阿里云开发者社区

日志服务数据加工系列培训

主题：扫平日志分析路上障碍，实时海量日志加工实践培训

讲师：丁来强（成喆）- 阿里高级技术专家 | 唐恺(风毅) - 阿里技术专家

分享介绍

8月7日 19:30-20:30	8月8日 19:30-20:30	8月13日 19:30-20:30	8月14日 19:30-20:30	8月20日 19:30-20:30	8月21日 19:30-20:30	8月28日 19:30-20:30	8月29日 19:30-20:30
数据加工 介绍与实战	数据加工DSL 核心语法介绍	数据加工DSL 语法实践	数据加工动态 数据分发汇集实践	非结构化数据 解析实践	结构化数据 解析实践	数据映射 富化实践	数据加工 可靠性与排错实践

数据加工: DSL语法实战

系列培训三

丁来强（成喆）

议题

- 数据加工通用机制
  - 字段提取模式
  - 正则表达式
  - GROK模式
  - JMES语法
- DSL语法最佳实践
  - 搜索字符串（补充）
  - 函数调用最佳实践
  - 事件判断最佳实践
  - 日期时间处理最佳实践

字段提取设置模式

一个例子

- 从request_uri提取动态字段时，如果order_id字段已经存在，是什么行为？

```
e_kv("request_uri")

request_uri: /order?order_id=100&uid=200

# 目标字段不存在时或为空串时添加
e_kv("request_uri", mode="fill-auto")

# 总是覆盖
e_kv("request_uri", mode="overwrite")
```

提取参数mode的可能值

模式	意义
fill	当目标字段不存在或者值为空时, 设置目标字段
fill-auto	**当新值非空,** 且目标字段不存在或者值为空时, 设置目标字段
add	当目标字段不存在时, 设置目标字段
add-auto	**当新值非空,** 且目标字段不存在时, 设置目标字段
overwrite	总是设置目标字段
overwrite-auto	**当新值非空,** 设置目标字段

所有字段设置类函数都有mode参数

类型	函数	说明	mode默认值
字段值赋值	**e_set**	**赋值**	**overwrite**
字段值提取	e_regex	正则提取	fill-auto
	e_json	json展开或提取	fill-auto
	e_kv	**自动提取键值对**	**fill-auto**
	e_kv_delimit	基于分隔符提取键值对	fill-auto
	e_csv	逗号或其他分隔符提取	fill-auto
	e_tsv	tab分隔符提取	fill-auto
	e_psv	pipe分隔符提取	fill-auto
	e_syslogrfc	根据syslog协议由已知priority值计算facility和severity，并且匹配相应的level信息	overwrite
字段富化	e_dict_map	字典映射	fill-auto
	e_table_map	表格映射	fill-auto
	e_search_map	搜索映射	fill-auto
	e_search_dict_map	搜索映射	overwrite
	e_search_table_map	搜索映射	fill-auto

提取字段名约束

- 提取的字段名必须满足字符条件，否则会被丢弃
- 正则规则：

```
u'_*[\u4e00-\u9fa5\u0800-\u4e00a-zA-Z][\u4e00-\u9fa5\u0800-\u4e00\\w\\.\\-]*'
```

  - 字符集：中文、字母、数字、_、-、.
  - 可以下划线开头
  - 非_的第一个字符必须是中文或字母

```
# 保留
_test_
字段名
字段名12
field1
字_段.名
a_b-c.d123

# 丢弃
__1__
1abc
1中文
a@b
```

正则表达式相关

一个例子

- 丢弃字段的字符串是的正则是如何匹配的?

```
# 有字段:
test1
test2
test123

# 会删除哪个字段?
e_drop_fields("test1")
```

相关函数

类型	函数	功能	匹配方式
全局操作函数	e_regex	使用正则从从字段值中提取值	部分
	e_keep_fields	使用正则匹配字段名	完全
	e_drop_fields	使用正则匹配字段名	完全
	e_rename	使用正则匹配字段名	完全
	e_kv	使用正则提取关键字与值	部分
	e_search_dict_map	关键字是搜索字符串, 支持正则	部分
	e_search_table_map	表格字段是搜索字符串, 支持正则	部分
表达式函数	e_match	使用正则匹配值	参数控制, 默认完全
	e_search	接受搜索字符串, 支持正则	部分
	regex_select	使用正则从值中提取值	部分
	regex_findall	使用正则搜索匹配值	部分
	regex_match	使用正则匹配值	参数控制, 默认部分
	regex_replace	对值正则替换值	部分
	regex_split	使用正则做分隔符	部分

