COMO(C++ Component Model)原理

# RPC（Remote Process Call）
远程过程调用

COMO RPC程序组成

- ☐ 客户端Client ---- Proxy
- ☐ 服务器端Server ---- Stub
- ☐ COMO远程对象服务管理ServiceManager
- ☐ 操作系统内核支持D-Bbus/IBinder

COMO远程接口自动列集\散集技术

　　当客户端Client和服务器端Server所在地址空间不同时，客户端进程对服务器端构件服务的调用，属于远程构件调用RPC。由于两个不同空间之间不允许彼此直接访问或者具有不同的访问权限，所以需要某种通讯机制来实现不同地址空间之间的数据交互。

　　COMO构件技术支持远程接口调用，通过数据的自动列集\散集技术进行不同地址空间的数据交互。构件服务和构件服务调用者可以处于操作系统的不同空间，而调用者可以如同在同一地址空间里面使用构件一样透明的进行远程接口调用，也就是说完全向用户屏蔽了底层使用的标准的列集\散集过程。

COMO对象流程调用示意图

内核空间　　内核Object对象　　内核空间

代理对象　　　　存根对象　　服务构件

1.调用
8 调用返回

客户端　　　　　　服务端
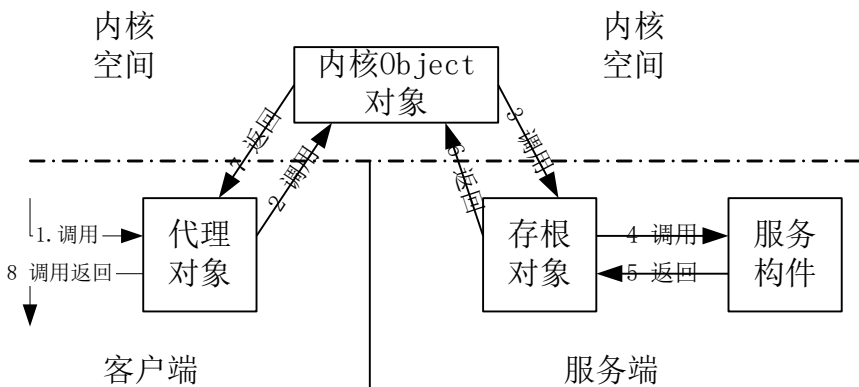
以存根\代理机制来实现远程接口自动列集\散集, 主要涉及到三个对象, 处于客户端的代理(Proxy)对象, 处于服务端的存根(stub)对象,以及处于内核的(Object)对象（基于D-Bus、IBinder的版本没有内核对象）。

一个客户端进程不一定只调用一个远程构件的服务，为了更方便有效的和各个远程构件交互数据，COMO在客户端为每一个对应的远程服务建立一个代理对象，记录一些客户进程的信息、远程服务的构件对象的信息以及一些调用的状态等，负责为客户进程与对应的远程服务联系。

COMO会为每个提供远程服务的构件对象建立一个存根对象，客户端代理不是直接与远程提供服务的构件对象联系的，而是与存根对象进行联系，通过存根对象来调用构件对象。

客户端Client -- Proxy

- 客户端是引用服务的一端。
- 通过这个机制，不只是让C++程序得以使用COMO构件服务，实现多语言的绑定也是很容易的。

客户端Client

□ 源码：/como/como/test/runtime/rpc/client/main.cpp

```
28   static AutoPtr<IService> SERVICE;
29
30   TEST(RPCTest, TestGetRPCService)
31   {
32       AutoPtr<IInterface> obj;
33       ServiceManager::GetInstance()->GetService("rpcservice", obj);
34       SERVICE = IService::Probe(obj);
35       EXPECT_TRUE(SERVICE != nullptr);
36   }
37
38   TEST(RPCTest, TestCallTestMethod1)
39   {
40       EXPECT_TRUE(SERVICE != nullptr);
41       ECode ec = E_REMOTE_EXCEPTION;
42       Integer result;
43       ec = SERVICE->TestMethod1(9, result);
44       EXPECT_EQ(9, result);
45       EXPECT_EQ(ec, NOERROR);
46   }
```

GetService中的obj哪里来的？
GetService()

