用户手册

1.3

(Pascal 版)

目录

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 PHPRPC 的进化版本 ,它除了拥有 PHPRPC 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	+86-010-80680756（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 Pascal 进行分布式程序开发的全过程。

本章提要

- 在 Delphi 中创建 Hello 客户端
- 在 C++ Builder 中创建 Hello 客户端
- 在 Lazarus 中创建 Hello 客户端

安装 Hprose for Pascal

Hprose for Pascal 支持的开发环境包括 Delphi/C++ Builder 6-2012。

Hprose for Pascal 同样支持 Lazarus 0.9.26/Free Pascal 2.2.0 及其更高版本。

在不同开发环境下，安装方法稍有不同。下面我们来分别介绍。

在 Delphi 中安装

Hprose for Pascal 直接提供了相应的组件安装包，只需要打开组件包点击安装即可。安装后，可以在组件面板的 Hprose 栏下找到您所安装的组件。

Hprose 提供了 4 个版本的基于 Indy 实现的组件安装包，分别对应 Indy8、Indy9、Indy10 和 Indy10.5，默认情况下，Delphi 6 自带的是 Indy8，Delphi 7 自带的是 Indy9，更高版本是 Indy10，Delphi 2011 及其更高版本是 Indy10.5。您也可以在 Delphi 6、7 上安装 Indy10，但需要自行安装相应版本的 Indy,并且需要源码版本的 Hprose 自行编译安装。这三个版本在使用上基本没有差别，在效率上版本越高性能越佳。

Hprose 还提供了基于 synapse 实现的组件，文件名为 HproseSynaHttpClient.pas，但是没有直接提供安装包。因为它需要先安装 synapse，或者同它一起安装，但 synapse 是以修改的 LGPL 许可证的发布的，因此不便于与它一起打包发布，但是这并不会影响您的使用。synapse 可以直接从它的官方网站获取，它官方下载地址是：http://www.synapse.ararat.cz/doku.php/download。但是您可能并不需要 synapse 的全部内容，并且因为 synapse 本身有一点小问题，您可能并不能幸运的一次编译通过，所以我们在这里提供了修正并精简过的版本：http://www.hprose.com/download/synapse.zip，它是免费开源的，其中还包含了 Delphi 和 Lazarus 的安装包。至于 HproseSynaHttpClient.pas 的安装，您可以参见 Indy 版本的安装包在 Delphi 中创建自己的 synapse 版本安装包并安装。

在 C++ Builder 中安装

C++ Builder 可以直接使用 Delphi 文件，虽然没有直接提供 bpk 安装包，但您可以直接在 C++ Builder 中创建组件包，并添加相应的 pas 文件编译安装即可。

在 Lazarus 中安装

Hprose 同样没有直接提供 Lazarus 的安装包，但是如果您使用过 Lazarus，应该已经熟悉在 Lazarus 中创建安装包的过程啦。

如果您希望使用 Indy 版本，那么首先从这里(http://www.indyproject.org/Sockets/fpc/index.en.aspx) 下载 Indy。然后参考 Free Pascal/Lazarus 官方 wiki(http://wiki.lazarus.freepascal.org/Indy_with_Lazarus) 来安装 Indy for Lazarus。最后再创建 Hprose 的 Indy 版本组件包安装即可。

synapse 版本的 Hprose 安装请参考在 Delphi 中安装的方法。

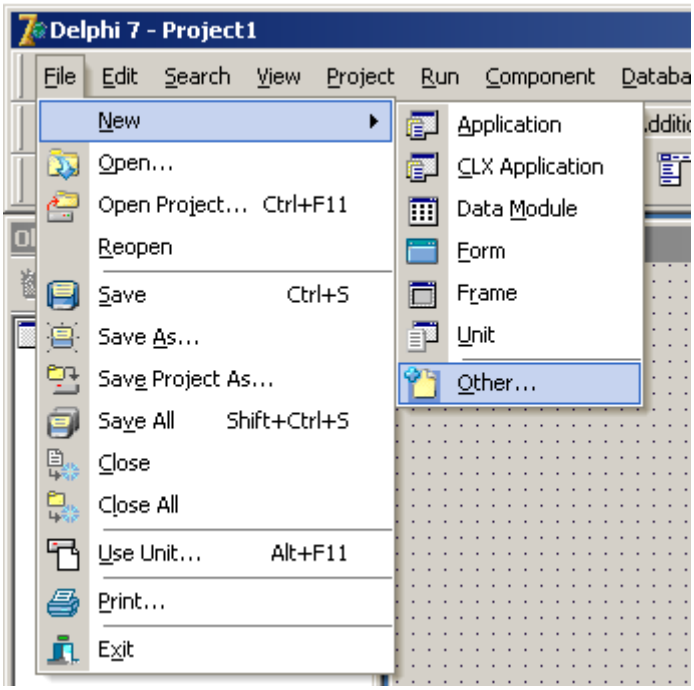
在 Delphi 中创建 Hello 客户端

Hprose for Pascal 目前的版本只提供了客户端功能，因此我们还需要一个服务器才能进行讲解和演示。不过这并不是什么问题，Hprose 的客户端和服务器所使用的语言是没有必然联系的，不论服务器是 Java、

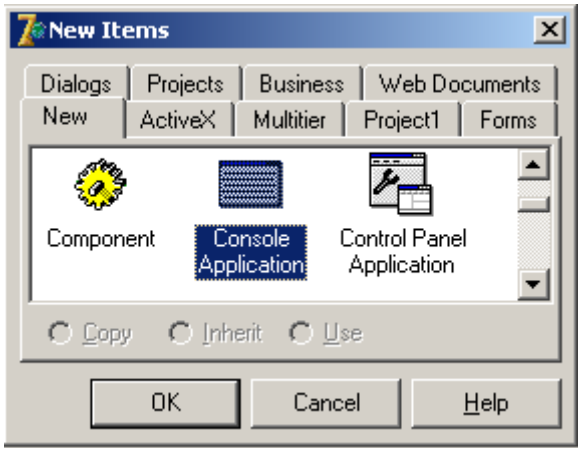
C#还是 PHP ,对客户端来说都是一样的。所以这里我们将使用 PHP 版本的 Hprose 服务器为例来进行讲解。为了方便大家可以直接使用本书中的例子进行练习，我们直接将服务器放在了官方网站上，地址为：http://www.hprose.com/example/，这是一个实际可用的服务，后面所有的例子，都以这个地址所提供的服务为例来进行说明。

这里我们以 Delphi7 和 Indy 版本的客户端为例来说明如何创建一个简单的 Hello 客户端。

首先确认在 Delphi7 已经安装了 Indy 组件，然后根据第一章中介绍的方法安装好 Indy 版本的 Hprose 客户端。这里有一点需要注意，安装之后，如果后面您发现编写程序时找不到相应的单元文件，很可能是它们不在搜索路径中，解决这个问题的最简单方法就是将编译出来的所有 dcu 文件复制到 Delphi 安装目录下的 Lib 目录下，另外，Hprose.inc 文件也要复制进去，这个不要忘记。然后在 Delphi7 中，选择“File→New→Other”:



之后选择“New→Console Application”:



