**iAuto Service Developer Guide**

窗体顶端

[Search](http://review.iauto.net/develophelp/iAuto-Service-Development)

窗体底端

* [Home](http://review.iauto.net/)
* [All](http://review.iauto.net/pages)
* [Files](http://review.iauto.net/fileview)
* [New](http://review.iauto.net/develophelp/iAuto-Service-Development)
* [Rename](http://review.iauto.net/develophelp/iAuto-Service-Development)
* [Edit](http://review.iauto.net/edit/develophelp/iAuto-Service-Development)
* [History](http://review.iauto.net/history/develophelp/iAuto-Service-Development)

Table of Contents

* [iAuto Service Developer Guide](http://review.iauto.net/develophelp/iAuto-Service-Development#iAuto-Service-Developer-Guide)
  + - * [1.1 Overview](http://review.iauto.net/develophelp/iAuto-Service-Development#1.1-Overview)
      * [1.2 Features](http://review.iauto.net/develophelp/iAuto-Service-Development#1.2-Features)
      * [1.3 Service 框架类](http://review.iauto.net/develophelp/iAuto-Service-Development#1.3-Service-%E6%A1%86%E6%9E%B6%E7%B1%BB)
      * [1.3.1 Inner Framework Service](http://review.iauto.net/develophelp/iAuto-Service-Development#1.3.1-Inner-Framework-Service)
      * [1.3.2 Public IDL Service](http://review.iauto.net/develophelp/iAuto-Service-Development#1.3.2-Public-IDL-Service)
      * [1.4 Service 生命周期](http://review.iauto.net/develophelp/iAuto-Service-Development#1.4-Service-%E7%94%9F%E5%91%BD%E5%91%A8%E6%9C%9F)
      * [1.5 Service 开发流程](http://review.iauto.net/develophelp/iAuto-Service-Development#1.5-Service-%E5%BC%80%E5%8F%91%E6%B5%81%E7%A8%8B)
      * [1.6 Inner Framework Service 代码示例](http://review.iauto.net/develophelp/iAuto-Service-Development#1.6-Inner-Framework-Service-%E4%BB%A3%E7%A0%81%E7%A4%BA%E4%BE%8B)
      * [1.6.1 Stub Base](http://review.iauto.net/develophelp/iAuto-Service-Development#1.6.1-Stub-Base)
      * [1.6.2 Stub](http://review.iauto.net/develophelp/iAuto-Service-Development#1.6.2-Stub)
      * [1.6.3 Stub Replier](http://review.iauto.net/develophelp/iAuto-Service-Development#1.6.3-Stub-Replier)
      * [1.6.4 Service](http://review.iauto.net/develophelp/iAuto-Service-Development#1.6.4-Service)
      * [1.6.5 Service Factory](http://review.iauto.net/develophelp/iAuto-Service-Development#1.6.5-Service-Factory)
      * [1.6.6 Sync Proxy](http://review.iauto.net/develophelp/iAuto-Service-Development#1.6.6-Sync-Proxy)
      * [1.6.7 Async Proxy](http://review.iauto.net/develophelp/iAuto-Service-Development#1.6.7-Async-Proxy)
      * [1.6.8 Replier](http://review.iauto.net/develophelp/iAuto-Service-Development#1.6.8-Replier)
      * [1.6.8 Caller](http://review.iauto.net/develophelp/iAuto-Service-Development#1.6.8-Caller)
      * [1.6.9 Note](http://review.iauto.net/develophelp/iAuto-Service-Development#1.6.9-Note)
      * [1.7 Public IDL Service 代码示例](http://review.iauto.net/develophelp/iAuto-Service-Development#1.7-Public-IDL-Service-%E4%BB%A3%E7%A0%81%E7%A4%BA%E4%BE%8B)
      * [1.7.1 IDL](http://review.iauto.net/develophelp/iAuto-Service-Development#1.7.1-IDL)
      * [1.7.2 Interface (自动生成)](http://review.iauto.net/develophelp/iAuto-Service-Development#1.7.2-Interface-%28%E8%87%AA%E5%8A%A8%E7%94%9F%E6%88%90%29)
      * [1.7.3 Stub Base (自动生成)](http://review.iauto.net/develophelp/iAuto-Service-Development#1.7.3-Stub-Base-%28%E8%87%AA%E5%8A%A8%E7%94%9F%E6%88%90%29)
      * [1.7.4 Stub Replier (自动生成)](http://review.iauto.net/develophelp/iAuto-Service-Development#1.7.4-Stub-Replier--%28%E8%87%AA%E5%8A%A8%E7%94%9F%E6%88%90%29)
      * [1.7.5 Stub](http://review.iauto.net/develophelp/iAuto-Service-Development#1.7.5-Stub)
      * [1.7.6 Service](http://review.iauto.net/develophelp/iAuto-Service-Development#1.7.6-Service)
      * [1.7.7 Service Factory](http://review.iauto.net/develophelp/iAuto-Service-Development#1.7.7-Service-Factory)
      * [1.7.8 Async Proxy (自动生成)](http://review.iauto.net/develophelp/iAuto-Service-Development#1.7.8-Async-Proxy-%28%E8%87%AA%E5%8A%A8%E7%94%9F%E6%88%90%29)
      * [1.7.9 Replier (自动生成)](http://review.iauto.net/develophelp/iAuto-Service-Development#1.7.9-Replier-%28%E8%87%AA%E5%8A%A8%E7%94%9F%E6%88%90%29)
      * [1.7.10 Caller](http://review.iauto.net/develophelp/iAuto-Service-Development#1.7.10-Caller)
      * [1.7.11 Note](http://review.iauto.net/develophelp/iAuto-Service-Development#1.7.11-Note)
      * [1.8 Service间的Event](http://review.iauto.net/develophelp/iAuto-Service-Development#1.8-Service%E9%97%B4%E7%9A%84Event)
      * [1.8.1 定义Event](http://review.iauto.net/develophelp/iAuto-Service-Development#1.8.1-%E5%AE%9A%E4%B9%89Event)
      * [1.8.2 发送Event](http://review.iauto.net/develophelp/iAuto-Service-Development#1.8.2-%E5%8F%91%E9%80%81Event)
      * [1.8.3 接收Event](http://review.iauto.net/develophelp/iAuto-Service-Development#1.8.3-%E6%8E%A5%E6%94%B6Event)
      * [1.9 Service间的大数据通信](http://review.iauto.net/develophelp/iAuto-Service-Development#1.9-Service%E9%97%B4%E7%9A%84%E5%A4%A7%E6%95%B0%E6%8D%AE%E9%80%9A%E4%BF%A1)
      * [1.10 Service广播(Notify)](http://review.iauto.net/develophelp/iAuto-Service-Development#1.10-Service%E5%B9%BF%E6%92%AD%28Notify%29)
      * [1.11 Service调用注意事项](http://review.iauto.net/develophelp/iAuto-Service-Development#1.11-Service%E8%B0%83%E7%94%A8%E6%B3%A8%E6%84%8F%E4%BA%8B%E9%A1%B9)
        + [异步调用的参数必须初始化](http://review.iauto.net/develophelp/iAuto-Service-Development#%E5%BC%82%E6%AD%A5%E8%B0%83%E7%94%A8%E7%9A%84%E5%8F%82%E6%95%B0%E5%BF%85%E9%A1%BB%E5%88%9D%E5%A7%8B%E5%8C%96)
      * [1.12 Service调试辅助工具](http://review.iauto.net/develophelp/iAuto-Service-Development#1.12-Service%E8%B0%83%E8%AF%95%E8%BE%85%E5%8A%A9%E5%B7%A5%E5%85%B7)
      * [1.13 Service状态变化Event](http://review.iauto.net/develophelp/iAuto-Service-Development#1.13-Service%E7%8A%B6%E6%80%81%E5%8F%98%E5%8C%96Event)

