



用户手册

1.3

(C#版)

目录

前言	1
本章提要	1
欢迎使用 Hprose	2
体例	3
菜单描述	3
屏幕截图	3
代码范例	3
运行结果	3
获取帮助	3
电子文档	3
在线支持	3
联系我们	3
第一章 快速入门	5
本章提要	5
安装 Hprose for C#	6
安装方法	6
二进制方式	6
源码方式	6
创建 Hprose 的 Hello 服务器	7
创建 Hprose 的 Hello 客户端	9
通过 Invoke 方法动态调用	10
通过接口方式调用	11
第二章 类型映射	13
本章提要	13
基本类型	14
值类型	14
引用类型	14
基本类型的映射	15
序列化类型映射	15
反序列化默认类型映射	15
反序列化有效类型映射	16
容器类型	17
列表类型	17
序列化类型映射	17
反序列化类型映射	17
字典类型	18

序列化类型映射	18
反序列化类型映射	18
对象类型	18
通过 ClassManager 来注册自定义类型.....	19
第三章 服务器	20
本章提要	20
通过独立服务器方式发布服务	21
发布实例方法	21
Methods 属性	26
AddMethod 方法	26
AddMethods 方法	26
AddInstanceMethods 方法	26
AddStaticMethods 方法	26
AddMissingMethod 方法	27
隐藏发布列表	27
调试开关	27
对象序列化模式	28
P3P 开关	28
跨域开关	28
服务器事件	29
OnBeforeInvoke 事件	29
OnAfterInvoke 事件	29
OnSendHeader 事件	30
OnSendError 事件	30
存取环境上下文	30
通过 aspx 方式发布服务	30
按全局发布方法	30
按会话发布方法	33
按请求发布方法	33
存取环境上下文	34
通过自定义 HTTP 处理程序来发布服务	35
第四章 客户端	36
本章提要	36
同步调用	37
通过 Invoke 方法进行同步调用	37
带名称空间 (别名前缀) 方法	37
可变的参数和结果类型	37
引用参数传递	38

自定义类型的传输	39
通过代理接口进行同步调用	40
接口定义	40
带名称空间 (别名前缀) 方法	41
可变的参数和结果类型	41
泛型参数和引用参数传递	42
自定义类型	42
异步调用	43
通过 Invoke 方法进行异步调用	43
通过代理接口进行异步调用	45
异常处理	46
同步调用异常处理	46
异步调用异常处理	47
超时设置	48
HTTP 参数设置	48
代理服务器	48
持久连接	48
HTTP 标头	48
调用结果返回模式	49
Serialized 模式	49
Raw 模式	49
RawWithEndTag 模式	49

前言

在开始使用 Hprose 开发应用程序前 ,您需要先了解一些相关信息。本章将为您提供这些信息 ,并告诉您如何获取更多的帮助。

本章提要

- 欢迎使用 Hprose
- 体例
- 获取帮助
- 联系我们

欢迎使用 Hprose

您还在为 Ajax 跨域问题而头疼吗？

您还在为 WebService 的低效而苦恼吗？

您还在为选择 C/S 还是 B/S 而犹豫不决吗？

您还在为桌面应用向手机网络应用移植而忧虑吗？

您还在为如何进行多语言跨平台的系统集成而烦闷吗？

您还在为传统分布式系统开发的效率低下运行不稳而痛苦吗？

好了，现在您有了 Hprose，上面的一切问题都不再是问题！

Hprose (High Performance Remote Object Service Engine) 是一个商业开源的新型轻量级跨语言跨平台的面向对象的高性能远程动态通讯中间件。它支持众多语言，例如.NET, Java, Delphi, Objective-C, ActionScript, JavaScript, ASP, PHP, Python, Ruby, C++, Perl 等语言，通过 Hprose 可以在这些语言之间实现方便且高效的互通。

Hprose 使您能高效便捷的创建出功能强大的跨语言，跨平台，分布式应用系统。如果您刚接触网络编程，您会发现用 Hprose 来实现分布式系统易学易用。如果您是一位有经验的程序员，您会发现它是一个功能强大的通讯协议和开发包。有了它，您在任何情况下，都能在更短的时间内完成更多的工作。

Hprose 是 PHRPC 的进化版本，它除了拥有 PHRPC 的各种优点之外，它还具有更多特色功能。Hprose 使用更好的方式来表示数据，在更加节省空间的同时，可以表示更多的数据类型，解析效率也更加高效。在数据传输上，Hprose 以更直接的方式来传输数据，不再需要二次编码，可以直接进行流式读写，效率更高。在远程调用过程中，数据直接被还原为目标类型，不再需要类型转换，效率上再次得到提高。Hprose 不仅具有在 HTTP 协议之上工作的版本，以后还会推出直接在 TCP 协议之上工作的版本。Hprose 在易用性方面也有很大的进步，您几乎不需要花什么时间就能立刻掌握它。

Hprose 与其它远程调用商业产品的区别很明显——Hprose 是开源的，您可以在相应的授权下获得源代码，这样您就可以在遇到问题时更快的找到问题并修复它，或者在您无法直接修复的情况下，更准确的将错误描述给我们，由我们来帮您更快的解决它。您还可以将您所修改的更加完美的代码或者由您所增加的某个激动人心的功能反馈给我们，让我们能够更好的来一起完善它。正是因为有这种机制的存在，您在使用该产品时，实际上可能遇到的问题会更少，因为问题可能已经被他人修复了。

Hprose 与其它远程调用开源产品的区别更加明显，Hprose 不仅仅在开发运行效率，易用性，跨平台和跨语言的能力上较其它开源产品有着明显的不可取代的综合优势，Hprose 还可以保证所有语言的实现具有一致性，而不会向其他开源产品那样即使是同一个通讯协议的不同实现都无法保证良好的互通。而且 Hprose 具有完善的商业支持，可以在任何时候为您提供所需的帮助。不会向其它没有商业支持的开源软件那样，当您遇到问题时只能通过阅读天书般的源代码的方式来解决。

Hprose 支持许多种语言，包括您所常用的、不常用的甚至从来不用的语言。您不需要掌握 Hprose 支持的所有语言，您只需要掌握您所使用的语言就可以开始启程了。

本手册中有些内容可能在其它语言版本的手册中也会看到，我们之所以会在不同语言的手册中重复这些内容是因为我们希望您只需要一本手册就可以掌握 Hprose 在这种语言下的使用，而不需要同时翻阅几本书才能有一个全面的认识。

接下来我们就可以开始 Hprose 之旅啦，不过在正式开始之前，先让我们对本文档的编排方式以及如何获得更多帮助作一下说明。当然，如果您对下列内容不感兴趣的话，可以直接跳过下面的部分。

体例

菜单描述

当让您选取菜单项时，菜单的名称将显示在最前面，接着是一个箭头，然后是菜单项的名称和快捷键。例如“文件→退出”意思是“选择文件菜单的退出命令”。

屏幕截图

Hprose 是跨平台的，支持多个操作系统下的多个开发环境，因此文档中可能混合有多个系统上的截图。

代码范例

代码范例将被放在细边框的方框中：

```
public class Hello {  
    public static void main(String[] args) {  
        System.out.println("Hello Hprose!");  
    }  
}
```

运行结果

运行结果将被放在粗边框的方框中：

```
Hello Hprose!
```

获取帮助

电子文档

您可以从我们的网站 <http://www.hprose.com/documents.php> 上下载所有的 Hprose 用户手册电子版，这些文档都是 PDF 格式的。

在线支持

我们的技术支持网页为 <http://www.hprose.com/support.php>。您可以在该页面找到关于技术支持的相关信息。

联系我们

如果您需要直接跟我们取得联系，可以使用下列方法：

公司名称	北京蓝慕威科技有限公司
公司地址	北京市海淀区马连洼东馨园 2-2-101 号
电子邮件	市场及大型项目合作 : manager@hprfc.com 产品购买及项目定制 : sales@hprfc.com 技术支持 : support@hprfc.com
联系电话	+86-15010851086 (周一至周五 , 北京时间早上 9 点到下午 5 点)

第一章 快速入门

使用 Hprose 制作一个简单的分布式应用程序只需要几分钟的时间 ,本章将用一个简单但完整的实例来带您快速浏览使用 Hprose for C#进行分布式程序开发的全过程。

本章提要

- 安装 Hprose for C#
- 创建 Hprose 的 Hello 服务器
- 创建 Hprose 的 Hello 客户端

安装 Hprose for C#

Hprose for .NET 全部代码使用 C#编写 ,透过 CLR 无缝支持.NET 其它语言 ,例如 :Visual Basic.NET、Visual C++等。

所有 .NET Framework 版本 (1.0, 1.1, 2.0, 3.0, 3.5, 4.0, 4.5) 全部支持。

所有 .NET Compact Framework 版本 (1.0, 2.0, 3.5) 全部支持。

支持 SilverLight 2.0, 3.0, 4.0, 5.0 所有版本。

支持微软最新的云计算平台 Windows Azure Platform。

支持 Windows Phone 所有版本。

支持 Mono。

安装方法

二进制方式

Hprose for C#提供了可以直接使用的二进制文件，针对不同版本的.NET 都有相应的编译好的版本。

.NET Framework 1.0~3.5 的二进制版本包含有 System.Numerics.dll、Hprose.dll 和 Hprose.Client.dll 这三个文件。而.NET Framework 4.0 的二进制版本只包含 Hprose.dll 和 Hprose.Client.dll 这两个文件。

其中 System.Numerics.dll 与.NET Framework 4.0 中内置的 System.Numerics.dll 基本上完全兼容，它是.NET Framework 4.0 中该文件在其它版本中的移植。Hprose 依赖于该文件，所以在使用 Hprose 时，该文件必须要被项目工程引用。

Hprose.dll 是 Hprose for C# 的完整版本，Hprose.Client.dll 是 Hprose for C# 的客户端版本。Hprose.dll 包含了 Hprose.Client.dll 中的所有内容，另外还包含了服务器端的组件。在您使用过程中，您不应该在同一个项目工程中同时引用这两个 dll。通常在只使用客户端功能时，选择 Hprose.Client.dll。需要服务器功能时，才需要选择 Hprose.dll。只有 Hprose for C#企业版中才包含 Hprose.dll。

.NET Compact Framework、Silverlight 和 Windows Phone 版本中只包含有 System.Numerics.dll 和 Hprose.Client.dll 这两个文件，在使用时，这两个文件都需要被引入您的项目工程中。

源码方式

Hprose for C#不是使用 Visual Studio 开发的，它的所有.NET 版本的源码只有一套，但是对不同的.NET 版本进行了针对性的优化，根据不同的编译条件，生成不同的版本。源码目录下的 make.bat 是用来编译所有版本的批处理文件。它直接调用 C#编译器来完成编译。所以，您即使仅仅安装了.NET Framework 运行时，也可以通过该批处理程序完成相应版本的编译，因为 C#编译器在.NET Framework 运行时中已经包含，所以您无需安装.NET Framework SDK。但对于.NET Compact Framework，SilverLight、Windows Phone 和 Mono，您只有安装相应的 SDK 之后才能够编译。

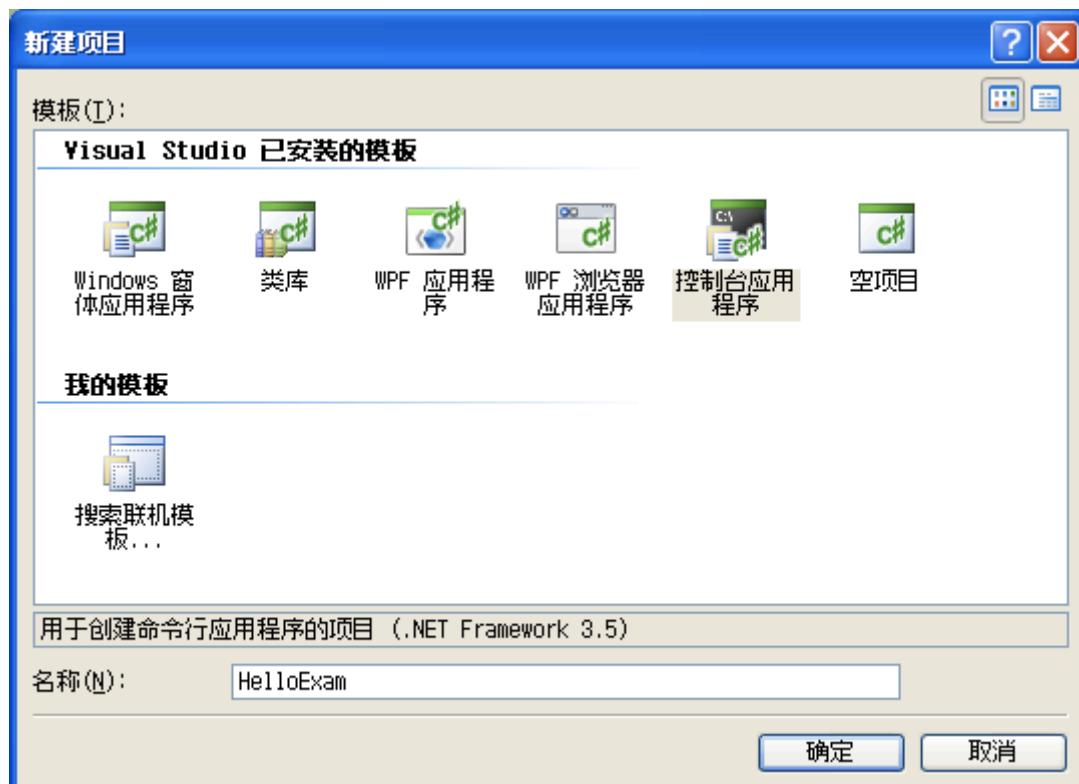
如果您不希望使用 dll，可以直接将源码添加到您的工程当中，在编译时，参考 make.bat 中的相关内容，在您的工程中设置好条件编译符号和编译选项即可。