正则转换

- 部分变完全：^正则$
- 完全变部分：.*正则.*

```
reg_match("abc123", r"\d+")                  # 会匹配(默认不完全)
reg_match("abc123", r"\d+", full=True)       # 不会匹配(打开完全参数)
reg_match("abc123", r"^\d+$")                # 不会匹配 (正则标记前轴匹配)
e_search(r'status~="\d+"')                   # 部分匹配
e_search(r'status~="^\d+$"')                 # 完全匹配
```

- 正则字符串变普遍字符串：

```
# 丢弃字段名符合abc?test的字段, 其中?表示任意字符
e_drop_fields("abc.test")

# 丢弃字段名为abc.test的字段
e_drop_fields(str_regex_escape("abc.test")
```

GROK模式

一个例子

- 判断content中是否包含合法的IPv4地址，正确吗？

```
e_search(r'content~="\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}"')
```

- IP：333.333.999.999 # 也会匹配
- 准确的正则表达式：

```
e_search(r'conetnt~="(?<![0-9])(?:(?:[0-1]?[0-9]{1,2}|2[0-4]'\
         '[0-9]|25[0-5])[.](?:[0-1]?[0-9]{1,2}|2[0-4][0-9]|'\
         '25[0-5])[.](?:[0-1]?[0-9]{1,2}|2[0-4][0-9]|25[0-5])'\
         '[.](?:[0-1]?[0-9]{1,2}|2[0-4][0-9]|25[0-5]))(?![0-9])"')
```

使用GROK简化

- 最简单的方法:

```
e_search(grok(r'client_ip~="%{IPV4}"'))
```

- 如果要提取字段

```
# . 匹配任意字符
e_regex("content", grok(r"%{IPV4:cip} log.in %{IPV4:sip}"))

# . 仅仅匹配.
e_regex("content", grok(r"%{IPV4:cip} log.in %{IPV4:sip}", escape=True))
```

模式形式

- 替换其中GROK模式，返回一个新的正则表达式

```
grok(pattern, escape=False, extend=None)
```

```
grok("...%{模式}...")
grok("...%{模式:捕获名}...")

grok("...%{模式}...", escape=True)
grok("...%{模式:捕获名}...", escape=True)
```

GROK模式 – 通用

- USERNAME
- USER
- EMAILLOCALPART
- EMAILADDRESS
- HTTPDUSER
- INT
- BASE10NUM
- NUMBER
- BASE16NUM
- BASE16FLOAT
- POSINT
- NONNEGINT
- WORD
- NOTSPACE
- SPACE
- DATA
- GREEDYDATA
- QUOTEDSTRING
- UUID
- MAC
- CISCOMAC
- WINDOWSMAC
- COMMONMAC
- IPV6
- IPV4
- IP
- HOSTNAME
- IPORHOST
- HOSTPORT
- PATH
- UNIXPATH
- WINPATH
- URIPROTO
- TTY
- URIHOST
- URIPATH
- URIPARAM
- URIPATHPARAM
- URI
- MONTH
- MONTHNUM
- MONTHDAY
- DAY
- YEAR
- HOUR
- MINUTE
- SECOND
- TIME
- DATE_US
- DATE_EU
- ISO8601_TIMEZONE
- ISO8601_SECOND
- TIMESTAMP_ISO8601
- DATE
- DATESTAMP
- TZ
- DATESTAMP_RFC822
- DATESTAMP_RFC2822
- DATESTAMP_OTHER
- DATESTAMP_EVENTLOG
- HTTPDERROR_DATE
- SYSLOGTIMESTAMP
- PROG
- SYSLOGPROG
- SYSLOGHOST
- SYSLOGFACILITY
- HTTPDATE
- SYSLOGBASE
- COMMONAPACHELOG
- COMBINEDAPACHELOG
- HTTPD20_ERRORLOG
- HTTPD24_ERRORLOG
- HTTPD_ERRORLOG
- LOGLEVEL

GROK模式- 高级

grok模式 - 插件

- aws
- bacula
- bro
- exim
- firewalls
- haproxy
- java
- junos
- linux-syslog
- mcollective-patterns
- mongodb
- nagios
- postgresql
- rails
- redis
- ruby

