文档版本：ADC 3.0-V1.0

可编程脚本介绍

ADC产品提供了丰富的负载均衡参数可供选择，可以满足绝大多数用户的需求。但是对于需求比较复杂的用户，某些情况下负载均衡选项仍然不够灵活。

因此，ADC提供了可编程脚本的接口给用户，让用户自己来编写脚本控制流量，由用户自己来控制流量，更加方便、灵活。

当前，可编程脚本支持HTTP、HTTPS、TCP和RADIUS类型的VS。

脚本语法

脚本以lua语法为基础，即lua的语法可以在脚本中直接使用。

另外结合ADC的应用负载框架，提供了灵活、可扩展的可编程接口。目前我们的可编程脚本分为事件和接口，实现了HTTP、SSL、PERSIST、MBLB等相关的功能操作。

事件

当流量到达某个处理阶段时，会触发对应的事件，目前ADC可编程脚本中支持如下事件：

`HTTP_REQUEST`：当设备收到完整的client请求的HTTP header后触发。

`HTTP_REQUEST_DATA`：当设备收到完整的client发送的HTTP body后触发。

`SERVER_BEFORE_CONNECT`：当设备选出RS，且未连接RS之前触发。

`HTTP_RESPONSE`：当设备收到完整的RS回复的HTTP header后触发。

`HTTP_RESPONSE_DATA`：当设备收到一定大小的RS回复的HTTP body后触发，该大小通过脚本提供的函数进行设置，每次收集的数据长度大于等于指定长度，则触发一次该事件。

`CLIENT_DATA`：当设备收到TCP数据后触发，如果脚本处理结束，关闭TCP连接。

`SERVER_DATA`：当设备创建与RS后触发，依赖CLIENT_DATA事件。

`USER_REQUEST`：当设备每次与client建立TCP连接且收到TCP数据后触发一次；当设备每次收到client的UDP数据后触发一次。

接口

SLB

SLB.rs

- **功能**
  将当前请求调度到指定的地址与端口，该地址可以为任意地址，包括没有配置在ADC中的地址，使用此命令强制调度的流量会导致会话保持功能和统计功能失效。
  当VS中没有配置server-pool或server-pool中没有可用RS时，VS状态为不可用的，此时即使脚本中使用了rs命令，由于VS状态不可用，流量也会不通。
- **语法**
  `SLB.rs(ip[, port])`
  `ip`：将当前请求强制调度到该ip。
  `port`：将当前请求强制调度到该port，默认值为client请求的port。

> **成功返回true，失败返回false。**

> **适用事件：HTTP_REQUEST, HTTP_REQUEST_DATA。**

- **示例**

```
when HTTP_REQUEST {
    LOG.debug(LOG.DEBUG, string.format("enter HTTP_REQUEST event!\n"))
    SLB.rs("1.1.1.1", "666") --balance the traffic to 1.1.1.1:666
}
```

```
when HTTP_REQUEST {
    LOG.debug(LOG.DEBUG, string.format("enter HTTP_REQUEST event!\n"))
    SLB.rs("1.1.1.1")
    --when client request http://x.x.x.x:123, balance the traffic to 1.1.1.1:123
}
```

SLB.pool

- **功能**
  在HTTP_REQUEST, HTTP_REQUEST_DATA, USER_REQUEST事件中，将当前请求调度到指定的server-pool，指定server-pool中的RS，或使用server-pool的均衡算法选出一个可用的RS，使用此命令不会产生会话保持表项。
  在CLIENT_DATA, SERVER_DATA事件中，返回指定server-pool中的RS。
  当VS中没有配置server-pool或server-pool中没有可用RS时，VS状态为不可用的，此时即使脚本中使用了pool命令，由于VS状态不可用，流量也会不通。
- **语法**
  `ok,rs = SLB.pool([pool_name][, rs_name])`
  `pool_name`：将当前请求强制调度到名字为pool_name的server_pool，如果不指定，则使用当前VS下默认的server-pool。
  `rs_name`：指定server-pool中的RS，如果不指定，则使用指定server-pool的均衡算法选择一个RS。

> **成功返回true, rs：{name，ip，port}。失败返回false,rs为nil。**
> **适用事件：HTTP_REQUEST, HTTP_REQUEST_DATA, CLIENT_DATA, SERVER_DATA, USER_REQUEST。**
> **注意：此接口不能和PERSIST.persist接口同时调用。如果先调用了PERSIST.persist，那么调用SLB.pool接口会返回失败。**

- **示例**

```
when HTTP_REQUEST {
    SLB.pool("pool_1", "rs_1") --balance traffic to rs_1 in pool_1
}
```

```
when HTTP_REQUEST {
    SLB.pool("pool_1") --balance traffic to pool_1, auto select RS
}
```

```
when CLIENT_DATA {
    local ok,rs = SLB.pool("pool_1") --balance traffic to pool_1, return RS
    if ok then
        LOG.debug(LOG.DEBUG, string.format("rs ip %d, name %s\n", rs.ip,
rs.name))
    end
}
```

SLB.rs_list(pool_name, [is_available])

- **功能**
  获取RS的配置对象集合。
- **语法**
  `SLB.rs_list(pool_name, [is_available])`
  `pool_name`：将当前请求强制调度到名字为pool_name的server_pool。
  `is_available`：为false，获取当前pool的所有RS对象集合；当为true，获取pool的所有有效的RS对象集合。

> **返回所有有效节点的数组{{ name，ip，port },{ name，ip，port },...}**
> **适用于所有事件。**

- **示例**

```
when CLIENT_DATA {
    local rs_set = SLB.rs_list("pool_1")
    for _,rs in pairs(rs) do
        LOG.debug(LOG.DEBUG, string.format("rs ip %s, port %d\n", rs.ip,
rs.port))
    end
}
```

SLB.rs_status(pool_name, rs_name)

- **功能**
  获取SP中RS的状态。
- **语法**
  SLB.rs_status(pool_name, rs_name)
  `pool_name`：指定server-pool的名字。
  `rs_name`：指定RS的名字。

> **失败返回nil，err为失败原因。**
> **RS可用返回available**
> **RS不可用返回unavailable**
> **适用于所有事件。**

- **示例**

```
when CLIENT_DATA {
    local status,err = SLB.rs_status(pool_name, rs_name)
    if status then
        LOG.debug(LOG.DEBUG, string.format("rs %s, status %s\n", rs_name,
status))
    else
        LOG.debug(LOG.DEBUG, string.format("get rs status faild, err %s\n",
err))
    end
}
```

SLB.get_pool([true|false])

- **功能**
  获取为当前请求服务的VS配置的server-pool。
- **语法**
  ok,rs = SLB.get_pool([true|false])
  `true`：返回当前访问的vs配置的server_pool。
  `false或nil`：返回当前访问的vs配置的backup_server_pool。

> **仅适用于HTTP请求方向，TCP事件。**

- **示例**

```
when CLIENT_DATA {
    local sp_name,err = SLB.get_pool(true)
    if sp_name then
        LOG.debug(LOG.DEBUG, string.format("vs server pool name %s\n", sp_name))
    else
        LOG.debug(LOG.DEBUG, string.format("get vs server pool name err:%s\n",
err))
    end
}
```

HTTP

HTTP.method

HTTP.method.get

- **功能**
  获取当前HTTP请求的方法，例如"GET"、"POST"等。
- **语法**
  `HTTP.method.get()`

> **成功返回HTTP方法，类型为字符串，失败返回nil。**

> **适用事件：适用于HTTP事件。**

- **示例**

```
when HTTP_REQUEST {
    local method = HTTP.method.get()
    LOG.debug(LOG.DEBUG, string.format("the method is %s\n", method))
}
```

HTTP.uri

HTTP.uri.get

- **功能**
  获取当前HTTP请求的uri。
- **语法**
  `HTTP.uri.get()`

> **成功返回HTTP请求的uri，类型为字符串，失败返回nil。**
>
> **适用事件：适用于HTTP事件。**

- **示例**

```
when HTTP_REQUEST {
    local uri = HTTP.uri.get()
    LOG.debug(LOG.DEBUG, string.format("the uri is %s\n", uri))
}
```

> 对请求 http://1.1.1.1/index.html ， uri为 /index.html。

HTTP.uri.set

- **功能**
  设置当前HTTP请求的uri，但不进行重定向跳转。
- **语法**
  `HTTP.uri.set(value)`
  `value`：要设置的uri字符串。

> **成功返回true，失败返回false。**
>
> **适用事件：HTTP_REQUEST, HTTP_REQUEST_DATA。**

- **示例**

```
when HTTP_REQUEST {
    local ret = HTTP.uri.set("/test.html")
    if ret then
        LOG.debug(LOG.DEBUG, string.format("set uri success\n"))
    end
}
```

HTTP.uri.get_args

- **功能**
  获取当前HTTP请求的uri中的参数。
- **语法**
  `HTTP.uri.get_args()`

> **成功返回HTTP请求的uri中的参数，类型为lua table，当参数中的key的value为一个值时，value为字符串，key的value有多个值时，value以table的形式存在，失败或不存在返回nil。**

> **适用事件：适用于HTTP事件。**

- **示例**

```
when HTTP_REQUEST {
    local arg_table = HTTP.uri.get_args()
    if arg_table then
        for key, value in pairs(arg_table) do
            if type(value) == "table" then
                LOG.debug(LOG.DEBUG, string.format("key: %s, value: %s",
                                                   key, table.concat(val,
","))
            else
                LOG.debug(LOG.DEBUG, string.format("key: %s, value: %s",
                                                   key, value))
            end
        end
    end
}
```

> 对uri /test?a=1&b=2，上述代码输出为：
> key: a, value: 1
> key: b, value: 2
>
> 对uri /test/?a=1&b=2&b=3，上述代码输出为：
> key: a, value: 1
> key: b, value: 2，3

HTTP.version

HTTP.version.get

- **功能**
  获取当前HTTP请求的HTTP协议版本。
- **语法**
  `HTTP.version.get()`

> **成功返回HTTP协议版本，类型为字符串，失败返回nil。**

> **适用事件：适用于HTTP事件。**

- **示例**

```
when HTTP_REQUEST {
    local version = HTTP.version.get()
    if version then
        LOG.debug(LOG.DEBUG, string.format("HTTP version is %s\n", version))
    end
}
```

HTTP.header

HTTP.header.get

- **功能**
  根据当前流量方向获取HTTP的header。
- **语法**
  `HTTP.header.get([name])`
  `name`：header中属性的名字，如果不指定，则获取整个header。

> **成功返回获取的属性值，当该属性有一个值时，返回类型为字符串；当该属性有多个值时，返回类型为lua table；当获取整个header时，返回类型为lua table，失败或不存在返回nil。**

> **适用事件：适用于HTTP事件。**

- **示例**

```
when HTTP_REQUEST {--get host from http header in request traffic
    local host = HTTP.header.get("host")
    if host then
        --host is a single value attribute, so no need to judge if type(host) is
table
        LOG.debug(LOG.DEBUG, string.format("HTTP host is %s\n", host))
    end
}
```

```
when HTTP_REQUEST {--get content-type from http header in response traffic
    local content_type = HTTP.header.get("content-type")
    if content_type then
        --content_type is a single value attribute,
        --so no need to judge if type(content_type) is table
        LOG.debug(LOG.DEBUG, string.format("HTTP content_type is %s\n",
                                           content_type))
    end
}
```

```
when HTTP_REQUEST {--get whole http header from request traffic, and print it
    local header = HTTP.header.get()
    if header then
        for key, value in pairs(header) do
            if type(value) == "table" then
                LOG.debug(LOG.DEBUG, string.format("key:%s, value:%s", key,
                                                   table.concat(value, ",")))
            else
                LOG.debug(LOG.DEBUG, string.format("key:%s, value:%s",
                                                   key, value))
            end
        end
    end
}
```

