

实时数据库控制任务

(使用说明书)

编写：蒋勇

审核：

日期：

2025.4.5 增加增量和积分统计 API 接口脚本函数。

2023.12.01 增加两个时标转换函数。

2022.12.31 增加 aarch64 平台描述。

2021.9.26 增加三个数值转换函数。

2021.9.18 增加三个统计函数。

2021.8.29 增加人工置数函数。

2020.7.11 V1.3 增加双机冗余部署支持

2020.4.4 v1.2 增加 JS 文件操作对象 cFile

2019.7.27 v1.1 增加 JS 对字符串标签的处理

2019.5.12 v1

重庆唐码软件有限公司

2025.4

内容目录

1 概述.....	1
1.1 运行平台.....	1
1.2 控制任务定义.....	2
2 安装.....	3
2.1 windows 平台安装.....	3
2.2 Linux 系统中安装.....	6
3 脚本系统.....	7
3.1 变量.....	7
3.2 立即数.....	8
3.3 循环.....	8
3.4 条件分支.....	9
3.5 switch 分支.....	10
3.6 内置类型.....	10
3.7 函数.....	11
3.8 脚本自定义对象.....	11
3.9 动态对象.....	12
4 控制任务中脚本规范.....	12
4.1 脚本编码.....	13
4.2 控制入口函数.....	13
4.3 初始化.....	13
4.4 扩展函数.....	14
4.4.1 输出到日志.....	14
4.4.2 值类型对象.....	14
4.4.3 读取实时库标签快照.....	17
4.4.4 写实时库标签快照.....	17
4.4.5 写设备控制输出.....	18
4.4.6 获取本地当前实时库时标.....	19
4.4.7 实时库时标转换.....	20
4.4.8 日期时间对象.....	20
4.4.9 字符串类型对象.....	22
4.4.10 读取实时库字符串标签快照.....	25
4.4.11 写实时库字符串标签快照.....	26
4.4.12 写设备控制输出字符串.....	26
4.4.13 文件操作对象 cFile.....	27
4.4.14 人工置数.....	32
4.4.15 统计值结果对象 r_count.....	33
4.4.16 值统计.....	33
4.4.17 状态改变结果对象 r_statuschange.....	34
4.4.18 状态改变统计.....	34
4.4.19 值运行时间结果对象 r_valtime.....	34
4.4.20 值运行时间统计.....	35
4.4.21 数值转换函数.....	35
4.4.22 增量统计函数.....	35
4.4.23 积分统计函数.....	35
5 脚本测试工具.....	37

1 概述

实时库控制任务(rdbctrl)以后台服务方式运行，管理和执行定时任务和周期性任务。任务采用类javascript 脚本编写，脚本提供快照读，快照写，写设备等接口。主要有以下几种用途：

- 利用控制任务可实现比如定时开启或停止设备，比如晚上开启路灯，早上关闭。
- 可实现设备之间的联动，比如根据隧道瓦斯浓度调节排风扇速度。
- 实现应用级的复杂计算标签，将多个实际标签定时计算后通过写快照方式写入到实时库的一个计算标签。
- 实现设备之间的数据传输，将一个设备的数据写入到另一个设备，或者将几个设备的数据计算后写入到另一个设备。

1.1 运行平台

rdbctrl 提供 windows/Linux(x86-64 和 aarch64)版本，均为 64 位后台服务版。

后台服务运行环境

Windows:

最低要求：Windows server2008，windows7 即以上 64 位系统。推荐 windows server 2016 x64 版。

Linux(X86-64):

内核 3.10 及以上，libstdc++.so.6.0.19, glibc 2.17 及以上；典型系统 centos7.6;推荐 centos8 和 ubuntu server1804

Linux(aarch64)

内核 4.4.131 及以上，libstdc++.so.6.0.21, glibc 2.23，典型系统银河麒麟 V10 系统,相当于 ubuntu16.04 的 aarch64 系统平台。

前端管理平台

采用 http/https 的 web 页面管理，实现任务的启停，增删，配置更改，查看日志，查看任务脚本。

WEB 管理界面可随时退出，不影响后台任务运行，由于采用了一些新的 WEB 前端技术，前端浏览器只支持 google 的 chrome 浏览器或者新版的 Edge 浏览器。

