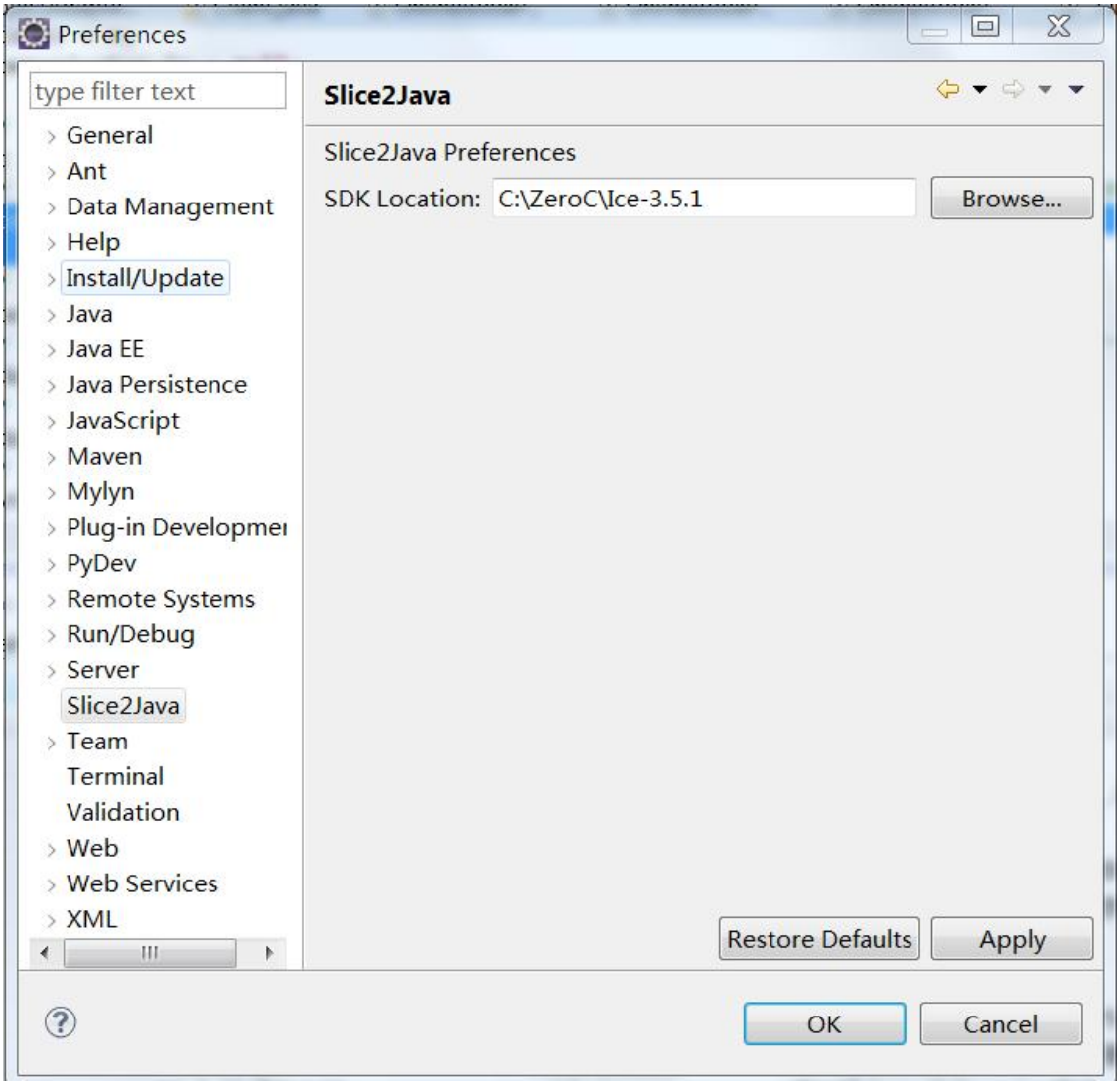
ICE For Java 开发指南

Leader 精品充电系列丛书，淘宝官网：http://shop113755760.taobao.com/

有大量高质量学习代码可供下载，技术交流 QQ：719867650

前言

ICE 版本为 3.5.1，开发环境为 Java 7+Eclipse，开发环境准备如下：

- 安装 JDK 7+Ecliplse
- 安装 ICE Windows 版本： http://www.zeroc.com/download/Ice/3.5/Ice-3.5.1-2.msi
- 安装 ICE for Eclipse 插件：http://www.zeroc.com/eclipse.html ，注意 Zero ICE 的 Home 路径,eclipse->windows-Preference 里 Slice2Java：
- 

Hello world 程序

为了快速掌握 ICE 的编程特性，让我们从最基础的 Hello World 程序开始，我们定义一个服务，取名为在线订票 OnlineBook，该服务提供一个方法 bookTick，实现订票功能，此方法需要一个参数，包括所订票的名称**(name)**，如变形金刚 **5**、票的类型**(type)**，如电影票、价格**(price)**、客户定制要求**(content)**、若票源暂时紧张也预订成功（**valid=true**）等，为此我们封装为 **Java Bean**，名称为 **Message**，如下定义：

```
public class Message
{
   public String name;
   public int type;
   public boolean valid;
   public double price;
   public String content;
}
```

若你细心点，会发现这个结构包括了大多数我们常用的数据类型：字符串、整型、布尔值、小数，这样做的原因很简单，为了验证多语言情况下的接口兼容问题。

我们定义的订票服务的接口如下：

```
public interface OnlineBooK {
   Message bookTick(Message msg)
}
```

该服务返回一个 Message 对象，这么做的原因是因为绝大多数方法调用都会有返回值，对于 RPC 调用来说，有返回值的调用代表着 RPC 方法调用是否成功，而没有返回值的调用，则无法确定是否成功，因为消息发出去也可能在服务端无法处理，另外，只有对有返回值的 RPC 方法进行性能测试，才是有代表性的结果。上面这个简单的 Hellow World 的程序设计，正好体现了编程中的经验和思考的重要性。

理解了上述订票服务的接口、数据结构，我们接下来看看怎样用 ICE 将它变成神奇的 RPC 调用，为了解决跨语言问题，ICE 中采用了业界通常的标准做法，即用一个“中立”的语法定义文件来定义 RPC 方法接口，并提供工具编译为各种第三方语言的接口，这个中立语法就是 ICE 设计的 slice 语言，后缀名为 ice，用 slice 语言定义我们的订票服务，就变成了如下的方式：

```
[["java:package:com.hp.tel.ice"]]
module book{
```

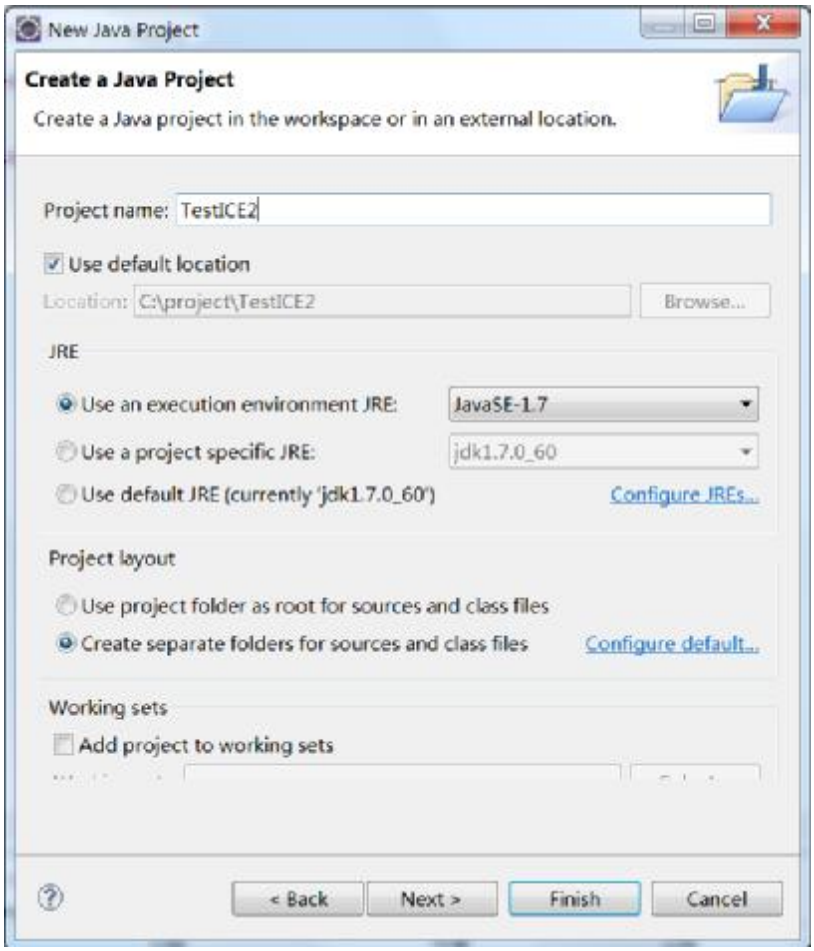
```
struct Message {
string name;
int type;
bool valid;
double price;
string content;
};
        interface OnlineBook{
        Message bookTick(Message msg);
        };
};
```

第一行，是特定含义的注释，告诉 Java 的 slice 转换器，转化为 Java 代码的时候，上述接口是在 com.hp.tel.ice 下产生代码。module 关键字定义了本接口的模块名字 module，这个 module 在 Java 中就是 package 的一部分，在其他语言中也有对于的概念。struct 关键字定义了一个结构体，来自 C 的概念，注意是 struct 不能嵌套 struct。Interface 关键字则定义了 RPC 服务和相关接口。从上面的定义来看，slice 语言还是很简单清晰，基本上你只要花费半天时间，就能覆盖 90%以上的工作需求。

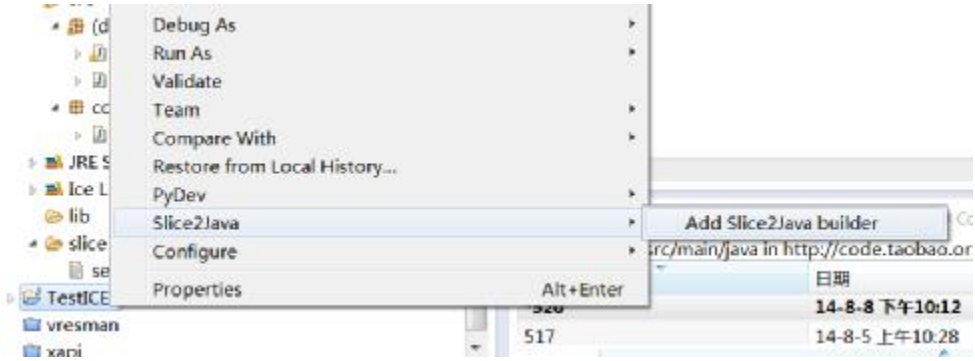
ICE 提供了从 slice 到大多数主流语言的转化工具，命名为 slice2XXX，比如 slice2java、slice2cplus、slice2py、slice2php、slice2rb、在 ICE 的安装目录的 bin 下，你可以看到上述这些命令。可以通过命令行执行 slice 接口到其他语言的转化，生成相应的代码：

```
slice2java xxx.ice
```

现在让我们开始正式编程吧，打开 Eclilpse，建立一个普通的 Java 工程：



点击工程的右键菜单，选择 slice2java 子菜单，执行”add slice2java builder”，



添加 slice2javabuilder 之后，项目自动产生了两个文件夹，slice 目录存放 slice 文件、generated 目录存放生成的 java 源文件，同时项目也添加了 ICE 依赖库(Ice Library)，ICE 依赖库只有一个：ICE.jar，没有复杂的第三方包，容易集成项目，这也是 ICE 的一个优点。

