日志系统说明书

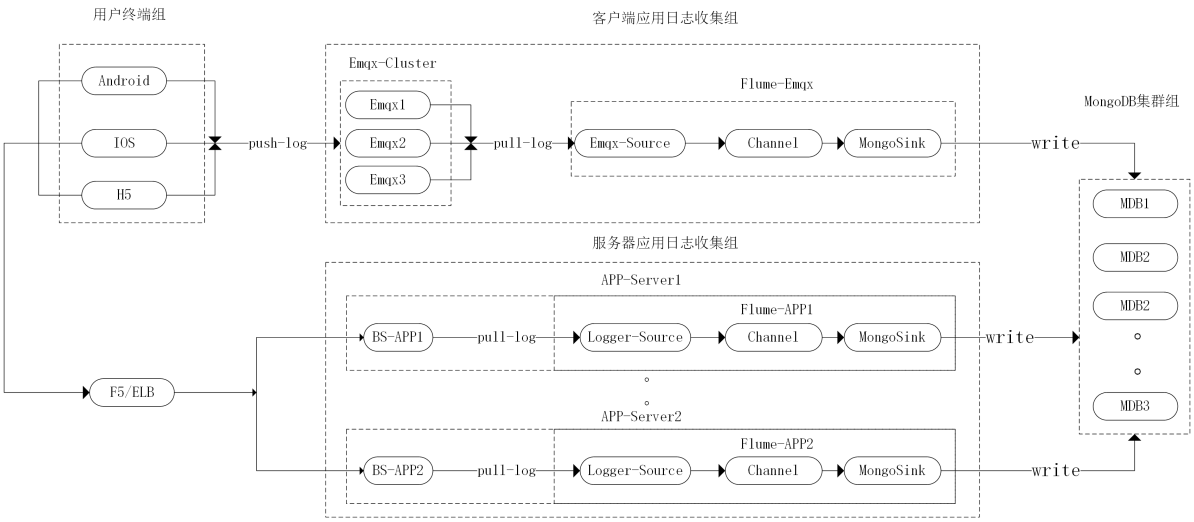
日志系统包括日志收集系统和数据分析平台两部分，日志收集系统是一套基于数据采集、转换、抽取和存储的ETL数据模型架构，其目的旨在于为数据分析平台持续提供用于统计分析的结构化数据；数据分析平台是通过对数据源进行读取、筛选、连接、分组、聚合等一系列操作的应用系统，其目的旨在于通过统计手段形成一系列可供上层分析的应用报表

一、日志收集系统

日志收集系统是一套基于数据采集、转换、抽取和存储的ETL数据模型架构，其目的旨在于为数据分析平台持续提供用于统计分析的结构化数据

1.1、日志收集系统架构图

本套日志收集系统采用Emqx+Flume+MongoDB+Metabase流程架构，架构示意图如下：



说明：

Flume被安装到每一台应用服务器上，Flume中不同的Source组件扫描来自不同应用日志目录下的日志文件（它们可能是来自服务端的应用日志或客户终端的应用日志），将来自日志文件或远端队列中的实时新增日志数据读取出来推送到通道组件中去，通道组件对接到MongoSink组件，由MongoSink组件将从通道接收到的日志数据进行过滤处理并转换成JSON对象后，将其推送到MongoDB集群服务组

