# **Ice开发过程示例Demo** -- 异步双向通信

## 1. 异步双向通信介绍

ICE中的异步程序设计方法（AMD，AMI）：

AMI：Asynchronous Method Invocation 异步方法调用

AMD：Asynchronous Method Dispatch 异步方法派发

同步模型与异步模型的区别

同步模型中发出调用的线程会阻塞到操作返回。异步模型则正好相反，发出调用的线程不会阻塞的等待调用结果的返回。

AMI：

使用AMI发出远地调用，在Ice run time等待答复的同时，发出调用的线程不会阻塞，相反，发出调用的线程可以继续进行各种活动，当答复最终到达时，Ice run time 会通知应用，通过回调发给应用对象通知。

AMD：

使用AMD时，服务器可以接受一个请求，然后挂起其处理，以尽快释放分派线程。当处理恢复、结果以得出时，服务器要使用Ice run time提供的回调对象，显式地发送响应。

AMD操作通常会把请求数据（也就是，回调对象和操作参数）放入队列，供应用的某个线程（或线程池）随后处理用。这样，服务器就使分派线程的使用率降到了最低限度，能够高效地支持数千并发客户。

默认情况下，一个服务器在同一时刻所能支持的同步请求数受到Ice run time的服务器线程池的尺寸限制。如果所有线程都在忙于分派长时间运行的操作，那么就没有线程可用于处理新的请求，客户就会体验到不可接受的无响应状态（Ice run time的服务器线程池尺寸受限于主机的CPU数）。

而ICE的异步方法是服务器收到请求时并不马上执行具体工作，而是把执行这项工作所需的参数以及回调类保存到一个地方（比如队列）后就返回。而另外的线程（或线程池）负责取出保存的参数并执行之，执行结束后使用回调类通知客户端工作已完成（或异常）。

异步双向通信是指客户端和服务端使用异步方式进行双向通信。

Ice自3.4以后，提供了一套全新的AMI（异步调用方式）的API，新的API已不需要在slice文件中标记[“ami”]这样的元数据，slice工具会自动的生成同步调用和异步调用的stub代码。begin\_\*\*\*Method、end\_\*\*\*Method即生成的异步调用方法。

客户端和服务端使用slice工具生成的异步调用方法来相互请求服务。首先由客户端发送本地请求，服务端接收请求后保存客户端对象信息和参数数据，然后使用服务端实现的、客户端请求的接口服务来处理这些参数数据，处理完毕后由服务端决定是否保存的客户端对象回调传送信息给客户端。

## 2. Ice安装

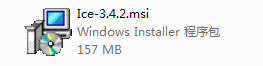
Ice（Internet Communications Engine），是ZeroC开发的一个面向对象的中间件平台。它提供了面向对象的远程过程调用、网格计算和发布/订阅功能，并有基于GPL的双许可协议和一个私有许可协议。它支持Linux、Solaris、Windows和Mac OS X等最主要的操作系统，和C++、Java、.NET语言（如C#或Visual Basic）、Objective-C、Python、PHP和Ruby等语言。如它的名字所表明，该中间件可以被用于应用程序，而不需要使用HTTP协议，并且有能力穿越防火墙（这一点不同于当时的其它中间件）。

下面主要介绍一下怎样使用C#编写基于Ice的网络应用程序。

开发环境： Windows 7, Visual Studio 2010

先到http://www.zeroc.com/download.html下载Ice的安装包，Windows下用msi的。跨平台通信时，要使用相同的ice版本,为工程应用中ICE版本一致起见，此处使用的

是“Ice-3.4.2.msi” 。



安装完毕后，将安装目录下的bin目录加入环境变量的Path路径，然后就可以在VS.NET中开发Ice应用了。

注意：添加路径

在“计算机”-右键点击-“属性”-“高级系统设置”-“环境变量”-编辑“path”

或添加环境变量ICE\_HOME,并将%ICE\_HOME%\bin添加到Path中：

[](http://images.cnblogs.com/cnblogs_com/SGSoft/WindowsLiveWriter/ICEHelloWord_1411B/image%7b0%7d%5b19%5d.png)