接下来我们只要在 slice 文件夹中创建一个后缀为 ice 的 slice 文件，保存以后，slice2java 插件就自动生成对应的 java 源文件了，保存在 generated 目录下。

service.ice

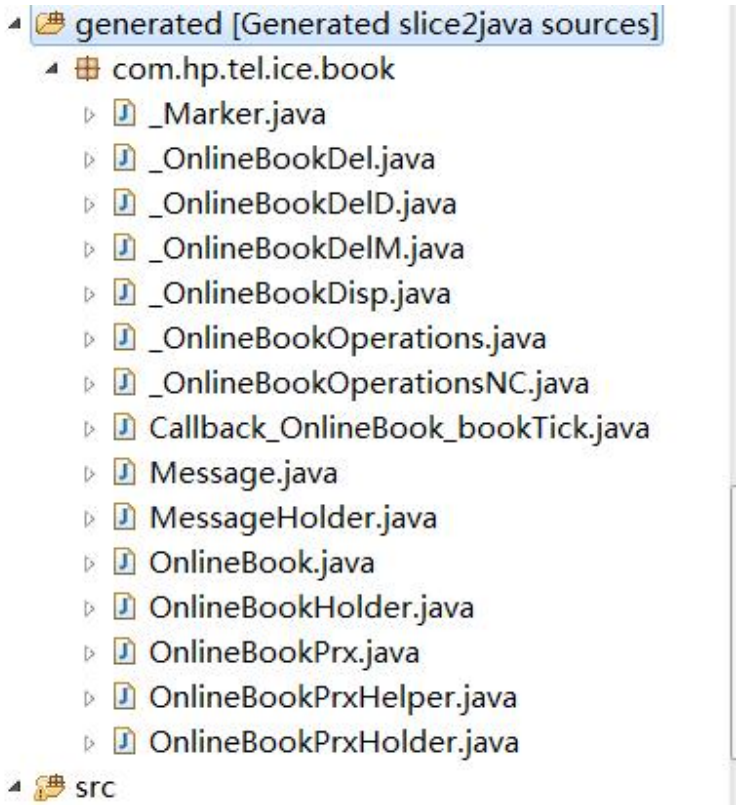
有没有 slice 的 maven 插件呢？很不幸，没有，但 ICE 官方给出一个利用 ant 来在中执行上述过程的实现参考，有兴趣可以试试，毕竟大项目中基本都用 maven 来完成构建过程了。

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>my.project.groupId</groupId>
    <artifactId>my.project.artifactId</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>my.project.name</name>
    <properties>
     <ice.include.path>/usr/share/Ice-3.4.2/slice/Ice</ice.include.path>
    </properties>
    <dependencies>
     <dependency>
      <groupId>com.zeroc</groupId>
      <artifactId>ice</artifactId>
      <version>3.5.1</version>
     </dependency>
    </dependencies>
```

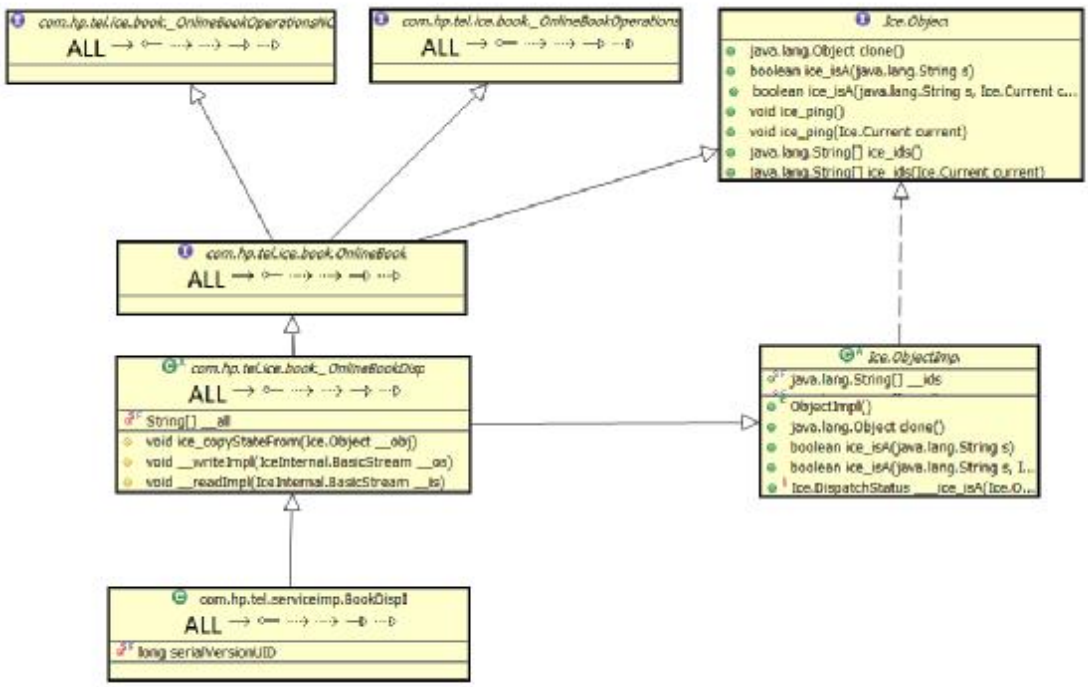
```
<build>
 <plugins>
  <plugin>
   <artifactId>maven-antrun-plugin</artifactId>
   <version>1.7</version>
   <executions>
    <execution>
     <phase>generate-sources</phase>
     <configuration>
      <target name="slice2java">
       <taskdef name="slice2java" classname="Slice2JavaTask"
classpathref="maven.plugin.classpath" />
       <slice2java outputdir="src/generated/java">
        <fileset dir="src/main/resources" includes="fileSystem.ice sale.ice" />
        <includepath>
         <pathelement path="${ice.include.path}" />
        </includepath>
       </slice2java>
      </target>
     </configuration>
     <goals>
      <goal>run</goal>
     </goals>
    </execution>
   </executions>
   <dependencies>
    <dependency>
     <groupId>com.zeroc</groupId>
     <artifactId>ant-ice</artifactId>
     <version>3.5.1</version>
```

```
      </dependency>
    </dependencies>
   </plugin>
   <plugin>
    <groupId>org.codehaus.mojo</groupId>
    <artifactId>build-helper-maven-plugin</artifactId>
    <version>1.7</version>
    <executions>
     <execution>
      <id>add-source</id>
      <phase>generate-sources</phase>
      <goals>
       <goal>add-source</goal>
      </goals>
      <configuration>
       <sources>
        <source>src/generated/java</source>
       </sources>
      </configuration>
     </execution>
    </executions>
   </plugin>
  </plugins>
 </build>
</project>
```

Slice2java 生成的 Java 类如下：



下面我们用 UML 工具（eclipse green uml 插件）看看 ICE 编译后的类之间的关系，



Slice 中的接口对象 OnlineBook 会产生三个相关的 Java 接口类：OnlineBookOperationsNC、OnlineBookOperations、以及继承了这两个接口和 ICE.Object 根接口的 OnlineBook 接口，OnlineBookOperations 与 OnlineBookOperationsNC 为姊妹接口，声明了 OnlineBook 服务中的方法，不同的是 OnlineBookOperations 在方法签名中增加了 ICE 的内部对象 Ice.Current，这个 Current 对象包括了当前调用的网络连接等信息，ICE.Object 对象则代表了一个远程对象，远程对象的 ID、以及验证远程对象是否存活的 Ping 方法，都在此接口中定义，一个实现了本接口的对象在 ICE 中被称之为服务实例——servant。OnlineBookDisp 则是实现了 OnlineBook 接口的抽象类，完成了基本的 RPC 过程，其中 Disp 代表着 Dispatch，即调用分发，从下面的 OnlineBookDisp 的下面方法即可看出 ICE RPC 的基本实现原理：

```
public static Ice.DispatchStatus __bookTick(OnlineBook _obj, IceInternal.Incoming
_inS, Ice.Current _current)
    {
        _checkMode(Ice.OperationMode.Normal, _current.mode);
    //从 Ice.Request 中读取 RPC 请求所关联的网络通道中读取 RCP 方法的参数
        IceInternal.BasicStream _is = _inS.startReadParams();
        Message msg;
        msg = new Message();
    //反序列化，填充 Message 对象
        msg.__read(_is);
        _inS.endReadParams();
    //调用用户实现的 OnlineBook 的具体业务接口
        Message _ret = _obj.bookTick(msg, _current);
    //将结果写回到 RCP 请求的网络应当包中
        IceInternal.BasicStream _os =
_inS.__startWriteParams(Ice.FormatType.DefaultFormat);
        _ret.__write(_os);
        _inS.__endWriteParams(true);
    //完成调用
        return Ice.DispatchStatus.DispatchOK;
    }
```

以下是 Message 参数对象的序列化与反序列化方法的具体实现：

```
public void    __write(IceInternal.BasicStream __os)
  {
    __os.writeString(name);
    __os.writeInt(type);
    __os.writeBool(valid);
    __os.writeDouble(price);
    __os.writeString(content);
  }
  public void
  __read(IceInternal.BasicStream __is)
  {
    name = __is.readString();
    type = __is.readInt();
    valid = __is.readBool();
    price = __is.readDouble();
    content = __is.readString();
  }
```

如果熟悉 Java 的 ObjectStream 类，则发现上述代码看起来很相似，这是 Java 里最基本的对象到字节流的转换逻辑，做过 java 网络编程的很多人都写过类似代码。通过上述分析过程，我们发现一个有趣的现象，即 RPC 调用过程中的请求参数与应答结果，在 ICE 中都统一抽象为“参数”，将读取 RPC 参数与返回 RCP 应答结果的实现逻辑巧妙的统一起来，代码简单明了。细节决定成败，ICE 号称是业界最快的 RPC 框架，其设计和代码质量处处显示功力。

这里顺便说下关于多字节类型的数据在网络上传输的字节顺序问题，这是网络编程中的一个技术细节，称之为“网络字节顺序”，通常有“Little endian （将低序字节存储在起始地址）”与“Big endian ：将高序字节存储在起始地址”两种，JAVA 采用 BIG- Big endian 方式，而 Zero ICE 的则采取了 Little endian 模式，其文档上这么说的：“Data is always encoded using little-endian byte order for numeric types. (Most machines use a little-endian byte order, so the Ice data encoding is "right" more often than not.)”，另外，还补充了一点：“Ice requires clients and servers that run on big-endian machines to incur the extra cost of byte swapping data into little-endian layout, but that cost is insignificant compared to the overall cost of sending or receiving a request.”，那么 ICE 提到的“Most machines”是那些呢？猜猜就知道了，即 X86 体系的机器，它们采用 Little endian 的主机字节顺序，而 RISE 等 Unix 系列的机器则普遍采用 BIG- Big endian 方式。这里若使用 ICE For Java，就

遗留一个悬念了，它这个 Java 版本的，是否是通过编程改变了 Java 默认的“Big endian”方式呢？，因为 Java ByteBuffer 类是可以指定编码方式的，方式如下：ByteBuffer.order(ByteOrder.LITTLE_ENDIAN)，这个问题就留给聪明的你去探索吧。

从上述的 UML 类图中的 Ice.Object 对象，我们还看到 ICE 中很重要的一个概念：对象标识（Identity），从其代码来看，一个 Ice.Object 可以绑定多个 ID，但其中有一个是它最确切的真正的 ID，ID 用来查找和定位 RPC 的远程对象，_OnlineBookDisp 中的下面代码就很清晰的说明了上述这些问题：