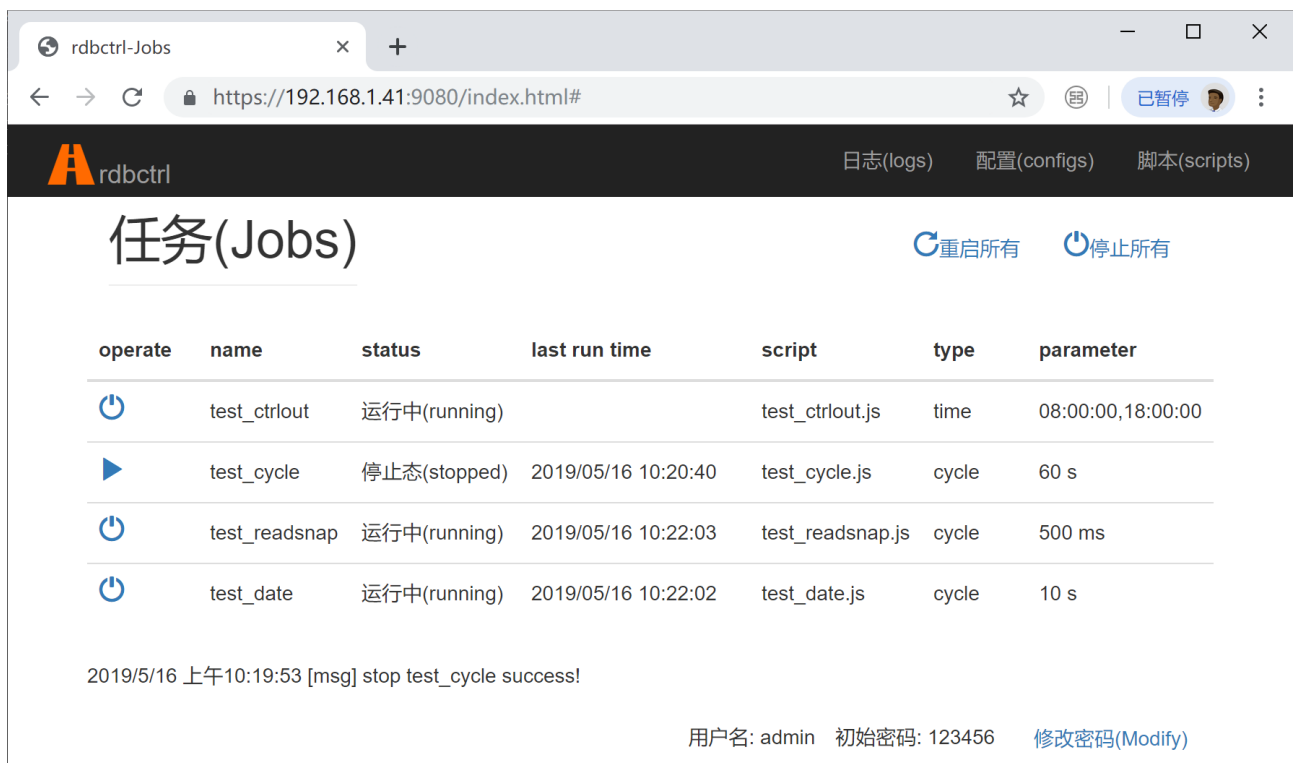


图 1 WEB 管理维护界面

1.2 控制任务定义

一个控制任务静态属性有：

- 任务名；全局唯一的一个字符串名称，不能含有中文字符。小于 40 字符。
- 类型；分为定时执行(time)和周期执行(cycle)。定时执行为指定的时刻(格式 hh:mm:ss)运行，最多可以有 24 个时刻。周期执行为每隔指定的秒数数执行一次，比如配置 60 秒表示每分钟执行一次。
- 脚本文件；每个控制任务有且只有一个脚本文件，控制逻辑编写在脚本文件里。脚本文件为后缀为.js 的简化版的 javascript 动态脚本。

动态属性包括：

- 运行状态；运行(running)和停止(stopped)，上次运行时间
- 操作接口；启动和停止。

2 安装

包含 windows 和 linux 平台安装，windows 平台推荐 windows server2012 及以上 64 位版本，linux 推荐 centos 7.5 及以上 64 位版本，当然也支持 ubuntu1604 及以上。windows 和 linux 平台默认安装的裸系统，无需安装其他软件包。

2.1 windows 平台安装

将 win/rdbctrl 目录复制到 c:/rdbctrl 目录下，先配置好 rdbctrl.ini 文件，配置日志的输出目录和日志输出级别，http/https 服务端口和证书。

如果配置了证书，则会启动 https 同时关闭 http 服务。没有配置证书则只能启动 http 服务。

参考下面的默认配置，注意'#'和';'都表示到行尾是注释。

```
# config for rdbctrl
```

```
[log]
```

```
path = c:/rdbctrl/log
```

```
level = msg # err, wrn, msg, dbg
```

```
[http]
```

```
port = 9080 # 0 as not use
```

```
ca_root = ;
```

```
ca_server = # /etc/rdbctrl/ca/rdbctrl.cer
```

```
private_key = #/etc/rdbctrl/ca/rdbctrl.key
```

```
[admin]
```

```
pswd = 123456 #Administrator password
```

```
[HA] #高可用双机冗余配置
```

```
role = master ; master 为主机，slave 为从机，不配置是独立主机
```

```
heartline = 192.168.100.9:925/ctrl_heartid ; 心跳线，主机提供心跳服务，从机作为客户端连接
```

再在配置 ctrljob.ini 文件，在[rdb]小节配置实时库：

```
[rdb] #主实时库主机
```

```
url = ws://192.168.1.58:921
```

```
user = admin
```

```
pswd = admin
```

```
[rdbslave] #实时库从机，可以不配置  
url = ws://192.168.1.90:921  
user = admin  
pswd = admin
```

在[ctrl]小节里配置控制任务，可以有多个控制任务，最后用一个空的[ctrl]小节结束。

```
[ctrl] #控制任务，可以有多个 ctrl 小结  
name = test_ctrlout #控制任务名称，唯一的名字,最大 39 字符,不要用中文  
runtype = time #指定时刻执行  
runtime = 8:00:00,18:00:00 #表示 8 点和 18 点执行一次，最多 24 个时间  
script = test_ctrlout.js #脚本文件全路径
```

```
[ctrl]  
name = test_cycle #综合例子  
runtype = cycle #周期执行  
runtime = 60 #多少秒执行一次,不带单位是秒  
script = test_cycle.js #相对路径文件名
```

```
[ctrl]  
name = test_readsnap #读快照  
runtype = cycle #周期执行  
runtime = 500ms #多少秒执行一次,带单位 ms 表示毫秒  
script = test_readsnap.js #相对路径文件名
```

```
[ctrl]  
name = test_date #测试日期函数  
runtype = cycle #周期执行  
runtime = 10 #多少秒执行一次  
script = test_date.js #相对路径文件名
```

[ctrl] #最后用一个空的结束

注意 **name** 不要重复，且不能是中文名。**runtime** 如果是 **time** 则 **runtime** 的多个时刻之间用西文逗号分开。

runtime 如果要指定毫秒周期，可以使用单位 **ms**，如上面例子所示，最小配置 100 毫秒即可，小于 100 毫秒无多大意义。

这个 **ctrljob.ini** 文件在运行后也可用通过 **web** 管理页面提交。

最后在管理员权限下打开命令行或者 **powershell** 窗口，在 **c:/rdbctrl** 目录下执行如下命令：

```
rdbctrl -install
```

安装服务。如下图所示：



图 2 命令行安装服务

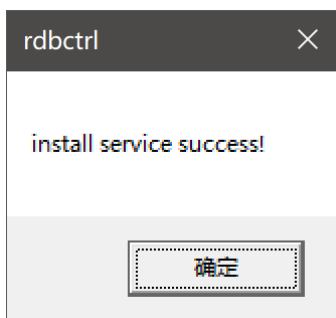


图 3 服务安装成功提示

看到安装成功的提示框后，到系统的"服务"中开启服务，该服务默认设置为自动启动并开启故障恢复。



图 4 rdbctrl 服务管理

启动后，查看 c:/rdbctrl/log 下的日志是否有错误。如果一切正常，可以使用 google 浏览器 chrome(由于使用了一些新的 WEB 前端技术，不支持其他浏览器，包括 firefox 也不支持)在地址栏输入如下 url 地址：

http://127.0.0.1:9080

进入 WEB 管理维护界面。将看到如图 1 所示的界面。默认账号为"admin",密码为"123456",引号中的字符串。

2.2Linux 系统中安装

在 Linux 系统中将会安装为 systemd 守护进程，支持开启自动启动和故障恢复。按照下列步骤安装：

- (1) 按照目录结构将文件拷贝到 linux 系统(rdbctrl.service 除外)。拷贝完成后，和 windows 系统相同的方式配置 Linux 目标系统/etc/rdbctrl/rdbctrl.ini 和/etc/rdbctrl/ctrljob.ini 两个文件。同样有证书开启 https，没有证书开启 http。
- (2) 为/usr/sbin/rdbctrl 添加可执行权限：sudo chmod +x /usr/sbin/rdbctrl
- (3) 将 rdbctrl.service 拷贝到/etc/systemd/system 下,然后执行 sudo systemctl enable rdbctrl 和 sudo systemctl daemon-reload 启动 systemd 守护进程后台服务。
- (4) 安装完成后按照如下命令启停和查看状态
启动：
sudo systemctl start rdbctrl