[](http://images.cnblogs.com/cnblogs_com/SGSoft/WindowsLiveWriter/ICEHelloWord_1411B/image%7b0%7d%5b22%5d.png)

## 3. 编写Slice定义文件

每个Ice对象都有一个接口，该接口具有一些操作。接口、操作，还有在客户及服务器间交换的数据类型，都是用Slice语言定义的。Slice是Ice规范语言，允许你以一种独立于特定编程语言（比如C++或Java）的方式定义客户-服务器的合约。

Slice定义由一个编译器编译成特定编程语言的API，也就是说，与你所定义的接口和类型对应的那一部分API，会由生成的代码组成。

编写Slice定义文件，定义客户端和服务端要使用的接口方法，文件保存为Callback.ice，其代码如下：

module ICEInterface

{

class Info

{

string strMobile; // 设备名字

string strMsg; // message

string strPwd;

string strUser;

};

exception RequestException //自定义异常

{

};

interface CallbackReceiver //客户端服务接口

{

void callback(int ReMsg);

};

interface CallbackSender //服务端服务接口

{

void initiateCallback(CallbackReceiver\* proxy, Info inf)

throws RequestException;

};

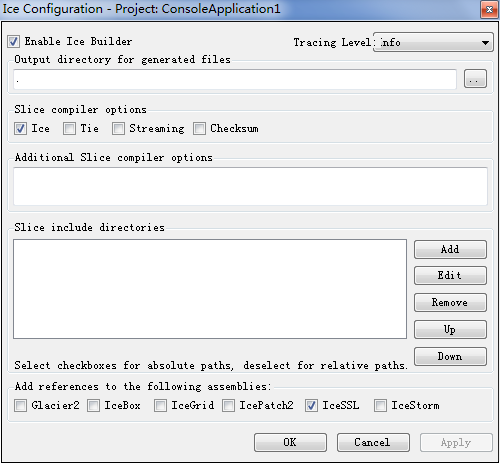
};

此处定义了一个类(class Info)用来传递设备信息，后面定义了两个接口：CallbackReceiver和CallbackSender，这两个接口将分别在客户端和服务端被继承，并重写其内部定义的方法为项目需要的业务逻辑，其中“CallbackReceiver\* proxy”表示客户端提供的回调接口服务对象。接口内部操作包含的参数可以使用其他符合slice规范的类型，便于开发人员根据实际情况定义符合要求的数据结构和操作方法。

## 4. 将Slice定义文件加入到项目

新建两个C#空项目：Server和Client，并添加对Ice.dll和IceSSL.dll的引用。将上面的Callback.ice文件分别添加到两个项目中。

在Ice 3.3.0之后提供了Visual Studio Plugin，可以在工程文件上点右键，会出现“Ice Configuration”菜单项,点此菜单项后会弹出一个配置对话框，勾选“Enable Ice Builder”，更改输出文件目录为当前工程所在目录，并选择“Slice compiler options”下的“Ice”，然后点击应用，就会自动跟踪工程中的.ice文件，自动生成对应的stub文件（当我们在项目中修改\*.ice文件时，系统会自动更新由\*.ice生成的\*.cs文件，而另外一种方法时使用命令“slice2cs \*.cs”去生成cs文件，使用这种方法，每修改一次需要输入一次命令，不便于文件的实时更新）。



## 5. 服务端编写CallbackSenderI接口服务

写在服务端的接口服务，将实例化绑定到一个指定标识的适配器中，当客户端要调用这个服务时，ICE将搜索指定的远程或本地服务端是否存在这个服务，若存在此服务，则客户端将需要的数据发送给对应服务端的接口，服务端将这些数据传递给服务方法进行处理。

源代码：

public sealed class CallbackSenderI : CallbackSenderDisp\_

{

/// <summary>

/// 服务端提供的接口服务，处理客户端发送过来的信息，并回传服务端响应

/// </summary>

/// <param name="proxy">客户端回调服务代理</param>

/// <param name="inf">客户端传送的信息</param>

/// <param name="current"></param>

public override void initiateCallback(CallbackReceiverPrx proxy, Info inf, Ice.Current current)