```
public static final String[] __ids =
  {
    "::Ice::Object",——所有的 Ice Object 都有此 ID
    "::book::OnlineBook"——这是 OnlineBook 这个 Object 的真正 ID
  };
  public boolean ice_isA(String s) ——用来查找是否是某个远程对象
  {
    return java.util.Arrays.binarySearch(__ids, s) >= 0;
  }
  public String[] ice_ids()
  {
    return __ids;
  }
  public String ice_id()
  {
    return __ids[1];
  }
```

如果你熟悉 J2EE 技术，应该对“Naming”这个词不陌生， Naming Service 是 J2EE 体系的最重要基础组建之一，它的原理就通过某个指定的 ID 来查找绑定的对应 EBJ 对象，ICE 里也有这个组件，它称之为 Locator。既然是远程调用框架，那么我们必须要将某个远程对象“绑定”到某个网络通道，才能完成远程调用，ICE 用 Ice.ObjectAdapter 这个对象来完成具体的相关工作，让我看看 ObjectAdapter 接口中的几个重要方法：

- void activate()——激活此 Adapter 上绑定的 ICE 服务实例，类似 J2EE 容器的加载和钝化 EJB 实例的，用于实现服务器资源的合理利用，按需启动或停止。

- Ice.ObjectPrx add(Ice.Object servant, Identity id)——将某个服务实例以某个名字绑定到本 Adapter 上。
- Ice.ObjectPrx createProxy(Identity id)——创建指定服务实例的 Proxy 对象。

最后我们要了解的一个概念是"Endpoint"，Endpoint 通常可以理解为一个访问地址，在 WebService 中也有这个词："当我们 Host 一个 Web Service 的时候，我们必须给他定义一个或多个 Endpoint，然后 service 通过这个定义的 Endpoint 进行监听来自 Client 端的请求。"，在 ICE 中，Endpiont 有两种：UDP 或 TCP，但基本上很少用 UDP，因为现在的高速网络带宽采用 TCP 长连接情况下，UDP 基本没有什么优势了。怎么创建一个 Endpoint 并绑定我们的 ICE 服务？答案就是 Communicator 对象，它是 ICE 核心对象，沟通 Client 与 Server 端，它的关键方法如下：

- Ice.ObjectPrx stringToProxy(String str)——将一个对象标识转换对应的服务代理对象，用于客户端调用
- ObjectAdapter createObjectAdapterWithEndpoints(String name, String endpoints)——创建 ObjectAdapter 并把 ObjectAdapter 绑定到指定的 Endpoints 上
- LocatorPrx getDefaultLocator()——获取默认的服务 Locator 对象，Locator 对象用于服务代理来定位具体的远程 ICE 服务实例。

至此，我们 Hello World 所需的 ICE 对象都介绍完毕，接下来就可以写具体的实现代码了，先写服务端，在写之前，你可以停下来想想下面几个问题：

- 我们的服务代码继承哪个类？
- 通常的网络编程中，服务端的实现是哪几个过程？

。。。。。。。。。。努力思考
中。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。
考。。。。。。。。。。。。。。。。。。。。。。。。。。。。。
中。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。
。。
。。
。。
。。
。。。

答案下页揭晓，敬请期待！

第一个问题的答案：服务实现 XXXDisp 抽象

第二个问题的答案如下：

- 创建 SocketServer 在一个 EndPoint 上，如 Localhost TCP 9999
- 启动此 SocketServer，在绑定的端口上进行监听
- 写消息接收和处理的无限循环逻辑，直到停止服务

Ok，我们先来写服务实现类（servant），按照 ICE 的建议，servant 类的后缀名为 I（Implement 缩写），为了将 Slice 生成的 java 代码与我们的实现类分开，我们建议定义另外的报名来存放我们的实现类，服务的代码为了简化，直接返回订单请求：

```
import Ice.Current;
import com.hp.tel.ice.book.Message;
import com.hp.tel.ice.book._OnlineBookDisp;
public class OnlineBookI extends _OnlineBookDisp{
    private static final long serialVersionUID = 1L;
    @Override
    public Message bookTick(Message s, Current __current) {
        return s;

    }
}
```

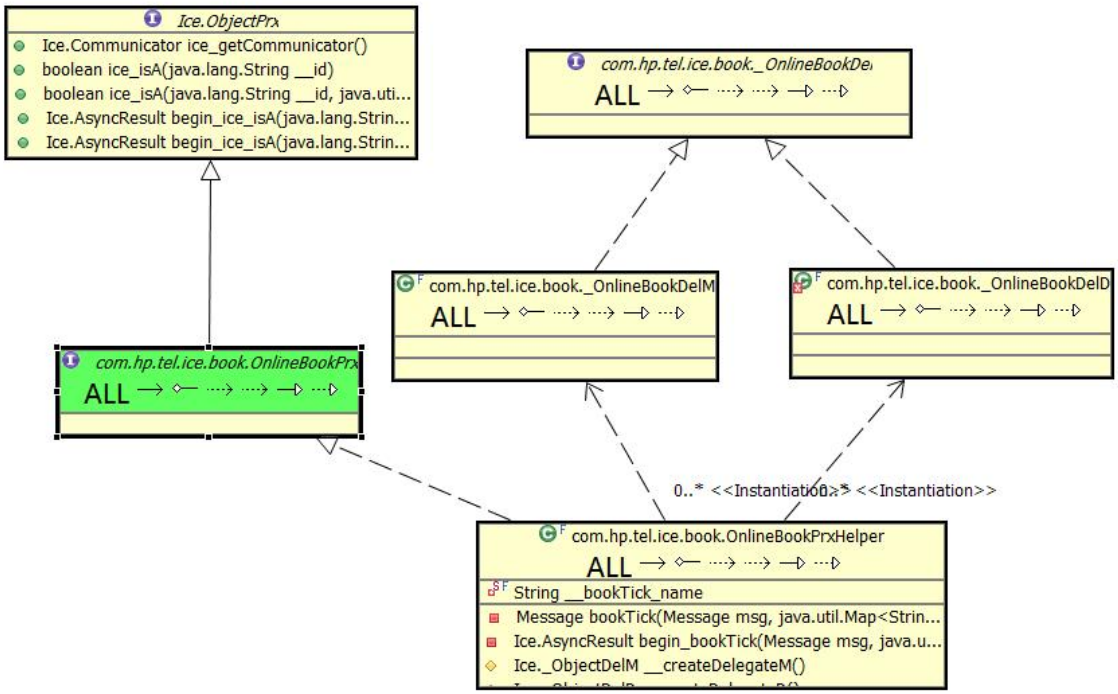
接下来实现 Sever 端代码：

```
public class Server {
    public static void main(String[] args) {
        int status = 0;
        Ice.Communicator ic = null;
        try {
            // 初始化 Communicator 对象，args 可以传一些初始化参数，如连接超时，初始化客户端连接池的数量等
            ic = Ice.Util.initialize(args);
            // 创建名为 OnlineBookAdapter 的适配器，并要求适配器使用缺省的协议（TCP/IP 端口为 10000 的请求）
            Ice.ObjectAdapter adapter = ic.createObjectAdapterWithEndpoints(
```

```
                                        "OnlineBookAdapter", "default -p 10000");
                        // 实例化一个 OnlineBook 服务对象
                        OnlineBookI object = new OnlineBookI();
                        // 将服务单元增加到适配器中，并给服务对象指定 ID 为
OnlineBook，该名称用于唯一确定一个服务单元
                        adapter.add(object, Ice.Util.stringToIdentity("OnlineBook"));
                        // 激活适配器
                        adapter.activate();
                        // 让服务在退出之前，一直持续对请求的监听
                        System.out.print("server started ");
                        ic.waitForShutdown();

                } catch (Exception e) {
                        e.printStackTrace();
                        status = 1;
                } finally {
                        if (ic != null) {
                                ic.destroy();
                        }
                }
                System.exit(status);
        }
}
```

ICE Server 的代码就这么简单，甚至比大多数 NIO Socket 框架用起来都简单，下面我们再看看客户端相关的 UML 图：



Ice.ObjectPrx 代表一个远程对象的代理，实现了基础的通信和方法调用框架，XXXDel 接口定义了远程访问接口，对于每个 ICE 服务，ICE 默认生成了两个实现类：本地调用（XXXDelD）和远程调用（XXXDelM），前者用在服务直接调用的优化过程中，即若多个 ICE 服务部署在一个 JVM 中并有相互调用关系的情况下，ICE 是能做优化的；后者则是通过网络进行远程调用，ICE 的这个设计与 J2EE EJB 的本地接口有远程接口很相似，只不过它这边把这个过程给自动化和隐蔽了，抽象类 ObjectPrxHelperBase 里创建代理的这段代码揭示了 ICE 服务调用优化的秘密：

```
_ObjectDel  createDelegate(boolean async)
  {
    if(_reference.getCollocationOptimized())——若具备优化条件，则创建本地直接调用代理对象 XXXDelD
    {
      ObjectAdapter adapter = _reference.getInstance().objectAdapterFactory().findObjectAdapter(this);
      if(adapter != null)
      {
        _ObjectDelD d = __createDelegateD();
        d.setup(_reference, adapter);
```

```
            return d;
        }
    }
    _ObjectDelM d = __createDelegateM();
    d.setup(_reference, this, async);
    return d;
  }
```

在这里，继承了 ObjectPrxHelperBase 的 OnlineBookPrxHelper 实现了 OnlineBookPrx 接口，具体的远程调用逻辑工作则交给 _OnlineBookDelD 或者或 _OnlineBookDelM 来完成，前者是本地调用，不涉及到网络通信，后者的代码如下：

```
    public Message   bookTick(Message msg, java.util.Map<String, String> __ctx,
Ice.Instrumentation.InvocationObserver __observer)
        throws IceInternal.LocalExceptionWrapper
    {
    //得到网络通信的输出通道
        IceInternal.Outgoing __og = __handler.getOutgoing("bookTick",
Ice.OperationMode.Normal, __ctx, __observer);
        try     {       try
        {
    //写 RCP 调用的参数，Message 对象序列化到输出流中
     IceInternal.BasicStream __os =
__og.startWriteParams(Ice.FormatType.DefaultFormat);
            msg.__write(__os);
            __og.endWriteParams();
        }
        catch(Ice.LocalException __ex)
        {
            __og.abort(__ex);
        }
    //发送数据并检查是否成功
        boolean __ok = __og.invoke();
```

```
        try
        {
            if(!__ok)
            {
                try
                {
                    __og.throwUserException();
                }
                catch(Ice.UserException __ex)
                {
                    throw new Ice.UnknownUserException(__ex.ice_name(), __ex);
                }
            }
//等待服务端的响应结果消息，并反序列化为 Message 应答对象
            IceInternal.BasicStream __is = __og.startReadParams();
            Message __ret;
            __ret = new Message();
            __ret.__read(__is);
            __og.endReadParams();
            return __ret;
        }
        catch(Ice.LocalException __ex)
        {
            throw new IceInternal.LocalExceptionWrapper(__ex, false);
        }
    }
    finally
    {//清理此次调用
        __handler.reclaimOutgoing(__og);
```