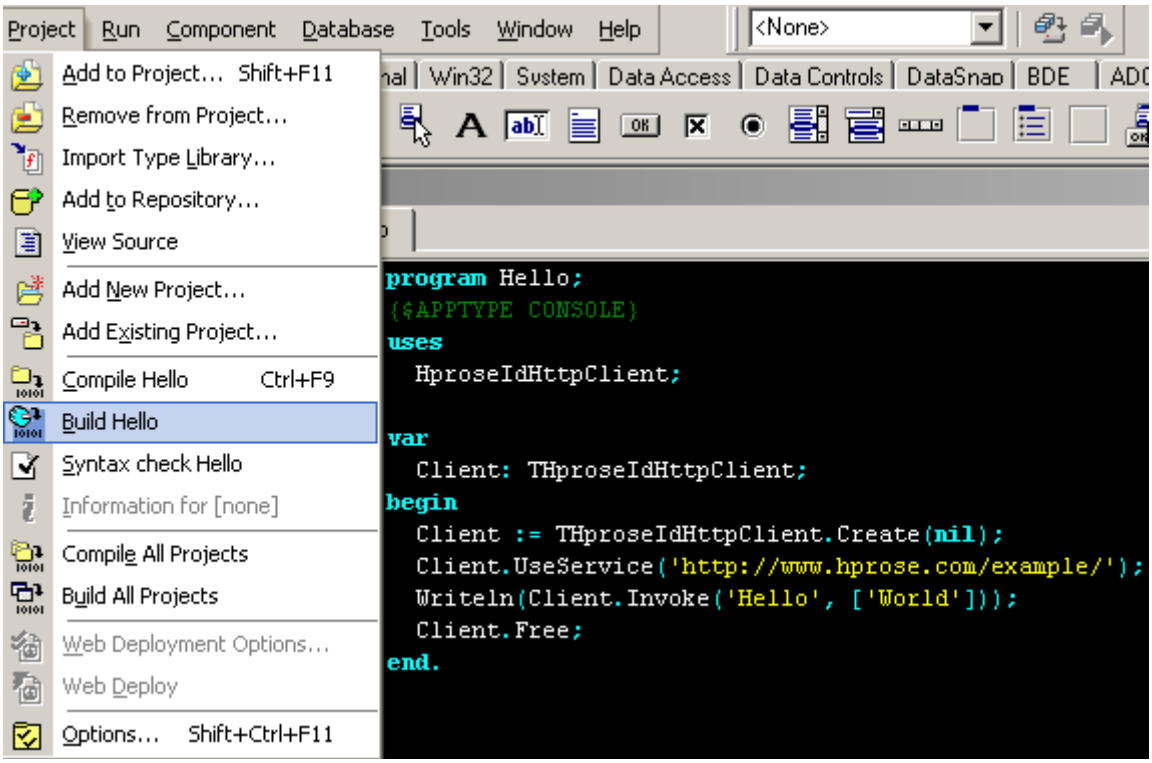
创建一个控制台程序，接下来我们在编辑窗口中键入以下程序：

```
program Hello;
{$APPTYPE CONSOLE}
uses
```

```
  HproseIdHttpClient;
var
  Client: THproseIdHttpClient;
begin
  Client := THproseIdHttpClient.Create(nil);
  Client.UseService('http://www.hprose.com/example/');
  Writeln(Client.Invoke('Hello', ['World']));
  Client.Free;
end.
```

然后选择“File→Save As...”, 将其保存为 Hello.dpr。

接下来选择“Project→Build Hello”, 如果报告找不到 HproseIdHttpClient 单元，则需要按照前面所述步骤检查是否正确地安装了 Hprose 的 Indy 版本组件包。如果顺利的话，程序将会被成功编译。



这是打开 Windows 命令行管理器，进入 Hello.dpr 所在目录，键入 Hello.exe 回车，您将会看到以下输出：

```
Hello World
```

这说明您的客户端已经正常运行了，当然，如果这时您没有联网，您可能会得到一个错误提示窗口，并在控制台窗口上显示类似于下面这样的错误信息：

```
Exception EIdSocketError in module Hello.exe at 0002D5D0.
Socket Error # 11001
Host not found.
```

这很正常，这恰恰说明了这个程序不是单纯在本地执行的，而是通过网络才获取到的结果。

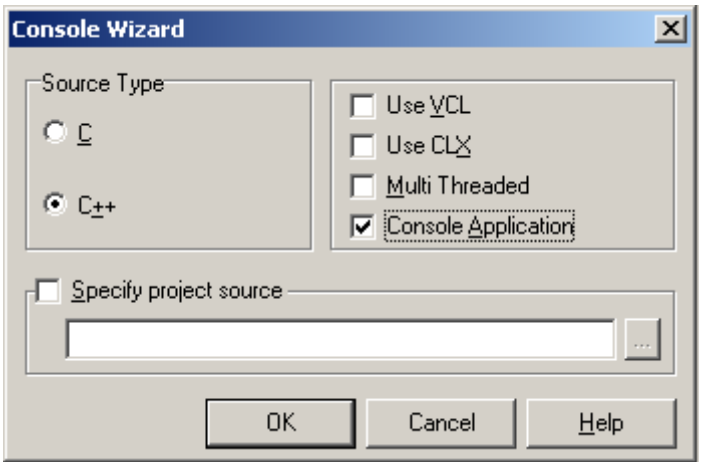
下面我们来看一下如何在 BCB 中使用 Hprose。

在 C++ Builder 中创建 Hello 客户端

这里我们以 Borland C++ Builder 6.0（后面我们简称 BCB6）为例来进行说明。

首先同样是安装 Indy，可以直接从 Indy 官网下载 Indy10 安装，关于 Indy 的安装这里就不做介绍了。之后创建组件包安装 Hprose，其中 HproseIdHttpClient.pas 一定要选择正确的版本目录下的安装。编译完之后会生成一些 hpp 和 obj 文件，将 hpp 文件复制到 BCB6 安装目录下的 Include 目录下，将 obj 文件复制到 BCB6 安装目录的 Lib/Obj 目录下。

然后我们通过选择“File→New→Other”打开新建选项面板，之后选择“New→Console Wizard”，创建控制台程序：



选择 C++，并且只选择控制台程序即可。

然后将编辑窗口中的代码改为如下内容：

```
#include <iostream>
#include <HproseIdHttpClient.hpp>
using namespace std;
int main(int argc, char* argv[]) {
    THproseIdHttpClient *Client = new THproseIdHttpClient(NULL);
    Client->UseService("http://www.hprose.com/example/");
    TVarRec v[] = { "World" };
    cout << VarToStr(Client->Invoke("Hello", v, 1));
    Client->Free();
}
```

现在如果直接编译，可能会在链接时报错，这个只要把安装 Hprose 时生成的 Obj 文件或者 lib 文件添加到项目中，就可以正常编译了。

但是这样编译出来的程序依然依赖于 bpl，如果想脱离 bpl，独立运行，需要这样几个步骤：

1. 在 Project Options 的 Packages 面板中将 Build with runtime packages 的选项禁止。
2. 在 Project Options 的 Compiler 面板中点击 Release 按钮。
3. 在 Project Options 的 Linker 面板中将 Use dynamic RTL 禁止。

4. 最后将 rtl.lib 添加到工程中。

这样再编译出来的 Hello.exe 就可以独立运行了。这个程序的运行结果跟 Delphi 版本是完全一致的。

也就是说，虽然 Hprose for Pascal 是使用 Object Pascal 写的，但是您可以在 BCB 中通过 C++来使用它。而且还是 Windows 原生程序。

Hprose for Pascal 同样支持跨平台的 Free Pascal/Lazarus。

在 Lazarus 中创建 Hello 客户端

为了体现出跨平台的特性，我们这里选择 Mac OS X 上的 Lazarus。

另外前面两个例子都是围绕 Indy 客户端来讲解的，我们这里以 synapse 版本的 Hprose 客户端为例来讲解。

首先您要安装并配置好 Lazarus、Free Pascal 及其源码，这里就不做介绍了。

然后是安装 synapse，这个很简单，从前面一章中所介绍的地址下载 synapse.zip，之后解压缩，其中有一个 synapse.lpk 的文件，这个就是安装包，双击后选择打开包。在包 synapse 面板上点击编译，安装即可。因为当前版本的 Lazarus 只支持静态方式编译，所以组件包要随同 IDE 一起被编译。不过这个不需要多长时间，经过很短的等待，Lazarus 就会编译安装完，并且自动重启。synapse 是非可视组件，所以您在组件面板上并找不到它们。