{

try

{

// 业务处理

Console.Out.WriteLine("initiating callback--server");

Console.Out.WriteLine(inf.strMobile);

Console.Out.WriteLine(inf.strMsg);

Console.Out.WriteLine(inf.strUser);

Console.Out.WriteLine(inf.strPwd);

proxy.begin\_callback(-1); // 异步回调

}

catch (System.Exception ex)

{

Console.Error.WriteLine(ex);

LogHelper.WriteTextLog("initiateCallback", "启用失败！原因为" + ex.Message, DateTime.Now);

}

}

}

服务端创建适配器，并将CallbackSenderI服务接口添加到适配器中，以提供客户端能请求CallbackSenderI中重写的服务方法。

// 此处的 “Callback.Server” 存在于 config.server 中(按照config.server文件中的配置来设置 Adapter)

Ice.ObjectAdapter adapter = communicator().createObjectAdapter("Callback.Server");

// 服务端添加 CallbackSenderI 接口服务，并指定服务对应的标识

adapter.add(new CallbackSenderI(), communicator().stringToIdentity("callbackSender"));

// 启用新建的适配器

adapter.activate();

// 监听适配器直到关闭通信器

communicator().waitForShutdown();

这里将CallbackSenderI 接口服务添加到了服务端的适配器中，当有客户端请求此服务时，将执行CallbackSenderI接口中实现的业务服务方法。

## 6. 客户端编写CallbackReceiverI接口服务

客户端实现CallbackReceiverI接口服务，就此而言，此时客户端相当于一个服务器，提供一个接口服务供服务端回调使用。

源代码：

public sealed class CallbackReceiverI : CallbackReceiverDisp\_

{

/// <summary>

/// 客户端接口服务，对服务端回调传输过来的信息进行处理

/// </summary>

/// <param name="ReMsg">服务端回调传递的信息</param>

/// <param name="current"></param>

public override void callback(int ReMsg, Ice.Current current)

{

System.Console.Out.WriteLine("received callback -- " + ReMsg);

}

}

ClientApp.cs中声明服务端的服务代理目标CallbackSenderPrx，并尝试与之能否建立连接：

// sender 调用服务端的 CallbackSenderPrx 服务

CallbackSenderPrx sender = CallbackSenderPrxHelper.checkedCast(

communicator().propertyToProxy("CallbackSender.Proxy").

ice\_twoway().ice\_timeout(-1).ice\_secure(false));

// 此处的"CallbackSender.Proxy"存在于config.client中

if (sender == null)

{

Console.Error.WriteLine("invalid proxy");

LogHelper.WriteTextLog("无效代理", "服务端代理无效或无法连接", DateTime.Now);

return 1;

}

创建客户端的接口服务对象，并将其加入到客户端新建的适配器中，以提供服务端能够与客户端建立连接并能通过回调客户端接口服务方法来实现双向通信。

//调用createObjectAdapter创建一个对象适配器(客户端)

Ice.ObjectAdapter adapter = communicator().createObjectAdapter("Callback.Client");

//我们调用适配器的add，告诉它有了一个新的servant

adapter.add(new CallbackReceiverI(), communicator().stringToIdentity("callbackReceiver"));

adapter.activate();

// 本地接收对象，将调用本地的 CallbackResiver 服务

CallbackReceiverPrx receiver = CallbackReceiverPrxHelper.uncheckedCast(

adapter.createProxy(communicator().stringToIdentity("callbackReceiver")));

//communicator().waitForShutdown();

ClientTest test = new ClientTest(); // 测试方法

test.Test(sender, receiver);

## 7. 异步双向调用

客户端：

2.4节中的“test.Test(sender, receiver);”用来测试客户端和服务端的异步双向通信，主要代码如下：

Info inf = new Info(); // ice文件中定义的类

inf.strMobile = "Callback.Client";

inf.strMsg = "Test";

inf.strUser = "scwj";

inf.strPwd = "123";