```
    }
  }
```

经过上述分析，我们基本能够理解 ICE 客户端的调用实现原理了，接下来就来实现 Client 代码：

```
public class Client {
    public static void main(String[] args) {
        int status = 0;
        Ice.Communicator ic = null;
        try {
            // 初始化通信器
            ic = Ice.Util.initialize(args);
            // 传入远程服务单元的名称、网络协议、IP 以及端口，获取 OnlineBook 的远程代理，这里使用 stringToProxy 方式
            Ice.ObjectPrx base = ic
                    .stringToProxy("OnlineBook:default -p 10000");
            // 通过 checkedCast 向下转型，获取 OnlineBook 接口的远程，并同时检测根据传入的名称获取服务单元是否 OnlineBook 的代理接口，如果不是则返回 null 对象
            OnlineBookPrx onlinBook = OnlineBookPrxHelper.checkedCast(base);
            if (onlinBook == null) {
                throw new Error("Invalid proxy");
            }
            // 调用服务方法
            Message msg = new Message();
            msg.name = "Mr Wang";
            msg.type = 3;
            msg.price = 99.99;
            msg.valid = true;
            msg.content = "abcdef";
```

```
                long start=System.currentTimeMillis();
                int count=100000;
                for(int i=0;i<count;i++)
                {
                onlinBook.bookTick(msg);
                }
                long used=System.currentTimeMillis()-start;
                System.out.print("tps "+count*1000.0/used);
            } catch (Exception e) {
                e.printStackTrace();
                status = 1;
            } finally {
                if (ic != null) {
                    ic.destroy();
                }
            }
            System.exit(status);
        }
}
```

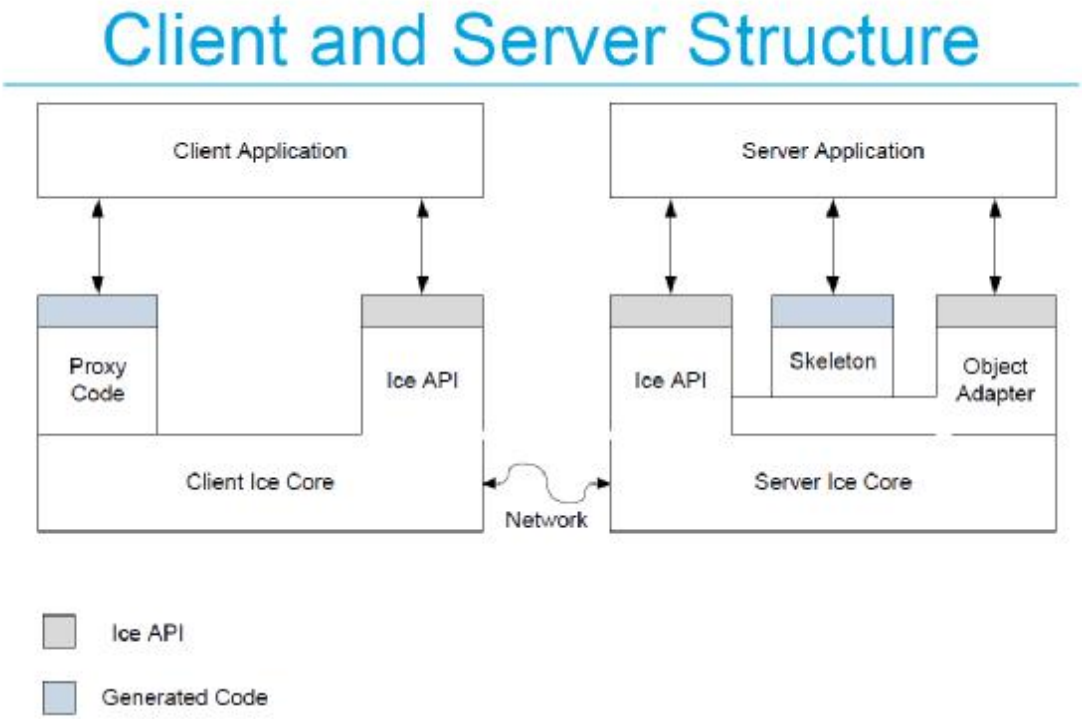
代码中 OnlineBookPrx onlinBook = OnlineBookPrxHelper.checkedCast(base);这一段实际上是从只包括基础的对象 ID、EndPoint 等相关信息的基础 ObjectPrx 类拷贝信息，并构造一个具体的 ObjectPrx 实现类—— OnlineBookPrxHelper，其源码如下：

```
public static OnlineBookPrx checkedCast(Ice.ObjectPrx _obj)
  {
    OnlineBookPrx _d = null;
    if(_obj != null)
    {
      if(_obj instanceof OnlineBookPrx)
      {
        _d = (OnlineBookPrx)_obj;
```

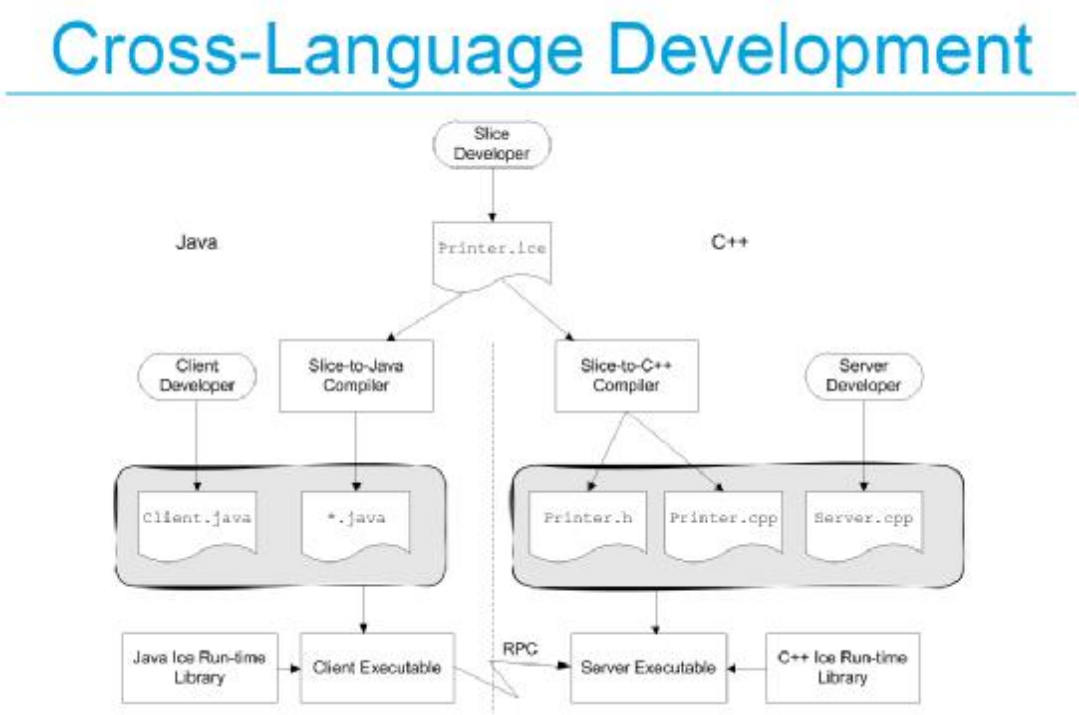
```
        }
        else
        {
          if(__obj.ice_isA(ice_staticId()))
          {
            OnlineBookPrxHelper __h = new OnlineBookPrxHelper();//构造具体的 Proxy 类
            __h.__copyFrom(__obj);//从基础 Proxy 中拷贝必要信息，包括连接等内部对象
            __d = __h;
          }
        }
      }
      return __d;
    }
```

启动 Server，然后启动 Client，测试一下，看看你本机调用结果如何，我本机上达到 2.5 万每秒的性能，这个性能差不多是 HTTP 通道的 10 倍左右，也验证了 ICE 的高性能。

下图是 ICE 客户端与服务器端的通信原理图，我们之前分析的哪些由 ICE 生成的代码，就分别存在于客户端与服务器端：



ICE 跨平台的实现原理，是通过 Slice 定义的接口文件，分别编译成不同语言的实现类，RPC 方法和参数采用特定的编码方式，使得各个语言都能解析，从而实现跨语言的调用，这类似 Cobra 的做法，下图是一个 C 开发的 ICE 服务被 Java 客户端调用的原理示意图：



本节最后留几个小问题：

- Endpoint 中的 default 是什么含义
- 若客户端在另外的机器上，客户端在构造 ObjectPrx 的时候，怎么指定服务端的 IP 地址？（提示：若熟悉 MySQL 命令行的连接，估计立即想到答案）
- 若服务在多个机器上注册，客户端怎么调用？

好用的 Icebox

在前面的 Hello World 一章中，我们初步了解了 ICE 的原理，掌握了其重要概念：ICE 服务（servant）、服务的 ID、servant 绑定到 ObjectAdaptory 上、服务的 Endpoint 等，通过编写简单的 Hello World 程序，我们也掌握了其基本用法。接下来，我们开始学习 ICE 的服务框架——Icebox，这个类似一个 J2EE 中间件，可以大大的简化服务端的开发，让我们“只开发服务本身”而不是“开发完整系统”。

简单的说，Icebox 就好象是一个 Tomcat 中间件，我们只要写 N 个 ICE 服务的代码，然后用一个装配文件定义需要加载的服务列表、服务的启动参数、启动次序等必要信息，然后启动 IceBox，我们的应用系统就能够正常运行了。与现在流行的 XML 文件配置不同，Icebox 采用的是 Unix 上通用的方式——属性文件的方式，可能的一个原因是省去复杂的 XML 解析和相关 lib 吧。

另外，一个 IceBox 内部创建一个 service manager 的对象（Ice.Admin）,这个对象专门负责 loading 和 initializing 那些配置好了的服务，你可以选择性地将这个对象暴露给远程的客户端,譬如 IceBox 或者 IceGrid 的 administrative utilities,这样 IceBox 和 IceGrid

就可以执行某些远程管理任务，Ice.Admin 默认是关闭的，而在 IceGrid 中部署的 IceBox,则会被自动地激活。

要将一个 ICE 服务纳入到 Icebox 中，我们需要引入 IceBox.jar 这个库，另外只需让这个服务的实现类实现 Ice 的 Service 接口，此接口定义如下：

- void start(String name, Ice.Communicator communicator, String[] args)—— 服务可以在 start 操作中初始化自身；这通常包括创建对象适配器和 servant。name 和 args 参数提供了来自服务的配置的信息，而 communicator 参数是服务管理器为供服务使用而创建的 Ice.Communicator 对象。取决于服务的配置，这个通信器实例可能会由同一个 IceBox 服务器中的其他服务共享，因此，你需要注意确保像对象适配器（ObjectAdapter）这样的对象的名字是唯一的。
- void stop()——stop 方法中回收所有被 service 使用的资源，一般在 stop 中 service 会使一个 object adapter 对象无效(deactivates)，不过也有可能对 object adapter 执行一个 wartForDeactivate 的方法来确保所有未完成的请求在资源清理前得到处理，默认会将它的 object adapter 作为 communicator 销毁的一部分连同 communicator 对象一起 destory 掉，但某些情况下不能做到，如 IceBox 中 services 使用共享的 communicator 对象时，此时你需要明确指明销毁它的 object adapter 对象。

继续之前的例子，我们把 OnlineBook 服务变成 IceBox 托管的服务，只要再增加一个类，实现上述接口，并在 start 里完成服务绑定即可，代码如下：

