

Ice 插件技术研究

王 宁, 王 铮

(重庆大学 计算机学院, 重庆 400030)

摘 要: Ice (International Communication Engine) 是一种跨平台跨语言的轻量级分布式计算平台。作为新兴的网络通信中间件, 是否能够适应各种日新月异的互联网技术值得考究, Ice 内建的插件模式对外提供了可扩展接口以支持特定通信协议及向第三方系统的集成。文中从核心源代码入手, 深入分析了 Ice 插件创建、初始化过程, 运行机制与实现技术, 并在不修改 Ice 源代码的情况下以插件方式为其扩展了基于串口的通信协议。实验结果验证了 Ice 传输层协议的可扩展性和 Ice 插件机制的可行性, 为进一步降低开发模块间的耦合度及 Ice 扩展技术的研究奠定了基础。

关键词: Ice; 中间件; 插件; 串口; 分布式系统; 协议

中图分类号: TP31

文献标识码: A

文章编号: 1673-629X(2012)05-0040-05

Research of Plug-In Technology Based on Ice

WANG Ning, WANG Zheng

(School of Computer, Chongqing University, Chongqing 400030, China)

Abstract: Ice (International Communication Engine) is a cross-platform and cross-language lightweight distributed computing platform. As a new network communication middleware, should be able to adapt to ever-changing internet technology. The build-in plug-in models of Ice provides a scalable interface to support specific communication protocols and systems integration to third parties. From the core source code to start, deeply investigated the Ice plug-in creation, initialization process, technology and operational mechanism, and finally extended Ice protocol based on serial communication without modifying its source code. The result shows the scalability of Ice transport layer protocol and the feasibility of Ice plug-in mechanism, which lay a foundation for reducing the coupling between different modules and the study of Ice expansion technology.

Key words: Ice; middleware; plug-in; serial; distributed system; protocol

0 引 言

Ice (International Communication Engine) 中间件是由 ZeroC 公司推出的分布式对象产品, 它吸取了 Corba 的优点并摒弃其不足, 是一种轻量级的网络通信引擎, 其跨平台跨语言和高效的性能在分布式网络通信系统中有很大的发展潜力^[1,2]。

Ice 基于 C/S 构架, 采用 RPC 远程调用作为基本通信方式, 与传统的 C/S 不同的是, Ice 在 RPC 之上建立了一层模型, 客户端可使用代理 (proxy) 与服务端的仆人 (servant) 进行通信, 从而隐藏了繁琐的 RPC 细节。作为支持异构环境开发的中间件, 客户端和服务端可以使用不同的语言, 并由 Ice run time 处理网络通信细节, 从而使得开发者专注于自己的业务逻辑。

目前应用的比较广泛的同类技术还有 Java RMI,

EJB, Jini, 以及微软平台的 WCF (Windows Communication Foundation)。RMI 是 Java 虚拟机之间对象相互调用函数, 启动对方进程的一种方式^[3], 实际上是模拟了应用于分布式计算中的 RPC, 并使用 Java 远程消息交换协议 JRMP (Java Remote Messaging Protocol) 进行通信, 所以具备 Java 平台的优点。EJB 和 Jini 都是基于 Java RMI 的技术, 其设计理念、服务对象及运行环境各有不同^[4]。WCF 是基于 SOA 构架的网络通信 API, 统一了多种 Microsoft 分布式技术, 提供了对跨供应商互操作性支持, 显式的面向服务特性使其成为 Microsoft 新一代分布式开发的核心技术^[5]。相对 RMI、WCF 这样的重量级平台, Ice 显得更加轻便, 且兼具了 .NET 跨语言、Java 跨平台的能力, 亦有研究报告从各个方面将三者进行比较展现出 Ice 在性能上的优势^[6], 不过一个产品需要放在工业环境中反复锤炼才能逐渐成熟, 从这个意义上看来, 尚未得到普及的 Ice 还需要努力。