模式	规则	说明
COMMONAPACHELOG	%{IPORHOST:clientip} %{HTTPDUSER:ident} %{USER:auth} \[%{HTTPDATE:timestamp}\] "(?:%{WORD:verb} %{NOTSPACE:request}(?: HTTP/%{NUMBER:httpversion})?|%{DATA:rawrequest})" %{NUMBER:response} (?:%{NUMBER:bytes}|-)	解析出clientip、ident、auth、timestamp、verb、request、httpversion、response、bytes字段内容
COMBINEDAPACHELOG	%{COMMONAPACHELOG} %{QS:referrer} %{QS:agent}	解析出上一行中所有字段，另外还解析出referrer、agent字段

JMES语法

相关函数

- 使用JMES提取值后操作（表达式传递、展开或提取）

```
# 全局操作函数: 提取值后展开
e_json(字段名, jmes="jmes表达式", ...)

# 全局操作函数: 提取值后分裂
e_split(字段名, ... jmes="jmes表达式", ...)

# 表达式函数: 提取值
json_select(值, "jmes表达式", ...)
```

常见语法（1）

原始日志

```
"data":{"a":
          {"b":
            {"c":
              {"d":"value"}
          },
          "arr": ["s", "h", "a",
                  "n", "g"],
          "cities1": [
            { "name": "sh", "pop": 2000 },
            { "name": "nj", "pop": 800 }
          ],
          "cities2": {
            "sh": { "prov": "sh", "pop": 2000 },
            "nj": { "prov": "js", "pop": 800 }
          }
        }
    }
```

```
层级提取: 返回 "value"
json_select(v("data"), "a.b.c.d")

数组提取: 返回 "h"
json_select(v("data"), "a.arr[1]")

数组切片: 返回 ["a", "n", "g"]
json_select(v("data"), "a.arr[2:]")
```

常见语法（2）

原始日志

```
"data":{"a":
          {"b":
            {"c":
              {"d":"value"}
          },
          "arr": ["s", "h", "a",
                  "n", "g"],
          "cities1": [
            { "name": "sh", "pop": 2000 },
            { "name": "nj", "pop": 800 }
          ],
          "cities2": {
            "sh": { "prov": "sh", "pop": 2000 },
            "nj": { "prov": "js", "pop": 800 }
          }
        }
    }
```

```
投影获取值-数组: 返回 ["sh", "nj"]
json_select(v("data"), "a.cities1[*].name")

投影获取值-对象: 返回 [2000, 800]
json_select(v("data"), "a.cities2.*.pop")

投影获取值-对象: 返回 [2000]
json_select(v("data"), "a.cities1[?name='sh'].pop")

计算数组长度: 返回 2
json_select(v("data"), "length(a.cities1)")
```

搜索字符串语法补充

字段判断

样例	场景
e_search("field: *")	字段存在
e_search("not field:*")	字段不存在
e_search('not field:""')	字段不存在
e_search('field: "?"')	字段存在, 且值不为空
e_search('field==""')	字段存在, 且值为空
e_search('field~=".+"')	字段存在, 值不为空
e_search('not field~=".+"')	字段不存在或值为空
e_search('not field==""')	字段不存在或值不为空

比较区别

```
f1: "abc xyz"
表示：字段f1里面搜索子串"abc xyz"

f1: (abc xyz)
f1: abc or f1: xyz  # 等价于
表示：字段f1里面搜索abc或xyz

f1: (abc and xyz)
f1: abc and f1: xyz # 等价于
表示：字段f1里面搜索abc且xyz
```

```
f1: abc and xyz
(f1: abc) and (xyz)  # 等价于
表示：字段f1里面搜索"abc"，且全文搜索xyz

f1: abc xyz
f1: abc or xyz       # 等价于
(f1: abc) or (xyz)   # 等价于
表示：字段f1搜索abc，或全文搜索xyz
```

函数调用最佳实践

#1 理解e_keep/KEEP的应用场景

- 无操作下事件默认保留并且最终被输出

```
e_keep(e_search(...) )                          满足保留，不满足丢弃
e_drop(e_search(...) )                          满足丢弃，不满足保留
e_if_else(e_search("..."), KEEP, DROP)          满足保留，不满足丢弃
e_if(e_search("not ..."), DROP)                 满足丢弃，不满足保留(默认)
e_if(e_search("..."), KEEP)                     满足保留(无意义代码)
```