```
public class BoxOnlineBookI  implements Service {

   private Ice.ObjectAdapter _adapter;

   private static final long serialVersionUID = 1L;

private Logger getLogger() {
//用来记录服务的日志信息
          return _adapter.getCommunicator().getLogger();
   }

   @Override

   public void start(String name, Communicator communicator, String[] args) {

          // 创建 objectAdapter，这里和 service 同名

          _adapter = communicator.createObjectAdapter(name);

          // 创建 servant 并激活

          Ice.Object object = new OnlineBookI();

          _adapter.add(object, communicator.stringToIdentity(name));

          _adapter.activate();

getLogger().trace(
```

```
                name,
                "service started ,with param size " + args.length + " ,detail:"
                        + Arrays.toString(args));

    }

    @Override
    public void stop() {
        _adapter.destroy();

    }
```

接下来我们看看 Icebox 的配置文件，其配置文件分为两部分，一部分是共性，一部分是具体服务定义相关的配置：

共性部分如下：

```
#本 Icebox 的实例名称
IceBox.InstanceName=MyAppIceBox 1
#在所有服务的初始化完成之后，服务管理器将打印"token ready"。如果有脚本想
#要等待所有服务准备就绪，这项特性很有用，下面例子就会输出日志：
#“ MyAppIceBox1 ready”
IceBox.PrintServicesReady= MyAppIceBox 1
#定义服务的启动顺序，解决服务启动过程中的先后顺序问题
IceBox.LoadOrder=serv1,serv2,serv3
#优化本地服务之间调用的重要参数，UseSharedCommunicator，若 Hello 与
Printer 两个服务存在调用关系，又部署在一个 Icebox 实例中，则可以定义两者使
用同一个 Communicator 对象，实现服务本地调用的优化。
IceBox.UseSharedCommunicator.Hello=1
IceBox.UseSharedCommunicator.Printer=1
#Ice 一些常用参数属性
    Ice.MessageSizeMax = 2048
    Ice.Trace.Network=1
```

Ice.Trace.ThreadPool=1

Ice.Trace.Locator=1

#定义 IceBox 服务管理器接口的端点，以激活 IceBox 管理服务使之能够被远程控制，默认是关闭的，在内网中可以打开，并绑定私有地址。

IceBox.ServiceManager.Endpoints=tcp -p 9999 -h localhost

下面是定义具体服务的时候参数格式，每个服务的参数格式如下：

IceBox.Service.name=entry_point [--key=value] [args]

- name 这个属性定义了 service 的名字，作为 start 方法的 name 参数，必须唯一。
- entry_point 是上面的 service 的完整类名，必须可以在 classpath 中可以找到）。
- [--key=value] 将会被作为 property 属性用于构造该服务的 communicator，用来更加精确的控制每个 Ice 服务的性能调优，这里也可以用--Ice.Config=xxx.cfg 的方式从具体服务的配置文件中加载参数，具体是一个大而全的配置文件，还是 N 个独立的小配置文件，这是一个仁者见仁的问题了，另外也可以用 IceBox.InheritProperties=1 这个属性，让所有的 Ice 服务实例都使用 IceBox 的配置属性。
- [args]部分则作为参数传入到 start 方法的参数 String[] args 中，作为服务的启动初始化参数。

继续上面的例子，我们的最终配置文件[config.properties]的内容如下：

#server properties

IceBox.InstanceName=MyAppIceBox 1

IceBox.InheritProperties=1

IceBox.PrintServicesReady= MyAppIceBox 1

IceBox.ServiceManager.Endpoints=tcp -p 9999 -h localhost

#performance properties

Ice.ThreadPool.Server.Size=4

Ice.ThreadPool.Server.SizeMax=100

Ice.ThreadPool.Server.SizeWarn=40

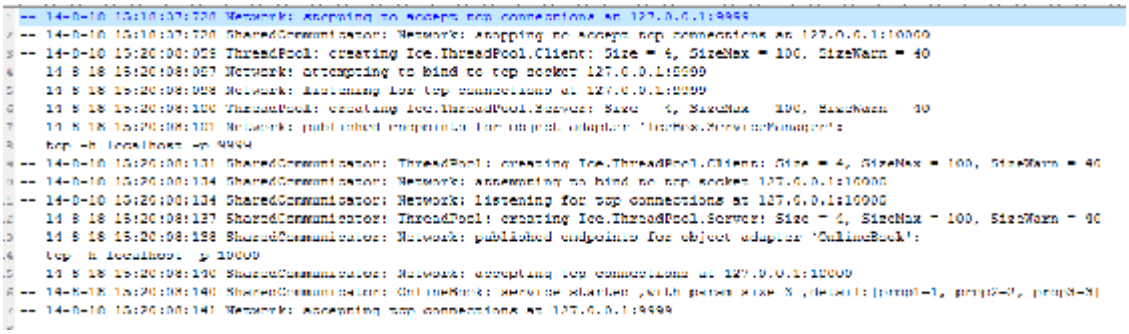
Ice.ThreadPool.Client.Size=4

```
Ice.ThreadPool.Client.SizeMax=100
Ice.ThreadPool.Client.SizeWarn=40
#for system stronger
Ice.ACM.Client=300
Ice.ACM.Server=300
# log and trace
#表明日志存放在日志文件中，否则会在控制台输出日志
#Ice.LogFile=iceserv.log
Ice.PrintStackTraces=1
Ice.Trace.Retry=2
Ice.Trace.Network=2
Ice.Trace.ThreadPool=1
Ice.Trace.Locator=2
Ice.Warn.Connections=1
Ice.Warn.Dispatch=1
Ice.Warn.Endpoints=1
#service define begin
IceBox.Service.OnlineBook=com.hp.tel.serviceimp.BoxOnlineBook prop1=1
prop2=2 prop3=3
OnlineBook.Endpoints=tcp -p 10000 -h localhost
#service define end
#service load order
IceBox.LoadOrder=OnlineBook
#service share communicator
IceBox.UseSharedCommunicator.OnlineBook=1
```

上述文件若在 Java classpath 里（如跟 java 类放在一起），则启动 Icebox 的命令如下，否则 Ice.Config 用绝对路径即可：--Ice.Config=C:\project\TestIce\config.properties。

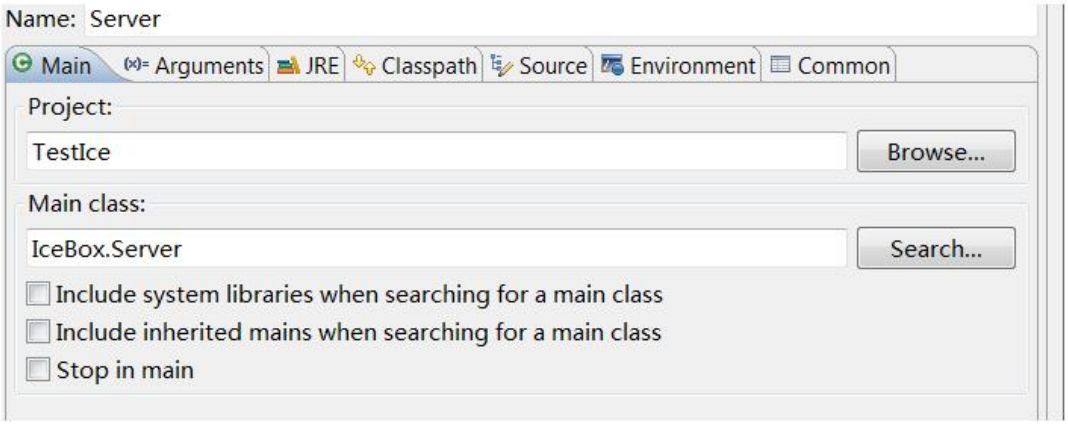
- java -cp ".\;C:\ZeroC\Ice-3.5.1\lib*" IceBox.Server --Ice.Config= config.properties

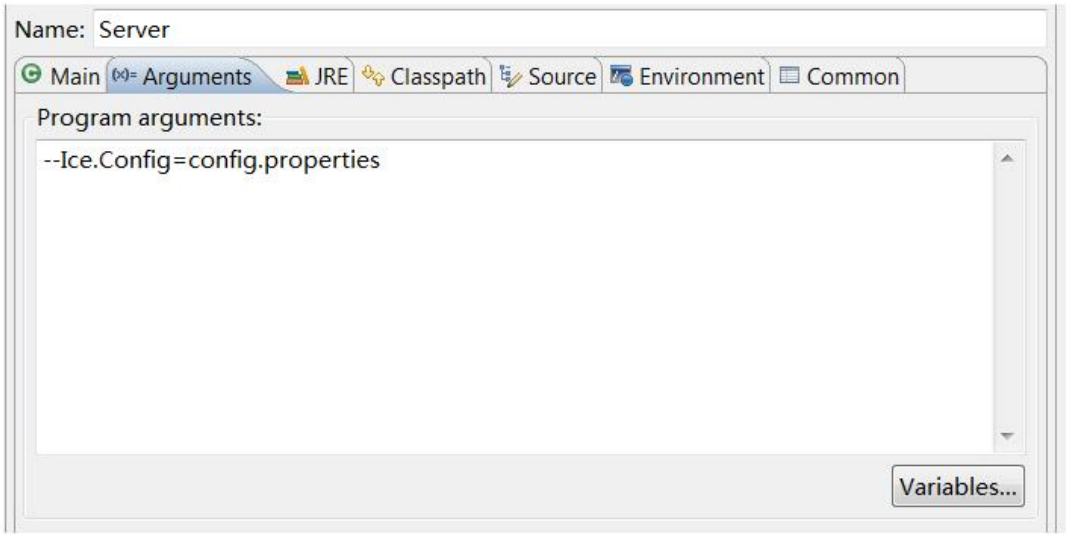
启动成功以后，日志文件的输出如下：



从日志中我们看到，一共启动两个端口：9999 为管理端口，10000 为我们的业务端口，线程池的数量大小也安装我们的要求进行了配置。

提示：eclipse 中也可以启动和调试 Icebox，界面如下：





下面我们增加另外一个服务，SMSService，来测试服务之间调用的问题，其 slice 文件定义如下（smsservice.ice）：

[["java:package:com.hp.tel.ice"]]

```
module message{
    interface SMSService{
        void sendSMs(string msg);
    };
};
```

为了省去 Icebox 的服务包装类，我们将包装类和业务实现类在一个对象里面实现，下面是 SMSService 的实现类：

```
public class SMSServiceI extends _SMSServiceDisp implements Service {
    private Logger log;
    private ObjectAdapter _adapter;
    private static final long serialVersionUID = 1889978048086570687L;

    @Override
    public void sendSMs(String msg, Current __current) {
        //log.trace("service", "send sms message " + msg);
    }
    @Override
    public void start(String name, Communicator communicator, String[] args) {
        this.log = communicator.getLogger().cloneWithPrefix(name);
        // 创建 objectAdapter，这里和 service 同名
        _adapter = communicator.createObjectAdapter(name);
        // 创建 servant 并激活
        Ice.Object object = this;
        _adapter.add(object, communicator.stringToIdentity(name));
        _adapter.activate();
        log.trace("control", " started ");
    }
    @Override
    public void stop() {
        log.trace("control", " stoped ");
```

