SQLAlert 脚本语言参考手册

南京云利来软件科技有限公司

Contents

1 概述

SQLAlert 是一个基于 ES（Elasticsearch）的异常检测与报警输出引擎，该引擎支持用户使用脚本定义报警检测规则。用户可以在脚本中使用 SQL 语句查询 ES，通过计算及过滤等得出报警数据，再将报警数据写回 ES 或者通过邮件输出。

本文档主要介绍 SQLAlert 支持的脚本语言 RDL（Rule Description Language）的语法，及其提供的库函数的使用方法。本文档需要读者对其他至少一门编程语言的语法有一定的了解，例如：C/C++/Java/JS/PHP 等，编程语言的基础知识不在本文档的描述范围之内。

1.1 说明

SQLAlert 支持的 RDL 脚本是一种函数式脚本语言，不支持面向对象，其设计目的是为了弥补静态配置的不足。RDL 脚本在设计时充分考虑了应用场景，功能强大且语法简单、灵活。支持（1）JSON 数据类型；（2）算术与逻辑运算；（3）无类型的变量；（4）函数数定义；（5）相关库函数。

RDL 脚本库函数提供了常用计算模型，来制定复杂的报警规则，同时用户还可以通过函数定义，来实现更加复杂的规则。

1.2 执行脚本

SQLAlert 使用 Go 语言开发，有非常高的执行效率。由于 RDL 脚本在解释执行时不进行编译，所以不建议在脚本中定义时间复杂度很高的算法或操作。RDL 的执行依赖其解释器 SQLAlert，请确保系统中已经存在。SQLAlert 的安装及使用，不在本文档的介绍范围内。SQLAlert 在解释执行 RDL 脚本文件时，约定其需以.rule 结尾，下面通过 “Hello World” 示例来介绍 RDL 脚本是如何执行的：

```
print("hello world");
```

将上述内容保存到以.rule 结尾的文件（例如 hello.rule）中，使用如下命令来执行：

```
sqlalert -t hello.rule
```

上述脚本代码中，通过 print() 函数来打印 “Hello World” 字符串，该语句以分号（;）结束。执行完该脚本后，会在终端上打印出 Hello World 信息。

RDL 脚本支持 UTF-8 编码，但其变量、操作符等均仅支持英文格式。用户可以使用 sqlalert -h/–help 查看 sqlalert 的更多选项。

2 RDL 语法

RDL 脚本汲取了其他编程语言较好的语法格式，并进行了一定的修改，以达到相同的功能使用户输入最少的内容。

2.1 语句与注释

RDL 脚本以语句为单位执行，每条执行语句必需以分号（;）结束。使用花括号（{}）来指定语句块，语句块可以不以分号（;）结束，但语句块内的每条语句都必需以分号（;）结束。例如下述代码片断，都是有效的语句：

```
1  print("world");          // 单独的语句
2
3  {                        // 包含两条语句的语句块
4      print("hello");
5      print("world");
6  }
```

在上述代码片断中，第 1 行与第 5 行为注释内容。在 RDL 脚本中可以用 # 和 // 来对内容进行注释，从 # 或 // 开始直到行尾均为注释内容，SQLAlert 在解释执行脚本时会忽略所有注释内容。在 RDL 脚本中不支持类 C 语言中的块级注释。

2.2 数据类型

RDL 支持五种基本的数据类型：（1）数（包括整型与浮点）；（2）字符串；（3）数组（List）；（4）字典（Dict）；（5）布尔（true、false），熟悉 JSON 的读者应该知道这四种数据均为 JSON 支持的数据类型。如下代码片断所示，为四种数据类型的示例：

```
1  // 数（整型与浮点），支持 10 进制整数、16 进制整数、10 进制浮点数。
2  123456    0xFF    0xFF    123.456
3  // 字符串，支持单引号（'）、双引号（"）、反引号（`）。
4  'string'    "string"    `string`
5  // 数组，元素可以是任意类型数据与表达式。
6  [ 123, 'string', 100 + 200, {"name": "zhang"}, [1, 2, 3] ]
7  // 字典，KEY 只能为字符串；VALUE 可以是任意类型数据与表达式。
8  { "value": "zhang", "list": [ 1, 2, 3 ], "dict": { "vlaue": 1 } }
9  // 布尔类型数据
10 true      false
```

其中字符串、List、Dict 类型的数据均可以分多行定义，但是字符串在多行定义时，换行符也是字符串内容的一部分。

2.3 变量

RDL 脚本中的变量是无类型（或弱类型）的，不需要任何声明，可以直接赋值使用。将什么类型的数据赋值给变量，则变量就是什么类型。如下述代码片断所示：

```
a = 123;                        // 将整型赋值给变量 a，   则变量 a 即为整型;
a = "a string";                 // 将字符串赋值给变量 a，则变量 a 即为字符串;
a = [ 1, 2, 3 ];                // 将 List 赋值给变量 a，则变量 a 即为 List;
a = { "value": [ 1, 2, 3 ] }; // 将 Dict 赋值给变量 a，则变量 a 即为 Dict;
```

如果一个变量在使用前没有赋值任何值，则这个变量的值为 null，表示空。数值 null 也可以单独作为一个值来使用，其地位与基本数据类型是一样的。

需要注意，变量名不能包含特殊字符及 RDL 脚本预置的所有符号。可以包含下划线与数字，但不能以数字开头。

2.4 字符串中的变量

在定义字符串时，可以使用%(name) 来引用当前执行环境中的变量值，相当于字符串的格式化。其中，name 为变量的名，如下述代码所示：

```
a = "zhang";
b = 123;
print("a = %(a); b= %(b); c = %(c)");
```

执行该示例代码后，得到如下输出内容：

```
a = zhang; b= 123; c = null
```

其中，%(a) 被变量 a 的值（字符串 zhang）替换；%(b) 被变量 b 的值（整数 123）替换；对于%(c)，由于变量 c 没有定义，被替换成 null。

2.5 表达式

RDL 脚本支持五种算术表达式：加（+）、减（-）、乘（*）、除（/）、取余（%）；六种比较表达式：小于（<）、小于等于（<=）、大于（>）、大于等于（>=）、等于（==）、不等于（!=）；三种逻辑表达式：且（&&, and）、或（||, or）、否（!, not）。

