RULEX 边缘设备开发指南

BY WWHAI

2022年11月24日

注意 1. 该文档基于 V0.4.0主分支。

摘要
本文主要讲解了 RULEX 边缘设备开发框架的开发技巧已经几个关键概念，从而让开发者加深印象，快速开发出边缘类设备。

目录

1 LUA引擎

RULEX框架引入了一个 LUA 脚本解释器，用来处理用户自定义业务逻辑。该 LUA 解释器基于LUA 5.0，可以运行标准LUA代码，同时提供了一系列 RULEX 框架特有的函数。

1 LUA基础

LUA 是一个小巧的编程语言，这里只做个简单介绍，更多资料可以参考：

- https://www.twle.cn/l/yufei/lua53/lua-basic-index.html
- https://www.lua.org/docs.html

2 RULEX标准库

RULEX 框架内置了一些函数，可参考此处：

- https://github.com/i4de/rulex/wiki/RULEX-%E5%87%BD%E6%95%B0%E5%BA%93

3 基础示例

经过上面的学习以后，相信你已经掌握了 LUA 语言的基础语法和 RULEX 的基础库。下面我们重点讲一下 RULEX 的 LUA 回调框架。

3.1 成功回调

当回调成功后，会执行 Success 函数，下面是一个简单的案例：

```
function Success()
    print("Success Callback")
end
```

上面的 Demo 表示，当成功后输出一个字符串信息。

3.2 失败回调

当调用失败以后，会执行 Falied 函数，下面是一个简单的案

```
function Failed(error)
    print("Callback Failed With Error:", error)
end
```

上面的 Demo 表示，当成功后输出一个字符串信息，其中 error 参数是失败后的错误原因，是个字符串类型。

3.3 规则回调

规则回调相对来说比较复杂，首先我们看看规则的函数原型：

```
Actions = {
     function(data)
       print("f1", data)
       return true, data
     end,
     function(data)
       print("f2", data)
       return true, data
     end
     -- other functions
}
```

规则回调 Actions 是一个 *LUA 函数列表*，可以在里面加入多个匿名函数，函数只有1个参数，该参数是原始数据，有2个返回值，第一个返回值表示是否继续执行下面的函数，第二个参数表示返回给下面的函数的值。

4 案例展示

下面展示一个案例，该案例是用来控制一个8路继电器的Demo：

```
Actions = {
    function(data)
        print('Received␣data␣from␣iothub:', data)
        local source = 'tencentIothub'
        local device = 'YK8Device1'
        local dataT, err = rulexlib:J2T(data)
        if (err ~= nil) then
            print('Received␣data␣from␣iothub␣parse␣to␣json␣error:', err)
            return false, data
        end
        -- Action
        if dataT['method'] == 'action' then
            local actionId = dataT['actionId']
            if actionId == 'get_state' then
                local readData, err = rulexlib:ReadDevice(0, device)
                if (err ~= nil) then
                    print('ReadDevice␣data␣from␣device␣error:', err)
                    return false, data
                end
                print('ReadDevice␣data␣from␣device:', readData)
                local readDataT, err = rulexlib:J2T(readData)
                if (err ~= nil) then
                    print('Parse␣ReadDevice␣data␣from␣device␣error:', err)
                    return false, data
                end
                local yk08001State = readDataT['yk08-001']
                print('yk08001State:', yk08001State['value'])
                local _, err = iothub:ActionSuccess(source, dataT['id'],
                    yk08001State['value'])
                if (err ~= nil) then
                    print('ActionReply␣error:', err)
```

```
                return false, data
            end
        end
    end
    -- Property
    if dataT['method'] == 'property' then
        local schemaParams = rulexlib:J2T(dataT['data'])
        print('schemaParams:', schemaParams)
        local n1, err = rulexlib:WriteDevice(device, 0, rulexlib:T2J({
            {
                ['function'] = 15,
                ['slaverId'] = 3,
                ['address'] = 0,
                ['quantity'] = 1,
                ['value'] = rulexlib:T2Str({
                    [1] = schemaParams['sw8'],
                    [2] = schemaParams['sw7'],
                    [3] = schemaParams['sw6'],
                    [4] = schemaParams['sw5'],
                    [5] = schemaParams['sw4'],
                    [6] = schemaParams['sw3'],
                    [7] = schemaParams['sw2'],
                    [8] = schemaParams['sw1']
                })
            }
        }))
        if (err ~= nil) then
            print('WriteDevice error:', err)
            local _, err = iothub:PropertyFailed(source, dataT['id'])
            if (err ~= nil) then
                print('Reply error:', err)
                return false, data
            end
        else
            local _, err = iothub:PropertySuccess(source, dataT['id'], {})
            if (err ~= nil) then
                print('Reply error:', err)
                return false, data
            end
        end
    end
    return true, data
end
}
```