1.2、Flume插件过滤器接口规范

1. 基于MongoDB存储的MongoSink插件过滤器接口规范

```
package com.github.lixiang2114.flume.plugin.mdb.filter;

import java.util.HashMap;
import java.util.Map;
```

```java
import java.util.Properties;

/**
 * @author Louis(LiXiang)
 * @description 自定义Sink过滤器接口规范
 */
public interface MdbSinkFilter {
    /**
     * 获取数据库名称
     * @return 索引名称
     */
    public String getDataBaseName();

    /**
     * 获取集合名称
     * @return 索引类型
     */
    public String getCollectionName();

    /**
     * 处理文档记录
     * @param record 文本记录
     * @return 文档字典对象
     */
    public HashMap<String,Object>[] doFilter(String record);

    /**
     * 获取文档ID字段名
     * @return ID字段名
     */
    default public String getDocId(){return null;}

    /**
     * 获取登录密码
     * @return 密码
     */
    default public String getPassword(){return null;}

    /**
     * 获取登录用户名
     * @return 用户名
     */
    default public String getUsername(){return null;}

    /**
     * 过滤器上下文配置(可选实现)
     * @param config 配置
     */
    default public void filterConfig(Properties properties){}

    /**
     * 设置MongoDB客户端
     * @return ID字段名
     */
    default public void setMongoClient(Object mongoClient){}

    /**
     * 插件上下文配置(可选实现)
```

```
     * @param config 配置
     */
    default public void pluginConfig(Map<String,String> config){}
}
```

2. 基于Emqx消息中间件的EmqxSource插件过滤器接口规范

```
package com.github.lixiang2114.flume.plugin.emqx.filter.EmqxSourceFilter;

import java.util.Map;
import java.util.Properties;

/**
 * @author Louis(LiXiang)
 * @description SourceFilter过滤器接口规范
 */
public interface EmqxSourceFilter {
    /**
     * 获取主题列表(必须重写)
     * @return 主题列表
     */
    public String[] getTopics();

    /**
     * 日志数据如何通过过滤转换成文档记录(必须重写)
     * @param record
     * @return 文档记录
     */
    public String[] doFilter(String record);

    /**
     * 日志数据的通信质量列表
     * @return 质量列表
     */
    default public int[] getQoses(){return null;}

    /**
     * 登录Emqx服务的密码(如果需要则重写)
     * @return 登录密码
     */
    default public String getPassword(){return null;}

    /**
     * 登录Emqx服务的用户名(如果需要则重写)
     * @return 登录用户
     */
    default public String getUsername(){return null;}

    /**
     * 生成登录Token的秘钥(如果需要则重写)
     * @return Token秘钥
     */
    default public String getJwtsecret(){return null;}

    /**
```

```
     * Token的有效期系数/因子(如果需要则重写)
     * @return 有效期系数
     */
    default public Integer getExpirefactor(){return 750;}

    /**
     * Token的过期时间(如果需要则重写)
     * @return 过期时间
     */
    default public Integer getTokenexpire(){return 3600;}

    /**
     * 使用过滤器配置初始化本过滤器实例成员变量(如果需要则重写)
     * @param properties 配置
     */
    default public void filterConfig(Properties properties){}

    /**
     * Token通过哪个字段携带到Emqx服务端
     * @return 携带字段
     */
    default public String getTokenfrom(){return "password";}

    /**
     * 使用Flume插件配置初始化本过滤器实例成员变量(如果需要则重写)
     * @param config 配置
     */
    default public void pluginConfig(Map<String,String> config){}
}
```

注意:

两个过滤器接口的设计规范完全相同，编写基于MongoDB的过滤器可以实现MongoSink接口，编写基于Emqx的过滤器可以实现EmqxSource接口，但实现这些接口并非是必须的，如果我们希望更易于编译过滤器源的代码，我们可以不用在代码中显式实现这些接口，而只需按照接口中定义的接口规范编写相应的方法即可

1.3、Flume插件过滤器接口实现

无论是MongoSink插件，还是EmqxSource插件，其实现的方式都是一样的，我们均可以采用两种方式去实现这些接口，一种是显式实现接口，另一种是直接编写方法实现(推荐)

1. 显式实现接口
   显式实现接口的方式是一种传统的适配方式，这种方式非常明确的体现出了接口定义的作用，它强制要求开发者必须实现哪些接口方法，而哪些接口方法又是可选实现的，同时规范了每一个方法实现的原型，让开发者完全按照接口中定义好的接口原型去对应实现这些方法