作为一套开源的网络中间件, 如何在不改动原有设计的情况下加入新的功能, 方便地集成第三方系统

收稿日期: 2011-09-16; 修回日期: 2011-12-20

作者简介: 王 宁 (1985-), 男, 重庆渝中人, 硕士研究生, 研究领域为网络管理、网络通信; 王 铮, 副教授, 硕士生导师, 研究方向为嵌入式操作系统、分布式系统、软件自动生成等。

显得额外重要,为此 Ice 建立了一套插件体系,继承其插件接口就可以方便地添加新功能,只要添加一些配置文件而无需修改源代码甚至不用重新编译,IceSSL 便是官方以插件形式补充的安全协议。Ice 插件扩展并不局限于某种协议,也可以是某种服务,这种方式使得编写服务器应用与编写具体服务得以分离,例如, IceBox 就是这样一种应用服务框架,服务被开发成可动态加载的组件,可以以任何组合方式配置服务,按需启动。

1 插件技术

1.1 插件基本特征

插件是一段可以单独编译和测试但不能单独运行的可执行文件,主程序可以根据需要选择想用的插件,彼此相互独立,只要外部接口一致,双方内部实现均无需改动^[7]。统一调用接口与运行时加载是插件的两个基本特征。

1.2 插件管理与运行方式

主程序通常为支持插件提供插件管理功能,它调用插件接口,其具体功能实际上是各插件自己去完成的。插件管理主要包括:注册、反注册插件;启用、禁用插件;显示插件信息;配置插件参数^[8]。通常插件管理已由主程序设计好,于是可以得出一个插件运行的一般解决方案:在动态链接库中实现插件接口,由主程序显式链接动态加载,在主程序中,插件管理负责插件的安装和卸载,并将安装插件的信息保存到合适的地方,如注册表或配置文件。主程序启动时根据插件的配置信息加载插件模块,然后获得插件的输出函数或输出类的指针得以使用。

1.3 插件实现方式

插件的实现方式可归纳为三种:普通输出函数的 DLL 方式;使用 C++ 多态性;使用 COM 类别 (category) 机制^[9]。不论使用哪种方法,关键在于主程序和插件程序之间的交互途径。

方法一:通过一系列函数交互,这些函数由插件 DLL 引出,再由主程序调用,该方法是面向过程的软件设计,也是后续方法的基础。

方法二:从面向对象的观点来看待问题,由一个或多个抽象基类作为交互途径,插件需要构造一个类来继承此抽象类并实现其接口,再由主程序提供一个创建和销毁此类对象的公共方法。Ice 的插件机制就是采用了这种实现方法。

方法三:通过一个或多个 COM 接口来交互,即使用 COM 技术作为开发插件的基础,插件成为了一个 COM 组件,它实现了接口并注册到约定的组件类别下。这种技术有利于插件与主系统的交互,编写插件

以及扩大插件使用范围,同时也牵涉到大量 COM 技术及原理,增加了开发难度^[10]。

1.4 插件与组件

二十世纪九十年代后,面向组件的软件开发方法得到普及,组件结构的应用程序按功能被划分成了各种组件,彼此相互独立,由管理程序调度通信,功能隐藏在组件内部,对外提供稳定的外观和接口。比如微软推出的 COM 系列技术,以及近年来出现的以 COM 为基础的 OLE, ActiveX^[11]。虽然插件式与组件式程序在结构上比较相似,但本质上是不同的。插件是遵守一定规范的应用程序接口而编写出的程序,组件是按照 COM 规范形成的程序。面向组件的开发方法其目的在于提高软件复用性,节约开发时间和维护成本,而插件式程序的主要目的是便于功能扩展,把各种功能即插即用集成到主系统中。另一方面,组件式应用程序需要大量考虑兼容性和重用性,这使得它对程序有许多额外要求,比如 COM 需要利用 Windows 注册表机制。

