Fluentd-Configuration

Riku

2021年5月6日

conf配置解释

1. 配置文件的语法

配置文件位置：

```
/etc/td-agent/td-agent.conf
```

指令列表：

1. source
   - 决定输入源
2. match
   - 决定输出目的地
3. filter
   - 决定事件处理管道
4. system
   - 设置系统范围的配置
5. label
   - 作为内部路由对输出和过滤器进行分组
6. @include
   - 包括其他文件

2. source：这些数据从何而来

通过使用源指令选择和配置所需的输入插件，可以启用Fluentd输入源。

Fluentd标准输入插件包括**http**和**forward**。

- http提供一个http端点来接受传入的http消息。
- forward提供一个TCP端点来接受TCP数据包。

当然，这两者也可以同时进行。您可以根据需要添加多个源配置。

```
# Receive events from 24224/tcp
# This is used by log forwarding and the fluent-cat command
<source>
  @type forward
  port 24224
</source>

# http://<ip>:9880/myapp.access?json={"event":"data"}
<source>
  @type http
  port 9880
</source>
```

每个source指令必须包含一个@type参数来指定要使用的输入插件。

3. Routing

source将事件提交给Fluentd路由引擎。

一个事件由三个实体组成:**tag**、**time** 和**record**。

- tag 是一个由点分隔的字符串(例如myapp.access)，用作Fluentd内部路由引擎的方向。
- time 字段由输入插件指定，且必须为Unix时间格式。
- record 是一个JSON对象。

> Fluentd接受所有非句号字符作为标记的一部分。然而，由于标签有时会在不同的上下文中被输出目的地使用(例如表名、数据库名、键名等)，因此强烈建议您坚持使用小写字母、数字和下划线(例如^[a-z0-9_]+$)。

在前面的例子中，HTTP输入插件提交了以下事件:

```
# generated by http://<ip>:9880/myapp.access?json={"event":"data"}
tag: myapp.access
time: (current time)
record: {"event":"data"}
```

4. match:告诉fluentd要做什么!

match指令查找带有匹配**tags**的事件并处理它们。match指令最常见的用法是将事件输出到其他系统。由于这个原因，与match指令对应的插件被称为**输出插件**。Fluentd标准输出插件包括**file**和**forward**。让我们将这些添加到配置文件中。

```
# Receive events from 24224/tcp
# This is used by log forwarding and the fluent-cat command
<source>
  @type forward
  port 24224
</source>

# http://<ip>:9880/myapp.access?json={"event":"data"}
<source>
  @type http
  port 9880
</source>

# Match events tagged with "myapp.access" and
# store them to /var/log/fluent/access.%Y-%m-%d
# Of course, you can control how you partition your data
# with the time_slice_format option.
<match myapp.access>
  @type file
  path /var/log/fluent/access
</match>
```

每个**match**指令必须包含一个匹配模式和一个@type参数。只有带有与模式匹配的**tag**事件才会被发送到输出目的地(在上面的示例中，只有带有标记**myapp.access**的事件才会匹配。**@type**参数指定要使用的输出插件。

5. filter:事件处理管道

filter指令具有与match相同的语法，但是filter可以被链接用于处理管道。使用**filters**，事件流是这样的:

```
Input -> filter 1 -> ... -> filter N -> Output
```

添加标准record_transformer过滤器来match示例。

```
# http://this.host:9880/myapp.access?json={"event":"data"}
<source>
  @type http
  port 9880
</source>

<filter myapp.access>
  @type record_transformer
  <record>
```

```
    host_param "#{Socket.gethostname}"
  </record>
</filter>

<match myapp.access>
  @type file
  path /var/log/fluent/access
</match>
```

Visualizaion可视化：

接收到的事件**{"event":"data"}**首先转到**record_transformer**过滤器。record_transformer过滤器将**host_param**字段添加到事件;

然后过滤事件**{"event":"data"， "host_param":"webserver1"}**转到**file**输出插件。

6. 设置系统范围的配置:system指令

系统范围的配置是由**system**指令设置的。它们中的大多数也可以通过命令行选项获得。例如，有以下配置:

- log_level
- suppress_repeated_stacktrace
- emit_error_log_interval
- suppress_config_dump
- without_source
- process_name（仅在系统指令中可用。没有fluentd选择）

例如：

```
<system>
  # equal to -qq option
  log_level error
  # equal to --without-source option
  without_source
  # ...
</system>
```

具体设置参考：

System Configuration

process_name:

如果设置此参数，fluentd的supervisor和worker进程名称将被更改。

```
<system>
  process_name fluentd1
</system>
```

通过配置，ps命令显示如下结果:

```
% ps aux | grep fluentd1
foo      45673   0.4  0.2  2523252  38620 s001  S+    7:04AM   0:00.44 worker:fluentd1
foo      45647   0.0  0.1  2481260  23700 s001  S+    7:04AM   0:00.40 supervisor:fluentd1
```

7. filter和output形成的组：label指令

label 指令组过滤并输出内部路由。**label** 降低了tag处理的复杂性。

label参数是一个内置插件参数，所以需要@前缀。

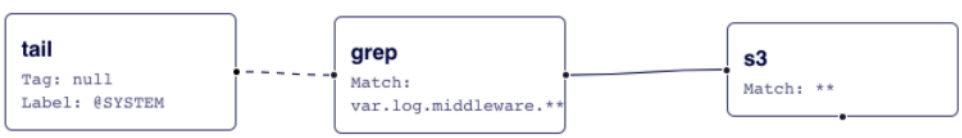
下面是一个配置示例:

```
<source>
  @type forward
</source>

<source>
  @type tail
  @label @SYSTEM
</source>

<filter access.**>
  @type record_transformer
  <record>
    # ...
  </record>
</filter>
<match **>
  @type elasticsearch
  # ...
</match>

<label @SYSTEM>
  <filter var.log.middleware.**>
    @type grep
    # ...
  </filter>
  <match **>
    @type s3
    # ...
  </match>
</label>
```



在此配置中，forward事件被路由到record_transformer filter 然后到 elasticsearch输出，in_tail事件被路由到@SYSTEM label内的grep filter 然后到 s3输出。

label参数对于没有标记前缀的事件流分离非常有用。

@ERROR标签

@ERROR标签是一个内置标签，用于插件的emit_error_event API发出的错误记录。

如果设置了<label @ERROR>，当相关错误被触发时(例如缓冲区已满或记录无效)，事件将被路由到这个标签。

8. 重用你的配置:@include指令

单独的配置文件中的指令可以使用@include指令导入

```
# Include config files in the ./config.d directory
@include config.d/*.conf
```

@include指令支持常规文件路径、**glob模式**和**http URL**约定:

```
# absolute path
@include /path/to/config.conf

# if using a relative path, the directive will use
# the dirname of this config file to expand the path
@include extra.conf

# glob match pattern
@include config.d/*.conf

# http
@include http://example.com/fluent.conf
```

注意，对于glob模式，文件是按字母顺序展开的。如果有a.conf和b.conf，那么fluentd首先解析a.conf。但是，你不应该写依赖于这个顺序的配置。它是如此容易出错，因此，为了安全起见，使用多个独立的@include指令。

```
# If you have a.conf, b.conf, ..., z.conf and a.conf / z.conf are important

# This is bad
@include *.conf

# This is good
@include a.conf
@include config.d/*.conf
@include z.conf
```

共享相同的参数

@include指令可以在section下使用，以共享相同的参数:

```
# config file
<match pattern>
  @type forward
  # ...
  <buffer>
    @type file
    path /path/to/buffer/forward
    @include /path/to/out_buf_params.conf
  </buffer>
```

```
</match>

<match pattern>
  @type elasticsearch
  # ...
  <buffer>
    @type file
    path /path/to/buffer/es
    @include /path/to/out_buf_params.conf
  </buffer>
</match>

# /path/to/out_buf_params.conf
flush_interval    5s
total_limit_size  100m
chunk_limit_size  1m
```

10. 匹配模式是如何工作的?