- 基于MongoDB的接口实现

```
import java.util.HashMap;
import java.util.Map;
import java.util.Properties;
import java.util.regex.Pattern;
```

```java
import com.github.lixiang2114.flume.plugin.mdb.filter.MdbSinkFilter;

/**
 * @author Louis(LiXiang)
 * @description 自定义日志过滤器
 */
public class MdbLoggerFilter implements MdbSinkFilter{
    /**
     * 字段列表
     */
    private String[] fields;

    /**
     * 登录MongoDB用户名
     */
    private static String userName;

    /**
     * 登录MongoDB密码
     */
    private static String passWord;

    /**
     * 文档索引类型
     */
    private String collectionName;

    /**
     * 文档索引名称
     */
    private String dataBaseName;

    /**
     * 日志记录字段分隔符
     */
    private String fieldSeparator;

    /**
     * 逗号正则式
     */
    private static Pattern commaRegex;

    @Override
    public String getDocId() {
        return fields[0];
    }

    @Override
    public String getPassword() {
        return passWord;
    }

    @Override
    public String getUsername() {
        return userName;
    }
```

```
    @Override
    public String getCollectionName() {
        return collectionName;
    }

    @Override
    public String getDataBaseName() {
        return dataBaseName;
    }

    @Override
    public HashMap<String, Object>[] doFilter(String record) {
        String[] fieldValues=commaRegex.split(record);
        HashMap<String,Object> map=new HashMap<String,Object>();
        map.put(fields[0], fieldValues[0].trim());
        map.put(fields[1], fieldValues[1].trim());
        map.put(fields[2], fieldValues[2].trim());
        return new HashMap[]{map};
    }

    @Override
    public void filterConfig(Properties properties) {
        commaRegex=Pattern.compile(fieldSeparator);
    }
}
```

- 基于Emqx的接口实现

```
import java.util.HashMap;
import java.util.Map;
import java.util.Properties;
import java.util.regex.Pattern;
import com.github.lixiang2114.flume.plugin.emqx.filter.EmqxSourceFilter;

/**
 * @author Louis(LiXiang)
 * @description 自定义日志过滤器
 */
public class EmqxLoggerFilter implements EmqxSourceFilter{
    /**
     * 通信质量表
     */
    private static int[] qoses;

    /**
     * 连接主题列表
     */
    private static String[] topics;

    /**
     * 登录Emqx密码
     */
    private static String passWord;

    /**
     * 登录Emqx用户名
     */
```

```
    private static String userName;

    /**
     * 登录验证Token的秘钥
     */
    private static String jwtSecret;

    /**
     * 携带Token的字段名
     */
    private static String tokenFrom;

    /**
     * Token过期时间
     */
    private static Integer tokenExpire;

    /**
     * Token过期时间因数
     */
    private static Integer expireFactor;

    @Override
    public int[] getQoses() {
        return qoses;
    }

    @Override
    public String[] getTopics() {
        return topics;
    }

    @Override
    public String getPassword() {
        return passWord;
    }

    @Override
    public String getUsername() {
        return userName;
    }

    @Override
    public String getJwtsecret() {
        return jwtSecret;
    }

    @Override
    public String getTokenfrom() {
        return tokenFrom;
    }

    @Override
    public Integer getTokenexpire() {
        return tokenExpire;
    }

    @Override
```

```
    public Integer getExpirefactor() {
        return expireFactor;
    }

    @Override
    public String[] doFilter(String record) {
        return new String[]{record};
    }

    @Override
    public void pluginConfig(Map<String, String> config) {
        String qosesStr=config.get("qoses");
        String topicsStr=config.get("topics");
        String jwtSecretStr=config.get("jwtSecret");
        String passWordStr=config.get("passWord");
        String userNameStr=config.get("userName");
        String tokenFromStr=config.get("tokenFrom");
        String tokenExpireStr=config.get("tokenExpire");
        String expireFactorStr=config.get("expireFactor");

        if(null!=qosesStr) {
            String qos=qosesStr.trim();
            if(0!=qos.length()) qoses=TypeUtil.toType(qos,int[].class);
        }

        if(null!=topicsStr) {
            String topic=topicsStr.trim();
            if(0!=topic.length()) topics=TypeUtil.toType(topic,String[].class);
        }

        if(null!=jwtSecretStr) {
            String secret=jwtSecretStr.trim();
            if(0!=secret.length()) jwtSecret=secret;
        }

        if(null!=passWordStr) {
            String pass=passWordStr.trim();
            if(0!=pass.length()) passWord=pass;
        }

        if(null!=userNameStr) {
            String user=userNameStr.trim();
            if(0!=user.length()) userName=user;
        }

        if(null!=tokenFromStr) {
            String from=tokenFromStr.trim();
            if(0!=from.length()) tokenFrom=from;
        }

        if(null!=tokenExpireStr) {
            String expire=tokenExpireStr.trim();
            if(0!=expire.length()) tokenExpire=Integer.parseInt(expire);
        }

        if(null!=expireFactorStr) {
            String factor=expireFactorStr.trim();
            if(0!=factor.length())expireFactor=Integer.parseInt(factor);
```

```
        }
    }
```