☐ 源码：/como/como/servicemanager/lib/linux/ServiceManager.cpp

```
131  ECode ServiceManager::GetService(
132      /* [in] */ const String& name,
133      /* [out] */ AutoPtr<IInterface>& object)
134  {
```

```
209          if (replyData != nullptr) {
210              AutoPtr<IParcel> parcel;
211              CoCreateParcel(RPCType::Local, parcel);
212              parcel->SetData(reinterpret_cast<HANDLE>(replyData), replySize);
213
214              AutoPtr<IInterfacePack> ipack;
215              CoCreateInterfacePack(RPCType::Local, ipack);
216              IParcelable::Probe(ipack)->ReadFromParcel(parcel);
217              ec = CoUnmarshalInterface(ipack, RPCType::Local, object);
218          }
```

# GetService中的obj哪里来的？
CoUnmarshalInterface()

□   源码：/como/como/como/runtime/rpc/comorpc.cpp

```
105   ECode CoUnmarshalInterface(
106       /* [in] */ IInterfacePack* data,
107       /* [in] */ RPCType type,
108       /* [out] */ AutoPtr<IInterface>& object)
109   {
110       if (data == nullptr) {
111           object = nullptr;
112           return NOERROR;
113       }
114
115       AutoPtr<IRPCChannelFactory> factory =
116               type == RPCType::Local ? sLocalFactory : sRemoteFactory;
117       return factory->UnmarshalInterface(data, object);
118   }
```

GetService中的obj哪里来的？
UnmarshalInterface()

☐   源码：/como/como/como/runtime/rpc/dbus/CDBusChannelFactory.cpp

```
113    ECode CDBusChannelFactory::UnmarshalInterface(
114        /* [in] */ IInterfacePack* ipack,
115        /* [out] */ AutoPtr<IInterface>& object)
```

```
131        AutoPtr<IObject> iobject;
132        ECode ec = FindImportObject(mType, ipack, iobject);
133        if (SUCCEEDED(ec)) {
134            InterfaceID iid;
135            ipack->GetInterfaceID(iid);
136            object = iobject->Probe(iid);
137            return NOERROR;
138        }
139
140        AutoPtr<IStub> stub;
141        ec = FindExportObject(mType, ipack, stub);
142        if (SUCCEEDED(ec)) {
143            CStub* stubObj = (CStub*)stub.Get();
144            InterfaceID iid;
145            ipack->GetInterfaceID(iid);
146            object = stubObj->GetTarget()->Probe(iid);
147            return NOERROR;
148        }
149
150        AutoPtr<IProxy> proxy;
151        ec = CoCreateProxy(ipack, mType, nullptr, proxy);
152        if (FAILED(ec)) {
153            Logger::E("CDBusChannel", "Unmarshal the interface in ReadInterface failed.");
154            object = nullptr;
155            return ec;
156        }
157
158        RegisterImportObject(mType, ipack, IObject::Probe(proxy));
159
160        InterfaceID iid;
161        ipack->GetInterfaceID(iid);
162        object = proxy->Probe(iid);
163    }
```

GetService中的obj哪里来的？
CoCreateProxy ()

☐ 源码：/como/como/runtime/rpc/comorpc.cpp

```
53   ECode CoCreateProxy(
54       /* [in]  */ IInterfacePack* ipack,
55       /* [in]  */ RPCType type,
56       /* [in]  */ IClassLoader* loader,
57       /* [out] */ AutoPtr<IProxy>& proxy)
58   {
59       AutoPtr<IRPCChannelFactory> factory =
60               type == RPCType::Local ? sLocalFactory : sRemoteFactory;
61       AutoPtr<IRPCChannel> channel;
62       ECode ec = factory->CreateChannel(RPCPeer::Proxy, channel);
63       if (FAILED(ec)) {
64           proxy = nullptr;
65           return ec;
66       }
67       channel->Apply(ipack);
68
69       CoclassID cid;
70       ipack->GetCoclassID(cid);
71       return CProxy::CreateObject(cid, channel, loader, proxy);
72   }
```

- 至此，一个假的C++对象就造出来了，它有虚表：sProxyVtable
- 元数据是通过Loader从本地装进来的

```
1674   ECode CProxy::CreateObject(
1675       /* [in]  */ const CoclassID& cid,
1676       /* [in]  */ IRPCChannel* channel,
1677       /* [in]  */ IClassLoader* loader,
1678       /* [out] */ AutoPtr<IProxy>& proxy)
1679   {
1680       proxy = nullptr;
1681
1682       if (loader == nullptr) {
1683           loader = CBootClassLoader::GetSystemClassLoader();
1684       }
1685
1686       AutoPtr<IMetaCoclass> mc;
1687       loader->LoadCoclass(cid, mc);
```

Obj给Client端源码的"假象"
InterfaceProxy是所有远程过来的Obj的"替身"

