历所有规则，如果有命中报警条件，则返回对应报警等级。
- 如果没有任何规则匹配，则默认返回 `-1` 表示不报警

## 2. 实时规则的接口性能测试和优化

Number of Threads　100（模拟 100 并发）

Ramp-Up Period　10（10 秒内启动完）

Loop Count　100（每线程请求 100 次）

**测试数据**：生成1000条csv数据导入jmeter

| 参数项 | 设置值 |
| --- | --- |
| Name | POST /api/warn |
| Protocol | http |
| Server Name | localhost |
| Port Number | 9000 |
| Method | POST |
| Path | /api/warn |
| Body Data | JSON |
| Content-Type Header | Content-Type: application/json |

**监听器**

Summary Report

Aggregate Report

View Results Tree

**优化方案**

1. 使用线程池，涉及到大量io操作，使用线程池异步写入
2. 规则解析根据类 `WarningLevelUtil` 添加缓存逻辑，避免每次重复解析字符串的耗时
3. 设计到数据库的写入，使用事务 `@Transactional` 保证正常写入

**性能指标**

| 指标 | 说明 | 值 |
| --- | --- | --- |
| **P99%** | 第 99 百分位响应时间（目标 < 1000ms） | 4ms |
| **Average** | 平均响应时间 | 1ms |
| **Throughput** | 每秒请求处理数 | 993.8 |
| **Error%** | 错误率，理想为 0% | 0 |

| Label | # 样本 | 平均值 | 中位数 | 90% 百分位 | 95% 百分位 | 99% 百分位 | 最小值 | 最大值 | 异常 % | 吞吐量 | 接收 KB/sec | 发送 KB/sec |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| HTTP请求 | 10000 | 1 | 2 | 2 | 3 | 4 | 0 | 128 | 0.00% | 993.8/sec | 242.44 | 267.10 |
| 总体 | 10000 | 1 | 2 | 2 | 3 | 4 | 0 | 128 | 0.00% | 993.8/sec | 242.44 | 267.10 |

| Label | # 样本 | 平均值 | 最小值 | 最大值 | 标准偏差 | 异常 % | 吞吐量 | 接收 KB/sec | 发送 KB/sec | 平均字节数 |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| HTTP请求 | 10000 | 1 | 0 | 128 | 1.48 | 0.00% | 993.8/sec | 242.44 | 267.10 | 249.8 |
| 总体 | 10000 | 1 | 0 | 128 | 1.48 | 0.00% | 993.8/sec | 242.44 | 267.10 | 249.8 |

## 3. 数据存储和查询方案

### 1. 数据写入优化

- 使用 **异步消息队列**（如 Kafka、RocketMQ）接收高并发信号数据，解耦数据产生与存储压力。
- 使用 **批量写入机制**，每 N 条数据批量入库（比如按 1 秒或 1000 条一次）。
- 对 JSON 信号做 **结构压缩** 或字段规整，降低数据体积。

### 2. 数据存储优化（冷热分层 + 查询友好）

- **实时存储**：高写入性能数据库，如 **ClickHouse**、**InfluxDB**
- **长期归档**：Hadoop HDFS、Object Storage（如 MinIO）、甚至定期压缩后写入冷存储表
- **JSON结构处理**：写入前结构化或扁平化处理；如“信号”为 4 个字段就拆成 Mx、Mi、Ix、Ii 四列写入表中

### 3. 查询性能优化策略

- **索引设计**：对查询中经常用的字段加复合索引，例如 `(car_id, timestamp)`
- **分库分表**（海量数据）：按carId 分库：carId % N 路由
- **缓存**：常用查询结果缓存到 Redis，配置布隆过滤器避免无效 carId 查询穿透

- **分布式：**采用微服务架构拆分服务，通过Dubbo或Feign调用服务
- **服务集群**：多实例部署，通过负载均衡找到对应服务
- **CDN**：将静态资源分发到多个不同的地方以实现就近访问，进而加快静态资源的访问速度
- **限流和熔断机制：**确保核心服务可用
- **超时和重试机制设置：**提高服务容错能力