#2 尽可能使用函数自身提供的功能

- 当原字段不存在或者为空时，为字段赋值

```
不推荐做法
e_if(op_not(v("result")), e_set("result", ".....value..."))

正确做法
e_set("result", ".....value...", mode="fill")
```

#3 使用e_compose减少重复判断

- 对某个条件的事件做一系列操作:

```
e_if(e_search("content==123"), e_drop_fields("age|name"))
e_if(e_search("content==123"), e_set("type", "__basic__"))
e_if(e_search("content==123"), e_rename("content", "ctx"))
```

- 可以合并为一个步骤, 效率更高:

```
e_if(e_search("content==123"),
     e_compose(e_drop_fields("age|name"),
               e_set("type", "__basic__")
               e_rename("content", "ctx")))
```

#4 注意表达式函数的参数类型

- op_add支持多种同类型值相加

```
e_set("a", 1)
e_set("b", 2)

op_add(v("a"), v("b"))    返回值为 "12"
op_add(ct_int(v("a")), ct_int(v("b"))) 返回值为 3
```

- op_mul支持多种类型值乘以数值

```
e_set("a", 2)
e_set("b", 5)

op_mul(v("a"), v("b"))                    非法
op_mul(ct_int(v("a")), ct_int(v("b")))    合法, 返回值为10
op_mul(v("a"), ct_int(v("b")))            合法, 返回值为22222
```

#5 注意表达式函数的异常处理情况

- 错误调用: 但data字段不存在时, 会报错

```
e_set("data_len": op_len(v("data")))
```

- 正确调用: 考虑异常, 传入默认值

```
e_set("data_len": op_len(v("data", default="")))
```

#6 理解e_if与e_switch的区别

- 语法形式

```
e_if(条件1, 操作1, 条件2, 操作2, 条件3, 操作3, ....)

e_switch(条件1, 操作1, 条件2, 操作2, 条件3, 操作3, ...., default=None)
```

#6 理解e_if与e_switch的区别 – e_if

- 原始日志

```
status1: 200
status2: 404
```

- 加工

```
e_if(e_match("status1", "200"), e_set("status1_info", "normal"),
    e_match("status2", "404"), e_set("status2_info", "error"))
```

- 新日志

```
status1: 200
status2: 404
status1_info: normal
status2_info: error
```

#6 理解e_if与e_switch的区别 – e_switch

- 原始日志

```
status1: 200
status2: 404
```

- 加工

```
e_switch(e_match("status1", "200"), e_set("status1_info", "normal"),
         e_match("status2", "404"), e_set("status2_info", "error"))
```

- 新日志

```
status1: 200
status2: 404
status1_info: normal
```

事件判断最佳实践

相关事件类函数

函数	说明
v	获取存在字段的值
e_has	判断字段是否存在
e_not_has	判断字段不存在
e_search	提供一种简化的搜索方式
e_match	是用于判断当前事件字段的值是否满足特定条件的表达式
e_match_all	是用于判断当前事件字段的值是否满足特定条件的表达式
e_match_any	是用于判断当前事件字段的值是否满足特定条件的表达式

相关逻辑类函数

函数	说明
op_and	and逻辑条件
op_or	or逻辑条件
op_not	not逻辑条件
op_nullif	如果表达式1等于表达式2，返回None，否则返回表达式1
op_ifnull	返回第一个值不为None的表达式的值
op_coalesce	返回第一个值不为None的表达式的值

#1 判断字段是否存在

- 前者更直观一些:

```
字段存在:
e_has("a")
e_search("a: *")

字段不存在:
e_not_has("a")
e_search("not a: *")
```

#2 判断字段值存在且不为空

```
推荐方法:
e_if(v("a"), ...)

其他方法:
e_if(e_search('a: "?"'), ...)
e_if(e_search('a~=".+"'), ...)
e_if(e_search('a: * and not a=""'), ...)
```

#3 判断字段值存在但为空

```
推荐方案:
e_if(e_search('a==""'), ...)

其他方案:
e_if(op_and(e_has("a"), op_not(v("a"))), ...)

错误方案:
e_if(op_not(v("a")), ...)
e_if(e_search('a:""'), ...)
```