表达式之间可能使用括号 () 进行无限嵌套，括号内的操作符优先级将高于括号外的操作符。如下代码片断给出了表达的部分示例：

```
a = ((1 + 5) * (4 - 2)) / 3;
print("a =", a);

b = (100 >= 200) && (1 > 2);
print("b =", b);
```

与其它编程语言一样，操作符否（!, not）为单目操作符，其它均为双目操作符。除此之外，RDL 还支持一种三目操作符，... ? ... : ... 熟悉 C 或 java 的读者应该对这种三目操作符比较熟悉。下面给出该三目操作符的使用示例：

```
name = 100 > 200 ? 'zhang' : 'wang';
print("name =", name);
```

上述代码中，比较表达式 $100 > 200$ 的值为假（false）所以赋值语句将字符串 'wang' 赋值给变量 name。

2.6 分支语句

RDL 脚本支持 if expr { } else if expr { } else {} 分支，根据条件来执行不同的语句块，语句块必需包含在花括号内，花括号后面不需要以分号结束。条件表达式可以是任务类型的数据、表达式或者函数，可以包含在括号 () 内，也可以不使用括号。

分支语句示例：

```
if      a > 200 { do_something(); }
else if a > 100 { do_something(); }
else            { do_something(); }
```

分支语句中，必需以 if 分支开始，中间可以有多个 else if 分支，也可以没有，结尾可以出现 else 分支，也可以没有，但最多只能有一个。

2.7 循环语句

RDL 脚本中支持 for 循环语句，包括（1）for expr { } 和（2）for init; cond; next { } 两种形式，第一种形式类似于 C 中的 while 循环。

第（1）种 for 循环示例，打印出 0 ~ 9 的 10 个数字，代码如下：

```
a = 0;
for (a > 10) {
    print("a =", a);
    a ++;
}
```

第（2）种 for 循环示例，同样打印出 0 ~ 9 的 10 个数字，代码如下：

```
for a = 0; a < 10; a ++ {
    print("a =", a);
}
```

循环中支持 continue 和 break 关键字，同时支持循环的嵌套。

2.8 函数定义

RDL 脚本中通过关键字 def 来定义函数，示例如下：

```
def my_print(value) {          // value 为函数的参数
    print("value =", value);
}
my_print("zhang");             // 函数调用
```

函数定义中支持使用 return 关键字从函数中退出，同时可以指定函数的返回值。

2.9 文件包含

RDL 脚本中可以通过 include 和 import 两个关键字来包含或者引入另一个 RDL 脚本文件。示例代码如下：

（1）第一个脚本文件 test1.rule：

```
def test_print(value) {
    print("value =", value);
}
```

（2）在第二个脚本文件中包含第一个脚本文件（test2.rule):

```
include "test1.rule";
test_print("zhang");          // 在脚本文件 test1.rule 中定义
```

上述代码示例中，在脚本 test2.rule 中调用了 test1.rule 中定义的函数 test_print()，在 test2.rule 中同时还可以引用 test1.rule 中定义的任何变量。

在 RDL 脚本中，可以在任意位置使用 include 和 import 来包含另一个脚本文件，二者的唯一区别是：通过 include 包含的脚本只会被执行一次，被引用的脚本只会在第一次 include 的时候执行；而通过 import 引入的脚本每次都会被执行，后面执行的结果将会覆盖前面的执行结果。另一个代码示例：

```
// 脚本文件 test1.rule 内容
name = "zhang";
age  = 18;

def print_name_age() {
    print("name =", name, "age =", age);
}


// 脚本文件 test2.rule 内容
name = "wang";

include "test1.rule";

age  = 25;
print_name_age();
```

请读者自行验证该示例的输出内容。

3 RDL 库函数

本章节将描述 SQLAlert 为 RDL 语言提供的库函数, 后续子章节将分类对其进行描述。

3.1 基础函数

本小节描述 RDL 函数库中的基础函数。

3.1.1 print(v, ...)

本函数接受任意多个参数，将给定的所有参数在同一行进行打印输出，打印时每个参数之间用空格分隔。对于 List 与 Dict 数据，按标准 JSON 进行格式化输出。

该函数代码示例如下：

```
num  = 100;
list = [ 1, "string" ];
dict = { "name": "zhang", "list": [ 1, 2, 3 ] };
print("values =", num, list, dict);
```

示例代码的输出内容如下：

```
values = 100 [1,"string"] {"list":[1,2,3],"name":"zhang"}
```

3.1.2 pprint(v, ...)

本函数为 print() 函数的美化版本，首字母 p 为 pretty 的简写，该函数将给定的参数按 JSON 格式进行缩进、分行格式化，然后再输出。在打印输出时，同样在多个参数之间以空格分隔。

该函数代码示例如下：

```
list = [
    { "name": "zhang", "age": 18, "desc": "A good man." },
    { "name": "wang",  "age": 22, "desc": "A good man too."}
];

pprint(list);
```

示例代码的输出内容如下：

```
[
    {
        "name": "zhang",
        "age": 18,
        "desc": "A good man."
    },
    {
        "name": "wang",
        "age": 22,
        "desc": "A good man too."
    }
]
```

3.1.3 print_list(list)

本函数对指定的 List 进行打印输出。对 List 的每一个元素按 JSON 进行格式化，并将每个元素在同一行打印输出。在 List 元素较多，且需要对元素进行比较时，该函数非常有用。该函数接受 1 个参数：

- list: 数组（List）类型，数据内的元素可以是任意类型。

本函数在打印 list 时，不输出起始符与结束符 []，如果 list 为空（即元素个数为零）时，将不输出任何内容。

该函数代码示例如下：

```
list = [
    { "name": "zhang", "age": 18, "desc": "A good man." },
    { "name": "wang",  "age": 22, "desc": "A good man too."}
];

print_list(list);
```

示例代码的输出内容如下：

```
{"age":18,"desc":"A good man.","name":"zhang"}
{"age":22,"desc":"A good man too.","name":"wang"}
```

3.1.4 print_ctx() || print_context()