2 Ice 插件机制剖析

2.1 插件接口定义

```
module Ice {  
    local interface Plugin  
    {  
  
        void initialize();  
        void destroy();  
  
    };  
}
```

一个 Ice 通信器 (Ice::Communicator) 载入插件时有两个步骤:首先创建所需插件,然后执行 initialize 方法。所有需要扩展的插件,都必须继承自 Ice::Plugin,实现初始化和销毁的方法,也可以根据自身需求定义功能模块及专有操作给核心模块调用。

当一个 Ice 通信器有多个插件时,可能需要一定的初始化顺序。默认情况下, Ice run time 会根据属性 Ice.PluginLoadOrder 定义的顺序装载 (load) 各插件,并调用它们的 initialize 方法进行初始化。

2.2 插件管理器 (PluginManager)

```
module Ice {  
    local interface PluginManager  
    {  
  
        void initializePlugins();  
        StringSeq getPlugins();  
        Plugin getPlugin(string name);  
        void addPlugin(string name, Plugin pi);  
        void destroy();  
  
    };  
}
```

```
};
}
```

继承该接口可以定义自己的插件管理器,但通常无需这样做,因为 Ice 已经实现了一个同名的内部对象, Ice 通信器能够获取该对象对所有插件进行管理,其函数方法的意义如下:

`void initializePlugins()`: 利用该函数可以延迟初始化行为 (delayed initialization): 先设置属性 `Ice. InitPlugins = 0`, 当需要初始化时再手动调用 `initializePlugins()`, 于是插件的装载和初始化得以分离。这种技术在一些应用中有用, 例如 SSL 密钥往往由密码保护着, 但出于安全考虑开发商通常不愿将密码写在配置文件中, 而倾向配置 IceSSL 插件并使用一个回调方法来获取密码, 这时就需要在初始化 IceSSL 之前加载 SSL 密钥。

`StringSeq getPlugins()`: 返回所有已安装插件列表。

`Plugin getPlugin (string name)`: 返回指定插件的引用, 如果在已安装插件中未找到匹配的插件名称, 则抛出异常 `NotRegisteredException`。

`void addPlugin (string name, Plugin pi)`: 该方法提供了一种直接安装插件的途径, 而不必在配置文件中设定属性。

2.3 插件初始化过程

插件初始化最初由 Ice 内部对象发起, 内部对象是指已由 Ice 实现的那些对象, 它们继承并实现了 Slice 接口, Ice 使用这些对象来执行通信任务, 其中最核心的对象即 Ice 通信器, 它提供了各种对象引用, 包括 Ice 插件管理器。所有内部对象的构造函数都是 private, 因为要使用这些对象需要一定的逻辑, 所以要防止外部程序肆意实例化内部对象。

正确的初始化操作应从 `Ice::initialize()` 开始, 它首先创造 Ice 通信器, Ice 通信器的构造函数会调用 `IceInternal::Instance` 的构造函数, 该函数执行各种 Ice 通信器要管理的对象的实际初始化操作, 包括创建插件管理器。接着 Ice 通信器执行 `finishSetup` 函数以完成对所有已安装插件的初始化, 但实际上会继续调用 `IceInternal::Instance` 的同名函数, 进而调用插件管理器的方法 `loadPlugins`, 若 `Ice. InitPlugins != 0`, 该函数会调用各插件的 `initialize` 方法完成初始化操作。最后 `Ice::initialize()` 返回 Ice 通信器的引用。

图 1 说明了插件的初始化过程 (函数形参省略):
_com: 内部核心对象 Ice 通信器, 是 Ice 程序的入口。

口。

_instance: 内部辅助对象, 是各种内部对象初始化实际执行者。

_pluginManager: 内部一般对象, 插件管理器。

为了能正确访问私有成员, `Ice::initialize()` 是 `Ice::CommunicatorI` 的友元函数, `IceInternal::Instance` 是所有欲创建内部对象类的友元类。