**1.1 Overview**

**1.2 Features**

* 兼容Binder Service，提供同设备上的Sync Call能力
* 提供Async Call，并支持跨设备异步访问
* API接口使用IDL定义，Proxy/Stub/Replier代码自动生成
* 支持Service间点到点的自定义数据(Message)传递
* 支持Service间广播、订阅自定义事件(Event)

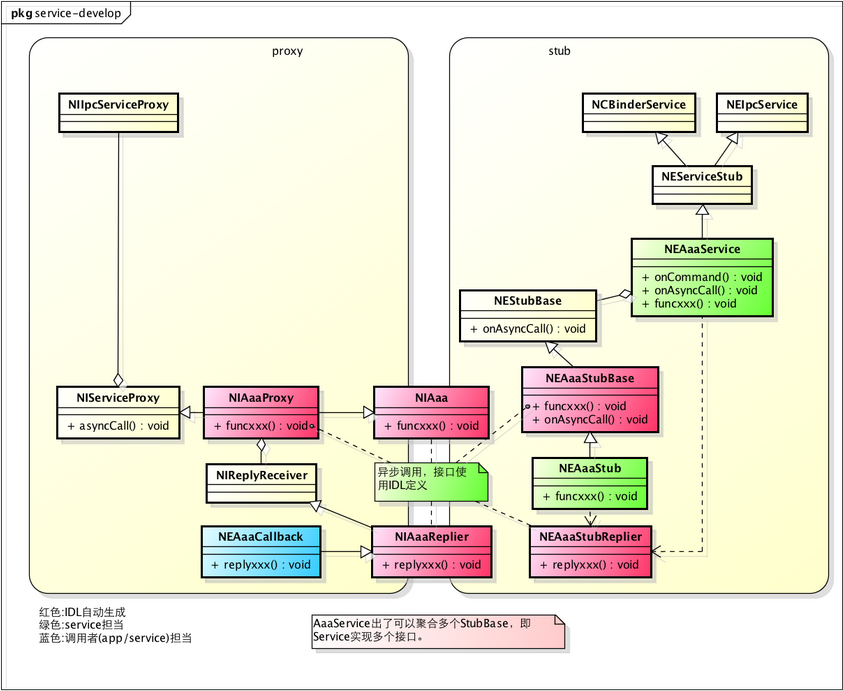
**1.3 Service 框架类**

**1.3.1 Inner Framework Service**

对于不提供API给App，只是Framework内部的Service，不使用IDL

**1.3.2 Public IDL Service**

需要提供API给App，使用IDL自动生成Service框架代码



**1.4 Service 生命周期**

* Service创建后注册到ServiceFactory
* Service的生命周期由ServiceManager统一管理
* 生命周期变化会调用Service的回调函数
  1. 初始化 onInit
  2. 启动 有2种方式
     + 2.1 直接启动 onNormalStart
     + 2.2 Snapshot模式启动 onHibernate
     + 2.3 Snapshot内存恢复 onAwake
  3. 停止 onStop
  4. 销毁 onDeinit

