Apache HertzBeat 日志监控能力实现方案第三版-陈阳

> 姓名：陈阳 (1597081640@qq.com)
> Github：bigcyy
> 项目：Apache HertzBeat 日志监控能力

一、项目介绍

Apache HertzBeat 是一个开源的实时监控系统，专注于提供高效、灵活的监控解决方案。它支持多种数据源的监控，广泛应用于云原生环境下的运维场景。然而，随着用户对日志监控需求的增加，本项目需要开发一个日志监控模块，以增强 HertzBeat 在日志监控领域的功能覆盖。 本项目的核心目标是构建一个轻量级的日志监控模块，为用户提供从日志采集、存储到查询展示的完整解决方案。具体而言，项目将实现以下关键能力：

- 日志采集与格式标准化：通过 OpenTelemetry SDK，Kafka 消息收集，FileBeat 上报等方式接收日志数据，并将其转换为统一的格式，便于后续处理。
- 实时日志查看：在 HertzBeat 支持实时的日志查看，并且支持过滤。
- 高效的日志存储：利用 GreptimeDB 的高性能时序数据库特性，设计合理的表结构，确保日志数据的高效存储与检索。
- 友好的查询与展示界面：开发直观的日志查询 API 和可视化页面，帮助用户快速定位问题并分析日志内容。
- 日志阈值告警规则：支持对日志配置阈值告警规则，支持 SQL 表达式。
- 生成 Vector 配置文件：方便用户将日志采集到 GreptimeDB。

二、个人介绍与优势

我来自重庆邮电大学，目前研一，擅长 Java 开发，同时在 TypeScript 方面也积累了丰富的经验。我还具备一定的算法经验（2022 ACM-ICPC 省铜）、大型项目开发与优化经验（2024 OceanBase 数据库大赛 24/1212）。

除此之外，我对开源充满热情，并渴望深入参与其中。此前我成功向 Langchain4j 贡献了一个已合并的 PR，并且还自己开源了 customized-chat（一个 LLM + RAG 应用）获得了120+ star。这些开源经历不仅让我收获了宝贵的实践经验，也进一步激发了我对开源社区的热爱和投入。

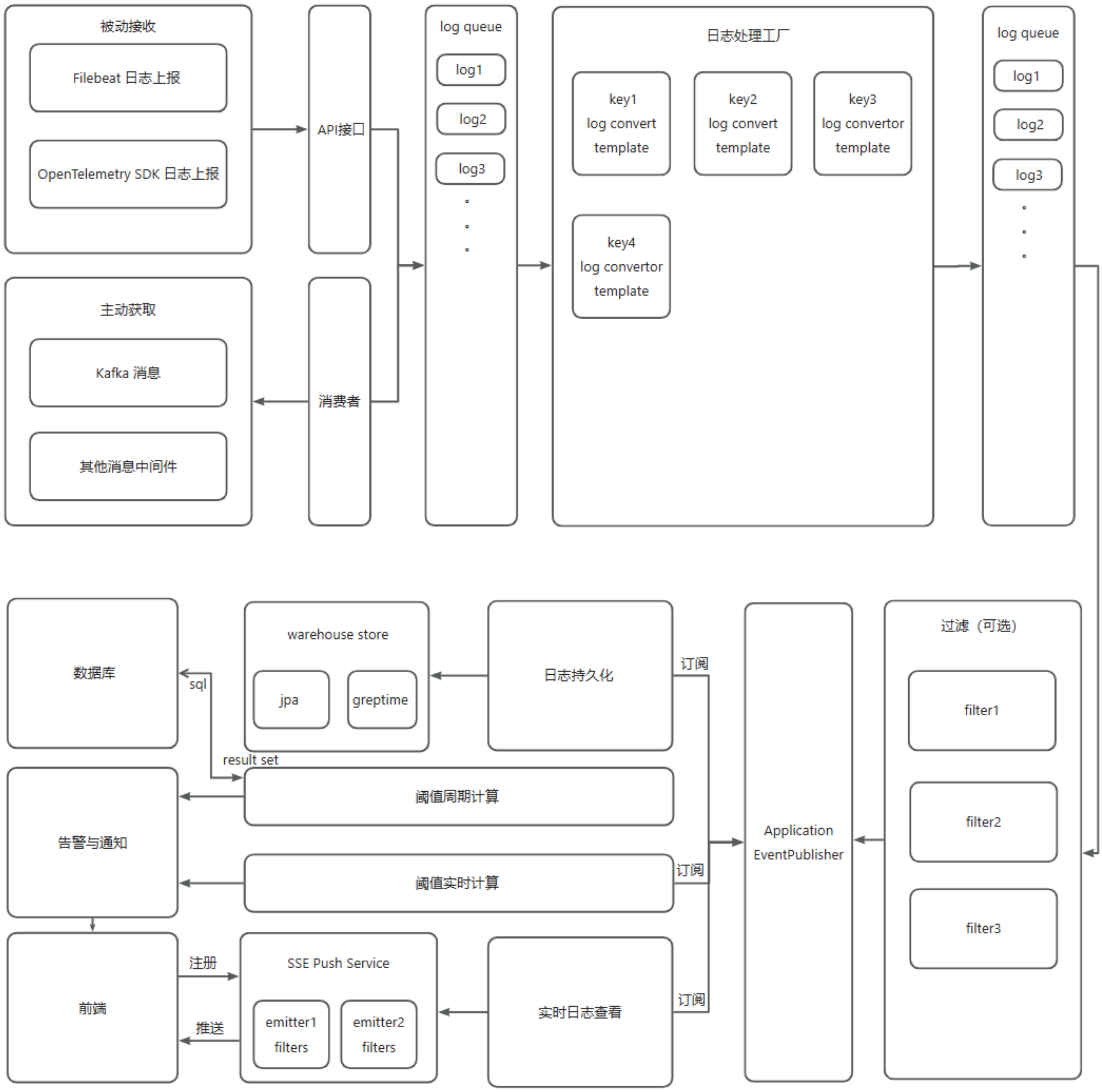
在此前，我已经深入阅读 HertzBeat 源码，并积极参与社区，这是我对 HertzBeat 社区做过的贡献。

三、实现方案

下面是我的实现方案，方案的架构部分参考了 HertzBeat 原有的设计，并且尽量复用原有的模块而不增加复杂性。

3.1 整体架构

日志的来源分为被动接收和主动获取两类，我们获取到日志数据后放入日志队列，日志处理工厂消费队列中的日志，将日志处理成 HertzBeat 统一的格式放入新的日志队列，后续可对日志进行过滤。经过过滤后的日志，经由事件发布者，将日志交给下游处理（持久化、实时阈值计算、实时日志推送）。除此之外，阈值周期计算器会周期性根据阈值规则判断是否触发告警，阈值规则触发后的告警和通知复用 HertzBeat 原有的逻辑。



3.2 日志接收

日志来源分为被动接收与主动获取两类。所有获取的日志，无论来源方式，均会标记数据源 Key 并送入日志队列，由下游统一处理。日志获取与转换的分离，其核心优势在于实现了日志接收与格式处理逻辑的解耦。

下游处理的对象格式为：

```
{
  "source_key": "xxx",
  "content": "xxx"
}
```

设置数据源 Key (source_key) 是为了让下游系统能根据该 Key 动态应用不同的日志转换模板。

例如，我们可以为 Kafka topic1 的日志设置 `source_key` 为 `kafka-topic1`，并关联一个日志转换模板；对 Kafka topic2 的日志则设置 `source_key` 为 `kafka-topic2` 并关联另一个日志转换模板。