Hprose for C#同样支持 Mono，但因为 Mono 本身存在很多问题，并不稳定，所以在 Mono 中使用 Hprose for C#可能会在某些特殊情况下遇到一些奇怪的问题。所以，我们不建议您使用 Mono。但如果一定要坚持使用 Mono，并在使用过程中也确实遇到了问题，请将问题反馈给 Mono 开发团队，由他们负责帮您解决。我们对您在 Mono 下遇到的任何问题（除非该问题在.NET Framework 下也同样存在），不提供技术支持。

创建 Hprose 的 Hello 服务器

我们以 Visual C# 2008 Express Edition 作为开发环境为例，来介绍一下如何创建一个 Hprose 服务器，按照传统惯例，都是以 Hello World 为例来作为开始的，我们这里稍稍做一下改变，我们创建的服务器将发布一个 sayHello 方法，这样客户端就可以调用它来对任何事物说 Hello 啦。

首先启动 Visual C# 2008 Express Edition 开发环境，打开菜单的“文件→新建项目”，选择“控制台应用程序”，然后填写好名称之后，点击确定：

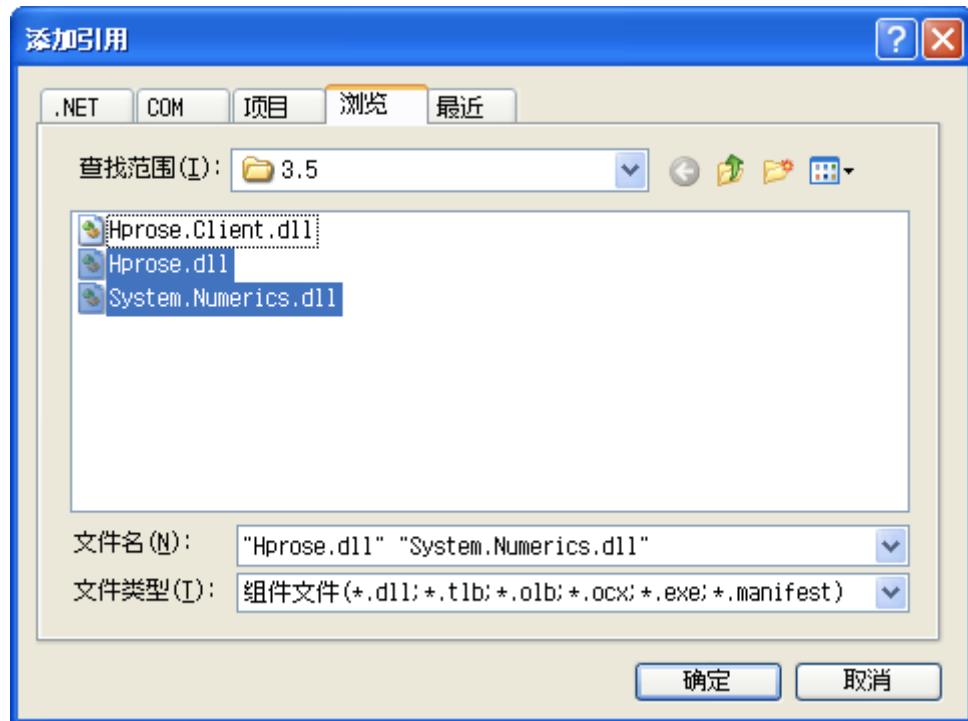


接下来，在“解决方案资源管理器”中，选择新建的项目，将其改名为 HelloServer。

项目引用中，除了 System 以外，其它引用皆可删除。然后在它的引用上点击右键，选择添加引用：



在打开的“添加引用”对话框中，选择浏览，之后选择到 Hprose 对应版本的二进制文件所在的目录下，选择添加 Hprose.dll 和 System.Numerics.dll：



接下来，我们就可以编写代码了：

```
using System;
using Hprose.Server;

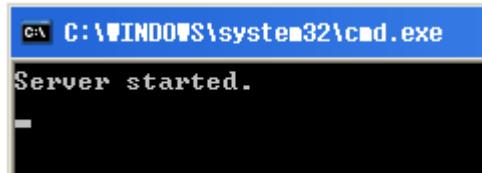
namespace HelloExam {
    class Hello {
        public string sayHello(string name) {
            return "Hello " + name + "!";
        }
    }
    class Program {
        static void Main(string[] args) {
            HproseHttpListenerServer server = new HproseHttpListenerServer("http://localhost:2010/");
            server.Methods.AddInstanceMethods(new Hello());
            server.Start();
            Console.WriteLine("Server started.");
            Console.ReadLine();
            Console.WriteLine("Server stoped.");
        }
    }
}
```

上面代码中，Hello 类提供了我们要发布的方法 sayHello。主程序前三行的作用分别是创建服务、发布服务、启动服务。后面的 ReadLine 的作用是保持程序不会立即退出，这样服务才能一直运行。程序运行后，一旦按回车键，程序就会立即终止，服务也会停止。

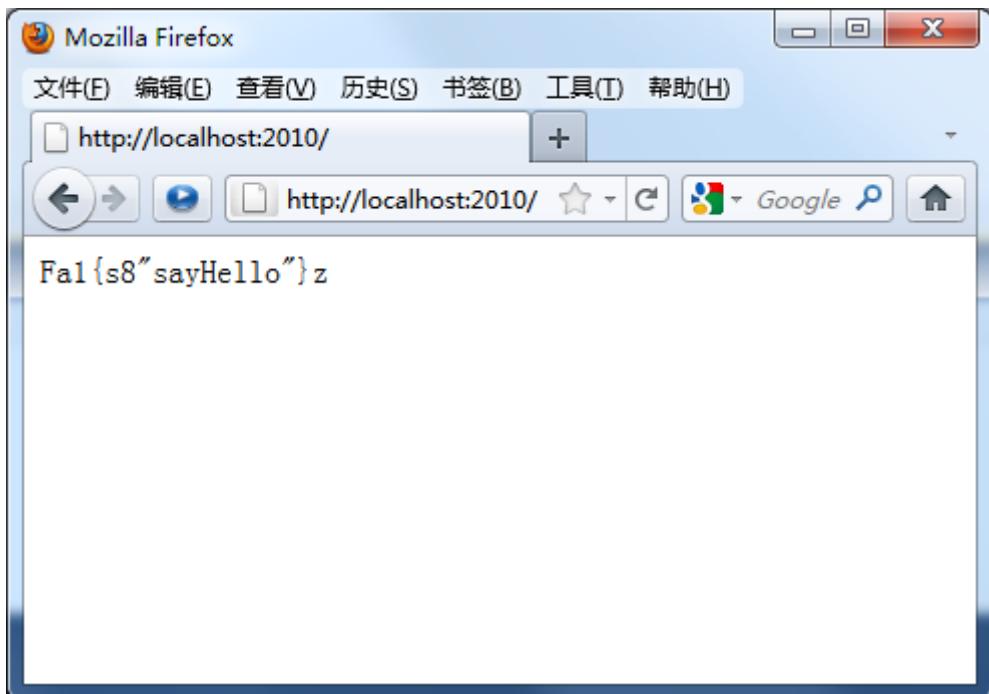
好了我们来看看效果吧，选择“调试→开始执行（不调试）”：



之后程序便会启动：



接着打开浏览器，在地址栏中输入：<http://localhost:2010/>，然后回车，如果看到如下页面就表示我们的服务发布成功啦。



接下来我们来看一下客户端如何创建吧。

创建 Hprose 的 Hello 客户端

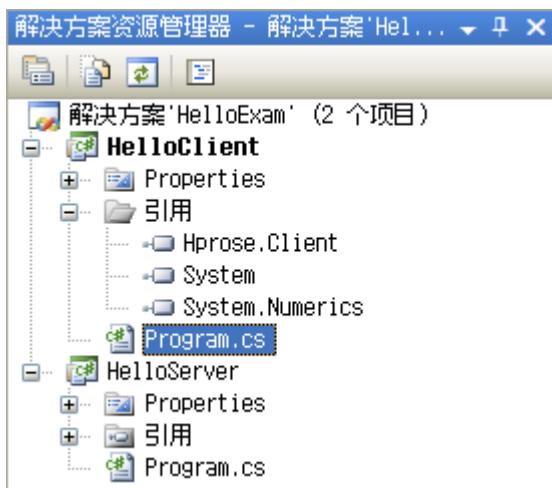
客户端我们以 C# 控制台程序为例，开发环境仍然为 Visual C# 2008 Express Edition。

客户端可以通过 Invoke 方法动态调用服务，也可以通过接口方式来调用，下面我们来分别介绍这两种方式。

通过 Invoke 方法动态调用

首先我们先来看看如何使用 Invoke 方法动态调用服务。

与上面创建服务器控制台应用程序步骤相同，客户端我们也是创建一个控制台应用程序，之后为其添加 Hprose.Client.dll 和 System.Numerics.dll 这两个引用：

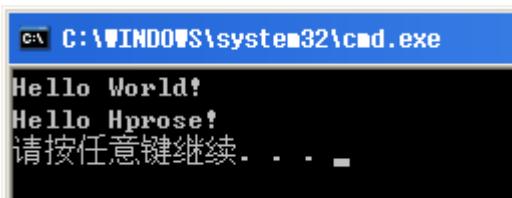


接下来开始编写 HelloClient 类的代码：

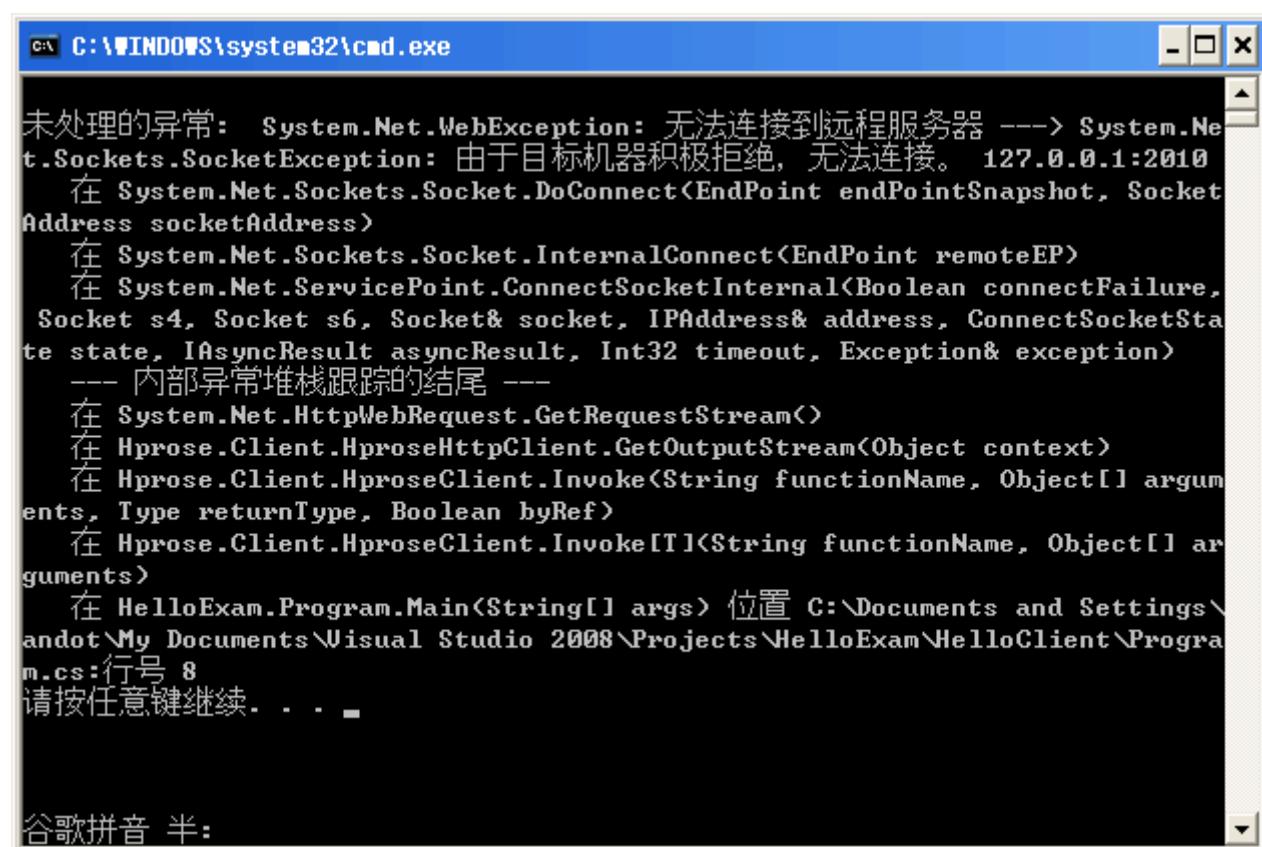
```
using System;
using Hprose.Client;

namespace HelloExam {
    class Program {
        static void Main(string[] args) {
            HproseHttpClient client = new HproseHttpClient("http://localhost:2010/");
            Console.WriteLine(client.Invoke<string>("sayHello", new object[] { "World" }));
            Console.WriteLine(client.Invoke("sayHello", new object[] { "Hprose" }));
        }
    }
}
```