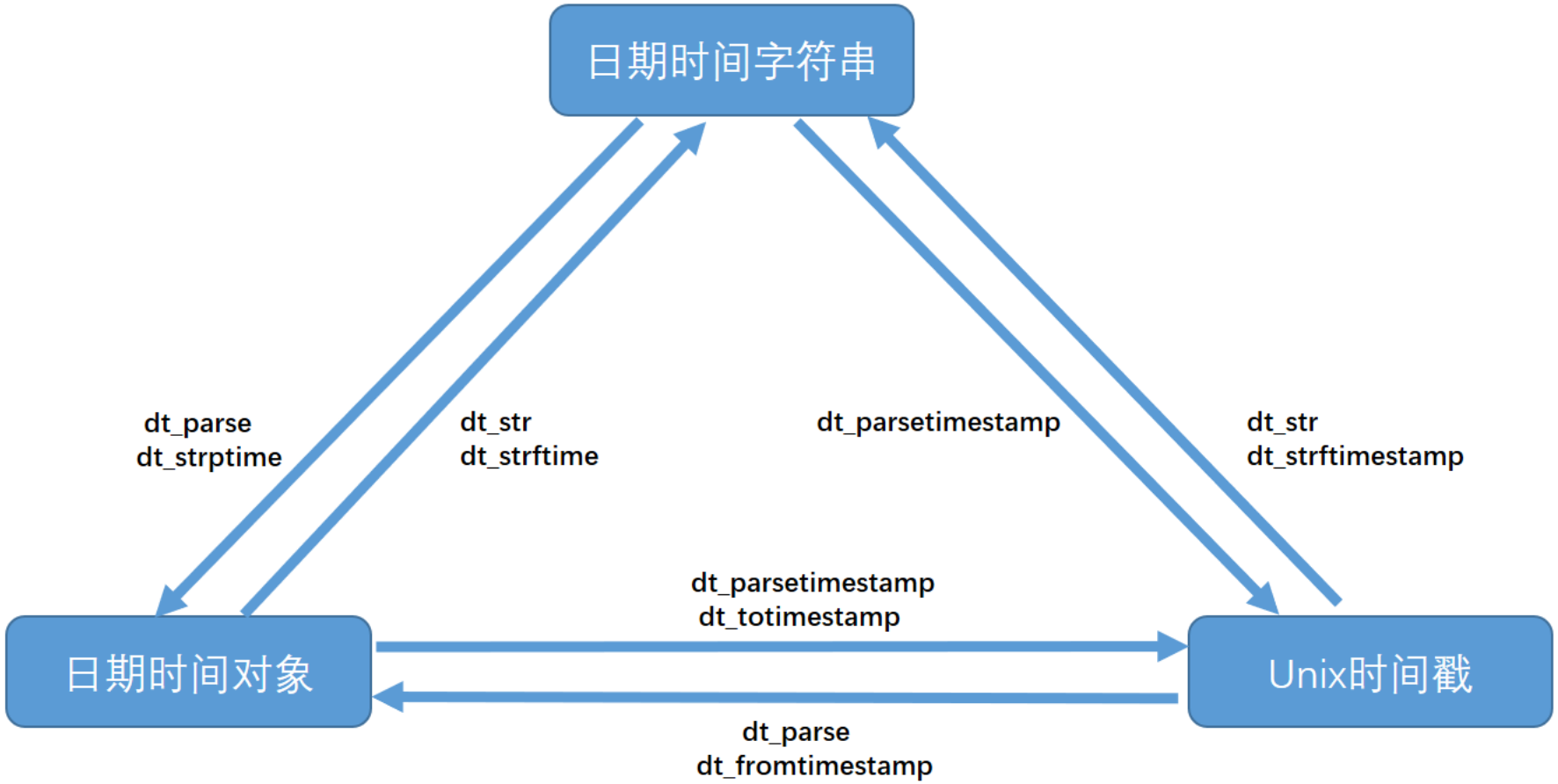
#4 使用组合逻辑做复杂判断

- 为所有：
  - status字段值为200
  - 且 request_method字段值为GET
  - 且 header_length和body_length的字段值之和小于等于1000的日志事件，
- 添加一个字段type，其值为normal。

```
e_if(op_and(e_search("status: 200 and request_method: GET"),
            op_le(op_sum(v("header_length"), v("body_length")), 1000)),
     e_set("type", "normal"))
```