HTTP.header.insert

- **功能**
  根据当前流量方向，向当前HTTP报文的header中插入指定内容，如果当前header中包含要插入的属性，则将要插入的值放到最后。
- **语法**
  `HTTP.header.insert(header_name, header_value)`
  `header_name`：要插入在header中的属性的名字。
  `header_value`：要插入在header中的属性的值。

  > **成功返回true，失败返回false。**

  > **适用事件：适用于HTTP事件。**

- **示例**

```
when HTTP_REQUEST {--insert a attribute in http header in request traffic
    local ret = HTTP.header.insert("test-script", "success")
    if ret then
        LOG.debug(LOG.DEBUG, string.format("insert header success\n"))
    end
}
```

HTTP.header.remove

- **功能**
  根据当前流量方向，在当前HTTP报文的header中删除指定内容，如果指定的属性有多个值，则多个值都会被删除。
- **语法**
  `HTTP.header.remove(header_name)`
  `header_name`：要插入在header中的属性的名字。

  > **成功返回true，失败返回false。**

  > **适用事件：适用于HTTP事件。**

- **示例**

```
when HTTP_RESPONSE {--delete a attribute in http header in response traffic
    local ret = HTTP.header.remove("last-modified")
    if ret then
        LOG.debug(LOG.DEBUG,
                  string.format("delete last-modified in header success\n"))
    end
}
```

HTTP.header.replace

- **功能**
  根据当前流量方向，在当前HTTP报文的header中替换指定属性的内容，如果指定的属性有多个值，则多个值都会被替换；如果当前header中不包括指定的属性，则不做操作。

- **语法**
  `HTTP.header.replace(header_name, header_value)`
  `header_name`：要替换的header中的属性的名字。
  `header_value`：要替换的header中的属性的值。

  > **成功返回true，失败返回false。**

  > **适用事件：适用于HTTP事件。**

- **示例**

```
when HTTP_RESPONSE {--replace a attribute in http header in request traffic
    local ret = HTTP.header.replace("user-agent", "script-test")
    if ret then
        LOG.debug(LOG.DEBUG,
                  string.format("replace user-agent in header success\n"))
    end
}
```

HTTP.status

HTTP.status.get

- **功能**
  获取当前HTTP应答的状态码。
- **语法**
  `HTTP.status.get()`

  > **成功返回HTTP应答状态码，类型为lua number，失败返回nil。**

  > **适用事件：HTTP_RESPONSE, HTTP_RESPONSE_DATA。**

- **示例**

```
when HTTP_RESPONSE {
    local status = HTTP.status.get()
    if status == 200 then
        LOG.debug(LOG.DEBUG, string.format("request success\n"))
    end
}
```

HTTP.body

HTTP.body.get

- **功能**
  根据当前流量方向，获取HTTP请求或应答方向的body。
- **语法**
  `HTTP.body.get()`

  > **成功返回HTTP的body，类型为字符串，失败或body不存在返回nil。**

  > **适用事件：HTTP_REQUEST_DATA，HTTP_RESPONSE_DATA。**

> 在HTTP_REQUEST_DATA事件中使用时，获取body的全部内容；在HTTP_RESPONSE_DATA中使用时，需要先使用HTTP.collect()函数指定收集的body大小，返回内容为当前收集到的body数据。

- **示例**

```
when HTTP_REQUEST_DATA {
    local body = HTTP.body.get()
    if body then
        LOG.debug(LOG.DEBUG, string.format("body in request is %s\n", body))
    end
}

when HTTP_RESPONSE {
    local ret = HTTP.collect(100)
    if ret then
        LOG.debug(LOG.DEBUG, string.format("set collect success\n"))
    end
}

when HTTP_RESPONSE_DATA {
    local body = HTTP.body.get()
    if body then
        LOG.debug(LOG.DEBUG,
                  string.format("current collect body in response is %s\n",
                                body))
    end
}
```

HTTP.body.set

- **功能**
  根据当前流量方向，设置HTTP请求或应答方向的body。
- **语法**
  `HTTP.body.set(body)`
  `body`：要设置的body内容，类型为字符串。

> 成功返回true，失败返回false。

> 适用事件：HTTP_REQUEST_DATA，HTTP_RESPONSE_DATA。

- **示例**

```
when HTTP_REQUEST_DATA {
    local body = HTTP.body.get()
    if body then
        local ret = HTTP.body.set("test-script")
        if ret then
            LOG.debug(LOG.DEBUG, string.format("set body in request success\n"))
        end
    end
}
```

```
when HTTP_RESPONSE {
    local ret = HTTP.collect(100)
    if ret then
        LOG.debug(LOG.DEBUG, string.format("set collect success\n"))
    end
}

when HTTP_RESPONSE_DATA {--change response body to lower
    local body = HTTP.body.get()
    if body then
        local lower_str = string.lower(body)
        local ret = HTTP.body.set(lower_str)
        if ret then
            LOG.debug(LOG.DEBUG,
                      string.format("set body in response success\n"))
        end
    end
}
```

HTTP.body.replace

- **功能**
  根据当前流量方向，对HTTP请求或响应方向的body中内容进行替换。
- **语法**
  `HTTP.body.replace(original, target)`
  `body`：要设置的body内容，类型为字符串。

> **成功返回true，失败返回false。**

> **适用事件：HTTP_REQUEST_DATA，HTTP_RESPONSE_DATA。**

- **示例**

```
when HTTP_REQUEST_DATA {
    local ret = HTTP.body.replace("hello", "test-script")
    if ret then
        LOG.debug(LOG.DEBUG, string.format("replace body in request success\n"))
    end
}

when HTTP_RESPONSE {
    local ret = HTTP.collect(100)
    if ret then
        LOG.debug(LOG.DEBUG, string.format("set collect success\n"))
    end
}

when HTTP_RESPONSE_DATA {
    local ret = HTTP.body.replace("world", "script")
    if ret then
        LOG.debug(LOG.DEBUG,
                  string.format("set body in response success\n"))
    end
}
```

HTTP.cookie

HTTP.cookie.get

- **功能**
  根据当前流量方向，获取HTTP请求或应答方向的cookie的名称，该函数只支持获取cookie的name-value中的value值，不支持获取属性值。
- **语法**
  `HTTP.cookie.get(name)`
  `name`：要获取的cookie名称。

> **成功返回cookie的值，类型为字符串，失败或不存在返回nil。**

> **适用事件：适用于HTTP事件。**

- **示例**

```
when HTTP_RESPONSE {
    local cookie_value = HTTP.cookie.get("user1")
    if cookie_value then
        LOG.debug(LOG.DEBUG, string.format("cookie_value in response cookie is
%s\n",
                                           cookie_value))
    end
}
```

> 对于cookie：Set-Cookie: test=2345;Domain=aaaa;Path=/test，该函数仅获取test=12345这一键值对，不支持之后的属性值的获取。

HTTP.cookie.get_attr

- **功能**
  获取应答方向的set-cookie中的属性值。
- **语法**
  `HTTP.cookie.get_attr(cookie_name, attr_name)`
  `cookie_name`：要获取的cookie的名称。
  `attr_name`：要获取的属性的名称。

> **成功返回cookie的内容，类型为字符串，失败或不存在返回nil。**

> **适用事件：HTTP_RESPONSE, HTTP_RESPONSE_DATA。**

- **示例**

```
when HTTP_RESPONSE {
    local cookie_domain = HTTP.cookie.get("user1", "Domain")
    if cookie_domain then
        LOG.debug(LOG.DEBUG,
                  string.format("domain of cookie user1 in response cookie is
%s\n",
                                cookie_domain))
    end
}
```

> 对于cookie：Set-Cookie: user1=2345;Domain=aaaa;Path=/test，上述代码输出为 domain of cookie user1 in response cookie is aaaa

HTTP.keepalive

HTTP.keepalive.get

- **功能**
  获取当前HTTP连接是否是keepalive的。
- **语法**
  `HTTP.keepalive.get()`

  > **如果当前连接是keepalive的，返回true，否则返回false。**

  > **适用事件：适用于HTTP事件。**

- **示例**

```
when HTTP_REQUEST {
    local is_keepalive = HTTP.keepalive.get()
    if is_keepalive then
        LOG.debug(LOG.DEBUG,
                  string.format("current connection is a keepalive
connection\n"))
    end
}
```

HTTP.request

HTTP.request.get

- **功能**
  获取当前HTTP连接，请求方向的整个header。
- **语法**
  `HTTP.request.get()`

  > **成功返回整个header的lua table，失败返回nil。**

  > **适用事件：适用于HTTP事件。**

- **示例**

```
when HTTP_RESPONSE {--print request header in response event
    local header = HTTP.request.get()
    if header then
        for key, value in pairs(header) do
            if type(value) == "table" then
                LOG.debug(LOG.DEBUG, string.format("key:%s, value:%s", key,
                                                   table.concat(value, ",")))
            else
                LOG.debug(LOG.DEBUG, string.format("key:%s, value:%s",
                                                   key, value))
            end
        end
```

```
    end
}
```

HTTP.response

HTTP.response.get

- **功能**
  获取当前HTTP连接，响应方向的整个header。
- **语法**
  `HTTP.response.get()`

  > **成功返回整个header的lua table，失败返回nil。**

  > **适用事件：HTTP_RESPONSE, HTTP_RESPONSE_DATA。**

- **示例**

```
when HTTP_RESPONSE {
    --print response header in response event; here, equal to HTTP.header.get()
    local header = HTTP.response.get()
    if header then
        for key, value in pairs(header) do
            if type(value) == "table" then
                LOG.debug(LOG.DEBUG, string.format("key:%s, value:%s", key,
                                                   table.concat(value, ",")))
            else
                LOG.debug(LOG.DEBUG, string.format("key:%s, value:%s",
                                                   key, value))
            end
        end
    end
}
```

HTTP.scheme

HTTP.scheme.get

- **功能**
  获取当前HTTP连接的scheme。
- **语法**
  `HTTP.scheme.get()`

  > **成功返回当前连接的scheme，为字符串"http"或"https"，失败返回nil。**

  > **适用事件：适用于HTTP事件。**

- **示例**

```
when HTTP_RESPONSE {
    --print http scheme, for http connect, scheme is "http", for https ,scheme
is "https"
    local scheme = HTTP.scheme.get()
    if scheme then
        LOG.debug(LOG.DEBUG, string.format("key:%s, value:%s",
                                                    key, value))
    end
}
```

HTTP.redirect

- **功能**
  向客户端发送重定向应答。
- **语法**
  `HTTP.redirect(url[, status])`
  `url`：要重定向到的新url地址。
  `status`：返回给客户端的状态码，类型为数字，当前支持301、302和307，默认为302。

  > **成功返回true，失败返回false。**

  > **适用事件：HTTP_REQUEST, HTTP_REQUEST_DATA。**

- **示例**

```
when HTTP_REQUEST {
    local url = HTTP.url.get()
    if url == "/" then
        HTTP.redirect("/test-script.html", 301)
    end
}
```

HTTP.close

- **功能**
  结束当前HTTP连接。
- **语法**
  `HTTP.close()`

  > **成功返回true，失败返回false。**

  > **适用事件：适用于HTTP事件。**

- **示例**

```
when HTTP_REQUEST { --close connection from whose user-agent is "test-script"
    local user-agent = HTTP.header.get("user-agent")
    if user-agent == "test-script" then
        HTTP.close()
    end
}
```