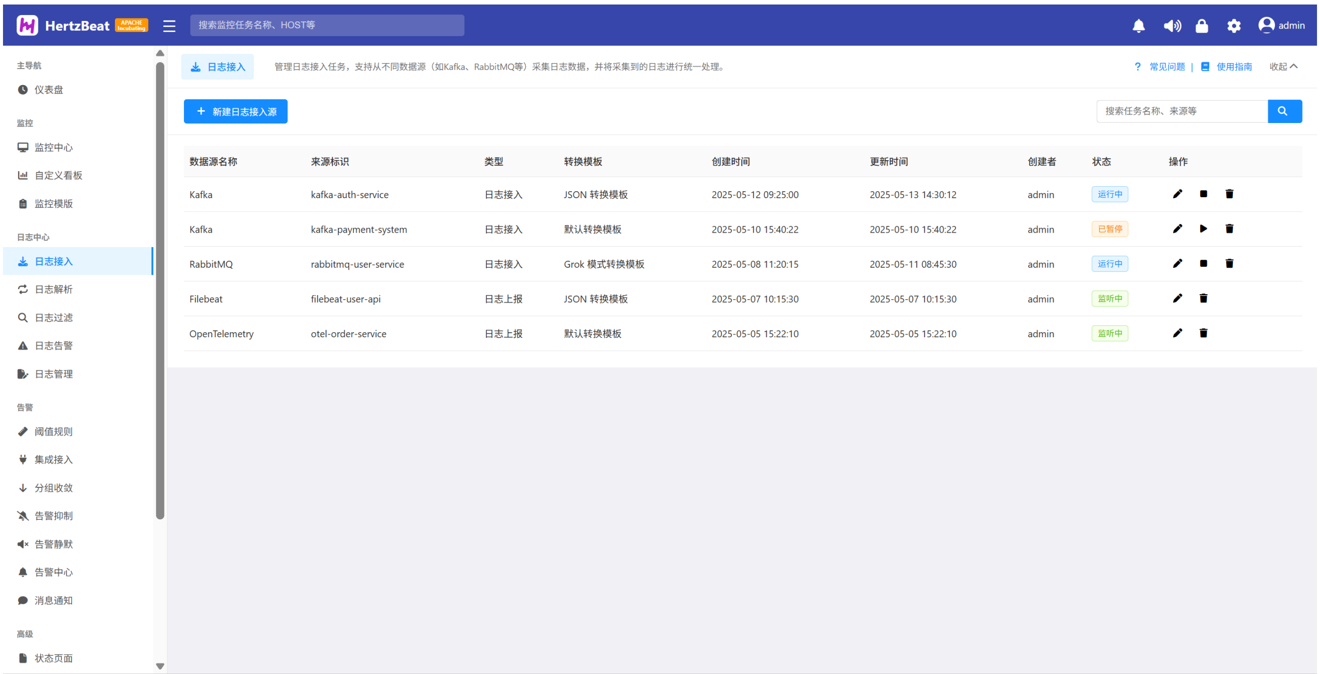
不仅如此，数据源 Key 未来还可充当消息队列的 Topic，将来自同一数据源的日志定向发送到特定队列中，下游处理时按 Topic 进行消费，从而有效扩展系统的吞吐能力。

被动接收需要配置：数据源名称和数据源的标识以及日志转换模板。
主动获取需要配置：数据源名称、数据源连接信息、数据源标识、日志转换模板。

因此需要建立一张日志数据源（`hzb_log_source`）表，表结构如下：

名称	类型	描述	约束
`id`	整数	主键，用于唯一标识数据源。	`PRIMARY KEY`
`source_name`	文本	数据源名称，方便用户识别。	`NOT NULL`
`source_key`	文本	日志来源标识，用于绑定日志转换器、告警。	`NOT NULL`，`UNIQUE`
`protocol`	JSON 结构	JSON 格式的文本，用于存储连接消息队列的配置信息（例如 Kafka、RabbitMQ 的连接参数）。	
`status`	数值	数据源获取的状态（运行中、监听中、暂停）	`NOT NULL`
`created_at`	日期时间	任务创建时间。(通常由应用程序在插入数据时记录，或数据库触发器设置。)	
`updated_at`	日期时间	任务更新时间。(通常由应用程序在更新数据时记录，或数据库触发器设置。)	
`creator`	文本	用于标识这一个数据源是由谁创建的。	`NOT NULL`

前端界面示例：



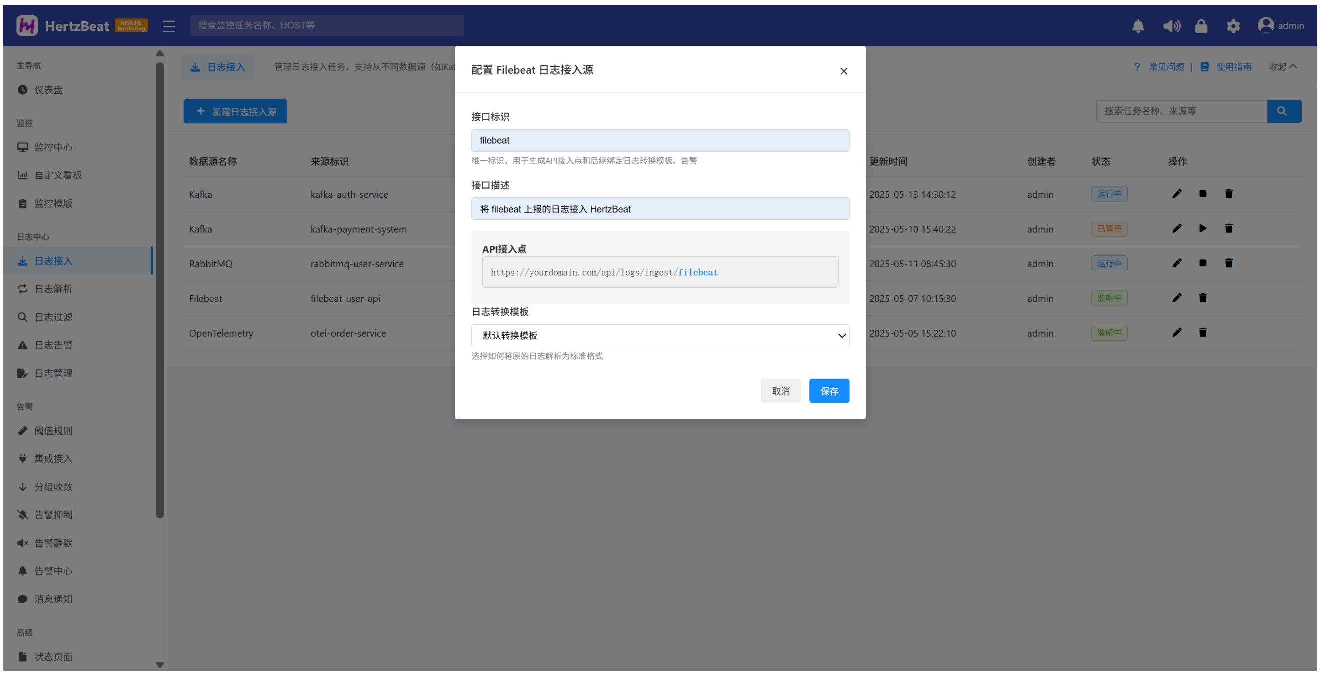
3.2.1 被动接收（日志上报）

- Filebeat 日志上报
- OpenTelemetry SDK 日志上报

对于被动接收类型的数据源，我们可以参考告警集成，提供接口被动接收。

```
// 代码为伪代码
@RestController
@RequestMapping("/api/logs")
public class LogIngestionController {
    @Autowired
    private CommonDataQueue rawLogQueue;

    @PostMapping("/ingest/{sourceKey}")
    public ResponseEntity<Message<Void>> ingestLog(
    @PathVariable("sourceKey") String sourceKey,
    @RequestBody String content) {
        // 这里还需要检查 sourceKey 是否在系统中配置过
        RawLog log = new RawLog(sourceKey, content);
        // 发送到队列中
        rawLogQueue.sendRawLog(log);
        return ResponseEntity.ok(Message.success("success"));
    }
}
```