或者

```
sudo rdbctrl -start
```

停止:

```
sudo systemctl stop rdbctrl
```

或者

```
sudo rdbctrl -stop
```

状态

```
sudo rdbctrl -status
```

版本

```
sudo rdbctrl -ver
```

(5) 运行日志

查看运行日志在 `/home/rdbctrl/log` 目录中。

3 脚本系统

脚本采用简化版的 javascript(ECMAScript 标准), 和编写普通的 JS 脚本类似。虽然可以不需要代码行最后的分号';' 但是为了规范和避免出错, 脚本代码行结束跟分号';'

3.1 变量

`var i;` // 未初始化的变量, 可以在第一次分配时获取任何值;

`auto j;` // 相当于 `var`

`var k = 5;` // 初始化为 5 (整数)

`var l := k;` // k 的引用

`auto &m = k;` // k 的引用

`global g = 5;` // 创建一个全局变量。 如果全局已存在, 则不会重新添加

`global g = 2;` // 全局变量'g'现在等于 2 了(重复)。

`global g2;`

`if (g2.is_var_undef()) { g2 = 4; } // 只初始化一次`

GLOBAL g3; // 大写 GLOBAL 也行

3.2 立即数

浮点值默认为 double 类型，整数默认为 int 类型。支持所有 C++ 后缀，如 f, ll, u 以及科学记数法。

1.0 // double

1.0f // float

1.0l // long double

1 // int

1u // unsigned int

1ul // unsigned long

1ull // unsigned long long

值是自动调整大小的，就像在 C++ 中一样。例如：10000000000 > 32bits，并使用适当的类型将其保存在您的平台上。

3.3 循环

// c-style for loops

```
for (var i = 0; i < 100; ++i) {  
    print(i);  
}
```

// while

```
while (some_condition()) {  
    /* do something */  
}
```

// ranged for

```
for (x : [1,2,3]) {  
    print(i);  
}
```

可以使用 break 语句打破每个循环样式。例如：

```
while (some_condition()) {
    /* do something */
    if (another_condition()) {
        break;
    }
}
```

注意循环体或者代码块要用大括号包括起来，即使只有一行也是，这点和 C/C++ 不一样。如下是错误的：

```
for (var i = 0; i < 100; ++i)
    print(i);
```

3.4 条件分支

```
var i = 0;
if (i == 0) {
    print("i is 0");
}
else if (i == 1) {
    print("i is 1");
}
else {
    print("i is not 0 or 1");
}
```

同样，每个分支下的代码块需要使用大括号，即使只有一行，这点和 C/C++ 不一样。如下是错误的：

```
if (i == 0)
    print("i is 0");
```

3.5 switch 分支

```
var myvalue = 2
switch (myvalue) {
    case (1) {
        print("My Value is 1");
    }
}
```

```

        break;
    }
    case (2) {
        print("My Value is 2");
        break;
    }
    default {
        print("My Value is something else.");
    }
}

```

同样，每个分支下的代码块需要使用大括号，即使只有一行，这点和 C/C++ 不一样。还有每个 case 条件使用小括号()而不是 C/C++ 的空格和冒号。

3.6 内置类型

有 Vector 和 Map 两种集合对象。

```
var v = [1,2,3u,4ll,"16", `+`]; // 创建异构值的向量
```

```
var m = ["a":1, "b":2]; // 字符串键-值对的 map
```

```
// 添加一个值到 vector
```

```
v.push_back(123);
```

```
// 添加对象引用到 vector
```

```
v.push_back_ref(m);
```

3.7 函数

一般函数，自动类型

```
function myfun(x, y) { x + y; } // last statement in body is the return value
```

```
function myfun(x, y) { return x + y; } // equiv
```

类型优化的函数

```
function myfun(x, int y) { x + y; } // requires y to be an int
```

Lambda 表达式

```
var l = fun(x) { x * 15; }
```

```
l(2) // returns 30
```

```
var a = 13
```

```
var m = fun[a](x) { x * a; }
```

```
m(3); // a was captured (by reference), returns 39
```

```
var n = bind(fun(x,y) { x * y; }, _, 10);
```

```
n(2); // returns 20
```

3.8 脚本自定义对象

预定义的

```
class MyType {
```

```
    var value;
```

```
    def MyType() { this.value = "a"; }
```

```
    def get_value() { "Value Is: " + this.value; }
```

```
};
```

替换

```
attr MyType::value;
```

```
def MyType::MyType() { this.value = "a"; }
```

```
def MyType::get_value() { "Value Is: " + this.value; }
```

应用

```
var m = MyType(); // calls constructor
```

```
print(m.get_value()); // prints "Value Is: a"
```

```
print(get_value(m)); // prints "Value Is: a"
```

3.9 动态对象

所有脚本定义的类型和通用的 `Dynamic_Object` 都支持动态参数。用 `Dynamic_Object` 创建一个动态对象，然后往里面添加属性和方法

```
var o = Dynamic_Object();
```

```
o.f = fun(x) { print(x); }
```

```
o.f(3); // prints "3"
```

Implicit 'this' is allowed:

```
var o = Dynamic_Object();
```

```
o.x = 3;
```

```
o.f = fun(y) { print(this.x + y); }
```

```
o.f(10); // prints 13
```

4 控制任务中脚本规范

规范了入口函数和添加的实时库访问函数以及时间函数。

4.1 脚本编码

脚本文件字符编码：UTF-8 编码。支持带 BOM 头和不带 BOM 头，建议用 notepad++ 编写脚本，能清楚的控制编码和格式。

脚本文件名后缀为.js 主要是便于有些编辑器自动识别高亮关键字。

4.2 控制入口函数

任务中约定了一个函数 `function run()` 为入口函数，`run` 函数会根据任务配置被定时执行或者周期性执行，参见例子脚本 `test_cycle.js`。

不要在 `run` 里面做长时间的循环或者死循环，要控制运行节奏，可以通过周期执行读取时间方式来处理，本次没有执行条件无需循环等待，可以先退出下次继续执行。

需要重复使用的变量应该用 `global` 定义为全局变量。

4.3 初始化

`run()` 之外的脚本之在加载是执行一次，因此可以在 `run()` 之外写初始化，如下例子：

```
global g_ct; //测试全局变量,这里用于计数被调用了多少次。
```

```
function init()
```

```
{
```

```

    g_ct = 0; //初始为 0

    logout("dbg","init cycletest1.js success, g_ct = " + g_ct.to_string());
}

function run() // 主运行函数,按照配置定时或者周期性被执行，注意不要有间接或者直接死循环在里面。
{
    g_ct += 1;

    logout("dbg","cycletest1 g_ct = " + g_ct.to_string()); // 测试全局变量,调用次数输出到日志
    test_writedevice(); //测试写入到设备
    test_writesnap(); //测试写入快照
    test_rdbtime(); //测试实时库时标解析
}

init(); //初始化只会执行一次

