RCM使用指南

1. 前言

产品简介及本手册的内容概况。

1.1. 产品简介

RCM（可靠一致组播，Reliable and Consistent Multicast）是恒生公司专门为高性能应用设计的消息传输总线，提供基于主题语义的低延时、高吞吐、大容量消息传输服务，拥有分布式、高可靠、易扩展等特点。RCM以dll（Windows下）或so（Linux下）的形式提供给开发者。

1.2. 读者对象

本指南主要适用于以下人员：

- 开发人员，仅限于C++开发

1.3. 手册概况

本手册各章节内容如下表所示

章节	内容
前言	产品简介及本手册的内容概况
开发包简介	开发包的相关信息
RCM介绍	RCM的概述和特性
安装与卸载	RCM的安装和卸载
快速开始	快速使用RCM
快速启用高可用场景	高可用场景下快速使用RCM
RCM C++接口开发指南	C++开发指南
RCM的高可用	如何使用RCM高可用
开发接口	具体的开发接口的函数说明
附录	介绍所有的配置项
Q&A	

1.4. 缩略语/术语

下面列出了本手册中出现的缩略语和术语

LDP	低延时分布式开发平台	恒生新一代的低延时分布式开发平台
RCM	可靠一致组播	恒生消息传输总线
Topic	主题	一类业务消息的总称

Partition	分区	恒生新一代的低延时分布式开发平台
Context	上下文	是RCM管理发送端、接收端、内存和线程等资源的基础单元

2. 开发包简介

```
名称：rcm.dll/librcm.so

开发语言：C++

支持操作系统：windows/linux
```

版本信息获取：

- WINDOWS：右键→属性→详细信息，里面有版本信息，格式是"1.0.0.*"
- LINUX：通过命令strings librcm.so | grep "rcm version"，如下所示：

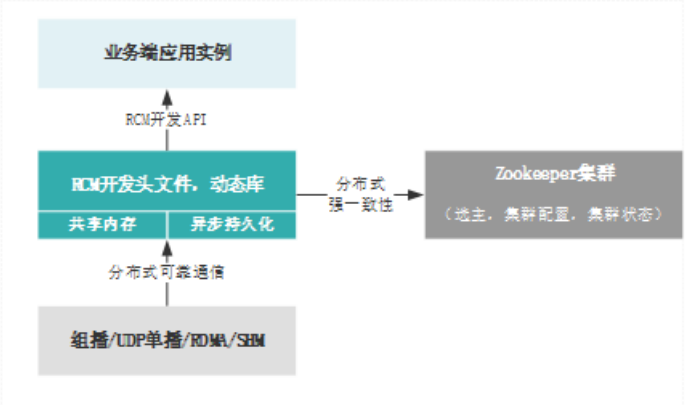
```
[huxb@docker-vm-1-23 bin]$ strings librcm.so | grep "rcm version"
rcm version V1.0.0.0 May 11 2020 14:27:01
```

获取方式：

- 恒生内部的员工，以项目组的形式向研发中心客服申请使用。
- 恒生外部开发商，向对应项目的恒生接口人获取开发包。

3. RCM介绍

RCM英文全称为Reliable and Consistent Multicast，即可靠一致组播，在通用组播的基础上，增加了消息传输可靠性和分布式一致性，通过简单易用的API，提供基于主题语义的低延迟、高可靠、分布式应用开发能力，开发模型如图所示：



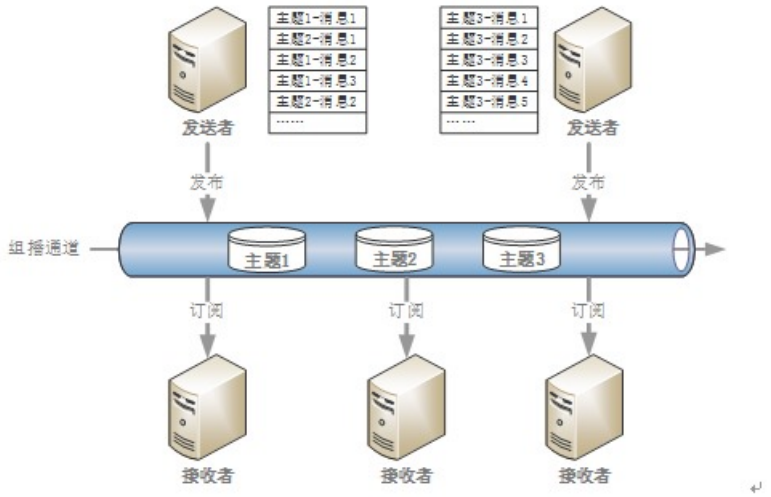
3.1. 可靠组播

RCM总线实现了可靠组播，保证消息不重、不乱、不丢。

3.2. 主题语义

RCM总线提供基于主题的消息传输机制，使得发送者和接收者互相解耦，方便用户理解和使用。分区也可以用于系

统扩展，如图所示：



3.3. 低延时

RCM发送和接收消息使用零拷贝、减少线程切换等优化措施，端对端的延时在2.5us左右，同时支持共享内存通信方式，延时在400ns左右。

3.4. 高性能

RCM采用延迟ACK、粘包等机制，提升通信效率，使得吞吐量达千万级别。

3.5. 支持大报文

RCM通过对大消息进行分片，最大支持1G的报文。

3.6. 分布式强一致性

RCM总线通过zookeeper选主，并通过主节点进行整个集群的消息复制和状态同步，保证整个集群的强一致性。

3.7. 数据零丢失

RCM支持持久化，一是发送端的数据持久化，采用共享内存方式，用户调用发送接口，会直接将数据写入共享内存，从而保证写数据的可靠。对于接收端，会异步的将数据写入磁盘。

3.8. 数据恢复

发送端数据恢复，主要通过共享内存恢复数据，发送端启动，会从上次发送成功的消息序号开始，从共享内存读取数据并发送后续数据。接收端数据恢复，节点启动时首先从本地磁盘读取持久化的数据，然后向集群主节点发送同步请求，恢复到和主节点状态一致后，正常处理业务。 对于整个集群重启恢复，集群启动前，需要用户在zookeeper中确定当前集群的主节点，首先将这个主节点启动，然后其他节点后续启动后，会恢复到与这个节点一致的状态。

如果zookeeper集群数据被清空，则需要用户对比每个节点的数据，哪个节点数据越多，就先启动。

---

4. 安装与卸载

本节主要介绍如何对RCM环境进行部署和卸载。

4.1. 发布包的目录结构

RCM以动态链接库的方式提供给用户使用，并提供相应的头文件、使用文档说明和代码示例，为方便用户可以更加便捷地使用和维护RCM，RCM发布包的目录结构如表所示：

目录名称	说明
bin	存放可执行程序、配置文件和脚本文件
include	RCM相关头文件
lib	编译后的RCM动态库
simple	样例程序，包括编译脚本，完成了简单的发送接收功能：客户端发送数据，服务端收到数据以后，直接打印数据内容
echo	样例程序，包括编译脚本，客户端发送数据给服务端，服务端收到以后，首先打印数据内容，然后将数据原样返回给客户端

4.2. 运行环境

4.2.1. 硬件要求

环境要素	说明
服务器	高性能PC服务器
CPU	1个或多个CPU，双核以上，如果需要高并发，建议多个CPU和核，同时高频CPU能很好的提升性能
内存	RCM会将用户数据定时异步刷入磁盘，但是会在内存保存排序数据，1亿条数据需要14GB内存，可以根据这个标准，推算业务需要的内存
网卡	建议使用万兆低时延网卡，同时也支持千兆网卡
交换机	1Gb及以上且支持组播的以太网交换机，支持IGMP V2及以上

4.2.2. 软件要求

环境要素	说明
操作系统	Linux 64位操作系统，目前支持的操作系统有： 1. Red Hat Enterprise Linux 7 2. Centos7以上 3. Windows 64位操作系统
GCC	GCC-4.8.5 on RHEL7， Centos 7
Python	Python 2.7.5 on RHEL7，Centos
Java	Java 8及以上
C#	C# 2017及以上
C/C++	C++ 11及以上

4.3. RCM安装

将安装包中的include，lib拷贝到工程项目中，参照快速开始的样例即可。

4.3.1. 设置动态库访问路径

```
1. vi ~/.bash_profile
2. 输入下面命令
3. export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:[lib文件夹的绝对路径]
4. source ~/.bash_profile
```