本函数打印出脚本执行到当前位置时，执行上下文中的所有变量的值，函数不接受任何参数。print_context() 是该函数的完整函数名，print_ctx() 是函数的别名，二者在功能及使用上是一致的，本文档后续不再对这种格式进行解释。

该函数代码示例如下：

```
name = "zhang";
list = [ 1, 2, 3, 4, 5  ];

print_ctx();
```

示例代码的输出内容如下：

```
Context-Global:
__etc_scripts__ = /usr/local/etc
name = zhang
list = [1,2,3,4,5]

Context-Local: empty
```

输出结果中，变量 *__etc_scripts__* 为 SQLAlert 执行 RDL 脚本时，自动添加的全局变量，用于指定脚本执行目录（搜索目录）。更多说明请参阅本文档附录部分。

3.1.5 exit()

该函数结束（终止）当前脚本的执行，该函数执行后，后面的所有语句将不再执行，函数不接受任何参数。

该函数代码示例如下：

```
print("hello");
print("world");
exit();
print("rest");
```

示例代码的输出内容如下：

```
hello
world
```

示例中，在 exit() 函数后面的语句 print("rest") 没有被执行。

3.1.6 error(msg)

与 exit() 函数类似，该函数结束（终止）当前脚本的执行，不同的是该函数会在结束脚本执行时，输出错误信息，函数接受 1 个参数：

- msg: 任意数据类型，需要打印输出的错误信息内容。

该函数代码示例如下：

```
def test_error() {
    error("An error message");
}
test_error();
```

示例代码的输出内容如下：

```
2018-02-03 18:52:03 SQLAlert: [ERR] An error message
in error(), in line 2 in file test.rule
called by test_error(), in line 5 in file test.rule in task 'test'
```

如示例中所示，该函数在输出错误信息的同时会输出函数的调用栈信息。

3.1.7 copy(v)

本函数深度拷贝给定的对象，并返回新拷贝的对象。对新对象的任何修改不会影响到旧对象，该函数接收 1 个参数：

- v: 任意数据类型，需要拷贝的对象。

该函数代码示例如下：

```
a = { "name": "zhang", "age": 18 };
b = copy(a);
b["age"] = 22;

print("a =", a)
print("b =", b)
```

示例代码的输出内容如下：

```
a = {"age":18,"name":"zhang"}
b = {"age":22,"name":"zhang"}
```

3.1.8 len(v)

本函数用于计算给定对象的长度：(1) 对于字符串，返回字符串的长度；(2) 对于数组，返回数组元素的个数；(3) 对于字典，返回字典 KEY 的个数；(4) 对于其他类型数据，则返回 0。该函数接收 1 个参数：

- v: 任意数据类型，需要计算长度的对象。

该函数代码示例如下：

```
a = "zhang";
print("len(str) =", len(a));

a = [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ];
print("len(list) =", len(a));

a = { "name": "zhang", "age": 10, "desc": "a good man" };
print("len(dict) =", len(a));
```

示例代码的输出内容如下：

```
len(str) = 5
len(list) = 10
```

```
len(dict) = 3
```

3.1.9 get_ctx(name) || get_context(name)

本函数返回当前执行上下文中的指定变量值，在搜索上下文时将忽略变量名的大小写。该函数接收 1 个参数：

- name: 字符串类型，需要获取的变量名。

该函数代码示例如下：

```
aAcC = "zhang";
print(get_ctx("aAcC"));
```

示例代码的输出内容如下：

```
zhang
```

3.1.10 set_session(name, value)

本函数在脚本的 session 中设置一个名为 name 值为 value 的变量，以记录脚本执行的状态。SQLAlert 在循环调度执行脚本时，会为每一次执行创建一个新的执行上下文，同时将上次执行的 session 保存到当前的执行上下文中。脚本可以在当前执行时访问上一次执行的状态，以实现特殊的需求。

需要注意的时，session 一旦被设置以后，将不会被自动回收，需要用户在脚本中显式的删除。通过 set_session(name, null) 设置某个名为 name 的 session 值为 null 即可将其使用的资源回收。在脚本中使用 session 功能需要特别小心，否则可能会产生内存泄漏的问题。

在脚本中可以将任意类型的数据保存到 session 中，该函数接收 2 个参数：

- name: 字符串类型，session 的名称。
- value: 任意数据类型，session 的值。

该函数代码示例如下：

```
name  = "test";
value = [ { "name": "zhang" }, { "name": "wang" } ];
set_session(name, value);
```

该示例不会输出任何内容，只是在脚本的 session 中设置了一个名为 test 的变量。请阅读 get_session() 函数以了解如果获取脚本 session 的值。

3.1.11 get_session(name)

本函数从脚本执行上下文的 session 中获取名为 name 的 session 值。该函数接收 1 个参数：

- name: 字符串类型，session 的名称。

该函数代码示例如下：

```
name  = "test";
value = [ { "name": "zhang" }, { "name": "wang" } ];
set_session(name, value);

mySession = get_session("test");
print("my session =", mySession);
```

示例代码的输出内容如下：

```
my session = [{"name":"zhang"},{"name":"wang"}]
```

用户可以通过如下指令来让 SQLAlert 循环执行指定的脚本：

```
sqlalert --interval 5m --from "2017-10-12 08:00:00" --to "2017-10-12 18:00:00" -t test.rule
```

上述指令中，sqlalert 将循环执行指定的脚本 test.rule，并且修改脚本运行时的系统时间戳，时间戳从 2017-10-12 08:00:00 按每次递增 5m 的间隔进行变化，直到时间戳等于或超过 2017-10-12 18:00:00 停止。更多说明请参阅 SQLAlert 安装使用文档。

3.1.12 load_json(path)

本函数将指定的文件以 JSON 格式加载到脚本上下文中，如果 path 为绝对路径，则直接加载；如果 path 为相对路径，则会在 path 之前添加 ETC 前缀。在使用 SQLAlert 执行 RDL 脚本时，可以通过选项 -e/–etc 来指定 ETC 路径，如果不指定，默认为 /usr/local/etc。函数接收 1 个参数：