上面代码很简单，主程序第一句创建了 Hprose 客户端，并且指定了服务器地址，第二句通过范型方法进行调用，范型方法 Invoke 中所指定的类型为返回结果的类型。第三句是通过非泛型方法调用，结果并不会自动转为 string，而是以 object 类型返回。但是因为 Console.WriteLine 支持 object 输出，所以这里我们不需要显式转型仍然可以正确输出。现在我们在确保服务器已经运行的情况下，启动该客户端程序，正常的话，您会看到如下结果：



如果您的服务器没有启动，您可能会看到如下的出错结果：



通过 `Invoke` 方法调用服务器方法很灵活，`Invoke` 方法具有多个重载，即使是对同一个服务器方法，您也可以通过指定不同的参数来获得不同类型的结果。后面我们会在详细介绍 Hprose 客户端时，再对 `Invoke` 方法作更详细的介绍。

但是您也会发现，通过 `Invoke` 调用不是那么的直观，参数需要自己写入数组，结果也需要自己转型，那么有没有方法可以向本地调用那样来进行远程调用呢？可以，那就是通过接口方式调用。

通过接口方式调用

我们来对上面的代码做一下小的改造，修改以后的代码如下：

```

using System;
using Hprose.Client;

namespace HelloExam {
    public interface IHello {
        string sayHello(string name);
    }

    class Program {
        static void Main(string[] args) {
            HproseHttpClient client = new HproseHttpClient("http://localhost:2010/");
            IHello hello = client.UseService<IHello>();
            Console.WriteLine(hello.sayHello("World"));
            Console.WriteLine(hello.sayHello("Hprose"));
        }
    }
}

```

```
    }  
}
```

现在代码行数虽然多了，但是在调用时，却方便了不少。

如果您曾经用过 WCF、.NET Remoting 或者其它的远程调用技术的话，您可能会惊讶的发现，Hprose 比那些远程调用技术要方便容易的多。这里的 IHello 接口在服务器端并没有实现，只是有相同的方法，但在客户端仍然可以直接通过接口进行调用。Hprose 就是这样灵活，您甚至可以定义不同于服务器实现的客户端接口，只要参数和结果类型是相容或者可以转换的类型，就可以正常的进行调用。所以在 Hprose 中，您不但可以通过接口方式来调用 C#服务器提供的服务，同样可以调用非 C#服务器提供的服务。



在 Hprose for C#中接口并不是用来约束服务器与客户端必须有一致方法签名的手段，而仅仅是用于实现客户端远程调用代理的工具。

第二章 类型映射

类型映射是 Hprose 的基础，正是因为 Hprose 设计有良好的类型映射机制，才使得多语言互通得以实现。本章将对 Hprose for C#的类型映射进行一个详细的介绍。

本章提要

- 基本类型
- 容器类型
- 对象类型

基本类型

值类型

类型	描述
整型	Hprose 中的整型为 32 位有符号整型数，表示范围是 -2147483648 ~ 2147483647 (-2 ³¹ ~ 2 ³¹ -1)。
长整型	Hprose 中的长整型为有符号无限长整型数，表示范围仅跟内存容量有关。
浮点型	Hprose 中的浮点型为双精度浮点型数。
非数	Hprose 中的非数表示浮点型数中的非数 (NaN)。
无穷大	Hprose 中的无穷大表示浮点型数中的正负无穷大数。
布尔型	Hprose 中的布尔型只有真假两个值。
字符	Hprose 中的 UTF8 编码的字符，仅支持单字元字符。
空	Hprose 中的空表示引用类型的值为空 (null)。
空串	Hprose 中的空串表示空字符串或零长度的二进制型。

其中非数和无穷大其实是特殊的浮点型数据，只不过在 Hprose 中它们有单独的表示方式，这样可以使它们占用更少的存储空间，并得到更快的解析。

另一个可能会引起您注意的是，这里把空和空串也作为值类型对待了。这里把它列为值类型而不是引用类型，是因为 Hprose 中的值类型和引用类型的概念与程序设计语言中的概念不完全相同。这里的值类型是表示在 Hprose 序列化过程中，不做引用计数的类型。在序列化过程中，当遇到相等的值类型时，后写入的值将与先写入的值保持相同的形式，而不是以引用的形式写入。

引用类型

类型	描述
二进制型	Hprose 中的二进制型表示二进制数据，例如字节数组或二进制字符串。
字符串型	Hprose 中的字符串型表示 Unicode 字符串数据，以标准 UTF-8 编码存储。
日期型	Hprose 中的日期型表示年、月、日，年份范围是 0 ~ 9999。
时间型	Hprose 中的时间型表示时、分、秒 (毫秒，微秒，毫微秒为可选部分)。
日期时间型	Hprose 中的日期时间型表示某天的某个时刻，可表示本地或 UTC 时间。
Guid 型	Hprose 中的 Guid 型表示全局唯一标识符。Guid 是一个 128 位整数 (16 字节)，可用于所有需要唯一标识符的计算机和网络。此标识符重复的可能性非常小。

空字符串和零长度的二进制型并不总是表示为空串类型，在某些情况下它们也表示为各自的引用类型。空串类型只是对二进制型和字符串型的特殊情况的一种优化表示。

引用类型在 Hprose 中有引用计数，在序列化过程中，当遇到相等的引用类型时，后写入的值是先前写入的值的引用编号。

后面介绍的容器类型和对象类型也都属于引用类型。

基本类型的映射

为了便于使用其它.NET 语言的用户，下面的类型映射介绍中我们使用.NET 类型，而不是 C#类型。

.NET 类型与 Hprose 类型的映射关系不是一一对应的。在序列化过程中可能会有多种.NET 类型对应同一种 Hprose 类型，在反序列化过程中还分为默认类型映射和有效类型映射，对于有效类型映射还分为安全类型映射和非安全类型映射两种。我们下面以列表的形式来说明。

序列化类型映射

.NET 类型	Hprose 类型
Byte , SByte , Int16 , UInt16 , Int32 , Enum	整型
UInt32 , Int64 , UInt64 , BigInteger , Enum , TimeSpan	长整型
Single , Double , Decimal	浮点型
Single.NaN , Double.NaN	非数
Single.PositiveInfinity , Double.PositiveInfinity	正无穷大
Single.NegativeInfinity , Double.NegativeInfinity	负无穷大
true	布尔真
false	布尔假
null , DBNull	空
Byte[] , Stream	二进制型 (或空串)
Char , Char[1] , 长度为 1 的 String , 长度为 1 的 StringBuilder	字符型
Char[] , String , StringBuilder	字符串型 (或空串)
DateTime	日期/时间/日期时间型
Guid	Guid 型

反序列化默认类型映射

默认类型是指在对 Hprose 数据反序列化时，在不指定类型信息的情况下得到的反序列化结果类型。

Hprose 类型	.NET 类型
整型	Int32
长整型	BigInteger
浮点型	Double
非数	Double.NaN

Hprose 类型	.NET 类型
正无穷大	Double.PositiveInfinity
负无穷大	Double.NegativeInfinity
布尔真	true
布尔假	false
空	null
空串	""
二进制型	Byte[]
字符型	Char
字符串型	String
日期/时间/日期时间型	DateTime
Guid 型	Guid

反序列化有效类型映射

有效类型是指在对 Hprose 数据反序列化时，可以指定的反序列化结果类型。当指定的类型为安全类型时，反序列化总是可以得到结果。当指定的类型为非安全类型时，只有当数据符合一定条件时，反序列化才能得到结果，不符合条件的情况下，可能会得到丢失精度的结果或者抛出异常。当指定的类型为非有效类型时，反序列化时会抛出异常。

Hprose 类型	.NET 类型 (安全)	.NET 类型 (非安全)
整型	Int32 , Int64 , BigInteger , String , Single , Double , Decimal	Byte , SByte , Int16 , UInt16 , UInt32 , UInt64 , Enum , Char , Boolean , DateTime , TimeSpan
长整型	BigInteger , String	Byte , SByte , Int16 , UInt16 , Int32 , UInt32 , Int64 , UInt64 , Single , Double , Decimal , Enum , Char , Boolean , DateTime , TimeSpan
浮点型	Double , Decimal , String	Byte , SByte , Int16 , UInt16 , Int32 , UInt32 , Int64 , UInt64 , BigInteger , Single , Enum , Char , Boolean , DateTime , TimeSpan
非数	Single.NaN , Double.NaN , String	无
正无穷大	Single.PositiveInfinity , Double.PositiveInfinity , String	无
负无穷大	Single.NegativeInfinity , Double.NegativeInfinity , String	无

Hprose 类型	.NET 类型 (安全)	.NET 类型 (非安全)
布尔真	true ,Byte ,SByte ,Int16 ,UInt16 ,Int32 , UInt32 , Int64 , UInt64 , Char , String , BigInteger , Single , Double , Decimal , Enum	无
布尔假	false , Byte , SByte , Int16 , UInt16 , Int32 , UInt32 , Int64 , UInt64 , Char , String , BigInteger , Single , Double , Decimal , Enum	无
空	null ,Byte ,SByte ,Int16 ,UInt16 ,Int32 , UInt32 , Int64 , UInt64 , Char , Single , Double , Decimal , Enum , Boolean , DBNull	无
空串	Byte[0] ,Char[0] ,"" ,StringBuffer ,Byte , SByte , Int16 , UInt16 , Int32 , UInt32 , Int64 , UInt64 , Char ,String ,BigInteger , Single , Double , Decimal , Enum , Boolean	无
二进制型	Byte[] , Stream , MemoryStream	String , Guid
字符串型	String , StringBuffer , Char[] , Byte[] , Stream , MemoryStream	Byte , SByte , Int16 , UInt16 , Int32 , UInt32 , Int64 , UInt64 , Char , BigInteger , Single , Double , Decimal , Guid
日期/时间/日期时间型	DateTime , TimeSpan , String , Int64	无
Guid 型	Guid , String , StringBuffer , Char[] , Byte[]	无

容器类型

Hprose 中的容器类型包括列表类型和字典类型两种。下面我们来分别介绍它们与.NET 类型的映射关系。

列表类型

序列化类型映射

除 Byte[] , Char[] 以外的所有其它数组类型和所有实现了 ICollection , ICollection<T> , IList , IList<T> 接口的类型均映射为 Hprose 列表类型。

反序列化类型映射

Hprose 列表类型默认映射为 .NET 的 ArrayList 类型。有效类型为：

- 所有数组类型
- 所有实现了 `ICollection<T>` , `IList` , `IList<T>` 接口的可实例化类型

字典类型

序列化类型映射

所有实现了 `IDictionary`、`IDictionary<TKey, TValue>` 接口的类型和匿名类型均映射为 Hprose 字典类型。

反序列化类型映射

Hprose 字典类型默认映射为 .NET 的 `HashMap` 类型。

什么 ? `HashMap` ? 没错 , 是 `HashMap` !

.NET 中的 `Hashtable` 和范型 `Dictionary` 都不允许键值为 `null` , 而大多数其它语言是支持 `null` 作为键值的。为了提供与其它语言的兼容性 , Hprose 为 .NET 增加了 `HashMap` 类型 , `HashMap` 与 `Hashtable` 的唯一区别就是它允许键值为 `null` 。它是作为 `Hashtable` 的子类实现的 , 所以您可以完全不必在您的程序中直接使用 `HashMap` 类型 , 使用 `Hashtable` 类型就可以接收 `HashMap` 的对象实例了。

Hprose 字典类型的有效映射类型为 :

- 所有实现了 `IDictionary`、`IDictionary<TKey, TValue>` 接口的可实例化类型
- 所有拥有与字典中 `Key` 所对应的属性或字段相同的自定义可序列化可实例化类型

对象类型

.NET 中自定义的可序列化对象类型在序列化时被映射为 Hprose 对象类型。

Hprose 对象类型在反序列化时被映射为 :

- .NET 中自定义的可序列化对象类型
- `Hashtable` (当上述类型定义不存在时)

Hprose 1.2 及其更早的版本所支持的 .NET 自定义可序列化对象类型仅是 .NET 可序列化类型的一个子集。另外 , Hprose 1.2 及其更早的版本还对 .NET 对象类型的序列化分为两种 , 一种是按属性序列化 , 一种是按字段序列化。这两种方式下都需要在定义类时标记 `[Serializable]` 属性。按属性序列化时 , 只有 `get` 和 `set` 存取方法都定义的 `public` 属性会被序列化。按字段序列化时 , 没有标记 `[NotSerialized]` 属性的字段都会被序列化。

在 Hprose 1.3 中 , Hprose 的可序列化对象类型有了新的扩展 , 可以不标记 `[Serializable]` 。只要这个类型的所有 `public` 的字段和可读写属性都是可序列化类型 , 那么这个类型就可以序列化 , 并且 , 没有标记 `[Serializable]` 的对象类型 , 在序列化时 , 不区分属性序列化还是字段序列化 , 所有的 `public` 的字段和可读写属性都会被序列化。这个新特性 , 可以让您的程序在编写和移植时都更加方便。