4.3.2. zookeeper部署（高可用场景）

RCM支持集群高可用模式，这种模式下需要部署zookeeper，请参考zookeeper部署方案。

4.4. RCM卸载

删除相关的包，删除本地的日志文件和持久化数据文件，删除/dev/shm/下相关的共享内存文件。

4.5. RCM版本更新

直接覆盖相关文件路径即可。

---

5. 快速开始

样例代码使用最简单和最基本的功能，快速展示如何使用RCM库。

5.1. 简单发送者和接收者

```
/*
   *可靠一致组播示例程序
   *单发送端、接收端
*/

#include <rcm.h>
#include <thread>           // for sleep_for
#include <locale.h>         // for setlocale
#include <string.h>
#include <string>

using namespace std;
using namespace rcm;

#ifndef _WIN32
#define PRIu64 "lu"
#else
#define PRIu64 "I64u"
#endif

class CLogger : IRcmLogger {
public:
    virtual void Log(int32_t nKey, int32_t nLevel, const wchar_t *lpContent)
    {
        printf("key:%d Level:%d Content:%ls\n", nKey, nLevel, lpContent);
    }
};

class CRcmAppClient
{
public:
    CRcmAppClient(IRcmFactory * lpRcmFactory, const char * szContext, const char * szTopic)
    {
        m_lpFactory = lpRcmFactory;
        m_lpContext = NULL;
        m_lpTransmitter = NULL;
```

```
        m_strContext = szContext;
        m_strTopic = szTopic;
    }

    int32_t Init()
    {
        m_lpContext = m_lpFactory->NewRcmContext(m_strContext.c_str());
        if (m_lpContext == NULL)
        {
            printf("NewRcmContext failed: %s\n", m_strContext.c_str());
            return -1;
        }

        m_lpTransmitter = m_lpContext->CreateTransmitter(m_strTopic.c_str());
        if (m_lpTransmitter == NULL)
        {
            printf("CreateTransmitter failed: %s\n", m_strTopic.c_str());
            return -1;
        }

        int32_t nErrorNo = m_lpContext->Start();
        if (nErrorNo)
        {
            printf("context start failed, errorno: %d\n", nErrorNo);
            return -1;
        }
        return 0;
    }

    int32_t SendReq()
    {
        const char * szData = "hello world";
        uint32_t ulength = (uint32_t)strlen(szData) + 1; //包含后面的\0
        int32_t nRet = m_lpTransmitter->SendData(szData, ulength);
        if (nRet)
        {
            printf("Send: %d\n", nRet);
        }
        return nRet;
    }
private:
    IRcmFactory *       m_lpFactory;
    IRcmContext *       m_lpContext;
    IRcmTransmitter *   m_lpTransmitter;
    string              m_strContext;
    string              m_strTopic;
};

class CRcmAppServer : public IRcmReceiverCallback
{
public:
    CRcmAppServer(IRcmFactory * lpRcmFactory, const char * szContext)
    {
        m_lpFactory = lpRcmFactory;
        m_lpContext = NULL;
        m_strContext = szContext;
    }

    int32_t Init()
    {
        m_lpContext = m_lpFactory->NewRcmContext(m_strContext.c_str());
        if (m_lpContext == NULL)
        {
            printf("NewRcmContext failed: %s\n", m_strContext.c_str());
            return -1;
        }

        IRcmReceiver *lpIRcmReceiver = m_lpContext->CreateReceiver(this);
        if (lpIRcmReceiver == NULL)
        {
            printf("CreateReceiver failed\n");
            return -1;
        }
```

```
        }

        int32_t nErrorNo = m_lpContext->Start();
        if (nErrorNo)
        {
            printf("context start failed, errorno: %d\n", nErrorNo);
            return -1;
        }
        return 0;
    }

    int32_t OnMessage(void *lpData, uint32_t nLength, uint64_t nMsgSqn, uint16_t nThreadNo, IRcmMsgInfo
*lpRcmMsg)
    {
        printf("OnMessage TxContext:%s Topic:%s PartitionNo:%u MsgNo:%" PRIu64 " Length:%u MsgSqn: %"
PRIu64 "\n"
            , lpRcmMsg->GetTxContextName()
            , lpRcmMsg->GetTopicName()
            , lpRcmMsg->GetPartitionNo()
            , lpRcmMsg->GetTxMsgNo()
            , nLength
            , nMsgSqn);

        return 0;
    }

private:
    IRcmFactory *       m_lpFactory;
    IRcmContext *       m_lpContext;
    string              m_strContext;
};

int main(int argc, char * argv[])
{
    if (argc < 5)
    {
        printf("Usage: simpledemo role isprint config context [topic]\n");
        exit(1);
    }

    //设置地域信息，主要作用于wchar的中文屏幕输出
    setlocale(LC_ALL, "");

    int role = atoi(argv[1]);
    int iPrintStastics = atoi(argv[2]);
    const char * szConfig = argv[3];
    const char * szContext = argv[4];
    const char * szTopic = "req";
    if (argc > 5)
    {
        szTopic = argv[5];
    }

    const char * szZkAddr = NULL;
    if(argc > 6)
    {
        szZkAddr = argv[6];
    }
    const char* szZkPath = "/rcm";
    if(argc > 7)
    {
        szZkPath = argv[7];
    }

    CRcmAppClient * lpAppClient = NULL;
    CRcmAppServer * lpAppServer = NULL;

    IRcmFactory * lpRcmFactory = NewRcmFactory(szConfig);
    if (lpRcmFactory == NULL)
    {
        printf("NewRcmFactory failed\n");
        exit(1);
    }
```

```
    CLogger logger;
    lpRcmFactory->SetLogArg(RCM_LOG_INFO, (IRcmLogger *)&logger);
    if (role == 0)          //创建发送端
    {
        lpAppClient = new CRcmAppClient(lpRcmFactory, szContext, szTopic);
        if (lpAppClient->Init())
        {
            printf("client init failed\n");
            exit(1);
        }

        lpAppClient->SendReq();

        for (size_t i = 0; i < 10; i++)
        {
            lpAppClient->SendReq();
            std::this_thread::sleep_for(std::chrono::milliseconds(100));
        }
    }
    else                    //创建接收端
    {
        if(szZkAddr != NULL)
        {
            lpRcmFactory->SetArbArg(szZkAddr, szZkPath);
        }
        lpAppServer = new CRcmAppServer(lpRcmFactory, szContext);
        if (lpAppServer->Init())
        {
            printf("server init failed\n");
            exit(1);
        }
    }

    //打印统计信息，主要用于查看内部状态，非必须
    while (true) {
        if (iPrintStastics)
        {
            char * lpResult = NULL;
            lpRcmFactory->GetStatistics("{}", &lpResult);
            if (lpResult != NULL)
            {
                puts(lpResult);
            }
        }

        std::this_thread::sleep_for(std::chrono::seconds(5));
    }

    DeleteRcmFactory(lpRcmFactory);

    return 0;
}
```

5.2. 简单发送者配置

```
{
    "Rcm": {
        "Transports": [
            {
                "Name": "Default",
                "PartitionNo": 1
            },
            {
                "Id": 1,
                "Topic": "req",
                "Addr": "235.0.0.1",
                "Port": 30000
            }
        ],
        "Contexts": [
            {
                "Name": "Default",
                "Ip": "192.168.86.102",
                "IsSingleton": true,
                "RepairPortStart": 30001,
                "RepairPortEnd": 40000
            },
            {
                "REF": "Default",
                "Name": "client1",
                "Id": 1,
                "TxTopics": [
                    {
                        "Name": "req"
                    }
                ]
            },
            {
                "REF": "Default",
                "Name": "server1",
                "Id": 2,
                "RxTopics": [
                    {
                        "Name": "req"
                    }
                ]
            },
            {
                "REF": "Default",
                "Name": "server2",
                "Id": 3,
                "RxTopics": [
                    {
                        "Name": "req"
                    }
                ]
            }
        ]
    }
}
```

配置中注意修改Ip地址为运行机器的真实地址。

5.3. 简单接收者配置

接收者配置和发送者配置一致，二者通过各自的上下文名创建上下文，一个创建发送者并指定发送主题，一个创建出接收者，并指定和发送者一致的主题。

5.4. 编译运行

通过编写上面发送者和接收者的代码后，文件树如下：

```
|-- bin
|-- echo
|-- include
|   `-- rcm.h
|-- lib
|   `-- linux.x64
|       `-- librcm.so
|-- simple
|   `-- Makefile
|   `-- simple.cpp
```

进入rcm/simple/文件夹，运行命令

```
make clean && make
```

打开一个终端，进入rcm/bin/文件夹，运行命令,开启一个接收端：

```
sh startserver.sh
```

打开一个终端，进入rcm/bin/文件夹，运行命令,开启一个发送端：

```
sh startclient.sh
```

接收端会输出如下内容：

```
key:21060 Level:1 Content:接收端 req.1 序号协商成功，会话号 26904 消息序号 0->0(rcm_rms_receiver.cpp:483,HandleSessionMsg)
OnMessage TxContext:client1 Topic:req PartitionNo:1 MsgNo:0 Length:12 MsgSqn: 0
OnMessage TxContext:client1 Topic:req PartitionNo:1 MsgNo:1 Length:12 MsgSqn: 1
OnMessage TxContext:client1 Topic:req PartitionNo:1 MsgNo:2 Length:12 MsgSqn: 2
OnMessage TxContext:client1 Topic:req PartitionNo:1 MsgNo:3 Length:12 MsgSqn: 3
OnMessage TxContext:client1 Topic:req PartitionNo:1 MsgNo:4 Length:12 MsgSqn: 4
OnMessage TxContext:client1 Topic:req PartitionNo:1 MsgNo:5 Length:12 MsgSqn: 5
OnMessage TxContext:client1 Topic:req PartitionNo:1 MsgNo:6 Length:12 MsgSqn: 6
OnMessage TxContext:client1 Topic:req PartitionNo:1 MsgNo:7 Length:12 MsgSqn: 7
OnMessage TxContext:client1 Topic:req PartitionNo:1 MsgNo:8 Length:12 MsgSqn: 8
OnMessage TxContext:client1 Topic:req PartitionNo:1 MsgNo:9 Length:12 MsgSqn: 9
OnMessage TxContext:client1 Topic:req PartitionNo:1 MsgNo:10 Length:12 MsgSqn: 10
```

同时我们可以观察到，在发送端当前目录下，会生成log日志文件，在发送端的/dev/shm/目录下会生成一些共享内存文件，client1_allocator文件,其中存储着传输的消息，client1_req_1_sendingwindow文件存储滑动窗口对象，client1_req_1_sendingwindowblock_1、client1_req_1_sendingwindowblock_2文件存储窗口节点信息。而在接收端当前目录，同样会生成log日志文件，若开启持久化，默认在当前目录下，会将接收到的数据持久化到磁盘。

---

6. 快速启用高可用场景

本节展示高可用场景下如何快速使用RCM。

6.1. 启用zookeeper集群

RCM高可用方案，采用zookeeper集群的强一致性选主功能，请参考zookeeper相关部署命令，搭建好zookeeper集群，并得到zookeeper集群的连接地址和端口，比如：

```
192.168.86.102:2181
```

6.2. 启用高可用集群

RCM的高可用集群，通过zookeeper进行集群角色管理，包括选择主节点，主节点为每条消息分配一个连续递增的全

局序号，接收集群所有节点的数据都和主节点保持一致，当发生节点宕机，会重新选择主节点，以及集群间进行数据同步，保证强一致性。

6.2.1. 修改配置

在rcm.json配置文件中，在上下文”Contexts”中增加如下配置：

```
            {
                "REF": "Default",
                "Name": "Tier",
                "IsSingleton": false,
                "SyncIp": "192.168.86.102",
                "SyncAddr": "239.4.1.1",
                "SyncRepairPortStart": 35000,
                "SyncRepairPortEnd": 40000,
                "UseSameMachine": false,
                "IsTotalOrder": true
            },
            {
                "REF": "Tier",
                "Name": "HaNode1",
                "Id": 4,
                "TierName": "HaTest",
                "SyncPort": 30000,
                "RxTopics": [
                    {
                        "Name": "req"
                    }
                ]
            },
            {
                "REF": "Tier",
                "Name": "HaNode2",
                "Id": 5,
                "TierName": "HaTest",
                "SyncPort": 30000,
                "RxTopics": [
                    {
                        "Name": "req"
                    }
                ]
            }
```

配置中注意修改Ip地址以及SyncIp地址为运行机器的真实地址。

6.2.2. 运行接收集群

打开第一个终端，运行如下命令，启动集群节点1

```
sh startnode1.sh
```

在另一台电脑上打开一个终端，运行如下命令，启动集群节点2

```
sh startnode2.sh
```

打开第三个终端，运行如下命令，启动发送端

```
sh startclient2.sh
```

第一个终端将输出如下内容：

```
key:21060 Level:1 Content:接收端 req.1 序号协商成功，会话号 14333 消息序号 0->0(rcm_rms_receiver.cpp:483,HandleSessionMsg)
OnMessage TxContext:client1 Topic:req PartitionNo:1 MsgNo:0 Length:12 MsgSqn: 0
OnMessage TxContext:client1 Topic:req PartitionNo:1 MsgNo:1 Length:12 MsgSqn: 1
OnMessage TxContext:client1 Topic:req PartitionNo:1 MsgNo:2 Length:12 MsgSqn: 2
OnMessage TxContext:client1 Topic:req PartitionNo:1 MsgNo:3 Length:12 MsgSqn: 3
OnMessage TxContext:client1 Topic:req PartitionNo:1 MsgNo:4 Length:12 MsgSqn: 4
OnMessage TxContext:client1 Topic:req PartitionNo:1 MsgNo:5 Length:12 MsgSqn: 5
OnMessage TxContext:client1 Topic:req PartitionNo:1 MsgNo:6 Length:12 MsgSqn: 6
OnMessage TxContext:client1 Topic:req PartitionNo:1 MsgNo:7 Length:12 MsgSqn: 7
OnMessage TxContext:client1 Topic:req PartitionNo:1 MsgNo:8 Length:12 MsgSqn: 8
OnMessage TxContext:client1 Topic:req PartitionNo:1 MsgNo:9 Length:12 MsgSqn: 9
OnMessage TxContext:client1 Topic:req PartitionNo:1 MsgNo:10 Length:12 MsgSqn: 10
```