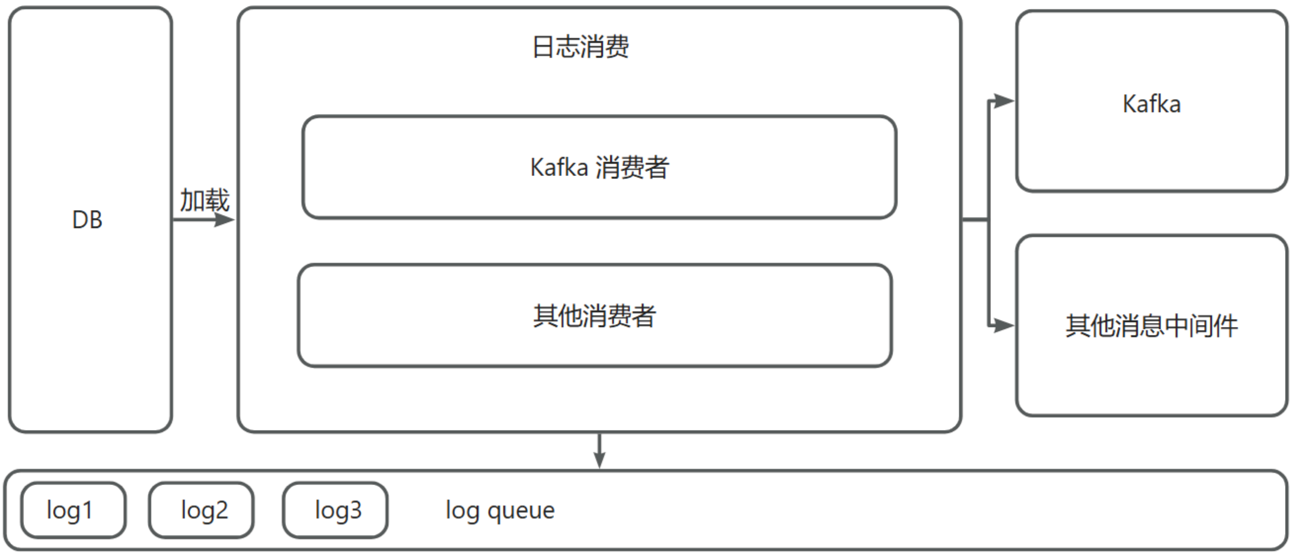
前端界面示例：



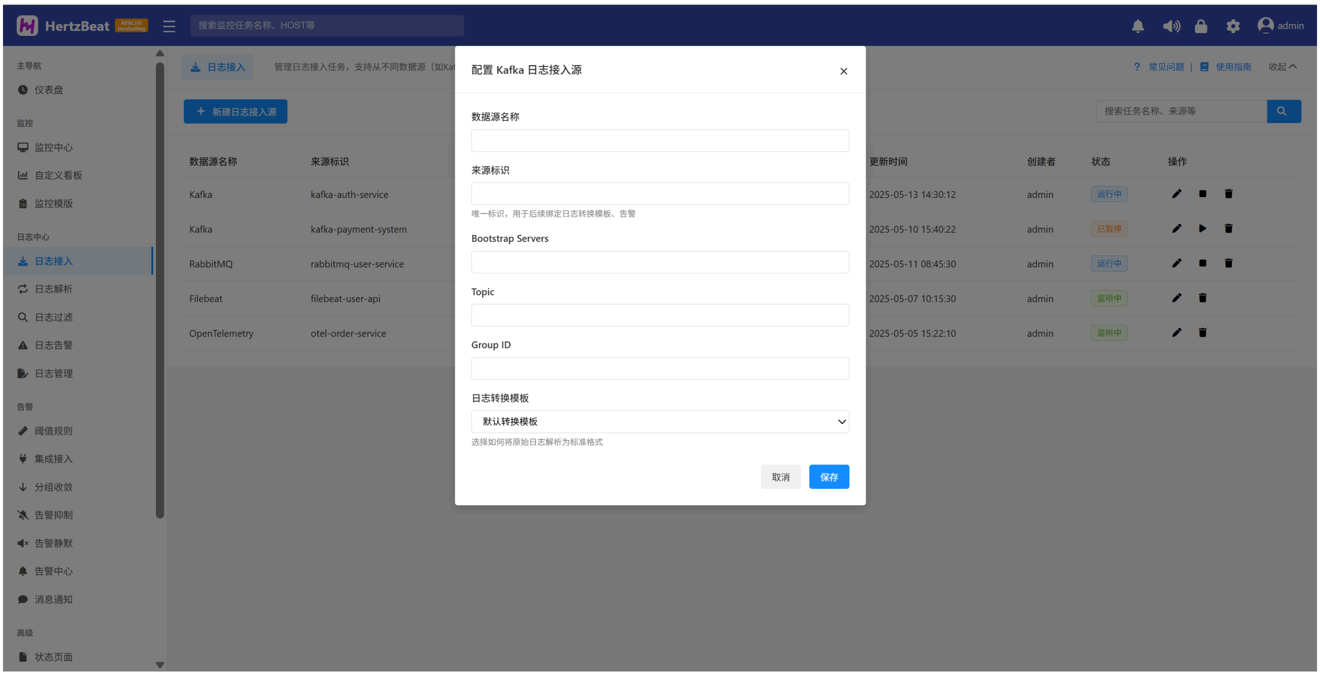
3.2.2 主动获取（从消息中间件获取）

- 消费 Kafka 数据
- 消费其他消息中间件的数据

日志主动获取的架构如下，在项目初始化时，从数据库加载数据源配置，根据数据源配置启动日志消费者主动获取数据，然后将获取到的日志放入日志队列。



前端界面示例：



3.2.3 配置文件生成

为了应对高并发或大数据量场景，日志可以不经过 HertzBeat 的接入层。用户只需要修改 HertzBeat 的配置，连接到 GreptimeDB，就可以使用 HertzBeat 的周期性阈值判断、日志告警通知等功能。

因此为了支持用户使用 Vector 直接将日志写入 GreptimeDB，我们需要提供配置文件生成功能，帮助用户快速配置 Vector，按照 HertzBeat 支持的日志格式输出到 GreptimeDB。

vector.yaml 示例:

```yaml
# Vector 主配置文件
# 设置全局选项
data_dir: "/var/lib/vector"

# 日志数据源配置
sources:
  app_logs:
    type: "file"
    include:
      - "${log_path}"
    ignore_older_secs: 86400  # 忽略1天以上的文件

# 日志格式转换和标准化
transforms:
  log_parser:
    type: "remap"
    inputs: ["app_logs"]
    source: |
      # 将数据源的日志解析为 JSON
      parsed_data = ${json_log_payload}
      . = {}
```

```
        .timestamp = ${timestamp_parse_script}
        .source_key = ${source_key_parse_script}
        .severity = ${severity_parse_script}
        .application = ${application_parse_script}
        .trace_id = ${trace_id_parse_script}
        .span_id = ${span_id_parse_script}
        .message = ${message_parse_script}

        # 其余的字段，用户自行添加
        .attributes = {}
        .attributes.${key} = ${value}

# 输出到GreptimeDB
sinks:
  greptime_sink:
    type: "greptimedb_metrics"
    inputs: ["log_parser"]
    endpoint: "${host}:${port}"  # GreptimeDB gRPC endpoint
    dbname: "public"  # 默认数据库名
    batch:
      max_events: 20  # 每批次最大事件数
      timeout_secs: 1  # 批次超时时间
    request:
      retry_attempts: 3  # 重试次数
    buffer:
      type: "memory"  # 使用内存缓冲
      max_events: 500  # 最大缓冲事件数
      when_full: "block"  # 缓冲区满时阻塞
```

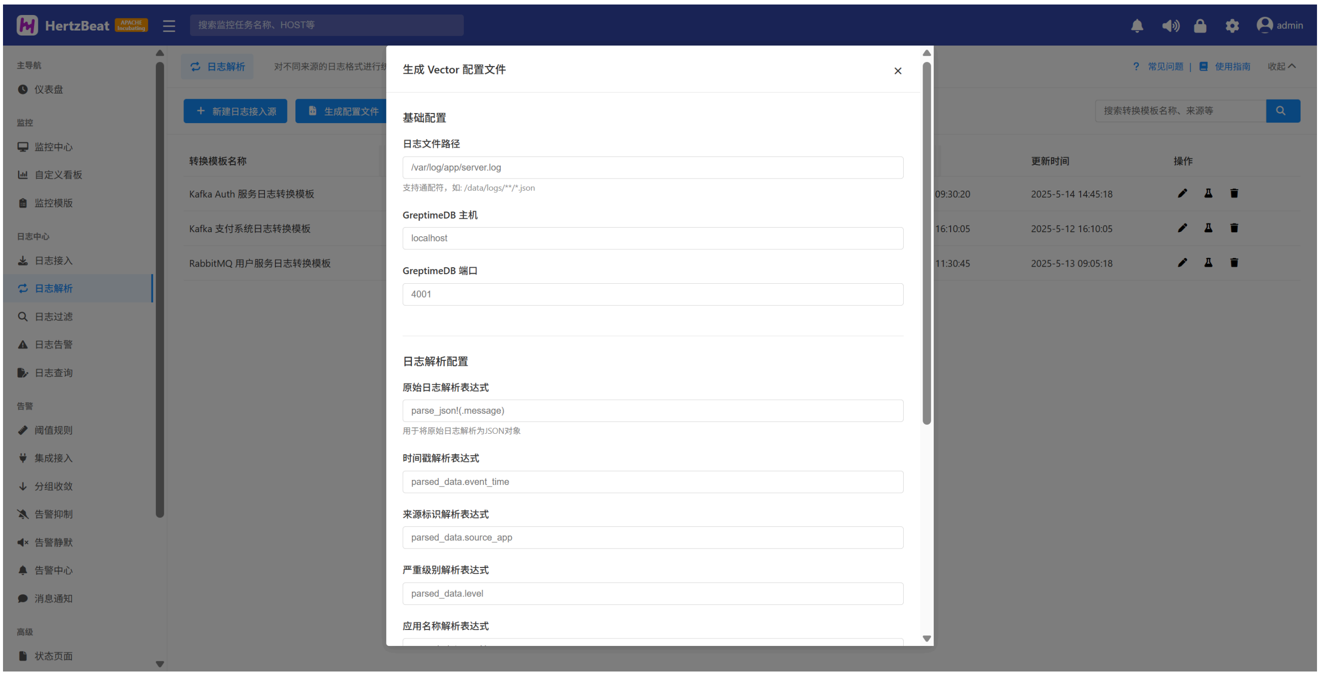
需要用户填写的值：

- `${log_path}`
  - 指定 Vector 监控的单个或多个日志文件。
    - **输入**: 字符串，表示有效的文件路径。可使用通配符如 `*` 或 `**`。
    - **示例**: `/var/log/app/server.log`, `/data/logs/**/*.json`
- `${json_log_payload}`
  - 用于将原始日志行（通常在 `.message` 或 `.` 中）解析为 `parsed_data` 对象。通常是 JSON 解析。
    - **示例**: `parse_json!(.message)`, `parse_json!(string!(.))`
- `${timestamp_parse_script}`
  - 从 `parsed_data` 中提取并转换为 `timestamp`。
    - **输入**: VRL 表达式字符串。
    - **示例**: `parsed_data.event_time`, `parse_timestamp!(parsed_data.logTime, "%Y-%m-%dT%H:%M:%S.%fZ")`, `parsed_data.time ?? now()`
- `${source_key_parse_script}`
  - 从 `parsed_data` 或事件元数据中提取日志来源标识。