- path: 字符串类型，文件路径名。

该函数代码示例如下：

```
pprint(load_json("data.json"));
pprint(load_json("/home/<user>/sqlalert/data/data.json"));
```

上述示例代码仅作为参考，无法执行，请用户换成自己的路径进行测试。

3.2 类型函数

本小节描述 RDL 中的数据类型相关函数。

3.2.1 is_num(v)

本函数检测给定的值是否是一个数（包括整数和浮点数），如果给定的参数是一个数，则返回 true，否则返回 false，该函数接收 1 个参数：

- v: 任意类型，需要检测的值或变量。

该函数代码示例如下：

```
a = 123;
print(a, "is number:", is_num(a));

a = 123.456;
print(a, "is number:", is_num(a));

a = "zhang";
print(a, "is number:", is_num(a));
```

示例代码的输出内容如下：

```
123 is number: true
123.456 is number: true
zhang is number: false
```

3.2.2 is_int(v)

本函数检测给定的值是否是一个整数，如果给定的参数是一个整数，则返回 true，否则返回 false，该函数接收 1 个参数：

- v: 任意类型，需要检测的值或变量。

该函数代码示例如下：

```
a = 123;
print(a, "is int:", is_int(a));

a = 123.456;
print(a, "is int:", is_int(a));

```

```
a = "zhang";
print(a, "is int:", is_int(a));
```

示例代码的输出内容如下：

```
123 is int: true
123.456 is int: false
zhang is int: false
```

3.2.3 is_float(v)

本函数检测给定的值是否是一个浮点数，如果给定的参数是一个浮点数，则返回 true，否则返回 false，该函数接收 1 个参数：

- v: 任意类型，需要检测的值或变量。

该函数代码示例如下：

```
a = 123;
print(a, "is float:", is_float(a));

a = 123.456;
print(a, "is float:", is_float(a));

a = "zhang";
print(a, "is float:", is_float(a));
```

示例代码的输出内容如下：

```
123 is float: false
123.456 is float: true
zhang is float: false
```

3.2.4 is_str(v)

本函数检测给定的值是否是一个字符串，如果给定的参数是一个字符串，则返回 true，否则返回 false，该函数接收 1 个参数：

- v: 任意类型，需要检测的值或变量。

该函数代码示例如下：

```
a = 123;
print(a, "is str:", is_str(a));

a = 123.456;
print(a, "is str:", is_str(a));

a = "zhang";
print(a, "is str:", is_str(a));
```

示例代码的输出内容如下：

```
123 is str: false
123.456 is str: false
zhang is str: true
```

3.2.5 is_list(v)

本函数检测给定的值是否是一个数组（List），如果给定的参数是一个数组，则返回 true，否则返回 false，该函数接收 1 个参数：

- v: 任意类型，需要检测的值或变量。

该函数代码示例如下：

```
a = 123;
print(a, "is list:", is_list(a));

a = [1, 2, 3 ];
print(a, "is list:", is_list(a));

a = { "name": "zhang" };
print(a, "is list:", is_list(a));
```

示例代码的输出内容如下：

```
123 is list: false
[1,2,3] is list: true
{"name":"zhang"} is list: false
```

3.2.6 is_dict(v)

本函数检测给定的值是否是一个字典（Dict），如果给定的参数是一个字典，则返回 true，否则返回 false，该函数接收 1 个参数：

- v: 任意类型，需要检测的值或变量。

该函数代码示例如下：

```
a = 123;
print(a, "is dict:", is_dict(a));

a = [1, 2, 3 ];
print(a, "is dict:", is_dict(a));

a = { "name": "zhang" };
print(a, "is dict:", is_dict(a));
```

示例代码的输出内容如下：

```
123 is dict: false
[1,2,3] is dict: false
{"name":"zhang"} is dict: true
```

3.2.7 is_func(v)

本函数检测是否存在指定的函数，如果指定的函数存在，则返回 true，否则返回 false，该函数接收 1 个参数：

- v: 字段串类型，需要检测的函数名。

该函数代码示例如下：

```
print("print is func:", is_func("print"));
print("xxxxx is func:", is_func("xxxxx"));
```

示例代码的输出内容如下：

```
print is func: true
xxxxx is func: false
```

3.2.8 is_null(v)

本函数检测给定的值是否为 null（未赋值的变量即为 null）如指定的参数值为 null，则返回 true，否则返回 false，该函数接收 1 个参数：

- v: 任意类型，需要检测的值或变量。

该函数代码示例如下：

```
print("null is null", is_null(null));

a = null;
print("a %(a) is null:", is_null(a));

a = 0;
print("a %(a) is null:", is_null(a));
```

示例代码的输出内容如下：

```
null is null true
a null is null: true
a 0 is null: false
```

3.3 字符串函数

本小节介绍 RDL 中字符串相关函数。

3.3.1 str(v)

本函数将指定的参数转化成字符串类型，返回转化后的字符串。该函数接收 1 个参数：

- v: 任意类型，需要转化值或变量。

该函数代码示例如下：

```
print(str(1));
print(str([1, 2, 3]));
```

示例代码的输出内容如下：

```
1
[1,2,3]
```

3.3.2 split(str, sep)

本函数将指定的字符串分割成一个字符串数组，该函数接收 2 个参数：

- str: 字符串类型，需要分割的字符串。
- sep: 字符串类型「可选」，字符串分割符。如果该参数不指定，则以所有空白为分隔符，即将字符串分割为若干个 “域”。

该函数代码示例如下：

```
print(split("zhang Wang  li han"));
print(split("zhang Wang  li han", " "));
```

示例代码的输出内容如下：

```
["zhang","Wang","li","han"]
["zhang","Wang","","li","han"]
```

请读者自行研究该示例两处 split() 函数用法的差异。

3.3.3 trim(str, sep)

本函数去除指定字符串左右两边的空白，返回新字符串，原字符串保持不变。该函数接收 1 个参数：

- str: 字符串类型，需要去除空白的字符串。

该函数代码示例如下：

```
print(trim("      zhang      "));
```

示例代码的输出内容如下：

```
zhang
```

3.3.4 fmt_int(num)