2. 直接实现方法
   这是一种约定优于配置的原则，也是值得极力推荐的一种方式，该方式下，我们无需绑定要实现的接口类型，直接按照接口类中定义的方法签名原型编写实现方法即可，插件会自动通过约定的接口原型到你所写的实现类中去查找这些方法，如果找到这些方法，则调用它们；如果没有找到对应的方法则判定查找的方法是否为必须实现的方法，如果是则抛出异常并终止过滤器调用，如果不是则忽略掉该方法的调用

- 基于MongoDB的接口实现

```
import java.util.HashMap;
import java.util.Map;
import java.util.Properties;
import java.util.regex.Pattern;

/**
 * @author Louis(LiXiang)
 * @description 自定义日志过滤器
 */
public class MdbLoggerFilter {
    /**
     * 字段列表
     */
    private String[] fields;

    /**
     * 登录MongoDB用户名
     */
    private static String userName;

    /**
     * 登录MongoDB密码
     */
    private static String passWord;

    /**
     * 文档索引类型
     */
    private String collectionName;

    /**
     * 文档索引名称
     */
    private String dataBaseName;

    /**
     * 日志记录字段分隔符
     */
    private String fieldSeparator;

    /**
     * 逗号正则式
     */
```

```
    private static Pattern commaRegex;

    public String getDocId() {
        return fields[0];
    }

    public String getPassword() {
        return passWord;
    }

    public String getUsername() {
        return userName;
    }

    public String getCollectionName() {
        return collectionName;
    }

    public String getDataBaseName() {
        return dataBaseName;
    }

    public HashMap<String, Object>[] doFilter(String record) {
        String[] fieldValues=commaRegex.split(record);
        HashMap<String,Object> map=new HashMap<String,Object>();
        map.put(fields[0], fieldValues[0].trim());
        map.put(fields[1], fieldValues[1].trim());
        map.put(fields[2], fieldValues[2].trim());
        return new HashMap[]{map};
    }

    public void filterConfig(Properties properties) {
        commaRegex=Pattern.compile(fieldSeparator);
    }
}
```