2 设备

本章节主要讲如何为 RULEX 网关开发一款设备，并将其与 RULEX 框架关联，实现设备和框架之间的通信。

在 RULEX 框架里，把和外部真实设备交互的功能称之为设备，本质上是对外部设备的一个抽象描述，比如可能是个 Modbus 客户端，也可能是个 TCP 客户端等。它描述的是一个外部硬件设备的通信行为。

1 设备接口

要为 RULEX 框架新增一个逻辑或者物理上的设备，必须实现以下接口：

```
type XDevice interface {
	Init(devId string, configMap map[string]interface{}) error
	Start(CCTX) error
	OnRead(cmd int, data []byte) (int, error)
	OnWrite(cmd int, data []byte) (int, error)
	Status() DeviceState
	Stop()
	Property() []DeviceProperty
	Details() *Device
	SetState(DeviceState)
	Driver() XExternalDriver
	OnDCACall(UUID string, Command string, Args interface{}) DCAResult
}
```

2 接口函数

2.1 Init

设备的初始化函数，RULEX 框架会把配置参数传递给该函数，devId 表示其 UUID，configMap 表示其工作配置，例如串口的波特率，奇偶校验等。

2.2 Start

Start 一般作为常驻任务存在，表示该设备的一个工作中状态，例如我们需要周期性采集某个传感器的数据的时候，Start 一般用来启动一个线程去动态轮询。如果该设备无常驻工作任务，该接口返回 nil 即可。

2.3 OnRead

设备提供给外界的读数据接口，当外界通过 LUA 调用 rulexlib:ReadDevice 的时候，参数会传递进来。该函数有2个参数，第一个是用来区分设备的命令，第二个是字节数组，也就是最终的数据保存地方。

cmd 参数主要用来做辅助性的工作。例如一个温湿度传感器有2个指令，一个是读温度，一个是读湿度。我们向设备读数据的时候，可以将 cmd 参数作为一个区分，例如对于底层来说，“1” 代表读温度，“2”代表读湿度，这样可以更方便的分辨出来命令的类型。

2.4 OnWrite

该接口主要用来向设备写入数据用，其参数含义和上面的 OnRead 一样。例如改变某个寄存器的值或者是对某个设备进行操作控制等。常见的操作就是控制某个寄存器的值来实现灯光关开状态控制。假设我们开灯，LUA 代码表示如下：

```
rulexlib:WriteDevice(UUID, 1, 1)
```

设备端处理代码如下：

```
OnWrite(cmd int, data []byte){
	if cmd ==1 {
		print("开灯")
	}
```

```
        if cmd ==0{
              print("关灯")
        }
        return 0, nil
}
```

2.5 Status

获取设备的状态，通常该状态应该由设备内部控制，RULEX 框架通过获取该状态来决定设备的生命周期。

2.6 Stop

停止设备接口，一般在这里释放资源。

2.7 Property

获取属性列表，表示这个设备能提供哪些参数，该接口常被用来做外部参考用，例如本网关可以提供温度、湿度、CO2 等参数，可以将其注册到该接口返回。

2.8 Details

获取该设备的详细信息，即外部传进来的参数。例如用户新建一个设备的时候可能会传进来设备名称，备注或者串口配置等，都通过这个接口关联。

2.9 SetState

设置状态，该函数由 RULEX 框架调用，会传进来一个状态值，用户要做的事情就是把这个值替换成自己的状态即可。

2.10 Driver

这个接口用来获取设备的驱动，驱动指的是桥接软硬件的一个软件。一般而言，设备都应该有个驱动，但是一些简单设备可以不用。所以视情况而定，此接口拿到的可能是 *nil* 值。