日期时间处理最佳实践

#1 理解Unix时间戳、日期对象与字符串关系



其他函数

类型	函数	说明
获取日期时间属性	dt_prop	智能获取(值或表达式表示时间的)值的特定属性。
获取日期时间	dt_now	获取当前日期时间
	dt_today	获取当前日期(不含时间)
	dt_utcnow	获取UTC时区的当前日期时间
获取Unix时间戳	dt_currentstamp	获取当前Unix时间戳
修改日期时间	dt_truncate	智能将(值或表达式表示时间的)值截取特定时间粒度
	dt_add	智能根据特定时间粒度修改（增加、减少、覆盖）(值或表达式表示时间的)的值
	dt_MO	dt_add下传递给weekday用于表示特定星期一的偏移,负数用op_neg(正数)表示
	dt_TU	dt_add下传递给weekday用于表示特定星期二的偏移,负数用op_neg(正数)表示
	dt_WE	dt_add下传递给weekday用于表示特定星期三的偏移,负数用op_neg(正数)表示
	dt_TH	dt_add下传递给weekday用于表示特定星期四的偏移,负数用op_neg(正数)表示
	dt_FR	dt_add下传递给weekday用于表示特定星期五的偏移,负数用op_neg(正数)表示
	dt_SA	dt_add下传递给weekday用于表示特定星期六的偏移,负数用op_neg(正数)表示
	dt_SU	dt_add下传递给weekday用于表示特定星期日的偏移,负数用op_neg(正数)表示
修改日期	dt_astimezone	智能将(值或表达式表示时间的)值转换（或覆盖）为特定时区的日期时间对象
获取差异	dt_diff	智能按照特定粒度获取两个(值或表达式表示时间的)值的差异值

#2 理解dt_parse等智能函数的限制

- 一个例子，如何转化如下特殊时间字符串为标准日期格式
  - time1: 2019/07/10 06-58-19

```
错误解析方法：

dt_parsetimestamp(v("time1")

正确解析方法：

dt_parsetimestamp(dt_strptime(v("time2"), fmt="%Y/%m/%d %H-%M-%S"))
```

#3 理解时区的概念

- 带有时区信息的日期时间字符串:　2019-06-02 18:41:26**+08:00**
- 不带时区信息的日期施时间字符串: 2019-06-02 10:41:26
  - 'time': '2019-06-02 18:41:26'

```
解析为上海时间
dt_parsetimestamp(v("time"), tz="Asia/Shanghai")

解析为洛杉矶时间
dt_parsetimestamp(v("time"), tz="America/Los_Angeles")

默认解析为UTC时间
dt_parsetimestamp(v("time"))
```

#4 不同时区下的日期时间相互转换（1）

- 某个时间字符串已知是洛杉矶时区，但不带时区信息：
  - 'time': '2019-06-02 18:41:26'
- 如何转化为上海时间？

```
先以洛杉矶时区解析为Unix时间
dt_parsetimestamp(v("time"), tz="America/Los_Angeles")

再以上海时区解析为上海时间
dt_parse(..., tz="Asia/Shanghai")

合并:
e_set("sh_time", dt_parse(dt_parsetimestamp(v("time"),
                                            tz="America/Los_Angeles"),
                          tz="Asia/Shanghai"))
```

#4 不同时区下的日期时间相互转换（2）

- 某个时间字符串已知是上海时区，带时区信息：
  - 'time': '2019-06-02 18:41:26+8:00'
- 如何转化为当时的洛杉矶时间？

```
直接使用dt_astimezone:

e_set("new_time", dt_astimezone(v("time"), tz="America/Los_Angeles"))
```

- 如何强制转化为洛杉矶时间？

```
dt_astimezone(v("time"), tz="America/Los_Angeles", reset=True)
```

#5 理解Unix时间戳的作用与dt_diff

- 计算2个时间的差值（秒）的直观方法
- dt_diff更加直接

#6 使用dt_add做灵活的日期偏移

- 原始时间： "time1": "2019-06-04 2:41:26”

```
年直接改成2018
e_set("time2", dt_add(v("time1"), year=2018))

年直接改成2018
e_set("time2", dt_add(v("time1"), years=op_neg(1)))
```

- 其他参数
  - year(s), day(s), hour(s),
  - minute(s), second(s), microsecond(s),
  - week(s), weekday