```

4.4 扩展函数

扩展函数用于访问实时库和提供一些便利性功能。

4.4.1 输出到日志

在脚本中输出文本到 rdbctrl 的日志文件，使用 logout 脚本函数：

```
logout(slev, strlog);
```

输出字符串到日志

参数：

slev 输出级别可以为"err","wrn","msg","dbg"之一

strlog 日志内容

返回：

无返回。

例子：

```
logout("dbg","this is debug logout string");
```

```
logout("msg","this is message level logout string");
```

如果 slev 级别在 rdbctrl.ini 配置的输出级别内，日志会被写入到 rdbctrl 的日志文件中，这个函数主要用于调试和记录关键的日志事件。

为了避免不同系统下字符集默认编码不同带来的中文显示麻烦，在日志中最好不要使用中文，如果要使用中文，统一使用 UTF-8 编码的中文。

4.4.2 值类型对象

通过 `t_val()` 函数或者 `rdb_getsnapval()` 函数可以获得一个值类型对象(`t_val`)。

`var val = t_val();` //创建并返回一个值类型。

参数： 无参数

返回值： 返回一个 `t_val` 对象

`t_val` 可读写属性：

`ti` 实时库标签时标，长整型，从 1970-1-1 开始的 100 毫秒,GMT 时间。

`qa` 数据质量，整形，

0 表示正常，

1 关机,数据不可靠;

2 错误数据或数据无效

3 无此标签;

4 时标错误

`t_val` 只读属性， 值数型，下面的 6 个值属性都是同一个值的不同表达方法：

`iv int32_t` 型值

`uv uint32_t` 型值

`fv float` 型值

`lv int64_t` 型值

`ulv uint64_t` 型值

`dblv double` 型值

设置值方法：

`setval (val)`

参数：

`val`：可以是整形，长整型，float，double。

返回值： 无

例子：

创建一个 `t_val` 并填写属性：


```
var v = t_val(); //创建一个属性全 0 的 t_val 对象。
```

```
v.ti=rdb_time(); //使用当前时标
```

```
v.qa = 0; //填写数据质量
```

```
v.setval(100); //设置值为 100
```

下面用 `logout` 输出到日志看看。

```
logout("dbg", "v = " +  
" time: " + rdbtime_to_str(v.ti) +  
", QA: " + v.qa.to_string() +  
", iv: " + v.iv.to_string() +  
", uv: " + v.uv.to_string() +  
", fv: " + v.fv.to_string() +  
", lv: " + v.lv.to_string() +  
", ulv: " + v.ulv.to_string() +  
", dblv: " + v.dblv.to_string()  
);
```

时间只读属性:

下面的时间属性可以直接访问

`year` 年, 整数, 比如 2019

`month` 月, 整数, [1,12]

`day` 日, 整数 [1,31]

`hour` 时, [0,23]

`minute` 分, 整数 [0-59]

`second` 秒, 整数 [0-59]

`millisecond` 毫秒, 整数[0-999]

例子:

```
var v = t_val(); //创建一个属性全 0 的 t_val 对象。
```

```
v.ti=rdb_time(); //使用当前时标
```

```
logout("dbg", "v.year=" + v.year.to_string());
```

设置时标

`settime(year,month,day,hour,minute,second,millisecond)`

参数:

year 年, 整数, 比如 2019

month 月, 整数, [1,12]

day 日, 整数 [1,31]

hour 时, [0,23]

minute 分, 整数 [0-59]

second 秒, 整数 [0-59]

millisecond 毫秒, 整数[0-999]

返回值: 0 表示成功, -1 失败

4.4.3 读取实时库标签快照

`rdb_getsnapval(tagname)`

参数:

tagname 标签名, 字符串

返回值:

返回一个 `t_val` 类型, 当 `qa` 不为 0 时表示有错误, 见 4.3.1 中的 `qa` 定义。

例子 1:

```
var val = rdb_getsnapval("m1.int1"); //读取标签值
```

例子 2:

```
var stag = "m1.int1";
```

```
var val = rdb_getsnapval(stag); //读取标签值
```

4.4.4 写实时库标签快照

写标签快照到实时库

`rdb_writesnap(tagname,time,qa,value)`

参数:

tagname : 标签名, utf-8 字符串, 不建议中文标签名。

time : rdb 格式时标, 如果此参数为 0 或 0 的长整数立即数表示), 则使用当前时间

qa : 数据质量,

0 表示正常;

- 1 关机,数据不可靠;
- 2 错误数据或数据无效;
- 3 无此标签;
- 4 时标错误

value : 可以是 int32_t,uint32_t,int64_t,uint64_t,float,double 之一

返回值:

返回 0 表示成功, 其他为错误码(错误码和 C/C++接口, java 接口等定义的相同)。

例子:

```
function test_writesnap() //测试写入快照到实时库
{
    var stag = "tst.int01";

    var vr = rdb_getsnapval(stag); //先读取标签值

    log_tagval("cycletest1," + stag, vr); // 输出到日志看是否正确

    var v = vr.iv;

    if(vr.qa == 0){
        v += 1; //每次增长 1
    }

    else{
        v = 1; //如果没有初值, 设置初始化值为 1
    }

    var nret = rdb_writesnap(stag, 0ll , 0 , v); //写入

    if(nret != 0){
        logout("err","writesnap " + stag + " failed, error code = " + nret.to_string());
    }

    else{
        logout("msg","writesnap " + stag + " success");
    }
}
```

4.4.5 写设备控制输出

控制输出，写值到设备。

`rdb_writedevicetagname,v`;

控制输出到设备

参数：

`tagname`：标签名，utf-8 字符串，不建议中文。

`v`：可以是 `int32_t,uint32_t,int64_t,uint64_t,float,double` 之一

返回值：返回 0 表示成功，其他为错误码(错误码和 C/C++接口，java 接口等定义的相同)。

例子：

`function test_writedevicetagname,v` //测试写入值到设备

```
{
    var stag = "m1.f4tof4rw";
    var vr = rdb_getsnapval(stag); //先读取标签值
    log_tagval("cyclctest1," + stag,vr); // 输出到日志看是否正确
    var v = vr.dblv;
    if(vr.qa == 0){
        v += 0.5; //每次增长 0.5
    }
    else{
        v = 1.0; //如果没有初值，设置初始化值为 1.0
    }
    var nret = rdb_writedevicetagname,v; //写入
    if(nret != 0){
        logout("err","write device " + stag + " failed, error code = " + nret.to_string());
    }
    else{
        logout("msg","write device " + stag + " success");
    }
}
```

4.4.6 获取本地当前实时库时标

获取本地环境当前时间，实时库时标格式，主要用于 `t_val` 对象的 `ti` 属性。

`rdb_time()` ;