Hprose 1.3 还对标记了 `[DataContract]` 的类型提供序列化支持 , 该类型中所有标记了 `[DataMember]` 的属性或字段都会被序列化。但 Hprose 1.3 对 `[DataContract]` 类型的高级特征还未提供支持 (例如字段别名 , `KnownType` 等) 。

通过 ClassManager 来注册自定义类型

自定义可序列化类型要跟其它语言交互时，默认情况下需要包名和类名都要跟其他语言的定义匹配。有没有办法让已有的类在不需要修改包名或类名的情况下，就能跟其它语言中的类型交互呢？

通过 Hprose.IO.ClassManager 的 Register 方法就可轻松实现这个需求。例如您有一个 My.Package.User 的类，希望传递给 PHP，但是在 PHP 中与之对应的类是 User，那么可以这样做：

```
ClassManager.Register(typeof(My.Package.User), "User");
```

如果其它语言中的类名也是包含名空间的，在注册时，要把名称空间的点分隔符改为下划线分隔符，例如假设要将 my.package.User 注册为 My.User，在调用 register 时要这样写：

```
ClassManager.Register(typeof(My.Package.User), "My_User");
```

这样就可以传给其它语言中定义为 My.User（例如 Java）或者 My_User（例如 PHP）的类了。

第三章 服务器

前面我们在快速入门一章里学习了如何创建一个简单的 Hprose for C#服务器，在本章中您将深入的了解 Hprose for C#服务器的更多细节。

本章提要

- 通过独立服务器方式发布服务
- 通过 aspx 方式发布服务
- 通过自定义 HTTP 处理程序来发布服务

通过独立服务器方式发布服务

因为在快速入门里面我们已经详细通过图解方式介绍了以独立服务器方式发布服务的整个过程，这里就不再通过图解方式介绍了，下面我们更多关注的是代码部分。

Hprose 服务器最核心的功能就是发布服务，而 Hprose 发布的服务在本质上就是方法，所以我们先来看如何发布方法。

发布实例方法

在前面快速入门一章里，您已经看过如何发布一个类中的实例方法了。在哪个例子中我们传输的是简单类型。下面我们主要以传输复杂类型为例，来介绍发布多个类中的实例方法，其中还包括了继承、覆盖、重载方法发布等内容。

先来看第一个类：

```
public class Exam1 {
    protected String id;
    public Exam1() {
        id = "Exam1";
    }
    public String GetID() {
        return id;
    }
    public int Sum(int[] nums) {
        int sum = 0;
        for (int i = 0, n = nums.Length; i < n; i++) {
            sum += nums[i];
        }
        return sum;
    }
    public Hashtable SwapKeyAndValue(Hashtable strmap) {
        DictionaryEntry[] entrys = new DictionaryEntry[strmap.Count];
        strmap.CopyTo(entrys, 0);
        strmap.Clear();
        foreach (DictionaryEntry entry in entrys) {
            strmap[entry.Value] = entry.Key;
        }
        return strmap;
    }
}
```

上面类中，有 3 个可以发布的方法 GetID，Sum 和 SwapKeyAndValue。

其中 GetID 返回的是一个实例字段 id，该字段定义成 protected 是为了后面在子类中可以修改其值，这样当客户端调用子类中发布的方法时，就会返回不同于父类方法的值了。如果看到这里您还不清楚说的是什么，没有关系，等后面看到子类定义和发布时，您就会明白啦，所以，暂时您可以不用理会这个细节。

Sum 方法的参数是一个整型数组，其结果是返回数组中所有整数的和。不过当您在客户端调用它时，并不必非要带入整型数组类型的参数，也可以是整型元素的列表。但仍然推荐使用类型相同的参数进行调用，以保证能够得到最好的效率和安全性。

SwapKeyAndValue 方法的参数是一个 Hashtable，其结果为键值对调后的 Hashtable。不管参数还是结果，我们使用的都是 Hashtable，但是如果在调用时，客户端使用带有 null 值的 Hashtable，在服务器端并不会抛出错误，而是会正确执行。这要归功于 Hprose 提供的允许 null 作为键的 HashMap 类。

下面我们来看第二个类，为了说明继承、覆盖和重载方法的发布，第二个类将作为第一个类的子类。不过为了说明传输自定义可序列化类型，我们这里要先看一下在第二个类中所使用的自定义可序列化类型的定义。

```
public enum Sex {
    Unknown, Male, Female, InterSex
}
```

上面是一个枚举类型，这个枚举类型将在下面的 User 类中使用，在介绍 User 类之前，先来说明一下这个枚举类型。

C#的枚举类型默认是从 0 开始，按照顺序给每个枚举值编号的。因此在 Sex 类型被序列化时，Unknown、Male、Female、InterSex 分别对应的是 0，1，2，3。现在您可能已经明白这个顺序不是乱写的了，从位运算的角度来说，这个顺序是有意义的。例如：

```
Unknown | Male = Male
Unknown | Female = Female
Male | Female = InterSex
```

不要为 InterSex 这个性别感到惊奇，我们这个社会并不是一个非男即女的二元性别社会，我们不能也不应该否认双性人的存在，承认 InterSex 这个性别是对这个群体的尊重。虽然我们这里讨论的问题与此无关。

下面我们继续回到正题上来。

C#中支持对枚举进行位运算，在其它不支持对枚举类型进行位运算的语言（例如 Java）中也可以按照这个顺序定义。经过这样合理的安排枚举类型的顺序后，就可以在这些语言中方便的交互了。毕竟 Hprose 是支持多语言互通的，因此我们在编程时应当考虑到这一点。

下面我们来看 User 类的定义吧。

```
[Serializable]
public class User {
    private string name;
    private Sex sex;
    private DateTime birthday;
    private int age;
    private bool married;
    public User() {
    }
    public User(string name, Sex sex, DateTime birthday, int age, bool married) {
        this.name = name;
        this.sex = sex;
```

```
        this.birthday = birthday;
        this.age = age;
        this.married = married;
    }
    public string Name {
        get {
            return name;
        }
        set {
            name = value;
        }
    }
    public Sex Sex {
        get {
            return sex;
        }
        set {
            sex = value;
        }
    }
    public DateTime Birthday {
        get {
            return birthday;
        }
        set {
            birthday = value;
        }
    }
    public int Age {
        get {
            return age;
        }
        set {
            age = value;
        }
    }
    public bool Married {
        get {
            return married;
        }
        set {
            married = value;
        }
    }
}
```

```
}
```

该类的定义有以下几点需要注意：

1. 必须要标记[Serializable]属性。
2. 必须要有一个无参构造方法。
3. 字段定义为私有，属性定义为共有。
4. 字段名和属性名应一一对应（仅首字母大小写不同）。

按照这种方式定义的类，既可以按照属性序列化传输，也可以按照字段序列化传输。

在 Hprose 1.3 中，则无需加上[Serializable]，这样在传输时也就不再区分字段序列化还是属性序列化了。

下面我们来看第二个要发布的类如何定义：

```
public class Exam2 : Exam1 {
    public Exam2() {
        id = "Exam2";
    }
    public List<User> GetUserList() {
        List<User> userlist = new List<User>();
        userlist.Add(new User("Amy", Sex.Female, new DateTime(1983, 12, 3), 26, true));
        userlist.Add(new User("Bob", Sex.Male, new DateTime(1989, 6, 12), 20, false));
        userlist.Add(new User("Chris", Sex.Unknown, new DateTime(1980, 3, 8), 29, true));
        userlist.Add(new User("Alex", Sex.InterSex, new DateTime(1992, 6, 14), 17, false));
    };
    return userlist;
}
}
```

这个类中，我们会发现在构造方法中，修改了父类中的 id 字段值，这样在该类的对象上调用继承来的 getID 方法时，将会返回"Exam2"这个值。

getUserList 方法很简单，就是创建一个 User 的 List，然后返回。

好了，我现在来看如何发布它们吧。我们先来看最简单的发布一个类 Exam2：

按照快速入门中的经验，发布 Exam2 只需要以下三句就可以了：

```
HproseHttpListenerServer server = new HproseHttpListenerServer("http://localhost:2010/");
server.Methods.AddInstanceMethods(new Exam2());
server.Start();
```

但是运行结果可能会出乎您的预料，当您打开浏览器浏览服务页面时，您会发现只有 GetUserList 这一个方法被发布了，从 Exam1 中继承来的方法都没有被发布，这是正确的吗？

是的，这是正确的，这是因为所有的类都是从祖先类 Object 继承下来的，如果将继承来的 public 方法都发布的话，势必会将从 Object 继承来的方法也一起发布，这肯定不是用户所期望的结果。另一个原因，用户定义的类的层次可能比较深，而其定义的基类上的方法可能并不想被一起发布。因此，默认情况下，只发布直接在该类中声明的 public 方法，而不会发布继承来的方法。

那如果想发布继承来的方法可以吗？当然没问题。您可以这样发布：

```
HproseHttpListenerServer server = new HproseHttpListenerServer("http://localhost:2010/");
Exam2 exam2 = new Exam2();
server.Methods.AddInstanceMethods(exam2, typeof(Exam1));
server.Methods.AddInstanceMethods(exam2, typeof(Exam2));
server.Start();
```

这里，AddInstanceMethods 方法的第一参数为创建发布方法的对象，第二个参数为要发布的类层次的类名。

```
server.Methods.AddInstanceMethods(exam2, typeof(Exam2));
```

与

```
server.Methods.AddInstanceMethods(exam2);
```

作用是等同的，采用上面的写法，只是让程序看上去更美观。

通过上面的写法，您就会发现 Exam2 上的声明的方法和继承自 Exam1 的方法都被发布了。

那如果还要发布一个 Exam1 对象上的方法怎么办呢？您可能会认为只要再创建一个 exam1 对象，使用 AddInstanceMethods 发布它就可以了。例如：

```
HproseHttpListenerServer server = new HproseHttpListenerServer("http://localhost:2010/");
Exam1 exam1 = new Exam1();
Exam2 exam2 = new Exam2();
server.Methods.AddInstanceMethods(exam1);
server.Methods.AddInstanceMethods(exam2, typeof(Exam1));
server.Methods.AddInstanceMethods(exam2, typeof(Exam2));
server.Start();
```

但实际上这样的话，Exam1 对象上的方法将无法被客户端调用，因为它被 Exam2 对象上的从 Exam1 继承来的方法覆盖了。

怎样才可以让他们不会冲突呢？很简单，通过增加名字空间（或者叫别名前缀）的方式就可以避免这种来自不同类的相同名称甚至相同参数的方法的重载问题。

正确的写法如下：

```
HproseHttpListenerServer server = new HproseHttpListenerServer("http://localhost:2010/");
Exam1 exam1 = new Exam1();
Exam2 exam2 = new Exam2();
server.Methods.AddInstanceMethods(exam1, "ex1");
server.Methods.AddInstanceMethods(exam2, typeof(Exam1), "ex2");
server.Methods.AddInstanceMethods(exam2, typeof(Exam2), "ex2");
server.Start();
```

我们发现，这里多了第三个参数，第三个参数就是名字空间，之所以又叫别名前缀是因为它实际上是通过附加在发布的方法前，并用下划线分隔名字空间和方法名来实现的。

从上面的写法中您还可以发现，第二个参数如果是字符串类型，它表示的也是名称空间（别名前缀）。现在运行您的服务器，并打开浏览器浏览发布页，应该可以看到如下的发布列表：

```
Fa7{s9"ex1_GetID"s7"ex1_Sum"s19"ex1_SwapKeyAndValue"s9"ex2_GetID"s7"ex2_Sum"s19"ex2_SwapK  
eyAndValue"s15"ex2_GetUserList"}z
```

这就表示服务发布成功了。

Methods 属性

上面我们在介绍发布服务时，只使用了 Methods 属性的 AddInstanceMethods 方法来发布对象上的实例方法。其实，您还可以通过它的 AddStaticMethods 方法来发布类上的静态方法。通过 AddMethod 和 AddMethods 方法还可以进行更细粒度的方法发布控制，通过 AddMissingMethod 还可以设置默认处理方法来处理客户端调用的而服务器上没有发布的方法。

我们下面就来详细介绍一下 Methods 属性上的这些方法。

AddMethod 方法

AddMethod 用来控制单个方法的发布，它有十一种重载形式。通过它可以发布任意对象上的任意 public 实例方法，或任意类上的任意 public 静态方法，并且可以为每个方法都指定别名。当您发布的办法具有相同个数参数的重载时，使用 AddMethod 是唯一可行的方法。

Hprose 1.3 中，为该方法增加了一个参数 HproseResultMode mode，该参数用来指明方法调用结果的类型。如果返回结果就是普通对象，那么不需要加这个参数，也就是默认值 HproseResultMode.Normal。如果返回结果是 Hprose 序列化之后的数据（byte[] 类型），那么设置该参数为 HproseResultMode.Serialized 可以避免该结果被二次序列化。如果返回结果是一个完整的响应，当这个响应结果不带 Hprose 结束符时，需要将该参数设置为 HproseResultMode.Raw，如果这个响应结果带 Hprose 结束符，则设置这个参数为 HproseResultMode.RawWithEndTag。这个参数主要用于存储转发的 Hprose 代理服务器。通常我们不需要用到这个参数。下面的几个方法也同样增加了这个参数，就不再重复说明了。