HTTP.bypass

- **功能**
  提供bypass功能，如果调用了此函数，对于当前连接，后续的脚本事件不再执行。
- **语法**
  `HTTP.bypass()`

  > **成功返回true，失败返回false。**

  > **适用事件：适用于HTTP事件。**

- **示例**

```
when HTTP_REQUEST { -- if user-agent is "test-script", do not execute following
event
    local user-agent = HTTP.header.get("user-agent")
    if user-agent == "test-script" then
        HTTP.bypass()
    end
}

when HTTP_RESPONSE {
    --this event won't be execute when user-agent from request is "test-script"
    LOG.debug(LOG.DEBUG, string.format("enter event HTTP_RESPONSE"))
}
```

HTTP.collect

- **功能**
  收集一定长度的HTTP响应中body的内容，并触发HTTP_RESPONSE_DATA事件，当前HTTP_RESPONSE_DATA事件只有调用了此函数后，才有可能会被触发。

  触发HTTP_RESPONSE_DATA事件时，会根据当前收到的报文内容进行拼接并判断长度，所以收集到的body长度会大于等于指定的长度；触发完该事件后，如果body继续传输，则收集够指定长度后，会再次触发该事件。
- **语法**
  `HTTP.collect(length)`
  `length`： 要收集的body字符串的长度，单位为B。

  > **成功返回true，失败返回false。**

  > **适用事件：HTTP_RESPONSE。**

- **示例**

```
when HTTP_RESPONSE {
    local ret = HTTP.collect(100)
    if ret then
        LOG.debug(LOG.DEBUG, string.format("set collect success\n"))
    end
}

when HTTP_RESPONSE_DATA {
```

```
    local body = HTTP.body.get()
    if body then
        LOG.debug(LOG.DEBUG,
                  string.format("current collect body in response is %s\n",
                                body))
    end
}
```

HTTP.respond

- **功能**
  向client端直接应答指定内容。
- **语法**
  `HTTP.respond(string[, status_code])`
  `string`： 要回复的内容，类型为字符串。
  `status_code`：向客户端回应时的状态码，如不指定，默认为200。

  > **成功返回true，失败返回false。**

  > **适用事件：HTTP_REQUEST, HTTP_REQUEST_DATA。**

- **示例**

```
when HTTP_REQUEST {
    local host = HTTP.header.get("host")
    if host == "www.google.com" then
        HTTP.respond("404 not found", 404)
    end
}
```

SSL

SSL.version

SSL.version.get

- **功能**
  获取当前连接SSL版本信息，即client与ADC设备协商的版本。
- **语法**
  `SSL.version.get()`

  > **成功返回ssl版本，类型为字符串，失败或不存在返回nil。**

  > **适用事件：适用于HTTP事件和USER_REQUEST事件。**

- **示例**

```
when HTTP_REQUEST {
    local ssl_version = SSL.version.get()
    if ssl_version then
        LOG.debug(LOG.DEBUG, string.format("ssl version is %s\n", ssl_version))
    end
}
```

> 当前支持的版本为"TLSv1"、"TLSv1.1"、"TLSv1.2"、"SSLv2"、"SSLv3"。

SSL.session

SSL.session.id.get

- **功能**
  获取当前连接中，SSL协商出来的session的id。
- **语法**
  `SSL.session.id.get()`

  > **成功返回session的id，类型为数字，失败或不存在返回nil。**

  > **适用事件：适用于HTTP事件和USER_REQUEST事件。**
- **示例**

```
when HTTP_REQUEST {
    local ssl_session_id = SSL.session.id.get()
    if ssl_session_id then
        LOG.debug(LOG.DEBUG, string.format("ssl session id is %s\n",
                                           ssl_session_id))
    end
}
```

SSL.session.ticket_id.get

- **功能**
  获取当前连接中，SSL协商出来的session的ticket id。
- **语法**
  `SSL.session.ticket_id.get()`

  > **成功返回session的ticket id，类型为数字，失败或不存在返回nil。**

  > **适用事件：适用于HTTP事件和USER_REQUEST事件。**
- **示例**

```
when HTTP_REQUEST {
    local ssl_session_ticket_id = SSL.session.ticket_id.get()
    if ssl_session_ticket_id then
        LOG.debug(LOG.DEBUG, string.format("ssl session ticket id is %s\n",
                                           ssl_session_ticket_id))
    end
}
```

SSL.client_verify

SSL.client_verify_result.string.get

- **功能**
  获取当前连接中，SSL对client证书的验证结果。
- **语法**
  `SSL.client_verify_result.string.get()`

  > **根据证书验证结果返回字符串"*SUCCESS*"或"*FAILED:<reason>*"，若证书不存在，返回"NONE"。**

  > **适用事件：适用于HTTP事件和USER_REQUEST事件。**

- **示例**

```
when HTTP_REQUEST {
    local verify_result_string = SSL.client_verify_result.string.get()
    if verify_result_string then
        LOG.debug(LOG.DEBUG, string.format("ssl client verify result is %s\n",
                                           verify_result_string))
    end
}
```

SSL.client_verify_result.code.get

- **功能**
  获取当前连接中，SSL对client证书的验证结果。
- **语法**
  `SSL.client_verify_result.code.get()`

  > **成功返回证书认证错误码，类型为数字，失败或不存在返回nil。**

  > **适用事件：适用于HTTP事件和USER_REQUEST事件。**

- **示例**

```
when HTTP_REQUEST {
    local verify_result_code = SSL.client_verify_result.code.get()
    if verify_result_code then
        LOG.debug(LOG.DEBUG, string.format("ssl client verify result is %s\n",
                                           verify_result_code))
    end
}
```

> 状态码与字符串对应关系如下

code	string
nil	NONE
0	SUCCESS
2	FAILED:unable to get issuer certificate
3	FAILED:unable to get certificate CRL
4	FAILED:unable to decrypt certificate's signature
5	FAILED:unable to decrypt CRL's signature
6	FAILED:unable to decode issuer public key
7	FAILED:certificate siganature failure
8	FAILED:CRL signature failure
9	FAILED:certificate is not yet valid
10	FAILED:certificate has expired
11	FAILED:CRL is not yet valid
12	FAILED:CRL has expired
13	FAILED:format error in certificate’s notBefore field
14	FAILED:format error in certificate’s notAfter field
15	FAILED:format error in CRL’s lastUpdate field
16	FAILED:format error in CRL’s nextUpdate field
17	FAILED:out of memory
18	FAILED:self signed certificate
19	FAILED:self signed certificate in certificate chain
20	FAILED:unable to get local issuer certificate
21	FAILED:unable to verify the first certificate
22	FAILED:certificate chain too long
23	FAILED:certificate revoked
24	FAILED:invalid CA certificate
25	FAILED:path length constraint exceeded
26	FAILED:unsupported certificate purpose
27	FAILED:certificate not trusted
28	FAILED:certificate rejected
29	FAILED:subject issuer mismatch

code	string
30	FAILED:authority and subject key identifier mismatch
31	FAILED:authority and issuer serial number mismatch
32	FAILED:key usage does not include certificate signing
33	FAILED:unable to get CRL issuer certificate
34	FAILED:unhandled critical extension
35	FAILED:key usage does not include CRL signing
36	FAILED:unhandled critical CRL extension
37	FAILED:invalid non-CA certificate (has CA markings)
38	FAILED:proxy path length constraint exceeded
39	FAILED:key usage does not include digital signature
40	FAILED:proxy certificates not allowed, please set the appropriate flag
41	FAILED:invalid or inconsistent certificate extension
42	FAILED:invalid or inconsistent certificate policy extension
43	FAILED:no explicit policy
44	FAILED:Different CRL scope
45	FAILED:Unsupported extension feature
46	FAILED:RFC 3779 resource not subset of parent's resources
47	FAILED:permitted subtree violation
48	FAILED:excluded subtree violation
49	FAILED:name constraints minimum and maximum not supported
50	FAILED:application verification failure
51	FAILED:unsupported name constraint type
52	FAILED:unsupported or invalid name constraint syntax
53	FAILED:unsupported or invalid name syntax
54	FAILED:CRL path validation error
56	FAILED:Suite B: certificate version invalid
57	FAILED:Suite B: invalid public key algorithm
58	FAILED:Suite B: invalid ECC curve
59	FAILED:Suite B: invalid signature algorithm
60	FAILED:Suite B: curve not allowed for this LOS

code	string
61	FAILED:Suite B: cannot sign P-384 with P-256
62	FAILED:Hostname mismatch
63	FAILED:Email address mismatch
64	FAILED:IP address mismatch
65	FAILED:Invalid certificate verification context
66	FAILED:Issuer certificate lookup error
67	FAILED:proxy subject name violation

SSL.tlsext

SSL.tlsext.sni.get

- **功能**
  获取当前连接中，client端SSL请求中的sni信息。
- **语法**
  `SSL.tlsext.sni.get()`

  > **成功返回sni内容信息，类型为字符串，失败或不存在返回nil。**

  > **适用事件：适用于HTTP事件和USER_REQUEST事件。**
- **示例**

```
when HTTP_REQUEST {
    local server_name = SSL.tlsext.sni.get()
    if server_name then
        LOG.debug(LOG.DEBUG, string.format("ssl client sni is %s\n",
                                           server_name))
    end
}
```

SSL.tlsext.alpn.get

- **功能**
  获取ssl连接中，client hello报文携带的ALPN信息。
- **语法**
  `SSL.tlsext.alpn.get()`

  > **成功返回client hello携带的alpn字段，类型为字符串，失败或不存在返回nil。**

  > **适用事件：适用于USER_REQUEST事件。**
- **示例**

```
when USER_REQUEST {
    local alpn = SSL.tlsext.alpn.get()
    if alpn then
        LOG.debug(LOG.DEBUG, string.format("ssl client alpn is %s\n", alpn))
    end
}
```

SSL.cipher

SSL.cipher.name.get

- **功能**
  获取当前连接中，SSL协商出来的cipher名称。
- **语法**
  `SSL.cipher.name.get()`

  > **成功返回cipher名称，类型为字符串，失败或不存在返回nil。**

  > **适用事件：适用于HTTP事件和USER_REQUEST事件。**

- **示例**

```
when HTTP_REQUEST {
    local cipher_name = SSL.cipher.name.get()
    if cipher_name then
        LOG.debug(LOG.DEBUG, string.format("cipher is %s\n", cipher_name))
    end
}
```

SSL.cipher.id.get

- **功能**
  获取当前连接中，SSL协商出来的cipher的id。
- **语法**
  `SSL.cipher.id.get()`

  > **成功返回cipher的id，类型为数字，失败或不存在返回nil。**

  > **适用事件：适用于HTTP事件和USER_REQUEST事件。**

- **示例**

```
when HTTP_REQUEST {
    local cipher_id = SSL.cipher.name.get()
    if cipher_id then
        LOG.debug(LOG.DEBUG, string.format("cipher id is %s\n", cipher_id))
    end
}
```