第二个终端将输出如下内容：


```
key:21060 Level:1 Content:接收端 req.1 序号协商成功，会话号 14333 消息序号 0->0(rcm_rms_receiver.cpp:483,HandleSessionMsg)
OnMessage TxContext:client1 Topic:req PartitionNo:1 MsgNo:0 Length:12 MsgSqn: 0
OnMessage TxContext:client1 Topic:req PartitionNo:1 MsgNo:1 Length:12 MsgSqn: 1
OnMessage TxContext:client1 Topic:req PartitionNo:1 MsgNo:2 Length:12 MsgSqn: 2
OnMessage TxContext:client1 Topic:req PartitionNo:1 MsgNo:3 Length:12 MsgSqn: 3
OnMessage TxContext:client1 Topic:req PartitionNo:1 MsgNo:4 Length:12 MsgSqn: 4
OnMessage TxContext:client1 Topic:req PartitionNo:1 MsgNo:5 Length:12 MsgSqn: 5
OnMessage TxContext:client1 Topic:req PartitionNo:1 MsgNo:6 Length:12 MsgSqn: 6
OnMessage TxContext:client1 Topic:req PartitionNo:1 MsgNo:7 Length:12 MsgSqn: 7
OnMessage TxContext:client1 Topic:req PartitionNo:1 MsgNo:8 Length:12 MsgSqn: 8
OnMessage TxContext:client1 Topic:req PartitionNo:1 MsgNo:9 Length:12 MsgSqn: 9
OnMessage TxContext:client1 Topic:req PartitionNo:1 MsgNo:10 Length:12 MsgSqn: 10
```


使用zookeeper工具，可以看到在路径/rcm/cluster_HaTest下面，存在下面三个子路径：

```
classer
    HaNode1
    HaNode2
master
running
```

classer 可以查看集群中有哪些节点，master表示当前的主节点信息，running标识主节点是否活跃。

---

7. RCM C++接口开发指南

RCM支持udp组播，udp单播，工作在TCP/IP之上的应用层，对外主要通过动态库，头文件提供服务，RCM提供应用层的消息发送、接收，高可用场景下分布式系统的强一致性消息队列管理能力，通过主题的发布和订阅，实现千万级别的高吞吐，以及5us以内的低延迟处理能力。应用程序通过RCM提供的API来调用和实现这些功能，下面主要针对C++接口进行阐述。

7.1. RCM工厂

RCM工厂最重要的是管理上下文的创建和销毁，同时管理日志，zookeeper仲裁，RCM事件等。

```
// 可靠一致组播工厂
class IRcmFactory
{
protected:
    virtual ~IRcmFactory() {}
public:
    /*
        * 创建可靠一致组播上下文
        * @param lpName 上下文名称
        * @return IRcmContext接口指针，NULL表示失败
    */
    virtual IRcmContext *NewRcmContext(const char *lpName) = 0;

    /*
        * 释放可靠一致组播上下文
        * @param lpIRcmContext IRcmContext接口指针
        * @return 0成功，其他失败
    */
    virtual int32_t DeleteRcmContext(IRcmContext *lpIRcmContext) = 0;

    /*
        * 获取rcm配置对象
        * @return 配置对象接口，NULL表示失败
    */
    virtual IRcmConfig *GetConfig() = 0;

    /*
        * 获取组播状态数据
        * @param lpRequest json格式请求字符串
        * @param lppResult json格式结果字符串
        * @return 无
    */
    virtual void GetStatistics(const char *lpRequest, char **lppResult) = 0;

    /*
        * 设置日志参数
        * @param nMinLogLevel 最小日志输出等级，小于该等级的日志不会回调，默认为INFO
        * @param lpRcmLogger 日志接口，如果为NULL，内部会写日志
        * @note 日志在工作目录下的log文件夹
        * @return 无
    */
    virtual void SetLogArg(int32_t nMinLogLevel, IRcmLogger *lpRcmLogger = NULL) = 0;

    /*
        * 设置仲裁参数
        * @param lpArbAddress Arb服务地址，目前Arb服务支持zookeeper
        * @param lpArbPath 提供给仲裁写数据的路径
        * @note 使用集群上下文时需要设置
        * @return 无
    */
    virtual void SetArbArg(const char *lpArbAddress, const char *lpArbPath) = 0;

    /*
        * 设置事件回调接口
        * @param lpRcmEventCallback 事件回调接口
        * @param lpUser 事件回调参数
        * @return 无
    */
    virtual void SetEventCallback(IRcmEventCallback *lpRcmEventCallback, void *lpUser = NULL) = 0;

    /*
        * 设置文件前缀
        * @param szFilePrefix 文件前缀名
        * @note 共享内存等的文件夹
        * @return 无
    */
    virtual void SetFilePrefix(const char *szFilePrefix) = 0;
};
```

7.1.1. 创建RCM工厂

```
/*
     * 创建可靠一致组播工厂
     * @param szConfig 配置文件名，json格式
     * @param szWorkDir 工作目录，默认为当前目录
     * @param szLang 日志语言，支持"zh_CN"，默认为"zh_CN"
     * @return IRcmFactory接口指针，NULL表示失败
*/
IRcmFactory *NewRcmFactory(const char *szConfig, const char *szWorkDir = NULL, const char *szLang =
NULL);
```

7.1.2. 销毁RCM工厂

```
/*
     * 释放可靠一致组播工厂
     * @param lpIRcmFactory 工厂接口指针
     * @return 无
*/
void DeleteRcmFactory(rcm::IRcmFactory *lpIRcmFactory);
```

7.2. RCM上下文

RCM上下文是RCM的重要组件，RCM通过上下文关联发送端和接收端的消息流，每个上下文可以有多个发送主题和多个接收主题，发送主题可以有多个分区，在负载均衡模式下，发送端会均衡地将消息分发给每个分区，接收端可以指定一个或者多个分区进行消息接收。

RCM接收端可以在上下文中配置多个接收主题，在非高可用场景下，不做全局排序，并且支持启动多个线程，每个线程接收一个主题的数据，从而实现高并发。在高可用场景下，通过zookeeper仲裁选取出leader，并由leader对所有的主题消息进行排序，并为每个消息分配一个连续递增的消息序号，所有集群中的节点保证消息序号以及对应数据强一致。

7.2.1. 上下文的创建和销毁

RCM上下文的创建和销毁都通过RCM中的接口完成

创建RCM上下文：

```
IRcmContext* m_lpRcmContext = m_lpRcmFactory->NewRcmContext(szContextName);
```

销毁RCM上下文

```
m_lpRcmContext->Stop();
m_lpRcmFactory->DeleteRcmContext(m_lpRcmContext);
```

销毁RCM上下文之前，需要调用Stop方法，停止所有发送者和接收者的工作线程，并等待线程退出，然后才通过工厂销毁上下文。

7.2.2. 发送者的启动和停止

RCM发送者通过上下文进行创建：

```
IRcmTransmitter* lpRcmTransmitter = m_lpRcmContext->CreateTransmitter(
                                    szTopicName,
                                    NULL,
                                    0);     // rcm采用主题订阅

if (lpRcmTransmitter == NULL)
{
    printf("CreateTransmitter failed: %s\n", szTopicName);
    return -1;
}

int32_t nRet = m_lpRcmContext->Start();

if (nRet)
{
    printf("context start failed, nRet: %d\n", nRet);
    return -1;
}
```

发送者创建成功后，才能调用上下文的Start方法，启动发送者的工作线程，否则发送者工作线程无法正确启动，数据将无法发送。

RCM发送者也通过上下文进行停止：

```
lpRcmContext->Stop()
```

调用上下文的Stop方法，会停止这个上下文中所有发送者和接收者的工作线程。

7.2.3. 接收者的启动和停止

接收者通过RCM上下文创建：

```
IRcmReceiver *lpIRcmReceiver = m_lpRcmContext->CreateReceiver(
                                &m_rxCallback,
                                NULL);

if (lpIRcmReceiver == NULL)
{
        printf("CreateReceiver failed\n");
        return -1;
}

int startRet = m_lpRcmContext->Start();

if (startRet)
{
        printf("context start failed, startRet: %d\n", startRet);
        return -1;
}
```

接收者创建成功后，才能调用上下文的Start方法，否则接收者的工作线程将无法成功启动，也就无法正常接收数据。

RCM接收者也通过上下文进行停止：

```
lpRcmContext->Stop()
```

调用上下文的Stop方法，会停止这个上下文中所有发送者和接收者的工作线程。

7.2.4. 发送端不分区配置实例

```
{
    "Rcm": {
        "Transports": [
            {
                "Name": "Default",
                "PartitionNo": 1
            },
            {
                "Id": 1,
                "Topic": "MiniTopic",
                "Addr": "239.0.1.1",
                "Port": 30000
            }
        ],
        "Contexts": [
            {
                "Name": "Default",
                "Ip": "192.168.86.102",
                "IsSingleton": true,
                "RepairPortStart": 30001,
                "RepairPortEnd": 40000
            },
            {
                "REF": "Default",
                "Name": "MiniContext",
                "Id": 1,
                "TxTopics": [
                    {
                        "Name": "MiniTopic"
                    }
                ]
            }
        ]
    }
}
```

上面的配置中，MiniContext这个上下文中有一个主题，这个主题在Transports中只配置了一个分区PartitionNo = 1，当这个主题发送数据，只向配置的这个主题的组播地址和端口发送数据。

7.2.5. 负载均衡配置实例

```
{
    "Rcm": {
        "Transports": [
            {
                "Name": "Default",
                "PartitionNo": 1
            },
            {
                "Id": 1,
                "Topic": "MiniTopic",
                "PartitionNo": 1,
                "Addr": "239.0.1.1",
                "Port": 30000
            },
            {
                "Id": 2,
                "Topic": "MiniTopic",
                "PartitionNo": 2,
                "Addr": "239.0.1.2",
                "Port": 30000
            }
        ],
        "Contexts": [
            {
                "Name": "Default",
                "Ip": "192.168.86.102",
                "IsSingleton": true,
                "RepairPortStart": 30001,
                "RepairPortEnd": 40000
            },
            {
                "REF": "Default",
                "Name": "MiniContext",
                "Id": 1,
                "TxTopics": [
                    {
                        "Name": "MiniTopic"
                    }
                ]
            }
        ]
    }
}
```

上面的配置中，MiniContext这个上下文中有一个主题，这个主题在Transports中只配置了两个分区，其中一个为PartitionNo = 1，组播地址是"239.0.1.1"，组播端口是30000；另外一个分区是PartitionNo = 2，组播地址是"239.0.1.2"，组播端口是30000。当这个主题发送数据，会均衡的将数据发送到这个两个分区上，达到负载均衡的目的。接收端可以配置接收其中一个分区，当多个节点接收不同分区的数据，就可以达到负载均衡的目的。

代码中，创建发送端的时候，可以指定可用的分区，也可以不指定，不指定则为所有分区都可用，创建Transmitter的代码如下：

```
uint16_t partitions[] = { 1, 2, 0 };  // 0 表示结束分区号
m_lpRcmTransmitter = m_lpRcmContext->CreateTransmitter(szTopicName, partitions);
```

Send可以实现负载均衡，如果nPartitionNo为0，则会均衡的发送给所有可用的分区，如果想要只发送到分区1，则在发送的时候需要指定发送的分区号，代码如下：

```
uint16_t nPartitionNo = 1;
auto ret = m_lpRcmTransmitter->Send(lpAllocator, nMsgSize, nPartitionNo);
```

上面的代码，将会把消息全部发送给分区1。

7.2.6. 通过共享内存通信

RCM支持通过共享内存通信，这种场景下，接收端和发送端需要在同一台机器上面，接收端直接从本地共享内存中读取消息数据，然后通过共享内存反馈确认信息给发送端，这种模式能够最大限度的降低延迟。要启用共享内存模式，需要修改配置文件，主题中，需要增加如下配置：

```
        "Transports": [
            {
                ……
                "Topic": "ShmTopic",
                "UseSharedMemory": 1,
                ……
            }
        ]
```