#7 理解日期时间格式化指令

- 相关函数

类型	函数	说明
格式化	dt_str	智能转换(值或表达式表示时间的)的值为字符串
解析	dt_strptime	将(值或表达式表示时间的)的字符串解析为日期时间对象
格式化	dt_strftime	将日期时间对象以格式化字符串转换为字符串
格式化	dt_strftimestamp	将(值或表达式的)的Unix时间戳以格式化字符串转换为字符串

指令一览

指令	意义	示例
%a	工作日的缩写。	Sun, Mon, ..., Sat
%A	工作日的全名。	Sunday, Monday, ..., Saturday
%w	以十进制数显示的工作日，其中0表示星期日，6表示星期六。	0, 1, ..., 6
%d	补零后，以十进制数显示的月份中的一天。	01, 02, ..., 31
%b	当地月份的缩写。	Jan, Feb, ..., Dec
%B	当地月份的全名。	January, February, ..., December
%m	补零后，以十进制数显示的月份。	01, 02, ..., 12
%y	补零后，以十进制数表示的，不带世纪的年份。	00, 01, ..., 99
%Y	十进制数表示的带世纪的年份。	0001, 0002, ..., 2013, 2014, ..., 9998, 9999
%H	小时 (24制) 由0填充的十进制	00, 01, ..., 23
%I	小时 (12制) 由0填充的十进制	01, 02, ..., 12
%p	本地化的 AM 或 PM 。	AM, PM
%M	补零后，以十进制数显示的分钟。	00, 01, ..., 59
%S	补零后，以十进制数显示的秒。	00, 01, ..., 59
%f	微秒，由0填充的十进制。	000000, 000001, ..., 999999
%z	UTC偏移形式：±HHMM[SS[.ffffff]] (日期时间不含时区时为空串)。	(empty), +0000, -0400, +1030, +063415
%Z	时区名（日期缺少时区时为空串）。	(empty), UTC, EST, CST
%j	每年的第几天。	001, 002, ..., 366
%U	每年的第几周，星期天是每周第一天。一年中第一个星期天前的日子都被视为week 0。	00, 01, ..., 53
%W	每年的第几周，星期一是每周第一天。一年中第一个星期一前的日子都被视为week 0。	00, 01, ..., 53
%c	本地化的适当日期和时间表示。	Tue Aug 16 21:30:00 1988
%x	本地化的适当日期表示。	08/16/88
%X	本地化的适当时间表示。	21:30:00
%%	字面的 '%' 字符。	%

最佳实践

基础

- 函数调用最佳实践
- 事件判断最佳实践
- 日期时间处理最佳实践

分发汇总

- 数据分发: 跨账号多目标logstore数据分发
- 数据汇总: 跨账号多源logstore数据汇总

非结构化文本解析

- 解析syslog协议框架: 解析syslog/Rsyslog的标准格式
- 一般性文本: 使用正则表达式与grok解析Ngnix日志
- 动态KV: 动态键值对KV解析
- 特定格式的: 特定格式文本的数据加工

持续更新
https://yq.aliyun.com/articles/712381

结构化文本解析

- 复杂JSON格式加工:
  - 多子键为数组的复杂JSON
  - 多层数组对象嵌套的复杂JSON
- CSV格式的: 解析CSV格式的日志

数据富化

- 构建字典与表格
- 从RDS-MySQL获取数据
- 从其他logstore获取数据
- 使用搜索映射做高级数据富化

Thanks

阿里云开发者社区

日志服务数据加工系列培训

欢迎加入
（上海、杭州）

主题：扫平日志分析路上障碍，实时海量日志加工实践培训

讲师：丁来强（成喆）- 阿里高级技术专家 | 唐恺(风毅) - 阿里技术专家

分享介绍

8月7日	8月8日	8月13日	8月14日	8月20日	8月21日	8月28日	8月29日
19:30-20:30	19:30-20:30	19:30-20:30	19:30-20:30	19:30-20:30	19:30-20:30	19:30-20:30	19:30-20:30
数据加工 介绍与实战	数据加工DSL 核心语法介绍	数据加工DSL 语法实践	数据加工动态 数据分发汇集实践	非结构化数据 解析实践	结构化数据 解析实践	数据映射 富化实践	数据加工 可靠性与排错实践