// 通过ICE去调用服务端 CallbackSenderI 下的 initiateCallback 服务

Ice.AsyncResult r = sender.begin\_initiateCallback(receiver, inf).whenCompleted(ClientApp.success, ClientApp.exception); // 异步请求

这里，客户端通过服务端的服务代理目标sender来调用服务端的接口服务，使用的是异步方式-- begin\_initiateCallback()，向服务端发送客户端本地提供的服务代理对象receiver和参数数据inf， whenCompleted()用来获取客户端的本次请求是得到服务端的成功响应还是产生异常，其中success和exception是两个方法，分别用来处理响应成功和异常。客户端发送请求之后即可执行后续步骤而不需要阻塞当前进程进行等待。

服务端：

不论客户端采用同步还是异步调用服务端的服务，在服务端最终都是运行自己重写的那个接口服务中的函数(因为建立适配器时添加的服务是自己写的那个接口服务CallbackSenderI)。

比如：客户端调用“sender.initiateCallback(receiver)”（同步方式请求，客户端会挂起等待服务端返回响应结果）或者“sender.begin\_initiateCallback(receiver). whenCompleted(success, exception)”（异步方式请求，请求之后可以继续其他操作），服务端都将运行接口服务CallbackSenderI下的“public override void initiateCallback (CallbackReceiverPrx proxy, Info inf, Ice.Current current)”。

这里，对于客户端的异步请求begin\_initiateCallback()，服务端将执行如下这个服务方法，其中，proxy 是发送请求的客户端所提供的服务代理对象，inf是客户端传送过来的数据，“proxy.begin\_callback(-1);”是服务端通过客户端提供的服务代理对象来异步调用客户端的callback服务，“-1”是服务端要传送给客户端的数据（可以根据需要改变服务端使用该方法时传递的数据）。

public override void initiateCallback(CallbackReceiverPrx proxy, Info inf, Ice.Current current)

{

try

{

……

……

proxy.begin\_callback(-1); // 异步回调

}

}

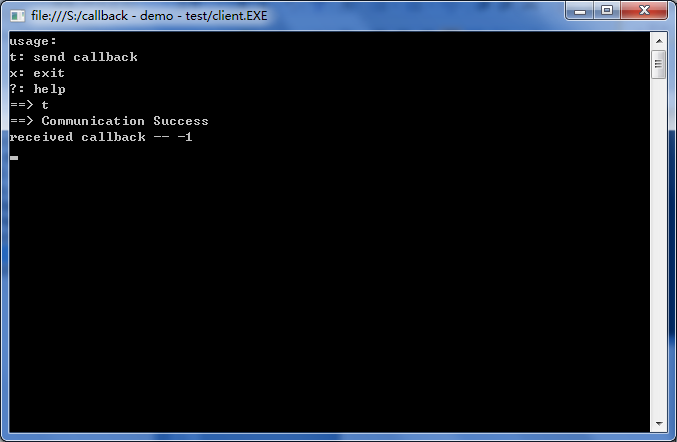
服务端根据发送请求的客户端提供的服务代理对象来回调客户端的接口服务，以此实现客户单与服务端的双向通信。

实际上客户端、服务端都使用异步方式进行回调（双向通信）时，客户端和服务端都有自己的消息存储队列（ICE提供的），当服务端尚未响应完客户端发送过来的消息，而关闭了客户端，此时服务端仍会对已存储的请求（客户端已经发送过来的）进行处理，只是此时服务端给客户端的响应传递不到（客户端重启后收不到服务端后面发送的响应）。当客户端和服务端已经建立通信，但某一时刻服务端重启了，此时客户端不需要重新启动仍旧可以与服务端进行通信。对于传递过来的请求，是按照接收到的顺序来执行的。