接下来是安装 Hprose，选择“Package→New Package”，然后在包面板上点添加，将：

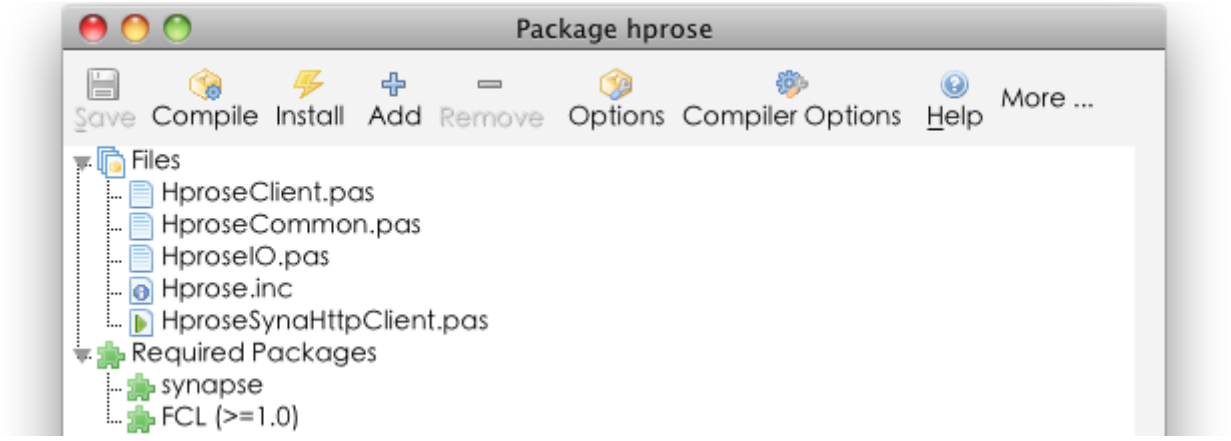
- HproseCommon.pas
- HproseIO.pas
- HproseClient.pas
- HproseSynaHttpClient.pas

作为单元添加到包中。

然后将 Hprose.inc 作为包含文件添加到包中。

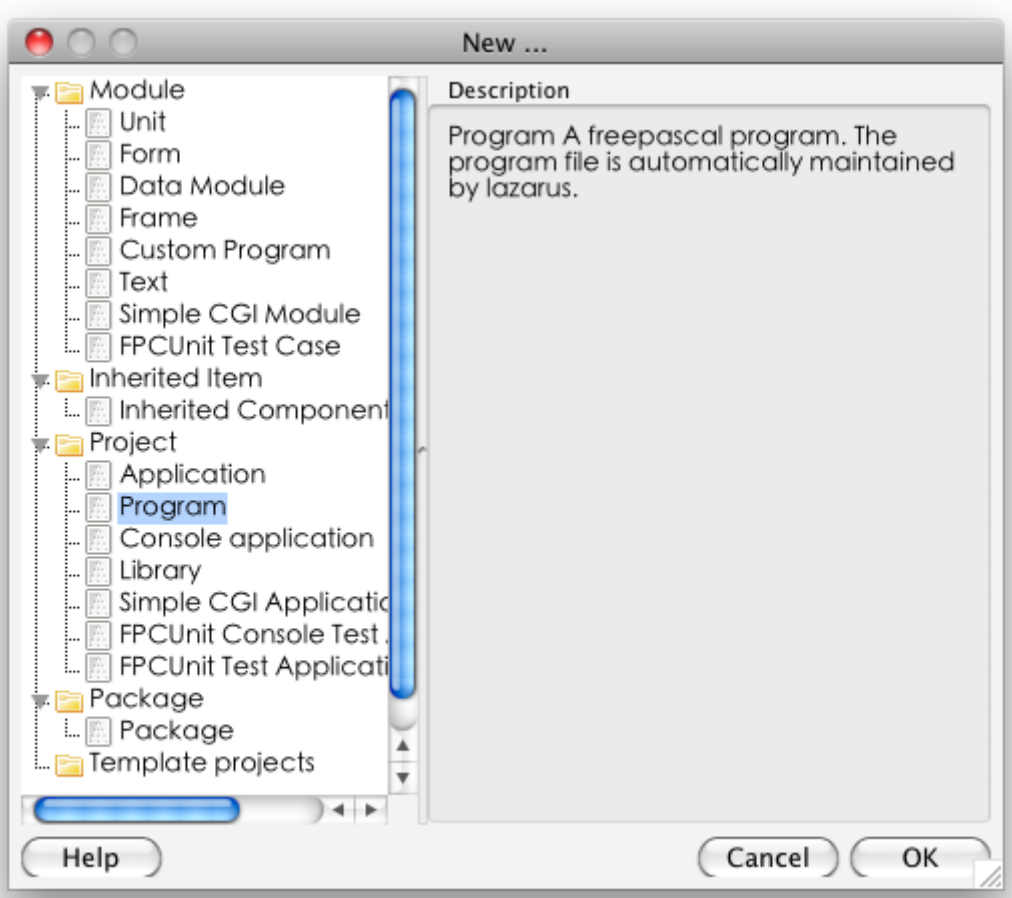
接下来将 synapse 作为需求的包添加到包中。

最后保存该包，例如叫 hprose.lpk，它同时会生成一个 hprose.pas 文件，您不需要对它做任何的修改。



之后点击编译，如果没有问题的话，点击安装即可。跟安装 synapse 组件一样，经过一段等待后，Lazarus 会自动重启。之后您就可以在组件面板上找到 Hprose 了。

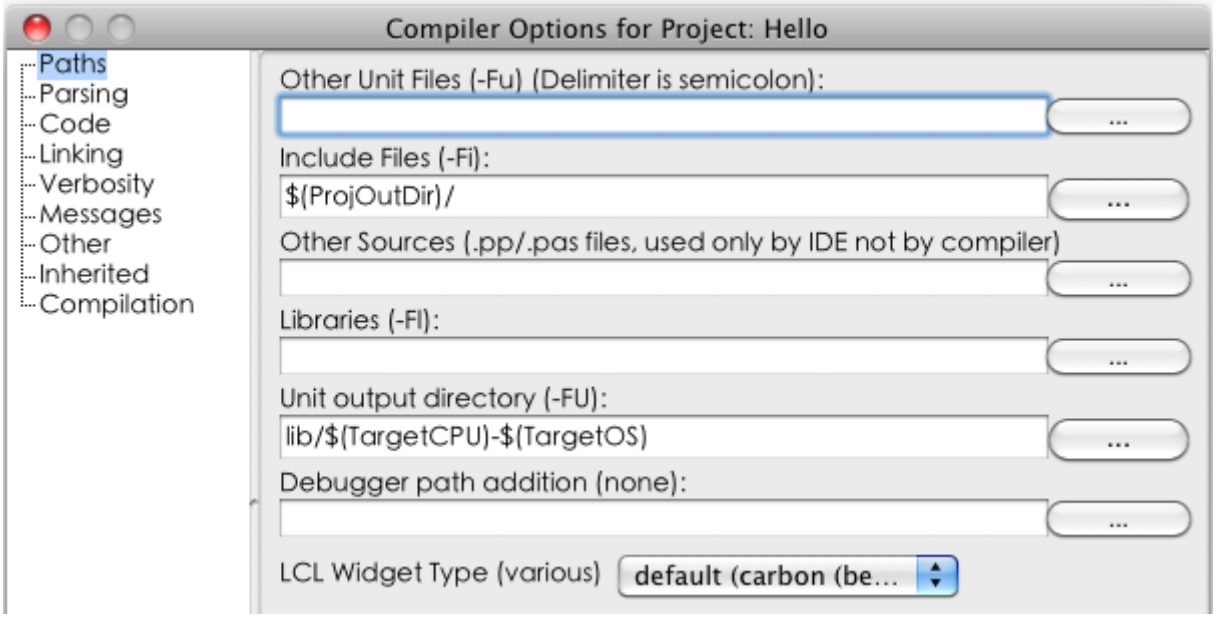
接下来，选择“File→New...”，在新建面板上选择“Project→Program”：