        - **输入**: VRL 表达式字符串。
        - **示例**: `parsed_data.source_app`, `"my-app-source"`
- `${severity_parse_script}`
    - 从 `parsed_data` 中提取日志级别 (如 `ERROR`, `INFO`)。
        - **输入**: VRL 表达式字符串。
        - **示例**: `parsed_data.level`, `upcase(parsed_data.log_level)`
- `${application_parse_script}`
    - 从 `parsed_data` 或事件元数据中提取应用名。
        - **输入**: VRL 表达式字符串。
        - **示例**: `parsed_data.appName`, `"payment-service"`
- `${trace_id_parse_script}`
    - 从 `parsed_data` 中提取 Trace ID。
        - **输入**: VRL 表达式字符串。
        - **示例**: `parsed_data.trace.id`, `parsed_data.contextMap.traceId ?? ""`
- `${span_id_parse_script}`
    - 从 `parsed_data` 中提取 Span ID。
        - **输入**: VRL 表达式字符串。
        - **示例**: `parsed_data.span.id`, `parsed_data.contextMap.spanId ?? ""`
- `${message_parse_script}`
    - 从 `parsed_data` 中提取或构建日志消息。
        - **输入**: VRL 表达式字符串。
        - **示例**: `parsed_data.log_message`, `"${parsed_data.user}: ${parsed_data.action}"`
- **属性名 (由用户定义，对应模板中的 `${key}`)**
    - 希望在 `.attributes` 对象中使用的键名。
        - **输入**: 字符串，作为属性的键。
        - **示例**: `user_id`, `http_status_code`, `region`
- **属性值表达式 (由用户定义，对应模板中的 `${value}`)**
    - 一个 VRL 表达式，用于从 `parsed_data` 提取该属性的值或定义静态值。
        - **输入**: VRL 表达式字符串。
        - **示例**: `parsed_data.internal_user_id`, `to_int(parsed_data.response.status_code)`, `"prod-cluster"`, `150.25` (数字), `true` (布尔)
- `${host}`
    - GreptimeDB 主机名或 IP 地址。
        - **输入**: 字符串，表示有效的主机名或 IP。
        - **示例**: `localhost`, `greptimedb.example.com`, `10.0.0.5`
- `${port}`
    - GreptimeDB gRPC 端口号。
        - **输入**: 数字或数字字符串。

- **示例**: `4001`

前端界面示例：



3.3 日志格式统一

获取到的日志需要处理成统一的格式，便于后续的过滤、阈值计算、告警。因此设计如下的日志格式：

```json
{
  "timestamp": "xxxxxxxxxxxxxxx",
  "source_key": "filebeat",
  "severity": "ERROR",
  "application": "auth-service",
  "trace_id": "abc123",
  "span_id": "def456",
  "message": "Database connection failed",
  "attributes": {
    "user_id": "1001",
    "device_id": "device-xyz",
    "response_time": "1200"
  }
}
```

字段说明：

- `timestamp`：用于记录日志生成的时间，便于进行查询、排序和存储到时序数据库，以支持后续的时间窗口分析和监控。
- `source_key`：用于标识日志来源（如 `filebeat`，`opentelemetry`，`kafka`），方便日志格式转换。

- `severity`：表示日志级别，包括 `DEBUG`，`INFO`，`WARN`，`ERROR`，`FATAL` 等标准取值，有助于统一筛选并进行告警规则的配置和执行。
- `application`：用于标识日志所属的应用或服务名称，便于基于应用维度进行聚合分析和异常定位，特别适用于多服务架构中的日志管理和监控。
- `trace_id`：用于追踪请求链路，通常由 `OpenTelemetry` 或分布式链路追踪系统生成，可以关联跨服务请求，支持分布式环境中的问题追踪和链路分析。
- `span_id`：表示当前请求链路中的具体节点，与 `trace_id` 结合使用，用于进一步细化定位具体服务节点的问题，帮助快速溯源和排查。
- `message`：用于存储日志的核心内容或描述信息，是异常告警、日志分析和快速排查问题的主要依据。
- `attributes`：可选字段，采用键值对（K-V）形式存储上下文数据（如用户ID、设备信息、响应时间等），方便扩展和深度分析，不影响日志主结构查询。

3.3.1 日志格式转换

不同的应用以及不同数据源的日志格式不尽相同。因此对于获取到的日志，我们可以让用户配置日志转换模板，在创建日志数据源时绑定模板。日志转换底层可以参照目前系统中有的 JsonPath 进行处理将其统一转为 HertzBeat 固定的日志格式。

所有需要设计 `hzb_log_source_bindings` 和 `hzb_log_converter_templates`，表结构如下：

> 一个数据源只能绑定一个转换模板，一个转换模板可以被多个数据源绑定

名称	类型	描述	约束
`id`	整数	主键，唯一标识一条绑定关系。	`PRIMARY KEY`
`source_key`	文本	日志来源的唯一标识（例如：`kafka-topic1`，`filebeat-appA`）。每个日志源只能绑定到一个转换器模板。	`NOT NULL`，`UNIQUE`
`template_id`	整数	关联到 `hzb_converter_templates` 表中的 `template_id`，指定该日志源使用哪个转换器模板。	`NOT NULL`
`created_at`	日期时间	绑定关系创建时间。	(通常由数据库或应用自动设置)
`updated_at`	日期时间	绑定关系最后更新时间。	(通常由数据库或应用自动设置)

名称	类型	描述	约束
`id`	整数	主键，唯一标识一个转换器模板。	`PRIMARY KEY`

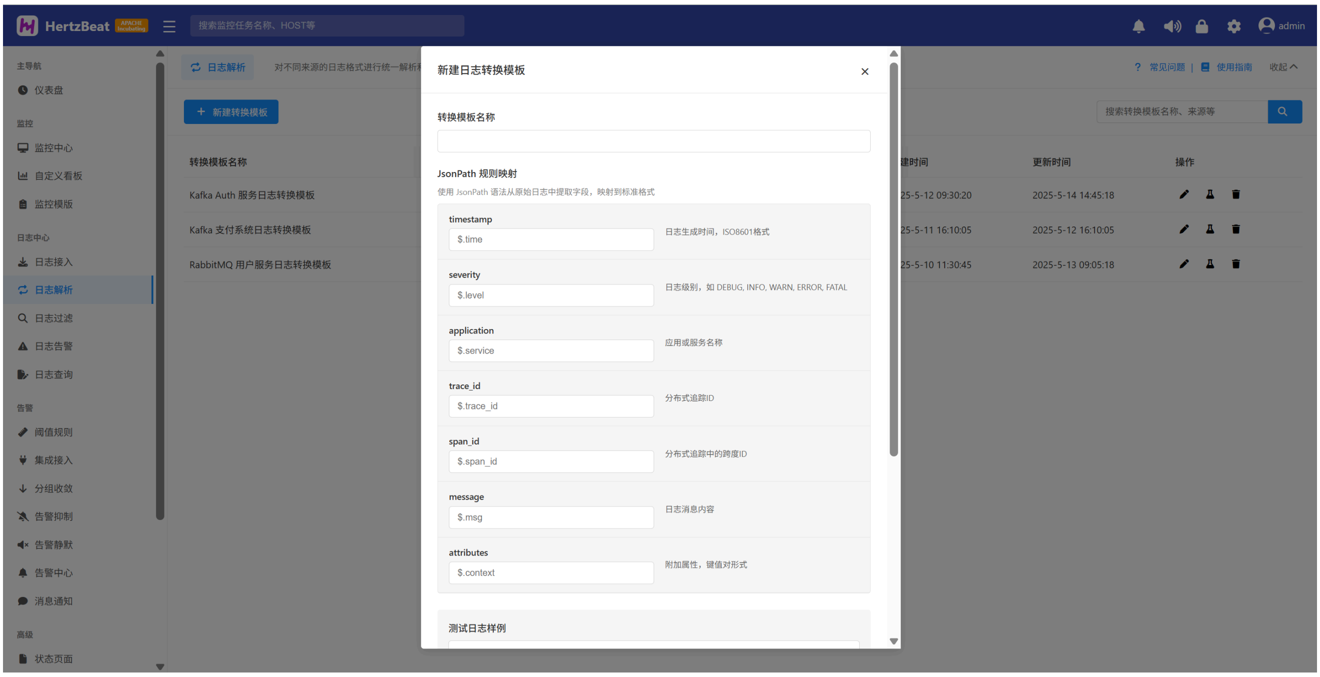
名称	类型	描述	约束
`template_name`	文本	转换器模板的名称，方便用户识别和选择（例如：“通用JSON解析器”，“Nginx访问日志格式化器”）。	`NOT NULL`
`json_path_rules`	JSON 结构	JSON 格式的文本，定义字段提取和映射规则。例如：`{"timestamp": "$.time", "severity": "$.level", "message": "$.msg"}`	`NOT NULL`
`template_notes`	文本	可选字段，用于对该转换器模板进行更详细的描述或备注。	
`created_at`	日期时间	模板创建时间。	(通常由数据库或应用自动设置)
`updated_at`	日期时间	模板最后更新时间。	(通常由数据库或应用自动设置)