RPCTestUnit.cdl

```
como > test > runtime > rpc > component > ☰ Service.cdl
37          version(0.1.0)
38      ]
39      interface IService
40      {
41          TestMethod1(
42              [in] Integer arg1,
43              [out] Integer& result1);
44
45          TestMethod2(
46              [in] Integer arg1,
47              [in] Long arg2,
48              [in] Boolean arg3,
49              [in] Char arg4,
50              [in] Short arg5,
```

RPCTestUnit.h

```
74      INTERFACE_ID(afa187cf-8193-4786-8de7-b576c0bdb162)
75      interface IService
76          : public IInterface
77      {
78          using IInterface::Probe;
79
80          inline static IService* Probe(
81              /* [in] */ IInterface* object)
82          {
83              if (object == nullptr) {
84                  return nullptr;
85              }
86              return (IService*)object->Probe(IID_IService);
87          }
88
89          inline static const InterfaceID& GetInterfaceID()
90          {
91              return IID_IService;
92          }
93
94          virtual como::ECode TestMethod1(
95              /* [in] */ Integer arg1,
96              /* [out] */ Integer& result1) = 0;
97
98          virtual como::ECode TestMethod2(
99              /* [in] */ Integer arg1,
100             /* [in] */ Long arg2,
101             /* [in] */ Boolean arg3,
102             /* [in] */ Char arg4,
103             /* [in] */ Short arg5,
104             /* [in] */ Double arg6,
```

- RPCTestUnit.h是COMO 编译阶段cdlc生成的代码，"夹壁墙"

CProxy.h   *替身*

```
45      class InterfaceProxy
46      {
47      private:
48          struct Registers
```
```
225         HANDLE* mVtable;        // m
226         HANDLE mProxyEntry;     // m
227         Integer mIndex;
228         InterfaceID mIid;
229         IMetaInterface* mTargetMe
230         CProxy* mOwner;
231     };
```

关于替身
Cproxy.cpp

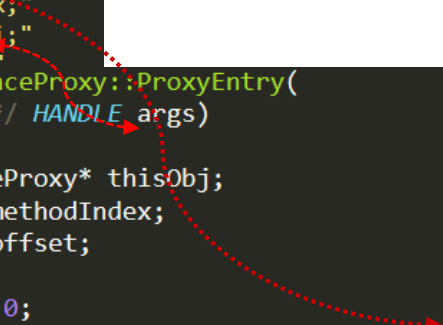### CProxy.h

```
45      class InterfaceProxy
46      {
47      private:
48          struct Registers
```

```
225         HANDLE* mVtable;        // must be the first member
226         HANDLE mProxyEntry;     // must be the second member
227         Integer mIndex;
228         InterfaceID mIid;
229         IMetaInterface* mTargetMetadata;
230         CProxy* mOwner;
231     };
```

```
186     EXTERN_C void __entry();
187
188     __asm__(
189         ".text;"
190         ".align 8;"
191         ".global __entry;"
192         "__entry:"
193         "pushq   %rbp;"
194         "pushq   %rdi;"
195         "subq    $8, %rsp;"
196         "movl    $0xff, (%rsp);"
197         "movq    %rdi, %rax;"
198         "movq    %rsp, %rdi;"
199         "call    *8(%rax);"
200
201
202
203
204
```

- 所有虚函数的入口是__entry，然后__entry再执行mProxyEntry函数指针变量中的函数
- Init_Proxy_Entry()构造了假的虚表
- x64传参ABI：%rdi，第1个参数

```
1438        ECode InterfaceProxy::ProxyEntry(
1439            /* [in] */ HANDLE args)
1440        {
1441            InterfaceProxy* thisObj;
1442            Integer methodIndex;
1443            Integer offset;
1444
1445            offset = 0;
1446            GET_STACK_INTEGER(args, offset, methodIndex);
```

CProxy.cpp

关于ABI
EXTERN_C void __entry();

```
186   EXTERN_C void __entry();
187
188   __asm__(
189       ".text;"
190       ".align 8;"
191       ".global __entry;"
192       "__entry:"
193       "pushq  %rbp;"
194       "pushq  %rdi;"
195       "subq   $8, %rsp;"
196       "movl   $0xff, (%rsp);"
197       "movq   %rdi, %rax;"
198       "movq   %rsp, %rdi;"
199       "call   *8(%rax);"
200       "addq   $8, %rsp;"
201       "popq   %rdi;"
202       "popq   %rbp;"
203       "ret;"
204   );
```