Snapshot内存恢复是指从HIBERNATE状态还原，类似虚拟机的快照还原功能。  
Service crush后重新启动时走normal start流程。

**1.5 Service 开发流程**

1. 申请Service，参见[Service管理](http://review.iauto.net/developguide/iauto2-fw-service)
2. 继承Framework Service基类，实现service类代码
3. 在ServiceFactory中添加Service
4. 在Process中添加ServiceFactory  
   #这一步完成后，service应该可以运行起来，接下来添加对外接口部分
5. 通过定义Service IDL，生成ServiceStub框架代码，将stub实例注册到service中
6. 如果不使用IDL，需要手工书写ServiceStub框架代码，将stub实例注册到service中

**1.6 Inner Framework Service 代码示例**

**1.6.1 Stub Base**

StubBase是Service的接口层，编写对外提供的I/F，以及异步调用的协议解析。

这里示例NEBbbStub只提供减法(sub)的异步调用，并异步返回结果给调用侧。  
输入／输出的参数均用proto包装。

package nutshell.bbb;

option optimize\_for = LITE\_RUNTIME;

message NDBbbSubIn

{

required int32 a = 1;

required int32 b = 2;

}

message NDBbbSubOut

{

required int32 c = 1;

}

#ifndef NEBBBSTUB\_H

#define NEBBBSTUB\_H

#include <evipservice/NEStubBase.h>

#include "NDBbb.pb.h"

#include "bbb/NEBbbReplier.h"

#include "ncore/NCLog.h"

#define NC\_LOG\_TAG "testservice"

namespace nutshell

{

class NEBbbStub : public NEStubBase

{

public:

NEBbbStub(NEIpcService \*owner) : NEStubBase(owner) {}

enum {

METHOD\_SUB = NESERVICE\_METHODID\_INNER,

NOTIFY\_STATE,

};

virtual VOID sub(NEContext &context, nutshell::bbb::NDBbbSubIn &in) = 0;

virtual NC\_BOOL isAcceptable(NEMessage &callMsg);

virtual NC\_BOOL onAsyncCall(NEContext &context, NEMessage &message);

virtual NCString serviceName()

{

return "nutshell.test.service2";

}

protected:

virtual ~NEBbbStub() {}

};

}

#endif

实现部分解析NEMessage，完成proto解析与接口调用。

#include "bbb/NEBbbStub.h"

namespace nutshell

{

NC\_BOOL NEBbbStub::isAcceptable(NEMessage &callMsg)

{

INT32 methodId = callMsg.methodId();

if(methodId < METHOD\_SUB || methodId > METHOD\_SUB) {

return NC\_FALSE;

}

return NC\_TRUE;

}

NC\_BOOL NEBbbStub::onAsyncCall(NEContext &context, NEMessage &message)

{

NCLOGI("NEBbbStub::onAsyncCall(methodId = 0x%x)", message.methodId());

switch (message.methodId()) {

case METHOD\_SUB:

{

nutshell::bbb::NDBbbSubIn in;

if (!in.ParseFromString(message.param())) {

NCLOGI("NEBbbStub::onAsyncCall - ParseFromString error");

break;

}

sub(context, in);

return NC\_TRUE;

}

}

return NC\_FALSE;

}

}

**1.6.2 Stub**

Stub实现Stub Base中定义的I/F，并且可以根据具体模块，在Stub中就回复结果给Proxy侧，  
或者调用绑定的Service接口来实现具体功能。

#ifndef NEBBB\_H

#define NEBBB\_H

#include "bbb/NEBbbStub.h"

#include "ncore/NCLog.h"

#define NC\_LOG\_TAG "testservice"

namespace nutshell

{

class NEBbb : public NEBbbStub

{

public:

NEBbb(NEIpcService \*owner) : NEBbbStub(owner) {}

virtual VOID sub(NEContext &context, nutshell::bbb::NDBbbSubIn &in);

protected:

virtual ~NEBbb() {}

};

}

#endif

#include "evipservice/NCPeerContext.h"

#include "bbb/NEBbb.h"

#include "bbb/NEBbbStubReplier.h"

namespace nutshell

{

VOID NEBbb::sub(NEContext &context, nutshell::bbb::NDBbbSubIn &in)

{

INT32 a = in.a();

INT32 b = in.b();

NCLOGI("NEBbb::sub(%d, %d)", a, b);

NCPeerContextSp pc = context.context<NCPeerContextSp>();

if (pc != NULL) {

NCLOGI("NEBbb::sub context : pid = %d, uid = %d, gid = %d, nickname = '%s'",

pc->pid(), pc->uid(), pc->gid(), pc->nickname().getString());

}

INT32 c = a - b;

// reply result

nutshell::bbb::NDBbbSubOut out;

out.set\_c(c);

NEBbbStubReplier<NEBbb> replier(this);

replier.replySub(context, out);

// notify test

nutshell::bbb::NDBbbState state;

state.set\_info("hello");

replier.notifyState(state);

}

}

**1.6.3 Stub Replier**

Stub Replier是Service侧回复结果给Proxy，或者*主动*通知Proxy的方法类。  
可以在接口Stub实现部分调用，也可以在Stub绑定的Service中调用。  
分别传递各自的class模版来实例化对象。

#ifndef NEBBBSTUBREPLIER\_H

#define NEBBBSTUBREPLIER\_H

#include "evip/NEReplyReceiver.h"

#include "evip/NEMessage.h"

#include "evipservice/NEContext.h"