具体的转换逻辑如日志处理工厂类所示：先从数据库读取，解析为 Map，开启线程拉取日志，根据日志key获取转换器，执行转换，放入新的队列。

```
// 下面是伪代码
// @Component
public class LogProcessingFactory {

    private final Map<String, LogConverter> converterMap = new
ConcurrentHashMap<>();
    private final ExecutorService conversionExecutor; // 用于执行转换任务
    private final CommonDataQueue sharedLogQueue;
    private final LogConverterService logConverterService;
    private final ScheduledExecutorService pollerScheduler; // 用于定时拉取共
享队列

    @Autowired
    public LogProcessingFactory(ExecutorService conversionExecutor,
                                CommonDataQueue sharedLogQueue,
                                LogConverterDao logConverterDao) {
        this.conversionExecutor = conversionExecutor;
        this.sharedLogQueue = sharedLogQueue;
        this.logConverterDao = logConverterDao;
        this.pollerScheduler = Executors.newScheduledThreadPool(1);
        init();
        start();
    }

    public void init() {
        Map<String, LogConverter> mapping = logConverterService.map();
```

```java
        this.converterMap.putAll(mapping);
    }

    public void start() {
        pollerScheduler.scheduleWithFixedDelay(this::pollAndDispatchLogs,
                                               0, 100,
  TimeUnit.MILLISECONDS);
    }

    private void pollAndDispatchLogs() {

        List<String> rawLogs = sharedLogQueue.pullRowLogs();

        for (String rawLog : rawLogs) {

            String actualSourceKey = rawLog.sourceKey();
            LogConverter converter = converterMap.get(actualSourceKey);
            // 将转换任务提交给专门的转换线程池
            conversionExecutor.execute(() -> {
                LogEntry logEntry = converter.convert(rawLog);
                // 放入日志队列
                sharedLogQueue.sendLog(logEntry);
            });
        }
    }
}
```

前端界面示例：



3.4 日志过滤（扩展内容）

当日志转为统一的结构后，就可以实现表达式过滤了，可以复用 HertzBeat 中已有的 JexlExpression引擎进行日志的过滤。

因此我们需要设计 `hzb_log_filtering_policies` 表，结构如下：

名称	类型	描述	约束
`id`	整数	策略的唯一标识符。	`PRIMARY KEY`
`name`	文本	策略的名称。	`NOT NULL`
`source_key`	文本	用于快速匹配日志的 `source_key`。	`NOT NULL`
`jexl_expression`	文本	JEXL 过滤表达式主体。当 `source_key` 匹配后，此表达式用于对单条日志进行求值。	`NOT NULL`
`priority`	整数	策略的执行优先级。数字越小，优先级越高。用于处理当多个策略的 `source_key` 都匹配同一条日志时的情况。(通常有一个默认值，如 100)	`NOT NULL`
`is_enabled`	布尔值	策略是否启用。	`NOT NULL`
`creater`	文本	策略创建者。	`NOT NULL`
`created_at`	日期时间	策略创建的时间戳。(通常由应用程序在插入数据时记录，或数据库触发器/默认值设置为当前时间。)	`NOT NULL`
`updated_at`	日期时间	策略最后更新的时间戳。(通常由应用程序在更新数据时记录，或数据库触发器/默认值自动更新为当前时间。)	`NOT NULL`

整个过滤的逻辑为：

- 从上游日志队列拉取日志。
- 获取日志的 `source_key`。
- 找出所有匹配的策略，并按 `priority` 升序排序。
- 使用策略的 `jexl_expression` 对日志进行求值。如果表达式返回 `true`，将日志过滤掉
- 未过滤的日志通过 `ApplicationEventPublisher` 发送到下游处理

3.5 日志持久化

根据日志格式统一中提到的 Json 结构，创建表 `hzb_log` 结构如下：

名称	类型	描述	约束
`id`	整数	日志唯一标识符。(在具体数据库实现时，通常配置为自动生成或自增。)	`PRIMARY KEY`

名称	类型	描述	约束
`timestamp`	日期时间	日志的原始时间戳。	`NOT NULL`
`level`	文本	日志级别 (例如 INFO, ERROR, WARN)。	`NOT NULL`
`source_key`	文本	日志来源标识，用于关联日志源。	`NOT NULL`
`message`	文本	日志消息主体。	`NOT NULL`
`trace_id`	文本	分布式追踪ID，可选。	
`span_id`	文本	分布式追踪中的跨度ID，可选。	
`attributes`	JSON结构	日志的附加属性，以JSON格式存储，可选。	
`created_at`	日期时间	日志记录入库时间。(通常由应用程序在插入数据时记录，或数据库通过默认机制设置为当前时间。)	

LogStorageListener 监听到日志事件后通过异步的方式将日志持久化到数据库。由于不是所有用户都需要日志监控这一部分功能，如果强制使用时序数据库存储日志，增加了项目的复杂度，因此可以参照指标数据持久化的策略，在 application 中配置日志持久化的数据库。

```
// 下面的内容为伪代码
@Component
public class LogStorageListener {

    // 持久化策略依然通过依赖注入获得
    private final Optional<LogPersistence> logPersistenceStrategy;
    @Autowired
    public LogPersistenceListener(Optional<LogPersistence> strategy){
        this.logPersistenceStrategy = strategy;
    }

    /**
     * 监听日志持久化事件，并使用自定义的线程池异步执行。
     * @param event 包含待持久化日志的事件对象
     */
    @Async(AsyncConfig.LOG_PERSISTENCE_EXECUTOR)
    @EventListener
    public void handleLogPersistence(LogPersistenceEvent event) {
        logPersistenceStrategy.ifPresentOrElse(
            actualPersistence -> {
                LogEntry log = event.getLogEntry();
                if (log == null) {
                    return; // 如果没有日志，则直接返回
                }
                actualPersistence.saveLog(log);
            },
            () -> {
```

```
                logger.warn("LogPersistence 策略未配置，跳过日志持久化。");
            }
        );
    }
}
```

```
// 下面的内容为伪代码
public abstract class LogPersistence {
    /**
     * 持久化日志
     * @param logEntry 要保存的日志
     */
    public abstract void saveLog(LogEntry logEntry);
}
```

```
// 下面的内容为伪代码
@Component
@ConditionalOnProperty(prefix = "log.store.jpa", name = "enabled",
havingValue = "true")
public class JpaLogPersistence extends LogPersistence {

    private final LogDao logDao;

    @Autowired
    public JpaLogPersistence(LogDao logDao) {
        this.logDao = logDao;
    }

    @Override
    public void saveLog(LogEntry logEntry) {
        logDao.insertLog(logEntry);
    }
}
```

```
// 下面的内容为伪代码
@Component
@ConditionalOnProperty(prefix = "log.store.greptime", name = "enabled",
havingValue = "true")
public class GreptimeLogPersistence extends LogPersistence{

    private GreptimeDB greptimeDb;

    @Autowired
    public GreptimeLogPersistence(GreptimeDB greptimeDb) {
        this.greptimeDb = greptimeDb;
    }
```

```java
    @Override
    public void saveLog(LogEntry logEntry) {
        TableSchema tableSchema = TableSchema.newBuilder("hzb_logs")
                .addTimestamp("timestamp", DataType.TimestampMillisecond)
                .addTag("source_key", DataType.String)
                .addTag("level", DataType.String)
                .addField("message", DataType.String)
                .addField("trace_id", DataType.String)
                .addField("span_id", DataType.String)
                .addField("attributes", DataType.Json)
                .build();

        Table table = Table.from(tableSchema);

        Object[] values = new Object[]{
            logEntry.timestamp(),
            logEntry.source_key(),
            logEntry.level(),
            logEntry.message(),
            logEntry.traceId(),
            logEntry.spanId(),
            logEntry.attributes()
        };
        table.addRow(values);

        greptimeDb.write(table);
    }
}
```