> cipher的id与name对应关系如下

id	name
0xC0,0x30	ECDHE-RSA-AES256-GCM-SHA384
0xC0,0x2C	ECDHE-ECDSA-AES256-GCM-SHA384
0xC0,0x28	ECDHE-RSA-AES256-SHA384
0xC0,0x24	ECDHE-ECDSA-AES256-SHA384
0xC0,0x14	ECDHE-RSA-AES256-SHA
0xC0,0x0A	ECDHE-ECDSA-AES256-SHA
0x00,0x9F	DHE-RSA-AES256-GCM-SHA384
0x00,0x6B	DHE-RSA-AES256-SHA256
0x00,0x39	DHE-RSA-AES256-SHA
0x00,0x88	DHE-RSA-CAMELLIA256-SHA
0xC0,0x32	ECDH-RSA-AES256-GCM-SHA384
0xC0,0x2E	ECDH-ECDSA-AES256-GCM-SHA384
0xC0,0x2A	ECDH-RSA-AES256-SHA384
0xC0,0x26	ECDH-ECDSA-AES256-SHA384
0xC0,0x0F	ECDH-RSA-AES256-SHA
0xC0,0x05	ECDH-ECDSA-AES256-SHA
0x00,0x9D	AES256-GCM-SHA384
0x00,0x3D	AES256-SHA256
0x00,0x35	AES256-SHA
0x00,0x84	CAMELLIA256-SHA
0xC0,0x2F	ECDHE-RSA-AES128-GCM-SHA256
0xC0,0x2B	ECDHE-ECDSA-AES128-GCM-SHA256
0xC0,0x27	ECDHE-RSA-AES128-SHA256
0xC0,0x23	ECDHE-ECDSA-AES128-SHA256
0xC0,0x13	ECDHE-RSA-AES128-SHA
0xC0,0x09	ECDHE-ECDSA-AES128-SHA
0x00,0x9E	DHE-RSA-AES128-GCM-SHA256
0x00,0x67	DHE-RSA-AES128-SHA256
0x00,0x33	DHE-RSA-AES128-SHA
0x00,0x9A	DHE-RSA-SEED-SHA

id	name
0x00,0x45	DHE-RSA-CAMELLIA128-SHA
0xC0,0x31	ECDH-RSA-AES128-GCM-SHA256
0xC0,0x2D	ECDH-ECDSA-AES128-GCM-SHA256
0xC0,0x29	ECDH-RSA-AES128-SHA256
0xC0,0x25	ECDH-ECDSA-AES128-SHA256
0xC0,0x04	ECDH-ECDSA-AES128-SHA
0x00,0x9C	AES128-GCM-SHA256
0x00,0x3C	AES128-SHA256
0x00,0x2F	AES128-SHA
0x00,0x96	SEED-SHA
0x00,0x41	CAMELLIA128-SHA
0xC0,0x12	ECDHE-RSA-DES-CBC3-SHA
0xC0,0x08	ECDHE-ECDSA-DES-CBC3-SHA
0x00,0x16	EDH-RSA-DES-CBC3-SHA
0xC0,0x0D	ECDH-RSA-DES-CBC3-SHA
0xC0,0x03	ECDH-ECDSA-DES-CBC3-SHA
0x00,0x0A	DES-CBC3-SHA
0x00,0x07	IDEA-CBC-SHA
0xC0,0x11	ECDHE-RSA-RC4-SHA
0xC0,0x07	ECDHE-ECDSA-RC4-SHA
0xC0,0x0C	ECDH-RSA-RC4-SHA
0xC0,0x02	ECDH-ECDSA-RC4-SHA
0x00,0x05	RC4-SHA
0x00,0x04	RC4-MD5

SSL.client_hello

SSL.client_hello.highest_version.get

- **功能**
  获取ssl连接中，从client hello中获取客户端支持的最高版本。
- **语法**
  `SSL.client_hello.highest_version.get()`

> **成功返回client hello中获取客户端支持的最高版本号，类型为字符串，失败或不存在返回nil。**

> **适用事件：适用于USER_REQEUST事件。**

- **示例**

```
when USER_REQUEST {
    local version = SSL.client_hello.highest_version.get()
    if version then
        LOG.debug(LOG.DEBUG, string.format("client support highest version is
%s\n", version))
    end
}
```

X509

X509.issuer

X509.issuer.dn.get

- **功能**
  获取当前连接client证书链中某一级证书的颁发者信息。
- **语法**
  `X509.issuer.dn.get(separator[, reverse[, cert_index]])`
  `separator`：各dn字段之间的分隔符，nil或不填表示默认分隔符，为逗号","。
  `reverse`：是否逆序获取，默认为false，即按rfc标准从最后一个元素向前读取，如果为true，则按相反顺序读取。
  `cert_index`：指定证书链中的某一级证书，0或不填表示证书链的第一个证书，一般为client的证书，1表示证书链的第二个证书，以此类推。

> **成功返回证书的颁发者信息，类型为字符串，按默认或指定字符串分隔，失败或不存在返回nil。**

> **适用事件：适用于HTTP事件和USER_REQUEST事件。**

- **示例**

```
when HTTP_REQUEST {
    local issuer_dn = X509.issuer.dn.get("|")
    if issuer_dn then
        LOG.debug(LOG.DEBUG, string.format("client issuer_dn is %s\n",
                                           issuer_dn))
    end
}
```

> 此代码输出可能为
> "client issuer_dn is C =ca_c|ST=ca_st|L=ca_l|O=ca_o|OU=ca_ou|CN=user_cn"

X509.issuer.dn.get_field

- **功能**
  获取当前连接client证书链中某一级证书颁发者的具体字段。

- **语法**

  `X509.issuer.dn.get_field(key[, cert_index])`

  `key`：具体dn字段的名字，如"emailAddress"。

  `cert_index`：指定证书链中的某一级证书，0或不填表示证书链的第一个证书，一般为client的证书，1表示证书链的第二个证书，以此类推。

> **成功返回证书的颁发者信息，类型为字符串，按默认或指定字符串分隔，失败或不存在返回nil。**

> **适用事件：适用于HTTP事件和USER_REQUEST事件。**

- **示例**

```
when HTTP_REQUEST {
    local email = X509.issuer.dn.get_field("emailAddress")
    if email then
        LOG.debug(LOG.DEBUG, string.format("client email is %s\n", email))
    end
}
```

> 常用的dn字段名称为

字段名称	字段含义
emailAddress	电子邮件
CN	名称
OU	公司/机构
O	部门
L	城市
ST	州/省份
C	国家

X509.subject

X509.subject.dn.get

- **功能**

  获取当前连接client证书链中某一级证书的拥有者信息。

- **语法**

  `X509.subject.dn.get(separator[, reverse[, cert_index]])`

  `separator`：各dn字段之间的分隔符，nil或不填表示默认分隔符，为逗号"，"。

  `reverse`：是否逆序获取，默认为false，即按rfc标准从最后一个元素向前读取，如果为true，则按相反顺序读取。

  `cert_index`：指定证书链中的某一级证书，0或不填表示证书链的第一个证书，一般为client的证书，1表示证书链的第二个证书，以此类推。

> **成功返回证书的拥有者信息，类型为字符串，按默认或指定字符串分隔，失败或不存在返回nil。**

> **适用事件：适用于HTTP事件和USER_REQUEST事件。**

- **示例**

```
when HTTP_REQUEST {
    local subject_dn = X509.subject.dn.get("|")
    if subject_dn then
        LOG.debug(LOG.DEBUG, string.format("client subject_dn is %s\n",
                                           subject_dn))
    end
}
```

> 此代码输出可能为
> "client subject_dn is C =user_c|ST=user_st|L=user_l|
> O=user_o|OU=user_ou|CN=user_cn"

X509.subject.dn.get_field

- **功能**
  获取当前连接client证书链中某一级证书拥有者的具体字段。
- **语法**
  `X509.subject.dn.get_field(key[, cert_index])`
  `key`：具体dn字段的名字，如"emailAddress"。
  `cert_index`：指定证书链中的某一级证书，0或不填表示证书链的第一个证书，一般为client的证书，1表示证书链的第二个证书，以此类推。

> **成功返回证书的拥有者信息，类型为字符串，按默认或指定字符串分隔，失败或不存在返回nil。**

> **适用事件：适用于HTTP事件和USER_REQUEST事件。**

- **示例**

```
when HTTP_REQUEST {
    local email = X509.subject.dn.get_field("emailAddress")
    if email then
        LOG.debug(LOG.DEBUG, string.format("client email is %s\n", email))
    end
}
```

X509.cert_chain_depth

X509.cert_chain_depth.get

- **功能**
  获取当前连接client证书链中证书的总数量。
- **语法**
  `X509.cert_chain_depth.get()`

> **成功返回证书链中证书的数量，类型为数字，失败或不存在返回nil。**

> **适用事件：适用于HTTP事件和USER_REQUEST事件。**

- **示例**

```
when HTTP_REQUEST {
    local index = X509.cert_chain_depth.get()
    if index then
        LOG.debug(LOG.DEBUG, string.format("client cert chain has %s certs\n",
index))
    end
}
```

X509.public_key

X509.public_key.type.get

- **功能**
  获取当前连接中client证书的公钥类型。
- **语法**
  `X509.public_key.type.get()`

  > **成功返回证书的公钥类型，类型为字符串，失败或不存在返回nil。**

  > **适用事件：适用于HTTP事件和USER_REQUEST事件。**

- **示例**

```
when HTTP_REQUEST {
    local public_key_type = X509.public_key.type.get()
    if public_key_type then
        LOG.debug(LOG.DEBUG,
                  string.format("public_key_type of client cert is %s\n",
                                public_key_type))
    end
}
```

> public_key_type可能为："RSA"、"DSA"、"ECC"、"SM2"、"UNKNOWN"

X509.public_key.algorithm.get

- **功能**
  获取当前连接中client证书的公钥算法。
- **语法**
  `X509.public_key.algorithm.get()`

  > **成功返回证书的公钥算法，类型为字符串，失败或不存在返回nil。**

  > **适用事件：适用于HTTP事件和USER_REQUEST事件。**

- **示例**

```
when HTTP_REQUEST {
    local public_key_algorithm = X509.public_key.algorithm.get()
    if public_key then
        LOG.debug(LOG.DEBUG, string.format("public_key_algorithm of client cert
is %s\n",
                                           public_key_algorithm))
    end
}
```

> 上述代码输出可能为"public_key_algorithm of client cert is rsaEncryption"

X509.public_key.bits.get

- **功能**
  获取当前连接中client证书的公钥长度。
- **语法**
  `X509.public_key.bits.get()`

  > **成功返回证书的公钥长度，类型为数字，失败或不存在返回nil。**

  > **适用事件：适用于HTTP事件和USER_REQUEST事件。**
- **示例**

```
when HTTP_REQUEST {
    local public_key_len = X509.public_key.bits.get()
    if public_key_len then
        LOG.debug(LOG.DEBUG,
                  string.format("public_key_len of client cert is %s\n",
                                public_key_len))
    end
}
```

X509.fingerprint

X509.fingerprint.get

- **功能**
  获取当前连接中client证书的指纹，使用SHA-1计算散列。
- **语法**
  `X509.fingerprint.get()`

  > **成功返回证书的指纹，使用SHA-1计算散列，类型为字符串，失败或不存在返回nil。**

  > **适用事件：适用于HTTP事件和USER_REQUEST事件。**
- **示例**

```
when HTTP_REQUEST {
    local fingerprint = X509.fingerprint.get()
    if fingerprint then
        LOG.debug(LOG.DEBUG, string.format("fingerprint of client cert is %s\n",
fingerprint))
    end
}
```