在接收端上下文配置中，也需要修改配置，如下：

```
        "Contexts": [
            {
                ……
                "ShmContext": ["ShmSender1", "ShmSender1"],
                ……
            }
        ]
```

ShmSender1, ShmSender2是发送端的上下文名，共享内存也可以和其他节点组建高可用集群，配置和使用方式一致。

7.2.7. 接收端单机多并发功能

为了达到千万级别的消息吞吐量，接收端支持单机多线程高并发能力，即为每个主题创建一个接收线程，进行并发处理，这个时候，不对消息进行全局排序处理，且用户回调函数OnMessage也不是线程安全的，OnMessage中的nThreadNo会返回处理数据的线程号，线程号从0开始递增，上限是接收主题的数量，这种模式下，接收端上下文的配置如下所示：

```
{
    "Rcm": {
        "Transports": [
            {
                "Name": "Default",
                "PartitionNo": 1
            },
            {
                "Id": 1,
                "Topic": "MiniTopic0",
                "Addr": "239.0.1.1",
                "Port": 30000
            },
            {
                "Id": 2,
                "Topic": "MiniTopic1",
                "Addr": "239.0.1.2",
                "Port": 30000
            }
        ],
        "Contexts": [
            {
                "Name": "Default",
                "Ip": "192.168.86.102",
                "IsSingleton": true,
                "IsTotalOrder": false,
                "RepairPortStart": 30001,
                "RepairPortEnd": 40000
            },
            {
                "REF": "Default",
                "Name": "MiniContext",
                "Id": 1,
                "RxTopics": [
                    {
                        "Name": "MiniTopic0"
                    },
                    {
                        "Name": "MiniTopic1"
                    }
                ]
            }
        ]
    }
}
```

上面配置中，上下文中的参数IsSingleton必须为true，IsTotalOrder必须为false，同时在RxTopics中配置多个主题。

7.2.8. 接收端单机排序

单机非并发，会全局的对多个主题的消息进行排序，并产生连续递增的消息序号，回调用户接口是线程安全的，配置实例如下：

```
{
    "Rcm": {
        "Transports": [
            {
                "Name": "Default",
                "PartitionNo": 1,
            },
            {
                "Id": 1,
                "Topic": "MiniTopic0",
                "Addr": "239.0.1.1",
                "Port": 30000
            },
            {
                "Id": 2,
                "Topic": "MiniTopic1",
                "Addr": "239.0.1.2",
                "Port": 30000
            }
        ],
        "Contexts": [
            {
                "Name": "Default",
                "Ip": "192.168.86.102",
                "IsSingleton": true,
                "IsTotalOrder": true,
                "RepairPortStart": 30001,
                "RepairPortEnd": 40000
            },
            {
                "REF": "Default",
                "Name": "MiniContext",
                "Id": 1,
                "RxTopics": [
                    {
                        "Name": "MiniTopic0"
                    },
                    {
                        "Name": "MiniTopic1"
                    }
                ]
            }
        ]
    }
}
```

上面配置中，上下文中的参数IsSingleton必须为true，IsTotalOrder必须为true，同时在RxTopics中配置多个主题。此处需要特别注意，接收的多个主题，非高可用场景下可以不一致，而在高可用场景下配置的端口号必须一致，否则会报错。

7.2.9. 接收端均衡负载

当发送端进行了分片发送，接收端可以指定主题的分区号，只接收这个分区的数据，配置实例如下：

```
{
    "Rcm": {
        "Transports": [
            {
                "Name": "Default",
                "PartitionNo": 1,
            },
            {
                "Id": 1,
                "Topic": "MiniTopic",
                "PartitionNo": 1,
                "Addr": "239.0.1.1",
                "Port": 30000
            },
            {
                "Id": 2,
                "Topic": "MiniTopic",
                "PartitionNo": 2,
                "Addr": "239.0.1.2",
                "Port": 30000
            }
        ],
        "Contexts": [
            {
                "Name": "Default",
                "Ip": "192.168.86.102",
                "IsSingleton": true,
                "IsTotalOrder": false,
                "RepairPortStart": 30001,
                "RepairPortEnd": 40000
            },
            {
                "REF": "Default",
                "Name": "MiniContext",
                "Id": 1,
                "RxTopics": [
                    {
                        "Name": "MiniTopic",
                        "Partitions": [1]
                    }
                ]
            }
        ]
    }
}
```

上面配置中，主题MiniTopic配置了两个分区，新增Partitions配置项，指定只接收分区1的数据。

7.2.10. 接收端高可用

RCM高可用场景，通过zookeeper仲裁选举出leader，leader负责将所有主题的消息进行排序，并将排序后的消息在集群中同步，保证整个集群的强一致性，配置实例如下：

```
{
    "Rcm": {
        "Transports": [
            {
                "Name": "Default",
                "PartitionNo": 1,
            },
            {
                "Id": 1,
                "Topic": "MiniTopic",
                "Addr": "239.0.1.1",
                "Port": 30000
            }
        ],
        "Contexts": [
            {
                "Name": "Default",
                "Ip": "192.168.86.102",
                "IsSingleton": true,
                "RepairPortStart": 30001,
                "RepairPortEnd": 40000
            },
            {
                "REF": "Default",
                "Name": "Tier",
                "IsSingleton": false,
                "SyncIp": "192.168.86.102",
                "SyncAddr": "239.4.1.1",
                "SyncPort": 30000,
                "SyncRepairPortStart": 30001,
                "SyncRepairPortEnd": 40000,
                "UseSameMachine": true,
                "IsTotalOrder": true
            },
            {
                "REF": "Tier",
                "Name": "HaNode1",
                "Id": 2,
                "TierName": "HaTest",
                "RxTopics": [
                    {
                        "Name": "MiniTopic"
                    }
                ]
            },
            {
                "REF": "Tier",
                "Name": "HaNode2",
                "Id": 3,
                "TierName": "HaTest",
                "RxTopics": [
                    {
                        "Name": "MiniTopic"
                    }
                ]
            }
        ]
    }
}
```

上面配置中，IsSingleton必须为false，IsTotalOrder必须为true，同时必须配置SyncAddr和SyncPort，其中，SyncAddr不能和主题的地址相同，SyncPort用于集群内序号的同步。TierName需要配置成一致，用于表示多个节点是同一个集群。

当创建接收端集群上下文的时候，需要指定zookeeper的地址，代码如下：

```
const char* szZkAddr = "192.168.86.102:2181";
const char* szZkPath = "/rcm";
m_lpRcmFactory->SetArbArg(szZkAddr, szZkPath);
```

szZkAddr是zookeeper集群的服务地址，szZkPath是这个集群的管理路径。

7.3. 主题

RCM基于主题的发布和订阅，一个主题可以有多个发送端发布消息，同时一个主题可以被多个接收端订阅，一个接收端也可以订阅多个主题，同时通过高可用场景，对所有主题进行排序后同步，保证集群的强一致性。

主题支持组播发布和订阅，也支持udp单播进行发布和订阅。

基于组播的主题，配置如下：

```
        "Transports": [
            {
                "Name": "Default",
                "PartitionNo": 1,
            },
            {
                "Id": 1,
                "Topic": "MiniTopic",
                "Addr": "239.0.1.1",
                "Port": 30000
            }
        ]
```

上面的Addr如果是有效的组播地址，范围是(224.0.0.0～239.255.255.255)，则走组播功能，如果配置的Addr不是有效的组播地址，而是多个IP组成的，则通过udp单播发送数据,多个ip之间 ',' 分割。配置实例如下：

```
        "Transports": [
            {
                "Name": "Default",
                "PartitionNo": 1,
            },
            {
                "Id": 1,
                "Topic": "MiniTopic",
                "Addr": "192.168.0.1,192.168.0.2",
                "Port": 30000
            }
        ]
```

7.4. 分区

分区可以用于负载均衡，提升集群数据的并发性能，使用方式之前已经阐述。

7.5. RCM事件

事件用于RCM通知应用方内部消息，包括节点角色切换等，事件的定义如下：

```
// 事件类型
enum
{
    RCM_MESSAGE_LOSS = 1,                // 消息丢失
    RCM_OUT_OF_SYNC,                     // 状态不同步
    RCM_HEARTBEAT_TIMEOUT,               // 心跳超时
    RCM_NO_TRANSMITTER,                  // 无发送端
    RCM_NO_RECEIVER,                     // 无接收端
    RCM_NEW_TRANSMITTER,                 // 新加入发送端
    RCM_NEW_RECEIVER,                    // 新接入接收端
    RCM_FILL_GAP,                        // 组播补缺
    RCM_RECEIVER_LATE_JOIN,              // 新加入备机节点
    RCM_TIER_RECEIVER_DOWN,              // 接收端集群节点下线
    RCM_EVENT_MAX
};

// 事件接口
class IRcmEvent
{
public:
    virtual ~IRcmEvent() {}

    virtual uint32_t GetEventType() = 0;
    virtual const char * GetRcmContext() = 0;
    virtual const char * GetTopicName() = 0;
    virtual uint16_t GetPartitionNo() = 0;
    virtual uint64_t GetLossMsgNoStart() = 0;
    virtual uint64_t GetLossMsgNoEnd() = 0;
};

// 事件回调接口
class IRcmEventCallback
{
public:
    virtual ~IRcmEventCallback() {}

    /*
        * 组播事件回调
        * @param lpUser 用户参数
        * @param lpRcmEvent 事件
        * @return 无
    */
    virtual void OnEvent(void *lpUser, IRcmEvent *lpRcmEvent) = 0;
};
```

用户定义事件，需要继承IRcmEventCallback并实现接口OnEvent，根据IRcmEvent接口访问事件信息，比如实现如下：

```
class CNodeEventCallback : public IRcmEventCallback {
public:
    virtual ~CNodeEventCallback() {}

    virtual void OnEvent(void *lpUser, IRcmEvent *lpRcmEvent) {
        std::cout << "event type callback: " << lpRcmEvent->GetEventType() << std::endl;
    }
};
```

在创建工厂后，需要设置用户事件回调，代码如下：

```
CNodeEventCallback eventCallback;
m_lpRcmFactory->SetEventCallback(&eventCallback, NULL);
```

7.6. RCM日志

RCM支持用户自定义日志接口，接口如下：

```
// 日志级别
enum
{
    RCM_LOG_DEBUG = 0,     // 调试
    RCM_LOG_INFO,          // 信息
    RCM_LOG_WARN,          // 告警
    RCM_LOG_ERROR,         // 错误
    RCM_LOG_IMP,           // 重要事件
    RCM_LOG_FATAL          // 致命错误
};

// 日志接口
class IRcmLogger
{
public:
    virtual ~IRcmLogger() {}

    /*
        * 日志输出
        * @param nKey 日志码
        * @param nLevel 日志级别
        * @param lpContent 日志内容
        * @return 无
    */
    virtual void Log(int32_t nKey, int32_t nLevel, const wchar_t *lpContent) = 0;
};
```

用户需要实现自己的日志接口，实例如下：

```
class CNodeLogger : public IRcmLogger {
public:
    virtual ~CNodeLogger() {}

    virtual void Log(int32_t nKey, int32_t nLevel, const wchar_t *lpContent) {
        std::cout << "Key: " << nKey << ", Level: " << nLevel
            << ", content: " << lpContent << std::endl;
    }
};
```

创建RCM工厂后，需要指定loger，代码如下：

```
CNodeLogger nodeLoger;
m_lpRcmFactory->SetLogArg(RCM_LOG_INFO, &nodeLoger);
```