AddMethods 方法

AddMethods 用来控制一组方法的发布，它有九种重载形式。如果您所发布的办法来自同一个对象，或是同一个类，使用 AddMethods 方法通常比直接使用 AddMethod 方法更为方便。因为您不但可以为每个方法都指定别名，还可以为这一组方法指定同一个名称空间（别名前缀）。但实际上我们很少情况会直接使用它。因为有更加简单的 AddInstanceMethods 和 AddStaticMethods 方法。

AddInstanceMethods 方法

AddInstanceMethods 用来发布指定对象上的指定类层次上声明的所有 public 实例方法。它有四种重载形式。如果您在使用 AddInstanceMethods 方法时，不指定类层次，则发布这个对象所在类上声明的所有 public 实例方法。这个方法也支持指定名称空间（别名前缀）。

AddStaticMethods 方法

AddStaticMethods 用来发布指定类上声明的所有 public 静态方法。它有两种重载形式。这个方法也支持指定名称空间（别名前缀）。

AddMissingMethod 方法

这是一个很有意思的方法，它用来发布一个特定的方法，当客户端调用的方法在服务器发布的办法中没有查找到时，将调用这个特定的方法。它有两种重载形式。

使用 AddMissingMethod 发布的方法可以是实例方法，也可以是静态方法，但是只能发布一个。如果多次调用 AddMissingMethod 方法，将只有最后一次发布的有效。

用 AddMissingMethod 发布的方法参数应为以下形式：

```
(String name, Object[] args)
```

第一个参数表示客户端调用时指定的方法名，方法名在传入该方法时全部是小写的。

第二个参数表示客户端调用时传入的参数列表。例如客户端如果传入两个参数，则 args 的数组长度为 2，客户端的第一个参数为 args 的第一个元素，第二个参数为 args 的第二个元素。如果客户端调用的方法没有参数，则 args 为长度为 0 的数组。

隐藏发布列表

上面例子中，我们发布服务之后，可以通过浏览器直接查看到所有发布的方法名称列表。这个发布列表的作用相当于 Web Service 的 WSDL，与 WSDL 不同的是，Hprose 的发布列表仅包含方法名，而不包含方法参数列表，返回结果类型，调用接口描述，数据类型描述等信息。这是因为 Hprose 是支持弱类型动态语言调用的，因此参数个数，参数类型，结果类型在发布期是不确定的，在调用期才会确定。所以，Hprose 与 Web Service 相比无论是服务的发布还是客户端的调用上都更加灵活。

当然您可能出于某些原因，并不希望用户直接通过浏览器就可以查看发布列表，那么您可以禁止服务器接收 GET 请求。方法很简单，只需要将服务器的 IsGetEnabled 属性设置为 false 即可：

```
server.IsGetEnabled = false;
server.Start();
```

该设置必须在服务器启动之前设置，之后再打开浏览器您将会看到浏览器中一片空白。这样通过 GET 方式访问就不再显示发布列表啦。但是客户端调用仍然可以正常执行，丝毫不受影响。

不过在调试期间，不建议禁用发布列表，否则将会给您的调试带来很大的麻烦。也许您更希望能够在调试期得到更多的调试信息，那这个可以做到吗？答案是肯定的，您只要打开调试开关就可以了。

调试开关

默认情况下，在调用过程中，服务器端发生错误时，只返回有限的错误信息。当打开调试开关后，服务器会将错误堆栈信息全部发送给客户端，这样，您在客户端就可以看到详细的错误信息啦。

设置方法与隐藏发布列表类似，只需要将服务器的 IsDebugEnabled 属性设置为 true 即可。

```
server.IsDebugEnabled = true;
server.Start();
```

对象序列化模式

前面我们发布的 GetUserList 方法传递的是自定义可序列化对象列表。在介绍自定义可序列化对象时我们也多次提到有两种序列化模式：字段模式和属性模式。

那么字段模式和属性模式究竟哪种更好呢？我们推荐您使用属性模式，因为属性模式不但可以更好的跟其它语言交互，而且效率上也高于字段模式。但默认情况下，Hprose 却是以字段模式进行序列化的，这样做主要是为了与 PHPRPC 中的设置相兼容。

那么如何来设置使用属性模式呢？通过设置服务器的 Mode 属性就可以了。假如要设置成属性模式，只需要加入以下语句即可：

```
server.Mode = Hprose.IO.HproseMode.PropertyMode;
server.Start();
```

设置为属性模式传输时，属性名首字母自动以小写表示，其它部分大小写不变，这样不但可以跟字段序列化模式统一，跟其它语言对象中的字段或属性命名也统一。在 Hprose 1.3 中，对未标记[Serializable]的对象和标记有[DataContract]的对象的序列化，不受该配置影响。

P3P 开关

在 Hprose 的 http 服务中还有一个 P3P 开关，这个开关决定是否发送 P3P 的 http 头，这个头的作用是让 IE 允许跨域接收的 Cookie。当您的服务需要在浏览器中被跨域调用，并且希望传递 Cookie 时（例如通过 Cookie 来传递 Session ID），您可以考虑将这个开关打开。否则，无需开启此开关。此开关默认是关闭状态。开启方法如下：

```
server.IsP3pEnabled = true;
server.Start();
```

跨域开关

Hprose 支持 JavaScript、ActionScript 和 SilverLight 客户端的跨域调用，对于 JavaScript 客户端来说，服务器提供了两种跨域方案，一种是 W3C 标准跨域方案，这个在服务器端只需要设置：

```
server.IsCrossDomainEnabled = true;
server.Start();
```

即可开启，当您在使用 Hprose 专业版提供的服务测试工具 Nepenthes（忘忧草）时，一定要注意必须要打开此开关才能正确进行调试，否则 Nepenthes 将报告错误的服务器。

另一种跨域方案同时适用于以上三种客户端，那就是通过设置跨域策略文件的方式。您可以通过 CrossDomainXmlFile 或者 CrossDomainXmlContent 这两个属性来设置跨域策略文件。CrossDomainXmlFile 的值为跨域策略文件的路径，您可以设置绝对地址，也可以设置成相对于当前可执行文件所在目录的相当地址。CrossDomainXmlContent 的值为跨域策略文件的内容。这两个属性，您不必同时设置，只需要设置其中之一即可。当设置这两个属性的其中一个时，另一个属性值也会有相应的改变，因此如果您同时设置了这两个属性的话，只有最后设置的那个属性生效。

最后，Hprose 还提供了专门适用于 SilverLight 的跨域方案，那就是通过设置客户端访问策略文件的方

式。您可以通过 ClientAccessPolicyXmlFile 或者 ClientAccessPolicyXmlContent 这两个属性来设置客户端访问策略文件。设置方法和注意事项与 CrossDomainXmlFile、CrossDomainXmlContent 相同，这里不再详细说明。

关于 crossdomain.xml 和 clientaccesspolicy.xml 的更多内容请参阅：

- http://www.adobe.com/devnet/articles/crossdomain_policy_file_spec.html
- http://www.adobe.com/devnet/flashplayer/articles/fplayer9_security.html
- <http://msdn.microsoft.com/en-us/library/cc197955%28v=VS.95%29.aspx>
- <http://msdn.microsoft.com/en-us/library/cc838250%28v=VS.95%29.aspx>

服务器事件

也许您可能还希望设置其它的 http 头，或者希望在发生错误时，能够在服务器端进行日志记录。甚至希望在调用发生的前后可以做一些权限检查或日志记录等。在 Hprose 中，这些都可以轻松做到。Hprose 提供了这样的事件机制。

Hprose 服务器提供了 4 个事件，它们在 HproseServiceEvent.cs 中被定义为四个委托：

```
namespace Hprose.Server {
    public delegate void BeforeInvokeEvent(string name, object[] args, bool byRef);
    public delegate void AfterInvokeEvent(string name, object[] args, bool byRef, object result);
    public delegate void SendHeaderEvent();
    public delegate void SendErrorEvent(string error);
}
```

它们分别对应 Hprose 服务的：

- OnBeforeInvoke
- OnAfterInvoke
- OnSendHeader
- OnSendError

这四个事件。

OnBeforeInvoke 事件

当服务器端发布的方法被调用前，OnBeforeInvoke 事件被触发，其中 name 为客户端所调用的方法名，args 为方法的参数，byRef 表示是否是引用参数传递的调用。

您可以在该事件中做用户身份验证，例如 IP 验证。也可以作日志记录。如果在该事件中想终止调用，抛出异常即可。

OnAfterInvoke 事件

当服务器端发布的方法被成功调用后，OnAfterInvoke 事件被触发，其中前三个参数与 OnBeforeInvoke 事件一致，最后一个参数 result 表示调用结果。

当调用发生错误时，OnAfterInvoke 事件将不会被触发。如果在该事件中抛出异常，则调用结果不会被返回，客户端将收到此事件抛出的异常。

OnSendHeader 事件

当服务器返回响应头部时，OnSendHeader 事件会被触发。

在该事件中，您可以发送您自己的头信息，例如设置 Cookie。该事件中不应抛出任何异常。

OnSendError 事件

当服务器端调用发生错误，或者在 OnBeforeInvoke、OnAfterInvoke 事件中抛出异常时，该事件被触发。

您可以在该事件中作日志记录，但该事件中不应再抛出任何异常。

存取环境上下文

在上面介绍的服务器事件中，您可能会需要存取环境上下文，例如 Request，Response。那么有什么方便的方法来获取它们吗？

Hprose 为独立服务器和基于 IIS 的服务器提供了不同的解决方案。对于独立服务器来说，您可以通过 HproseHttpListenerService 类的 CurrentContext 静态属性就可以存取独立服务器的环境上下文了。

独立服务器的 CurrentContext 静态属性是 HttpListenerContext 类型的，您可以通过它的 Request、Response、User 属性来存取这些上下文。

HproseHttpListenerService 类的 CurrentContext 静态属性可以在服务器事件中使用，也可以在服务器发布的方法中使用。另外，在服务器发布的方法中还有另外一种方式来获取环境上下文，那就是将方法的最后一个参数定义为相应的环境上下文类型。

例如要获取 Request 对象，只需要将最后一个参数定义为 HttpListenerRequest 类型即可。客户端在调用该方法时，不需要代入这个参数，服务器会自动传递 Request 参数给该方法。

通过参数方式可以传递 HttpListenerContext 本身和它所包含的所有的环境上下文类型，但是一次只能传递一个环境上下文参数。

尽管直接通过参数传递的方式在写法更简洁一些，但我们并不鼓励用户在业务函数中包含对环境上下文的存取，这样会破坏系统的松散耦合性。

好了，到这里独立服务器基本上就介绍完了，除了上面这些内容外，您还可以使用 IsStarted 属性来判断服务是否启动，通过 Stop 方法来停止服务。有了这些，您完全可以做一个 Winform 控制的服务器了。

通过 aspx 方式发布服务

以独立服务器方式发布服务时，发布的所有方法都是全局的，也就是说服务对象从服务器启动时被创建，直到服务停止时才会被销毁。另外独立服务器方式对环境上下文的存取也是有限的，它不能对 Session、Application 这些环境上下文进行操作。

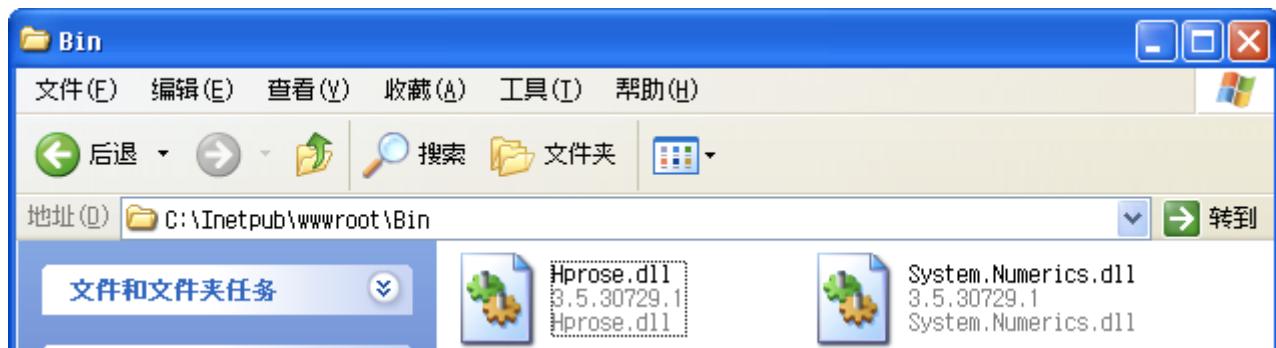
如果您希望对方法的发布进行更进一步的控制，或者希望能够对 Session、Application 这些环境上下文进行操作。那么您可以选择以 aspx 方式发布服务。