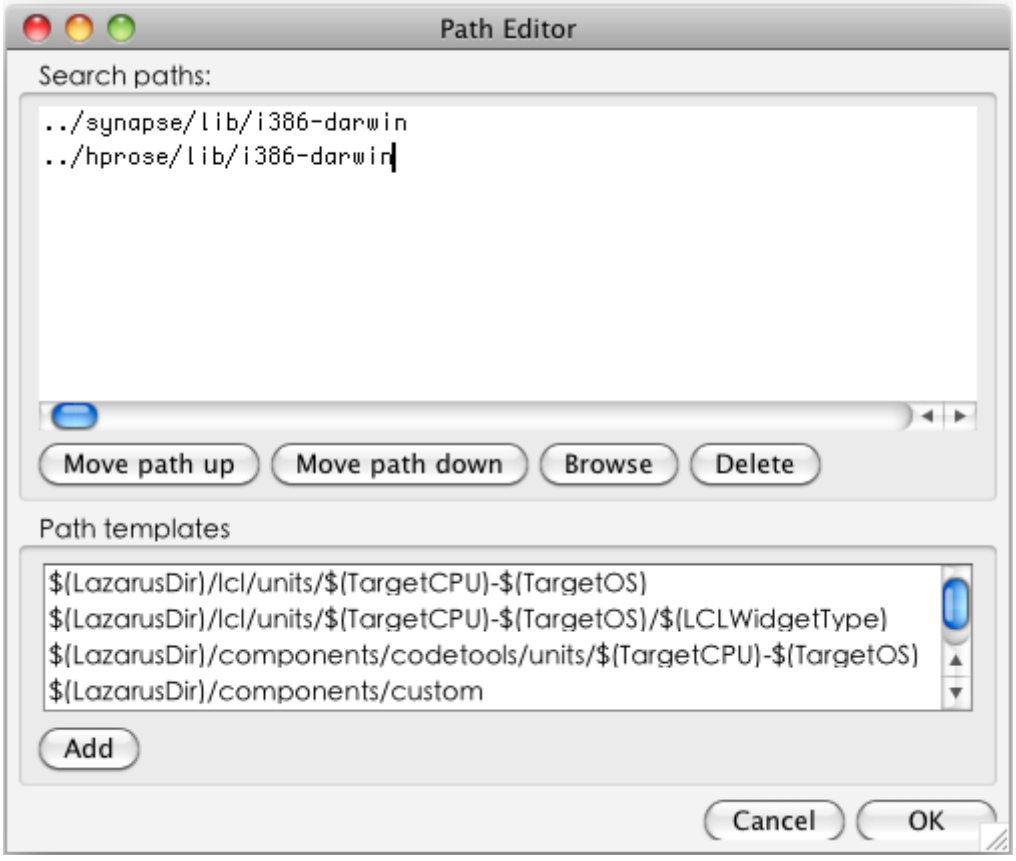
点击确定，然后在编辑窗口中将代码改写为：

```
program Hello;
uses
  HproseSynaHttpClient;
var
  Client: THproseSynaHttpClient;
begin
  Client := THproseSynaHttpClient.Create(nil);
  Client.UseService('http://www.hprose.com/example/');
  Writeln(Client.Invoke('Hello', ['World']));
  Client.Free;
end.
```

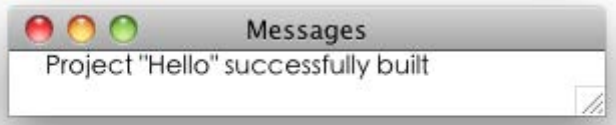
您会发现上面的程序除了 HproseIdHttpClient 换成了 HproseSynaHttpClient 以外，基本上与在 Delphi 中的程序是一样的。现在先不要着急编译，否则可能会发现编译出错，首先需要打开“Project→Compiler Options...”:



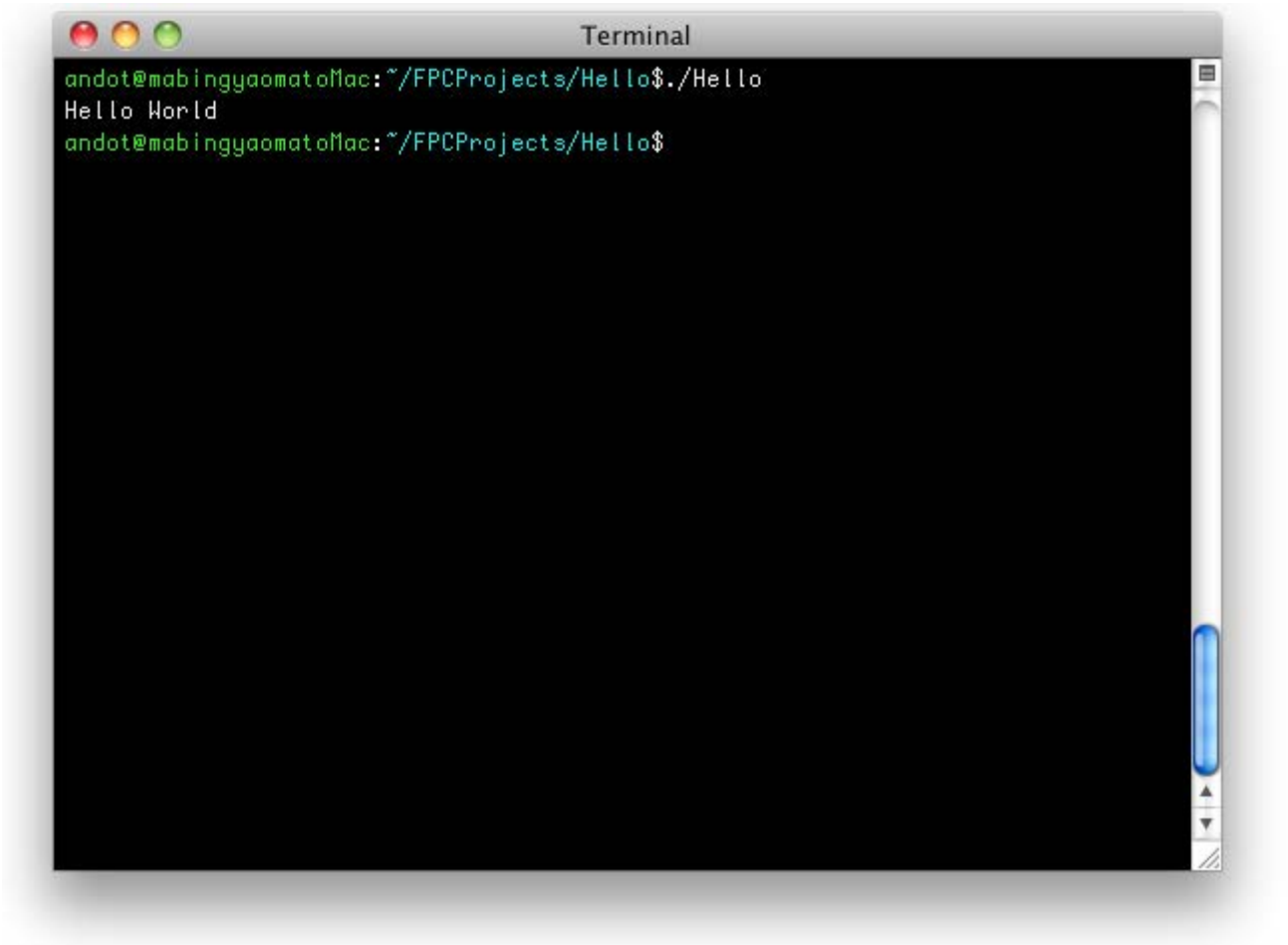
然后选择“路径→补充单元文件(-Fu)（分号格开）”，设置好 synapse 和 hprose 的路径：



之后选择“Run→Build”，成功后会有如下提示：



然后在终端上切换到该程序目录下，执行改程序会得到如下结果：



到这里 ,您应该已经掌握 Hprose for Pascal 的基本用法啦。接下来 ,就让我们一起对 Hprose for Pascal 进行深层探秘吧。

第二章 通用工具

Hprose for Pascal 除了提供方便的远程调用功能以外，还提供了许多灵活的通用工具，虽然这些工具本身是为了方便使用远程调用而准备的，但是您仍然可以将它们用于其它目的，熟练掌握这些通用工具的使用，将会使您的工作事半功倍。

本章提要

- 可变类型的 List
- 可变类型的 Map
- 可序列化自定义类型
- 其它帮助方法

可变类型的 List

尽管 Delphi 的 VCL/CLX 和 Free Pascal 的 FCL 本身提供了一些 List 类型(例如 :TList ,TInterfaceList，TStringList 等)，但这些 List 类型在处理混合类型数据时并不方便。

比如 TList 中的元素都是无类型指针，虽然指针可以指向任何类型，但是您却无法在同一个 TList 对象中同时存放指向几种不同类型数据的指针，因为这样原本的数据类型在 TList 对象中丢失了，当取出元素指针时您无法获知这个指针指向的究竟是何种类型的数据。