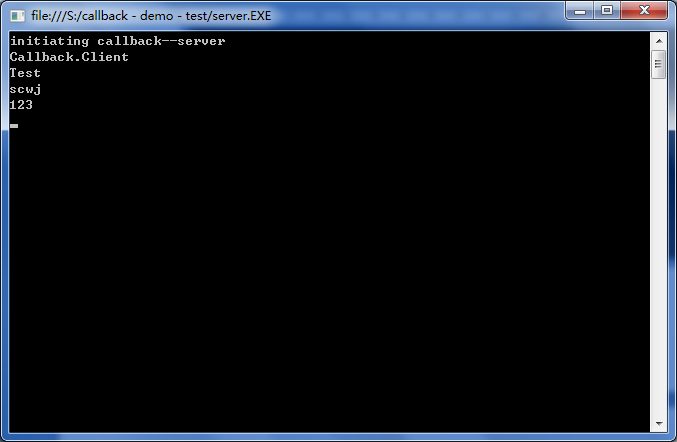
## 8. 运行结果

客户端输入命令“t”，向服务端异步发送一个请求，后面的“Communication Success

”表示服务端对客户端的请求响应成功，“received callback -- -1”是客户端对服务端回调传输的整型数据进行显示。



服务端对客户端当前请求传输过来的参数数据inf进行显示，并在后台调用客户端提供服务代理操作（即回调）。



## 9. 项目源文件及解析

### 9.1 Server项目

#### 9.1.1 Server.cs

using ICEInterface;

using System;

using System.Reflection;

[assembly: CLSCompliant(true)]

[assembly: AssemblyTitle("IceCallbackServer")]

[assembly: AssemblyDescription("Ice callback demo server")]

[assembly: AssemblyCompany("ZeroC, Inc.")]

public class Server

{

public static int Main(string[] args)

{

App app = new App();

return app.main(args, "config.server"); // 调用config.server配置文件来进行配置

}

}

#### 9.1.2 App.cs

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

public class App : Ice.Application

{

public override int run(string[] args)

{

try

{

// 此处的 “Callback.Server” 存在于 config.server 中(按照config.server文件中的配置来设置 Adapter)

Ice.ObjectAdapter adapter = communicator().createObjectAdapter("Callback.Server");

// 服务端添加 CallbackSenderI 接口服务

adapter.add(new CallbackSenderI(), communicator().stringToIdentity("callbackSender"));

// 启用新建的适配器

adapter.activate();

// 监听适配器直到关闭通信器

communicator().waitForShutdown();

}

catch (Exception ex)

{

Console.Error.WriteLine(ex);

LogHelper.WriteTextLog("App run", "应用启用失败！原因为" + ex.Message, DateTime.Now);

}

return 0;

}

}

#### 9.1.3 CallbackSenderI.cs

using ICEInterface;

using System;

public sealed class CallbackSenderI : CallbackSenderDisp\_

{

/// <summary>

/// 服务端提供的接口服务，处理客户端发送过来的信息，并回传服务端响应

/// </summary>

/// <param name="proxy">客户端回调服务代理</param>

/// <param name="inf">客户端传送的信息</param>

/// <param name="current"></param>

public override void initiateCallback(CallbackReceiverPrx proxy, Info inf, Ice.Current current)

{

try

{

Console.Out.WriteLine("initiating callback--server");

Console.Out.WriteLine(inf.strMobile);

Console.Out.WriteLine(inf.strMsg);

Console.Out.WriteLine(inf.strUser);

Console.Out.WriteLine(inf.strPwd);

proxy.begin\_callback(-1); // 异步回调

}

catch (System.Exception ex)

{

Console.Error.WriteLine(ex);

LogHelper.WriteTextLog("initiateCallback", "启用失败！原因为" + ex.Message, DateTime.Now);

}

}

}

### 9.2 Client项目

#### 9.2.1 Client.cs

using ICEInterface;

using System;

using System.Reflection;

using System.Net;

[assembly: CLSCompliant(true)]

[assembly: AssemblyTitle("IceCallbackClient")]

[assembly: AssemblyDescription("Ice callback demo client")]

[assembly: AssemblyCompany("ZeroC, Inc.")]

public class Client

{

public static int Main(string[] args)