X509.md5

X509.md5.get

- **功能**
  获取当前连接中client证书的md5。
- **语法**
  `X509.md5.get()`

  > **成功返回证书的md5，类型为字符串，失败或不存在返回nil。**

  > **适用事件：适用于HTTP事件和USER_REQUEST事件。**
- **示例**

```
when HTTP_REQUEST {
    local md5 = X509.md5.get()
    if md5 then
        LOG.debug(LOG.DEBUG, string.format("md5 of client cert is %s\n", md5))
    end
}
```

X509.whole

X509.whole.get

- **功能**
  获取当前连接中client证书的全部内容（PEM格式）。
- **语法**
  `X509.whole.get()`

  > **成功返回证书的内容，类型为字符串，失败或不存在返回nil。**

  > **适用事件：适用于HTTP事件和USER_REQUEST事件。**
- **示例**

```
when HTTP_REQUEST {
    local whole = X509.whole.get()
    if whole then
        LOG.debug(LOG.DEBUG,
                  string.format("the whole client cert in PEM is %s\n", whole))
    end
}
```

X509.der_whole

X509.der_whole.get

- **功能**
  获取当前连接中client证书的全部内容（DER格式，经Base64编码）。
- **语法**
  `X509.der_whole.get()`

  > **成功返回经Base64编码的DER格式的证书，类型为字符串，失败或不存在返回nil。**

  > **适用事件：适用于HTTP事件和USER_REQUEST事件。**

- **示例**

```
when HTTP_REQUEST {
    local der_whole = X509.der_whole.get()
    if der_whole then
        LOG.debug(LOG.DEBUG,
                  string.format("the whole client cert in DER encode by
Base64 is %s\n", der_whole))
    end
}
```

X509.version

X509.version.get

- **功能**
  获取当前连接中client证书的版本。
- **语法**
  `X509.version.get()`

  > **成功返回证书的版本，类型为字符串，失败或不存在返回nil。**

  > **适用事件：适用于HTTP事件和USER_REQUEST事件。**

- **示例**

```
when HTTP_REQUEST {
    local version = X509.version.get()
    if version then
        LOG.debug(LOG.DEBUG, string.format("version of client cert is %s\n",
                                           version))
    end
}
```

> 当前，version的值可能为"1"、"2"、"3"。

X509.serial_num

X509.serial_num.get

- **功能**
  获取当前连接中client证书的序列号。
- **语法**
  `X509.serial_num.get()`

  > **成功返回证书的序列号，类型为字符串，失败或不存在返回nil。**

  > **适用事件：适用于HTTP事件和USER_REQUEST事件。**

- **示例**

```
when HTTP_REQUEST {
    local serial_num = X509.serial_num.get()
    if serial_num then
        LOG.debug(LOG.DEBUG, string.format("serial_num of client cert is %s\n",
                                           serial_num))
    end
}
```

X509.signature_algorithm

X590.signature_algorithm.get

- **功能**
  获取当前连接中client证书的签名算法。
- **语法**
  `X509.signature_algorithm.get()`

  > **成功返回证书的签名算法，类型为字符串，失败或不存在返回nil。**

  > **适用事件：适用于HTTP事件和USER_REQUEST事件。**

- **示例**

```
when HTTP_REQUEST {
    local signature_algorithm = X509.signature_algorithm.get()
    if signature_algorithm then
        LOG.debug(LOG.DEBUG,
                  string.format("signature_algorithm of client cert is %s\n",
                                signature_algorithm))
    end
}
```

> 上述代码输出可能为："signature_algorithm of client cert is sha1WithRSAEncryption"

X509.not_valid_before

X509.not_valid_before.get

- **功能**
  获取当前连接中client证书的有效时间（在此之前无效）。
- **语法**
  `X509.not_valid_before.get()`

> **成功返回证书的有效时间，类型为字符串，失败或不存在返回nil。**

> **适用事件：适用于HTTP事件和USER_REQUEST事件。**

- **示例**

```
when HTTP_REQUEST {
    local time = X509.not_valid_before.get()
    if time then
        LOG.debug(LOG.DEBUG,
                  string.format("client cert will be valid after %s\n", time))
    end
}
```

> 上述代码输出可能为："client cert will br valid after Dec 31 11:28:25 2019 GMT"

X509.not_valid_after

X509.not_valid_after.get

- **功能**
  获取当前连接中client证书的有效时间（在此之后无效）。
- **语法**
  `X509.not_valid_after.get()`

> **成功返回证书的有效时间，类型为字符串，失败或不存在返回nil。**

> **适用事件：适用于HTTP事件和USER_REQUEST事件。**

- **示例**

```
when HTTP_REQUEST {
    local time = X509.not_valid_after.get()
    if time then
        LOG.debug(LOG.DEBUG,
                  string.format("client cert is valid before %s\n", time))
    end
}
```

PERSIST

PERSIST.add

- **功能**
  在会话保持表中添加一条记录。
- **语法**
  `PERSIST.add(pool_name, rs_name, type, ...)`
  `pool_name`：指定该条会话保持记录中server-pool的名字。
  `rs_name`：指定该条会话保持记录中RS的名字。
  `type`：指定会话保持类型，当前支持的类型有"hash-cookie"，"hash-header"，"hash-url"，"http-version"，"request-method"，"src-ip"，"real-src-ip"，"message"，

"radius-avp"。

`...`：可变参数，根据参数`type`的不同，输入不同的参数，具体参加下方。

> PERSIST.add(pool_name, rs_name, "**hash-cookie**", cookie_name, cookie_value[, expire[, match-across]])
> `cookie_value`：指定cookie的值，类型为字符串。
> `expire`：指定该条会话保持记录的超时时间，类型为数字，范围为1-3153600（单位：秒），如果传入值大于65535，按65535处理，默认为300秒，**下同。**
> `match-across`：该条会话保持作用范围，如果为"virtual-server"，则此条会话保持记录为全局跨多个虚服务器可用，默认为nil，**下同。**
>
> PERSIST.add(pool_name, rs_name, "**hash-header**", header_name, header_value[, expire[, match-across]])
> `header_name`：指定header的属性名称，类型为字符串。
> `header_value`：指定header的属性值，类型为字符串。
>
> PERSIST.add(pool_name, rs_name, "**hash-url**", url[, expire[, match-across]])
> `url`：指定url的值，类型为字符串。
>
> PERSIST.add(pool_name, rs_name, "**http-version**", http-version[, expire[, match-across]])
> `http-version`：指定http版本，类型为字符串。
>
> PERSIST.add(pool_name, rs_name, "**request-method**", request-method[, expire[, match-across]])
> `request-method`：指定的request方法，类型为字符串，如"POST"。
>
> PERSIST.add(pool_name, rs_name, "**src-ip**", ip[, expire[, match-across]])
> `ip`：指定的ip地址，类型为字符串。
>
> PERSIST.add(pool_name, rs_name, "**real-src-ip**", ip[, expire[, match-across]])
> `ip`：指定的ip地址，类型为字符串。
>
> PERSIST.add(pool_name, rs_name, "**message**", message[, expire[, match-across]])
> `message`：解析的8583消息字段，类型为字符串，最大长度为256
>
> PERSIST.add(pool_name, rs_name, "**radius-avp**", avps[, expire[, match-across]])
> `avps`：指定的RADIUS属性，类型为Lua table。
> Lua table中包含一个成员，名称为data。data的类型也是Lua table，可以包含多条RADIUS属性。
> 每条RADIUS属性包含成员为type和val。其中type为必填项，类型为数字，范围为1-255。其中val的类型为字符串。
> 如果type的值为26（即表示Vendor-Specific属性），还需要传入vendor_id和vendor_type。
> 当type的值为26时，vendor_id为必填项，类型为数字。vendor_type类型为数字，默认值为0。

> **成功返回true，失败返回false。**

> **适用事件：HTTP_REQUEST, HTTP_REQUEST_DATA。**
> **在CLIENT_DATA、SERVER_DATA、USER_REQUEST事件脚本中使用"hash-cookie","hash-header","hash-url","http-version","request-method","real-src-ip"类型，返回失败。**
> **HTTP*事件不支持此接口的message类型，返回失败。**
> **"radius-avp"类型只支持USER_REQUEST事件，在其他事件中使用此会话保持类型，返回失败。**

- **示例1**

```
when HTTP_REQUEST {--add a persist with hash-cookie type, don't match-across
    local cookie_domain = HTTP.cookie.get("domain")
    if cookie_domain then
        local ret = PERSIST.add(pool_1, rs_1, "hash-cookie", cookie_domain, 400)
        if ret then
            LOG.debug(LOG.DEBUG, string.format("add persist success\n"))
        end
    end
}
```

- **示例2**

```
when USER_REQUEST {--add a persist with radius-avp type, match across virtual
server
    local content = {
      ["data"] = {
        {
          ["type"] = 26,
          ["vendor_id"] = 2011,
          ["vendor_type"] = 26,
          ["val"] = struct.pack(">I", 1000)
        },
        {
          ["type"] = 1,
          ["val"] = "bob"
        }
      }
    }
    local ret = PERSIST.add(pool_1, rs_1, "radius-avp", content, 400, "virtual-
server")
    if ret then
        LOG.debug(LOG.DEBUG, string.format("add persist success\n"))
    end
}
```

PERSIST.delete

- **功能**
  在会话保持表中删除一条记录。
- **语法**
  `PERSIST.delete(pool_name, type, ...)`
  `pool_name`：指定该条会话保持记录中server-pool的名字。
  `type`：指定会话保持类型，当前支持的类型有"hash-cookie"，"hash-header"，"hash-url"，"http-version"，"request-method"，"src-ip"，"real-src-ip"，"message"，"radius-avp"。
  `...`：可变参数，根据参数`type`的不同，输入不同的参数，具体参加下方。

> PERSIST.delete(pool_name, "**hash-cookie**", cookie_name, cookie_value[, match-across])
> `cookie_value`：指定cookie的值，类型为字符串。
> `match-across`：该条会话保持作用范围，如果为"virtual-server"，则此条会话保持记录为全局跨多个虚服务器可用，默认为nil，**下同**。

> PERSIST.delete(pool_name, "**hash-header**", header_name, header_value[, match-across])
> `header_name`：指定header的属性名称，类型为字符串。
> `header_value`：指定header的属性值，类型为字符串。
>
> PERSIST.delete(pool_name, "**hash-url**", url[, match-across])
> `url`：指定url的值，类型为字符串。
>
> PERSIST.delete(pool_name, "**http-version**", http-version[, match-across])
> `http-version`：指定http版本，类型为字符串。
>
> PERSIST.delete(pool_name, "**request-method**", request-method[, match-across])
> `request-method`：指定的request方法，类型为字符串，如"POST"。
>
> PERSIST.delete(pool_name, "**src-ip**", ip[, match-across])
> `ip`：指定的ip地址，类型为字符串。
>
> PERSIST.delete(pool_name, "**real-src-ip**", ip[, match-across])
> `ip`：指定的ip地址，类型为字符串。
>
> PERSIST.delete(pool_name, "**message**", message[, match-across])
> `message`：解析的8583消息字段，类型为字符串，最大长度为128
>
> PERSIST.delete(pool_name, "**radius-avp**", avps[, match-across])
> `avps`：指定的RADIUS属性，类型为Lua table。
> Lua table中包含一个成员，名称为data。data的类型也是Lua table，可以包含多条RADIUS属性。
> 每条RADIUS属性包含成员为type和val。其中type为必填项，类型为数字，范围为1-255。其中val的类型为字符串。
> 如果type的值为26（即表示Vendor-Specific属性），还需要传入vendor_id和vendor_type。
> 当type的值为26时，vendor_id为必填项，类型为数字。vendor_type类型为数字，默认值为0。