---

8. RCM的高可用

RCM高可用主要基于CAP三原则，CAP原则又称CAP定理，指的是在一个分布式系统中的一致性（Consistency）、可用性（Availability）、分区容忍性（Partition tolerance）。RCM集群是强一致性的，它通过zookeeper仲裁选举出leader，通过leader将数据状态同步给集群中其他节点，并保证同步成功后才认为数据写成功。RCM也是高可用的，通过zookeeper进行集群管理，并将数据写入集群多个节点上。在前面两个条件严格满足之下，RCM通过组播功能来提升性能，这里重点阐述组播的功能。

RCM的组播分为两种数据的组播，包括原始消息数据，以及leader的排序序号，原始消息从发送端直接组播给所有节点，保证了消息在网络中只传输一次，同时leader的排序序号也通过组播发送给集群中的节点，保证序号也只发送一次，通过上述方案的保证，使RCM的高可用场景能够达到5us以内的延迟，吞吐量达到千万级别。

同时RCM是一个软件系统，在其部署，运维，升级的过程中，以及节点的可靠性上，不能保证一定没有失误，RCM通过数据恢复，数据持久化，异地灾备等功能，最大限度的保证系统的可用性和安全性。

8.1. Zookeeper仲裁

在高可用场景下，RCM通过zookeeper管理集群的配置，集群节点管理，leader选举，比如通过上述实例中，会在zookeeper中创建出下面的zookeeper目录：

```
|-- rcm
|    |-- config
|        -- 1
|    |-- cluster_HaTest
|        |-- classer
|            -- HaNode1
|            -- HaNode2
|    |-- master
|        -- running
|    |-- observer
```

/rcm/config/1 是集群的配置，所有节点都从这个节点读取配置数据，保证集群配置的一致性，如果有多个配置，可以配置多个，比如1,2,3，RCM会以此将1,2,3的内容进行字符串拼接，组成一个完整的配置。

/rcm/cluster_HaTest/classer 管理集群所有节点，这里有两个节点HaNode1和HaNode2。

/rcm/cluster_HaTest/master 记录主节点信息，这个节点的数据在集群所有节点退出后，数据仍然保留，当集群第一个节点加入，这个节点不是master记录的节点，则会失败报错退出。

/rcm/cluster_HaTest/master/running 如果主节点是active的，则这个zk节点将存在，并且记录活跃主节点的信息。

/rcm/cluster_HaTest/observer 保存管理所有观察者节点。

8.2. Leader选举

在RCM高可用场景中，leader的选举分为下面几种情况进行讨论。

1. 集群节点为空，加入一个新节点，这时新加入的节点会成为Leader节点，但是在RCM内部，会将节点角色标记为Single，对于single节点，会直接处理接收到的所有主题的消息，同时向发送端确认数据已经被集群处理，发送端会清除数据，这个时候数据存在单点风险，这种场景需要应用方特别注意，是否符合应用方的安全需求。
2. 集群节点为空，同时加入多个节点，这个时候会触发Leader选举，选举规则是竞争/rcm/cluster_HaTest/master/running的创建权限，谁先创建成功，谁就是leader。
3. 集群中有一个主节点，新节点加入，这种情况新加入的节点直接成为备节点。
4. 主节点宕机，所有备节点会收到主节点下线的通知，从而触发竞争/rcm/cluster_HaTest/master/running的创建权限，谁先创建成功，谁就是leader。
5. 备节点宕机，集群所有节点会收到这个节点下线的消息，但是不触发选主。

8.3. 发送端数据持久化

为了提升io吞吐，RCM发送端采用了滑动窗口机制，每次发送数据包，会根据配置的MTU计算最大包长，并从滑动窗口中取出多个消息，直到写满最大包长，滑动窗口分为两种，一种是发送端原始消息，一种是RCM集群内序号同步消息，其配置如下所示：

```
        "Contexts": [
            {
                ……
                "SendWindowSize": 1024,
                "SyncSendWindowSize": 1024,
                ……
            }
        ]
```

发送端调用Send接口，会将数据直接写入共享内存中，通过RCM的allocator申请共享内存，将数据写入，然后调用Send接口：

```
char* lpData = (char *)lpAllocator->Malloc(nMsgSize);  // 通过rcm分配器申请内存，避免二次拷贝
memcpy(lpData, msg.c_str(), msg.size());
int32_t ret = m_lpRcmTransmitter->Send(lpAllocator, nMsgSize);
```

另外一种方式是，直接将buffer传入SendData：

```
std::string szData = "test data";
int32_t ret = m_lpRcmTransmitter->SendData(szData.c_str(), szData.size()+1);
```

第二种方式，RCM会自动从共享内存分配空间。

对于快速开始中的例子，RCM会在发送端的/dev/shm路径下创建下面四个文件：

```
client1_allocator
client1_req_1_sendingwindow
client1_req_1_sendingwindowblock_1
client1_req_1_sendingwindowblock_2
```

用户数据会写入client1_allocator，共享内存的大小有限制，可以通过配置文件修改最大共享内存，用户写满共享内存后，如果接收端没有确认，则无法继续写数据，当发送端收到接收端的确认，会清理掉已经确认的消息，然后释放共享内存，应用方可以继续写数据，共享内存大小配置是：

```
        "Contexts": [
            {
                ……
                "MaxMemoryAllowedMBytes": 256,
                ……
            }
        ]
```

上面的实例表示共享内存大小是256M。

发送端需要注意，因为用户数据直接写入共享内存成功，则告诉应用方写入成功，但是其实数据并没有真正发送到接收集群，如果发送端宕机或者共享内存损坏，已经写入的数据不能保证成功写入RCM集群并做到高可用。

MTU的配置也分为发送端发送的MTU以及集群内消息同步的MTU，发送端MTU配置在主题中，实例如下：

```
        "Transports": [
            {
                ……
                "MTU": 1500,
                ……
            }
        ]
```

集群内数据同步MTU：

```
        "Contexts": [
            {
                ……
                "MTU": 1500,
                ……
            }
        ]
```

MTU最好配置为1500，一般局域网内的MTU都是1500，太大会在数据链路层出现分片，太小会浪费传输效率。

8.4. 异步持久化

为了避免节点宕机，RCM支持将数据持久化到磁盘，在节点恢复时，首先从磁盘读取数据进行恢复，然后从主节点同步最新完整数据。因为是集群模式，异步持久化的数据即使不完整，也可以从集群中的主节点同步完整数据。

使用持久化，需要在配置中增加配置项：

```
        "Contexts": [
            {
                ……
                "IsRecord": true,
                ……
            }
        ]
```

持久化存储的磁盘路径，需要在创建RCM工厂的时候指定，代码如下：

```
const char* szConfigPath = "./rcm.json";
const char* szWorkPath = "/var/rcm"
const char* szLanguage = "zh_CN";
IRcmFactory* m_lpRcmFactory = NewRcmFactory( szConfigPath,
                                             szWorkPath,
                                             szLanguage );
```

根据创建指定的工作路径，会在接收端的/var/rcm下创建HaNode2/rcm_sqn_data_0.dat，集群的数据会写入rcm_sqn_data_0.dat中。当节点重新启动，会首先加载这个数据文件，然后再从主节点或者发送端同步最新数据。

8.5. 集群响应机制

RCM高可用场景，主节点向备节点同步数据，需要收到备节点的确认消息后，才提交当前的消息，确认备节点的数量可以配置，实例如下：

```
        "Contexts": [
            {
                ……
                "AckCount": 1,
                ……
            }
        ]
```

AckCount表示主节点需要多少个备节点对相同的消息序号进行确认，如果配置为0或者1，则只要有备节点确认就提交。如果配置为-1，则需要所有备节点的确认才提交，这种情况，主节点是通过zookeeper获取整个集群的备节点信息，只要zookeeper中的所有节点进行了确认，就认为得到了所有节点的确认，此时需要应用方确认，没有及时加入zookeeper集群的节点的安全性。如果集群中节点数量固定，且需要强安全性，则最好配置确定的数量，比如集群节点5个，需要4个节点的确认，则将AckCount配置为4，只要有节点宕机，整个集群会暂时阻塞直到节点恢复。

8.6. 主备切换

当主节点宕机，会触发主备切换，所有备节点会触发竞争/rcm/cluster_HaTest/master/running的创建权限，谁先创建成功，谁就是leader。

主备发生切换，需要注意，如果集群配置了AckCount为 -1（或者配置成集群固定数量），因为通过zookeeper管理的集群所有节点都收到主节点一致的同步数据，所以新当选的主节点能保证集群的强一致性。如果AckCount配置的数量小于备节点数量，而此时当选的主节点恰好是没有收到上一届主节点完整同步数据的节点，集群中的序号和数据可能不一致。

8.7. 启动

启动分为几种场景：

1. 所有节点清空启动，包括/dev/shm下面的共享内存数据，本地磁盘持久化数据。这时所有节点竞争选主，然后组建集群。
2. 节点不清空启动，zookeeper清空，这个时候，每个节点都可能有数据，但是zookeeper上没有保存主节点信息，需要应用方比对每个节点，确认最多持久化的节点先启动并成为leader，然后启动其他节点，保证集群数据从上一次状态恢复。
3. 节点不清空，zookeeper保留了上一次的主节点信息，这个时候只能是zookeeper中保留的主节点才能启动成功，其他节点启动将报错。主节点启动后，其他节点可以成功启动。
4. 发送端清流重启，整个RCM系统，都通过消息序号进行一致性检查，如果发送端清流重启，序号默认是0，那么已经发送的数据将被丢弃，直到序号达到之前最后发送的序号，如果需要保证清流重启后都是发送新数据，则需要配置发送端的起始序号。
5. 接收端清流重启，接收端清流重启，参照新节点加入。

8.8. 新节点加入

新节点加入是指集群中已经有主节点，备节点加入集群，因为集群中节点已经在处理数据，所有备节点需要从集群中的主节点将数据同步后，才能加入集群进行正常业务处理，它分为下面几个流程：

1. 加载本地持久化数据。
2. 向集群主节点发送延迟加入同步请求，获取和更新主节点最大序号。
3. 根据缺失的序号向主节点同步序号和消息数据。
4. 检查是否达到同步容忍范围，如果达到则启动成功。

同步容忍范围是指本地收到的序号和主节点的序号之间的差距在可接收范围内，可以通过配置设置：

```
        "Contexts": [
            {
                ……
                "SyncMaxDiff": 10,
                ……
            }
        ]
```

8.9. 观察者模式

RCM支持观察者模式，在组建的集群中，如果节点是观察者，则它只接收主题数据和主节点的状态同步数据，只需要保证和主节点的状态一致，不响应序号确认等信息，也不参与主备切换，节点成为观察者，需要修改配置文件：

```
        "Contexts": [
            {
                ……
                "IsObserver": true,
                ……
            }
        ]
```

如果节点成功成为观察者，则在zookeeper的/rcm/cluster_HaTest/observer路径下，会有节点信息。

---

9. 开发接口

本章主要介绍开发包提供的所有接口，函数参数说明。

9.1. 引出函数

9.1.1. 创建可靠一致组播工厂（NewRcmFactory）

函数原型：

```
IRcmFactory *NewRcmFactory(const char *szConfig, const char *szWorkDir = NULL, const char *szLang =
NULL);
```

输入参数：

```
szConfig json格式可靠一致组播配置文件名
szWorkDir 工作目录，默认取当前目录，rcm会在工作目录下创建recoder、log目录
szLang 日志语言 目前支持"zh_CN"，默认为"zh_CN"
```

返回：

```
返回IRcmFactory接口指针，NULL表示失败
```

用法说明：

9.1.2. 释放可靠一致组播工厂（DeleteRcmFactory）

函数原型：

```
void DeleteRcmFactory(rcm::IRcmFactory *lpIRcmFactory);
```

输入参数：

```
lpIRcmFactory 工厂接口指针
```

返回：

```
无
```

用法说明：

9.1.3. 获取RCM库版本号（GetRcmVersion)