- X64。在调用C++非静态成员函数时使用此约定。基于所使用的编译器和函数是否使用可变参数，有两个主流版本的thiscall。 对于GCC编译器，thiscall几乎与cdecl等同：调用者清理堆栈，参数从右到左传递。差别在于this指针，thiscall会在最后把this指针推入栈中，即相当于在函数原型中是隐式的左数第一个参数。https://zh.wikipedia.org/wiki/X86%E8%B0%83%E7%94%A8%E7%BA%A6%E5%AE%9A

- ARM64 ABI 约定概述 https://docs.microsoft.com/zh-cn/cpp/build/arm64-windows-abi-conventions?view=msvc-160

用InterfaceProxy伪造任意远程访问对象，把对InterfaceProxy虚函数的访问，全转向对InterfaceProxy::ProxyEntry()的访问。按平台上ABI的约定得到函数参数，把参数转向ABI规范的InterfaceProxy::ProxyEntry()需要的参数的样子，然后调用InterfaceProxy::ProxyEntry()。
利用这个伪造的机制，得到你想访问的函数(方法)的信息(methodIndex)，用户访问虚函数时，我们不可能要求用户告诉我们它要访问哪个函数(方法)

服务器端Server -- Stub

服务器端是真正实施COMO构件运行时服务的，也就是程序的逻辑功能是在这里执行的。

Server端注册服务对象

- /como/como/test/runtime/rpc/service/main.cpp

```cpp
17  #include "RPCTestUnit.h"
18  #include <comoapi.h>
19  #include <comosp.h>
20  #include <ServiceManager.h>
21  #include <cstdio>
22  #include <unistd.h>
23
24  using como::test::rpc::CService;
25  using como::test::rpc::IService;
26  using como::test::rpc::IID_IService;
27  using jing::ServiceManager;
28
29  int main(int argv, char** argc)
30  {
31      AutoPtr<IService> srv;
32      CService::New(IID_IService, (IInterface**)&srv);
33
34      ServiceManager::GetInstance()->AddService(String("rpcservice"), srv);
35
36      printf("==== rpc service wait for calling ====\n");
37      while (true) {
38          sleep(5);
39      }
40
41      return 0;
42  }
43
```

- COMO机制New出对象

- RPCTestUnit.cpp **COMO的New与C++的new是不同的**

```cpp
44  // CService
45  ECode CService::New(
46      /* [in] */ const InterfaceID& iid,
47      /* [out] */ como::IInterface** object)
48  {
49      return CoCreateObjectInstance(CID_CService, iid, nullptr, object);
50  }
```

Server端注册服务对象
ServiceManager::AddService

☐   /como/como/servicemanager/lib/linux/ServiceManager.cpp

```
30    ECode ServiceManager::AddService(
31        /* [in] */ const String& name,
32        /* [in] */ IInterface* object)
33    {
34        if (name.IsEmpty() || object == nullptr) {
35            return E_ILLEGAL_ARGUMENT_EXCEPTION;
36        }
37
38        AutoPtr<IInterfacePack> ipack;
39        ECode ec = CoMarshalInterface(object, RPCType::Local, ipack);
```

- CDBusChannelFactory::MarshalInterface() 里，创建了IStub,

- CoCreateStub() → CStub::CreateObject()

```
80        AutoPtr<IStub> stub;
81        ECode ec = FindExportObject(mType, IObject::Probe(object), stub);
82        if (SUCCEEDED(ec)) {
83            CDBusChannel* channel = CDBusChannel::GetStubChannel(stub);
84            pack->SetDBusName(channel->mName);
85            pack->SetCoclassID(((CStub*)stub.Get())->GetTargetCoclassID());
86        }
87        else {
88            IProxy* proxy = IProxy::Probe(object);
89            if (proxy != nullptr) {
90                CDBusChannel* channel = CDBusChannel::GetProxyChannel(proxy);
91                pack->SetDBusName(channel->mName);
92                pack->SetCoclassID(((CProxy*)proxy)->GetTargetCoclassID());
93            }
94            else {
95                ec = CoCreateStub(object, mType, stub);
96                if (FAILED(ec)) {
97                    Logger::E("CDBusChannel", "Marshal interface failed.");
98                    ipack = nullptr;
99                    return ec;
00                }
01                CDBusChannel* channel = CDBusChannel::GetStubChannel(stub);
02                pack->SetDBusName(channel->mName);
03                pack->SetCoclassID(((CStub*)stub.Get())->GetTargetCoclassID());
04                RegisterExportObject(mType, IObject::Probe(object), stub);
05            }
06        }
```