> **成功返回true，失败返回false。**

> **适用事件：HTTP_REQUEST, HTTP_REQUEST_DATA。**
> **在CLIENT_DATA、SERVER_DATA、USER_REQUEST事件脚本中使用"hash-cookie","hash-header","hash-url","http-version","request-method","real-src-ip"类型，返回失败。**
> **HTTP*事件不支持此接口的message类型，返回失败。**
> **"radius-avp"类型只支持USER_REQUEST事件，在其他事件中使用此会话保持类型，返回失败。**

- **示例1**

```
when HTTP_REQUEST {--delete a persist with hash-cookie type, don't match-across
    local cookie_domain = HTTP.cookie.get("domain")
    if cookie_domain then
        local ret = PERSIST.delete(pool_1, "hash-cookie", cookie_domain)
        if ret then
            LOG.debug(LOG.DEBUG, string.format("delete persist success\n"))
        end
    end
}
```

- **示例2**

```
when USER_REQUEST {--delete a persist with radius-avp type, match across virtual server
```

```
    local content = {
      ["data"] = {
        {
          ["type"] = 26,
          ["vendor_id"] = 2011,
          ["vendor_type"] = 26,
          ["val"] = struct.pack(">I", 1000)
        },
        {
          ["type"] = 1,
          ["val"] = "bob"
        }
      }
    }
    local ret = PERSIST.delete(pool_1, "radius-avp", content, "virtual-server")
    if ret then
        LOG.debug(LOG.DEBUG, string.format("delete persist success\n"))
    end
}
```

PERSIST.persist

- **功能**
  对当前请求做会话保持调度，如果所选的类型在会话保持表中没有查到记录，则会在均衡算法选出RS后，创建一条会话保持记录添加到会话保持表中。如果均衡算法无法选出RS，那么根据虚拟服务器上的交换规则、服务池等配置进行调度。
- **语法**
  `PERSIST.persist(pool_name, type, ...)`
  `pool_name`：指定该条会话保持记录中server-pool的名字，
  `type`：指定会话保持类型，当前支持的类型有"hash-cookie"，"hash-header"，"hash-url"，"http-version"，"request-method"，"src-ip"，"real-src-ip"，"message"，"radius-avp"。
  `...`：可变参数，根据参数`type`的不同，输入不同的参数，具体参加下方。

> PERSIST.persist(pool_name, "**hash-cookie**", cookie_name, cookie_value[, expire[, match-across]])
> `cookie_value`：指定cookie的值，类型为字符串。
> `expire`：指定该条会话保持记录的超时时间，类型为数字，默认为300秒，**下同**。
> `match-across`：该条会话保持作用范围，如果为"virtual-server"，则此条会话保持记录为全局跨多个虚服务器可用，默认为nil，**下同**。
>
> PERSIST.persist(pool_name, "**hash-header**", header_name, header_value[, expire[, match-across]])
> `header_name`：指定header的属性名称，类型为字符串。
> `header_value`：指定header的属性值，类型为字符串。
>
> PERSIST.persist(pool_name, "**hash-url**", url[, expire[, match-across]])
> `url`：指定url的值，类型为字符串。
>
> PERSIST.persist(pool_name, "**http-version**", http-version[, expire[, match-across]])
> `http-version`：指定http版本，类型为字符串。
>
> PERSIST.persist(pool_name, "**request-method**", request-method[, expire[, match-across]])
> `request-method`：指定的request方法，类型为字符串，如"POST"。

> PERSIST.persist(pool_name, "**src-ip**", ip[, expire[, match-across]])
> `ip`：指定的ip地址，类型为字符串。
>
> PERSIST.persist(pool_name, "**real-src-ip**", ip[, expire[, match-across]])
> `ip`：指定的ip地址，类型为字符串。
>
> PERSIST.persist(pool_name, "**message**", message[, expire[, match-across]])
> `message`：解析的8583消息字段，类型为字符串，最大长度为128
>
> PERSIST.persist(pool_name, "**radius-avp**", avps[, expire[, match-across]])
> `avps`：指定的RADIUS属性，类型为Lua table。
> Lua table中包含一个成员，名称为data。data的类型也是Lua table，可以包含多条RADIUS属性。
> 每条RADIUS属性包含成员为type和val。其中type为必填项，类型为数字，范围为1-255。其中val的类型为字符串。
> 如果type的值为26（即表示Vendor-Specific属性），还需要传入vendor_id和vendor_type。
> 当type的值为26时，vendor_id为必填项，类型为数字。vendor_type类型为数字，默认值为0。

> **成功返回true，失败返回false。**

> **适用事件：HTTP_REQUEST, HTTP_REQUEST_DATA。**
> **在CLIENT_DATA、SERVER_DATA、USER_REQUEST事件脚本中使用"hash-cookie","hash-header","hash-url","http-version","request-method","real-src-ip"类型，返回失败。**
> **HTTP*事件不支持此接口的message类型，返回失败。**
> **"radius-avp"类型只支持USER_REQUEST事件，在其他事件中使用此会话保持类型，返回失败。**
> **注意：此接口不能和SLB.pool接口同时调用。如果先调用了SLB.pool，那么调用PERSIST.persist接口会返回失败。**

- **示例1**

```
when HTTP_REQUEST {--persist traffic with hash-cookie type, don't match-across
    local cookie_domain = HTTP.cookie.get("domain")
    if cookie_domain then
        local ret = PERSIST.persist(pool_1, "hash-cookie", cookie_domain, 400)
        if ret then
            LOG.debug(LOG.DEBUG, string.format("persist traffic success\n"))
        end
    end
}
```

- **示例2**

```
when USER_REQUEST {--persist traffic with radius-avp type, match across virtual server
    local content = {
      ["data"] = {
        {
          ["type"] = 26,
          ["vendor_id"] = 2011,
          ["vendor_type"] = 26,
          ["val"] = struct.pack(">I", 1000)
        },
        {
          ["type"] = 1,
          ["val"] = "bob"
```

```
        }
      }
    }
    local ret = PERSIST.persist(pool_1, "radius-avp", content, 400, "virtual-
server")
    if ret then
        LOG.debug(LOG.DEBUG, string.format("persist traffic success\n"))
    end
}
```

PERSIST.find

- **功能**
  在会话保持表中查找一条记录。
- **语法**
  `PERSIST.find(pool_name, type, ...)`
  `pool_name`：指定该条会话保持记录中server-pool的名字。
  `type`：指定会话保持类型，当前支持的类型有"hash-cookie"，"hash-header"，"hash-url"，"http-version"，"request-method"， "src-ip"， "real-src-ip"，"message"，"radius-avp"。
  `...`：可变参数，根据参数`type`的不同，输入不同的参数，具体参加下方。

> PERSIST.find(pool_name, "**hash-cookie**", cookie_name, cookie_value[, match-across])
> `cookie_value`：指定cookie的值，类型为字符串。
> `match-across`：该条会话保持作用范围，如果为"virtual-server"，则此条会话保持记录为全局跨多个虚服务器可用，默认为nil，**下同**。
>
> PERSIST.find(pool_name, "**hash-header**", header_name, header_value[, match-across])
> `header_name`：指定header的属性名称，类型为字符串。
> `header_value`：指定header的属性值，类型为字符串。
>
> PERSIST.find(pool_name, "**hash-url**", url[, match-across])
> `url`：指定url的值，类型为字符串。
>
> PERSIST.find(pool_name, "**http-version**", http-version[, match-across])
> `http-version`：指定http版本，类型为字符串。
>
> PERSIST.find(pool_name, "**request-method**", request-method[, match-across])
> `request-method`：指定的request方法，类型为字符串，如"POST"。
>
> PERSIST.find(pool_name, "**src-ip**", ip[, match-across])
> `ip`：指定的ip地址，类型为字符串。
>
> PERSIST.find(pool_name, "**real-src-ip**", ip[, match-across])
> `ip`：指定的ip地址，类型为字符串。
>
> PERSIST.find(pool_name, "**message**", message[, match-across])
> `message`：解析的8583消息字段，类型为字符串，最大长度为128
>
> PERSIST.find(pool_name, "**radius-avp**", avps[, match-across])
> `avps`：指定的RADIUS属性，类型为Lua table。
> Lua table中包含一个成员，名称为data。data的类型也是Lua table，可以包含多条RADIUS属性。
> 每条RADIUS属性包含成员为type和val。其中type为必填项，类型为数字，范围为1-255。其中val的类型为字符串。
> 如果type的值为26（即表示Vendor-Specific属性），还需要传入vendor_id和vendor_type。

> 当type的值为26时，vendor_id为必填项，类型为数字。vendor_type类型为数字，默认值为0。

- **注意:**

> **成功返回true, rs：{name，ip，port}。失败返回false,rs为nil**
> **在CLIENT_DATA、SERVER_DATA、USER_REQUEST事件脚本中使用"hash-cookie","hash-header","hash-url","http-version","request-method","real-src-ip"类型，返回失败。**
> **HTTP*事件不支持此接口的message类型，返回失败。**
> **"radius-avp"类型只支持USER_REQUEST事件，在其他事件中使用此会话保持类型，返回失败。**

- **示例1**

```
when CLIENT_DATA {
  local client = TCP.client:get()
  local message = client:read(5)
  LOG.debug(LOG.DEBUG, string.format("message[5] %s\n", message))
  rs = PERSIST.find("pool_1", "message", message, "virtual-server")
  if rs ~= nil then
    LOG.debug(LOG.DEBUG, string.format("rs ip %s, name %s\n",
              rs.ip, rs.name))
  end
}
```

- **示例2**

```
when CLIENT_DATA {--find persist with radius-avp type, match across virtual
server
  local content = {
    ["data"] = {
      {
        ["type"] = 26,
        ["vendor_id"] = 2011,
        ["vendor_type"] = 26,
        ["val"] = struct.pack(">I", 1000)
      },
      {
        ["type"] = 1,
        ["val"] = "bob"
      }
    }
  }
  rs = PERSIST.find("pool_1", "radius-avp", content, "virtual-server")
  if rs ~= nil then
    LOG.debug(LOG.DEBUG, string.format("rs ip %s, name %s\n",
              rs.ip, rs.name))
  end
}
```

IP

IP.client_addr

IP.client_addr.get

- **功能**
  获取当前连接中client端的ip地址。
- **语法**
  `IP.client_addr.get()`

> **成功返回ip地址，类型为字符串，失败返回nil。**

> **适用事件：所有事件。**

- **示例**

```
when HTTP_REQUEST {
    local client_ip = IP.client_addr.get()
    if client_ip then
        LOG.debug(LOG.DEBUG, string.format("client ip is %s\n", client_ip))
    end
}
```

IP.server_addr

IP.server_addr.get

- **功能**
  获取均衡算法选出的RS的ip地址。
- **语法**
  `IP.server_addr.get()`

> **成功返回ip地址，类型为字符串，失败返回nil。**

> **适用事件：HTTP_RESPONSE, HTTP_RESPONSE_DATA。**

- **示例**

```
when HTTP_RESPONSE {
    local rs_ip = IP.server_addr.get()
    if rs_ip then
        LOG.debug(LOG.DEBUG, string.format("rs ip is %s\n", rs_ip))
    end
}
```

IP.local_addr

IP.local_addr.get

- **功能**
  在HTTP请求方向时，获取client请求的目的地址，即VS的地址；在HTTP响应的方向，获取ADC设备用于连接RS的地址。
- **语法**
  `IP.local_addr.get()`