本函数将指定的数值格按照整型进行格式化成字符串，在格式化浮点数时，将舍去所有小数部分。该函数接收 1 个参数：

- num: 整型或浮点型，需要格式化的数值。

该函数代码示例如下：

```
print(fmt_int(10));
print(fmt_int(123.567));
print(fmt_int(123.456));
```

示例代码的输出内容如下：

```
10
123
123
```

3.3.5 fmt_float(float, n)

本函数将指定的浮点数格式成字符串，保留指定位数的小数。舍去多余的小数位数时，按“四舍五入”进行进位操作。该函数接收 1 个参数：

- float: 浮点数，需要格式化的浮点数；
- n: 需要保留的小位个数。

该函数代码示例如下：

```
print(fmt_float(123.567898, 0));
print(fmt_float(123.456253, 2));
```

示例代码的输出内容如下：

```
124
123.46
```

3.3.6 fmt_bytes(int) || fmt_bits(int)

本函数将指定的整数格式化成带字节（B）或比特单位（b）的字符串，格式化时按 1024 进行取余操作，并保留小数点后面 1 位。该函数接收 1 个参数：

- int: 整数，需要格式化的整数。

该函数代码示例如下：

```
print(fmt_bytes(1234859));
print(fmt_bits(190933388));
```

示例代码的输出内容如下：

```
1.2 MB
182.1 Mb
```

3.3.7 fmt_pct(num) || fmt_percentage(num)

本函数将指定的整数或浮点数据格式化成百分比字符串，并保留小数点后 2 位。该函数接收 1 个参数：

- num: 整数或浮点数，需要格式化的整数。

该函数代码示例如下：

```
print(fmt_pct(0.12303));
print(fmt_pct(0.12344));
```

示例代码的输出内容如下：

```
12.30%
12.34%
```

3.3.8 fmt_time(int, type)

本函数将指定的整数格式化带单位的时间间隔，并保留秒后面小数点 3 位，来表示毫秒。该函数接收 2 个参数：

- int: 整数，需要格式化的时间间隔。
- type: 字符串类型「可选」，时间间隔类型 ms（毫秒）或 us （微秒），如果不指定，则默认为 ms。

该函数代码示例如下：

```
print(fmt_time(1234567));
print(fmt_time(1234567, "ms"));
print(fmt_time(1234567, "us"));
```

示例代码的输出内容如下：

```
20m 34.567s
20m 34.567s
1.234s
```

3.4 数组函数

本小节介绍 RDL 中数组相关的操作函数。

3.4.1 append(list, v, ...)

本函数将指定的值或变量添加到 list 的尾部，本函数接收 2 个以上参数：

- list: 数组类型，所以其他参数将被添加到该数组的尾部；
- v, ...: 任意数据类型，n 个（n >= 1）需要添加的值或变量；

该函数代码示例如下：

```
list = [];
list = append(list, 1, 2);
print(list);
```

示例代码的输出内容如下：

```
2 [1,2]
```

3.4.2 append_first(list, v, ...)

本函数将指定的值或变量添加到 list 的头部，函数接收 2 个以上参数：

- list: 数组类型，所以其他参数的值将被添加到该数组的头部；
- v, ...: 任意数据类型，n 个（n >= 1）需要添加的值或变量；

该函数代码示例如下：

```
list = [1, 2];
list = append_first(list, 3);
print(len(list), list);
```

示例代码的输出内容如下：

```
3 [3,1,2]
```

3.4.3 append_list(listDst, listSrc)

本函数将数组 listSrc 中的所有元素添加到数组 listDst 内，函数接收 2 个参数：

- listDst: 数组类型，listSrc 数组中的所有元素都将被添加到该参数指定的数组中；
- listSrc: 数组类型，需要添加的数组；

该函数代码示例如下：

```
list = [1, 2];
list = append_list(list, [3, 4]);
print(len(list), list);
```

示例代码的输出内容如下：

```
4 [1,2,3,4]
```

3.4.4 remove_first(list)

本函数删除指定数组的第一个元素，并返回元素删除后的数组。函数接收 1 个参数：

- list: 数组类型，需要删除元素的数组；

该函数代码示例如下：

```
list = [1, 2,  3,  4,  5];
list = remove_first(list);
print(len(list), list);
```

示例代码的输出内容如下：

```
4 [2,3,4,5]
```

3.4.5 remove_first(list)

本函数删除指定数组的最后一个元素，并返回元素删除后的数组。函数接收 1 个参数：

- list: 数组类型，需要删除元素的数组；

该函数代码示例如下：

```
list = [1, 2,  3,  4,  5];
list = remove_last(list);
print(len(list), list);
```

示例代码的输出内容如下：

```
4 [1,2,3,4]
```

3.4.6 join(list, sep)

本函数将数组中的所有元素连接成一个字符串，函数接收 2 个参数：

- list: 数组类型，需要连接的数组；
- sep: 字符串类型，元素与元素之间的分隔符。

该函数代码示例如下：

```
list = [1, 2, 3, 4, 5];
print(join(list));
print(join(list, "_"));
```

示例代码的输出内容如下：

```
12345
1_2_3_4_5
```

3.4.7 slice(list, from, to)

本函数截取指定数组的一部分作为子数组（或切片），并将子数组返回给调用者，子数组与原数组共享元素的引用。对子数组的任何修改，将直接影响到原数组的数据，所以不建议在脚本中对截取的子数组做任何修改，除非在这样的需求。函数接收 3 个参数：

- list: 数组类型，需要截取的数组；
- from: 整型数据，截取元素的起始位置；
- to: 整型数据「可选」，截取元素的结束位置。如果该参数不指定则返回的子数组包括从 from 直到原数组的结束位置。

该函数代码示例如下：

```
list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
print(slice(list, 3, 5));
print(slice(list, 3));

list2 = slice(list, 3, 5);
list2[0] = 100;
print(list);
```

示例代码的输出内容如下：

```
[4,5]
[4,5,6,7,8,9,10]
```

```
[1,2,3,100,5,6,7,8,9,10]
```

如果想对截取出一个新的、与原数组完成不相干的子数据，请使用 copy() 函数。

3.4.8 sort(list)