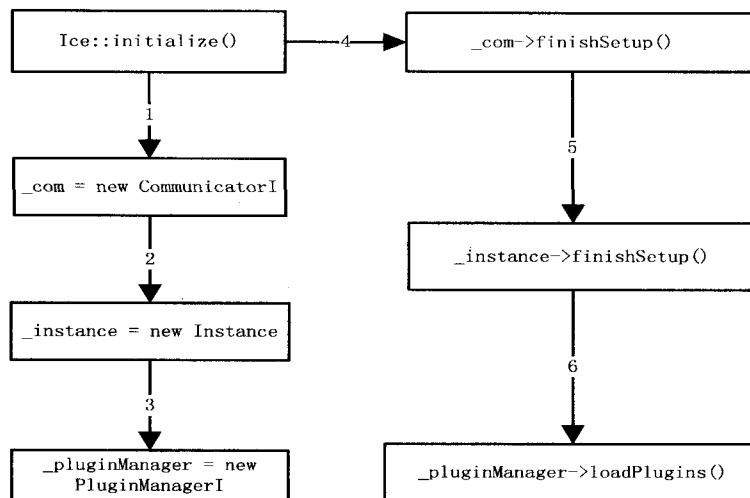


图 1 Ice 插件初始化过程

2.4 插件调用过程

以 IceSSL 为例说明 Ice 应用程序调用插件步骤:

(1) 获取插件管理器的引用。

```
Ice::PluginManagerPtr manager = communicator
->getPluginManager();
```

(2) 向管理器请求获取名为 IceSSL 的插件。

```
Ice::PluginPtr plugin = manager->getPlugin(
"IceSSL");
```

(3) 将获取的插件对象向下转换为具体的插件对象。

```
IceSSL::PluginPtr sslPlugin = IceSSL::Plugin
Ptr::dynamicCast (plugin);
```

(4) 获取了对象引用后, 便可以通过该引用去调用插件的各种操作了。

3 基于串口协议的通信

3.1 Ice 协议构架简析

Ice 协议主要由三个部分组成:

(1) 一组数据编码规则, 确定各种数据的序列化方式。

(2) 一些消息类型, 在客户和服务端之间进行交换; 还有一些规则, 确定在任何情况下要发送何种消息。

(3) 一组规则, 确定客户和服务端怎样就特定的协议和编码版本达成一致。

Ice 协议是直接建立在传输层之上的,支持 TCP、UDP,传输效率极高,并以插件方式增添了强化安全的 IceSSL 协议。Ice 协议的基本结构如图 2 所示。

消息层将请求格式转化为 Ice 规定的消息结构,数据编码层负责编码解码,传输层将整编后的数据在线路上传输。客户应用发起请求或接收返回结果,服务器接收分派请求并返回结果,所以整个过程是双向的。

3.2 串口协议扩展实现

不同的传输机制有不同的特点,这些特点的不同主要体现在对端点对象的设计上^[12]。与按字节传送的 TCP、UDP 不同,串口通信按数据位传输,最少只需一根线,成本低而速度慢。根据其本身特点,将串口设计成面向无连接的协议,并使用第三方跨平台库来实现具体数据的发送和接收,然后编译成 dll 文件作为 Ice 插件。

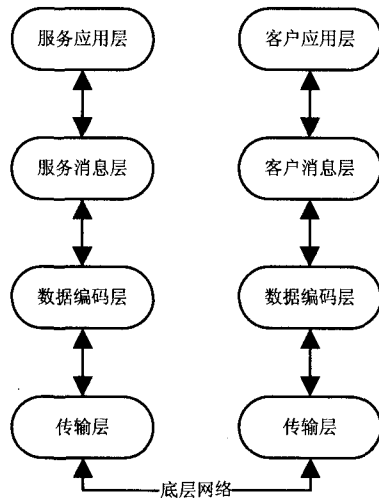


图 2 Ice 协议基本结构

3.2.1 继承插件接口 Ice::Plugin

```

namespace IceComm {
class PluginI: public Ice::Plugin
{
public:
IceComm ( const Ice:: CommunicatorPtr &com ): _com(com) {}

virtual void initialize();
virtual void destroy();
// 其他可选方法
private:
Ice::CommunicatorPtr _com;
};
// 定义智能指针
typedef IceUtil::Handle<PluginI> PluginPtr;
}
  