- 基于Emqx的接口实现

```
import java.util.HashMap;
import java.util.Map;
import java.util.Properties;
import java.util.regex.Pattern;

/**
 * @author Louis(LiXiang)
 * @description 自定义日志过滤器
 */
public class EmqxLoggerFilter {
    /**
     * 通信质量表
     */
    private static int[] qoses;

    /**
     * 连接主题列表
     */
    private static String[] topics;
```

```
    /**
     * 登录Emqx密码
     */
    private static String passWord;

    /**
     * 登录Emqx用户名
     */
    private static String userName;

    /**
     * 登录验证Token的秘钥
     */
    private static String jwtSecret;

    /**
     * 携带Token的字段名
     */
    private static String tokenFrom;

    /**
     * Token过期时间
     */
    private static Integer tokenExpire;

    /**
     * Token过期时间因数
     */
    private static Integer expireFactor;

    public int[] getQoses() {
        return qoses;
    }

    public String[] getTopics() {
        return topics;
    }

    public String getPassword() {
        return passWord;
    }

    public String getUsername() {
        return userName;
    }

    public String getJwtsecret() {
        return jwtSecret;
    }

    public String getTokenfrom() {
        return tokenFrom;
    }

    public Integer getTokenexpire() {
        return tokenExpire;
    }
```

```
    public Integer getExpirefactor() {
        return expireFactor;
    }

    public String[] doFilter(String record) {
        return new String[]{record};
    }

    public void pluginConfig(Map<String, String> config) {
        String qosesStr=config.get("qoses");
        String topicsStr=config.get("topics");
        String jwtSecretStr=config.get("jwtSecret");
        String passWordStr=config.get("passWord");
        String userNameStr=config.get("userName");
        String tokenFromStr=config.get("tokenFrom");
        String tokenExpireStr=config.get("tokenExpire");
        String expireFactorStr=config.get("expireFactor");

        if(null!=qosesStr) {
            String qos=qosesStr.trim();
            if(0!=qos.length()) qoses=TypeUtil.toType(qos,int[].class);
        }

        if(null!=topicsStr) {
            String topic=topicsStr.trim();
            if(0!=topic.length()) topics=TypeUtil.toType(topic,String[].class);
        }

        if(null!=jwtSecretStr) {
            String secret=jwtSecretStr.trim();
            if(0!=secret.length()) jwtSecret=secret;
        }

        if(null!=passWordStr) {
            String pass=passWordStr.trim();
            if(0!=pass.length()) passWord=pass;
        }

        if(null!=userNameStr) {
            String user=userNameStr.trim();
            if(0!=user.length()) userName=user;
        }

        if(null!=tokenFromStr) {
            String from=tokenFromStr.trim();
            if(0!=from.length()) tokenFrom=from;
        }

        if(null!=tokenExpireStr) {
            String expire=tokenExpireStr.trim();
            if(0!=expire.length()) tokenExpire=Integer.parseInt(expire);
        }

        if(null!=expireFactorStr) {
            String factor=expireFactorStr.trim();
            if(0!=factor.length())expireFactor=Integer.parseInt(factor);
        }
```

```
    }
```

1.4、Flume插件过滤器配置

Flume插件过滤器可以读取两个部分的配置，一个是Flume的配置上下文，这个配置文件是在使用flume-ng命令启动Flume例程时通过-f参数指定的，如：

```
[root@CC7 bin]# pwd
/software/flume-1.9.0/bin
[root@CC7 bin]# flume-ng agent -c /software/flume-1.9.0/conf -f /software/flume-1.9.0/process/conf/example.conf -n a1 -Dflume.root.logger=INFO,console
```

上面的example.conf就是Flume启动a1例程所需的配置文件，配置文件中的内容根据flume设计规范进行定义。为便于管理，我们通常将该配置文件放置于flume安装目录下的process/conf目录中（该目录原本不存在，安装Flume之后需要自行创建），该配置文件中的内容通过过滤器接口中的pluginConfig方法注入，为了实现过滤器编程上的解耦，注入的类型是一个Map类型，而不是与具体存储类型发生紧耦合的扩展类型，开发者编程时无需导入第三方包，只需要在有JDK的环境中编写自己的代码即可。
第二个配置文件是放置于flume安装目录下的filter/conf目录中（该目录原本不存在，安装好flume之后需要自行创建）的过滤器配置文件，配置文件的名字由process/conf目录下的example.conf文件中的filterName参数给定，如：

```
[root@CC7 conf]# pwd
/software/flume-1.9.0/process/conf
[root@CC7 conf]# cat example.conf|grep k2.filterName
a1.sinks.k2.filterName=mdbFilter

[root@CC7 conf]# pwd
/software/flume-1.9.0/filter/conf
[root@CC7 conf]# ls
mdbFilter.properties
```