参数：无

返回：返回实时库时标格式读取当前时间，即从给 1970-1-1 开始的 100 毫秒,GMT 时间。

4.4.7 实时库时标转换

实时库时标和字符串之间的互转。

`rdbtime_to_str(timeval)`

实时库时标转换为字符串。

参数： `timeval` 实时库时标格式读取当前时间，从给 1970-1-1 开始的 100 毫秒,GMT 时间

返回值：字符串，格式 "2019-4-29 12:32:56.200"

`rdbtime_to_isostr(timeval)`

实时库时标转换为字符串，2023-12 发布的 1.0.1.9 版增加。

参数： `timeval` 实时库时标格式读取当前时间，从给 1970-1-1 开始的 100 毫秒,GMT 时间

返回值：字符串，格式 "2019-4-29T12:32:56.200+08:00"

`str_to_rdbtime(str)`

字符串换为实时库时标，2023-12 发布的 1.0.1.9 版增加 ISO 时标格式支持。

参数： `str timeval` 字符串时标 格式 "2019-4-29 12:32:56.200"

返回值：实时库时标格式读取当前时间，从给 1970-1-1 开始的 100 毫秒,GMT 时间

4.4.8 日期时间对象

`date` 是扩展的日期时间对象，精度为毫秒。

创建一个 `date` 对象，有 5 个版本。

`date(strdate)`

`date(year,month,day)`

`date(year,month,day,hour,minute,second)`

`date(year,month,day,hour,minute,second)`

`date(year,month,day,hour,minute,second,millisecond)`

参数：

strdate: 字符串格式的日期时间，格式"年-月-日 时:分:秒.毫秒"，其中时间部分和毫秒部分可选，2023-12 发布的 1.0.1.9 版增加 ISO 时标格式支持。

例 1: "2019-5-11"

例 2: "2019-5-11 20:32:46"

例 3: "2019-5-11 20:32:46.345"

例 4: "2019-5-11T20:32:46.345+08:00"

year 年，整数，比如 2019

month 月，整数，[1,12]

day 日，整数 [1,31]

hour 时，[0,23]

minute 分，整数 [0-59]

second 秒，整数 [0-59]

millisecond 毫秒，整数[0-999]

方法:

[to_string\(\)](#)

转换为字符串

参数: 无

返回: 返回字符串，格式"2019-5-11 20:32:46.345"

[toISOString\(\)](#)

转换为 ISO 字符串

参数: 无

返回: 返回字符串，格式"2019-5-11T20:32:46.345+08:00"

直接访问的只读属性

[year](#), [month](#), [day](#), [hour](#), [minute](#), [second](#), [millisecond](#)

综合例子,参见 [datetest.js](#)

[function testdate\(\)](#)

{

[var now = date\(\);](#)//当前时间

[logout\("dbg","test date 日期时间对象"\);](#)

```

logout("dbg","now = " + now.to_string());
var date1 = date(2019,5,6); //年，月，日
logout("dbg","date(2019,5,6) = " + date1.to_string());
var date2 = date(2019,5,6,12,32,24); //年，月，日，时，分，秒
logout("dbg","date(2019,5,6,12,32,24) = " + date2.to_string());
var date3 = date(2019,5,6,12,32,24,360); //年，月，日，时，分，秒，毫秒
logout("dbg","date(2019,5,6,12,32,24,360) = " + date3.to_string());
var date4 = date("2019/5/6 8:25:36"); // 日期时间字符串
logout("dbg","date(\"2019/5/6 8:25:36\") = " + date4.to_string());
var date5 = date("2019/5/6 8:25:36.100"); // 日期时间字符串，带毫秒
logout("dbg","date(\"2019/5/6 8:25:36.100\") = " + date5.to_string());
var date6 = date("2019-5-6 18:25:36.100"); // 日期时间字符串，带毫秒
logout("dbg","date(\"2019-5-6 18:25:36.100\") = " + date6.to_string());
logout("dbg","test read only attributes");
logout("dbg","date6.year = " + date6.year.to_string());
logout("dbg","date6.mouth = " + date6.month.to_string());
logout("dbg","date6.day = " + date6.day.to_string());
logout("dbg","date6.hour = " + date6.hour.to_string());
logout("dbg","date6.minute = " + date6.minute.to_string());
logout("dbg","date6.second = " + date6.second.to_string());
logout("dbg","date6.millisecond = " + date6.millisecond.to_string());
var dateiso = date("2023-12-1T11:37:20.100Z");
logout("dbg",logflag() + "dateiso.toISOString() = " + dateiso.toISOString());
}

```

4.4.9 字符串类型对象

用于方便脚本访问实时库中的字符串类型(DT_STRING)类型标签。

通过 `t_str()` 函数或者 `rdb_getsnapstr()` 函数可以获得一个值类型对象(`t_str`)。

`var val = t_str();` //创建并返回一个值类型。

参数： 无参数

返回值: 返回一个 t_str 对象

t_str 可读写属性:

ti 实时库标签时标, 长整型, 从 1970-1-1 开始的 100 毫秒,GMT 时间。

qa 数据质量, 整形,

0 表示正常,

1 关机,数据不可靠;

2 错误数据或数据无效

3 无此标签;

4 时标错误

t_val 方法:

getstr()返回一个字符串。

setstr(string) 设置字符串到 t_str 对象。

时间只读属性:

下面的时间属性可以直接访问

year 年, 整数, 比如 2019

month 月, 整数, [1,12]

day 日, 整数 [1,31]

hour 时, [0,23]

minute 分, 整数 [0-59]

second 秒, 整数 [0-59]

millisecond 毫秒, 整数[0-999]

综合例子:

```
/*!
```

```
\file test_str.js
```

测试字符串 t_str 对象

```
*/
```

```
global g_jobname = "test_str";
```

```
function logflag() //日志标志, 用于在日志中却分是哪个脚本产生的日志
```

```
{
```



```

        return "[" + g_jobname + "] ";
    }
function teststr() //测试字符串对象
{
    var v = t_str();
    v.setstr("set string");
    logout("dbg", logflag() +
"\n time: " + rdbtime_to_str(v.ti) +
"\n QA : " + v.qa.to_string() +
"\n str : " + v.getstr()
    );
}
function getsnapstr(string stag) //读取字符串标签快照
{
    var v = rdb_getsnapstr(stag);
    logout("dbg", logflag() +
"\n time: " + rdbtime_to_str(v.ti) +
"\n QA : " + v.qa.to_string() +
"\n str : " + v.getstr()
    );
}
function writesnap(string stag,string vstr)
{
    var nret = rdb_writesnapstr(stag,0,0,vstr);//默认使用当前时间和0质量
    if(nret != 0){
        logout("err",logflag() + "rdb_writesnapstr(" + stag + ") error " + nret.to_string());
    }
    else{
        logout("dbg",logflag() + "rdb_writesnapstr(" + stag + ") success.");
    }
}