2.11 OnSCACall

该接口属于高级接口，用来实现边缘设备之间相互调用功能，但是本版本不支持，仅仅保留位置即可。

3 返回值

常见返回值：

1. **n**：返回字节数目
2. **error**：错误状态
3. **[]DeviceProperty**：属性表

4 案例展示

下面展示一个串口设备的案例。

```
package device

import (
        "context"
        "encoding/json"
        "errors"
        "sync"
        "time"

        "github.com/i4de/rulex/common"
        "github.com/i4de/rulex/driver"
        "github.com/i4de/rulex/glogger"
        "github.com/i4de/rulex/typex"
        "github.com/i4de/rulex/utils"
        serial "github.com/wwhai/goserial"
)

type genericUartDevice struct {
        typex.XStatus
        status     typex.DeviceState
        RuleEngine typex.RuleX
        driver     typex.XExternalDriver
        mainConfig common.GenericUartConfig
        locker     sync.Locker
}

/*
*
* 通用串口透传
*
 */
func NewGenericUartDevice(e typex.RuleX) typex.XDevice {
        uart := new(genericUartDevice)
        uart.locker = &sync.Mutex{}
        uart.mainConfig = common.GenericUartConfig{}
        uart.RuleEngine = e
        return uart
}

// 初始化
func (uart *genericUartDevice) Init(devId string, configMap
map[string]interface{}) error {
        uart.PointId = devId
        if err := utils.BindSourceConfig(configMap, &uart.mainConfig); err !=
nil {
                glogger.GLogger.Error(err)
                return err
        }
        if !contains([]string{"N", "E", "O"}, uart.mainConfig.Parity) {
                return errors.New("parity value only one of 'N','O','E'")
        }
        return nil
}

// 启动
func (uart *genericUartDevice) Start(cctx typex.CCTX) error {
```

```
        uart.Ctx = cctx.Ctx
        uart.CancelCTX = cctx.CancelCTX

        // 串口配置固定写法
        // 下面的参数是传感器固定写法
        config := serial.Config{
                Address:  uart.mainConfig.Uart,
                BaudRate: uart.mainConfig.BaudRate,
                DataBits: uart.mainConfig.DataBits,
                Parity:   uart.mainConfig.Parity,
                StopBits: uart.mainConfig.StopBits,
                Timeout:  time.Duration(uart.mainConfig.Timeout) * time.Second,
        }
        serialPort, err := serial.Open(&config)
        if err != nil {
                glogger.GLogger.Error("rawUartDriver start failed:", err)
                return err
        }
        uart.driver = driver.NewRawUartDriver(uart.Ctx, uart.RuleEngine,
uart.Details(), serialPort)
        if !uart.mainConfig.AutoRequest {
                uart.status = typex.DEV_UP
                return nil
        }
        go func(ctx context.Context) {
                ticker :=
time.NewTicker(time.Duration(uart.mainConfig.Frequency) * time.Second)
                buffer := make([]byte, common.T_64KB)
                uart.driver.Read(0, buffer) //清理缓存
                for {
                        <-ticker.C
                        select {
                        case <-ctx.Done():
                                ticker.Stop()
                                return
                        default:
                                uart.locker.Lock()
                                n, err := uart.driver.Read(0, buffer)
                                uart.locker.Unlock()
                                if err != nil {
                                        glogger.GLogger.Error(err)
                                        continue
                                }
                                mapV := map[string]interface{}{
                                        "tag":   uart.mainConfig.Tag,
                                        "value": string(buffer[:n]),
                                }
                                bytes, _ := json.Marshal(mapV)
                                uart.RuleEngine.WorkDevice(uart.Details(),
string(bytes))
                        }
                }

        }(uart.Ctx)
        uart.status = typex.DEV_UP
        return nil
```

```
}

// 从设备里面读数据出来:
//
//      {
//          "tag":"data tag",
//          "value":"value s"
//      }
func (uart *genericUartDevice) OnRead(cmd int, data []byte) (int, error) {
        uart.locker.Lock()
        n, err := uart.driver.Read(0, data)
        uart.locker.Unlock()
        buffer := make([]byte, n)
        mapV := map[string]interface{}{
                "tag":   uart.mainConfig.Tag,
                "value": string(buffer[:n]),
        }
        bytes, _ := json.Marshal(mapV)
        copy(data, bytes)
        return n, err
}

// 把数据写入设备
func (uart *genericUartDevice) OnWrite(cmd int, b []byte) (int, error) {
        return uart.driver.Write(0, b)
}

// 设备当前状态
func (uart *genericUartDevice) Status() typex.DeviceState {
        return typex.DEV_UP
}

// 停止设备
func (uart *genericUartDevice) Stop() {
        if uart.driver != nil {
                uart.driver.Stop()
        }
        uart.CancelCTX()
        uart.status = typex.DEV_STOP
}

// 设备属性，是一系列属性描述
func (uart *genericUartDevice) Property() []typex.DeviceProperty {
        return []typex.DeviceProperty{}
}

// 真实设备
func (uart *genericUartDevice) Details() *typex.Device {
        return uart.RuleEngine.GetDevice(uart.PointId)
}

// 状态
func (uart *genericUartDevice) SetState(status typex.DeviceState) {
        uart.status = status
}
```

```
// 驱动
func (uart *genericUartDevice) Driver() typex.XExternalDriver {
        return uart.driver
}


func contains(s []string, e string) bool {
        for _, a := range s {
                if a == e {
                        return true
                }
        }
        return false
}

func (uart *genericUartDevice) OnDCACall(UUID string, Command string, Args
interface{}) typex.DCAResult {
        return typex.DCAResult{}
}
```