#include "NDBbb.pb.h"

#include "bbb/NEBbbStub.h"

namespace nutshell

{

template<class T>

class NEBbbStubReplier

{

public:

NEBbbStubReplier(T\* t\_) : t(t\_) {}

virtual ~NEBbbStubReplier() { t = NULL; }

virtual VOID replySub(NEContext &context, nutshell::bbb::NDBbbSubOut out)

{

std::string param;

if (!out.SerializeToString(&param)) {

return;

}

NEMessage msg;

msg.setType(NEMessage::TYPE\_REPLY);

//msg.setService(serviceName());

msg.setMethodId(NEBbbStub::METHOD\_SUB);

msg.setCallId(context.callId());

msg.setParam(param);

if (NC\_FALSE == t->replyAsyncCall(context, msg)) {

}

}

virtual VOID notifyState(nutshell::bbb::NDBbbState out)

{

std::string param;

if (!out.SerializeToString(&param)) {

return;

}

NEMessage msg;

msg.setType(NEMessage::TYPE\_NOTIFY);

//msg.setService(serviceName());

msg.setMethodId(NEBbbStub::NOTIFY\_STATE);

msg.setParam(param);

t->notifyChanged(msg);

}

private:

T \*t;

NEBbbStubReplier();

};

}

#endif

**1.6.4 Service**

Service可以只支持异步调用，也可以同时支持同步异步，  
分别从NEIpcService和NEServiceStub派生。

#ifndef NEBBBSERVICE\_H

#define NEBBBSERVICE\_H

#include <evipservice/NEServiceStub.h>

#include "NDBbb.pb.h"

#include "bbb/NEBbb.h"

#include "ncore/NCLog.h"

#define NC\_LOG\_TAG "testservice"

namespace nutshell

{

class NEBbbService : public NEServiceStub

{

public:

NEBbbService(const NCString name) : NEServiceStub(name) {}

virtual VOID onInit()

{

registerInterface(new NEBbb(this));

}

protected:

virtual ~NEBbbService() {}

};

}

#endif

NEServiceStub 派生的 Service， 如果需要重载 onCommand， 请务必在派生类的default 处理中加上 NEServiceStub::onCommand， 否则会出现 NEServiceStub 机能失效的情况。

**1.6.5 Service Factory**

Service实装完成后，需要注册到Service Factory中，并创建Server Process才能提供服务。  
下述示例代码的Server同时创建了2个Service，除了这里描述的NEBbbService，还有下节说明的IDL自动生成Service。

#include "evipprocess/NCServerProcess.h"

#include "evipprocess/NCServiceFactory.h"

#include "NEAaaService.h"

#include "NEBbbService.h"

#include "ncore/NCLog.h"

#define NC\_LOG\_TAG "testservice"

namespace nutshell

{

class NCTestFactory : public NCServiceFactory

{

public:

virtual ~NCTestFactory() {}

VOID createService()

{

registerService(new NEAaaService("nutshell.test.service0"));

registerService(new NEBbbService("nutshell.test.service2"));

}

};

class NCTestProcess : public NCServerProcess

{

public:

NCTestProcess(INT argc, CHAR\*\* argv) : NCServerProcess(argc, argv) {}

virtual ~NCTestProcess() {}

VOID OnInitialize() {}

VOID OnStart() {}

VOID OnStop() {}

VOID OnDeinitialize() {}

};

}

using namespace nutshell;

int main(int argc, char \*argv[])

{

NCLOGI("main");

ncsp<NCTestFactory>::sp factory = new NCTestFactory;

NCTestProcess process(argc, argv);

NCLOGI("process initialize");

process.initialize(factory);

NCLOGI("process start");

process.start();

NCLOGI("process enterloop");

process.enterloop();

NCLOGI("process stop");

process.stop();

NCLOGI("process deinitialize");

process.deinitialize();

factory = 0;

return 0;

}

**1.6.6 Sync Proxy**

#include <ncservice/NCBinderServiceProxy.h>

class NCXXXProxy

{

public:

virtual ~NCXXXProxy() {}

INT32 methodAAA(…) // methods…

{

NCParcel param;

NCParcel reply;

param.write(…);

NCBinderServiceProxy proxy("nutshell.xxx.xxx");

proxy.callFunction(METHOD\_AAA, param, reply);

reply.read(…); // check reply values

return 0;

}

}

Note: 示例NEBbbService并未实现同步调用，如要对应，需要重写onCommand()函数。 从param的NCParcel中获取到调用参数传递给sub()函数，结果可以通过parcel同步返回给调用侧。 与iAuto1.0的BinderService实现方法一致。

**1.6.7 Async Proxy**

#ifndef NEBBBPROXY\_H

#define NEBBBPROXY\_H

#include <evip/NEServiceProxy.h>

#include "NDBbb.pb.h"

#include "bbb/NEBbbReplier.h"

#include "ncore/NCLog.h"

#define NC\_LOG\_TAG "testservice"

namespace nutshell

{

class NEBbbProxy : public NEServiceProxy

{

public:

NEBbbProxy(ncsp<NEReplyReceiver>::sp callback = NULL)

: NEServiceProxy(callback) {}

enum {

METHOD\_SUB = NESERVICE\_METHODID\_INNER,

NOTIFY\_STATE, // notify from service

};

virtual UINT64 sub(nutshell::bbb::NDBbbSubIn &in);

virtual VOID onReceiveMessage(NEMessage &message);

virtual NCString serviceName()