按全局发布方法

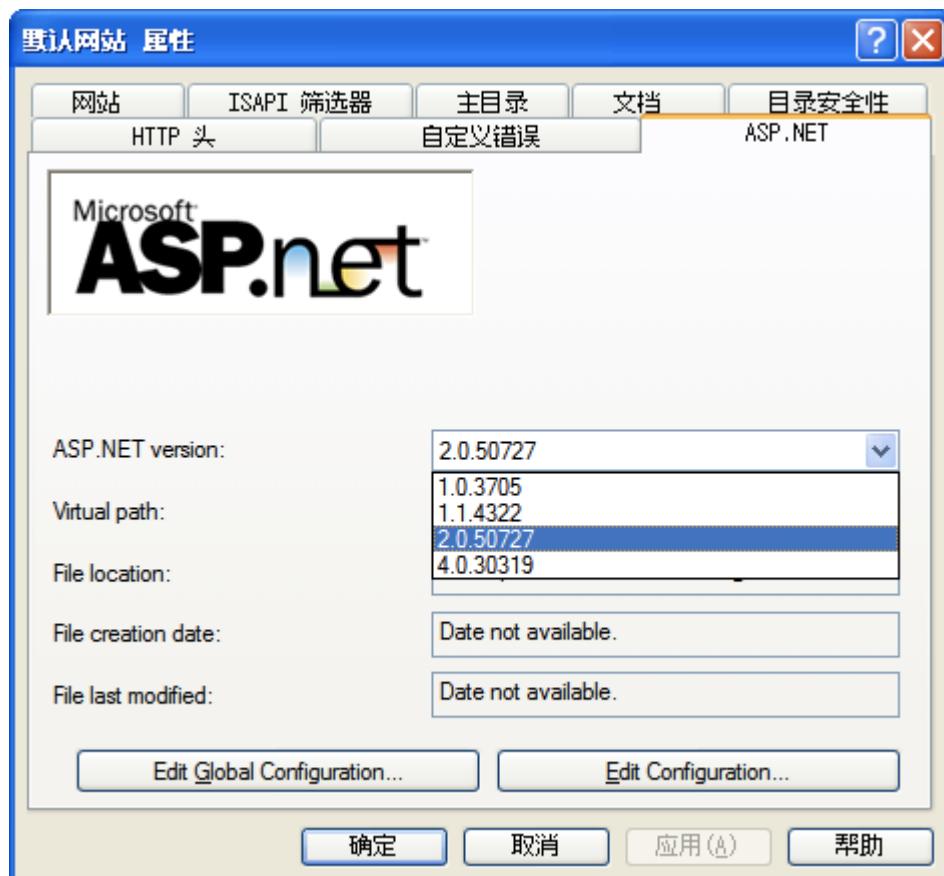
我们先来看看要实现一个跟独立服务器部分最后一个发布例子相同功能的 aspx 版本需要怎样做。

首先把正确版本的 Hprose.dll 和 System.Numerics.dll 复制到 IIS 发布目录的 ASP.NET 应用程序目录

下的 Bin 目录下：



然后将您的 IIS 服务器的 ASP.NET 配置成正确的版本：



注意，ASP.NET 2.0 是可以运行 3.5 版本的 Hprose 的。

然后在发布目录下创建一个 App_Code 目录，然后将定义 Exam1 和 Exam2 这两个类的 cs 文件，放入其中。因为前面在独立服务器部分已经将这两个类定义的代码以分块列出，这里就不再重复。

接下来我们在发布目录下创建 Global.asax 文件：

```
<%@ Application Language="C#" %>
<object id="ExamService" class="Hprose.Server.HproseHttpService" runat="server" scope="application" />
<script runat="server">
    void Application_Start(object sender, EventArgs e) {
        Exam1 exam1 = new Exam1();
        Exam2 exam2 = new Exam2();
```

```

ExamService.GlobalMethods.AddInstanceMethods(exam1, "ex1");
ExamService.GlobalMethods.AddInstanceMethods(exam2, typeof(Exam1), "ex2");
ExamService.GlobalMethods.AddInstanceMethods(exam2, typeof(Exam2), "ex2");
ExamService.Mode = Hprose.IO.HproseMode.PropertyMode;
ExamService.IsDebugEnabled = true;
ExamService.IsP3pEnabled = true;
ExamService.IsCrossDomainEnabled = true;
}
</script>

```

上面的代码我们创建了一个全局的 ExamService 对象，并在 Application_Start 事件中对其进行了初始化。ExamService 是 HproseHttpService 类型的，它是一个线程安全的类，因此它可以被创建为全局对象。

这里 ExamService 的 GlobalMethods 属性是 HproseHttpMethods 类型的，跟独立服务器的 Methods 属性功能是一样的，虽然它与独立服务器的 Methods 属性并不是相同的类，但它们的父类是相同的，都是 HproseMethods 类型。

ExamService 同样拥有 Mode、IsDebugEnabled、IsP3pEnabled、IsGetEnabled、IsCrossDomainEnabled 这些属性，并且可以在全局对它们进行设置。但是它并不包含 CrossDomainXmlFile、CrossDomainXmlContent、ClientAccessPolicyXmlFile 和 ClientAccessPolicyXmlContent 这四个属性，因为这两个文件可以直接放在 IIS 的发布目录下，而无需通过 Hprose 来配置。

ExamService 同样拥有跟独立服务器相同的那四个事件，因为用法很简单，不过这里不进行举例说明了。

在全局初始化 ExamService 后，我们就可以在 aspx 中发布它了，我们接下来在发布目录下，建立一个 ExamService.aspx 文件：

```

<%@ Page Language="C#" %>
<script runat="server">
    void Page_Load(object sender, EventArgs e) {
        ExamService.Handle();
    }
</script>

```

这里面只有一句，就是调用 ExamService 上的 Handle 方法来处理每一个客户端的调用。

好了，现在在浏览器中键入该 aspx 的地址后，如果看到：

```
Fa7{s9"ex1_GetID"s7"ex1_Sum"s19"ex1_SwapKeyAndValue"s9"ex2_GetID"s7"ex2_Sum"s19"ex2_SwapKeyAndValue"s15"ex2_GetUserList"}z
```

就说明服务发布成功了。

注意，如果采用 CodeBehind 的方式，您不可以通过：

```
<object id="ExamService" class="Hprose.Server.HproseHttpService" runat="server" scope="application" />
```

这种方式来创建 HproseHttpService 对象，您只能采用代码方式创建，并放入 Application 中进行存取。这是因为 CodeBehind 代码的编译早于这段代码的执行，所以在编译时根本找不到 ExamService 这个对象。

另外要注意保存 asax 和 aspx 文件时请使用 UTF-8 编码，并去掉 BOM。aspx 文件中不要包含多余的 html 代码和多余空白。

按会话发布方法

按 Session 发布也很简单，这时 Global.asax 这样写：

```
<%@ Application Language="C#" %>
<object id="ExamService" class="Hprose.Server.HproseHttpService" runat="server" scope="application" />
<object id="exam1" class="Exam1" runat="server" scope="session" />
<object id="exam2" class="Exam2" runat="server" scope="session" />
<object id="methods" class="Hprose.Server.HproseHttpMethods" runat="server" scope="session" />
<script runat="server">
    void Application_Start(object sender, EventArgs e) {
        ExamService.Mode = Hprose.I0.HproseMode.PropertyMode;
        ExamService.IsDebugEnabled = true;
        ExamService.IsP3pEnabled = true;
        ExamService.IsCrossDomainEnabled = true;
    }
    void Session_Start(object sender, EventArgs e) {
        methods.AddInstanceMethods(exam1, "ex1");
        methods.AddInstanceMethods(exam2, typeof(Exam1), "ex2");
        methods.AddInstanceMethods(exam2, typeof(Exam2), "ex2");
    }
</script>
```

而 ExamService.aspx 这样写：

```
<%@ Page Language="C#" %>
<script runat="server">
    void Page_Load(object sender, EventArgs e) {
        ExamService.Handle(methods);
    }
</script>
```

这里我们在创建了 ExamService 之后并没有为它的 GlobalMethods 属性添加任何方法，而是在 Session 开始时，创建并初始化了一个 methods 对象，这个对象跟 GlobalMethods 是属于同一个类型的，之后，我们在每个页面被载入时，将 methods 对象作为参数传入了 ExamService 的 Handle 方法。

这样发布的方法的生命周期就跟会话的生命周期相同了。

按请求发布方法

最后我们来看一下按请求如何发布方法，我想这个不用说大家也已经想到了，这个 Global.asax 应该这

样写：

```
<%@ Application Language="C#" %>
<object id="ExamService" class="Hprose.Server.HproseHttpService" runat="server" scope="application" />
<script runat="server">
    void Application_Start(object sender, EventArgs e) {
        ExamService.Mode = Hprose.IO.HproseMode.PropertyMode;
        ExamService.IsDebugEnabled = true;
        ExamService.IsP3pEnabled = true;
        ExamService.IsCrossDomainEnabled = true;
    }
</script>
```

而 ExamServer.aspx 这样写：

```
<%@ Page Language="C#" %>
<%@ Import NameSpace="Hprose.Server" %>
<script runat="server">
    void Page_Load(object sender, EventArgs e) {
        Exam1 exam1 = new Exam1();
        Exam2 exam2 = new Exam2();
        HproseHttpMethods methods = new HproseHttpMethods();
        methods.AddInstanceMethods(exam1, "ex1");
        methods.AddInstanceMethods(exam2, typeof(Exam1), "ex2");
        methods.AddInstanceMethods(exam2, typeof(Exam2), "ex2");
        ExamService.Handle(methods);
    }
</script>
```

相信大家有了前面的基础，这个不做解释，也应该可以看明白了。

存取环境上下文

在 aspx 中存取环境上下文要比独立服务器方式灵活的多，可以存取的环境上下文变量也多。

您可以直接通过 `HttpContext.Current` 来获取当前的环境上下文，也可以通过 `HproseHttpService` 的静态属性 `CurrentContext` 获取。它们通常情况下是一致的，除非当您在调用 `ExamService.Handle` 时传入的是其他的环境上下文。

在获取到 `CurrentContext` 之后，您便可以存取 `Request`、`Response`、`Server`、`Session`、`Application` 这些环境上下文变量啦。

另一种方法与独立服务器方式类似，就是通过发布方法的最后一个参数传入，这种方法我们不推荐用户使用，仅为与 PHRPC 兼容而准备。所以，这里我们不对该方式进行详细介绍。

通过自定义 HTTP 处理程序来发布服务

除了直接以 aspx 发布之外，您也可以通过自定义 HTTP 处理程序来发布服务，例如：

```
using System.Web;
using Hprose.Server;

public class HproseHttpHandler : IHttpHandler {
    private static HproseHttpService service = new HproseHttpService();
    static HproseHttpHandler() {
        Exam1 exam1 = new Exam1();
        Exam2 exam2 = new Exam2();
        service.GlobalMethods.AddInstanceMethods(exam1, "ex1");
        service.GlobalMethods.AddInstanceMethods(exam2, typeof(Exam1), "ex2");
        service.GlobalMethods.AddInstanceMethods(exam2, typeof(Exam2), "ex2");
        service.Mode = Hprose.IO.HproseMode.PropertyMode;
        service.IsDebugEnabled = true;
        service.IsP3pEnabled = true;
        service.IsCrossDomainEnabled = true;
    }
    public void ProcessRequest(HttpContext context) {
        service.Handle(context);
    }
    public bool IsReusable {
        get { return true; }
    }
}
```

这个效果是跟使用 aspx 按全局发布方法是相同的，不过效率上来说，aspx 还要高一点，因此这种方式我们通常用不到。这里就不再做更多介绍了。

第四章 客户端

前面我们在快速入门一章里学习了如何创建一个简单的 Hprose for C#客户端，在本章中您将深入的了解 Hprose for C#客户端的更多细节。

本章提要

- 同步调用
- 异步调用
- 异常处理
- 超时设置
- HTTP 参数设置

同步调用

Hprose 客户端在与服务器通讯时，分同步调用和异步调用两种方式。同步调用的概念和用法相对简单一些，所有我们先来介绍同步调用方式。

在同步调用方式下，如果服务器执行出错，或者通讯过程中出现问题（例如连接中断，或者调用的服务器不存在等），则客户端会抛出异常。

SilverLight 客户端不支持同步调用方式，仅支持异步调用方式。如果您使用 SilverLight 客户端，请跳到下一节。

直接使用 HproseHttpClient 上的 `Invoke` 方法或者采用代理接口方式都可以进行同步调用，但是在.NET Compact Framework 中，只能使用 `Invoke` 方法。

Hprose 1.3 还提供了对 Windows Phone 7 的支持，并且对每个 Windows Phone 7.x 的版本都提供了单独的优化版本。但是请注意，Windows Phone 7 同 SilverLight 一样，不支持同步调用。

在下面的例子中，我们以调用前一章中第一节第一小节最后发布的服务为例来进行说明讲解。

通过 `Invoke` 方法进行同步调用

通过 `Invoke` 方法调用是最直接、最基本的方式，所以我们先来介绍它。因为在前面快速入门一节中，我们已经举过完整实例了，下面的代码我们仍然以控制台应用程序为例，但是为了节省篇幅，突出重点，这里仅列出关键代码。

带名称空间（别名前缀）方法

先来看调用最简单的例子：

```
HproseHttpClient client = new HproseHttpClient("http://localhost:2010/");
Console.WriteLine(client.Invoke("ex1_getId"));
Console.WriteLine(client.Invoke<string>("ex2_getId"));
```

这个例子调用的是服务器端发布的 `GetId` 方法，因为我们在发布时发布了 `Exam1` 和 `Exam2` 两个类的对象上的这个方法，并且分别指定了名称空间（别名前缀），所以这里调用时，我们分别都加了 `ex1` 和 `ex2` 这两个前缀，并且用下划线“`_`”分隔别名前缀和方法名。