函数原型：

```
const char * GetRcmVersion();
```

输入参数：

```
无
```

返回：

```
可靠组播库版本信息
```

用法说明：

9.2. 发送端内存分配器接口【IRcmAllocator】

9.2.1. 申请内存（Malloc）

函数原型：

```
virtual void *Malloc(uint32_t nSize) = 0;
```

输入参数：

```
nSize 内存大小
```

返回：

```
内存地址，NULL表示失败
```

用法说明：

9.2.2. 扩大/缩小当前内存（Realloc）

函数原型：

```
virtual void *Realloc(uint32_t nSize) = 0;
```

输入参数：

```
nSize 内存大小
```

返回：

```
内存地址，可能和Malloc返回的地址不同，NULL表示失败
```

用法说明：

9.3. 发送端接口【IRcmTransmitter】

9.3.1. 获取线程内存分配器（GetAllocator）

函数原型：

```
virtual IRcmAllocator *GetAllocator(uint32_t nThreadNo) = 0;
```

输入参数：

```
nThreadNo 线程编号，上层用户所编的线程编号
```

返回：

```
返回内存分配器指针，NULL表示失败
```

用法说明：

```
内存分配器在发送端内有效，每个发送线程必须使用独立的内存分配器，不能共用
```

9.3.2. 发送创建好的消息（Send）

函数原型：

```
virtual int32_t Send(IRcmAllocator *lpAllocator, uint32_t nLength, uint16_t nPartitionNo = 1,uint16_t
nMsgArg = 0) = 0;
```

回调参数：

```
lpAllocator 消息使用的内存分配器
nLength 消息长度
nPartitionNo 分区号，0表示负载均衡，其他为分区号，默认为1号分区
nMsgArg 任意消息相关数据，接收端通过IRcmMsgInfo的接口GetMsgArg()取到该参数
```

返回：

```
返回0表示发送成功，其他失败
```

用法说明：

```
零拷贝数据发送，非线程安全，Send失败后，下次调用Malloc会覆盖当前分配的内存
```

9.3.3. 发送消息（SendData）

函数原型：

```
virtual int32_t SendData(const void *lpData, uint32_t nLength, uint16_t nPartitionNo = 1, uint16_t
nMsgArg = 0) = 0;
```

回调参数：

```
lpData 消息内存
nLength 消息长度
nPartitionNo 分区号，0表示负载均衡，其他为分区号，默认为1号分区
nMsgArg 任意消息相关数据，接收端通过IRcmMsgInfo的接口GetMsgArg()取到该参数
```

返回：

```
返回0表示发送成功，其他失败
```

用法说明：

```
拷贝数据发送，非线程安全
```

9.3.4. 设置是否发送数据（SetSendFlag）

函数原型：

```
virtual void SetSendFlag(bool bSendFlag) = 0;
```

输入参数：

```
bSendFlag  true表示发送，false表示只占用消息序号，底层不通过网络发送出去，默认为true
```

返回：

```
无
```

用法说明：

9.4. 接收端接口【IRcmReceiver】

9.4.1. 确认消息处理完成（SendAck）

函数原型：

```
virtual int32_t SendAck(uint64_t nAckSqn) = 0;
```

输入参数：

```
nAckSqn Ack序号
```

返回：

```
返回0表示发送成功，其他失败
```

用法说明：

```
不在通信线程内处理消息的话，应用层需要调用SendAck确认消息处理完成
```

9.5. 接收端消息回调接口【IRcmReceiverCallback】

9.5.1. 消息回调，处理接收消息（OnMessage）

函数原型：

```
virtual int32_t OnMessage(void *lpData, uint32_t nLength, uint64_t nMsgSqn, uint16_t nTxArg, IRcmMsgInfo
*lpRcmMsgInfo) = 0;
```

回调参数：

```
lpData 消息数据
nLength 消息长度
nMsgSqn 消息排队序号，不需要排队时，等于发送端消息序号
nTxArg 发送端设置的任意参数，一般用来设置发送端接收应答的分区
lpRcmMsgInfo 获取其他消息信息，回调返回后指针失效
```

返回：

```
返回0表示成功 其他失败,如果失败，将从失败序号继续回调
```

用法说明：

9.5.2. 角色改变通知（OnRoleChanged）

函数原型：

```
virtual void OnRoleChanged(int32_t nRole) {}
```

回调参数：

```
nRole  新的角色
```

用法说明：

9.6. 集群同步对象接口【IRcmTierSync】

9.6.1. 主动发起同步（Sync）

函数原型：

```
virtual int32_t Sync(const void* lpData, uint32_t nLength, uint64_t nTransNo = -1, bool bCopyData =
false) = 0;
```

输入参数：

```
lpData 消息内存
nLength 消息长度
nTransNo 事务号，-1表示底层按照发送时序顺序编号，其他表示应用层主动编号，底层按照编号顺序同步
bCopyData 底层是否拷贝数据
```

返回：

```
返回0表示发送成功，其他失败
```

用法说明：

```
非线程安全，多线程调用Sync接口需要加锁
```

9.6.2. 回复应用确认（SendAck）

函数原型：

```
virtual int32_t SendAck(uint64_t nTransNo) = 0;
```

输入参数：

```
nTransNo 待接收事务号
```

返回：

```
return 0表示发送成功，其他失败
```

用法说明：

9.7. 集群数据同步回调接口【IRcmTierSyncCallback】

9.7.1. 接收同步消息（OnSyn）

函数原型：

```
virtual int32_t OnSyn(void* lpData, uint32_t nLength, uint64_t nTransNo, void* lpUser) = 0;
```

输入参数：

```
lpData 消息数据
nLength 消息长度
nTransNo 消息序号
lpRcmMsgInfo 获取其他消息信息，回调返回后指针失效，可用于获取消息来源上下文等信息
```

返回：

```
返回0表示成功 其他失败，如果失败，将从失败事务号继续回调
```

用法说明：

9.7.2. 角色改变通知（OnRoleChanged）

函数原型：

```
virtual void OnRoleChanged(int32_t nRole) {}
```

回调参数：

```
nRole  新的角色
```

用法说明：

9.8. RCM消息接口【IRcmMsgInfo】

消息接口，主要是在请求或应答消息中设置或获取各种需要关心的字段。

9.8.1. 获取主题名称（GetTopicName）

函数原型：

```
virtual const char * GetTopicName() = 0;
```

输入参数：

```
无
```

返回：

```
主题名称
```

用法说明：

9.8.2. 获取分区号（GetPartitionNo）

函数原型：

```
virtual uint16_t GetPartitionNo() = 0;
```

输入参数：

```
无
```

返回：

```
返回分区信息
```

用法说明：

9.8.3. 获取用户参数（GetUserArg）

函数原型：

```
virtual void * GetUserArg() = 0;
```

输入参数：

```
无
```

返回：

```
返回用户参数
```

用法说明：

9.8.4. 获取发送端上下文名称（GetTxContextName）

函数原型：

```
virtual const char * GetTxContextName() = 0;
```

输入参数：

```
无
```

返回：

```
发送端上下文名称
```

用法说明：

9.8.5. 获取发送端消息序号（GetTxMsgNo）

函数原型：

```
virtual uint64_t GetTxMsgNo() = 0;
```

输入参数：

```
无
```

返回：

```
发送端消息编号
```

用法说明：

```
一个发送端的消息编号也是连续递增编号的，多个发送端的数据排队以后，会重新生成一个连续递增的编号
```

9.8.6. 获取消息排队后时间戳（GetMsgTimeMilli）

函数原型：

```
virtual uint32_t GetMsgTimeMilli() = 0;
```

输入参数：

```
无
```

返回：

```
消息排队时间戳，精确到毫秒
```

用法说明：

```
单位毫秒
```

9.8.7. 获取回调线程号（ GetThreadNo）

函数原型：

```
virtual uint16_t GetThreadNo() = 0;
```

输入参数：

```
无
```

返回：

```
返回回调线程号
```

用法说明：

9.8.8. 获取消息参数（GetMsgArg）

函数原型：

```
virtual uint16_t GetMsgArg() = 0;
```

输入参数：

```
无
```

返回：

```
获取消息参数
```

用法说明：

9.9. 可靠一致组播上下文接口【IRcmContext】

9.9.1. 创建发送端（CreateTransmitter）

函数原型：

```
virtual IRcmTransmitter * CreateTransmitter(const char *lpTopicName, uint16_t *lpPartitionNos = NULL,
uint16_t nTxArg = 0) = 0;
```

输入参数：

```
lpTopicName 主题名称
lpPartitionNos 为NULL表示使用所有分区都会发送，否则为分区号数组，0结束
nTxArg 本发送端设置的任意参数，可以用于请求应答分区匹配，接收端通过OnMessage接口取到该参数
```

返回：

```
返回IRcmTransmitter接口指针，NULL表示失败
```

用法说明：

一个上下文可以创建多个发送端，用户应保证同一集群中的上下文创建的发送端和接收端一致

9.9.2. 创建接收端（CreateReceiver）

函数原型：

```
virtual IRcmReceiver* CreateReceiver(IRcmReceiverCallback *lpIRcmReceiverCallback, void *lpUser = NULL) =
0;
```

输入参数：

```
lpIRcmReceiverCallback 消息回调函数接口指针
lpUser 用户设置的参数，用于消息回调函数OnMessage
```

返回：

返回接收句柄

用法说明：

每个上下文最多只能创建一个接收端对象。接收端接收的主题、分区信息在配置文件中配置

9.9.3. 集群数据同步对象（CreateTierSync）

函数原型：

```
virtual IRcmTierSync * CreateTierSync(uint64_t uStartTransNo, IRcmTierSyncCallback*
lpIRcmTierSyncCallback, void* lpUser = NULL) = 0;
```

输入参数：

```
uStartTransNo 开始的消息序号
lpIRcmTierSyncCallback 集群同步消息回调函数接口指针
lpUser 用户设置的参数，用于回调函数OnSync
```

返回：

返回复制接收句柄 IRcmTierSync*

用法说明：

创建集群数据同步对象，同步模式(同步、异步、半同步)等参数在上下文配置中

9.9.4. 启动上下文，如果为集群，获取集群中角色（Start）

函数原型：

```
virtual int32_t Start() = 0;
```

输入参数：

```
无
```

返回：

```
返回0表示成功，其他表示失败
```

用法说明：

```
上下文加入集群，同时上下文会获取到自身的身份
```

9.9.5. 停止上下文，退出集群（Stop）

函数原型：

```
virtual int32_t Stop() = 0;
```

输入参数：

```
无
```

返回：

```
返回0表示成功，其他表示失败
```

用法说明：

```
上下文停止，会退出当前集群
```

9.9.6. 获取角色（GetRole）

函数原型：

```
virtual int32_t GetRole() = 0;
```

输入参数：

```
无
```

返回：

```
返回当前角色
```

用法说明：

```
角色的枚举值如下：
enum
{
    RCM_ROLE_SEPARATE       = 0x00,             // 分离状态，应用应退出
    RCM_ROLE_INIT           = 0x01,             // 初始状态
    RCM_ROLE_SYNC           = 0x02,             // 正在同步数据
    RCM_ROLE_INACTIVE       = 0x04,             // 备机
    RCM_ROLE_ACTIVE         = 0x10,             // 主机，有备机
    RCM_ROLE_SINGLE         = 0x11              // 单机，无备机
};
```