{

return "nutshell.test.service2";

}

protected:

virtual ~NEBbbProxy() {}

};

}

#endif

#include "bbb/NEBbbProxy.h"

namespace nutshell

{

UINT64 NEBbbProxy::sub(nutshell::bbb::NDBbbSubIn &in)

{

NCLOGI("NEBbbProxy::sub(%d, %d)", in.a(), in.b());

static uint64\_t callId = 1;

std::string param;

if (!in.SerializeToString(&param)) {

return 0;

}

NEMessage message;

message.setType(NEMessage::TYPE\_REQUEST);

message.setMethodId(METHOD\_SUB);

message.setService(remoteServiceName());

message.setParam(param);

message.setCallId(callId);

if (!asyncCall(message)) {

return 0;

}

return callId++;

}

VOID NEBbbProxy::onReceiveMessage(NEMessage &message)

{

NCLOGI("NEBbbProxy::onReceiveMessage(methodId = 0x%x)", message.methodId());

switch (message.methodId()) {

case METHOD\_SUB:

{

nutshell::bbb::NDBbbSubOut out;

if (!out.ParseFromString(message.param())) {

NCLOGI("NEBbbProxy::onReceiveMessage - ParseFromString error");

break;

}

NEBbbReplier \*replier = dynamic\_cast<NEBbbReplier\*>(m\_replyReceiver.get());

if (replier) {

replier->replySub(out, message.callId());

}

break;

}

case NOTIFY\_STATE:

{

nutshell::bbb::NDBbbState out;

if (!out.ParseFromString(message.param())) {

NCLOGI("NEBbbProxy::onReceiveMessage - ParseFromString error");

break;

}

NEBbbReplier \*replier = dynamic\_cast<NEBbbReplier\*>(m\_replyReceiver.get());

if (replier) {

replier->notifyState(out, message.callId());

}

break;

}

}

}

}

**1.6.8 Replier**

#ifndef NEBBBREPLIER\_H

#define NEBBBREPLIER\_H

#include "evip/NEReplyReceiver.h"

namespace nutshell

{

class NEBbbReplier : public NEReplyReceiver

{

public:

virtual VOID replySub(nutshell::bbb::NDBbbSubOut &out, UINT64 callId) = 0;

virtual VOID notifyState(nutshell::bbb::NDBbbState &out, UINT64 callId) = 0;

protected:

virtual ~NEBbbReplier() {}

};

}

#endif

**1.6.8 Caller**

派生出一个replier，并且重写replySub，replyState等函数。  
调用Proxy->setReplyReceiver()设置到Proxy中即可接收回复／通知。

class NETestBbbReplier : public NEBbbReplier

{

public:

virtual ~NETestBbbReplier() {}

virtual VOID replySub(nutshell::bbb::NDBbbSubOut &out, UINT64 callId)

{

NCLOGI("NETestBbbReplier::replySub(%lld, %d)", callId, out.c());

}

virtual VOID notifyState(nutshell::bbb::NDBbbState &out, UINT64 callId)

{

NCLOGI("NETestBbbReplier::notifyState(%s)", callId, out.info().c\_str());

}

};

下列代码描述了如何调用Service的函数。 执行下述代码后，通过logcat可以发现replySub()函数返回了“100 - 200”的结果“－100”

// Service -> service2 : Bbb

do {

NCString name = "nutshell.test.service2";

ncsp<NEIpcServiceProxy>::sp ipc = NEIpcServiceHelper::findService(name.getString(), -1, "bbb nickname");

if (ipc == NULL) {

NCLOGI("%s ipc is NULL", name.getString());

break;

}

ncsp<NEBbbProxy>::sp proxy = new NEBbbProxy();

ncsp<NEReplyReceiver>::sp replier = new NETestBbbReplier();

proxy->setReplyReceiver(replier);

// bind ipc to proxy

if (!proxy->bindService(ipc)) {

NCLOGI("%s bind ipc to proxy error", name.getString());

break;

}

proxy->registerNotify(NEBbbProxy::NOTIFY\_STATE);

nutshell::bbb::NDBbbSubIn in;

in.set\_a(100);

in.set\_b(200);

proxy->sub(in);

sleep(1);

proxy->unbindService();

proxy = 0;

} while (0);

**1.6.9 Note**

关于Service的名字，示例test代码中写的字符串，正式Service申请完成后，Service名字有字符串常量定义  
1. platform/base/AppFramework/include/system/PfServiceDefine.h  
2. framework/base/AppFramework/include/system/FwServiceDefine.h

sync call的METHODID的取值范围在下述文件中也有定义  
platform/base/AppFramework/include/evip/NEServiceDefine.h  
从NESERVICE\_FIRST\_CALL\_TRANSACTION开始分配

async call的METHOD的取值范围，在面向APP和Service的头文件中均有定义：

面向APP定义在NIMessage.h

#define NISERVICE\_METHODID\_FIRST 0

面向Service定义在NEMessage.h

#define NESERVICE\_METHODID\_FIRST 0

#define NESERVICE\_METHODID\_INNER 0x80000000

对于Inner Service请从NESERVICE\_METHODID\_INNER开始定义METHODID。

**1.7 Public IDL Service 代码示例**

**1.7.1 IDL**

IDL文件使用eclipse环境编写。  
详见：

* [IDL代码生成工具介绍](http://review.iauto.net/develophelp/iAuto-IDL-generator-new)
* [IDL开发基本规则](http://review.iauto.net/develophelp/iAuto-IDL-rule)

这里使用IDL工具自动生成代码的示例NIAaa。

package nutshell.aaa

interface aaa {

method add {

in {

Int32 a

Int32 b

}

out {

Int32 c

}

}

}

**1.7.2 Interface (自动生成)**

接口代码类似如下形式，因为是面向APP的接口，可以注意到类名有NI的prefix。  
且数据类型均为系统标准或STL类型。

#ifndef NIAAA\_H

#define NIAAA\_H

#include <stdint.h>

#include <nievip/NIServiceInterface.h>

namespace nutshell

{

class NIAaa : public NIServiceInterface

{

public:

/\*\*

\* calculates "a + b", reply result async.