3.6 日志告警

对于日志的告警，我们可以参照指标告警，支持阈值实时计算以及周期检测。

首先我们需要创建 `hzb_log_alert_define` 表，表结构如下：

名称	类型	描述	约束
`id`	整数	告警定义唯一标识符。	`PRIMARY KEY`
`name`	文本	规则名称。	`NOT NULL`
`creator`	文本	创建者。	`NOT NULL`
`modifier`	文本	修改者。	`NOT NULL`
`enable`	布尔值	是否启用 (0-禁用，1-启用)。(通常有一个默认值，如 0)	`NOT NULL`
`expr`	文本	告警表达式。	`NOT NULL`
`labels`	文本	标签 (key:value 格式，支持多个标签)，可选。	

名称	类型	描述	约束
`period`	整数	周期 (日志告警周期，单位：秒)。	`NOT NULL`
`type`	文本	告警规则类型 (例如 周期计算 / 实时计算)。	`NOT NULL`
`create_at`	日期时间	创建时间。(通常由应用程序在插入数据时记录，或数据库通过默认机制设置为当前时间。)	
`update_at`	日期时间	更新时间。(通常由应用程序在更新数据时记录，或数据库通过默认机制在更新时自动设置为当前时间。)	

3.6.1 实时阈值计算

实时阈值计算可利用现有的 JexlExpression 引擎处理。由于实时计算通常仅需分析单条日志的字段值，而非执行 SQL 统计查询，因此我们专注于日志字段的实时判断。

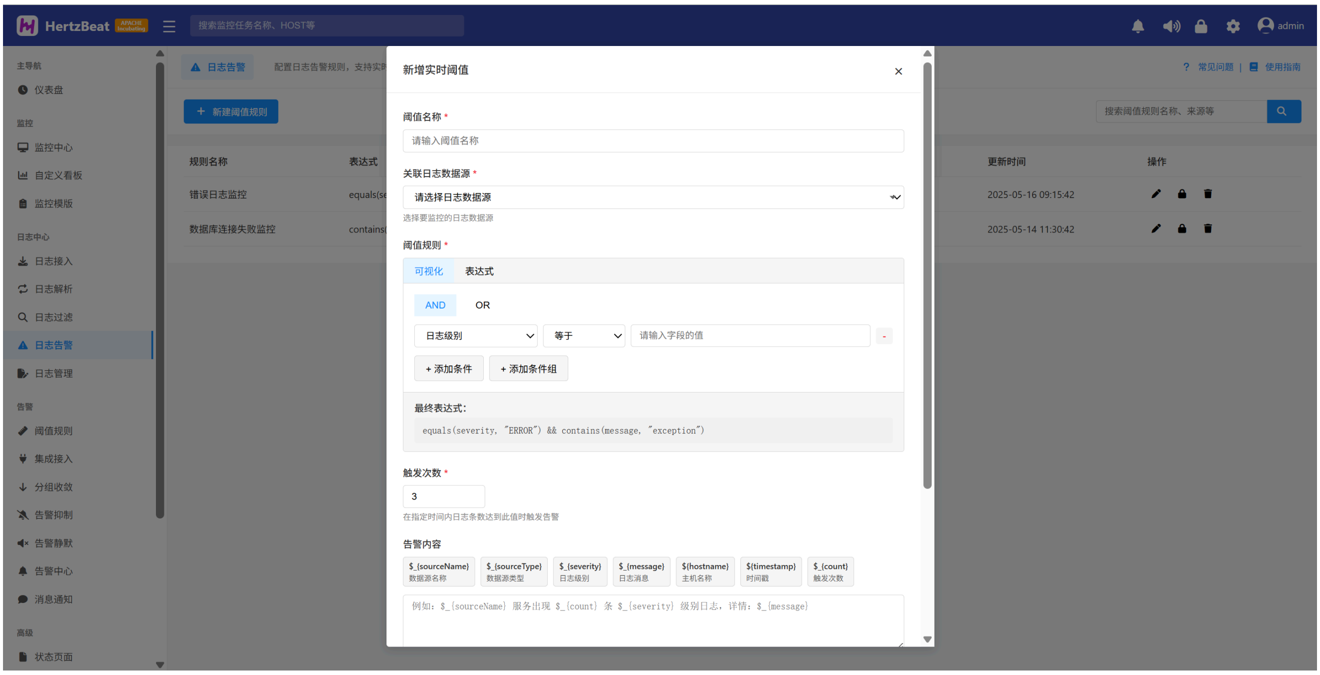
步骤：

1. 阈值规则过滤：先用数据源 key 做快速匹配
2. Eval：命中规则后用 JEXL 解析器运行 `expr`
3. 告警：表达式返回 `true` 即传入后续告警

示例：

类别	描述	示例表达式
关键字匹配	日志消息中包含某些关键字	contains(message, "Connection Timeout")
多字段组合	基于多个字段的组合规则	equals(application, "auth-service") && equals(severity, "ERROR")
状态监控	某些字段值出现异常	toInteger(attributes.response_time) > 1000

前端界面示例



3.6.2 周期阈值计算

对于日志告警，周期性计算是常用策略。它依赖用户定义时间窗口，并执行 `SQL` 对窗口内日志进行聚合统计。 `SQL` 查询结果有严格限制：仅返回单行，且列类型为数值或布尔，以避免复杂解析。 `SQL` 执行结果可直接复用现有的表达式进行告警判断。

1. 调度器根据配置的周期触发检查
2. 连接数据库执行 `SQL` ，返回 `ResultSet` -> `List<Map<String,Object>>`
3. 得到整体表达式结果
4. 结果为 `true` 触发告警

示例1：数量阈值

在指定时间窗口内，日志数量超过指定阈值，例如：每分钟 `ERROR` 日志 > 100 条

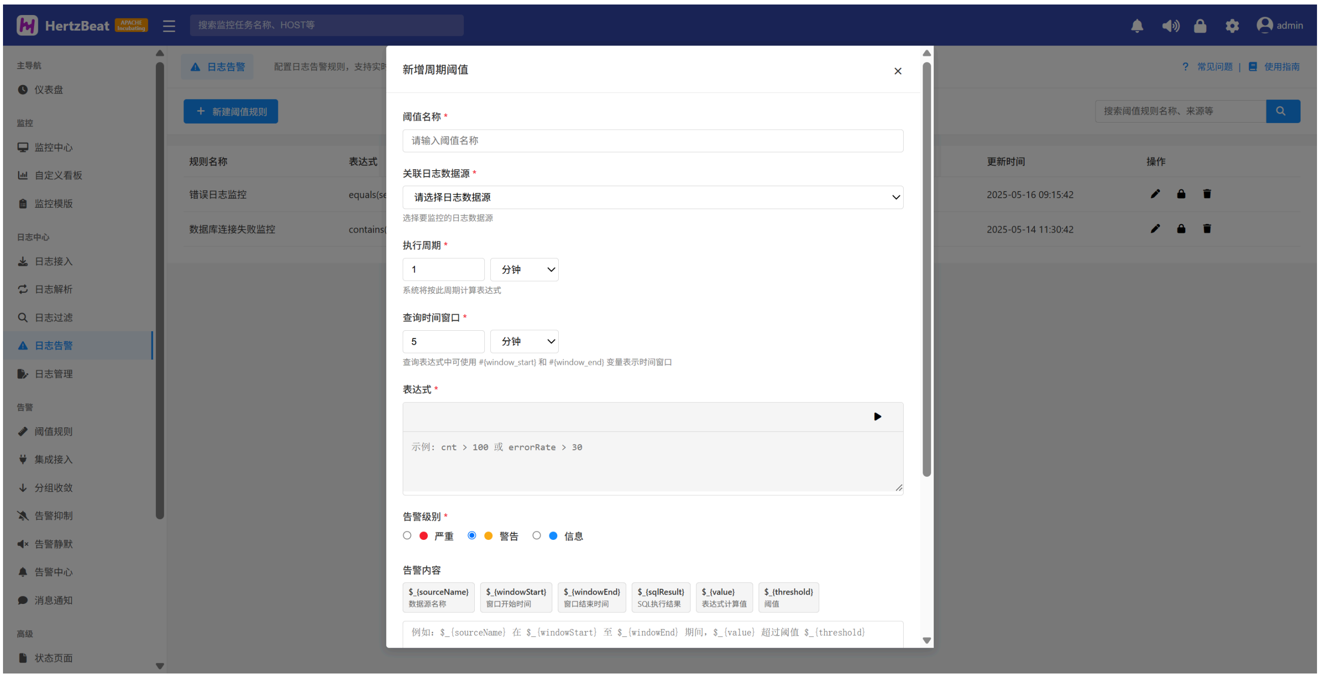
```
-- 聚合 SQL：统计窗口内 ERROR 日志总数
sql("SELECT COUNT(*) AS cnt FROM hzb_logs WHERE severity = 'ERROR'   AND
timestamp BETWEEN xxx AND xxx") > 100
```