如上所述，Fluentd允许根据事件的标记路由事件。虽然您可以指定要匹配的确切标记(如)，但您可以使用许多技术来更有效地管理数据流。

通配符、扩展和其他提示

下面的匹配模式可以用于和标签:

- *匹配单个标记部分。
  - 例如，模式a.*匹配a.b，但不匹配a或a.b.c
- **匹配零个或多个标记部分。
  - 例如，模式a.**匹配a, a.b, a.b.c
- {X,Y,Z}匹配X,Y或Z，其中X,Y和Z是匹配模式。
  - 例如，模式{a,b}匹配a和b，但不匹配c
  - 这可以与*或**模式结合使用。例子:包括a.{b,c}.* 和a.{b,c.**}
- /regular expression/用于复杂模式
  - 例如，模式 `/(?!a\.).*` 匹配非a.开头的标签,比如b.xxx
  - 从fluentd v1.11.2开始就支持该特性
- #{…}以Ruby表达式的形式计算括号内的字符串。(参见下面嵌入Ruby表达式一节)。
- 当多个模式被列在单个**tag** (由一个或多个空白分隔)内时，它将匹配所列的任何模式。例如:
  - 模式 匹配a和b。
  - <match a.** b.*> match a, a.b, a.b.c(第一个模式)和b.d(第二个模式)

11. 注意匹配顺序

Fluentd试图按照标记在配置文件中出现的顺序匹配它们。所以，如果你有以下配置:

```
# ** matches all tags. Bad :(
<match **>
  @type blackhole_plugin
</match>

<match myapp.access>
  @type file
  path /var/log/fluent/access
</match>
```

然后myapp.access从不匹配。更广泛的匹配模式应该定义再严格匹配之后。

```
<match myapp.access>
  @type file
  path /var/log/fluent/access
</match>

# Capture all unmatched tags. Good :)
<match **>
  @type blackhole_plugin
</match>
```

当然，如果您使用两个相同的模式，第二个匹配将永远不会被匹配。如果你想把事件发送到多个输出，考虑out_copy插件。

常见的陷阱是当你在后面放一个块时。由于上述原因，它永远不会工作，因为事件永远不会通过过滤器。

```
# You should NOT put this <filter> block after the <match> block below.
# If you do, Fluentd will just emit events without applying the filter.

<filter myapp.access>
  @type record_transformer
  ...
</filter>

<match myapp.access>
  @type file
  path /var/log/fluent/access
</match>
```

12. 嵌入Ruby表达式

由于Fluentd v1.4.0，您可以使用#{...}来将任意的Ruby代码嵌入匹配模式中。下面是一个例子:

```
<match "app.#{ENV['FLUENTD_TAG']}">
  @type stdout
</match>
```

如果将环境变量FLUENTD_TAG设置为dev，则计算结果为app.dev。

13. Values的支持数据类型

每个Fluentd插件都有自己特定的参数集。例如，in_tail有**rotate_wait**和**pos_file**等参数。每个参数都有一个与之相关联的特定类型。类型定义如下:

- 字符串:该字段被解析为一个字符串。这是最通用的类型，每个插件决定如何处理字符串。
  - 字符串有三个字面值:不带引号的一行字符串，`'`单引号字符串和`"`双引号字符串。
- Integer:解析为整数。
- float:字段被解析为浮点型。
- Size:该字段被解析为字节数。有几个符号的变化:

```
<INTEGER>k or <INTEGER>K: number of kilobytes
<INTEGER>m or <INTEGER>M: number of megabytes
<INTEGER>g or <INTEGER>G: number of gigabytes
<INTEGER>t or <INTEGER>T: number of terabytes
# 否则，该字段被解析为整数，该整数是字节数。
```

- Time:该字段被解析为一个时间持续时间。

```
<INTEGER>s: seconds
<INTEGER>m: minutes
<INTEGER>h: hours
<INTEGER>d: days
# 否则，该字段被解析为float，该float是秒数。此选项用于指定次秒持续时间，如0.1(0.1秒= 100毫秒)。
```

- array:该字段被解析为JSON数组。它还支持速记语法。这些是相同的值:

```
normal: ["key1", "key2"]
shorthand: key1,key2
```