9.9.7. 获取上下文可靠等级（GetReliability）

函数原型：

```
virtual int32_t GetReliability() = 0;
```

输入参数：

```
无
```

返回：

```
RCM可靠等级
enum
{
    RCM_RELIABLE_UNRELIABLE     = 0,    // 不可靠
    RCM_RELIABLE_NOT_ORDERED    = 1,    // 单点无序
    RCM_RELIABLE_ORDERED        = 2,    // 单点有序
    RCM_RELIABLE_TIER           = 3     // 集群有序
};
```

用法说明：

9.9.8. 获取集群名称（GetTierName）

函数原型：

```
virtual const char * GetTierName() = 0;
```

输入参数：

```
无
```

返回：

```
返回集群名称
```

用法说明：

9.9.9. 初始化参数设置（SetInitParam）

函数原型：

```
virtual void SetInitParam(const char* key, const char* value) = 0;
```

输入参数：

```
key 字符串，当前只有"replay_start_point"
value value是一个uint64_t转化的字符串，表示起始加载序号
```

返回：

```
无
```

用法说明：

```
注意： key和value需要避免截断
```

9.10. 日志回调接口【IRcmLogger】

9.10.1. 日志输出（Log）

函数原型：

```
virtual void Log(int32_t nKey, int32_t nLevel, const wchar_t *lpContent) = 0;
```

回调参数：

```
nKey 日志码
nLevel 日志级别
lpContent 日志内容
```

返回：

```
无
```

用法说明：

```
日志级别包括：
enum
{
    RCM_LOG_DEBUG = 0,       // 调试
    RCM_LOG_INFO,            // 信息
    RCM_LOG_WARN,            // 告警
    RCM_LOG_ERROR,           // 错误
    RCM_LOG_IMP,             // 重要事件
    RCM_LOG_FATAL            // 致命错误
};
```

9.11. 事件接口【IRcmEvent】

9.11.1. 获取事件类型（GetEventType）

函数原型：

```
virtual uint32_t GetEventType() = 0;
```

输入参数：

```
无
```

返回：

```
事件类型
```

用法说明：

```
事件类型包括：
enum
{
    RCM_MESSAGE_LOSS = 1,               // 消息丢失
    RCM_OUT_OF_SYNC,                    // 状态不同步
    RCM_HEARTBEAT_TIMEOUT,              // 心跳超时
    RCM_NO_TRANSMITTER,                 // 无发送端
    RCM_NO_RECEIVER,                    // 无接收端
    RCM_NEW_TRANSMITTER,                // 新加入发送端
    RCM_NEW_RECEIVER,                   // 新接入接收端
    RCM_FILL_GAP,                       // 组播补缺
    RCM_RECEIVER_LATE_JOIN,             // 新加入备机节点
    RCM_TIER_RECEIVER_DOWN,             // 接收端集群节点下线
    RCM_EVENT_MAX
};
```

9.11.2. 获取发生事件的上下文（GetRcmContext）

函数原型：

```
virtual const char * GetRcmContext() = 0;
```

输入参数：

```
无
```

返回：

```
返回发生事件的上下文名称
```

用法说明：

```
可调用事件：
RCM_MESSAGE_LOSS
RCM_OUT_OF_SYNC
RCM_NO_TRANSMITTER
RCM_NO_RECEIVER
RCM_NEW_TRANSMITTER
RCM_NEW_RECEIVER
RCM_RECEIVER_LATE_JOIN
```

9.11.3. 获取发生事件的主题（GetTopicName）

函数原型：

```
virtual const char * GetTopicName() = 0;
```

输入参数：

```
无
```

返回：

```
发生事件的主题名
```

用法说明：

```
可调用事件：
RCM_MESSAGE_LOSS
RCM_OUT_OF_SYNC
RCM_NO_TRANSMITTER
RCM_NO_RECEIVER
RCM_NEW_TRANSMITTER
RCM_NEW_RECEIVER
RCM_RECEIVER_LATE_JOIN
```

9.11.4. 获取发生事件的分区编号（GetPartitionNo）

函数原型：

```
virtual uint16_t GetPartitionNo() = 0;
```

输入参数：

```
无
```

返回：

```
发生事件的分区编号
```

用法说明：

```
可调用事件：
RCM_MESSAGE_LOSS
RCM_OUT_OF_SYNC
RCM_NO_RECEIVER
RCM_NEW_TRANSMITTER
RCM_NEW_RECEIVER
RCM_RECEIVER_LATE_JOIN
```

9.11.5. 获取丢失的消息最小序号（GetLossMsgNoStart）

函数原型：

```
virtual uint64_t GetLossMsgNoStart() = 0;
```

输入参数：

```
无
```

返回：

```
丢失的消息最小序号
```

用法说明：

```
可调用事件：
RCM_OUT_OF_SYNC
```

9.11.6. 获取丢失的消息最大序号（GetLossMsgNoEnd）

函数原型：

```
virtual uint64_t GetLossMsgNoEnd() = 0;
```

输入参数：

```
无
```

返回：

```
丢失的消息最大序号
```

用法说明：

```
可调用事件：
RCM_OUT_OF_SYNC
```

9.12. 事件回调接口【IRcmEventCallback】

9.12.1. 组播事件回调（OnEvent）

函数原型：

```
virtual void OnEvent(void *lpUser, IRcmEvent *lpRcmEvent) = 0;
```

回调参数：

```
lpUser 用户参数
lpRcmEvent 事件
```

返回：

```
无
```

用法说明：

```
可靠组播会将底层的事件回调应用层，应用根据事件类型，然后通过lpRcmEvent获取具体的事件信息
```

9.13. 可靠一致组播工厂接口【IRcmFactory】

9.13.1. 创建可靠一致组播上下文（NewRcmContext）

函数原型：

```
virtual IRcmContext *NewRcmContext(const char *lpName) = 0;
```

输入参数：

```
lpName 上下文名称
```

返回：

```
返回IRcmContext接口指针，NULL表示失败
```

用法说明：

9.13.2. 释放可靠一致组播上下文（DeleteRcmContext）

函数原型：

```
virtual int32_t DeleteRcmContext(IRcmContext *lpIRcmContext) = 0;
```

输入参数：

```
lpIRcmContext IRcmContext接口指针
```

返回：

```
返回0成功，其他失败
```

用法说明：

9.13.3. 获取Rcm配置对象（GetConfig）

函数原型：

```
virtual IRcmConfig *GetConfig() = 0;
```

输入参数：

```
无
```

返回：

```
返回配置对象接口，NULL表示失败
```

用法说明：

9.13.4. 获取组播状态数据（GetStatistics）

函数原型：

```
virtual void GetStatistics(const char *lpRequest, char **lppResult) = 0;
```

输入参数：

```
lpRequest json格式请求字符串，"{}"为json格式
lppResult json格式结果字符串
```

返回：

```
无
```

用法说明：

```
lppResult指向的字符串指针为状态数据，内存不需要调用释放
```

lppResult参数说明如下：

```
发送端统计信息
{
    "ctxs": [
        {
            "ContextId": 1,                                                 // 上下文Id
            "Context": "MiniContext",                                       // 上下文名称
            "txs": [                                                        // 发送端统计信息
                [
                    {
                        "Addr": "239.0.1.1,",                               //组播地址
                        "Port": 5001,                                       //组播通讯端口
                        "MaxUdpSize": 1472,                                 //理论上最大可能的UDP包长度
                        "AvailableSpace": 1448,                             //组播非分片报文能携带的最大数据
                        "RegularLength": 1440,                              //组播分片报文能携带的最大数据
                        "SendFlag": true,                                   //是否传输数据
                        "TransportId": 1,                                   //主题Id
                        "TopicName": "MiniTopic",                           //主题名
                        "PartitionNo": 1,                                   //分区号
                        "SendCount": 18,                                    //发送报文个数
                        "LastSendMilli": 709725964,                         //上次发送时间
                        "CurPhaseTickMilli": 709720860,                     //进入当前阶段时间
                        "LastSendSynTickMilli": 709724863,                  //上次发送SYN时间
                        "ConnectionId": 8620,                               //会话号
                        "Phase": "Started",                                 //当前序号协商阶段
                        "Role": "Single",                                   //当前角色
                        "UsedMsgNo": 0,                                     //发送窗口可用消息起始序号
                        "FirstIndex": 0,                                    //发送窗口未发送的起始序号
                        "FirstMsgNo": 0,                                    //未发送消息起始序号
                        "FirstFragNo": 0,                                   //未发送消息分片起始序号
                        "ToSendMsgNo": 0,                                   //当前最大消息号
                        "ToSendFragNo": 0,                                  //当前最大分片号
                        "CurrentMsgNo": 10,                                 //当前发送消息号
                        "CurrentIndex": 10,                                 //当前发送窗口位置
                        "Ack": [                                            //收到的接收端ack信息
                            {
                                "ContextId": 1,                             //上下文Id
                                "Address": "192.168.86.101:5001",           //接收端Ip地址
                                "MsgNo": 0,                                 //接收端待接收序号
                                "FragNo": 0,                                //接收端待接收分片号
                                "LastRecvMilli": 709725864,                 //接收端最近回复时间
                                "AckCount": 0,                              //接收端回复ACK报文个数
                                "NakCount": 0,                              //接收端回复NAK报文个数
                                "Status": "CONNECTED",                      //连接状态
                                "IsMandotaryAcker": false                   //是否主接收者
                            },
                            {
```

```
                                "ContextId": 2,
                                "Address": "",
                                "MsgNo": 0,
                                "FragNo": 0,
                                "LastRecvMilli": 0,
                                "AckCount": 0,
                                "NakCount": 0,
                                "Status": "DISCONNECT",
                                "IsMandotaryAcker": false
                            },
                            {
                                "ContextId": 3,
                                "Address": "",
                                "MsgNo": 0,
                                "FragNo": 0,
                                "LastRecvMilli": 0,
                                "AckCount": 0,
                                "NakCount": 0,
                                "Status": "DISCONNECT",
                                "IsMandotaryAcker": false
                            }
                        ]
                    }
                ]
            ]
        }
    ]
}
```

接收端统计信息

```
{
    "ctxs": [
        {
            "ContextId": 1,                                          //上下文Id
            "Context": "MiniContext",                                //上下文名称
            "txs": [],
            "rx": {
                "Rms": [                                             //接收端统计信息
                    {
                        "ContextId": 1,                              //上下文Id
                        "LastRecvMilli": 710577472,                  //上一次收到数据的时间戳
                        "ConnectionId": 0,                           //会话号
                        "LastRstTickMilli": 710577472,               //上次回复RST时间
                        "TransportId": 1,                            //主题Id
                        "TopicName": "MiniTopic",                    //主题名
                        "PartitionNo": 1,                            //分区号
                        "NextMsgNo": 0,                              //下一个待接收消息序号
                        "NextMsgNoDeliver": 0,                       //下一个待处理消息序号
                        "MsgNoAcked": 0,                             //已经确认的消息序号
                        "MaxMsgNo": 0,                               //收到的最大消息序号
                        "MaxFragNo": 0,                              //收到的最大消息的最大分片号
                        "NextFragNo": 0,                             //当前待接收分片号
                        "MaxMsgNoNak": 0,                            //发起补缺的最大消息序号
                        "MaxFragNoNak": 0,                           //发起补缺的最大消息的最大分片号
                        "LastAckMilli": 710577472,                   //最后发送ACK的时间
                        "LastNakMilli": 0,                           //最后发送NAK的时间
                        "RmsPhase": "Init"                           //当前序号协商阶段
                    }
                ],
                "Tier": {                                            //集群统计信息
                    "Deliver": {                                     //已经接收并回调用户信息
                        "TierName": "MiniContext",                   //集群名
                        "Role": "Single",                            //节点角色
                        "FirstMsgSqn": 0,                            //第一个消息序号
                        "NextMsgSqn": 0,                             //下一个排队消息序号
                        "NextMsgSqnDeliver": 0,                      //下一个处理消息序号
                        "MsgsDelivered": 0,                          //已经处理的消息数量
                        "NextMsgSqnRecord": 0,                       //下一个持久化消息序号
                        "MsgsRecorded": 0,                           //已经持久化的消息数量
                        "WaitContextId": 0,                          //正在等待该发送端上下文ID
                        "WaitTransportId": 0,                        //正在等待该发送端主题ID
                        "CommittedSqn": 0                            //已提交最大序号
                    },
```

```
                    "Send": {
                        "ToSendMsgSqn": 0,                                  //当前待发送序号
                        "ToSendFragNo": 0,                                  //当前待发送分片号
                        "FlowControlCount": 0,                              //发送流控次数
                        "WaitAckMsgSqn": 0,                                 //所有成员的最大待确认序号
                        "WaitAckFragNo": 0,                                 //待确认分片号
                        "LastSendTickMilli": 0,                             //上一次发送消息的时间
                        "Ack": []
                    }
                }
            }
        }
    ]
}
```