```
            _adapter.destroy();
        }
    }
```

这里将服务的日志通过克隆方法变成了另外的日志：

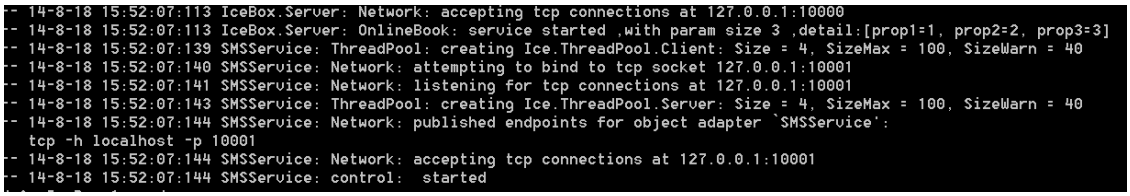
communicator.getLogger().cloneWithPrefix(name)，目的是日志输出更加符合规范，log.trace 的第一个参数并非日志级别，而是属于分类这样的概念，我们修改 config.properties，增加下面的定义：

IceBox.Service.SMSService=com.hp.tel.serviceimp.SMSServiceI

SMSService.Endpoints=tcp -p 10001 -h localhost

IceBox.LoadOrder=OnlineBook,SMSService

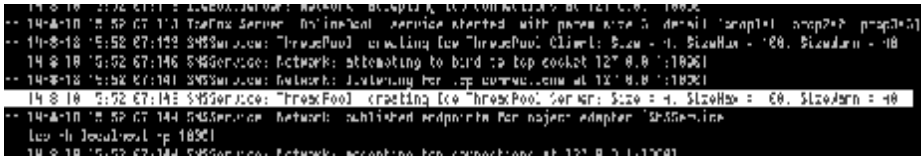
启动以后，观察到日志里面 SMSService 与 OnlineBook 的输出有所不同：



另外，有一个有趣的问题，我们是否可以将多个不同的服务绑定到同一个 Endpoint 上去？用之前的 BoxOnlineBookI 这种单独的类，在 Start 方法里面通过调用 _adapter.add(object, communicator.stringToIdentity(name))，增加多个 IceObject 的方式是可以做到的，但此时这个类已经不是 IceBox 的原本定义的“服务”概念了，其官方不建议多个服务绑定到同一个 EndPoint 上，并解释如下：

“It would be possible for two IceBox services to share an endpoint, but only if the services cooperated with one another to share a single object adapter instance (since object adapters own the endpoints). This would certainly not be a typical use case for IceBox. Even if several services were configured to share a single communicator, each service would normally create its own object adapter and therefore get its own endpoint.”

至于是否多个 Service 绑定到一个 Endpoint，这里还有另外一个重要原因需要考虑，看下日志：



每一个服务都有自己的隔离的线程池，这样会避免一个服务的问题导致另外的服务不可用，因此，Icebox 从端口上和线程池上分离不同的服务，对于系统的稳定可靠运行，是有很大好处的。与此相关的服务之间本地调用的参数：IceBox.UseSharedCommunicator 设

置以后，共用同一个 communicator 的服务会共用同一个线程池，后端日志上也能发现这个现象。

接下来我们来看看服务之间怎么调用的问题，先修改 SMSServiceI 服务代码如下：

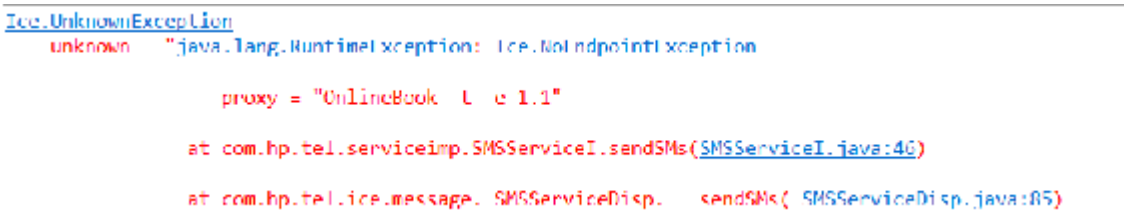
```
public void sendSMs(String msg, Current __current) {
        // log.trace("service", "send sms message " + msg);
        if (msg.startsWith("book")) {
            Ice.ObjectPrx base;
            try {
    base=_adapter.getCommunicator().stringToProxy("OnlineBook");
                OnlineBookPrx onlinBook =
OnlineBookPrxHelper.checkedCast(base);
                // log.trace("service", "find proxy "+onlinBook);
                Message bookMsg = new Message();
                bookMsg.name = "Mr Wang";
                bookMsg.type = 3;
                bookMsg.price = 99.99;
                bookMsg.valid = true;
                bookMsg.content = "abcdef";
                onlinBook.bookTick(bookMsg);
            } catch (Exception e) {
                throw new RuntimeException(e);
            }           }       }
```

客户端调用代码如下：

```
public class SMSClient {
    public static void main(String[] args) {
        int status = 0;
        Ice.Communicator ic = null;
        try {
            // 初始化通信器
```

```
                ic = Ice.Util.initialize(args);
                // 传入远程服务单元的名称、网络协议、IP 以及端口，获取
OnlineBook 的远程代理，这里使用 stringToProxy 方式
                Ice.ObjectPrx base = ic
                                .stringToProxy("SMSService:default -p 10001");
                // 通过 checkedCast 向下转型，获取接口的远程
                SMSServicePrx smsSrvPrx =
SMSServicePrxHelper.checkedCast(base);
                if (smsSrvPrx == null) {
                        throw new Error("Invalid proxy");
                }
                // 调用服务方法
                long start = System.currentTimeMillis();
                int count = 100000;
                for (int i = 0; i < count; i++) {
                        smsSrvPrx.sendSMs("book msg to you ");
                }
                long used = System.currentTimeMillis() - start;
                System.out.print("tps " + count * 1000.0 / used);
        } catch (Exception e) {
                e.printStackTrace();
                status = 1;
        } finally {
                if (ic != null) {
                        ic.destroy();
                }
        }
        System.exit(status);
    }
}
```

启动服务，运行 SMSClient，会发现下面这个错误：



原因是上述两个服务不是用同一个 Communicator，因此虽在一个 JVM 里，还是无法看到彼此，为了解决这个问题，ICE 有两个办法：

- 将相互调用的 ICE 服务设为共用 Communicator 的方式
- 采用 Locator 来解决服务地址查询的问题

第一种方式，只要在配置文件中增加下面的配置即可解决：

IceBox.UseSharedCommunicator.OnlineBook=1

IceBox.UseSharedCommunicator.SMSService=1

第二种方式，则演进成为 ICE 分布式的模式——IceGrid，这一章有点难度，先休息一天，然后继续。

IceGrid 的魔法

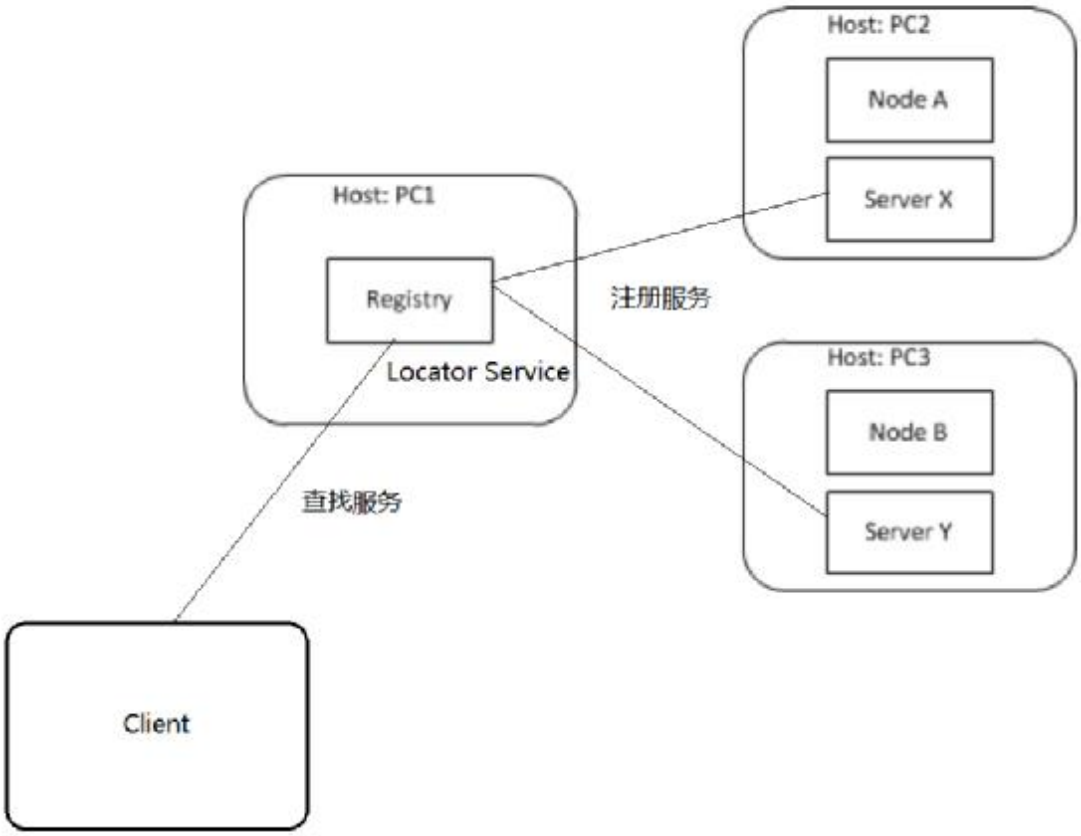
开始学习 IceGrid 之前，让我们先回顾下 Ice 以及 Icebox 存在的问题吧：

- 每个服务都要绑定不同的 Endpoint，就算是在一个 Icebox 里，客户端访问的时候还是要记住每个服务的 Endpoint 地址，才能访问。
- 若有两个 Icebox 做集群，需要客户端自己来实现负载均衡算法

为了解决上述问题，我们需要有一个“服务定位(Servant Locator)”的组件供客户端来查找一个服务所有可用的 Endpoints，最后这个服务定位组件还能顺便提供以下便利：

- 自动实现多种可选择的负载均衡算法，客户端无需再自己实现
- 服务的部署位置和部署数量发生变化后，客户端无限重启，自动感知和适应

上述第二点的确有点难度，需要客户端 API 透明的与服务定位组件进行互动，当服务的 Endpoint 发生变化后，及时反馈给客户端。我们来看看 IceGrid 怎么实现的吧，下图是其框架原理图：



IceGrid 的注册表 Registry 同时实现了 Ice 的服务查询接口——Locator，与大多数常规的实现有所不同，它返回的不是包含服务访问地址的直接代理对象 (Direct Proxy)，而是返

回一个间接代理 (Indirect Proxy)，客户端第一次访问服务的时候，需要通过 Registry 再次查询此代理具体对应的服务的 Endpoint 信息，然后建立起服务之间的直接 TCP 通道，完成具体的服务调用逻辑。这样设计以后，IceGrid 就满足了之前我们提到的两个特性，而且还实现了下面场景的巧妙优化：

当客户端首先发起 node1 上的服务 A 的请求后，接下来访问服务 B 的时候，若 B 恰好也在 node1 上存在，则 Registry 可以将 node1 上的服务 B 的 Endpoint 返回给客户，这样就省去了新连接的建立，并且优化了整体流程的响应时间。

架构设计是一门艺术还是一门技术，或者是一门哲学？当你看到很多种不同的设计，看似矛盾而又完美的解决了他们各自所针对的业务领域的特定问题时，你会怎么思考？代码量不代表着实力，你不去研究代码本身，不去思考，没有经历矛盾和不断的艰难选择过程，你恐怕只能改行了。

IceGrid 的 Registry 采用文件方式存放服务注册信息，可以实现主从 HA，其从节点也同时可以担负服务查询的功能，下面是一个典型的 Registry 进程的配置文件（registry.cfg）：