本函数对指定的数组进行升序（从小到大）排序，并返回排序后的新数组，原数组保持不变。函数接收 1 个参数：

- list: 数组类型，需要排序的数组；

该函数代码示例如下：

```
list = [2, 4, 3, 10, 1, 6, 7, 8, 5, 9];
print(sort(list));
```

示例代码的输出内容如下：

```
[1,2,3,4,5,6,7,8,9,10]
```

3.4.9 sort_r(list)

本函数对指定的数组进行降序（从大到小）排序，并返回排序后的新数组，原数组保持不变。函数接收 1 个参数：

- list: 数组类型，需要排序的数组；

该函数代码示例如下：

```
list = [2, 4, 3, 10, 1, 6, 7, 8, 5, 9];
print(sort_r(list));
```

示例代码的输出内容如下：

```
[10,9,8,7,6,5,4,3,2,1]
```

3.4.10 reverse(list)

本函数将指定的数组中的元素进行反转，并返回反转后的新数组，原数组保持不变。函数接收 1 个参数：

- list: 数组类型，需要反转的数组；

该函数代码示例如下：

```
list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
print(reverse(list));
```

示例代码的输出内容如下：

```
[10,9,8,7,6,5,4,3,2,1]
```

3.4.11 list_to_dict(list, keys, sep)

本函数所处理的数组中，每个元素必需是一个字典类型数据。函数将数组中每个字典中指定的 keys 连接起来作为 KEY，对应的元素作为 VALUE 生成一个新的字典，并将新生成的字典返回给调用者。新字典中的 VALUE 为一个数组，数组的元素为 KEY 相同的原数组的所有元素。

函数接收 3 个参数：

- list: 数组类型，需要转换的数组；
- keys: 数组类型，数组的元素必需为字符串。需要连接的 KEY 列表；
- sep: 字符串类型「可选」，连接 KEY 时，每个字之间的分隔符，如果不指定则分隔符为空。

该函数代码示例如下：

```
list = [
    { "name": "zhang", "age": 18, "desc": "a good man" },
    { "name": "zhang", "age": 22, "desc": "a good man too" },
    { "name": "wang",  "age": 25, "desc": "a good doctor" },
    { "name": "wang",  "age": 30, "desc": "a good doctor too." }
];
pprint(list_to_dict(list, ["name"]));
```

示例代码的输出内容如下：

```
{
    "zhang": [
        {
            "name": "zhang",
            "age": 18,
            "desc": "a good man"
        },
        {
            "age": 22,
            "desc": "a good man too",
            "name": "zhang"
```

```
        }
    ],
    "wang": [
        {
          "name": "wang",
          "age": 25,
          "desc": "a good doctor"
        },
        {
          "name": "wang",
          "age": 30,
          "desc": "a good doctor too."
      }
    ]
}
```

3.5 字典函数

本小节介绍 RDL 中字典相关的操作函数。

3.5.1 keys(dict)

本函数返回指定字典的所有 KEY 的值，该函数返回一个数组，包含所有的 KEY。由于字典是基于动态 hash 表实现的，所以每次获取的 KEY 数组元素的顺序是不一致的，但 KEY 数组的最量是一样的。函数接收 1 个参数：

- dict: 字典类型，指定的字典。

该函数代码示例如下：

```
dict = { "name": "zhang", "age": 18, "desc": "a good man" };
print(keys(dict));
```

示例代码的输出内容如下：

```
["name","age","desc"]
```

3.5.2 values(dict)

本函数返回指定字典的所有 VALUE 的值，该函数返回一个数组，包含所有的 VALUE。由于字典是基于动态 hash 表实现的，所以每次获取的 VALUE 数组元素的顺序是不一致的，但 VALUE 数组的最量是一

样的。函数接收 1 个参数：

- dict: 字典类型，指定的字典。

该函数代码示例如下：

```
dict = { "name": "zhang", "age": 18, "desc": "a good man" };
print(values(dict));
```

示例代码的输出内容如下：

```
["a good man","zhang",18]
```

3.5.3 delete(dict, key)

本函数从指定的字典中删除指定的 KEY，该函数不返回任何值。函数接收 2 个参数：

- dict: 字典类型，需要删除元素的字典；
- key: 字符串类型，需要删除的元素的 KEY。

该函数代码示例如下：

```
dict = { "name": "zhang", "age": 18, "desc": "a good man" };
delete(dict, "name");
print(dict);
```

示例代码的输出内容如下：

```
{"age":18,"desc":"a good man"}
```

3.5.4 dict_get(dict, key, def)

本函数从指定的字典中读取指定 KEY 的值，如果 KEY 不存在则返回指定的默认值 def。函数接收 3 个参数：

- dict: 字典类型，指定的字典；
- key: 字符串类型，需要获取的元素的 KEY；
- def: 任意类型「可选」，KEY 不存在时返回的默认值，如果不指定，则默认值为 null。

该函数代码示例如下：

```
dict = { "name": "zhang", "age": 18, "desc": "a good man" };
print(dict_get(dict, "name"));
```

```
print(dict_get(dict, "dummy"));
print(dict_get(dict, "dummy", "hello"));
```

示例代码的输出内容如下：

```
zhang
null
hello
```

3.5.5 join_values(dict, keys, sep)

本函数将字典中指定的 KEY 对应的值连接起来，生成一个字符串并返回。函数接收 3 个参数：

- dict: 字典类型，指定的字典；
- keys: 数组类型，数组的元素必需为字符串，需连接的 KEY 列表；
- sep: 字符串类型，连接时元素之间的分隔符。

该函数代码示例如下：

```
dict = { "name": "zhang", "age": 18, "desc": "a good man" };
print(join_values(dict, ["name", "desc"], " = "));
```

示例代码的输出内容如下：

```
zhang = a good man
```

3.6 时间函数

本小节介绍 RDL 中字典相关的操作函数。

3.6.1 time(sep)

本函数获取当前时间戳，并格式化成字符串。如果脚本中配置了全局变量 *_now_*，则该函数返回 *_now_* 所表示的时间戳。函数接收 1 个参数：

- sep: 字符串类型，日期与时间之间的分隔符。

该函数代码示例如下：

```
print(time());
print(time("T"));