```

构造函数中必须指定 Ice::CommunicatorPtr, 因为插件亦在 Ice 通信器管理范围内。插件管理器初始化时会调用 initialize(), 若初始化过程失败则执行 destroy(), 一切与启动串口协议相关的操作都应该在初始化中完成。其他可选方法通过 PluginManager 获取 PluginI 的引用来调用。智能指针不但方便使用, 且提供了向下转换的方法(实际上是对 C++ 的 dynamic_cast 的包装)。

3.2.2 继承内部端点与内部端点工厂

```

namespace IceComm {
class EndpointI: public IceInternal:: Endpoint
{
public:
virtual Ice::Short type() const;
virtual Ice::Int timeout() const;
virtual bool compress() const;
// 更多方法
};

class EndpointFactoryI: public IceInternal::
EndpointFactory {
public:
Ice::Short type() const; // 返回协议类型
string protocol() const; // 返回协议名称
IceComm:: EndpointPtr create ( string&, bool )
const; // 创建端点
IceComm:: EndpointPtr read( IceInternal:: Basic
Stream) // 读取端点;
void destroy();
};
}
  
```

为了有效管理不同传输协议的 Endpoint, Ice 内部采用了工厂模式。端点类负责定义新端点的详细信息与支持操作, 如协议名字、协议压缩等。端点工厂类主要负责创建新端点。为了让 Ice 能够解析新协议, Ice 提供了两个内部接口位于 IceInternal 名字空间中, 任何新增协议都需要继承并实现这些接口, Ice run time 会在初始时调用端点工厂的 create 方法。

3.2.3 定义工厂函数, 创建插件实例

```

extern "C"
{
Ice::Plugin * createIceComm( const Ice::
CommunicatorPtr &com, const string &name, const Ice::
StringSeq &arg)
{
return new PluginI( com);
}
}
  
```

}
该函数返回类型是基类 `Ice::Plugin` 的指针,所以在获取 `IceComm` 对象引用时需要进行向下转换。

3.2.4 设置配置文件

在通信双方的配置文件中增加一条属性:`Ice. Plugin. IceComm = dllname; createIceComm`

`IceComm`:自定义的插件名字,`PluginManager` 将通过它来唯一标识插件。

`dllname`:所生成的 `dll` 文件的原始名称,即不包括版本号等信息。亦可指定版本号,如 `mydll.3.4` 代表 windows 中的 `mydll34.dll` 或 `mydll34d.dll` (调试状态),linux 中的 `libmydll.so.3.4`。

3.2.5 实验结果验证

新建 `dll` 工程,按照 3.2 节前三步的方法编写新的串口通信协议 `comm`,新插件命名为 `IceComm`,其中串口读写访问采用了第三方库。编译后将生成的 `IceComm34d.dll` 拷贝到系统目录下。使用 `Ice` 的官方例程 `Hello World` 按照 3.2 节第四步修改相关配置文件:

客户端:

`Hello. Proxy = hello; comm - p 10000`

`Ice. Plugin. IceComm = IceComm; createIceComm`

服务端:

`Hello. Endpoints = comm - p 10000`

`Ice. Plugin. IceComm = IceComm; createIceComm`

最后拔掉两台 PC 的网线并连接 RS-232 串口,分别运行服务与客户端的 `Hello World` 例程,结果显示例程依然正常运行,然而从头到尾并没有改变例程本身的代码,如何证明该例程确实是通过新增的串口协议在通信而不是用的原本的 TCP 或者 UDP 呢?且不论试验中已经拔掉了网线,`Ice` 在通信中采用哪种协议也需要在配置文件中指出,若将配置文件中的 `comm` 替换为其他未知字符,则程序启动时报错:读取配置文件错误。这说明了 `Ice` 是认得 `comm` 这种协议的,当然它原本是不认识的,当继承并实现了端点和端点工厂类时,可以认为 `Ice` 对该协议的内部构造已经相当了解了,但它还不知道如何创建、初始化它,而对插件基类的继承和实现在这里起了桥梁的作用,使得这种新协议(插件)获得了初始化方式;最后 `Ice` 例程在启动时通过配置文件中的第二条属性得知了创建插件实例的

入口地址,于是一切都串起来了。这也证明了 `Ice` 对扩充插件的整合做的非常到位,可以不改动主程序仅通过配置文件就能“即插即用”,亦能通过插件管理获得各种插件对象,在程序中对其做高级控制。

4 结束语

文中简要介绍了一个轻量级分布式网络通讯引擎 `Ice` 和插件技术原理,讨论了 `Ice` 插件创建及初始化过程,并揭示了其内部运作机制,最后以串口为例扩展 `Ice` 协议,从而验证了 `Ice` 传输层协议扩展的可行性及 `Ice` 应用的可伸缩性,亦为进一步解除模块耦合度奠定了良好的基础。

参考文献:

- [1] Henning M. A new approach to object oriented middleware [J]. IEEE Internet Computing, 2004, 8(1): 66-75.
- [2] Henning M, Spruiell M. Distributed programming with ICE [EB/OL]. 2010. www. zero. com.
- [3] 赖宇阳. 采用 Java RMI 实现自动并行计算的叶型优化设计平台[J]. 计算机工程与应用, 2003, 14(1): 205-207.
- [4] 黄小琴, 黎星星, 朱庆生. 分布式应用开发中的 Java 技术分析与比较[J]. 计算机工程与设计, 2004, 25(4): 589-592.
- [5] 严 商. 基于 WCF 的分布式程序的研究与实现[D]. 武汉: 武汉理工大学, 2008.
- [6] Henning M, Spruiell M. Chosing middleware: why performance and scalability do (and do not) mater [EB/OL]. 2011. http://www. zeroc. com/.
- [7] 王 博. ICE 中间件关键技术的研究与实现[D]. 西安: 西安电子科技大学, 2006.
- [8] 陈方明, 陈 奇. 基于插件思想的可重用软件设计与实现[J]. 计算机工程与设计, 2005, 26(1): 172-176.
- [9] 李延春. 软件插件技术的原理与实现[J]. 计算机系统应用, 2003, 7(1): 24-26.
- [10] 潘爱民. COM 原理与应用[M]. 北京: 清华大学出版社, 2000.
- [11] 徐宏兴. 插件体系结构软件开发方法研究[D]. 成都: 四川大学, 2005.
- [12] Ritter T, Schreiner R, Lang U. Integrating security policies via container portable interceptors[J]. IEEE Distributed Systems Online, 2006, 7(7): 1-10.
- [10] 潘建江, 郑建民, 杨勋年. 图像的局部约束变形技术[J]. 计算机辅助设计与图形学学报, 2002, 14(5): 387-393.
- [11] 孙家广, 杨长贵. 计算机图形学(新版)[M]. 北京: 清华大学出版社, 1995.
- [12] Gonzalez R C, Woods R E. 数字图像处理[M]. 北京: 电子工业出版社, 2004.

(上接第 39 页)

161-172.

- [8] 吴宗敏. 函数的径向基表示[J]. 数学进展, 1998, 27(3): 202-208.
- [9] George W. Image morphing: a survey[J]. The Visual Computer, 1998(14): 360-372.