而 TInterfaceList 的元素都是接口，无法存放接口以外的类型。

TStringList 既可以作为一个纯粹存放字符串元素的 List ,也可以作为一个通过字符串来存取对象的 Map，不过它不是通过 Hash 方式查找的，所以效率不高，操作方式也不同于一般的 Map 类型，不是很方便。而且，您用它也无法存取字符串和对象以外的类型。

为了解决这个问题，Hprose 提供了一套可变类型的 List。

IList 接口

Hprose 提供的所有 List 类都实现了 IList 接口 ,定义接口的好处是 ,当您使用接口来存取 List 对象时，所有的 List 对象您都可以使用统一的方式来操作，而且您不需要关心它的生存周期，也不用担心忘记 Free 而产生内存泄漏啦。

IList 接口中定义了枚举器操作 ,因此 ,在高版本的 Delphi(Delphi 2005 以上)或 FreePascal(FreePascal 2.5.1 以上) 中您可以通过 for...in 语句来操作 IList 接口的对象。

IList 接口上定义了添加(Add、AddAll),插入(Insert),移动(Move),交换(Exchange),查找(Contains、IndexOf)，删除 (Delete、Remove)，清空 (Clear)，复制 (Assign)，同步 (Lock、UnLock) 等操作和到动态数组和可变数组转换的帮助方法 (ToArray)。

IList 接口还定义了直接通过索引返回元素的默认属性，可以让您向操作数组一样存取元素。另外，它还定义了容量 (Capacity) 和元素个数 (Count) 这两个属性。

TAbstractList 类

该类是所有可变类型 List 的基类，它实现了 IList 接口上的枚举器，复制和同步操作。它继承自 TInterfacedObject，因此它也继承了接口生存周期管理。它是一个抽象类，您不应该对它进行实例化。

如果您打算实现自己的可变类型 List，那么您应该直接或间接的继承自它。因为在 Hprose 序列化和反序列化时，判断一个类是否是可变类型的 List 是通过判断 TAbstractList 是否是这个类的祖先类的方式完成的。

该类上还实现了两个比较有用的方法：Join 和 Split。但因为 TAbstractList 是抽象类，所以您只能在它的可实例化子类上调用它们。这两个方法的具体用法 ,我们在下面介绍 TArrayList 类的小节中来详细介绍。

TArrayList 类

该类直接继承自 TAbstractList，它是最常用的 List。它内部是通过动态数组来实现的。

创建 TArrayList 对象

TArrayList 的构造方法定义如下：

```
constructor Create(Capacity: Integer = 4);
```

这个构造方法需要指定一个初始化容量，如果您事先确知要放入的元素个数，那么将容量设置为与元素个数相同或多于元素个数，可以避免后面放入元素时内存的重新分配。默认初始化容量为 4。

初始化容量不代表这个 List 只能放多少个元素，也不代表这个 List 有多少个元素。它只代表在不重新分配内存的情况下，能放入的元素个数。当元素个数增长到超过这个容量时，这个容量会自动扩大，您不需要做任何特殊操作。

C++ Builder 不支持带默认参数的构造方法，不过我们已经对 C++ Builder 作了特殊处理，所以在 C++ Builder 中有两个构造方法，它们分别对应不带默认参数和带默认参数的：

```
__fastcall virtual TArrayList(int Capacity);
__fastcall virtual TArrayList(void);
```

添加元素

您可以通过 Add 方法一次添加一个元素，或通过 AddAll 方法一次添加多个元素。看下面的例子：

```
procedure AddDemo;
var
  List, List2: IList;
  I: Integer;
begin
  List := TArrayList.Create;
  List.Add(1);
  List.Add('abc');
  List.Add(3.14);
  List.Add(True);
  List2 := TArrayList.Create;
  List2.AddAll(List);
  List2.AddAll(VarArrayOf(['a','b','c']));
  for I := 0 to List2.Count - 1 do Writeln(List2[I]);
  List2.Add(List);
  List := VarToList(List2[List2.Count - 1]);
  for I := 0 to List.Count - 1 do Writeln(List[I]);
end;
```

通过这个例子，您会发现可以添加任意类型（可转换为 Variant 类型的类型）的数据到 IList 对象中，甚至 IList 对象本身也可以作为元素添加到 IList 对象中，并且可通过 AddAll 将 IList 对象或者动态数组中的元素批量添加到 IList 对象中。

上面例子的运行结果为：

```
1
abc
3.14
True
a
b
c
1
abc
3.14
True
```

上面程序的 C++ Builder 写法如下：

```
void AddDemo() {
    _di_IList List = (_di_IList)*new TArrayList;
    List->Add(1);
    List->Add("abc");
    List->Add(3.14);
    List->Add(true);
    _di_IList List2 = (_di_IList)*new TArrayList;
    List2->AddAll(List);
    Variant A[3];
    A[0] = "a";
    A[1] = "b";
    A[2] = "c";
    List2->AddAll(VarArrayOf(A, 2));
    for (int i = 0; i < List2->Count; i++) {
        cout << List2->Item[i] << endl;
    }
    List2->Add(static_cast<IUnknown*>(List));
    List = VarToList(List2->Item[List2->Count - 1]);
    for (int i = 0; i < List->Count; i++) {
        cout << List->Item[i] << endl;
    }
}
```

在 C++ Builder 中使用 IList 接口时需要注意，这里的接口变量需要用_di_IList 类型来声明，而不要用 IList*，否则在对象的生存周期管理上会出问题。在这个程序中我们可以更清楚的看到 List 本身可以作为元素添加是因为它可以转换为 IUnknown*类型，而 IUnknown*类型是可以转换为 Variant 类型的，但 C++ Builder 不向 Delphi 那样可以自动转换，所以这里需要作显示转换类型。运行结果稍有不同，cout 不但打印出了数据，还打印出了类型：

```
varInteger: 1
varString: abc
```

```
varDouble: 3.14
varBoolean: 1
varOleStr: a
varOleStr: b
varOleStr: c
varInteger: 1
varString: abc
varDouble: 3.14
varBoolean: 1
```

插入元素

使用 Insert 方法可以在列表的任何位置插入元素，例如：

```
procedure InsertDemo;
var
  List: IList;
  I: Integer;
begin
  List := TArrayList.Create;
  List.Add(1);
  List.Add('abc');
  List.Add(3.14);
  List.Add(True);
  List.Insert(0, 'top');
  List.Insert(3, 'middle');
  List.Insert(6, 'bottom');
  for I := 0 to List.Count - 1 do Writeln(List[I]);
end;
```

运行结果如下：

```
top
1
abc
middle
3.14
True
bottom
```

上面程序的 C++ Builder 写法如下：

```
void InsertDemo() {
   _di_IList List = (_di_IList)*new TArrayList;
   List->Add(1);
```

```
    List->Add("abc");
    List->Add(3.14);
    List->Add(true);
    List->Insert(0, "top");
    List->Insert(3, "middle");
    List->Insert(6, "bottom");
    for (int i = 0; i < List->Count; i++) {
        cout << List->Item[i] << endl;
    }
}
```

运行结果如下：

```
varString: top
varInteger: 1
varString: abc
varString: middle
varDouble: 3.14
varBoolean: 1
varString: bottom
```

移动元素

通过 Move 方法可以将一个元素从一个位置移动到另一个位置，例如：

```
procedure MoveDemo;
var
  List: IList;
  I: Integer;
begin
  List := TArrayList.Create;
  List.Add(1);
  List.Add('abc');
  List.Add(3.14);
  List.Add(True);
  List.Move(2, 0);
  for I := 0 to List.Count - 1 do Writeln(List[I]);
end;
```

运行结果如下：

```
3.14
1
abc
True
```

上面程序的 C++ Builder 写法如下：

```
void MoveDemo() {
    _di_IList List = (_di_IList)*new TArrayList;
    List->Add(1);
    List->Add("abc");
    List->Add(3.14);
    List->Add(true);
    List->Move(2, 0);
    for (int i = 0; i < List->Count; i++) {
        cout << List->Item[i] << endl;
    }
}
```