示例2：比例阈值

在指定时间窗口内，某类日志占比超过阈值，例如：ERROR 日志占所有日志比例 > 30 %

```
sql("
SELECT (COUNT(CASE WHEN severity = 'ERROR' THEN 1 END) * 100.0 / COUNT(*))
AS errorRate FROM hzb_logs WHERE timestamp BETWEEN #{window_start} AND #
{window_end}
") > 30
```

前端界面示例：



3.7 告警通知

阈值触发告警后的逻辑可以完全复用现有的告警通知逻辑。构造 SingleAlert 后使用 alarmCommonReduce 进行告警的分组、抑制、静默、通知。

3.8 实时日志查看

将收集到的日志经过统一处理后，放入实时推送队列，前端与后端建立 SSE 连接，后端将日志推送到前端。前端在建立连接时可指定日志的过滤条件（sourceKey，severity，application，trace_id，span_id，message）。

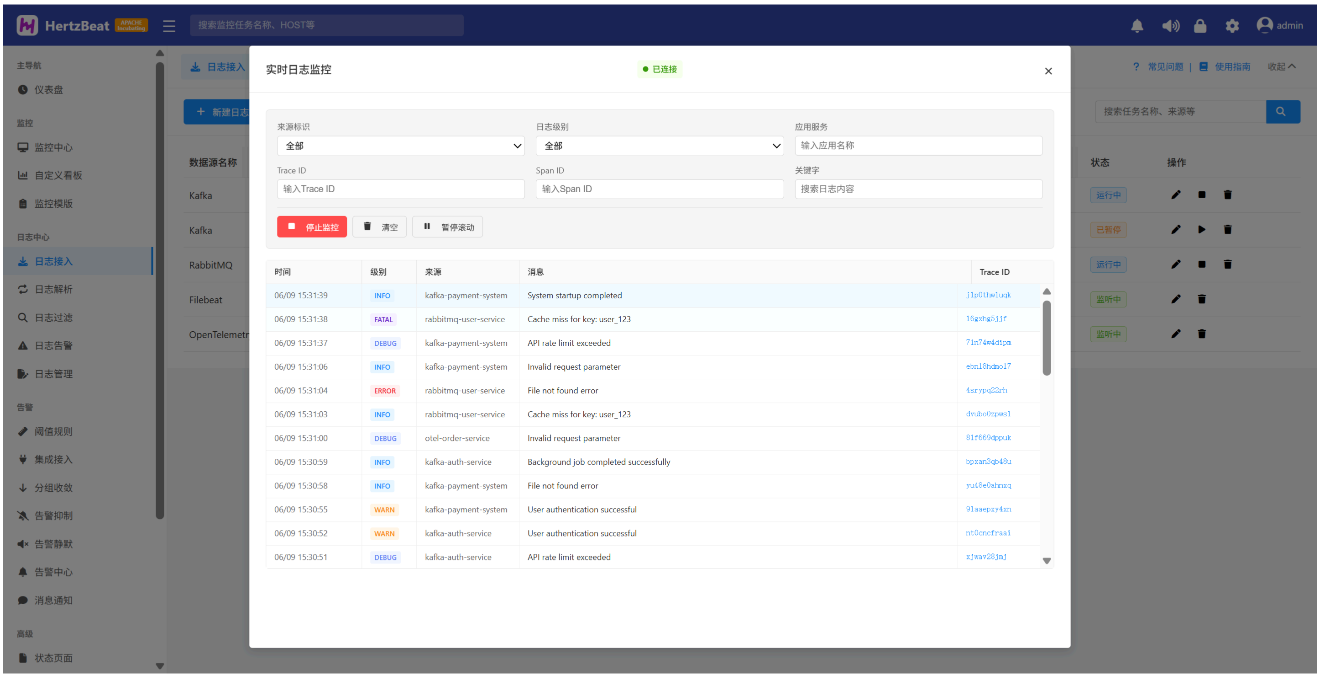
伪代码：

```
// 以下为伪代码
@Data
public class LogSSEFilterCriteria {
    private String sourceKey;
    private String severity;
    private String application;
    private String traceId;
    private String spanId;
    private String message;

    /**
     * 核心过滤逻辑
     * @param log 待检查的日志
     * @return boolean 是否匹配
     */
    public boolean matches(logEntry log) {
        if (StringUtils.hasText(severity) &&
!severity.equalsIgnoreCase(log.getSeverity())) return false;
```

```java
        if (StringUtils.hasText(application) &&
!application.equalsIgnoreCase(log.getApplication())) return false;
        if (StringUtils.hasText(traceId) &&
!traceId.equalsIgnoreCase(log.getTraceId())) return false;
        if (StringUtils.hasText(spanId) &&
!spanId.equalsIgnoreCase(log.getSpanId())) return false;
        if (StringUtils.hasText(sourceKey) &&
!sourceKey.equalsIgnoreCase(log.getSourceKey())) return false;
        if (StringUtils.hasText(message) &&
!log.getMessage().toLowerCase().contains(message.toLowerCase())) return
false;
        return true;
    }
}

@Component
public class SsePushService {

    private static class SseSubscriber {
        final SseEmitter emitter;
        final LogFilterCriteria filters;

        public SseSubscriber(SseEmitter emitter, LogFilterCriteria filters){
            ....
        }
    }

    private final Map<String, List<SseSubscriber>> subscriberMap = new
ConcurrentHashMap<>();

    public SseEmitter register(String sourceKey, LogFilterCriteria filters)
{
        // 将 new SseSubscriber(...) 添加到 appSubscriberMap 中
        // 并设置 emitter.onCompletion/onTimeout/onError 清理回调
    }

    public void push(LogEntity log){
        if (log == null) {
            return;
        }
        // 根据日志的 application 找到所有订阅者
        List<SseSubscriber> subscribers =
appSubscriberMap.get(log.sourceKey());
        if (subscribers == null || subscribers.isEmpty()) {
            return;
        }

        // 遍历订阅者，执行过滤和推送
        subscribers.forEach(sub -> {
            if (sub.filters.matches(log)) {
```

```java
                try {
                    sub.emitter.send(log, MediaType.APPLICATION_JSON);
                } catch (IOException e) {
                    sub.emitter.completeWithError(e); // 触发清理
                }
            }
        });
    }
}

/**
* 这个类主要用于监听日志推送事件，发送到前端
*/
@Component
public class LogPushListener {

    private final SsePushService ssePushService;

    /**
     * 监听日志推送事件，并使用专用的线程池异步执行。
     * @param event 包含待推送日志的事件对象
     */
    @Async(AsyncConfig.LOG_PUSH_EXECUTOR)
    @EventListener
    public void handleLogPush(LogPushEvent event) {

        LogEntity logToPush = event.getLogEntity();
        if (logToPush == null) {
            return;
        }
        ssePushService.push(logToPush);
    }
}
```