- hash:该字段被解析为一个JSON对象。它还支持速记语法。这些是相同的值:

```
normal: {"key1": "value1", "key2": "value2"}
shorthand: key1:value1,key2:value2
```

array和**hash**类型是JSON，因为几乎所有编程语言和基础设施工具都可以轻松生成JSON值，而不是其他任何不寻常的格式。

14. 常见的插件参数

这些参数被保留，并以@符号作为前缀:

- @type:插件类型
- @id:插件id. `In_monitor_agent` 将此值用于plugin_id领域
- @label:标签符号。
- @log_level:每个插件的日志级别

15. 检查配置文件

配置文件可以在不启动插件的情况下使用——dry-run选项进行验证:

```
fluentd --dry-run -c fluent.conf
```

16. 介绍配置文件的一些有用特性

多行支持“引用字符串、数组和散列值”

```
str_param "foo  # Converts to "foo\nbar". NL is kept in the parameter
bar"
array_param [
  "a", "b"
]
hash_param {
  "k": "v",
  "k1": 10
}
```

fluentd假定[或{开头的为数组或哈希，因此，如果想设置[或{开头,但不是json的格式，请使用`'`,`“`

比如：

mail plugin

```
<match **>
  @type mail
  subject "[CRITICAL] foo's alert system"
</match>
```

map plugin

```
<match tag>
  @type map
  map '[["code." + tag, time, { "code" => record["code"].to_i}], ["time." + tag, time, { "time" => record["time"].to_i}]]'
  multi true
</match>
```

这个限制将随着配置解析器的改进而消除。

嵌入**Ruby**代码

你可以用"引号字符串"中的#{}来计算Ruby代码。这对于设置机器信息很有用，例如主机名。

```
host_param  "#{Socket.gethostname}" # host_param is actual hostname like `webserver1`.
env_param   "foo-#{ENV["FOO_BAR"]}" # NOTE that foo-"#{ENV["FOO_BAR"]}" doesn't work.
```

从v1.1.0开始，主机名和worker_id快捷方式可用:

```
host_param  "#{hostname}"  # This is same with Socket.gethostname
@id         "out_foo#{worker_id}" # This is same with ENV["SERVERENGINE_WORKER_ID"]
```

worker_id快捷方式在多个worker下是有用的。例如，对于单独的插件id，添加worker_id来存储s3中的路径，以避免文件冲突。

从1.8.0版本开始，helper方法use_nil和use_default是可用的:

```
some_param  "#{ENV["FOOBAR"] || use_nil}"     # Replace with nil if ENV["FOOBAR"] isn't set
some_param  "#{ENV["FOOBAR"] || use_default}" # Replace with the default value if ENV["FOOBAR"] isn't set
```

注意，这些方法不仅会将内嵌的Ruby代码替换掉，还会将整个字符串替换为nil或默认值。

```
some_path   "#{use_nil}/some/path" # some_path is nil, not "/some/path"
```

config-xxx mixins使用“${}”，而不是“#{}”。这些嵌入式配置是两个不同的东西。

在双引号字符串字面量中，\是转义字符

正斜杠\被解释为转义字符。你需要“设置”,“r”,“n”,“t”或一些双引号字符串字面量中的字符。

```
str_param   "foo\nbar" # \n is interpreted as actual LF character
```

Routing Examples

本文展示了典型路由场景的配置示例。

简单：Input -> Filter -> Output

```
<source>
  @type forward  #(收集tcp数据包)
</source>

<filter app.**>
  @type record_transformer
  <record>
    hostname "#{Socket.gethostname}"
  </record>
</filter>

<match app.**>
  @type file
  # ...
</match>
```

两个输入：forward and tail

```
<source>
  @type forward
</source>

<source>
  @type tail
  tag system.logs
  # ...
</source>

<filter app.**>
  @type record_transformer
  <record>
    hostname "#{Socket.gethostname}"
  </record>
</filter>

<match {app.**,system.logs}>
  @type file
  # ...
</match>
```

如果要分隔每个源的数据管道，请使用Label。

带Label: Input -> Filter -> Output

标签通过分离数据管道减少了复杂的标签处理。

```
<source>
  @type forward
</source>

<source>
  @type dstat
  @label @METRICS # dstat events are routed to <label @METRICS>
  # ...
</source>

<filter app.**>
  @type record_transformer
```

```
  <record>
    # ...
  </record>
</filter>

<match app.**>
  @type file
  # ...
</match>

<label @METRICS>
  <match **>
    @type elasticsearch
    # ...
  </match>
</label>
```

