APRUS Lua-数据中心 配置说明

概述

数据中心是 Aprus 适配器内建的数据交换层，内部可容纳多个数据堆，下称为数据库。这些数据库可以作为某些数据采集模块的数据中间层，也可以作为 lua 的临时数据区使用。另外，数据中心内的数据库是 Aprus 提供 Modbus 服务器的数据基础。数据中心为 Aprus 核心程序的内置功能，在使用的时候不需要 lua 引用外部模块，也不需要初始化，直接使用其提供的 lua 接口函数即可。

数据中心中的数据库包含以下几个特征：名称、起始地址、单元长度、数据容量。

❶名称：长度在 30 个字符内的字符串，可用此名称来获取数据库的句柄以便更深入地操作数据库。

❷起始地址：数据库内第一个数据使用的序号。注意：一个数据库内的数据可以从 0 开始，也可以从任意大于 0 的数值开始。小于此数值的序号的数据在数据库中并不存在，对这样序号数据的读写也不会成功。

❸单元长度：每一个数据单元所占用的字节数。对数据单元的一般操作都是以单元长度为操作单位的。比如单元长度为 2，那每次写入和读出都是读写 2 个字节。通常情况下，一个整形数据或浮点型数据占用 4 个字节，普通字符占用两个字节；一个字符占用 1 个字节；Modbus 中的保持寄存器和输入寄存器是短整型或无符号短整型数据，占用 2 个字节；Modbus 中的线圈和输入离散量在 Modbus 协议中只占用 1bit，但是数据中心的最小操作单位是 1 字节，所以如果数据库用于 Modbus 协议中这两种数据类型的数据交换时，单元长度应该设置为 1 个字节。

❹数据容量：数据库中数据单元的数量。当起始地址和数据容量确定后，数据库内可供索引的数据序号为[起始地址 ~ （起始地址+数据容量-1）]。例：起始地址为 5，数据容量为 10，那么可供索引的序号为 5、6、7、8、9、10、11、12、13、14，共 10 个数据。