运行结果如下：

```
varDouble: 3.14
varInteger: 1
varString: abc
varBoolean: 1
```

交换元素

通过 Exchange 方法可以将一个元素从一个位置移动到另一个位置，例如：

```
procedure ExchangeDemo;
var
  List: IList;
  I: Integer;
begin
  List := TArrayList.Create;
  List.Add(1);
  List.Add('abc');
  List.Add(3.14);
  List.Add(True);
  List.Exchange(2, 0);
  for I := 0 to List.Count - 1 do Writeln(List[I]);
end;
```

运行结果如下：

```
3.14
abc
1
True
```

上面程序的 C++ Builder 写法如下：

```
void ExchangeDemo() {
    _di_IList List = (_di_IList)*new TArrayList;
    List->Add(1);
    List->Add("abc");
    List->Add(3.14);
    List->Add(true);
    List->Exchange(2, 0);
    for (int i = 0; i < List->Count; i++) {
        cout << List->Item[i] << endl;
    }
}
```

运行结果如下：

```
varDouble: 3.14
varString: abc
varInteger: 1
varBoolean: 1
```

查找元素

通过 Contains 或 IndexOf 方法我们可以确认一个元素是否在 IList 对象，例如：

```
procedure FindDemo;
var
  List: IList;
begin
  List := TArrayList.Create;
  List.Add(1);
  List.Add('abc');
  List.Add(3.14);
  List.Add(True);
  Writeln(List.Contains('hello'));
  Writeln(List.Contains('abc'));
  Writeln(List.IndexOf(3.14));
  Writeln(List.IndexOf(False));
end;
```

运行结果如下：

```
FALSE
TRUE
2
```

```
-1
```

通过上面的例子我们还可以看出 Contains 或 IndexOf 方法的不同之处 ,Contains 方法只返回存在与否，IndexOf 方法会返回具体位置，如果不存在，则返回-1。

这两个方法在 TArrayList 中的时间复杂度都是 O(n)，它们都是按照从头到尾的顺序查找方式进行查找的，因此，当查找的值在 TArrayList 对象中有多个相同元素对应时，则遇到第一个匹配的元素时就返回结果，在 TArrayList 对象上 IndexOf 返回的索引值是相同元素的索引值中最小的，这与后面将要介绍的 THashedList 和 TCaseInsensitiveHashedList 的行为不同，后面我们在介绍它们时再举例说明。

上面程序的 C++ Builder 写法如下：

```
void FindDemo() {
    _di_IList List = (_di_IList)*new TArrayList;
    List->Add(1);
    List->Add("abc");
    List->Add(3.14);
    List->Add(true);
    cout << List->Contains("hello") << endl;
    cout << List->Contains("abc") << endl;
    cout << List->IndexOf(3.14) << endl;
    cout << List->IndexOf(false) << endl;
}
```

运行结果如下：

```
0
1
2
-1
```

删除元素

通过 Delete 或 Remove 都可以删除元素，不同的是：

Delete 是根据索引删除并返回删除的元素；

Remove 是根据元素删除并返回删除的索引。

例如：

```
procedure DeleteDemo;
var
  List: IList;
  I: Integer;
begin
  List := TArrayList.Create;
  List.Add(1);
  List.Add('abc');
```

```
  List.Add(3.14);
  List.Add(True);
  Writeln(List.Delete(2));
  Writeln(List.Remove('abc'));
  Writeln;
  for I := 0 to List.Count - 1 do Writeln(List[I]);
end;
```

该程序运行结果如下：

```
3.14
1

1
True
```

上面程序的 C++ Builder 写法如下：

```
void DeleteDemo() {
    _di_IList List = (_di_IList)*new TArrayList;
    List->Add(1);
    List->Add("abc");
    List->Add(3.14);
    List->Add(true);
    cout << List->Delete(2) << endl;
    cout << List->Remove("abc") << endl;
    cout << endl;
    for (int i = 0; i < List->Count; i++) {
        cout << List->Item[i] << endl;
    }
}
```

运行结果如下：

```
varDouble: 3.14
1

varInteger: 1
varBoolean: 1
```

清空

通过 Clear 方法可以清空所有元素，如果要删除 IList 对象中的所有元素，使用 Clear 比使用 Delete 要快很多。

该方法非常简单，这里不再举例说明。

复制

通过 Assign 方法可以将本对象变成指定对象的一个副本。

该方法非常简单，这里不再举例说明。

同步

当您在多个线程中操作同一个 IList 对象时，为了保证数据的线程安全性，需要对这个对象进行同步操作。通过 Lock、UnLock 方法可以非常方便的实现对 IList 对象自身的线程同步。只需要在操作该对象前，调用该对象的 Lock 方法，所有操作完成后，再调用 Unlock 方法即可。

该方法非常简单，这里不再举例说明。

在 Hprose 1.2 for Pascal 及其之后的版本中，所有的 IList 类都提供了一个带有 Sync 参数的构造函数，该参数默认值为 True。如果你不需要该类支持 Lock、UnLock 方法，那么可以将该参数设置为 False。在创建不带锁的 IList 对象后，还可以通过调用 InitLock 方法来初始化锁，之后 Lock 和 UnLock 方法就可以使用了。

Hprose 1.2 for Pascal 及其之后的版本中，所有的 IList 类还增加了多读单写锁的支持，如果需要该功能，在通过构造函数创建对象时，只需要将 ReadWriteSync 参数设置为 True 即可，该参数默认为 False。之后可以通过 BeginRead、EndRead、BeginWrite、EndWrite 这四个方法来操作读写锁。也可以在创建不带锁的对象后，单独调用 InitReadWriteLock 来初始化多读单写锁。

转换为数组

通过 ToArray 方法，可以将 IList 对象直接转换为动态数组。例如：

```
procedure ToArrayDemo;
var
  List: IList;
  A: array of Integer;
  I: Integer;
begin
  List := TArrayList.Create;
  List.Add(1);
  List.Add(3);
  List.Add(5);
  List.Add(7);
  List.Add(9);
  A := List.ToArray(varInteger);
  for I := 0 to Length(A) - 1 do Writeln(A[I]);
end;
```

运行结果如下：

```
1
3
5
7
```

```
9
```

上面程序的 C++ Builder 写法如下：

```
void ToArrayDemo() {
    _di_IList List = (_di_IList)*new TArrayList;
    List->Add(1);
    List->Add(3);
    List->Add(5);
    List->Add(7);
    List->Add(9);
    Variant A = List->ToArray(varInteger);
    for (int i = A.ArrayLowBound(); i <= A.ArrayHighBound(); i++) {
        cout << A.GetElement(i) << endl;
    }
}
```

运行结果如下：

```
varInteger: 1
varInteger: 3
varInteger: 5
varInteger: 7
varInteger: 9
```

这个 C++的例子里面是直接操作 Variant 数组，而不是转化为动态数组。

for...in 支持

如果您使用 Delphi 2005 或更高版本，或者 FreePascal 2.5.1 或更高版本，您可以通过 for...in 语句来枚举 IList 接口的元素：

```
procedure ForInDemo;
var
  List: IList;
  V: Variant;
begin
  List := TArrayList.Create;
  List.Add(1);
  List.Add('abc');
  List.Add(3.14);
  List.Add(True);
  for V in List do Writeln(V);
end;
```

分割字符串

通过 TArrayList 上的类方法 Split，可以直接将一个字符串分割为 TArrayList 对象。

Split 有五个参数，这五个参数的含义分别是：

Str 表示待分割的字符串。

Separator 表示分割符，默认值是逗号“,"，它可以是单个字符，也可以是字符串。如果该分割符为空字符串，则返回只包含一个完整 Str 元素的 TArrayList 对象。

Limit 表示最多分割为几个元素，默认值是 0，表示没有限制。

TrimItem 表示是否去掉分割后的字符串的首尾空格，默认为 false。

SkipEmptyItem 表示是否忽略空串元素，默认为 false。

下面我们来看例子：

```
procedure SplitDemo;
  procedure PrintList(List: IList);
  var
    I: Integer;
  begin
    for I := 0 to List.Count - 1 do Writeln(List[I]);
    Writeln('--------');
  end;
var
  S: String;
begin
  S := 'Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday,';
  PrintList(TArrayList.Split(S));
  PrintList(TArrayList.Split(S, ', '));
  PrintList(TArrayList.Split(S, ',', 4));
  PrintList(TArrayList.Split(S, ',', 0, true));
  PrintList(TArrayList.Split(S, ',', 0, true, true));
end;
```