前端示例：



3.9 日志管理

3.9.1 日志管理可视化等相关 API

主要是提供日志查询、删除 API，用于前端数据展示。

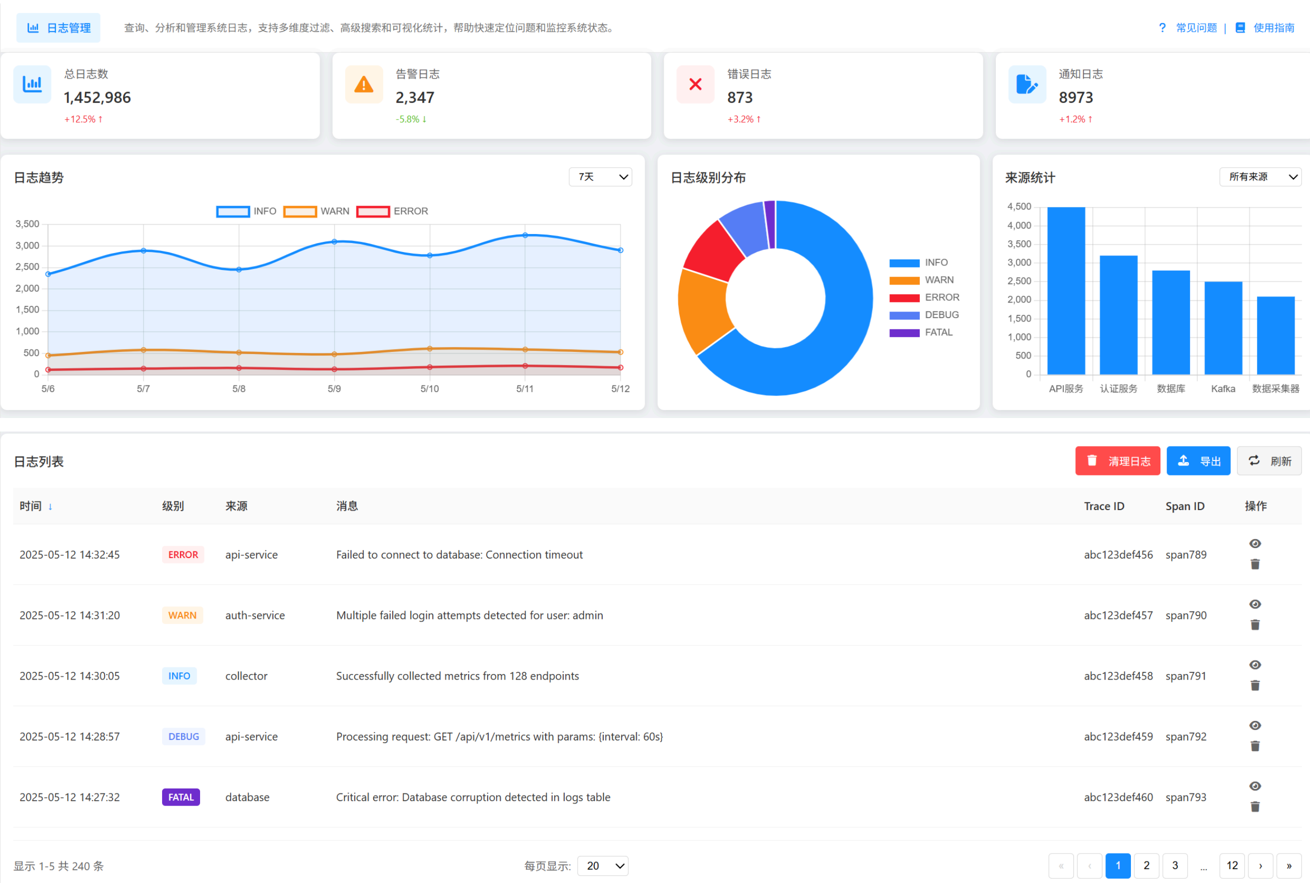
- `GET /api/logs`
  - 参数：`timestampStart`，`timestampEnd`，`level`，`source`，`order`，`sort`，`search`，`pageIndex`，`pageSize`
  - 返回：分页日志列表
- `GET /api/logs/{id}`
  - 返回：指定日志详细信息，包括 `trace_id`，`span_id`，`attributes`
- `DELETE /api/logs/{id}`
  - 删除指定日志记录。
- `GET /api/logs/statistics`
  - 返回：日志数量统计（按时间、等级、来源分类）
- `DELETE /api/logs/cleanup`
  - 参数：`timestampBefore` （删除指定时间之前的日志数据）

3.9.2 日志管理与展示页面

- **页面设计：**
  - **查询表单：**
    - 时间范围选择器：支持时间范围快速选择（今天、近7天、近30天）
    - 日志等级下拉菜单（INFO, WARN, ERROR, FATAL, DEBUG）
    - 来源选择器
    - Trace ID/Span ID 输入框
  - **日志列表：**

    - 列表字段：`timestamp`，`level`，`source`，`message`，`trace_id`，`span_id`
    - 支持分页、排序、关键字搜索
    - 日志详情展开面板：显示 `attributes` 的 JSON 格式化视图
- **日志详情页面：**
    - 显示完整日志信息，包括 `attributes` 中的详细信息
    - 支持 JSON 树状结构视图展开
- **可视化组件：**
    - 折线图：展示按时间范围的日志数量变化
    - 饼图：按日志等级的占比分析
    - 柱状图：按来源划分的日志数量分布

前端界面示例：



四、实施计划

我预计每周可投入 25-35 小时，平均每天 5 小时进行开发与学习。核心开发周期为 **7月1日 - 9月30日**。

计划将项目开发分为四个主要阶段，外加一个开发预热期。每个阶段都有明确的目标和任务分解，确保项目稳步推进。

4.1 开发预热 (6月25日 - 6月30日)

目标： 为正式开发做好充分准备，熟悉现有系统，确认技术细节。

时间	预热任务
2天	学习 HertzBeat 现有项目架构，特别是指标采集、告警处理、数据持久化模块。
1天	再次详细阅读并与导师确认日志监控功能的设计方案，明确所有细节和潜在问题。
2天	根据设计方案，完成 `hzb_log_source`，`hzb_log_source_bindings`，`hzb_log_converter_templates`，`hzb_log` 表的最终设计与数据库迁移脚本。。
1天	创建个人开发分支，规划日志监控模块的代码组织结构。

4.2 第一阶段：日志的获取转换与存储 (7月1日 - 7月28日，4周)

目标： 实现日志从接收、格式转换到基础持久化的完整核心流程。

时间	周任务
第1周	1. 实现日志被动接收 API (`LogIngestionController`) 及原始日志队列 (`rawLogQueue`) 功能。 2. 定义 HertzBeat 统一日志格式。 3. 设计日志监控栏的侧边栏。 4. 数据源接入界面开发。
第2周	1. 实现日志处理工厂 (`LogProcessingFactory`) 框架，支持从数据库动态加载并解析为 `LogConverter` 。 2. 实现基于 JsonPath 的日志格式转换核心逻辑 (`LogConverter.convert`)。 3. 实现从原始日志队列消费日志，进行格式转换，并通过 ApplicationEventPublisher 告知下游处理。
第3周	1. 实现日志持久化监听器 (`LogStorageListener`)，持久化日志。 2. 实现日志持久化策略 (`JpaLogPersistence` 、 `GreptimeLogPersistence`)，将日志存入 `hzb_log` 表。 3. 实现 Kafka 消费者：支持从数据库加载配置，启动消费者，将日志放入原始日志队列。 4. 实现配置文件的生成。
第4周	1. 数据源接入的删、查、改。 2. 日志转换模板界面开发。 3. 日志转换模板的删、查、改。 4. 编写核心模块的测试。

4.3 第二阶段：日志告警机制实现 (7月29日 - 8月25日，4周)

目标： 基于已标准化的日志数据，实现实时告警和周期性告警功能，并与现有告警通知体系集成。

时间	周任务
第5周	1. 实现实时阈值计算逻辑：根据 `hzb_log_alert_define` 中配置的实时规则 (JEXL 表达式) 进行匹配和计算。

时间	周任务
	2. 阈值管理界面。 3. 阈值的删查改。
第6周	1. 实现周期阈值计算框架：包括调度器按配置周期触发任务，从数据库加载周期性告警规则。 2. 实现周期性告警规则中的 SQL 查询执行功能，支持动态替换时间窗口参数 (`#{window_start}`, `#{window_end}`)。
第7周	1. 实现周期阈值计算逻辑。 2. 将触发的日志告警 (实时和周期性) 构造成 `SingleAlert` 对象，并对接到 HertzBeat 现有的 `alarmCommonReduce` 模块进行后续处理（分组、抑制、静默、通知）。
第8周	1. 周期阈值配置的前端界面。 2. 编写告警模块的测试。

4.4 第三阶段：日志管理实现 (8月26日 - 9月15日，3周)

目标： 提供日志数据的管理接口和基础的前端查询与展示页面。

时间	周任务
第9周	1. 实现日志实时推送监听器。 2. 实现日志实时显示前端。 3. 实现日志查询、删除相关API。
第10周	1. 实现日志统计API。 2. 日志管理前端页面。
第11周	1. 日志统计的可视化图表。 2. 完成所有功能前后端联调。

4.5 第四阶段：功能完善、扩展功能、测试 (9月16日 - 9月30日，2周)

目标： 对系统进行全面的测试和优化，准备最终的项目交付

时间	周任务
第12周	1. 编写详细的项目文档，包括功能说明、配置指南、API 文档和用户手册。 2. 进行代码审查，准备最终的项目 PR/MR。

时间	周任务
第13周	1. (扩展) 日志过滤：如果时间和优先级允许设计并实现 `hzb_log_filtering_policies` 表及相关管理接口 。 2. (扩展) 实现基于 JEXL 的日志过滤逻辑，在日志处理工厂和标准化日志队列之间加入过滤步骤。 3. (扩展) 实现过滤策略前端界面