数据中心支持添加、删除数据库，支持更改现有数据库的大小，支持读取、写入数据库中任意位置的数据。

1 APRUS.lua 配置说明

```
package.cpath="./?.so"
package.path="./?.lua"
cjson = require "cjson"
config = require "config"
require "AXio"
require "ModbusServer"

function ModbusServer_init()
	mbSvrObj = ModbusServer.new("ModbusServer")
	local MBCfg = cjson.encode(config.ModbusServer.ServerConfig)
	local ret = ModbusServer.config(mbSvrObj, MBCfg)
	--ModbusServer.debug(mbSvrObj, 0x01010101)   --ModbusServer 显示详尽的调试信息
end

function MBserver_load_station(svr, cfg)
	for k,v in pairs(cfg)
	do
		print("load Station" .. cjson.encode(v))
		ModbusServer.setStation(svr, cjson.encode(v))
	end
end

function updateU16(num16)
		--小于 65535 的数，只需要保存在一个数据格子内就可以了。
		DataCenter.DBsetRecord(dbAXio_AI, 17 , num16)
end

function updateU32(num32)
		--大于 65535 的数，需要分段保存在两个或更多个数据格子内。
		--num32 = 0x12345678
		t1 = math.floor(num32 % 0x10000)		--取余
		th,f = math.modf(num32 / 0x10000)		--取整
		DataCenter.DBsetRecord(dbAXio_AI, 16 , th)
		DataCenter.DBsetRecord(dbAXio_AI, 17 , t1)
end

function act_control(m, json)
	for k,v in pairs(json) do
		if k ~= "Act" then
		AXio.write(axioObj, k, v)
		end
	end
end

function mqttdata_handle(m, topic, data)
```

```
	local json = cjson.decode(data)
	if json.Act == "Control" then
		act_control(m, json)
	end
end

function mqttsys_handle(m, code)
	if code == 0 then
--		AXio.stop(axioObj)
	elseif code == 1 then
		--AXio.run(axioObj)
	end
end

function ioevent_handle(session, code, style_L, val_L, style_E, val_E, z)
	if code == 15 then
		local json = cjson.encode(iobuf[session])
		mqtt.publish(mqtt30bj, iostr[session], "r", json)
		iobuf[session] = {}
	elseif code > 10 then
		iobuf[session][style_L] = val_L
		iobuf[session][style_E] = val_E
	else
		local json = cjson.encode({[style_L] = val_L, [style_E] = val_E, ["Z"] = z})
		mqtt.publish(mqtt30bj, iostr[session], "r", json)
	end
end

function io_load_nodes(session, nodes)
	for k,v in pairs(nodes)
	do
		AXio.add_node(session, v.m, v.index, v.interval, v.dtype, v.cntmode, v.pMode, v.dStyle, v.dOffset, v.dExt)
		--AXio.add_node(session, v.m, v.index, v.direction, v.interval, v.dtype, v.cntmode, v.pMode, v.dStyle,
v.dOffset, v.dExt)
	end
end

function loadDatabase(dbs)
	for k,v in pairs(dbs)
	do
		DataCenter.addDB(cjson.encode(v))
	end
end

function init()
```

```
	mqtt30bj = mqtt.new()
	user.setluaver(config.AprusX.luaver)
	user.setdevinfo(config.AprusX.devinfo)
	user.ipconfig(config.AprusX.ipmode, config.AprusX.inet_addr, config.AprusX.netmask)
end

function start()
	print("<--------------------user lua start------------------------>")
	init()
	print("<--------------------DataCenter_load_database()------------------------>")
	loadDatabase(config.DataCenter)

	IOCfg = config.AXIO
	axioObj = nil
	iobuf = {axioObj}
	iostr = {axioObj}

	print("<--------------------AXio.new()------------------------>")
	axioObj = AXio.new()
	dbAXio_AI = DataCenter.getDB("AXio_AI")
	DataCenter.DBresize(dbAXio_AI,20)
	iobuf[axioObj] = {}
	iostr[axioObj] = "io"
	print("<--------------------io_load_nodes()------------------------>")
	io_load_nodes(axioObj, IOCfg.node)

	print("<--------------------ModbusServer_init()------------------------>")
	ModbusServer_init()
	print("<--------------------ModbusStation_init()------------------------>")
	MBserver_load_station(mbSvrObj, config.ModbusServer.Stations)
	print("<--------------------ModbusServer.run------------------------>")
	ModbusServer.run(mbSvrObj)
	AXio.run(axioObj)
	mqtt.run(mqtt30bj)

	counter = 0
	while true do
		local msg = user.waitmsg()
		if msg.from == "mqtt-sys" then
			mqttsys_handle(msg.session, msg.code)
		elseif msg.from == "mqtt-msg" then
			mqttdata_handle(msg.session, msg.topic, msg.payload)
		elseif msg.from == "io" then
			ioevent_handle(msg.session, msg.code, msg.style_L, msg.val_L, msg.style_E, msg.val_E, msg.z)

			counter = counter+1
```

```
                DataCenter.DBsetRecord(dbAXio_AI, 18 , counter)
                --updateU32（counter）
                --print("update counter" .. counter)
            end
        end
end
start()
```

1.1 此处仅解释一下数据中心相关的函数和操作：

1、loadDatabase(config.DataCenter)：
使数据中心加载 config.lua 中 DataCenter 段配置的数据库。

函数原型如下（使用循环方式添加 DataCenter 段的每一行）：

```
function loadDatabase(dbs)
    for k,v in pairs(dbs)
    do
        DataCenter.addDB(cjson.encode(v))
    end
end
```

2、dbAXio_AI = DataCenter.getDB("AXio_AI")：
通过名称"AXio_AI"获取一个数据库的句柄。（AXio_AI 为 AXio 模块内部自带的数据库，AXio 模块初始化后此数据库就已经就绪，可以直接使用）

3、DataCenter.DBresize(dbAXio_AI,20)：
重设数据库的数据容量为 20 个（AXio_AI 的原始长度为 12 个，现在增加到 20 个，方便将一些运算结果放置到此位置）。重设数据库的数据容量时，如果新容量比旧容量小，那么旧数据库中多出的部分就不再存在，里面保存的数据也会丢失。另外，对于其他模块的内置数据库，最好只增大容量，不减小容量，以免影响其正常运行。

4、DataCenter.DBsetRecord(dbAXio_AI, 18 , counter)：

将 dbAXio_AI 这个数据库的 18 号数据单元的值设置为变量 counter 的值。设置数值时有两点需要注意：一是索引序号必须在数据库的索引范围内；二是放入的数据类型与数据库本身单元长度的匹配关系。