> **成功返回ip地址，类型为字符串，失败返回nil。**

> **适用事件：所有事件。**

- **示例**

```
when HTTP_REQUEST {
    local local_ip = IP.local_addr.get()
    if local_ip then
        LOG.debug(LOG.DEBUG, string.format("vs ip is %s\n", local_ip))
    end
}
when HTTP_RESPONSE {
    local local_ip = IP.local_addr.get()
    if local_ip then
        LOG.debug(LOG.DEBUG, string.format("adc use ip %s to connect RS\n",
                                           local_ip))
    end
}
```

IP.remote_addr

IP.remote_addr.get

- **功能**
  在HTTP请求方向时，获取client的地址；在HTTP响应的方向，获取均衡算法选出的RS的地址。
- **语法**
  `IP.remote_addr.get()`

  > **成功返回ip地址，类型为字符串，失败返回nil。**

  > **适用事件：所有事件。**

- **示例**

```
when HTTP_REQUEST {
    local remote_ip = IP.remote_addr.get()
    if remote_ip then
        LOG.debug(LOG.DEBUG, string.format("client ip is %s\n", remote_ip))
    end
}
when HTTP_RESPONSE {
    local remote_ip = IP.remote_addr.get()
    if remote_ip then
        LOG.debug(LOG.DEBUG, string.format("RS ip is %s\n", remote_ip))
    end
}
```

IP.real_client_addr

IP.real_client_addr.get

- **功能**
  获取真实的client的ip地址。
- **语法**
  `IP.real_client_addr.get()`

> 成功返回ip地址，类型为字符串，失败或不存在返回nil。

> 适用事件：适用于HTTP事件。

- **示例**

```
when HTTP_REQUEST {
    local real_client_ip = IP.real_client_addr.get()
    if real_client_ip then
        LOG.debug(LOG.DEBUG, string.format("real client ip is %s\n",
                                           real_client_ip))
    end
}
```

TCP

TCP.client_port

TCP.client_port.get

- **功能**
  获取当前连接中client端的端口号。
- **语法**
  `IP.client_port.get()`

> 成功返回端口号，类型为字符串，失败返回nil。

> 适用事件：所有事件。

- **示例**

```
when HTTP_REQUEST {
    local client_port = TCP.client_port.get()
    if client_port then
        LOG.debug(LOG.DEBUG, string.format("client port is %s\n", client_port))
    end
}
```

TCP.server_port

TCP.server_port.get

- **功能**
  获取均衡算法选出的RS的端口号。
- **语法**
  `TCP.server_port.get()`

> 成功返回端口号，类型为字符串，失败返回nil。

> 适用事件：HTTP_RESPONSE, HTTP_RESPONSE_DATA。

- **示例**

```
when HTTP_RESPONSE {
    local rs_port = TCP.server_port.get()
    if rs_port then
        LOG.debug(LOG.DEBUG, string.format("rs port is %s\n", rs_port))
    end
}
```

TCP.local_port

TCP.local_port.get

- **功能**
  在HTTP请求方向时，获取client请求的目的端口，即VS的端口号；在HTTP响应的方向，获取ADC设备用于连接RS的端口号。
- **语法**
  `TCP.local_port.get()`

  > **成功返回端口号，类型为字符串，失败返回nil。**

  > **适用事件：所有事件。**
- **示例**

```
when HTTP_REQUEST {
    local local_port = TCP.local_port.get()
    if local_port then
        LOG.debug(LOG.DEBUG, string.format("vs port is %s\n", local_port))
    end
}
when HTTP_RESPONSE {
    local local_port = TCP.local_port.get()
    if local_port then
        LOG.debug(LOG.DEBUG, string.format("adc use port %s to connect RS\n",
                                           local_port))
    end
}
```

TCP.remote_port

TCP.remote_port.get

- **功能**
  在HTTP请求方向时，获取client的端口号；在HTTP响应的方向，获取均衡算法选出的RS的端口号。
- **语法**
  `TCP.remote_port.get()`

  > **成功返回端口号，类型为字符串，失败返回nil。**

  > **适用事件：所有事件。**
- **示例**

```
when HTTP_REQUEST {
    local remote_port = TCP.remote_port.get()
    if remote_port then
        LOG.debug(LOG.DEBUG, string.format("client port is %s\n", remote_port))
    end
}
when HTTP_RESPONSE {
    local remote_port = IP.remote_port.get()
    if remote_port then
        LOG.debug(LOG.DEBUG, string.format("RS port is %s\n", remote_port))
    end
}
```

TCP.real_client_port

TCP.real_client_port.get

- **功能**
  获取真实的client的端口号。
- **语法**
  `TCP.real_client_addr.get()`

  > **成功返回端口号，类型为字符串，失败或不存在返回nil。**

  > **适用事件：适用于HTTP事件。**
- **示例**

```
when HTTP_REQUEST {
    local real_client_port = TCP.real_client_port.get()
    if real_client_port then
        LOG.debug(LOG.DEBUG, string.format("real client port is %s\n",
                                           real_client_port))
    end
}
```

TCP.client

TCP.client:get()

- **功能**
  获取客户端TCP连接的对象。
- **语法**
  `TCP.client:get()`

  > **适用于TCP事件。**
- **示例**

```
when CLIENT_DATA {
    local client = TCP.client:get()
    local bytes, err = client:send("hello lua")
    if not bytes then
        LOG.debug(LOG.ERROR, string.format("send message failed, err %s\n",
err))
    end
}
```

client:setopts(opts)

- **功能**
  设置客户端TCP连接对象的属性。
- **语法**
  `client:setopts(opts)`
  `opts`：为table，包括如下属性send_timeout, read_timeout, congestion_send_timeout,congestion_read_timeout

  > **适用于TCP事件。**
  > **所有参数必须设置**

- **示例**

```
when CLIENT_DATA {
    local client = TCP.client:get()
        -- 设置客户端TCP连接对象的发送和接收超时、拥塞等待时间为1s。
        local err = client:setopts({send_timeout = 1000, read_timeout = 1000,
congestion_send_timeout = 1000,congestion_read_timeout = 1000})
        if err then
            LOG.debug(LOG.ERROR, string.format("set opts failed, err %s\n",
err))
        end
}
```

client:send(data)

- **功能**
  向客户端发送数据。
- **语法**
  `bytes, err = client:send(data)`
  `data`：客户端发送数据。

  > **发送成功是bytes为发送的长度，如果出现失败bytes为nil，err表示错误原因。**
  > **适用于TCP事件。**

- **示例**

```
when CLIENT_DATA {
    local client = TCP.client:get()
    local bytes, err = client:send("hello lua")
    if not bytes then
        LOG.debug(LOG.ERROR, string.format("send message failed, err %s\n",
err))
    end
}
```

client:read(...)

- **功能**
  从客户端读取数据。
- **语法**
  `data, err, partial = client:read(["*l"])`
  `data, err, partial = client:read("*a")`
  `data, err, partial = client:read(size)`
  `["*l"]`：可选参数，从客户端读取数据，一般读取到'\n'(ASCII 10)返回。
  `"*a"`：从客户端读取数据，一般读取直到客户端关闭。
  `size`：从客户端读取数据，一般读取size表示的字节数。

> **data, err, partial：当成功时，data为读取的数据；当失败时，data为nil，err表示出错信息，partail表示读取的部分内容。**
> **适用于TCP事件。**

- **示例**

```
when CLIENT_DATA {
    local client = TCP.client:get()
    local data, err, partail = client:read(5)
    if data then
        LOG.debug(LOG.DEBUG, string.format("read message %s\n", data))
    else
        LOG.debug(LOG.DEBUG, string.format("read message failed, err %s, partial
message %s\n", err, partial))
    end
}
```

client:peek(size)

- **功能**
  查看客户端发送的数据。
- **语法**
  `data, err = client:peek(size)`
  `size`：查看客户端的发送数据，size表示要查看的字节数。
- **注意:**

> **当成功时，data为查看的数据；当失败时，data为nil，err表示出错信息。**
> **适用于USER_REQUEST事件。**

- **示例**

```
when USER_REQUEST {
    local client = TCP.client:get()
    local data, err, partail = client:peek(1) -- 数据长度，1字节
    if data then
        LOG.debug(LOG.DEBUG, string.format("peek message length %d\n", data))
    end
}
```

TCP.server

TCP.server:new(pool_name, rs_name, opts)

- **功能**
  获取服务端TCP连接的对象。
- **语法**
  `TCP.server:new()`
  `pool_name`：为SP名称；
  `rs_name`：为RS名称；
  `opts`：为table，包括connect_timeout, send_timeout, read_timeout; congestion_send_timeout,congestion_read_timeout发送、接收消息过程中出现拥塞时的等待时间。

> **适用于CLIENT_DATA事件、server_data事件。**
> **建立连接后不仅是发送一次数据，会涉及保持功能，保持功能依赖rs_id，故暂不支持仅使用ip、port创建后端rs的连接对象。**
> **如果pool_name配置包含rs_name，则创建一个连接对象，后续使用其他pool_name+rs_name调用接口时返回之前创建的连接对象，日志记录中的服务池记录创建时的pool_name名字**

- **示例**

```
when SERVER_DATA {
    local client = TCP.client:get()
    -- server is Visibility
    local data, err = server:read(4)
    if not data then
        LOG.debug(LOG.ERROR, string.format("read message failed, err %s\n",
err))
    elseif data == "pong" then
        client:send(data)
    else
        client:send("unexpected response")
    end
}
when CLIENT_DATA {
    local server = TCP.server:new(pool_name, rs_name, {connect_timeout = 1000,
send_timeout = 1000, read_timeout = 1000})
    local bytes, err = server:send("ping")
    if not bytes then
        LOG.debug(LOG.ERROR, string.format("send message failed, err %s\n",
err))
    end
}
```

TCP.server:get()

- **功能**
  获取服务端TCP连接的对象。
- **语法**
  `TCP.server:get()`

  > **适用于CLIENT_DATA事件、server_data事件。**

- **示例**

```
when CLIENT_DATA {
    local server = TCP.server:get()
    local bytes, err = server:send("hello lua")
    if not bytes then
        LOG.debug(LOG.ERROR, string.format("send message failed, err %s\n",
err))
    end
}
```

server:setopts(opts)

- **功能**
  设置服务端TCP连接对象的属性。
- **语法**
  `server:setopts(opts)`
  `opts`：table，包括如下属性send_timeout, read_timeout, congestion_send_timeout,congestion_read_timeout

> **适用于CLIENT_DATA事件、server_data事件。**
> **所有参数必须设置**

- **示例**

```
when CLIENT_DATA {
    local server = TCP.server:new(pool_name, rs_name, {connect_timeout = 1000,
send_timeout = 1000, read_timeout = 1000})
    -- 设置服务端TCP连接对象的发送和接收超时、拥塞等待时间为1s。
    local err = server:setopts({send_timeout = 1000, read_timeout = 1000,
congestion_send_timeout = 1000,congestion_read_timeout = 1000})
    if err then
        LOG.debug(LOG.ERROR, string.format("set opts failed, err %s\n", err))
    end
}
```

server:send(data)

- **功能**
  向服务端发送数据。
- **语法**
  `server:send(data)`

bytes, err = server:send(data)：向服务端发送数据，发送成功是bytes为发送的长度，如果出现失败bytes为nil，err表示错误原因。

> **适用于CLIENT_DATA事件、server_data事件。**

- **示例**

```
when CLIENT_DATA {
    local server = TCP.server:new(pool_name, rs_name, {connect_timeout = 1000,
send_timeout = 1000, read_timeout = 1000})
    local bytes, err = server:send("hello lua")
    if not bytes then
        LOG.debug(LOG.ERROR, string.format("send message failed, err %s\n",
err))
    end
}
```

server:read(...)

