Flume插件过滤器开发说明书

Flume插件过滤器接口规范

1. 基于Elasticsearch存储的过滤器接口规范

```
package com.bfw.flume.plugin.es.filter;

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

/**
 * @author Louis(LiXiang)
 * @description ElasticSink过滤器接口规范
 */
public interface SinkFilter {
    /**
     * 日志数据过滤之后存储的索引类型(必须重写)
     * @return 索引类型
     */
    public String getIndexType();

    /**
     * 日志数据过滤之后存储的索引名称(必须重写)
     * @return 索引名称
     */
    public String getIndexName();

    /**
     * 日志数据如何过滤之后转换成的文档对象(必须重写)
     * @param record 文本记录
     * @return 文档字典对象
     */
    public Map<String,Object> doFilter(String record);

    /**
     * 文档对象存储时_id字段使用的ID字段名
     * @return ID字段名
     */
    default public String getDocId(){return null;}

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

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

    /**
```

```
     * 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){}

    /**
     * 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 config 配置
     */
    default public void filterConfig(Properties properties){}
}
```

2. 基于MongoDB存储的过滤器接口规范

```
package com.bfw.flume.plugin.mdb.filter;

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

/**
 * @author Louis(LiXiang)
 * @description MongoDBSink过滤器接口规范
 */
public interface SinkFilter {
    /**
     * 日志数据过滤之后存储的数据库名称(必须重写)
     * @return 索引名称
     */
    public String getDataBaseName();

    /**
     * 日志数据过滤之后存储的集合(表)名称(必须重写)
     * @return 索引类型
     */
    public String getCollectionName();

    /**
     * 日志数据如何过滤之后转换成的文档对象(必须重写)
     * @param record 文本记录
     * @return 文档字典对象
     */
    public Map<String,Object> doFilter(String record);

    /**
     * 文档对象存储时_id字段使用的ID字段名
     * @return ID字段名
     */
    default public String getDocId(){return null;}

    /**
     * 登录MongoDB服务所需的密码(如果需要则重写)
     * @return 密码
     */
    default public String getPassword(){return null;}
```

```
    /**
     * 登录MongoDB服务所需的用户名(如果需要则重写)
     * @return 用户名
     */
    default public String getUsername(){return null;}

    /**
     * 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 config 配置
     */
    default public void filterConfig(Properties properties){}

    /**
     * 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:

这两个过滤器接口虽然名字相同，但是所在包名不同，接口的设计规范完全相同，Elasticsearch中的索引名称相当于MongoDB中的数据库名称，Elasticsearch中的索引类型相当于MongoDB中的集合(表)，Elasticsearch中的主键_id相当于MongoDB中的主键ObjectId
编写基于Elasticsearc的过滤器可以实现ElasticSink接口，编写基于MongoDB的过滤器可以实现MongoSink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

Flume插件过滤器接口实现

无论是ElasticSink插件，还是MongoSink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Elasticsearch的接口实现

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

import com.bfw.flume.plugin.es.filter.SinkFilter;

/**
 * @author Louis(LiXiang)
 * @description 自定义日志过滤器
 */
public class ElasticLoggerFilter implements SinkFilter{
    /**
     * 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 indexType;

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

    /**
     * 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 getIndexType() {
        return indexType;
    }

    @Override
    public String getIndexName() {
        return indexName;
    }
```

```
    @Override
    public Map<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 map;
    }

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

- 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;

import com.bfw.flume.plugin.mdb.filter.SinkFilter;

/**
 * @author Louis(LiXiang)
 * @description 自定义日志过滤器
 */
public class MdbLoggerFilter implements SinkFilter{
    /**
     * 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 Map<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 map;
    }

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

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Elasticsearch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 ElasticLoggerFilter {
    /**
     * 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 indexType;

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

    /**
     * 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 getIndexType() {
        return indexType;
    }

    public String getIndexName() {
        return indexName;
    }

    public Map<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 map;
    }

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

- 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 Map<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 map;
    }

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

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=myLogFilter

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

上面的elasticFilter.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：elasticFilter.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：elasticFilter.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Elasticsearch存储的过滤器配置文件
   在基于Elasticsearch存储的过滤器配置文件中，除了通常的type参数以外，还有indexName和indexType参数，这两个参数用于返回Elasticsearch的索引名称和索引类型，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 elasticFilter.properties
type=ElasticLoggerFilter
indexType=mylog
indexName=mytest
fieldSeparator=,
fields=docId,level,msg
```

2. 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=,
```

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SinkFilter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SinkFilter.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/SinkFilter.jar
[root@CC7 test1]# ls
ElasticLoggerFilter.java  SinkFilter.jar
[root@CC7 test1]# javac -version
javac 1.8.0_271
[root@CC7 test1]# javac -cp SinkFilter.jar ElasticLoggerFilter.java
[root@CC7 test1]# ls
ElasticLoggerFilter.class  ElasticLoggerFilter.java  SinkFilter.jar
```

2. 直接实现SinkFilter接口的过滤器编译

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

备注：

上面编译的是基于Elasticsearch存储的过滤器源码，编译基于MongoDB存储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

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/ElasticLoggerFilter.class lib/
[root@CC7 filter]# ls lib/
ElasticLoggerFilter.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Elasticsearch存储的过滤器编译结果，拷贝基于MongoDB存储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Elasticsearch、MongoDB和Flume例程开始测试编写的过滤器代码了。