\* @note return value "uint64\_t" is callId.

\*/

virtual uint64\_t add(int32\_t a, int32\_t b) = 0;

protected:

virtual ~NIAaa() {}

};

}

#endif

**1.7.3 Stub Base (自动生成)**

Stub Base的接口部分类似Interface，但每个函数都多一个context参数。  
context纪录了哪个proxy，以及哪一次（callId）调用。

NEContext存储了NEPeerContext的弱引用，可以通过如下代码获取需要的信息，如通道名字等。

NCPeerContextSp pc = context.context<NCPeerContextSp>();

Service（接口）实现者还可以从NCPeerContext派生自己的class，  
通过重载NEIpcService::createContext()函数返回派生class的new对象来扩展peer context的内容。

#ifndef NEAAASTUBBASE\_H

#define NEAAASTUBBASE\_H

#include <evipservice/NEStubBase.h>

namespace nutshell

{

class NEIpcService;

class NEAaaStubBase : public NEStubBase

{

public:

NEAaaStubBase(NEIpcService \*owner)

: NEStubBase(owner) {}

enum {

METHOD\_ADD = NESERVICE\_METHODID\_FIRST,

};

virtual VOID add(NEContext &context, INT32 a, INT32 b) = 0;

virtual NC\_BOOL isAcceptable(NEMessage &callMsg);

virtual NC\_BOOL onAsyncCall(NEContext &context, NEMessage &message);

virtual NCString serviceName();

protected:

virtual ~NEAaaStubBase() {}

};

}

#endif

**1.7.4 Stub Replier (自动生成)**

Stub Replier是一个模版类，用户在Stub或者Service中需要回复结果给proxy侧时，  
传递Stub/Service类名实例化临时对象的方式使用。

#ifndef NEAAASTUBREPLIER\_H

#define NEAAASTUBREPLIER\_H

#include "proto/NDAaa.pb.h"

#include "ncore/NCLog.h"

#define NC\_LOG\_TAG "testservice"

namespace nutshell

{

template<class T>

class NEAaaStubReplier

{

public:

NEAaaStubReplier(T \*t) : m\_t(t) {}

virtual ~NEAaaStubReplier() {}

NC\_BOOL replyAdd(NEContext &context, INT32 c)

{

if (NULL == m\_t) {

return NC\_FALSE;

}

std::string param;

nutshell::aaa::NDAaaAddOut out;

out.set\_c(c);

if (!out.SerializeToString(&param)) {

return NC\_FALSE;

}

NEMessage msg;

msg.setType(NEMessage::TYPE\_REPLY);

msg.setService(m\_t->serviceName());

msg.setMethodId(NEAaaStubBase::METHOD\_ADD);

msg.setCallId(context.callId());

msg.setParam(param);

if (NC\_FALSE == m\_t->replyAsyncCall(context, msg)) {

return NC\_FALSE;

}

return NC\_TRUE;

}

private:

T \*m\_t;

};

}

#endif

使用代码类似：

INT32 c = a + b;

NEAaaStubReplier<NEAaaStub> replier(this);

replier.replyAdd(context, c);

**1.7.5 Stub**

自动生成代码Stub Base封装了接口实现层，具体每个接口函数的实现，需要Service(接口)实现者派生出自己的Stub。

#ifndef NEAAASTUB\_H

#define NEAAASTUB\_H

#include "aaa/stub/NEAaaStubBase.h"

#include "aaa/stub/NEAaaStubReplier.h"

#include "ncore/NCLog.h"

#define NC\_LOG\_TAG "testservice"

namespace nutshell

{

class NEAaaStub : public NEAaaStubBase

{

public:

NEAaaStub(NEIpcService \*owner)

: NEAaaStubBase(owner) {}

virtual ~NEAaaStub() {}

virtual VOID add(NEContext &context, INT32 a, INT32 b)

{

NCLOGI("NEAaaStub::add(%d, %d)", a, b);

INT32 c = a + b;

NEAaaStubReplier<NEAaaStub> replier(this);

replier.replyAdd(context, c);

}

};

}

#endif

本示例中不仅实现了接口，而且立即发送结果给调用侧。  
如果是耗时操作等，不能立即返回的操作，可以保存context并交给内部的工作线程来实现。  
或者调用接口绑定的Service提供的函数来实现功能。

绑定的Service可以通过

dynamic\_cast<NEAaaService\*>(m\_owner)

来获取。

Note:  
其中每个函数携带的context，在远端proxy连接和断开时分别会通过onBind()/onUnbind()回调来通知。  
即每个proxy连接对应到service侧的调用时序如下：  
onBind() -> funcXxx() ... funcXxx() -> onUnbind()