上面的mdbFilter.properties文件名就是在example.conf文件中通过filterName参数来予以指定的，该配置文件中的内容完全由开发者根据自己编写过滤器的需要自行给定，但配置文件的类型必须是属性文件类型（即：*.properties格式或其兼容格式），该配置文件中的内容通过过滤器接口中的filterConfig方法注入，为了实现过滤器编程上的解耦，注入的类型是一个Properties类型，而不是与具体存储类型发生紧耦合的扩展类型，开发者编程时无需导入第三方包，只需要在有JDK的环境中编写自己的代码即可；当然开发者可以视具体情况不需要该配置文件，该配置文件也并不是必须的。
Flume的Sink插件被设计为优先回调开发者定义的pluginConfig接口实现以加载例程配置文件（如：example.conf），然后自动加载过滤器配置文件（如：mdbFilter.properties）以初始化过滤器类中的任何实例成员或静态成员，最后再回调filterConfig接口以加载过滤器配置文件（如：mdbFilter.properties），如果这两个配置文件中定义了同名参数，则参数值将按上述流程依次产生覆盖，开发者可以在上述流程中的任何一个环节读取对应配置文件中的参数值来初始化过滤器类的成员值。
每一个过滤器配置文件中都有一个type参数，该参数用于定义过滤器入口类的全类名（包名+类名），如果没有找到这个type参数，那么插件将启用系统默认的过滤器，通常情况下，插件系统默认的过滤器并不能满足我们的要求，所以我们通常都需要定义好自己的过滤器。

1. 基于MongoDB存储的过滤器配置文件
   在基于MongoDB存储的过滤器配置文件中，除了通常的type参数以外，还有dataBaseName和collectionName参数，这两个参数用于返回MongoDB的数据库名称和集合名称，开发者也可以根据具体需要定义其它参数，一个典型的过滤器配置文件内容如下：

```
[root@CC7 conf]# pwd
/software/flume-1.9.0/filter/conf
[root@CC7 conf]# cat mdbFilter.properties
type=MdbLoggerFilter
dataBaseName=mdbtest
collectionName=logger
fields=docId,level,msg
fieldSeparator=,
```

2. 基于Emqx存储的过滤器配置文件
   在基于Emqx存储的过滤器配置文件中，除了通常的type参数以外，还有topics和qoses参数，这两个参数用于返回Emqx的主题列表和主题列表中各个主题的通信质量列表，开发者也可以根据具体需要定义其它参数，一个典型的过滤器配置文件内容如下：

```
[root@CC7 conf]# pwd
/software/flume-1.9.0/filter/conf
[root@CC7 conf]# cat emqxFilter.properties
type=EmqxLoggerFilter
topics=Test-Topics
```

1.5、Flume插件过滤器编译

编写插件过滤器可以借助于IDE工具（如：Eclipse、IDEA等），也可以直接使用扩展记事本（如：Notepad++、Uedit等），就像编写普通Shell脚本一样，编写过滤器的代码是很少的，所以我们可以使用记事本直接搞定它并非难事，另外需要注意的是不要在过滤器代码中置入任何IO阻塞或线程等待的代码，这将不利于提升日志抽取的性能和效率，切记！
如果你的过滤器代码是显式实现Filter接口的，那么这意味着过滤器代码是编译器绑定的，因此在编译这部分过滤器代码时需要Filter.jar包的支持；否则，我们无需那么麻烦，直接编译即可

1. 显式实现SinkFilter接口的过滤器编译

```
[root@CC7 test1]# pwd
/install/java/test1
[root@CC7 test1]# ls
ElasticLoggerFilter.java
[root@CC7 test1]# wget
https://github.com/lixiang2114/Document/raw/main/plugin/flume1.9/face/FlumePlugi
nFilter.jar
[root@CC7 test1]# ls
MdbLoggerFilter.java  FlumePluginFilter.jar
[root@CC7 test1]# javac -version
javac 1.8.0_271
[root@CC7 test1]# javac -cp FlumePluginFilter.jar MdbLoggerFilter.java
[root@CC7 test1]# ls
MdbLoggerFilter.class  MdbLoggerFilter.java  FlumePluginFilter.jar
```