1.2 DataCenter API

1.2.1 DataCenter.addDB

功能	添加新的数据库
接口描述	DataCenter.addDB(arg)
arg	数据库属性描述<字符串>
例	DataCenter.addDB("dbName="DB_3_100", dStart=0, count=100, dSize=2, dType=2")

dbName	数据库名称<字符串>，用来查找、索引数据库
dStart	数据单元起始编号（Modbus 等需要使用。默认为 0）
count	数据单元数量
dSize	数据单元的大小（占用字节数）
dType	数据单元的类型（暂时无作用）

1.2.2 DataCenter.getDB

功能	获取数据库指针
接口描述	pDB = DataCenter.getDB(dbName)
dbName	要查找的数据库名称<字符串>
pDB	返回值<数据库指针>。如果未找到，则返回 NULL

1.2.3 DataCenter.removeDB

功能	删除数据库
接口描述	DataCenter.removeDB(dbName)
dbName	要删除的数据库名称<字符串>

1.2.4 DataCenter.removeAllDB

功能	删除所有数据库
接口描述	DataCenter.removeAllDB()

1.2.5 DataCenter.debug

功能	设置数据库 LOG 打印级别
接口描述	DataCenter.debug(LogLevel)
LogLevel	0x00000101

1.2.6 DataCenter.DBresize

功能	重设数据库的数据单元数量
接口描述	DataCenter.DBresize(pDB, count)
pDB	数据库指针
count	数据单元数量（如果新数量比原来的数量小，多余的数值将会丢失）

1.2.7 DataCenter.DBdestroy

功能	销毁数据库内部的资源
接口描述	DataCenter.DBdestroy(dbName)
dbName	要销毁的数据库名称<字符串>

1.2.8 DataCenter.DBclear

功能	清空数据库内所有数据
接口描述	DataCenter.DBclear(dbName)
dbName	要清空的数据库名称<字符串>

1.2.9 DataCenter.DBupdateRecord

功能	更新数据库中某个数据单元（附带更新时间）
接口描述	DataCenter.DBupdateRecord(pDB,index,value,time)
pDB	数据库指针
index	数据单元编号
value	新的数值
time	新的时间<Unix 时间戳（u32 类型）>

1.2.10 DataCenter.DBupdateRecords

功能	更新数据库中多个数据单元（附带更新时间）
接口描述	DataCenter.DBupdateRecords(pDB,index,count,pvalue,time)
pDB	数据库指针
index	数据单元编号
count	更新的数据单元数量
pvalue	新的数值指针（必须与数据库中数据同样字长）
time	新的时间<Unix 时间戳（u32 类型）>

1.2.11 DataCenter.DBsetRecord

功能	更新数据库中某个数据单元
接口描述	DataCenter.DBsetRecord(pDB,index,value)
pDB	数据库指针
index	数据单元编号
value	新的数值

1.2.12 DataCenter.DBsetRecords

功能	更新数据库中多个数据单元

接口描述	DataCenter.DBsetRecords(pDB,index,count,pvalue)
pDB	数据库指针
index	数据单元编号
count	更新的数据单元数量
pvalue	新的数值指针（必须与数据库中数据同样字长）

1.2.13 DataCenter.DBgetRecord

功能	**获取数据库中某个数据单元的数值**
接口描述	ret,val = DataCenter.DBgetRecord(pDB,index)
ret	函数返回值，ret=0 时证明函数运行无错，val 有效
val	数据单元的数值
pDB	数据库指针
index	数据单元编号

1.2.14 DataCenter.DBgetRecords

功能	**获取数据库中多个数据单元的数值**
接口描述	ret = DataCenter.DBgetRecords(pDB,index,count,pvalue)
ret	函数返回值，ret=0 时证明函数运行无错，pvalue 内数据有效
pDB	数据库指针
index	需要获取的起始数据单元编号
count	需要获取的数据单元数量
pvalue	输出数值指针，获取的多个数据会被放置到这个数组中。（必须与数据库中数据同样字长）

2 config.lua 配置说明

```
DataCenter=
{
        --{dbName="名称"（必要）, dStart=起始地址（必要）, count=数据容量（必要）, dSize=数据单元长度（必要）, dType=数据类型（暂时无效、非必要)},          --自定义数据库
        {dbName="DB_1_100", dStart=0, count=100, dSize=1, dType=1},
        {dbName="DB_2_100", dStart=0, count=100, dSize=1, dType=0},
        {dbName="DB_3_100", dStart=0, count=100, dSize=2, dType=2},
        {dbName="DB_4_100", dStart=0, count=100, dSize=2, dType=4},
},
```