需要广播给每个连接的proxy侧时，可以自行通过onBind()/onUnbind()来维护一张context列表。

**1.7.6 Service**

只是接口的Stub部分还不能提供服务，需要创建Service实体并注册到Service Factory才能工作。  
一个Service可以注册一个或多个接口(Stub)。

如下代码展示了如何注册一个Stub到Service。  
其中Service类似Inner Service，可以选择只是异步或同步异步都有。

#ifndef NEAAASERVICE\_H

#define NEAAASERVICE\_H

#include <evipservice/NEServiceStub.h>

#include <NEAaaStub.h>

namespace nutshell

{

class NEAaaService : public NEServiceStub

{

public:

NEAaaService(const NCString name) : NEServiceStub(name) {}

virtual VOID onInit()

{

registerInterface(new NEAaaStub(this));

}

protected:

virtual ~NEAaaService() {}

};

}

#endif

注意registerInterface函数只能在onInit()当中调用，且无需unregister。（在onDeinit前自动unregister）

NEServiceStub 派生的 Service， 如果需要重载 onCommand， 请务必在派生类的default 处理中加上 NEServiceStub::onCommand， 否则会出现 NEServiceStub 机能失效的情况。

**1.7.7 Service Factory**

与Inner Service相同。

**1.7.8 Async Proxy (自动生成)**

#ifndef NIAAAPROXY\_H

#define NIAAAPROXY\_H

#include <nievip/NIServiceProxy.h>

#include "aaa/interface/NIAaa.h"

namespace nutshell

{

class NIAaaProxy : public NIServiceProxy, public NIAaa

{

public:

NIAaaProxy() {}

virtual ~NIAaaProxy() {}

enum {

METHOD\_ADD = NISERVICE\_METHODID\_FIRST,

};

virtual uint64\_t add(int32\_t a, int32\_t b);

virtual void onReceiveMessage(NIMessage &message);

virtual std::string serviceName();

};

}

#endif

**1.7.9 Replier (自动生成)**

与调用接口相对，回复部分也生成接口class

#ifndef NIAAAREPLIER\_H

#define NIAAAREPLIER\_H

#include <nievip/NIReplyReceiver.h>

namespace nutshell

{

class NIAaaReplier : public NIReplyReceiver

{

public:

virtual void replyAdd(uint64\_t callId, int32\_t c) = 0;

protected:

virtual ~NIAaaReplier() {}

};

}

#endif

**1.7.10 Caller**

调用方式与Inner Service类似，但各种class名字都变为NI前缀开头。

do {

std::string name = "nutshell.test.service0";

NISP<NIIpcServiceProxy> ipc = NIIpcServiceHelper::findService(name.c\_str(), -1);

if (ipc == NULL) {

NCLOGI("%s ipc is NULL", name.c\_str());

break;

}

NISP<NIAaaProxy> proxy = new NIAaaProxy();

NISP<NIReplyReceiver> replier = new NETestAaaReplier();

proxy->setReplyReceiver(replier);

// bind ipc to proxy

if (!proxy->bindService(ipc)) {

NCLOGI("%s bind ipc to proxy error", name.c\_str());

break;

}

proxy->add(1, 2);

sleep(1);

proxy->setReplyReceiver(NULL);

proxy->unbindService();

replier = NULL;

proxy = NULL;

} while (0);

**1.7.11 Note**

包括上述Inner service的示例代码均在git的如下位置:  
platform/base/AppFramework/evipservice/testservice

**1.8 Service间的Event**

**1.8.1 定义Event**

各个Service需要确定本模块有哪些Event，以及每个Event携带的数据类型，并更新到Wiki。

Service的名称由两部分构成，eventspace和eventId。  
eventspace使用service name+.+sub module的形式。  
eventId是具体的event名称(NCString)。

Service携带的数据在2个INT32以内的时候，可以直接使用Event的setArg1()/setArg2()。  
对于复杂的数据使用protobuffer，proto名字以ND开头，package名称使用nutshell.<模块名>。  
proto序列化后的string调用Event的setExtraData()设置。

**1.8.2 发送Event**

Event对象不能自己创建，需要通过NCEvSysContext的obtainEvent()函数来获取。

NCEvSysContext context;

NCEvent \*event = context.obtainEvent("service\_name.sub\_module", "xxx\_event");

NCEvMessageIF \*eventSender = queryEventSender();

if (event && eventSender) {

if (NC\_FALSE == eventSender->sendEvent(event)) {

NCLOGE("send event error");

}

} else {

NCLOGE("create event error");

}

context.releaseEvent(event);

**1.8.3 接收Event**

Event的接收者需要派生自己的EventHandler，并配合NCRunnableThread使用。

class NCXXXEventHandler : public NCEventHandler

{

public:

NCXXXEventHandler(const NCRunnableLooper& looper);

void handleEvent(const NCEvent& event);

};

m\_thread = new NCRunnableThread(); // event handler thread

m\_thread->startThread(“thread name”); // make the thread alive

NCEvSysContext evSysContext;

m\_pcEventHandler= new NCXXXEventHandler(m\_thread->looper()); // make event handler

std::list<NCString> filter; // decide which event to receive

// filter中需要填充希望接收的eventSpace和eventId，可以多个。

filter.push\_back(evSysContext.buildEventFullName("service\_name.sub\_module", "xxx\_event"));

evSysContext.registerHandler(m\_pcEventHandler, filter); // publish the event handler

在handleEvent当中对接收到的event进行处理，或触发其他工作线程等。

m\_pcEventHandler必须是NCEventHandlerStrongPointer,  
或者自定义ncsp<NCXXXEventHandler>::sp的强指针类型，不能用裸指针(NCEventHandler\*)。