调用 `ex1_getId` 时，我们采用的是非泛型方法，调用 `ex2_getId` 时，我们采用的是范型方法。非泛型方法是为了兼容.NET Framework 1.0、1.1 和.NET Compact Framework 1.0 的，通常我们现在使用的是.NET Framework 都是 2.0 以上的版本，因此推荐使用范型版本。

这个例子的运行结果是：

```
Exam1
Exam2
```

从结果中我们可以很清楚的分辨出它们确实是调用了两个不同类的对象的方法。

可变的参数和结果类型

下面我们再来看对 `Sum` 方法的调用：

```

HproseHttpClient client = new HproseHttpClient("http://localhost:2010/");
Console.WriteLine(client.Invoke("ex1_sum", new Object[] { new int[] {1,2,3,4,5} }));
Console.WriteLine(client.Invoke("ex1_sum", new Object[] { new short[] {6,7,8,9,10} }));
Console.WriteLine(client.Invoke("ex1_sum", new Object[] { new long[] {11,12,13,14,15} }));
Console.WriteLine(client.Invoke("ex1_sum", new Object[] { new double[] {16,17,18,19,20} }));
;
Console.WriteLine(client.Invoke("ex1_sum", new Object[] { new string[] {"21","22","23","24",
", "25"} }));
ArrayList intList = new ArrayList();
intList.Add(26);
intList.Add(27);
intList.Add(28);
intList.Add(29);
intList.Add(30);
Console.WriteLine(client.Invoke("ex2_sum", new Object[] { intList }));

```

因为，Sum 也是有别名的，所以调用时要加 ex1 或 ex2 的前缀，虽然这两个方法对应服务器上不同对象上的两个方法，但是这两个方法作用是一致的，所以不管调用哪个，结果都是一致的。

下面看运行结果：

```

15
40
65
90
115
140

```

大家还会发现，Sum 在服务器端声明时，参数为 int[] 类型，结果为 int 类型，但是在调用时，我们除了可以带入 int[] 类型的参数外，还可以带入 short[]、long[]，甚至是 double[]、String[] 等类型的数组和 ArrayList，只要它们元素的值能够转换为 int 就可以。

引用参数传递

下面我们来继续看对 SwapKeyAndValue 方法的调用：

```

HproseHttpClient client = new HproseHttpClient("http://localhost:2010/");
Hashtable map = new Hashtable();
map["January"] = "Jan";
Object[] arguments = new Object[] { map };
client.Invoke("ex1_swapKeyAndValue", arguments);
foreach (DictionaryEntry e in map) {
    Console.WriteLine("Key={0}, Value={1}", e.Key, e.Value);
}
foreach (DictionaryEntry e in (Hashtable)arguments[0]) {
    Console.WriteLine("Key={0}, Value={1}", e.Key, e.Value);
}

```

```

client.Invoke("ex2_swapKeyAndValue", arguments, true);
foreach (DictionaryEntry e in map) {
    Console.WriteLine("Key={0}, Value={1}", e.Key, e.Value);
}
foreach (DictionaryEntry e in (Hashtable)arguments[0]) {
    Console.WriteLine("Key={0}, Value={1}", e.Key, e.Value);
}

```

运行结果：

```

Key=January, Value=Jan
Key=January, Value=Jan
Key=January, Value=Jan
Key=Jan, Value=January

```

从上面的调用演示了引用参数传递。在对 ex1_swapKeyAndValue 进行调用时，我们没有设置引用参数传递，所以运行后，参数值 map 和 arguments[0]并没有对调。但是在对 ex2_swapKeyAndValue 进行调用时，我们设置了引用参数传递，所以调用后，参数 arguments[0]的值也发生了变化，但是需要注意的是，原始的 map 并没有改变，改变的是参数数组 arguments 当中的值。

自定义类型的传输

最后我们来看一下对 getUserList 方法的调用：

```

HproseHttpClient client = new HproseHttpClient("http://localhost:2010/");
List<User> userList = client.Invoke<List<User>>("ex2_getUserList");
foreach (User user in userList) {
    Console.Write("name: {0}, ", user.Name);
    Console.Write("age: {0}, ", user.Age);
    Console.Write("sex: {0}, ", user.Sex);
    Console.Write("birthday: {0}, ", user.Birthday);
    Console.Write("married: {0}.", user.Married);
    Console.WriteLine();
}
Console.WriteLine();
User[] users = client.Invoke<User[]>("ex2_getUserList");
foreach (User user in users) {
    Console.Write("name: {0}, ", user.Name);
    Console.Write("age: {0}, ", user.Age);
    Console.Write("sex: {0}, ", user.Sex);
    Console.Write("birthday: {0}, ", user.Birthday);
    Console.Write("married: {0}.", user.Married);
    Console.WriteLine();
}

```

运行结果：

```

name: Amy, age: 26, sex: Female, birthday: 1983-12-3 0:00:00, married: True.
name: Bob, age: 20, sex: Male, birthday: 1989-6-12 0:00:00, married: False.
name: Chris, age: 29, sex: Unknown, birthday: 1980-3-8 0:00:00, married: True.
name: Alex, age: 17, sex: InterSex, birthday: 1992-6-14 0:00:00, married: False.

name: Amy, age: 26, sex: Female, birthday: 1983-12-3 0:00:00, married: True.
name: Bob, age: 20, sex: Male, birthday: 1989-6-12 0:00:00, married: False.
name: Chris, age: 29, sex: Unknown, birthday: 1980-3-8 0:00:00, married: True.
name: Alex, age: 17, sex: InterSex, birthday: 1992-6-14 0:00:00, married: False.

```

上例中，我们发现，不管是采用范型 `List<User>` 作为返回值类型，还是采用 `User` 数组作为返回值类型。结果都是正确的。这是因为 Hprose 在接收数据时自动将数据反序列化成了我们指定的类型。

通常，如果在不需要对返回的结果作增删的时候，指定数组类型作为返回结果要比范型 `List` 更高效和方便一些。

另外，还要注意客户端和服务器端在传输自定义类型时，一定要确保客户端和服务器端的定义要完全一致（名称空间、类名、甚至包括大小写都要一致），否则请使用 `Hprose.IO.ClassManager.Register` 方法为自定义类注册统一的别名。

通过代理接口进行同步调用

看完通过 `Invoke` 进行同步调用的方式后，再来看一下通过接口进行同步调用的方式。

接口定义

为了调用上面的方法，我们需要先定义接口，下面是接口的定义：

```

public interface IExam1 {
    string GetId();
    int Sum(int[] nums);
    int Sum(short[] nums);
    int Sum(long[] nums);
    int Sum(double[] nums);
    int Sum(string[] nums);
    double Sum(ArrayList nums);
    Dictionary<String, String> SwapKeyAndValue(Dictionary<String, String> strmap);
    Dictionary<String, String> SwapKeyAndValue(ref Dictionary<String, String> strmap);
}

```

这个是与 `Exam1` 对应的接口，其中除了跟 `Exam1` 声明相同的方法以外，还有另外一些重载的方法，以及参数和结果类型不同但是兼容类型的方法。它们都能在调用过程中自动转换为正确的类型进行调用。

```

public interface IExam2 {
    User[] GetUserList();
}

```

这个接口与 Exam2 对应，返回类型我们改成了 User[] 类型。

从上面的接口我们可以看出，在 Hprose 中，客户端和服务器端的接口不必完全一致，这就大大增加了灵活性，也更加方便了跟弱类型语言进行交互。

带名称空间（别名前缀）方法

在使用 Invoke 对带有名称空间（别名前缀）的方法进行调用时，需要将方法名写为带有前缀形式的，但是使用接口调用时，在生成代理对象时，这个工作可以自动帮您完成，看下面的例子：

```
HproseHttpClient client = new HproseHttpClient("http://localhost:2010/");
IExam1 exam1 = (IExam1)client.UseService(typeof(IExam1), "ex1");
IExam1 exam2 = client.UseService<IExam1>("ex2");
Console.WriteLine(exam1.GetId());
Console.WriteLine(exam2.GetId());
```

在这个例子中，exam1 和 exam2 都是 IExam1 的接口对象，exam1 是通过非泛型 UseService 方法得到的，而 exam2 是通过范型 UseService 方法得到的，这里仅仅是语法的不同，这并不是它们的本质区别。它们两个的本质区别是在调用 UseService 方法时被指定了不同的名称空间（别名前缀），因此在后面对它们的 GetId 方法调用时，调用的就是两个不同的方法了。

运行结果如下：

```
Exam1
Exam2
```

可变的参数和结果类型

下面我们再来看对 Sum 方法的调用：

```
HproseHttpClient client = new HproseHttpClient("http://localhost:2010/");
IExam1 exam = client.UseService<IExam1>("ex1");
Console.WriteLine(exam.Sum(new int[] {1,2,3,4,5}));
Console.WriteLine(exam.Sum(new short[] {6,7,8,9,10}));
Console.WriteLine(exam.Sum(new long[] {11,12,13,14,15}));
Console.WriteLine(exam.Sum(new double[] {16,17,18,19,20}));
Console.WriteLine(exam.Sum(new string[] {"21","22","23","24","25"}));
ArrayList intList = new ArrayList();
intList.Add(26);
intList.Add(27);
intList.Add(28);
intList.Add(29);
intList.Add(30);
Console.WriteLine(exam.Sum(intList));
```

这个程序运行结果跟前面用 Invoke 实现的 ClientExam2 是相同的，这个程序看上去更简洁一些，不过需要事先将接口定义好才可以。用接口方式就不需要在调用时指定返回值类型了，Hprose 自动从接口声明中就可以获取返回值类型。

泛型参数和引用参数传递

下面继续来看对 SwapKeyValue 方法的调用：

```
HproseHttpClient client = new HproseHttpClient("http://localhost:2010/");
IExam1 exam = client.UseService<IExam1>("ex1");
Dictionary<String, String> map = new Dictionary<String, String>();
map["January"] = "Jan";
foreach (KeyValuePair<String, String> e in map) {
    Console.WriteLine("Key={0}, Value={1}", e.Key, e.Value);
}
exam.SwapKeyValue(map);
foreach (KeyValuePair<String, String> e in map) {
    Console.WriteLine("Key={0}, Value={1}", e.Key, e.Value);
}
exam.SwapKeyValue(ref map);
foreach (KeyValuePair<String, String> e in map) {
    Console.WriteLine("Key={0}, Value={1}", e.Key, e.Value);
}
```

运行结果如下：

```
Key=January, Value=Jan
Key=January, Value=Jan
Key=Jan, Value=January
```

当参数在接口声明中被标记为 ref 或 out 时，该参数将被以引用方式传递。

自定义类型

下面代码中您会发现接口中的方法签名虽然跟服务器的方法签名不同，但是仍然可以正常调用：

```
HproseHttpClient client = new HproseHttpClient("http://localhost:2010/");
IExam2 exam = client.UseService<IExam2>("ex2");
User[] users = exam.GetUserList();
foreach (User user in users) {
    Console.Write("name: {0}, ", user.Name);
    Console.Write("age: {0}, ", user.Age);
    Console.Write("sex: {0}, ", user.Sex);
    Console.Write("birthday: {0}, ", user.Birthday);
    Console.Write("married: {0}.", user.Married);
    Console.WriteLine();
}
```

运行结果如下：

```
name: Amy, age: 26, sex: Female, birthday: 1983-12-03, married: true.
name: Bob, age: 20, sex: Male, birthday: 1989-06-12, married: false.
name: Chris, age: 29, sex: Unknown, birthday: 1980-03-08, married: true.
name: Alex, age: 17, sex: InterSex, birthday: 1992-06-14, married: false.
```

同样，这个例子已经很好的说明了 Hprose 使用的易用性和灵活性，不用多做解释相信您也已经看懂了。

异步调用

下面我们来开始另一个重要的话题，那就是异步调用。

异步调用相对于同步调用来说确实要难以掌握一些，但是在很多情况下我们却很需要它。那究竟什么时候我们需要使用异步调用呢？

很多时候我们并不确定在进行远程调用时是否能够立即得到返回结果，因为可能由于带宽问题或者服务器本身需要对此调用进行长时间计算而不能马上返回结果给客户端。这种情况下，如果使用同步远程调用，客户端执行该调用的线程将被阻塞，并且在主线程中执行同步远程调用会造成用户界面冻结，这是用户无法忍受的。这时，我们就需要使用异步调用。

因此我们通常在 WinForm、WPF 程序中我们应该使用异步调用而不应该使用同步调用，SilverLight 和 Windows Phone 7 更是强制性的必须使用异步调用，因为在 SilverLight 和 Windows Phone 7 上是没有同步调用支持的。

虽然您也可以使用多线程加同步调用来完成异步调用，但您完全不必这样做。强烈推荐您直接使用 Hprose 提供的异步调用方式，这将更加简单，更加高效。

在.NET Compact Framework 之上，同样支持异步调用，但是只能使用 Invoke 方式。

Windows Phone 7 目前的.NET 环境其实是.NET Compact Framework 3.7，它所提供的.NET 运行库也不是完整的，因此它目前也仅支持 Invoke 方式调用，而不支持代理接口调用方式。

通过 Invoke 方法进行异步调用

通过 Invoke 方式进行异步调用跟同步调用差不多，唯一的区别就是异步调用多了一个回调方法参数。

C#不能向 C/C++ 那样传递函数或方法指针，但是 C# 支持传递方法委托，回调方法类型被定义为以下四个委托类型：

```
public delegate void HproseCallback<T>(T result, object[] args);
public delegate void HproseCallback1<T>(T result);
public delegate void HproseCallback(object result, object[] args);
public delegate void HproseCallback1(object result);
```