- **功能**
  从服务端读取数据。
- **语法**
  `data, err, partial = server:read(["*l"])`
  `data, err, partial = server:read("*a")`
  `data, err, partial = server:read(size)`
  `["*l"]`：可选参数，从服务端读取数据，一般读取到'\n'(ASCII 10)返回。
  `"*a"`：从服务端读取数据，一般读取直到服务端关闭。
  `size`：从服务端读取数据，一般读取size表示的字节数。

> **data, err, partial：当成功时，data为读取的数据；当失败时，data为nil，err表示出错信息，partail表示读取的部分内容。**
> **适用于CLIENT_DATA事件、server_data事件。**

- **示例**

```
when CLIENT_DATA {
    local server = TCP.server:new(pool_name, rs_name, {connect_timeout = 1000,
send_timeout = 1000, read_timeout = 1000})
    local data, err, partail = server:read(5)
    if data then
        LOG.debug(LOG.DEBUG, string.format("read message %s\n", data))
    else
        LOG.debug(LOG.DEBUG, string.format("read message failed, err %s, partial
message %s\n", err, partial))
    end
}
```

server:close()

- **功能**
  关闭服务端TCP连接。
- **语法**
  `ok, err = server:close()`

> **当成功时，ok为1；当失败时，ok为nil，err表示出错信息。**
> **适用于CLIENT_DATA事件、server_data事件。**

示例
when CLIENT_DATA {
local server = TCP.server:new(pool_name, rs_name,{connect_timeout = 1000, send_timeout = 1000, read_timeout = 1000})
--[[

```
process message
--]]
server:close()
}
```

UDP

UDP.peek(size)

- **功能**
  从客户端预读UDP数据。
- **语法**
  `data, err = UDP.peek(size)`
  `size`：表示预读的字节数。

> **data, err：当成功时，data为读取的数据；当失败时，data为nil，err表示出错信息。**
> **适用于USER_REQUEST事件。**

- **示例**

```
when USER_REQUEST {
    local data, err = UDP.peek(20)
    if data then
        LOG.debug(LOG.DEBUG, string.format("read message %s\n", data))
    else
        LOG.debug(LOG.DEBUG, string.format("read message failed, err %s\n",
err))
    end
}
```

LOG

LOG.debug

- **功能**
  打印debug信息，需要在设备中开启对应开关*debug slb_script xxx*才能显示出对应信息。
- **语法**
  `LOG.debug(level, str_format)`
  `level`：打印debug信息的级别，目前支持级别"LOG.DEBUG"，"LOG.INFO"，"LOG.NOTICE"，"LOG.WARNING"，"LOG.ERROR"，"LOG.CRITICAL"，"LOG.ALERT"，"LOG.EMERGENCY"。
  `str_format`：指定打出的debug信息内容。

> **无返回值。**

> **适用事件：所有事件。**

- **示例**

```
when HTTP_REQUEST {
    LOG.debug(LOG.DEBUG, "enter event HTTP_REQUEST\n")
    local real_client_port = TCP.real_client_port.get()
    if real_client_port then
        LOG.debug(LOG.DEBUG, string.format("real client port is %s\n",
                                           real_client_port))
    else
        LOG.debug(LOG.WARNING, string.format("real client port is not exist\n"))
    end
}
```

LOG.log

- **功能**
  生成日志。
- **语法**
  `ok, err = LOG.log(rs_name,log_text)`
  `rs_name`：指定rs_name名字
  `log_text`：表示由用户生成的需要记录的文本。

> **当成功时，ok为1；当失败时，ok为nil，err表示出错信息。**
> **适用于CLIENT_DATA、SERVER_DATA事件。**
> **需要开启日志相关配置才能生成日志。**

示例

```
when CLIENT_DATA {
    --[[
    process message
    --]]
    LOG.log(rs_name, string.format("send test message to %s.",rs_name))
}
```

库

bit库

bit库提供位运算操作。注意，所有bit库接口都返回32位带符号整数。示例中print函数为lua标准实现，printx辅助函数如下

```
function printx(x)
  print("0x"..bit.tohex(x))
end
```

bit.tobit (x)

转成位格式

- **示例**

```
printx(bit.tobit(0xffffffff))    --> 0xffffffff
```

bit.tohex (x [,n])

将其第一个参数转换为十六进制字符串。十六进制数字的数目由可选的第二个参数的绝对值给出。1到8之间的正数会生成小写的十六进制数字。负数生成大写的十六进制数字。仅最低有效的4 * | n | 使用位。默认值是生成8个小写的十六进制数字。

- **示例**

```
print(bit.tohex(1))              --> 00000001
print(bit.tohex(-1))             --> ffffffff
print(bit.tohex(0xffffffff))     --> ffffffff
print(bit.tohex(-1, -8))         --> FFFFFFFF
print(bit.tohex(0x21, 4))        --> 0021
print(bit.tohex(0x87654321, 4))  --> 4321
```

bit.bnot (x)

按位取反

- **示例**

```
print(bit.bnot(0))            --> -1
printx(bit.bnot(0))           --> 0xffffffff
print(bit.bnot(-1))           --> 0
print(bit.bnot(0xffffffff))   --> 0
printx(bit.bnot(0x12345678))  --> 0xedcba987
```

bit.band (x1 [,x2...])

与操作

- **示例**

```
printx(bit.band(0x12345678, 0xff))        --> 0x00000078
```

bit.bor (x1 [,x2...])

或操作

- **示例**

```
print(bit.bor(1, 2, 4, 8))                --> 15
```

bit.bxor (x1 [,x2...])

异或操作

- **示例**

```
printx(bit.bxor(0xa5a5f0f0, 0xaa55ff00))  --> 0x0ff00ff0
```

bit.lshift (x, n)

逻辑左移

- **示例**

```
print（bit.lshift（1，0））-> 1
print（bit.lshift（1，8））-> 256
print（bit.lshift（1，40））-> 256
printx（bit.lshift（0x87654321，12））-> 0x54321000
```

bit.rshift (x, n)

逻辑右移

- **示例**

```
print（bit.rshift （256，8 ））-> 1
print（bit.rshift（-256，8））-> 16777215
printx（bit.rshift（0x87654321，12））-> 0x00087654
```

bit.arshift (x, n)

算数右移

- **示例**

```
print（bit.arshift（256，8））-> 1
print（bit.arshift（-256， 8））-> -1
printx（bit.arshift（0x87654321，12））- -> 0xfff87654
```

bit.rol (x, n)

按照给的位数n，将参数x循环左移。n的低5比特被使用。

- **示例**

```
printx(bit.rol(0x12345678, 12))   --> 0x45678123
```

bit.ror (x, n)

按照给的位数n，将参数x循环右移。n的低5比特被使用。
示例
printx(bit.ror(0x12345678, 12)) --> 0x67812345

bit.bswap (x)

交换其参数的字节并返回它。这可用于将小端32位数字转换为大端32位数字，反之亦然。

- **示例**

```
printx(bit.bswap(0x12345678)) --> 0x78563412
printx(bit.bswap(0x78563412)) --> 0x12345678
```

struct库

为解析应用层协议，提供消息的拆解方法。该库提供了从C结构转换成Lua值的基本功能，主要函数接口有struct.pack(将多个Lua值封装为结构化的二进制数据)和struct.unpack(从结构化的二进制数据解包为多个Lua的值)。

格式fmt

格式字符串，描述了结构的层次。格式字符串是一个转换序列，用来描述当前字节序和当前对齐要求的序列。默认字节序是机器的本地字节序顺序，不要求对齐，当然你可以在格式字符串适当的位置进行更改这些设置。

模式	说明
" "	忽略空格
"!n"	指定按n字节对齐(n必须为2的幂)，缺省使用机器对齐方式
">"	设置为大端
"<"	设置为小端
"x"	填充0字节
"b"	有符号字符
"B"	无符号字符
"h"	有符号短整型
"H"	无符号短整型
"l"	有符号长整型
"L"	无符号长整型
"T"	size_t
"in"	n字节的有符号整型
"In"	n字节的无符号整型
"f"	浮点型
"d"	双精度浮点型
"s"	\0结尾的字符串
"cn"	n个字符的字符串，默认为1，封装时，给定的字符串必须至少有n个字符(多余的字符被丢弃)
"c0"	和"cn"类似,除了n是通过其他方式:封装时,n是给定字符串的长度;解包时,n是指定待解包的长度

struct.pack(fmt, ...)

封装数据包：struct.pack(fmt, d1, d2, ...)，这个函数将根据格式字符串fmt的定义返回一个包括d1,d2...的字符串。

struct.unpack(fmt, data)

解压数据包: struct.unpack(fmt, s, [i])，根据格式字符串fmt解封装字符串s，可选参数i标记读取位置，默认为1；读取数据后，此函数同时返回读取的结束位置，用来进行获取剩下的字符串。

struct.size(fmt)

计算数据大小:计算根据指定的格式字符串格式化之后的字符串大小。fmt格式化字符串中不能包括选项"s"和"c0"。

示例

- **示例**

```
（1）输出当前机器上整型占的字节数

print(struct.size("i"))    --输出当前机器上整型占的字节数

（2）封装和解封装下面的数据结构，可以使用格式字符串"<!4biiii"。

struct Str {
    char b;
    int i[4];
}

（3）如果需要封装或者解封装一个很大的数组，可以使用string.rep来自动生成格式字符串，例如，针对下面的结构体，可以使用"d"..string.rep("i",400)来生成格式串。

struct Str {
    doubble x;
    int i[400];
}

（4）x = struct.pack("Bc0", string.len(s), s) --将字符串长度封装在字符串的首部

     s = struct.unpack("Bc0", x) -解封装字符串x

（5）封装指定宽度的字符，不足时用空串填充，可以使用如下操作：

     x = struct.pack("c10",  s .. string.rep("", 10))
```

CONVERT库

CONVERT库提供进制转换操作。注意，所有CONVERT库接口都返回32bit所表示的数字字符串，不足部分用0补足。

CONVERT.dec2bin(x)

将十进制数字转换成二进制数据字符串

- **语法**
  `local big_str = CONVERT.dec2bin(dec_num)`

> **dec_num为十进制数字**
> **bit_str为二进制数据的字符串**
> **适用事件：适用于USER_REQUEST事件**

- **示例**

```
when USER_REQUEST {
    local bit_str = CONVERT.dec2bin(255) -bit_str is
00000000000000000000000011111111,type of bit_str is string
}
```

CONVERT.dec2oct(x)

将十进制数字转换成八进制数据字符串

- **语法**
  `local oct_str = CONVERT.dec2oct(dec_num)`

> **dec_num为十进制数字**
> **oct_str为八进制数据的字符串**
> **适用事件：适用于USER_REQUEST事件**

- **示例**

```
when USER_REQUEST {
    local oct_str = CONVERT.dec2oct(255) -oct_str is 00000000377,type of
oct_str is string
}
```

CONVERT.dec2hex(x)

将十进制数字转换成十六进制数据字符串

- **语法**
  `local hex_str = CONVERT.dec2hex(dec_num)`

  > **dec_num为十进制数字**
  > **hex_str为十六进制数据的字符串**
  > **适用事件：适用于USER_REQUEST事件**

- **示例**

```
when USER_REQUEST {
    local hex_str = CONVERT.dec2hex(255) -hex_str is 000000ff,type of
hex_str is string
}
```