**1.9 Service间的大数据通信**

大数据通信时，采用独立的Data通道方式来实现。  
Data通道是C/S模型，需要一个Server端start通道，另外N个Client端bind通道。

* Data Channel Server

NCEvSysContext context;

NCEvDataIf \*dataIf = context.startDataChannel(addr, handler);

dataIf可以用来发送数据给client端，发送时需要指定client端的地址peer。

* Data Channel Client

NCEvSysContext context;

NCEvDataIf \*dataIf = context.bindDataChannel(addr, handler);

* addr

上述函数用到的addr是通道的地址，为了确保唯一性，推荐采用如下方式：  
ipc://dev/ncmq/[modelname].[servicename]\_da

* handler

上述函数用到的handler是从通道接收数据的callback，需要与NCRunnableThread配合使用。

class NEXxxDataHandler : public NCDataHandler

{

public:

/\*\*

\* constructor

\*\*/

NEXxxDataHandler(const NCRunnableLooper &looper)

: NCDataHandler(looper) {}

/\*\*

\* destructor

\*\*/

virtual ~NEXxxDataHandler() {}

/\*\*

\* handler data

\*\*/

VOID handleData(NCString& peer, NCData& data)

{

}

VOID onBind(NCString &self, NCString &peer)

{

}

VOID onUnbind(NCString &self, NCString &peer)

{

}

}

m\_runnableThread = new NCRunnableThread;

m\_runnableThread->startThread(NCString("thread name") + name);

m\_dataHandler = new NEXxxDataHandler(this, m\_runnableThread->looper());

m\_dataHandler必须是NCDataHandlerStrongHolder,  
或者自定义ncsp<NCXXXDataHandler>::sp的强指针类型，不能用裸指针(NCDataHandler\*)。

**1.10 Service广播(Notify)**

Service可以发广播给所有监听消息的Proxy，Proxy通过调用registerNotify()函数注册希望接收的消息。  
作为扩展，Service侧也提供给指定的一个Proxy发点对点的消息。

IDL中的broadcast关键字会生成2个notifyXXX的函数，其中一个带有NEContext参数。  
手写Inner Service时按照IDL代码的结构编码即可。

缺省情况下，Proxy不接收任何notify。

Inner Service的Proxy侧使用registerNotify(NotifyXXXId)函数注册。  
IDL Service的Proxy会自动生成registerXxx()/unregisterXxx()，调用对应的函数即可完成注册／反注册。

**1.11 Service调用注意事项**

**异步调用的参数必须初始化**

函数的参数不论是原生类型，struct或者外部导入的proto，都必须初始化后再传给I/F调用。  
没有初始化的变量，其值可能随机，超出取值范围会导致protocolbuf序列化时失败，导致系统abort!  
同样，外部导入的proto数据，各个required字段均需设置后才能使用。

**1.12 Service调试辅助工具**

[Service调试辅助工具](http://review.iauto.net/develophelp/iAuto-Service-Tools)

**1.13 Service状态变化Event**

Service状态变化时，系统会发送Event  
需要检测其他Service状态的时候，可以通过注册event filter来检测Service的状态变化。

Event的接收方法参考章节 [1.8.3 接收Event](http://review.iauto.net/develophelp/iAuto-Service-Development#1.8.3-%E6%8E%A5%E6%94%B6Event) Event Space和service状态均定义在文件：servicemanager/NCServiceManagerDefine.h

Event space固定为：

#define EVIP\_SVRMGR\_SERVICE\_STATUS\_SPACE "nutshell.servicemanager.status"

如果希望监听所有Service的状态，可以不设置Event Identity，即匹配所有Event Identity。  
如果希望监听指定Service的状态，设置Service的名字为Identity，且多次注册Filter来监听多个Service的状态。  
结合1.8.3章节的示例代码，可以如下设置filter:

// 监听所有Service状态

filter.push\_back(evSysContext.buildEventFullName(EVIP\_SVRMGR\_SERVICE\_STATUS\_SPACE));

// 监听名字为”nutshell.xxx.yyy”的Service状态

filter.push\_back(evSysContext.buildEventFullName(EVIP\_SVRMGR\_SERVICE\_STATUS\_SPACE, "nutshell.xxx.yyy"));

收到Event后，通过查看event.arg1()的值来判断Service的状态，定义为如下的宏：

///< used by service to set status to service manager

#define EVIP\_SERVICE\_STATUS\_UNKNOWN 0 // initialize value

#define EVIP\_SERVICE\_STATUS\_PUBLISHED 1 // called in publish

#define EVIP\_SERVICE\_STATUS\_HIBERNATE\_READY 2 // after onHibernate

#define EVIP\_SERVICE\_STATUS\_READY 3 // after onAwake, or after onResume or after onNormalStart

#define EVIP\_SERVICE\_STATUS\_SUSPENDED 4 // after onSuspend

#define EVIP\_SERVICE\_STATUS\_SHUTDOWN 5 // after onStop

#define EVIP\_SERVICE\_STATUS\_CANCELED 6 // called in cancel

通常情况下，状态变成EVIP\_SERVICE\_STATUS\_READY时意味着Service可以接收请求。

Last edited by **caoxuejun**, 2015-05-11 19:33:30

[Delete this Page](http://review.iauto.net/develophelp/iAuto-Service-Development)

窗体顶端

窗体底端