2. 直接实现PluginFilter接口的过滤器编译

```
[root@CC7 java]# pwd
/install/java
[root@CC7 java]# ls
ElasticLoggerFilter.java
[root@CC7 java]# javac MdbLoggerFilter.java
[root@CC7 java]# ls
MdbLoggerFilter.class  MdbLoggerFilter.java
```

备注：

上面编译的是基于MongoDB存储的过滤器源码，编译基于Emqx中间件的过滤器源码与之类同，这里不再叙述。
如果使用的IDE工具开发的过滤器代码，则可以直接使用IDE工具编译即可，如果过滤器项目是基于Maven构建的，还可以直接使用Maven编译，如果编译后的过滤器类不止一个，则需要将其打成jar包待后续部署

1.6、Flume插件过滤器部署

理论上来讲，只需要将编译后的字节码文件或jar包拷贝到flume安装目录下的filter目录下即可，但为了便于后期维护，我们可以先在filter目录下创建conf和lib两个子目录，conf目录用于存放过滤器所需的配置文件（如果需要），lib目录用于存放过滤器编译后的字节码文件或字节码打包后的jar文件

1. 创建过滤器的结构化目录

```
[root@CC7 filter]# pwd
/software/flume-1.9.0/filter
[root@CC7 filter]# ls
[root@CC7 filter]# mkdir conf lib
```

2. 拷贝过滤器编译结果到目录

```
[root@CC7 filter]# pwd
/software/flume-1.9.0/filter
[root@CC7 filter]# cp -a /install/java/MdbLoggerFilter.class lib/
[root@CC7 filter]# ls lib/
MdbLoggerFilter.class
```

备注：

如果filter、conf、lib等目录不存在则依次创建它们，如果已经存在则可直接使用
上面拷贝的是基于MongoDB存储的过滤器编译结果，拷贝基于Emqx中间件的过滤器编译结果与之类同，这里不再叙述
过滤器部署完之后，就可以依次启动Emqx、MongoDB和Flume例程开始测试编写的过滤器代码了。

1.7、Flume使用中的注意事项

- 文档型数据库不比关系数据库，它们对表连接、分组、聚合、子查询等支持相对较弱，因此我们应该采用逆范式设计表结构，即将尽可能多的相关字段冗余到一个表中去，这样在后续的分析业务操作中不至于产生表连接和分组聚合的诉求。
- MongoDB的优势是可完成大数据集场景下的分组、聚合以及部分表连接等统计类查询（若存在分片表则仅支持以分片表作为左表的连接查询，不支持所有连接查询）；同时也支持数据筛选、排序和分页操作，我们在结合业务需求做MongoDB表设计时，应该充分的考虑到个性化特征。表结构设计的输出结果将用于指导我们如何基于不同的存储编写不同的过滤器