3 驱动

驱动实际上就是真实和硬件设备通信的软件。关于驱动的概念可以参考下同类知识点，例如 Linux 驱动等，此处不过多赘述。下面主要讲一下 RULEX 框架里面的设备驱动的概念。

在 RULEX 里，驱动负责读取一些来自低级硬件的数据，比如串口，或者485链路等。驱动保证了和设备的数据交互，例如读写 Modbus 寄存器等。其工作原理如图1所示。

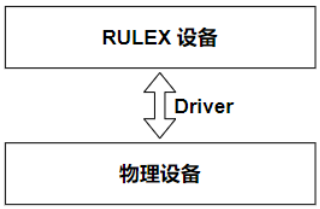

图 1.

1 驱动接口

下面是驱动的接口定义：

```
type XExternalDriver interface {
        Test() error
        Init(map[string]string) error
        Work() error
        State() DriverState
        Read(cmd int, data []byte) (int, error)
        Write(cmd int, data []byte) (int, error)
        DriverDetail() DriverDetail
        Stop() error
```

```
}
```

2 接口函数

2.1 Test

Test 接口用来测试驱动的可用性，一般是设备刚启动的时候，给驱动发一个 Test 信号，常用来判断物理设备是否链路通达。例如可以尝试给某个 IP 地址发送 ICMP 包测试网络可达性等。该接口是一个辅助性接口。

2.2 Init

驱动的初始化函数，设备框架会把配置参数传递给该函数，configMap 表示其工作配置，例如串口的波特率，奇偶校验等。

2.3 Work

Work 一般作为常驻任务存在，表示该驱动的一个工作中状态，例如我们需要周期性采集某个传感器的数据的时候，Work 一般用来启动一个线程去动态轮询。如果该设备无常驻工作任务，该接口返回 nil 即可。