9.13.5. 设置日志参数（SetLogArg）

函数原型：

```
virtual void SetLogArg(int32_t nMinLogLevel, IRcmLogger *lpRcmLogger = NULL) = 0;
```

输入参数：

```
nMinLogLevel 最小日志输出等级，小于该等级的日志不会回调，默认为INFO
lpRcmLogger 日志接口，如果为NULL，内部会写日志
```

返回：

```
无
```

用法说明：

```
日志在工作目录下的log文件夹
```

9.13.6. 设置仲裁参数（SetArbArg）

函数原型：

```
virtual void SetArbArg(const char *lpArbAddress, const char *lpArbPath) = 0;
```

输入参数：

```
lpArbAddress Arb服务地址，目前Arb服务支持zookeeper
lpArbPath 提供给仲裁写数据的路径
```

返回：

```
无
```

用法说明：

```
使用集群上下文时需要设置
```

9.13.7. 设置文件前缀（SetFilePrefix）

函数原型：

```
virtual void SetFilePrefix(const char *szFilePrefix) = 0;
```

输入参数：

```
szFilePrefix 文件前缀名
```

返回：

```
无
```

用法说明：

```
共享内存等的文件夹
```

9.13.8. 设置事件回调接口（SetEventCallback）

函数原型：

```
virtual void SetEventCallback(IRcmEventCallback *lpRcmEventCallback, void *lpUser = NULL) = 0;
```

输入参数：

```
lpRcmEventCallback 事件回调接口
lpUser 事件回调参数
```

返回：

```
无
```

用法说明：

9.14. rcm配置对象【IRcmConfig】

9.14.1. 获取组播上下文配置对象（GetRcmContextConfig）

函数原型：

```
virtual IRcmContextConfig * GetRcmContextConfig(const char *lpName) = 0;
```

输入参数：

```
lpName 上下文名称
```

返回：

```
返回IRcmContextConfig配置接口指针，NULL表示失败
```

用法说明：

9.15. rcm上下文配置对象【IRcmContextConfig】

9.15.1. 获取发送主题数目（GetTxTopicCount）

函数原型：

```
virtual uint16_t GetTxTopicCount() = 0;
```

输入参数：

```
无
```

返回：

```
返回发送主题数目
```

用法说明：

9.15.2. 获取发送主题信息（GetTxTopic）

函数原型：

```
virtual const char * GetTxTopic(uint16_t nIndex, uint16_t **lppPartitionNos) = 0;
```

输入参数：

```
nIndex 编号，从0开始
lppPartitionNos 返回分区数组，0结束
```

返回：

```
返回主题名称，NULL表示没有该编号的主题
```

用法说明：

9.15.3. 获取接收主题数目（GetRxTopicCount）

函数原型：

```
virtual uint16_t GetRxTopicCount() = 0;
```

输入参数：

```
无
```

返回：

```
返回接收主题数目
```

用法说明：

9.15.4. 获取接收主题信息（GetRxTopic）

函数原型：

```
virtual const char * GetRxTopic(uint16_t nIndex, uint16_t *lpPartitionNo) = 0;
```

输入参数：

```
nIndex 编号,从0开始
lpPartitionNo 返回分区号
```

返回：

```
返回主题名称，NULL表示没有该编号的主题
```

用法说明：

---

10. 附录

10.1. 配置说明

RCM	类型	说明	备注
Transports	数组	每个成员表示一个主题（或者是主题的配置模板）	
Name	字符串	模板名称，Transport名称或者Context名称	
REF	字符串	引用的模板名	其中Default为默认引用的模板
Id	整型	Transport编号或者Context编号	有效范围是[1, 65535]，默认为0
Topic	字符串	主题名	最大有效长度为127字节
PartitionNo	整型	分区编号	有效范围[1, 65535]，默认为1
UseSharedMemory	布尔型	是否启用共享内存通讯	默认为0
HeartbeatIntervalMilli	整型	心跳间隔	发送端定时发送心跳，单位毫秒，默认为1000毫秒
HeartbeatTimeoutMilli	整型	心跳超时时间	接收端一定时间没有收到发送端心跳，则认为该发送端不正常，单位毫秒，默认为3000毫秒
AckIntervalMilli	整型	确认间隔	接收端定时回复确认，单位毫秒，默认为1000毫秒

AckTimeoutMilli	整型	确认超时时间	发送端一定时间没有收到接收端确认，则认为接收端不正常，单位毫秒，默认为3000毫秒
AsynchronousRms	整数	是否采用异步发送	0 采用同步发送，直接调用sendto接口， 1 写入内存就返回，采用异步发送
Addr	字符串	通讯地址	可以配置为组播地址、单个ip地址、多个ip地址。多个地址以逗号分隔
Port	整型	通讯端口	
Context	字符串	上下文名	表示该Transport配置属于特定的上下文，上下文查找Transport时，优先查找属于本上下文的Transport配置
Contexts	数组		每个成员表示一个上下文（或者是上下文的配置模板）
Ip	字符串	网卡地址	需加入组播的IP地址（网卡），可以配置网段
RecordDir	字符串	持久化目录	
BindCoreId	整型	绑定CPU的CoreId	默认-1不绑定，否则为cpu核心编号，只有接收线程绑定
McLoop	整型	是否开启本地回环	作用于组播发送端，用于控制跟发送端处于同一台机器的接收端是否能够收到组播数 据，默认为1，表示允许接收本机组播
TTL	整型	存活时间	用于控制数据包在网络上的存活时间（转发跳数），默认为8
IsSingleton	布尔型	是否为单机模式	默认为1，表示单机模式
IsRecord	布尔型	是否持久化	默认为1
IsHandshake	整型	是否序号协商	默认为0，表示不进行序号协商
StartTimeoutSecs	整型	清流启动序号协商超时时间	默认值为5，单位秒
IsTotalOrder	布尔型	消息是否排序	默认为1，表示排序
MaxCacheMsgCount	整型	发送端缓存的消息数	发送端缓存的消息数，即使已经被确认也不会删除，默

			认为10240
RecordReplayStartPoint	整型	重放持久化的数据	
UsePreAllocate	布尔型	是否开启预分配	默认为0
LatencyStatisticCount	整型	是否开启延时统计	默认为0
MTU	整型	最大传输单元	默认是1500
SendWindowSize	整型	发送窗口大小	上下文内发送端的发送窗口大小，当发送与收到确认的UDP包数超过设置值，将暂停组播,默认为256
MaxMemoryAllowedMBytes	整型	上下文使用内存上限	上下文内所有发送端数据缓存的总大小，单位MB，默认为256MB
SocketBufferSizeKBytes	整型	套接字接收缓存大小	上下文内的接收端套接字的接收缓存大小，单位KB，默认为256KB
RepairPortStart	整型	补缺端口范围下限	上下文内的发送端会在此范围内选择一个可用的端口作为补缺端口，下限默认为30001，上限默认为40000
RepairPortEnd	整型	补缺端口范围上限	
RepairPort	整型	补缺端口	如果上下文只有一个发送端，那么可以通过RepairPort直接指定一个补缺端口，默认为0。注意：如果该端口不可用，会导致发送端创建失败
ShmContext	字符串	shm上下文名	
TierName	字符串	集群名	该上下文所属的集群名称
TierRmsLimitCount	整型	rms消息与sqn差距容忍范围，超过则不回ack，不接收rms消息	如果rms消息多于已经确认的sqn，则不再接收rms消息，只做序号同步
TierAsync	整型	异步结果复制	默认为0
SyncIp	字符串	需加入组播的IP地址（网卡），可以配置网段	
SyncAddr	字符串	集群同步组播地址	
SyncPort	整型	集群同步组播端口	
SyncMcLoop	整型	集群间同步是否开启本地回环	等于0时禁止，本机将收不到自己组播出去的数据，默认为1，表示放开本机接收组播数据

SyncTTL	整型	集群间同步的报文存活时间	默认为8
SyncMTU	整型	集群间同步网络的最大传输单元	默认值为1500
SyncMode	整型	集群数据同步模式	0表示状态机复制，1表示主备同步复制，2表示主备异步复制，默认值为0
SyncHeartbeatIntervalMilli	整型	集群同步组播心跳间隔	主节点定时发送心跳，单位毫秒，默认为1000毫秒
SyncHeartbeatTimeoutMilli	整型	集群同步组播心跳超时时间	从节点一定时间没有收到主节点心跳，则认为该主节点不正常，单位毫秒，默认为3000毫秒
SyncAckIntervalMilli	整型	集群同步组播ACK间隔	从节点定时回复确认，单位毫秒，默认为1000毫秒
SyncAckTimeoutMilli	整型	集群同步组播ACK超时时间	主节点一定时间没有收到从节点确认，则认为从节点不正常，单位毫秒，默认为3000毫秒
SyncSendWindowSize	整型	集群间同步的发送窗口大小	表示未收到任何确认时，主节点最多能够发送的排队序号个数，主要用于集群主节点发送流控，默认为256
SyncSocketBufferSizeKBytes	整型	套接字缓冲大小	单位KB，默认为256
SyncRepairPortStart	整型	集群间同步的补缺端口范围下限	集群主节点会在此范围内选择一个可用端口作为补缺端口，接收集群内其他成员的确认信息和补缺请求，下限默认为40001，上限默认为50000
SyncRepairPortEnd	整型	集群间同步的补缺端口范围上限	
SyncRepairPort	整型	集群间同步的补缺端口	非0表示集群主节点直接使用该值作为补缺端口，接收集群内其他成员的确认和补缺请求，默认为0。注意：如果该端口不可用，会导致主接收者创建失败
SyncMaxDiff	整型	集群备节点开机重启从主节点同步数据容忍范围	默认为10000
AckCount	整型	接收集群备节点确认数	-1表示全部确认，默认为1
AsynchronousSqn	整型	集群建sqn同步，采用异步方式（高吞吐下，开启）	默认为0，表示快速同步sqn
IsObserver	布尔型	是否为观察者	观察者只能用于接收集群数据，不能参与选主，默认为

			0
UseSameMachine	布尔型	是否集群中有节点部署在同一台机器上	默认为0
UseCommitted	整型	是否启用两阶段提交	0表示不启用，1表示启用，默认为0
TxTopics	数组	上下文发送主题信息	每个成员表示发送的主题信息
RxTopics	数组	上下文接收主题信息	每个成员表示接收该主题的分区信息
Partitions	数组	分区列表	每个成员表示一个分区号

11. Q&A