```

```

}
function writedevice(string stag,string vstr)
{
    var nret = rdb_writedevicestr(stag,vstr);
    if(nret != 0){
        logout("err",logflag() + "rdb_writedevicestr(" + stag + ") error " + nret.to_string());
    }
    else{
        logout("dbg",logflag() + "rdb_writedevicestr(" + stag + ") success.");
    }
}
function run()
{
    teststr();
    getsnapstr("d0.str00.pv");
    writesnap("d0.str00.pv","test write snap!");
    writedevice("d0.str00.pv","test write device!");
}

```

4.4.10 读取实时库字符串标签快照

`rdb_getsnapstr (tagname)`

参数:

tagname 标签名，utf-8 字符串，不建议中文

返回值:

返回一个 `t_str` 类型，当 `qa` 不为 0 时表示有错误，见 4.3.1 中的 `qa` 定义。

例子 1:

```
var val = rdb_getsnapstr("m1.int1"); //读取标签值
```

例子 2:

```
var stag = "m1.int1";
var val = rdb_getsnapstr(stag); //读取标签值
```

参见 4.4.9 中的例子 `function getsnapstr(string stag) //读取字符串标签快照`

4.4.11 写实时库字符串标签快照

写标签快照到实时库

`rdb_writesnapstr(tagname,time,qa,strvalue)`

参数:

`tagname`: 标签名, utf-8 字符串, 不建议中文标签名。

`time`: rdb 格式时标, 如果此参数为 0 或 (0 的长整数立即数表示), 则使用当前时间

`qa`: 数据质量,

0 表示正常;

1 关机, 数据不可靠;

2 错误数据或数据无效;

3 无此标签;

4 时标错误

`strvalue`: 字符串, 小于 1000 字节

返回值:

返回 0 表示成功, 其他为错误码(错误码和 C/C++ 接口, java 接口等定义的相同)。

例子:

参见 4.4.9 中的例子 `function writesnap(string stag,string vstr)`

4.4.12 写设备控制输出字符串

控制输出, 写值到设备。

`rdb_writedevicestr(tagname,vstr);`

控制输出到设备

参数:

`tagname`: 标签名, utf-8 字符串, 不建议中文。

`vstr`: 字符串, 小于 1000 字节

返回值: 返回 0 表示成功, 其他为错误码(错误码和 C/C++ 接口, java 接口等定义的相同)。

例子:

参见 4.4.9 中的例子 `function writedevice(string stag,string vstr)`

4.4.13 文件操作对象 cFile

文件对象 cFile 是一个流文件对象，是 C/C++ 的 fopen 系列函数的简单包装。

创建一个流文件对象：

```
var myfile = cFile();
```

方法列表：

序号	方法	定义和说明
1	open	<p>打开或者创建文件。</p> <pre>bool open(string filename,string mode);</pre> <p>参数：</p> <p>filename: 文件名</p> <p>mode : 打开方式，和 C 函数的 fopen 一样。</p> <p>"r" read: Open file for input operations. The file must exist.</p> <p>"w" write: Create an empty file for output operations. If a file with the same name already exists, its contents are discarded and the file is treated as a new empty file.</p> <p>"a" append: Open file for output at the end of a file. Output operations always write data at the end of the file, expanding it. Repositioning operations (fseek, fsetpos, rewind) are ignored. The file is created if it does not exist.</p> <p>"r+" read/update: Open a file for update (both for input and output). The file must exist.</p> <p>"w+" write/update: Create an empty file and open it for update (both for input and output). If a file with the same name already exists its contents are discarded and the file is treated as a new empty file.</p> <p>"a+" append/update: Open a file for update (both for input and output) with all output operations writing data at the end of the file. Repositioning operations (fseek, fsetpos, rewind) affects the next input operations, but output operations move the position back to the end of file. The file is created if it does not exist.</p> <p>具体参见： http://www.cplusplus.com/reference/cstdio/fopen/</p> <p>返回值： true 成功； false 失败</p>
2	close	关闭文件

		<p><code>int close();</code></p> <p>返回值： 0：成功； -1：失败或者已经关闭。</p>
3	seek	<p>移动文件读写位置。 <code>int seek(long offset, int origin);</code> 参数： offset: 偏移值 origin: 起始位置</p> <p>SEEK_CUR 1 文件指针的当前位置。 SEEK_END 2 文件结尾。 SEEK_SET 0 文件开头。</p> <p>返回值： 0:成功 -1: 失败</p> <p>注：和 C++功能一样，参见 注：参见 http://www.cplusplus.com/reference/cstdio/fseek/</p>
4	tell	<p>返回文件当前读写位置 <code>long tell();</code></p> <p>返回值： >=0 :当前位置 -1:错误</p>
4	read	<p>从当前位置读取指定字节数 <code>string read(size_t size);</code></p> <p>参数： size：需要读取的字节数</p> <p>返回值： 返回 <code>string</code> 对象，读取的内容可能小于指定的长度。</p>
5	write	<p>从当前位置写入文件 <code>size_t write(string str);</code></p> <p>参数： str 需要写入的字符串对象</p> <p>返回值： 返回写入的字节数</p>

6	getc	<p>从当前位置读取一个字符</p> <p>int getc();</p> <p>返回读取到的字符，-1 表示读到文件尾部。</p>
7	putc	<p>从当前位置写入一个字符</p> <p>int putc(int ch);</p> <p>参数： ch:写入的字符</p> <p>返回值： 成功则返回写入的字符，失败返回-1;</p>
8	flush	<p>将缓冲内容属性到磁盘；</p> <p>int flush();</p> <p>返回值： 0:成功 -1:失败</p> <p>注： 和 C++功能一样，参见 http://www.cplusplus.com/reference/cstdio/fflush/</p>

具体例子参见 test_file.js

```
/*!
```

```
\file test_file.js
```

```
测试文件读写
```

```
*/
```

```
global g_jobname = "test_file";
```

```
global SEEK_CUR = 1;
```

```
global SEEK_END = 2;
```

```
global SEEK_SET = 0;
```

```
function logflag() //日志标志，用于在日志中区分是哪个脚本产生的日志
```

```
{
    return "[" + g_jobname + " ";
}
```

```
function test_write() //测试创建文件并写入文件
```

```

{
    var fl = cFile();
    var filename = "./tst1.txt";
    if (!fl.open(filename, "w+")) {
        logout("dbg", logflag() + "open failed!");
        return;
    }
    if (0 == fl.write("this is write string!")) {
        logout("dbg", logflag() + "write failed!");
        return;
    }
    fl.close();
}

```

function test_read() //测试读一块内容

```

{
    var fl = cFile();
    var filename = "./tst1.txt";
    if (!fl.open(filename, "r")) {
        logout("dbg", logflag() + "open failed!");
        return;
    }
    var txt = fl.read(4096);
    logout("dbg", logflag() + "read:" + txt.to_string());
    fl.close();
}