```
#registry config for icegrid
IceGrid.Registry.Client.Endpoints=tcp -p 4061
IceGrid.Registry.Server.Endpoints=tcp
IceGrid.Registry.Internal.Endpoints=tcp
IceGrid.Registry.AdminPermissionsVerifier=IceGrid/NullPermissionsVerifier
IceGrid.Registry.Data=./registry
IceGrid.Registry.DynamicRegistration=1
```

Registry 进程不依赖其他进程，相反每个 Node 上的服务 IceGrid 进程以及客户端都需要跟 Registry 通信，他们的配置文件中需要定义 Registry 的 Service Locator 访问地址：

Ice.Default.Locator=IceGrid/Locator:tcp -h registryhost -p 4061

启动 Icegrid 的 Regsitry 进程，采用如下命令，可以采用 registry.cfg 的绝对路径：

icegridregistry --Ice.Config=registry.cfg

由于 IceGrid 要求（其实也是 Ice 的要求）每个 ObjectAdapter 是唯一的，因此发布到 IceGrid 的 ObjectAdapter 需要手工指定唯一的 Id 值，客户端则通过 Service Locator 服务用“serviceId@adapterId”的名字来定位一个服务的 Indirect Proxy，随后再通过 Regsitry 二次查询此服务的具体可用的 Endpoint，完成点对点的通讯过程。下面我们修改之前的 Icebox 配置文件（config.properties），增加下面的内容，重新启动 icebox，就完成了服务注册过程。

Ice.Default.Locator=IceGrid/Locator:tcp -h localhost -p 4061

OnlineBook.AdapterId=OnlineBookAdapter

SMSService.AdapterId=SMSServicesAdapter

下面是服务注册过程中相关的日志输出：

OnlineBook: Network: trying to establish tcp connection to 127.0.0.1:4061

OnlineBook: Locator: updated object adapter `OnlineBookAdapter' endpoints with the locator registry

接下来，修改客户端，使用 Service Locater 服务完成服务查找：

String[] initParams = new String[] { "--Ice.Default.Locator=IceGrid/Locator:tcp -h localhost -p 4061" };

ic = Ice.Util.initialize(initParams);

Ice.ObjectPrx base = ic.stringToProxy("SMSService@SMSServicesAdapter");

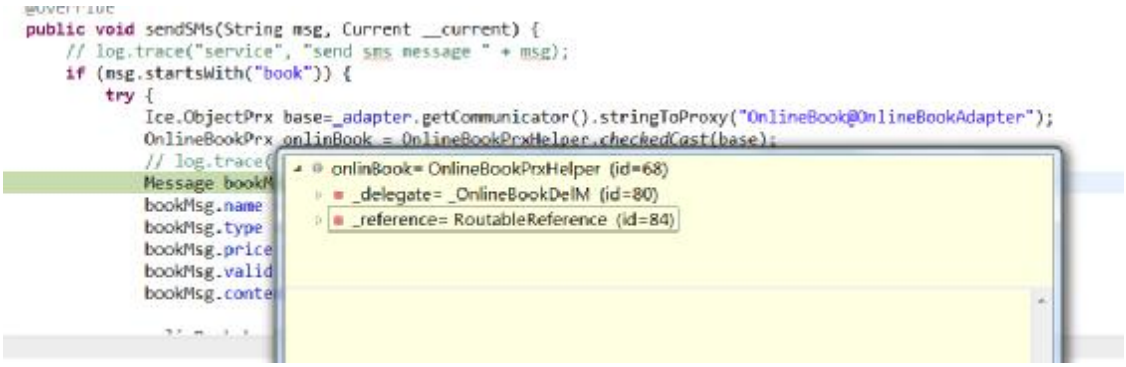
SMSServicePrx smsSrvPrx = SMSServicePrxHelper.checkedCast(base);

同样的，服务调用服务的代码也修改如下：

Ice.ObjectPrx
base=_adapter.getCommunicator().stringToProxy("OnlineBook@OnlineBookAdapter");

OnlineBookPrx onlinBook = OnlineBookPrxHelper.checkedCast(base);

若 Debug Server，你会发现获取到的这个代理是一个“远程代理”对象

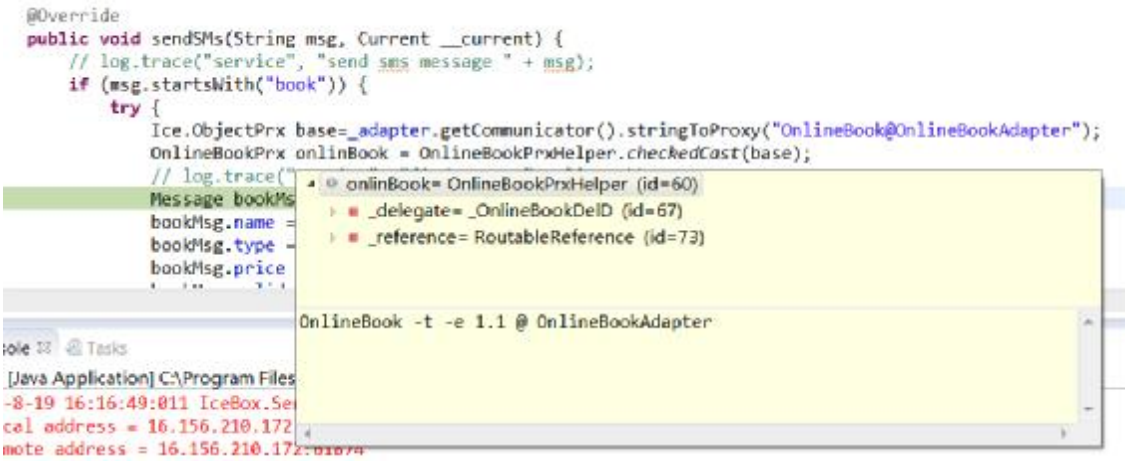


由于这两个服务是相互调用，并且在一个 Icebox 中部署，因此可以优化为本地调用，只要在配置文件中增加下面的配置：

IceBox.UseSharedCommunicator.OnlineBook=1

IceBox.UseSharedCommunicator.SMSService=1

重启 Icebox，进行断点调试，发现已经改用“本地服务代理”了：



顺便提一下 IceGrid 的管理命令行工具——icegridadmin，可以用来部署发布服务到注册表、进行基本的管理，用法如下：

icegridadmin -u test -p test --Ice.Default.Locator="IceGrid/Locator:tcp -h localhost -p 4061"

登录进去后，出现如下的交互式界面：

```
C:\Users\wuzhih>icegridadmin  --Ice.Default.Locator="IceGrid/Locator:tcp -h localhost -p 4061"
user id: user
password:
Ice 3.5.1  Copyright (c) 2003-2013 ZeroC, Inc.
>>> help;
help                        Print this message.
exit, quit                  Exit this program.
CATEGORY help               Print the help section of the given CATEGORY.
COMMAND help                Print the help of the given COMMAND.

List of help categories:

  application: commands to manage applications
  node: commands to manage nodes
  registry: commands to manage registries
  server: commands to manage servers
  service: commands to manage services
  adapter: commands to manage adapters
  object: commands to manage objects
  server template: commands to manage server templates
  service template: commands to manage service templates

>>>
```

每个命令后面都可以跟 help，来显示具体的帮助，如 server help，以下是几个命令的解释：

- node list，查看 IceGrid 节点
- server list ，查看 IceGrid server 列表
- service list xxx,查看某个 IceGrid
- registry list，查看 registry 节点
- adapter list，查看注册的 object adapter
- adapter endpoint xxxx，查看某个 adapter 上的 endpoint 列表

目前这种方式运行的话 node list 、server list 为空，因为我们的 IceGrid 为动态注册 Adapter 并且没有定义 IceGrid 引入的 node，server ,service 等 IceGrid 相关的组件，除了刚才例子所展示的 IceGrid 的注册表，IceGrid 还定义了一个类似 J2EE 的 XML 文件，用来描述一个业务系统（Application）的服务（Service）在哪些服务器（Node）上部署，这个文件可以被 IceGrid 用来实现应用的发布管理，其实质是将 XML 中描述的组件注册到 Registery 中，下面是一个具体的例子：

```
<icegrid>
    <application name="MyPowerICE">
        <node name="node1">
            <icebox id="IceBox1" activation="on-demand" exe="java">
                <!-- <option>-server</option> -->
                <option>IceBox.Server</option>
                <env>CLASSPATH=C:\ZeroC\Ice-3.5.1\lib\*;C:\project\TestIce\bin</env>
                <service name="OnlineBook" entry="com.hp.tel.serviceimp.BoxOnlineBook">
                    <adapter name="OnlineBook" id="OnlineBook" endpoints="default" />
                </service>
            </icebox>
        </node>
    </application>
</icegrid>
```

上述定义有如下要求：

- node1 名称不能重复，一个 node 里的 Icebox 名称不能重复，一个 Icebox 里面的 service 名称不能重复，并且 service 名称和 endpoint 的 name 和 ID 要相同。
- serverant 的类需要打包放在指定的目录，在 classpath 中能访问，如上述的 C:\project\TestIce\bin。

接下来，我们创建名称为 node1 的 Icegridnode 的运行期配置文件（node1.cfg ）如下：

```
#指定主注册节点的位置
Ice.Default.Locator=IceGrid/Locator:tcp -h localhost -p 4061
```

#设置节点 1 相关数据的存储目录

IceGrid.Node.Data=.\node\data

#指定节点 1 用于监听客户端连接的端口号

IceGrid.Node.Endpoints=tcp -p 5062

IceGrid.Node.Name=node1

#指定节点 1 的名称

Ice.StdErr=.\node\node.stderr.log

#指定错误日志文件

Ice.StdOut=.\node\node.stdout.log

定义好配置文件后，命令行启动 node1：

icegridnode --Ice.Config=node1.cfg

启动成功以后，在 icegridadmin 里运行命令进行应用部署，登录 icegridadmin 后，执行命令 application add icegrid.xml（或者 application update icegrid.xml 进行更新），IceBox 的" activation="on-demand" 属性会自动启动服务，但未来验证部署的正确性，可以用命令 server start IceBox1 看看是否启动成功（前提是 对应的 gridnode 进程启动）。

>>> server start IceBox1

服务的日志在.\node\node.stderr.log 下，若服务启动失败，则可以查看日志来分析出错原因，通过执行下述命令来看当前 Grid 里的 ObjectAdapter 以及上面捆绑的 Endpoints:

>>> adapter list

OnlineBook

>>> adapter endpoints OnlineBook

dummy -t -e 1.1:tcp -h 16.156.210.172 -p 64039:tcp -h 192.168.56.1 -p 64039

icegrid 服务部署启动成功以后，就可以客户端访问了，与之前的方式一样，被访问的服务 ID 为"service@adapterid"：

Ice.ObjectPrx base = ic.stringToProxy("OnlineBook@OnlineBook");

另外，通过 XML 来发布服务以后，服务实际上是在某个具体的 gridnode 的 data 目录下生成对应的 icebox 配置文件，以这里的例子为例，node1 的 data 路径为.\node\data，IceBox1 相关的文件生成在目录 data\servers\IceBox1 下，其中 config 子目录存放 icebox 的配置文件，我们打开 config 子目录，里面有两个文件：config 以及 config_OnlineBook，内容分别如下：