通过Tag重新路由事件

使用fluent-plugin-route插件。这个插件重写标签并重新发出事件到其他match或Label。

```
<match worker.**>
  @type route
  remove_tag_prefix worker
  add_tag_prefix metrics.event

  <route **>
    copy # For fall-through. Without copy, routing is stopped here.
  </route>
  <route **>
    copy
    @label @BACKUP
  </route>
</match>

<match metrics.event.**>
  @type stdout
</match>

<label @BACKUP>
  <match metrics.event.**>
    @type file
    path /var/log/fluent/backup
  </match>
</label>
```

根据记录内容重新路由事件

使用 fluent-plugin-rewrite-tag-filter

```
<source>
  @type forward
</source>

# event example: app.logs {"message":"[info]: ..."}
<match app.**>
  @type rewrite_tag_filter
  <rule>
    key message
    pattern ^\[(\w+)\]
    tag $1.${tag}
```

```
  </rule>
  # more rules
</match>

# send mail when receives alert level logs
<match alert.app.**>
  @type mail
  # ...
</match>

# other logs are stored into a file
<match *.app.**>
  @type file
  # ...
</match>
```

See also: out_rewrite_tag_filter

重新路由事件到其他标签

使用**out_relabel**插件。这个插件只是向Label发送事件，而不需要重写tag

```
<source>
  @type forward
</source>

<match app.**>
  @type copy
  <store>
    @type forward
    # ...
  </store>
  <store>
    @type relabel
    @label @NOTIFICATION
  </store>
</match>

<label @NOTIFICATION>
  <filter app.**>
    @type grep
    regexp1 message ERROR
  </filter>

  <match app.**>
    @type mail
  </match>
</label>
```

配置:常见的参数

一些常用参数可用于所有或部分Fluentd插件。本页面描述这些参数。

所有插件的参数

`@type`

@type参数指定插件的类型。

```
<source>
  @type my_plugin_type
</source>

<filter>
  @type my_filter
</filter>
```

`@id`

@id参数为配置指定一个唯一的名称。它被用作缓冲区、存储、日志记录和其他用途的路径。

```
<match>
  @type file
  @id service_www_accesslog
  path /path/to/my/access.log
  # ...
</match>
```

应该为所有插件指定这个参数，以便全局启用root_dir和workers特性。

See also: System Configuration

`@log_level`

此参数指定插件特定的日志级别。默认日志级别为info。全局日志级别可以通过在 section 中设置log_level或使用-v/-q命令行参数来指定。@log_level参数只覆盖指定插件实例的日志级别。

```
<system>
  log_level info
</system>

<source>
  # ...
  @log_level debug # shows debug log only for this plugin
</source>
```

该参数的主要用途是:

1. 为插件抑制过多的日志;
2. 显示调试日志以帮助调试过程;

Please see the logging article for further details.

触发事件的插件参数

`@label`

@label参数将输入事件路由到部分，即和一组的子部分在下的集合。

```
<source>
  @type ...
  @label @access_logs
  # ...
</source>

<source>
  @type ...
  @label @system_metrics
  # ...
</source>
```

```
12
13 <label @access_log>
14   <match **>
15     @type file
16     path ...
17   </match>
18 </label>
19
20 <label @system_metrics>
21   <match **>
22     @type file
23     path ...
24   </match>
25 </label>
```

注意:@label参数的值必须以@字符开头。

强烈建议使用@label将事件路由到任何插件，而无需修改标签。它有助于使复杂的配置模块化和简单化。

配置:解析部分

一些Fluentd插件支持部分来指定如何解析原始数据。

解析部分概述

parse 语句可以位于， 或 语句下。它为支持解析器插件特性的插件启用。

```
1 <source>
2   @type tail
3   # ...
4   <parse>
5     # ...
6   </parse>
7 </source>
```

解析器插件类型

部分的@type参数指定了解析器插件的类型。Fluentd核心捆绑了一些有用的解析器插件。

```
1 <parse>
2   @type apache2
3 </parse>
```