2.4 State

获取状态，该状态代表了驱动的可用性，驱动有以下几个状态：

- DRIVER_STOP：状态一般用来直接停止一个资源，监听器不需要重启
- DRIVER_UP：工作态
- DRIVER_DOWN：资源挂了，属于工作意外，需要重启

2.5 Read

设备会调用此接口，主要用来从物理设备拿回原始数据，例如从一个 Modbus 从设备取某些寄存器的原始值等。

2.6 Write

设备会调用此接口，主要用来向物理设备写原始数据，例如向一个 Modbus 从设备取写某些寄存器的值等。

2.7 DriverDetail

获取该设备的详细信息，一般会标记该设备的底层元信息，例如名称，版本等，下面展示一个环境传感器的设备详情元数据：

```
return typex.DriverDetail{
        Name:        "TC-S200",
        Type:        "UART",
        Description: "TC-S200␣系列空气质量监测仪",
}
```

2.8 Stop

设备停止后调用该函数，可在此释放硬件资源。

3 返回值

常见返回值：

1. **n**：返回字节数目
2. **error**：错误状态
3. **DriverDetail**：设备详情

4 案例展示

下面我们简单展示一个串口驱动的示例。

```
// uart_driver相当于是升级版，这个是最原始的基础驱动
package driver

import (
        "context"
        "errors"

        "github.com/i4de/rulex/typex"
        serial "github.com/wwhai/goserial"
)

type rawUartDriver struct {
        state      typex.DriverState
        serialPort serial.Port
        ctx        context.Context
        RuleEngine typex.RuleX
        device     *typex.Device
}

// 初始化一个驱动
func NewRawUartDriver(
        ctx context.Context,
        e typex.RuleX,
        device *typex.Device,
        serialPort serial.Port,
) typex.XExternalDriver {
        return &rawUartDriver{
```

```
                RuleEngine: e,
                ctx:        ctx,
                serialPort: serialPort,
                device:     device,
        }
}

func (a *rawUartDriver) Init(map[string]string) error {
        a.state = typex.DRIVER_UP

        return nil
}

func (a *rawUartDriver) Work() error {
        return nil
}
func (a *rawUartDriver) State() typex.DriverState {
        return a.state
}
func (a *rawUartDriver) Stop() error {
        a.state = typex.DRIVER_STOP
        return a.serialPort.Close()
}

func (a *rawUartDriver) Test() error {
        if a.serialPort == nil {
                return errors.New("serialPort is nil")
        }
        _, err := a.serialPort.Write([]byte("\r\n"))
        return err
}

func (a *rawUartDriver) Read(cmd int, b []byte) (int, error) {
        return a.serialPort.Read(b)
}
```

```
func (a *rawUartDriver) Write(cmd int, b []byte) (int, error) {
        return a.serialPort.Write(b)
}
func (a *rawUartDriver) DriverDetail() typex.DriverDetail {
        return typex.DriverDetail{
                Name:        "Raw Uart Driver",
                Type:        "RAW_UART",
                Description: "Raw Uart Driver",
        }
}
```

上述案例展示了一个基于串口来读写数据的驱动，该案例可以用来做常见串口设备的基础驱动。

4 总结

本文档主要讲述了 RULEX 边缘网关开发框架的高级用法，其中重点讲解了 LUA 引擎、设备、驱动三者的细节以及简单示例。

1 名词理解

最后我们将将开发中常见名词整理进表格1，供给参考理解。

表格 1.

名词	解释
MDevice	表示是静态数据，保存在数据库里面的，相当于配置
Device	MDevice加载到的内存映射，也是静态的
XDevice	实际上这个静态模型的实现，这才是真实工作者
Driver	实际的工作态驱动
XExternalDriver	通用驱动接口，驱动必须实现

2 开发建议

1. 驱动永远关注“读”和“写”，而不要关注读出来的是不是对的
2. 设备只管问驱动要数据，然后将其加工成应用格式，设备不关注读写细节
3. 资源只管数据流向而不用管怎么来的，只管输送或者接收

设备、资源本质上是两类东西，设备是真实物理器材，资源是逻辑存在的软件源或者目标。