该程序的运行结果为：

```
Monday
 Tuesday
 Wednesday
 Thursday
 Friday
 Saturday
 Sunday
--------
Monday
Tuesday
Wednesday
```

```
Thursday
Friday
Saturday
Sunday,
--------
Monday
 Tuesday
 Wednesday
 Thursday, Friday, Saturday, Sunday,
--------
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
Sunday
--------
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
Sunday
--------
```

这五个参数的作用和区别通过上面例子的运行结果可以很好的反应出来（注意开头空格、空行、和结尾逗号的区别）。

因为 Split 是类方法，在 C++ Builder 中无法直接调用它，所以我们提供了一个帮助函数 ListSplit，它实现的是相同的功能，而且它在 C++ Builder 中也是可用的。

上面程序的 C++ Builder 写法如下：

```
void PrintList(_di_IList List) {
   for (int i = 0; i < List->Count; i++) {
      cout << VarToStr(List->Item[i]) << endl;
   }
   cout << "--------" << endl;
}
void SplitDemo() {
   AnsiString S = "Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday,";
   PrintList(ListSplit(__classid(TArrayList), S));
   PrintList(ListSplit(__classid(TArrayList), S, ", "));
   PrintList(ListSplit(__classid(TArrayList), S, ",", 4));
   PrintList(ListSplit(__classid(TArrayList), S, ",", 0, true));
```

```
    PrintList(ListSplit(__classid(TArrayList), S, ",", 0, true, true));
}
```

运行结果跟 Delphi 和 FreePascal 是完全一致的，这里不再重复。

连接字符串

如果您的 IList 对象中所有的元素都是字符串，或者都可以转换为字符串类型，那么您可以通过 Join 方法，将这些元素以指定的方式连接成一个字符串，例如：

```
procedure JoinDemo;
var
  S: String;
  List: IList;
begin
  S := 'Monday,Tuesday,Wednesday,Thursday,Friday,Saturday,Sunday';
  List := TArrayList.Split(S);
  Writeln(List.Join);
  Writeln(List.Join('; '));
  Writeln(List.Join('"', "', '"', '"'));
end;
```

运行结果如下：

```
Monday,Tuesday,Wednesday,Thursday,Friday,Saturday,Sunday
Monday; Tuesday; Wednesday; Thursday; Friday; Saturday; Sunday
"Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"
```

上面程序的 C++ Builder 写法如下：

```
void JoinDemo() {
    AnsiString S = "Monday,Tuesday,Wednesday,Thursday,Friday,Saturday,Sunday";
    _di_IList List = ListSplit(__classid(TArrayList), S);
    cout << List->Join() << endl;
    cout << List->Join("; ") << endl;
    cout << List->Join("\", \"", "\"", "\"") << endl;
}
```

运行结果跟 Delphi 和 FreePascal 是完全一致的，这里不再重复。

如果您是在 Linux、BSD 或 Mac OS X 下使用 Lazarus/FreePascal，那么请不要忘记在程序开头添加对 cwstring 单元的引用，否则，虽然您的程序编译没有问题，但是运行时，可能会报告类似下面这样的错误：

This binary has no unicodestrings support compiled in.

Recompile the application with a unicodestrings-manager in the program uses clause.

THashedList 类

该类继承自 TArrayList。它内部增加了一个 HashBucket 来管理每个元素的 Hash 值与索引之间的关系。这使得在该类的对象上执行 Contains、IndexOf 操作的时间复杂度为 O(1)，而 TArrayList 上执行这两个操作的时间复杂度为 O(n)。

它与 TArrayList 的另一点区别是，当 List 对象中含有重复元素时，执行 IndexOf 操作，THashedList 返回的是索引值最大的一个，这个与 TArrayList 正好相反，与接下来要介绍的 TCaseInsensitiveHashedList 相同。

TCaseInsensitiveHashedList 类

该类继承自 THashedList ,因此在该类的对象上执行 Contains、IndexOf 操作的时间复杂度也为 O(1)。与 THashedList 不同的时，如果您执行 Contains、IndexOf 操作查找的是字符串元素，该类是不区分大小写的。

下面这个例子很好的说明了它与 TArrayList 和 THashedList 的区别：

```
procedure IndexOfDemo;
var
  List: IList;
  S: String;
begin
  S := 'abc,Abc,Abc,ABC';
  List := TArrayList.Split(S);
  Writeln(List.IndexOf('Abc'));
  List := THashedList.Split(S);
  Writeln(List.IndexOf('Abc'));
  List := TCaseInsensitiveHashedList.Split(S);
  Writeln(List.IndexOf('Abc'));
end;
```

它的运行结果是：

```
1
2
3
```

上面程序的 C++ Builder 写法如下：

```
void IndexOfDemo() {
    AnsiString S = "abc,Abc,Abc,ABC";
    _di_IList List = ListSplit(__classid(TArrayList), S);
    cout << List->IndexOf("Abc") << endl;
    List = ListSplit(__classid(THashedList), S);
    cout << List->IndexOf("Abc") << endl;
    List = ListSplit(__classid(TCaseInsensitiveHashedList), S);
```

```
    cout << List->IndexOf("Abc") << endl;
}
```

运行结果跟 Delphi 和 FreePascal 是完全一致的，这里不再重复。

到这里，Hprose 中提供的可变类型的 List 就全部介绍完了。最后，我们再次强烈推荐您在使用可变类型 List 写程序时，也采用上面的方式在程序中只使用 IList 接口变量，而不使用具体类的变量。

可变类型的 Map

Hprose 除了提供了方便的可变类型的 List，还提供了非常好用的可变类型的 Map。

IMap 接口

同 List 一样，Hprose 提供的所有 Map 类也都是通过接口来存取的，它们都实现了 IMap 接口。

IMap 接口中定义了枚举器操作，因此，在高版本的 Delphi（Delphi 2005 以上）或 FreePascal（FreePascal 2.5.1 以上）中您可以通过 for...in 语句来操作 IMap 接口的对象。枚举的元素类型为 TMapEntry，该类型是一个记录体，它有 2 个元素：Key 和 Value，这两个元素都是 Variant 类型的。

IMap 接口上定义了添加（Put、PutAll），获取（Get），查找（ContainsKey、ContainsValue），删除（Delete），清空（Clear），复制（Assign），同步（Lock、UnLock）等操作。

IMap 接口还定义了直接通过键值返回元素值的默认属性，可以让您向操作数组一样存取元素值，还定义了通过元素值获取键值的索引属性（Key）。另外，它还定义了元素个数（Count）、所有键值（Keys）和所有元素值（Values）这三个属性。

TAbstractMap 类

该类是所有可变类型 Map 的基类，它实现了 IMap 接口上的枚举器，复制和同步操作。它继承自 TInterfacedObject，因此它也继承了接口生存周期管理。它是一个抽象类，您不应该对它进行实例化。

如果您打算实现自己的可变类型 Map，那么您应该直接或间接的继承自它。因为在 Hprose 序列化和反序列化时，判断一个类是否是可变类型的 Map 是通过判断 TAbstractMap 是否是这个类的祖先类的方式完成的。

该类上还实现了两个比较有用的方法：Join 和 Split。但因为 TAbstractMap 是抽象类，所以您只能在它的可实例化子类上调用它们。这两个方法的具体用法，我们在下面介绍 THashMap 类的小节中来详细介绍。

THashMap 类

该类直接继承自 TAbstractMap，它是最常用的 Map。与其它语言中的 HashMap 不同，它的 Hash 存取是基于 THashedList 实现的，因此它不但可以高速存取数据，还可以保持数据插入的顺序。

创建 THashMap 对象

THashMap 的构造方法定义如下：

```
constructor Create(Capacity: Integer = 16; Factor: Single = 0.75);
```

这个构造方法需要指定一个初始化容量，如果您事先确知要放入的元素个数，那么将容量设置为与元素个数相同或多于元素个数，可以避免后面放入元素时内存的重新分配。默认初始化容量为 16。

初始化容量不代表这个 Map 只能放多少个元素，也不代表这个 Map 有多少个元素。它只代表在不重新分配内存的情况下，能放入的元素个数。当元素个数增长到超过这个容量时，这个容量会自动扩大，您不需要做任何特殊操作。