{

ClientApp app = new ClientApp();

return app.main(args, "config.client");

}

}

#### 9.2.2 ClientApp.cs

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using ICEInterface;

class ClientApp : Ice.Application

{

public static void success()

{

Console.Out.WriteLine("Communication Success");

}

public static void exception(Ice.Exception ex)

{

if (ex is RequestException)

{

Console.Error.WriteLine("RequestException");

LogHelper.WriteTextLog("RequestException", "请求异常！原因为" + ex.Message, DateTime.Now);

}

else

{

Console.Error.WriteLine("Server call failed:");

Console.Error.WriteLine(ex);

LogHelper.WriteTextLog("Server call failed", "请求失败！原因为" + ex.Message, DateTime.Now);

}

}

public override int run(string[] args)

{

try

{

// sender 调用服务端的 CallbackSenderPrx 服务

CallbackSenderPrx sender = CallbackSenderPrxHelper.checkedCast(

communicator().propertyToProxy("CallbackSender.Proxy").

ice\_twoway().ice\_timeout(-1).ice\_secure(false));

// 此处的"CallbackSender.Proxy"存在于config.client中

if (sender == null)

{

Console.Error.WriteLine("invalid proxy");

LogHelper.WriteTextLog("无效代理", "服务端代理无效或无法连接", DateTime.Now);

return 1;

}

//调用createObjectAdapter创建一个对象适配器(客户端)

Ice.ObjectAdapter adapter = communicator().createObjectAdapter("Callback.Client");

//我们调用适配器的add，告诉它有了一个新的servant

adapter.add(new CallbackReceiverI(), communicator().stringToIdentity("callbackReceiver"));

adapter.activate();

// 本地接收对象，将调用本地的 CallbackResiver 服务

CallbackReceiverPrx receiver = CallbackReceiverPrxHelper.uncheckedCast(

adapter.createProxy(communicator().stringToIdentity("callbackReceiver")));

//communicator().waitForShutdown();

ClientTest test = new ClientTest(); // 测试方法

test.Test(sender, receiver);

}

catch (Exception ex)

{

Console.Error.WriteLine(ex);

LogHelper.WriteTextLog("ClientApp run", "应用启用失败！原因为" + ex.Message, DateTime.Now);

}

return 0;

}

}

#### 9.2.3 CallbackReceiverI.cs

using ICEInterface;

using System;

public sealed class CallbackReceiverI : CallbackReceiverDisp\_

{

/// <summary>

/// 客户端接口服务，对服务端回调传输过来的信息进行处理

/// </summary>

/// <param name="ReMsg">服务端回调传递的信息</param>

/// <param name="current"></param>

public override void callback(int ReMsg, Ice.Current current)

{

System.Console.Out.WriteLine("received callback -- " + ReMsg);

}

}

#### 9.2.4 ClientTest.cs

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using ICEInterface;

class ClientTest

{

private static void menu()

{

Console.Out.Write("usage:\n"

+ "t: send callback\n"

+ "x: exit\n"

+ "?: help\n");

}

/// <summary>

/// 测试方法

/// </summary>

/// <param name="sender">服务端服务代理</param>

/// <param name="receiver">客户端服务代理</param>

public void Test(CallbackSenderPrx sender, CallbackReceiverPrx receiver)

{

menu();

string line = null;

do

{

try

{

Console.Out.Write("==> ");

Console.Out.Flush();

line = Console.In.ReadLine();

if (line == null)

{

break;

}

if (line.Equals("t"))

{

Info inf = new Info();

inf.strMobile = "Callback.Client";

inf.strMsg = "Test";

inf.strUser = "scwj";

inf.strPwd = "123";

//通过ICE去调用服务端CallbackSenderI下的initiateCallback服务

Ice.AsyncResult r = sender.begin\_initiateCallback(receiver, inf).whenCompleted(ClientApp.success, ClientApp.exception); // 异步请求

}

else if (line.Equals("x"))

{

// Nothing to do

}

else if (line.Equals("?"))