其中，前两个是范型委托，后两个是非泛型委托。通常在.NET 2.0 及其更高版本上面使用范型委托，而非泛型委托是为.NET Framework 1.0、1.1 和.NET Compact Framework 1.0 准备的。

委托中第一个参数表示返回结果，第二个参数表示调用参数。如果您不是在进行引用参数传递的调用，那么第二个参数的参数值，跟您调用时的参数值是一致的。在非引用参数传递调用中，如果您在处理结果的回调方法中并不关心调用时的参数，可以使用只有结果参数的回调方法。

当您通过 Invoke 所调用的方法执行完毕时，您所指定的回调方法将会被调用，其中的参数将会自动被传入。

关于通过 Invoke 进行异步调用，我想不用举太多例子，下面这个简单的例子就可以很好的说明如何来使用了：

```
HproseHttpClient client = new HproseHttpClient("http://localhost:2010/");
client.Invoke("ex1_getId", new HproseCallback1(delegate(object result) { Console.WriteLine(result); }));
client.Invoke("ex1_getId", delegate(string result) { Console.WriteLine(result); });
client.Invoke("ex2_getId", (result) => { Console.WriteLine(result); });
client.Invoke<string>("ex2_getId", (result) => { Console.WriteLine(result); });
Console.ReadKey();
```

这个例子展示了同时进行 ex1_getId 和 ex2_getId 这两个方法的异步调用，并对每个方法都做了两次不同形式的调用。

第一次调用 ex1_getId 是通过非泛型方法调用的，这里的 new HproseCallback1 是必须要写的，即使是在.NET 2.0 及其更高版本上，因为有范型方法存在的原因，在不写 new HproseCallback1 的情况下，.NET 无法推断出要调用的是范型版本还是非范型版本的方法。而如果使用的是.NET 1.0、1.1，虽然不会有范型方法和非泛型方法的推断冲突问题，但是.NET 1.0、1.1 本身就不支持委托推断，因此 new HproseCallback1 还是要必须写，而且还不能够使用匿名方法，因为.NET 1.0、1.1 不支持。

而第二次调用 ex1_getId，虽然在方式上看不出调用的是 Invoke 方法的范型版本，但实际上它调用的正是范型版本，这是通过匿名方法的参数类型推断出来的。

而下面第一次调用 ex2_getId 时，回调方法采用了更为简单的 lambda 表达式方式，因为这里 Invoke 的形式是非泛型方法的形式，所以这里 result 的类型通过 Invoke 来推断最后确定为 object 类型。

第二次调用 ex2_getId 时，采用了同样的 lambda 表达式，但因为 Invoke 采用的是范型形式，所以这里的 result 的类型为 string 类型。

您可能还注意到，我们在程序的最后加上了 Console.ReadKey()，因为这里是异步调用，如果不加这一句，调用还没有执行完，程序就已经退出了，这样您将看不到任何执行结果。不过一般情况下，都是在图形用户界面的程序中才会使用异步调用，所以，在那种情况下，您可能需要用其它方法来保证异步调用被完整执行。

这个程序的执行结果为：

```
Exam1
Exam1
Exam2
Exam2
```

但也可能为：

```
Exam1
Exam2
Exam1
Exam2
```

或其它顺序，因为调用是异步的，服务器端不一定首先返回哪个结果，所以结果打印的顺序也是不定的。

通过代理接口进行异步调用

除了可以通过 Invoke 方式外，您也可以通过接口方式来进行异步调用，这里我们来举一个稍微复杂点的例子，来说明引用参数传递，容器类型和自定义类型传输，以及如何在调用中指定要返回的结果类型。

先看接口定义：

```
public interface IAsyncExam {
    void SwapKeyAndValue(Hashtable strmap, HproseCallback<Hashtable> callback);
    void SwapKeyAndValue(ref Hashtable strmap, HproseCallback<Hashtable> callback);
    void GetUserList(HproseCallback1<User[]> callback);
}
```

这个接口中，第一个 SwapKeyAndValue 是引用参数传递，第二个是非引用参数传递。

下面看主程序部分。

```
HproseHttpClient client = new HproseHttpClient("http://localhost:2010/");
IAsyncExam exam = client.UseService<IAsyncExam>("ex2");
Hashtable map = new Hashtable();
map["January"] = "Jan";
exam.SwapKeyAndValue(map, (result, arguments) => {
    foreach (DictionaryEntry e in map) {
        Console.WriteLine("Key={0}, Value={1}", e.Key, e.Value);
    }
    foreach (DictionaryEntry e in result) {
        Console.WriteLine("Key={0}, Value={1}", e.Key, e.Value);
    }
    foreach (DictionaryEntry e in (Hashtable)arguments[0]) {
        Console.WriteLine("Key={0}, Value={1}", e.Key, e.Value);
    }
});
exam.SwapKeyAndValue(ref map, (result, arguments) => {
    foreach (DictionaryEntry e in map) {
        Console.WriteLine("Key={0}, Value={1}", e.Key, e.Value);
    }
    foreach (DictionaryEntry e in result) {
        Console.WriteLine("Key={0}, Value={1}", e.Key, e.Value);
    }
    foreach (DictionaryEntry e in (Hashtable)arguments[0]) {
        Console.WriteLine("Key={0}, Value={1}", e.Key, e.Value);
    }
});
exam.GetUserList((result) => {
    foreach (User user in result) {
        Console.Write("name: {0}, ", user.Name);
        Console.Write("age: {0}, ", user.Age);
```

```

        Console.WriteLine("sex: {0}, ", user.Sex);
        Console.WriteLine("birthday: {0}, ", user.Birthday);
        Console.WriteLine("married: {0}.", user.Married);
        Console.WriteLine();
    }
});

Console.ReadKey();

```

下面是该程序在按顺序输出时的结果：

```

Key=January, Value=Jan
Key=Jan, Value=January
Key=January, Value=Jan
Key=January, Value=Jan
Key=Jan, Value=January
Key=Jan, Value=January
name: Amy, age: 26, sex: Female, birthday: 1983-12-3 0:00:00, married: True.
name: Bob, age: 20, sex: Male, birthday: 1989-6-12 0:00:00, married: False.
name: Chris, age: 29, sex: Unknown, birthday: 1980-3-8 0:00:00, married: True.
name: Alex, age: 17, sex: InterSex, birthday: 1992-6-14 0:00:00, married: False.

```

我们第一次对 SwapKeyAndValue 进行调用是调用的非引用参数传递的方法，结果如预期一样，map 没有改变。回调方法中的 arguments 也没有改变。

我们第二次对 SwapKeyAndValue 进行调用是调用的引用参数传递的方法，但在回调方法中您会发现 map 并没有改变，只有回调方法中的 arguments 改变了。这是因为异步调用的缘故，是正常的。

调用结果类型是通过在接口中使用范型委托指定的。如果您使用非范型委托来作为回调，则需要在委托之后再定义一个类型参数，通过它来传入返回结果类型。这样可以做到在调用时动态指定返回结果类型。

异常处理

同步调用异常处理

同步调用下的发生的异常将被直接抛出，使用 try...catch 语句块即可捕获异常，通常服务器端调用返回的异常是 HproseException 类型。但是在调用过程中也可能抛出其它类型的异常，为了保险，您可以使用 catch 捕获 Exception 类型来处理全部可能发生的异常。例如：

```

HproseHttpClient client = new HproseHttpClient("http://localhost:2010/");
IExam1 exam1 = client.UseService<IExam1>("ex1");
try {
    exam1.Sum(new double[] { double.NaN, double.NegativeInfinity });
}
catch (Exception e) {
    Console.WriteLine(e.Message);
}

```

运行结果如下：

```
Hprose.Common.HproseException: NaN can't change to System.Int32
在 Hprose.I0.HproseReader.CastError(String srctype, Type desttype)
在 Hprose.I0.HproseReader.Deserialize(Int32 tag, Type type)
在 Hprose.I0.HproseReader.ReadInt32Array(Int32 count)
在 Hprose.I0.HproseReader.ReadList(Boolean includeTag, Type type)
在 Hprose.I0.HproseReader.Deserialize(Int32 tag, Type type)
在 Hprose.Server.HproseService.DoInvoke(HproseMethods methods)
在 Hprose.Server.HproseService.Handle(HproseMethods methods)
```

异步调用异常处理

异步调用时，如果调用过程中发生异常，异常将不会被抛出。

例如下面这个程序：

```
HproseHttpClient client = new HproseHttpClient("http://localhost:2010/");
client.Invoke("HelloWorld", (result) => {
    Console.WriteLine(result);
});
Console.ReadKey();
```

调用了一个服务器端不存在的方法 HelloWorld，这个程序运行后，不会有任何输出，也不会有异常抛出，但确实有异常发生了。

如果您希望能够处理这些异常，只需要给 Hprose 客户端对象指定合适的 OnError 事件即可，OnError 事件的类型为 Hprose.Common.HproseErrorEvent，它是一个委托类型，使用非常简单，例如上面程序可以改为：

```
HproseHttpClient client = new HproseHttpClient("http://localhost:2010/");
client.OnError += (name, e) => {
    Console.WriteLine(name);
    Console.WriteLine(e.Message);
};
client.Invoke("HelloWorld", (result) => {
    Console.WriteLine(result);
});
Console.ReadKey();
```

这时运行结果变为：

```
HelloWorld
System.MissingMethodException: Can't find this method helloworld
在 Hprose.Server.HproseService.DoInvoke(HproseMethods methods)
在 Hprose.Server.HproseService.Handle(HproseMethods methods)
```

该事件第一个参数为抛出异常的方法名，第二个为抛出的异常。

该事件只对异步调用有效，同步调用下的异常将被直接抛出，而不会被该事件捕获并处理。

另外，在 Hprose 1.3 中，HproseErrorEvent 可以直接作为参数传入 Invoke 方法，当您需要为每个调用指定不同的错误处理事件时，就可以这样做了。

超时设置

Hprose 1.2 for C#及其之后的版本中增加了超时设置。只需要设置客户端对象上的 Timeout 属性即可，单位为毫秒。当调用超过 timeout 的时间后，调用将被中止，并触发错误事件。

HTTP 参数设置

目前的版本只提供了 http 客户端实现，针对于 http 客户端，有一些特别的设置，例如代理服务器、持久连接、http 标头等设置，下面我们来分别介绍。

代理服务器

默认情况下，代理服务器是被禁用的。可以通过 Proxy 属性进行设置，Proxy 为 IWebProxy 类型。SilverLight 客户端不支持该属性。

持久连接

默认情况下，持久连接是关闭的。通常情况下，客户端在进行远程调用时，并不需要跟服务器保持持久连接，但如果您有连续的多次调用，可以通过开启这个特性来优化效率。

跟持久连接有关的属性有两个，它们分别是 KeepAlive 和 KeepAliveTimeout，将 KeepAlive 属性设置为 true 时，表示开启持久连接特征。KeepAliveTimeout 表示持久连接超时时间，单位是秒，默认值是 300 秒。

SilverLight 客户端不支持这两个属性。

HTTP 标头

有时候您可能需要设置特殊的 http 标头，例如当您的服务器需要 Basic 认证的时候，您就需要提供一个 Authorization 标头。设置标头很简单，只需要调用 SetHeader 方法就可以啦，该方法的第一个参数为标头名，第二个参数为标头值，这两个参数都是字符串型。如果将第二个参数设置为 null，则表示删除这个标头。

标头名不可以为以下值：

- Context-Type
- Content-Length
- Connection
- Keep-Alive
- Host

因为这些标头有特别意义，客户端会自动设定这些值。

另外，Cookie 这个标头不要轻易去设置它，因为设置它会影响 Cookie 的自动处理，如果您的通讯中用到了 Session，通过 SetHeader 方法来设置 Cookie 标头，将会影响 Session 的正常工作。

调用结果返回模式

有时候调用的结果需要缓存到文件或者数据库中，或者需要查看返回结果的原始内容。这时，单纯的普通结果返回模式就有些力不从心了。Hprose 1.3 提供更多的结果返回模式，默认的返回模式是 Normal，开发者可以根据自己的需要将结果返回模式设置为 Serialized，Raw 或者 RawWithEndTag。

Serialized 模式

Serialized 模式下，结果以序列化模式返回，在 C# 中，序列化的结果以 MemoryStream 类型返回。用户可以将其转化为 byte[] 后，通过 HproseFormatter.unserialize 方法来将该结果反序列化为普通模式的结果。因为该模式并不对结果直接反序列化，因此返回速度比普通模式更快。

在调用时，通过在回调方法参数之后，增加一个结果返回模式参数来设置结果的返回模式，结果返回模式是一个枚举值，它的有效值在 HproseResultMode 枚举中定义。

Raw 模式

Raw 模式下，返回结果的全部信息都以序列化模式返回，包括引用参数传递返回的参数列表，或者服务器端返回的出错信息。该模式比 Serialized 模式更快。

RawWithEndTag 模式

完整的 Hprose 调用结果的原始内容中包含一个结束符，Raw 模式下返回的结果不包含该结束符，而 RawWithEndTag 模式下，则包含该结束符。该模式是速度最快的。

这三种模式主要用于实现存储转发式的 Hprose 代理服务器时使用，可以有效提高 Hprose 代理服务器的运行效率。