```

```
__now__ = "2015-03-02 08:10:00.234";
print(time());
```

示例代码的输出内容如下：

```
2018-02-07 11:04:25.824+08:00
2018-02-07T11:04:25.824+08:00
2015-03-02 08:10:00.234+08:00
```

3.6.2 sys_time(sep)

本函数获取当前时间戳，并格式化成字符串。函数返回的数值不受全局变量 *__now__* 的影响。函数接收 1 个参数：

- sep: 字符串类型，日期与时间之间的分隔符。

该函数代码示例如下：

```
print(time());
print(sys_time());

__now__ = "2015-03-02 08:10:00.234";
print(time());
print(sys_time());
```

示例代码的输出内容如下：

```
2018-02-07 11:08:27.865+08:00
2018-02-07 11:08:27.865+08:00
2015-03-02 08:10:00.234+08:00
2018-02-07 11:08:27.865+08:00
```

3.6.3 now(sep)

本函数获取当前时间戳整数值，精确到毫秒。如果脚本中配置了全局变量 *__now__*，则该函数返回 *__now__* 所表示的时间戳，函数不接收任何个参数。

该函数代码示例如下：

```
print(now());

```

```
__now__ = "2015-03-02 08:10:00.234";
print(now());
```

示例代码的输出内容如下：

```
1517972720018
1425255000234
```

3.6.4 sys_now(sep)

本函数获取当前时间戳整数值，精确到毫秒。函数返回的数值不受全局变量 *_now_* 的影响，函数不接收任何个参数。

该函数代码示例如下：

```
print(now());
print(sys_now());

__now__ = "2015-03-02 08:10:00.234";
print(now());
print(sys_now());
```

示例代码的输出内容如下：

```
1517972986212
1517972986213
1425255000234
1517972986213
```

3.6.5 check_datetime(list)

本函数检测当前日期和时间是否被配置成 on 状态，如果是则返回 true，否则返回 false。如果日期及时间未配置，则返回 true 表示默认为 on 状态。与 on 状态相反的状态为 off 状态，可以通过如下格式对日期和时间进行配置：

```
[
    { "type": "on",  "months": "1 TO 12", "week_days": "1 to 5", "hours": "9 to 18" },
    { "type": "off", "months": 10,        "month_days": 1 },
    { "type": "off", "datetime": "2017-11-02 09" }
]
```

日期和时间配置说明：

- 数组中每一项为一个配置，通过 type 域指定功能 on 或者 off；
- 支持对 months, month_days, week_days, hours, minutes 进行配置，如果不配置则不进行检测；
- 支持使用 datetime 来配置具体的时间段；
- 所有配置支持三种格式：(1) 单个整数；(2) 整数的数组；(3) 字符串 "m to n" 的格式，其中 m 和 n 为整数，mounths 的范围为 1 到 12，month_days 的范围为 1 到 31，week_days 的范围为 1 到 7，hours 的范围从 0 到 23，minutes 的范围从 0 到 59；
- 对于第一个配置项，如果 type 为 on 则其中不在当前配置项中的其他时间段均被置成 off，后续所有配置只影响对应的配置时间段，不会影响到其他时间段；
- 可以重复地 on 或者 off 某时间段，以最后一个配置项作为最终结果。

函数接收 1 个参数：

- list: 数组类型，list 内的元素必需为上述日期和时间配置格式。如果不指定配置，默认搜索全局变量 *__workdays__* 作为配置。

该函数代码示例如下：

```
list = [
    { "type": "on",  "months": "1 TO 12", "week_days": "1 to 5", "hours": "9 to 18" },
    { "type": "off", "months": 10,        "month_days": 1 },
    { "type": "off", "datetime": "2017-11-02 09" }
];
print(time(), "is on ?", check_datetime(list));

__now__ = "2017-11-02 09:00:00";
print(time(), "is on ?", check_datetime(list));

__now__ = "2017-10-01 08:00:00";
print(time(), "is on ?", check_datetime(list));
```

示例代码的输出内容如下：

```
2018-02-07 11:40:21.919+08:00 is on ? true
2017-11-02 09:00:00+08:00 is on ? false
2017-10-01 08:00:00+08:00 is on ? false
```

需要注意，如果没有配置日期与时间或者配置的时期与时间非法，则该函数返回 true 表示默认为 on 状态。

3.7 脚本执行函数

本小节介绍 RDL 中脚本执行相关函数。

3.7.1 call(name, args, ...)

本函数使用指定的参数来调用指定的函数，并返回被调函数的返回值。函数接收 n（n >= 1）个参数：

- name: 字符串类型，被调用函数名。
- args, ...: 任意类型，name 后面的所有参数都将被直接传递给被调函数。

该函数代码示例如下：

```
def myPrint(value) {
    print("value =", value);
}

call("myPrint", "hello world");
```

示例代码的输出内容如下：

```
value = hello world
```

3.7.2 call_builtin(name, args, ...)

本函数使用指定的参数来调用指定的函数，并返回被调函数的返回值。该函数只搜索 RDL 库提供的函数，而不调用用户自定义的函数。函数接收 n（n >= 1）个参数：

- name: 字符串类型，被调用函数名。
- args, ...: 任意类型，name 后面的所有参数都将被直接传递给被调函数。

该函数代码示例如下：

```
def myPrint(value) {
    print("value =", value);
}

call_builtin("myPrint", "hello world");
```

示例代码的输出内容如下：

```
2018-02-07 12:04:20 SQLAlert: [ERR] builtin 'myPrint' not found
in call_builtin(), in line 9 in file test.rule in task 'test'
```

3.7.3 call_list(list)

本函数按顺序调用指定的函数列表，将函数的返回值传递给下一个函数，并返回最后一个函数的返回值。函数列表的配置如下所示：

```
__funclist__ = [
    { "name": "func1", "args": { "arg1": "value1" } },
    { "name": "func2", "args": { "arg2": "value2" } }
];
```

上述函数列表配置中，name 为函数名，args 为函数的参数，参数可以是任意类型，如果多个参数的话，一般使用字典来传递。被调用的函数必需定义为如下格式：

```
def name(result, args); // 第一个参数为上一函数的返回值；第二个参数为配置的参数。
```