☐   /como/como/como/runtime/rpc/dbus/CDBusChannelFactory.cpp

```
59    ECode CDBusChannelFactory::MarshalInterface(
60        /* [in] */ IInterface* object,
61        /* [out] */ AutoPtr<IInterfacePack>& ipack)
```

Server端服务线程
CDBusChannel/CBinderChannel

☐ /como/como/runtime/rpc/dbus/CDBusChannel.cpp

```
627  ECode CDBusChannel::StartListening(
628      /* [in] */ IStub* stub)
629  {
630      ECode ec = NOERROR;
631      if (mPeer == RPCPeer::Stub) {
632          AutoPtr<ThreadPoolExecutor::Runnable> r = new ServiceRunnable(this, stub);
633          ec = ThreadPoolExecutor::GetInstance()->RunTask(r);
634      }
635
636      {
637          Mutex::AutoLock lock(mLock);
638          while (!mStarted) {
639              mCond.Wait();
640          }
641      }
642      return ec;
643  }
```

- 服务提供体实现IRPCChannel接口
- CDBusChannel::ServiceRunnable::Run()除非资源耗尽，否则一直不退出，

```
33   ECode CDBusChannel::ServiceRunnable::Run()
```

- CDBusChannel::ServiceRunnable::HandleMessage负责执行具体的方法
- IStub *mTarget

```
178          ECode ec = thisObj->mTarget->Invoke(argParcel, resParcel);
```

Server端服务线程
CDBusChannel

□　/como/como/runtime/rpc/dbus/CDBusChannel.cpp

```
73  ECode ThreadPoolExecutor::RunTask(
74      /* [in] */ Runnable* task)
75  {
76      AutoPtr<Worker> w = new Worker(task, this);
77      {
78          Mutex::AutoLock lock(w->mLock);
79
80          pthread_attr_t threadAddr;
81          pthread_attr_init(&threadAddr);
82          pthread_attr_setdetachstate(&threadAddr, PTHREAD_CREATE_DETACHED);
83
84          pthread_t thread;
85          int ret = pthread_create(&thread, &threadAddr, ThreadPoolExecutor::ThreadEntry, (void*)w);
86          if (ret != 0) {
87              return E_RUNTIME_EXCEPTION;
88          }
```

- ThreadPoolExecutor::GetInstance()->RunTask(r)在线程中，把任务w起来 CDBusChannel::ServiceRunnable::Run()
- 直到线程中对mStarted赋值为真，CDBusChannel::StartListening才返回。

```
117  void* ThreadPoolExecutor::ThreadEntry(void* arg)
118  {
119      AutoPtr<Worker> w = (Worker*)arg;
120      {
121          Mutex::AutoLock lock(w->mLock);
122          assert(w->mThread != 0);
123      }
124
125      ECode ec = w->Run();
```

服务管理ServiceManager

服务管理是对所有远程COMO服务对象进行管理的进程。

服务管理ServiceManager

☐ /como/como/servicemanager/exe/linux/main.cpp

```
17  #include "ServiceManager.h"
18  #include <comolog.h>
19  #include <dbus/dbus.h>
20  #include <cstdio>
21
22  using como::Logger;
23  using jing::ServiceManager;
24
25  int main(int argv, char** argc)
26  {
27      DBusError err;
28
29      dbus_error_init(&err);
30
31      DBusConnection* conn = dbus_bus_get_private(DBUS_BUS_SESSION, &err);
32      if (dbus_error_is_set(&err)) {
33          Logger::E("servicemanager", "Connect to bus daemon failed, error is \"%s\".",
34                  err.message);
35          dbus_error_free(&err);
36          return -1;
37      }
38
39      dbus_bus_request_name(conn, ServiceManager::DBUS_NAME,
40              DBUS_NAME_FLAG_REPLACE_EXISTING, &err);
```

- AddService
- GetService
- RemoveService

D-Bus/IBinder RPC机制简介

D-Bus是一个进程间通信及远程过程调用机制，可以让多个不同的计算机程序在同一台电脑上同时进行通信。D-Bus作为freedesktop.org项目的一部分，其设计目的是使Linux桌面环境提供的服务标准化。

OpenBinder是用于进程间通信的系统。它由Be Inc.和Palm，Inc.开发，是Google开发的Android操作系统中现在使用的Binder框架的基础。OpenBinder允许进程显示可由其他线程调用的接口。每个进程都维护一个线程池，该线程池可用于服务此类请求。