```

function test_read2() //把文件所有内容读到 string 中

```

{
    var fl = cFile();
    var filename = "./tst1.txt";
    if (!fl.open(filename, "r")) {
        logout("dbg", logflag() + "open failed!");
        return;
    }
    var txt = "";
    var c = fl.getc();
    while (c != -1) {
        txt.push_back(c);
        c = fl.getc();
    }
    logout("dbg", logflag() + "read:" + txt);
}

```

```

    fl.close();
}

function test_rewrite() //测试 seek, 改写内容
{
    var fl = cFile();
    var filename = "./tst1.txt";
    if (!fl.open(filename, "r+")) {
        logout("dbg", logflag() + "open failed!");
        return;
    }
    if (fl.seek(4, SEEK_SET) < 0) { //移动到开始到 4 的位置
        logout("dbg", logflag() + "seek failed");
        return;
    }
    if (0 == fl.write(" rewrite ")) {
        logout("dbg", logflag() + "write failed!");
        return;
    }
    if (fl.seek(0, SEEK_SET) < 0) { //移动到开始
        logout("dbg", logflag() + "seek failed");
        return;
    }
    var txt = "";
    var c = fl.getc();
    while (c != -1) {
        txt.push_back(c);
        c = fl.getc();
    }
    logout("dbg", logflag() + "read:" + txt);
    fl.close();
}

```

```

function test_filsize() //测试 teel 和 seek
{
    var fl = cFile();
    var filename = "./tst1.txt";
    if (!fl.open(filename, "r")) {
        logout("dbg", logflag() + "open failed!");
        return;
    }
    if (fl.seek(0, SEEK_END) < 0) { //移动到文件尾

```



```

        logout("dbg", logflag() + "seek to end failed");
        return;
    }
    var lsize = fl.tell();
    if (lsize < 0) {
        logout("dbg", logflag() + "tell failed!");
        return;
    }
    logout("dbg", logflag() + "file size: " + lsize.to_string());
    fl.close();
}

function run() {
    test_write();
    test_read();
    test_read2();
    test_rewrite();
    test_filsize();
}

```

4.4.14 人工置数

人工置数有两个函数。

1) 设置人工置数

```
rdb_manual_set(tagname,v);
```

参数:

tagname : 标签名, utf-8 字符串, 不建议中文。

v : 可以是 int32_t,uint32_t,int64_t,uint64_t,float,double 之一

返回值: 返回 0 表示成功, 其他为错误码(错误码和 C/C++接口, java 接口等定义的相同)。

2) 取消人工置数

```
rdb_manual_del(tagname);
```

参数:

tagname : 标签名, utf-8 字符串, 不建议中文。

返回值: 返回 0 表示成功, 其他为错误码(错误码和 C/C++接口, java 接口等定义的相同)。

4.4.15 统计值结果对象 r_count

rdb_countvalue 函数返回的对象, 用于存储统计执行结果。全部为只读属性

属性名	数据类型	说明
-----	------	----

retcode	Int32	执行结果，0 表示成功; 其他为错误码
avgval	Double	平均值
sumval	Double	算数累加值
numrecs	Int	参与统计的样本值记录数
minvalrec	t_val	最小值记录对象，参见 4.4.2
maxvalrec	t_val	最大值记录对象，参见 4.4.2

4.4.16 值统计

`rdb_countvalue(tagname, exp, lts, lte)`

tagname: 字符串，标签名

exp: 字符串，查询表达式，可以为空

lts: 开始时标，长整数，长整型，从 1970-1-1 开始的 100 毫秒,GMT 时间。

lte: 结束时标，长整数，长整型，从 1970-1-1 开始的 100 毫秒,GMT 时间。

返回值: `r_count` 对象;

说明: 这个函数对应 api 中的 `rdb_countvalue` 接口函数，为适应脚本系统，参数有不同。

4.4.17 状态改变结果对象 `r_statuschange`

用于接收状态改变统计查询函数 `rdb_countstatuschang` 的结果。全部为只读属性。

属性名	数据类型	说明
retcode	Int32	执行结果，0 表示成功; 其他为错误码
numl2h	Int	从低值变为高值的次数
numh2l	Int	从高值变为低值的次数

4.4.18 状态改变统计

`rdb_countstatuschang` 用于统计高低值变化次数，一般用于统计开关变位次数，对应 `rdbapi` 同名函数。

`rdb_countstatuschang(tagname, lts, lte, nlval, nhval)`

tagname: 字符串，标签名

lts: 开始时标，长整数，从 1970-1-1 开始的 100 毫秒,GMT 时间。

lte: 结束时标, 长整数, 从 1970-1-1 开始的 100 毫秒,GMT 时间。

nlval: 整数, 低值;

nhval: 整数, 高值;

返回值: r_statuschange 对象;

4.4.19 值运行时间结果对象 r_valtime

用于存储统计值运行时间函数 rdb_countvaltime 返回的结果数据对象。

属性名	数据类型	说明
retcode	Int32	执行结果, 0 表示成功; 其他为错误码
ltime	Int64	运行时间, 从 1970-1-1 开始的 100 毫秒, GMT 时间。
numrecs	Int32	参与统计的样本值记录数。

4.4.20 值运行时间统计

rdb_countvaltime 对应 rdbapi 里的同名函数, 一般用于统计开关维持一个状态的时间, 或者某个标签值符合条件表达式的时间。

rdb_countvaltime(tagname, exp, lts, lte)

tagname: 字符串, 标签名

exp: 字符串, 条件表达式, 不能为空

lts: 开始时标, 长整数, 从 1970-1-1 开始的 100 毫秒,GMT 时间。

lte: 结束时标, 长整数, 从 1970-1-1 开始的 100 毫秒,GMT 时间。

返回值: r_valtime 对象。

4.4.21 数值转换函数

三个将字符串转换为数值函数: atoi, atof, atoll 分别返回整数, 浮点数和长整型。参数为字符串。

例子:

```
var vi = atoi("12");
```

```
var vl = atoll("1234567890");
```

```
var vf = atof("12.5");
```

4.4.22 增量统计函数

rdb_countincrement 是 rdb2025.4(inner version 5123)版新增, 统计一个时间段内累积量的增量。

```
rdb_countincrement(tagname, time_start, time_end);
```

三个参数均为字符串,utf8 编码,时间为 ISO 格式。返回值为 4.4.2 值类型对象。如 rdbctrl/js/test_count.js 中例子:

```
var vinc = rdb_countincrement("d0.f01.pv", "2025-03-01T01:0:0.000+08:00", "2025-03-01T02:0:0.000+08:00");
```

4.4.23 积分统计函数

rdb_countintegral 是 rdb2025.4(inner version 5123)版新增,统计一个时间段内瞬时量产生的积分量累积量。四个参数,前三个为字符串,最后一个为整数。

```
rdb_countintegral(tagname, time_start, time_end,timeunit);
```