函数 call_list() 接收 1 个参数：

- list: 数组类型，其元素必需为上述配置格式。

该函数代码示例如下：

```
def print_zhang(result, args) {
    print("zhang result =", result);
    print("zhang args   =", args);
    return "zhang";
}
def print_wang(result, args) {
    print("wang result =", result);
    print("wang args   =", args);
}
def print_liang(result, args) {
    print("liang result =", result);
    print("liang args   =", args);
    return "liang";
}
funclist = [
    { "name": "print_zhang", args: { "value": "hello" } },
    { "name": "print_wang",  args: { "value": "world" } },
    { "name": "print_liang" }
];
print(call_list(funclist));
```

示例代码的输出内容如下：

```
zhang result = null
zhang args   = {"value":"hello"}
wang result = zhang
wang args   = {"value":"world"}
liang result = null
liang args   = null
liang
```

3.7.4 run(name)

本函数执行指定的函数，并将返回值保存到执行上下文中，如果上下文中设置了全局变量 *__sub_rules__* 则断续执行该变量所配置的子脚本。变量 *__sub_rules__* 必需是一个数组，数组内的元素是字符串格式，每一个元素为一个脚本文件名。通过该函数可以实现脚本关联执行或者分级执行的功能。函数接收 1 个参数：

- name: 字符串类型，被调用函数名。

该函数在执行子脚本时，为每个脚本执行分配一个线程，即对所配置的子脚本并行执行，并等待所有脚本的返回。限于篇幅，该函数暂不列出代码示例。

3.8 查询函数

本小节介绍 RDL 中查询 ES 相关函数，RDL 提供的查询函数使用 SQL 查询 ES，本文档中的示例只是对 SQL 简单的使用，SQLalert 中对 SQL 的支持请参阅 SQLAlert SQL 用户手册。

3.8.1 query(sql, filter)

本函数使用指定的 SQL 语句查询 ES，并返回查询结果。如果指定了过滤条件，则对查询的结果进行过滤，返回过滤后的结果。函数接收 2 个参数：

- sql: 字符串类型，SQL 查询语句。
- filter: 字符串类型，查询结果过滤表达式。

在使用 query() 函数前，需要在脚本中定义 ES 服务器地址信息，该函数搜索 *__es_host__* 和 *__es_host_query__* 变量作为 ES 集群的地址，地址信息为字符串类型，且包含端口号。

该函数代码示例如下：

```
1 __es_host__ = "192.168.0.122:9222";
2
3 result = query("SELECT sip, sport, dip, sport FROM 'tcp-*' LIMIT5");
4 pprint(result);
```

示例代码的输出内容如下：

```
[
    {
        "sip": "192.168.0.13",
        "sport": 56167,
        "dip": "192.30.253.124"
    },
    {
        "sip": "192.168.0.17",
        "sport": 59312,
        "dip": "180.149.132.165"
    },
    {
        "sip": "192.168.0.17",
        "sport": 59206,
        "dip": "106.39.162.37"
    },
    {
        "sip": "192.168.0.12",
        "sport": 65223,
        "dip": "64.233.189.139"
    },
    {
        "sip": "192.168.0.13",
        "sport": 56380,
        "dip": "74.125.204.113"
    }
]
```

为方便阅读，后续示例将使用 print_list() 函数打印查询结果，上述示例代码的结果使用 print_list() 函数打印出来如下：

```
{"dip":"192.30.253.124","sip":"192.168.0.13","sport":56167}
{"dip":"180.149.132.165","sip":"192.168.0.17","sport":59312}
{"dip":"106.39.162.37","sip":"192.168.0.17","sport":59206}
{"dip":"64.233.189.139","sip":"192.168.0.12","sport":65223}
{"dip":"74.125.204.113","sip":"192.168.0.13","sport":56380}
```

query() 函数支持对查询结果的二次过滤，通过参数 filter 将 RDL 支持表达式传递给该函数即可，示例代码如下：

```
__es_host__ = "192.168.0.122:9222";

sql = "SELECT sip, sport, dip, sport FROM 'tcp-*' LIMIT 5";
result = query(sql, "sip == '192.168.0.13'");
print_list(result);
```

示例代码的输出内容如下：

```
{"dip":"192.30.253.124","sip":"192.168.0.13","sport":56167}
{"dip":"74.125.204.113","sip":"192.168.0.13","sport":56380}
```

需要注意，过滤表达式中，支持 RDL 所有支持的表达式，但不支持函数调用。

3.8.2 query_avgby_num(sql, num, filter)

本函数使用指定的 SQL 语句查询 ES，并返回查询结果。如果指定了过滤条件，则对查询的结果进行过滤，返回过滤后的结果。该函数会将查询结果中所有的数值型字段，除以指定的数，然后再返回处理后的结果。其他方面与 query() 函数是一致的，query_avgby_num() 函数接收 3 个参数：

- sql: 字符串类型，SQL 查询语句。
- num: 数值类型，除数。
- filter: 字符串类型，查询结果过滤表达式。

请用户参阅 query() 函数示例，自行验证本函数。

3.8.3 query_avgby_field(sql, name, filter)

本函数使用指定的 SQL 语句查询 ES，并返回查询结果。如果指定了过滤条件，则对查询的结果进行过滤，返回过滤后的结果。该函数会将查询结果中所有的数值型字段，除以指定的字段值，然后再返回处理后的结果，指定的字段必需为数值。其他方面与 query() 函数是一致的，query_avgby_field() 函数接收 3 个参数：

- sql: 字符串类型，SQL 查询语句。
- name: 字符串类型，除数字段名。
- filter: 字符串类型，查询结果过滤表达式。

请用户参阅 query() 函数示例，自行验证本函数。

3.9 报警输出函数

本小节介绍 RDL 中报警输出函数。

3.9.1 alert_es(list)

本函数将指定数组中的数据写入到 ES 的索引中。脚本中通过全局变量 *_es_host_* 和 *_es_host_insert_* 来指定 ES 服务器的配置，通过全局变量 *_alert_index_* 指定输出的索引信息。

3.9.2 alert_email(list)

3.9.3 alert(list)