{

menu();

}

else

{

Console.Out.WriteLine("unknown command '" + line + "'");

menu();

}

}

catch (System.Exception ex)

{

Console.Error.WriteLine(ex);

LogHelper.WriteTextLog("Test", "测试出现异常！原因为" + ex.Message, DateTime.Now);

}

}

while (!line.Equals("x"));

}

}

### 9.3 配置文件

#### 9.3.1 config.server

config.server文件中“Callback.Server.Endpoints=default -p 10000”，是设置服务端适配器服务代理的通信方式和使用的通信接口，这里保持不变，server端可自行判断client端来源。后面使用SSL来保障通信安全。

#

# The server creates one single object adapter with the name

# "Callback.Server". The following line sets the endpoints for this

# adapter. #服务端的适配器服务代理

#

Callback.Server.Endpoints=default -p 10000

#

# Warn about connection exceptions

#

Ice.Warn.Connections=1 #连接异常警告

#

# Network Tracing #网络连接

#

# 0 = no network tracing

# 1 = trace connection establishment and closure

# 2 = like 1, but more detailed

# 3 = like 2, but also trace data transfer

#

#Ice.Trace.Network=1

#

# Protocol Tracing #传输协议 - messages

#

# 0 = no protocol tracing

# 1 = trace protocol messages

#

#Ice.Trace.Protocol=1

#

# Security Tracing

#

# 0 = no security tracing

# 1 = trace messages

#

IceSSL.Trace.Security=1 #开启SSL安全连接跟踪功能

#

# SSL Configuration

#

Ice.Plugin.IceSSL=IceSSL:IceSSL.PluginFactory

IceSSL.DefaultDir=certs #指定证书和密钥所在目录，需调整

IceSSL.ImportCert.CurrentUser.Root=cacert.pem

IceSSL.CertFile=s\_rsa1024.pfx #证书，需调整

IceSSL.Password=password #密钥文件的密码，有则设置，无则删除

注意：这里需要配置证书和密钥文件，并指定其存放目录“IceSSL.DefaultDir”。如果IceSSL缺少验证将会报错“IceSSL:certificate file not found”，此时需要将文件夹certs拷贝到程序所在目录下，并修改config.client和config.server文件中的

“IceSSL.DefaultDir=certs”，此处指定“certs”文件夹的所在路径。

#### 9.3.2 config.client

需要配置config.client中“CallbackSender.Proxy=callbackSender:tcp -h 192.168.0.231 -p 10000”，指定其中的IP地址为 server 端的IP地址。

“Callback.Client.Endpoints=default”是配置客户端适配点的通信方式为默认方式。

#

# The client reads this property to create the reference to the

# "CallbackSender" object in the server. #服务端的适配器服务代理

#

CallbackSender.Proxy=callbackSender:tcp -h 192.168.0.231 -p 10000

#

# The client creates one single object adapter with the name

# "Callback.Client". The following line sets the endpoints for this

# adapter. #客户端建立适配点

#

Callback.Client.Endpoints=default

#

# Warn about connection exceptions #连接异常警告

#

Ice.Warn.Connections=1

#

# Network Tracing #网络连接

#

# 0 = no network tracing

# 1 = trace connection establishment and closure

# 2 = like 1, but more detailed

# 3 = like 2, but also trace data transfer

#

#Ice.Trace.Network=1

#

# Protocol Tracing #传输协议 - messages

#

# 0 = no protocol tracing

# 1 = trace protocol messages

#

#Ice.Trace.Protocol=1

#

# Security Tracing

#

# 0 = no security tracing

# 1 = trace messages

#

IceSSL.Trace.Security=1 #开启SSL安全连接跟踪功能

#

# SSL Configuration

#

Ice.Plugin.IceSSL=IceSSL:IceSSL.PluginFactory

IceSSL.DefaultDir=certs #指定证书和密钥所在目录，需调整

IceSSL.ImportCert.CurrentUser.Root=cacert.pem

IceSSL.CertFile=c\_rsa1024.pfx #证书，需调整

IceSSL.Password=password #密钥文件的密码，有则设置，无则删除