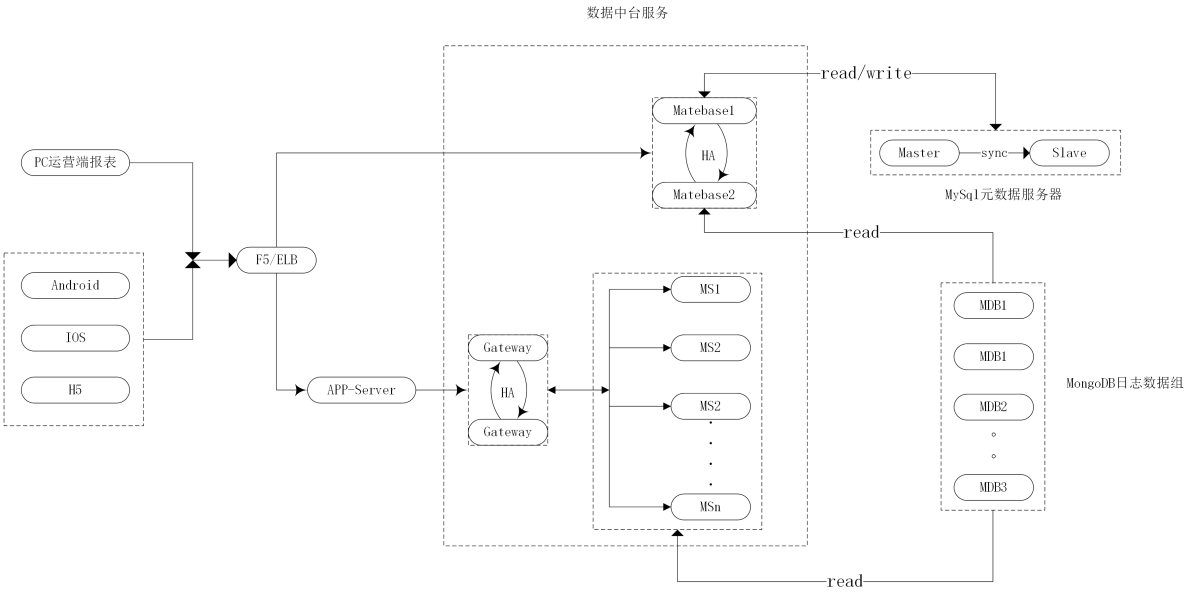
- 抽取流程的最终输出都对应到一个特定的表结构，如果希望将日志数据按某种特定指标进行分类存储到不同的表或数据库，那么我们需要新增更多的表或数据库，这只需要我们在Flume中新增一个或多个抽取流程的配置即可，然而日志分析的最佳实践是将数据尽可能多的冗余并统一抽取到一个大表中，后续再通过离线计算来分表或将其统计计算结果抽取到高效存储中去；所以我们更多期望的是MongoDB用于存储原生的日志数据（因为MongoDB支持大数据集的统计聚合查询）
- 不要针对同一个应用服务在Flume中开启过多的ETL流程，每一个ETL日志抽取流程都将启动一个独立而完整的JVM进程，这将给物理节点带来更多的计算负载和存储负载，同时这也与我们一致倡导“尽可能多的数据冗余抽取到一个表”的理念不相符合。
- 如果我们需要更换过滤器、Sink插件甚至Flume版本，那么我们势必会有带电升级ETL流程的诉求，这可以通过高可用两套ETL流程来实现，先启动新的ETL流程，然后再停掉旧的ETL流程，这样可以做到ETL流程的平滑升级，避免产生日志丢失的情况发生。

二、数据分析平台

数据分析平台是通过对数据源进行读取、筛选、连接、分组、聚合等一系列操作的应用系统，其目的旨在于通过统计手段形成一系列可供上层分析的应用报表

2.1、数据分析平台架构图

本套数据分析平台使用Metabase来实现数据统计、聚合查询以及UI报表展现，数据分析平台架构示意图如下：



说明：

数据中台服务分为两个部分，第一个部分是服务于报表分析类业务的单体应用，这些应用可通过中台服务直接对接到出云网关（处于安全性考虑，这个部分我们后续需要新增一条流线，将线上日志数据复制抽取一份副本到本地机房，以实现报表的本地可视化），第二个部分是服务于业务方的数据API应用，业务方的任何数据请求都可以通过该API应用来获取（这个部分我们后续需要将其构建成SpringCloud架构，中台第一期暂不考虑）

2.2、数据分析应用开发

数据分析类应用的输出结果是分析报表，这些报表服务于公司内部使用，对公司优化战略方向提供决策和指导性。对于公司内部使用的报表类应用可以直接构建为SpringBoot的单体应用，无须构建成分布式应用，这可以使得应用更加的简单和可操作化；如果使用开源的BI产品（如Metabase、Superset等）则可以减少更多的开发工作量

备注：

关于如何在SpringBoot框架构建的报表单体应用中集成ElasticSearch和MongoDB的整合使用，可以参考Spring官网文档，单体应用采用流行的MVC层+持久层的工程结构即可
对于报表的可视化部分，建议自定义前端UI页面，前端可支持H5、Android、IOS等