参数

`@type`

@type参数指定解析器插件的类型。

```
1 <parse>
2   @type regexp
3   # ...
4 </parse>
```

以下是内置解析器插件的列表:

- `regexp`
- `apache2`
- `apache_error`
- `nginx`
- `syslog`
- `csv`
- `tsv`
- `ltsv`
- `json`
- `multiline`
- `none`

解析参数

以下参数的默认值将被各个解析器插件覆盖:

- `types` (hash)(可选):指定将字段转换为另一个字段的类型
  - Default: `nil`
  - 基于字符串的哈希: `field1:type, field2:type, field3:type:option, field4:type:option`
  - JSON格式: `{"field1":"type", "field2":"type", "field3":"type:option", "field4":"type:option"}`
  - example: types user_id:integer,paid:bool,paid_usd_amount:float
- `time_key` (string)(可选):指定事件时间字段。如果事件没有此字段，使用当前时间。
  - Default： `nil`
  - 注意: json,ltsv和regexp覆盖此值的默认值参数，并默认设置为time
- `null_value_pattern` (string)(可选):指定空值模式。
  - Default: `nil`
- `null_empty_string` (bool)(可选):如果为真，则空字符串字段被替换为nil
  - default: false
- `estimate_current_event` (bool)(可选): 当time_key被指定时，如果为true，使用Fluent::EventTime.now(current time)作为时间戳
  - Default: true
- `keep_time_key` (bool)(可选):如果为true，则keep time字段在record里。
  - Default:false
- `timeout` (time)(可选):指定解析处理超时。这主要用于检测错误的regexp模式。
  - Default：nil

types 参数

对于types参数，支持以下类型:

- string: 将字段转换为字符串类型。这使用to_s方法进行转换。
- bool: 将字符串"true"， "yes"或"1"转换为true。否则,false。
- integer: (not int):将字段转换为Integer类型。这使用to_i方法进行转换。例如，字符串“1000”可以转换为1000。
- float: 将字段转换为Float类型。这使用to_f方法进行转换。例如，字符串"7.45"可以转换成7.45。
- time: 将字段转换为Fluent::EventTime类型。它使用Fluentd时间解析器进行转换。对于time类型，第三个字段指定的时间格式类似于time_format。

```
date:time:%d/%b/%Y:%H:%M:%S %z # for string with time format
date:time:unixtime             # for integer time
date:time:float                # for float time
```

- array: 将字符串字段转换为Array类型。对于Array类型，第三个字段指定分隔符(默认为逗号“，”)。例如，如果字段item_ids包含值“3,4,5”，则将item_ids:array解析为["3"，"4"，"5"]。或者，如果值为"Adam|Alice|Bob"，则types item_ids:array:| 将其解析为["Adam"， "Alice"， "Bob"]。

Time 参数

- time_type: (enum)(可选):根据此类型解析/格式化该值
  - Default: `float`
  - Available values: `float` , `unixtime` , `string` , `mixed`
- time_format：(string)(可选):根据指定的格式处理值。只有time_type为字符串时才可用。
  - Default: `nil`
- localtime： (bool)(可选):如果为true，则使用本地时间。否则,使用UTC。这与utc是独占的。
- utc： 如果为true，使用UTC。否则,使用local time。这与本地时间是独占的。
- timezone： (string)(可选):使用指定的时区。一个人可以将时间值解析为指定的时区格式。
  - Default: `nil`
- time_format_fallbacks： (可选):以指定的顺序使用指定的时间格式作为回退。您可以使用time_format_fallbacks解析未确定的时间格式。当混合使用time_type时启用此选项。
  - Default: `nil`

```
time_type mixed
time_format unixtime
time_format_fallbacks %iso8601
```

在上面的用例中，时间戳首先被解析为**unixtime**，如果它失败，那么它被解析为**%iso8601**备用。注意，time_format_fallbacks是解析混合时间戳格式的最后一种方法。这会造成性能损失(通常，在time_format_fallbacks中指定了N个回退，如果使用最后指定的格式作为回退，在最坏的情况下会慢N倍)。