```
# Configuration file (08/20/14 17:59:32.417)
# Server configuration
Ice.Admin.ServerId=IceBox1
Ice.Admin.Endpoints=tcp -h localhost
Ice.ProgramName=IceBox1
IceBox.Service.OnlineBook=com.hp.tel.serviceimp.BoxOnlineBook --Ice.Config='c:\project\TestIce/.\node\data/servers/IceBox1/config/config_OnlineBook'
IceBox.InstanceName=IceBox1
IceBox.LoadOrder=OnlineBook
Ice.Default.Locator=IceGrid/Locator:tcp -h localhost -p 4061
```

config_OnlineBook 的内容如下：

```
# Configuration file (08/20/14 17:59:32.417)
# Object adapter OnlineBook
OnlineBook.Endpoints=default
OnlineBook.AdapterId=OnlineBook
Ice.Default.Locator=IceGrid/Locator:tcp -h localhost -p 4061
```

仔细看看这两个文件，是不是与之前我们写的 icebox 配置文件一样？需要注意的是，icegridnode 的 data 目录不能被删除，这里实际上是其运行期的配置文件，若使用了 ice 的数据库组件，则数据库的相关信息也在这里、另外，若通过 icegrid 的 IcePatch2 工具实现应用系统程序包的自动下载安装，则具体的程序代码也在 data 目录下。因此 ice 官方建议是备份 data 目录，经测试，停止 node 的情况下，将 data/servers 目录删除，然后重新启动 node，客户端访问该 node 上的服务的时候，又会自动生成对应的配置文件，服务可以正常访问。

使用 IceGrid 以后，我们就具备了部署分布式 Ice 服务实例的能力，为了将多个 Ice 服务实例“汇聚”在一起，对外提供一组服务，Ice 引入了 Service Replication 的概念：将一组相同接口的服务实例定义在一个 Replication 组中，并给这个组起一个唯一的名字，客户端就用中这个 Replication 的名字来定位服务，从而实现灵活的负载均衡和服务容错机制，下面的定义给出了一个名为 ReplicatedAdapter 的 Replication 组，这个组里定义了一个 Well-known 的 Ice Object，名字为 OnlineBook，类型为::book::OnlineBook，服务里的 OnlineBook 这个 Adapter 绑定在这个 Replication 组上（replica-group="ReplicatedAdapter"），负载均衡采用 adaptive 算法，即获取当前系统负载最低的一个机器（linux system load）。因此，我们可以用 OnlineBook 这个 Well-known 的 Ice Object 来替代 OnlineBook @ReplicatedAdapter 并用于客户端访问。

```
<icegrid>
```

```
<application name="MyPowerICE">
<replica-group id="ReplicatedAdapter">
<load-balancing type="adaptive"/>
<object identity="OnlineBook" type="::book::OnlineBook" />
</replica-group>
<node name="node1">
<icebox id="IceBox1" activation="on-demand" exe="java">
<!-- <option>-server</option> -->
<option>IceBox.Server</option>
<env>CLASSPATH=C:\ZeroC\Ice-3.5.1\lib\*;C:\project\TestIce\bin</env>
<service name="OnlineBook" entry="com.hp.tel.serviceimp.BoxOnlineBook">
<adapter name="OnlineBook" id="OnlineBook" endpoints="default" replica-group="ReplicatedAdapter"/>
</service>
</icebox>
</node>
</application>
</icegrid>
```

通过 icegridadmin 命令行更新 grid.xml 并查看 adapter，看到增加了我们刚定义的 ReplicatedAdapter：

```
>>> application update icegrid.xml
>>> adapter list;
OnlineBook
ReplicatedAdapter
>>> adapter list;
OnlineBook
ReplicatedAdapter
```

此时，客户端访问服务 OnlineBook 的代码改为如下的写法：

```
Ice.ObjectPrx base = ic.stringToProxy("OnlineBook ");
```

接下来我们看看服务模板的问题，对电信软件开发比较熟悉的同学应该不陌生“模板”这个词，模板是有一组参数的通用定义文件，为了快速批量定义类似的一组服务参数。由于一个 icegrid 应用中常常有几十个服务，这些服务的定义基本上类似：定义服务名称，实现类名称、adapterName、adapterId、java 启动类、启动方式、以及 Replicated 策略等，因此我们通常定义一个或几个服务模板，然后相应的具体服务套用此模板定义，则配置工作的负担也大大减轻，grid xml 文件的可维护性大大增强，下面我们定义个包含 OnlineBook 与 SMSService 两个服务在一个 IceBox 中的 Ice Server 模板：

```
<server-template id="HelloServerTemp">
    <parameter name="name" />
    <icebox id="${name}" exe="java" activation="on-demand">
    <option>IceBox.Server</option>
        <env>CLASSPATH=C:\ZeroC\Ice-3.5.1\lib\*;C:\project\TestIce\bin</env>
        <service name="OnlineBook" entry="com.hp.tel.serviceimp.BoxOnlineBook">
         <adapter name="OnlineBook" id="OnlineBook" endpoints="default" replica-group="ReplicatedAdapter"/>
        </service>
    <service name="SMSService" entry="com.hp.tel.serviceimp.SMSServiceI">
         <adapter name="SMSService" id="SMSService" endpoints="default" replica-group="ReplicatedAdapter"/>
        </service>
   </icebox>
</server-template>
```

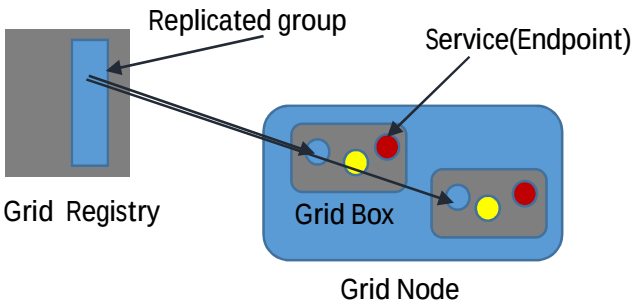
然后，icegrid.xml 里面就可以引用上述 Server 模板来创建几个 node 实例（完整的 xml 在 icegrid2.xml 里）。

```
<node name="node1">
      <server-instance template="HelloServerTemp" name="hellowserver1" />
</node>
<node name="node2">
      <server-instance template="HelloServerTemp" name="hellowserver2" />
</node>
```

你可以部署这个 xml 文件，然后看看效果。这里顺便说下 IceGrid Xml 的欺骗性，从它的 XML 结构来看，是 Application-> replica-group->node->server，但实际上，运行期间是没有这个层次关系的，replica-group 在一个注册表里是不能重复的，adapter 也是不能重复的，Well-known Object 也都不能重复，这点可以从 icegridadmin 的命令可以看出，server list,adapter list ,object list 这些命令都没有"application"参数，因此，我们其实是无法分离不同 application 的这些组件信息，只能从名字上加以限定区分，或者采用多个 Grid 注册表的多个 Grid 系统，但那样又造成了彼此隔离的单独系统，因此，最好的办法是服务的名字上面增加模块前缀，用于区分，如下所示：

```
<replica-group id="ReplicatedAdapter">
<object identity="book_OnlineBook" type="::book::OnlineBook" />
<object identity="message_SMSService" type="::message::SMSService" />
</replica-group>
```

结束本节之前，我们用示意图的方式展示 Ice Core、Icebox、Icegrid 之间的关系，如下图所示：



应用部署

对于具有一定规模的分布式系统，除了开发工作之外，应用的部署和升级都是工作量比较大而且容易出错的环节，另外这种大规模系统往往还存在着开发验证环境、集成测试环境、用户验收环境、线上环境等多个环境并存的情况，单靠手工方式去部署和升级应用，在很多时候，将是一个不能承受的工作，好在 Ice Grid 提供了解决此类问题的好工具——IcePatch2。

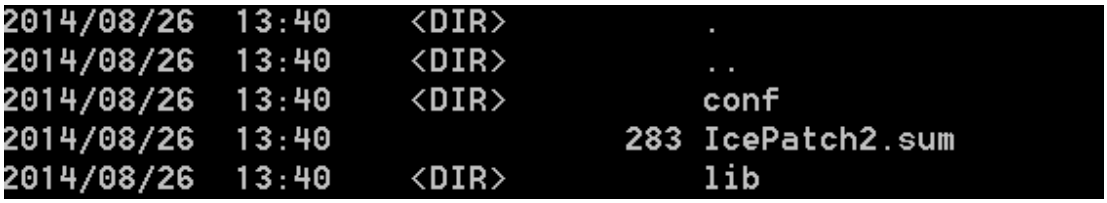
IcePatch2 属于 ice-utils 程序包里的一个工具，它的原理是提供一个类似 FTP Server 的程序部署包远程仓库，仓库中的部署包文件被压缩并计算文件的 checksum 值，压缩是为了高效传输文件到客户端，checksum 是为了查找发生变化的文件并进行 Patch。客户端工具 icepatch2client 则通过连接 icepatch2 server 服务器进行部署包的下载和更新。对于客户端存在而 Server 上不存在的文件（文件夹），IcePatch2 默认不会删除，但若执行彻底的 pactch 命令（-t 参数 thorough）则会完全同步客户端与服务器上的程序文件而删除，为了避免删除，则可以设置参数 IcePatch2Client.Remove=0 来实现。

IcePatch2 工具的使用步骤如下：

- 先在某个服务器上建立 ICE 应用程序的运行时期目录结构，包括 Jar 文件、配置文件等部署文件
- 运行 icepatch2calc，计算上述目录的文件签名并进行压缩
- 启动 icepatch2server，进行服务
- 在各个客户端计算节点上启动 icepatch2client，从 server 上进行同步。

举例如下，在 C:\test-server-data 目录下建立 lib 与 conf 文件，分别存放 Jar 程序与配置文件，

运行命令：C:\test-server-data>icepatch2calc .（后面的“.” 参数表示当前目录），看到生成.sum 文件以及压缩文件，如下图所示：

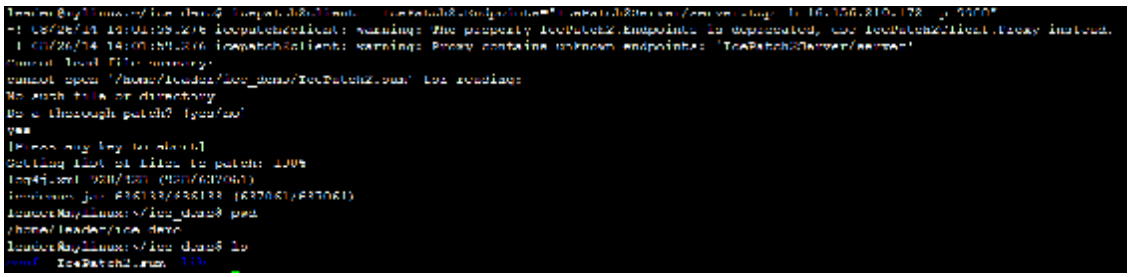


icepatch2calc 执行成功以后，运行下述命令启动 icepatch2server，注意参数的引号：

icepatch2server --IcePatch2.Directory="C:\test-server-data" --IcePatch2.Endpoints="tcp -p 9988"

启动成功以后，在另外一个机器上输入命令：icepatch2client，

icepatch2client --IcePatch2.Endpoints="IcePatch2Server/server:tcp -h 16.156.210.172 -p 9988"



可以看到客户端进行了全同步，server 上的所有文件都被下载了，包括 sum 文件，与 Server 上不同的是，客户端这里没有压缩文件。

可以制作一个简单的脚本，自动登录到各个客户机上执行上述命令，完成 Icegrid 集群的统一升级，简化人工操作。