标签名为 utf8 编码,时间为 ISO 格式, timeunit 为积分用瞬时量时间单位秒数; 升/分钟就填 60; 立方/小时就填 3600

返回值为 4.4.2 值类型对象。

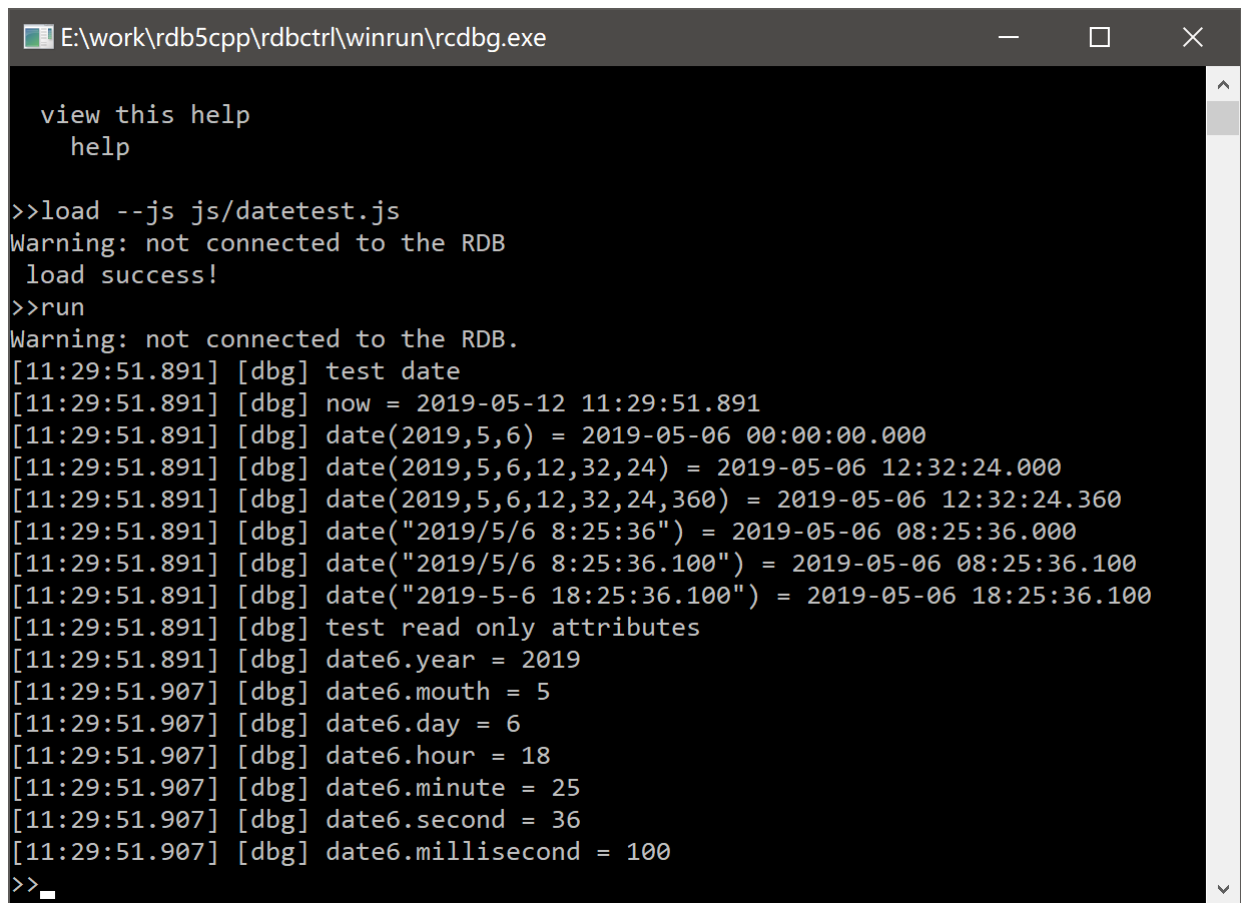
如 rdbctrl/js/test_count.js 中例子:

```
var vintegral = rdb_countintegral("d0.f01.pv", "2025-03-01T01:0:0.000+08:00", "2025-03-01T02:0:0.000+08:00", 60);
```

5 脚本测试工具

提供了一个 windows 命令行版的脚本测试工具 rcdbg.exe，脚本上线运行前使用该工具测试。

运行 rcdbg，查看屏幕帮助提示，随时可键入 help 获得帮助信息。



```
E:\work\rdb5cpp\rdbctrl\winrun\rcdbg.exe

view this help
help

>>load --js js/datetest.js
Warning: not connected to the RDB
load success!
>>run
Warning: not connected to the RDB.
[11:29:51.891] [dbg] test date
[11:29:51.891] [dbg] now = 2019-05-12 11:29:51.891
[11:29:51.891] [dbg] date(2019,5,6) = 2019-05-06 00:00:00.000
[11:29:51.891] [dbg] date(2019,5,6,12,32,24) = 2019-05-06 12:32:24.000
[11:29:51.891] [dbg] date(2019,5,6,12,32,24,360) = 2019-05-06 12:32:24.360
[11:29:51.891] [dbg] date("2019/5/6 8:25:36") = 2019-05-06 08:25:36.000
[11:29:51.891] [dbg] date("2019/5/6 8:25:36.100") = 2019-05-06 08:25:36.100
[11:29:51.891] [dbg] date("2019-5-6 18:25:36.100") = 2019-05-06 18:25:36.100
[11:29:51.891] [dbg] test read only attributes
[11:29:51.891] [dbg] date6.year = 2019
[11:29:51.907] [dbg] date6.mouth = 5
[11:29:51.907] [dbg] date6.day = 6
[11:29:51.907] [dbg] date6.hour = 18
[11:29:51.907] [dbg] date6.minute = 25
[11:29:51.907] [dbg] date6.second = 36
[11:29:51.907] [dbg] date6.millisecond = 100
>>
```

图 5 脚本测试工具

脚本测试流程：

- (1) rdb 命令连接实时库，不使用实时库的可以不连接，load 和 run 时会有警告提示。连接实时库的例子：

```
rdb --ip 192.168.1.230 --port 19089 --usr admin --psw admin
```

- (2) 使用 load 命令加载脚本，通过--js 开关指定脚本文件名，相对于 rddb.exe 所在目录的路径。

加载例子：

```
load --js js/datetest.js
```

表示加载 js 子目录下的 datetest.js 脚本文件。

load 时会对脚本进行编译，如果有错会提示错误信息，编译成功后会执行初始化部分的代码。

重新执行 load 命令会清除当前已加载的脚本重新加载和初始化。

(3) 使用 `run` 命令运行脚本

在 `load` 成功后才能 `run`，直接键入 `run` 运行，无参数。

`run` 相当于调用脚本的 `function run()` 函数，可以加载后多次运行。

在 `rcdbg` 测试环境运行，`logout` 输出的全部级别日志被直接输出到屏幕上。