另一个参数是负载因子 Factor，其默认值为 0.75，这是时间和空间成本上一种折衷：减小负载因子可以减少 Hash 表所占用的内存空间，但会增加查询数据的时间开销；增加负载因子会提高数据查询的性能，但会增加 Hash 表所占用的内存空间。所以，通常不需要改变这个默认值。

C++ Builder 不支持带默认参数的构造方法，所以我们对 C++ Builder 作了特殊处理，在 C++ Builder 中提供有三个构造方法，它们分别对应不带默认参数和带默认参数的：

```
    __fastcall virtual THashMap(int Capacity, float Factor);
    __fastcall virtual THashMap(int Capacity);
    __fastcall virtual THashMap(void);
```

通过 Key 来存取 Value

通过 Put 方法可以在 THashMap 实例对象中加入数据，通过 Get 方法可以从 THashMap 实例对象中获取数据，但通常使用默认属性来实现这两个操作会更加方便，例如：

```
procedure AccessValueDemo;
var
  Map: IMap;
begin
  Map := THashMap.Create;
  Map['name'] := 'Lex';
  Map.Put('age', 32);
  Map['birthday'] := EncodeDate(1977, 3, 6);
  Writeln(Map.Get('name'));
  Writeln(Map['age']);
  Writeln(Map['birthday']);
end;
```

它的运行结果是：

```
Lex
32
1977-3-6
```

上面程序的 C++ Builder 写法如下：

```
void AccessValueDemo() {
    _di_IMap Map = (_di_IMap)*new THashMap;
    Map->Value["name"] = "Lex";
    Map->Put("age", 32);
```

```
    Map->Value["birthday"] = EncodeDate(1977, 3, 6);
    cout << Map->Get("name") << endl;
    cout << Map->Value["age"] << endl;
    cout << Map->Value["birthday"] << endl;
}
```

因为 C++ Builder 中不支持默认属性，所以需要显式的指明 Value 属性，它的运行结果是：

```
varString: Lex
varInteger: 32
varDate: 1977-3-6
```

批量加入数据

如果你希望把一个 List、Map 或者动态数组中的数据都加入到当前的 Map 对象中，那么可以使用 PutAll 方法，例如：

```
procedure PutAllDemo;
var
  Map1, Map2: IMap;
  List: IList;
begin
  Map1 := THashMap.Create;
  Map1['name'] := 'Lex';
  Map1['age'] := 32;
  Map1['birthday'] := EncodeDate(1977, 3, 6);
  List := TArrayList.Split('Monday,Tuesday,Wednesday,Thursday,Friday,Saturday,Sunday');
  Map2 := THashMap.Create;
  Map2.PutAll(Map1);
  Map2.PutAll(List);
  Writeln(Map2['name']);
  Writeln(Map2['age']);
  Writeln(Map2['birthday']);
  Writeln(Map2[1]);
  Writeln(Map2[2]);
  Writeln(Map2[3]);
end;
```

它的运行结果如下：

```
Lex
32
1977-3-6
Tuesday
Wednesday
```

```
Thursday
```

上面程序的 C++ Builder 写法如下：

```
void PutAllDemo() {
    _di_IMap Map1 = (_di_IMap)*new THashMap;
    Map1->Value["name"] = "Lex";
    Map1->Value["age"] = 32;
    Map1->Value["birthday"] = EncodeDate(1977, 3, 6);
    _di_IList List = ListSplit(__classid(TArrayList),
                           "Monday,Tuesday,Wednesday,Thursday,Friday,Saturday,Sunday");
    _di_IMap Map2 = (_di_IMap)*new THashMap;
    Map2->PutAll(Map1);
    Map2->PutAll(List);
    cout << Map2->Value["name"] << endl;
    cout << Map2->Value["age"] << endl;
    cout << Map2->Value["birthday"] << endl;
    cout << Map2->Value[1] << endl;
    cout << Map2->Value[2] << endl;
    cout << Map2->Value[3] << endl;
}
```

它的运行结果是：

```
varString: Lex
varInteger: 32
varDate: 1977-3-6
varString: Tuesday
varString: Wednesday
varString: Thursday
```

通过 Value 来获取 Key

通过给属性 Key 指定索引 Value 可以获取到与其匹配的 Key。当 Value 重复时 ,返回第一个匹配的 Key。例如：

```
procedure GetKeyDemo;
var
  Map: IMap;
begin
  Map := THashMap.Create;
  Map['name'] := 'Lex';
  Map['age'] := 32;
  Map['birthday'] := EncodeDate(1977, 3, 6);
  Map['alias'] := 'Lex';
```

```
  Writeln(Map.Key[EncodeDate(1977, 3, 6)]);
  Writeln(Map.Key[32]);
  Writeln(Map.Key['Lex']);
end;
```

上面的程序运行结果是：

```
birthday
age
name
```

上面程序的 C++ Builder 写法如下：

```
void GetKeyDemo() {
    _di_IMap Map = (_di_IMap)*new THashMap;
    Map->Value["name"] = "Lex";
    Map->Value["age"] = 32;
    Map->Value["birthday"] = EncodeDate(1977, 3, 6);
    Map->Value["alias"] = "Lex";
    cout << Map->Key[EncodeDate(1977, 3, 6)] << endl;
    cout << Map->Key[32] << endl;
    cout << Map->Key["Lex"] << endl;
}
```

它的运行结果是：

```
varString: birthday
varString: age
varString: name
```

查找

通过 ContainsKey 方法可以判断指定的 Key 是否存在 ,通过 ContainsValue 方法可以判断指定的 Value 是否存在。

这两个方法非常简单，这里不再举例说明。

删除

通过 Delete 方法可以删除指定的 Key 与其所对应的 Value，并且将 Value 作为返回值方法返回值。

该方法非常简单，这里不再举例说明。

清空

通过 Clear 方法可以清空所有数据，如果要删除 IMap 对象中的所有数据，使用 Clear 比使用 Delete 要快很多。

该方法非常简单，这里不再举例说明。

复制

通过 Assign 方法可以将本对象变成指定对象的一个副本。

该方法非常简单，这里不再举例说明。

同步

当您在多个线程中操作同一个 IMap 对象时，为了保证数据的线程安全性，需要对这个对象进行同步操作。通过 Lock、UnLock 方法可以非常方便的实现对 IMap 对象自身的线程同步。只需要在操作该对象前，调用该对象的 Lock 方法，所有操作完成后，再调用 Unlock 方法即可。

该方法非常简单，这里不再举例说明。

在 Hprose 1.2 for Pascal 及其之后的版本中，所有的 IMap 类都提供了一个带有 Sync 参数的构造函数，该参数默认值为 True。如果你不需要该类支持 Lock、UnLock 方法，那么可以将该参数设置为 False。在创建不带锁的 IMap 对象后，还可以通过调用 InitLock 方法来初始化锁，之后 Lock 和 UnLock 方法就可以使用了。

Hprose 1.2 for Pascal 及其之后的版本中，所有的 IMap 类还增加了多读单写锁的支持，如果需要该功能，在通过构造函数创建对象时，只需要将 ReadWriteSync 参数设置为 True 即可，该参数默认为 False。之后可以通过 BeginRead、EndRead、BeginWrite、EndWrite 这四个方法来操作读写锁。也可以在创建不带锁的对象后，单独调用 InitReadWriteLock 来初始化多读单写锁。

转换为 TArrayList 对象

如果 THashMap 对象中所有的 Key 都是非负整数，那么可以通过 ToArrayList 方法直接转换为 TArrayList 对象。该方法并没有定义在 IMap 接口中，因为该方法并不常用，所以这里不再举例说明。

转换为 IList 对象

Hprose 1.2 for Pascal 提供了一个比 ToArrayList 更通用的 ToList 方法，该方法是定义在 IMap 接口上的，只要满足上面 ToArrayList 的条件，就可以转换为任意指定的 IList 类型，并且可以设置转换之后的对象是否带有同步机制。

for...in 支持

如果您使用 Delphi 2005 或更高版本，或者 FreePascal 2.5.1 或更高版本，您可以通过 for...in 语句来枚举 IMap 接口的数据，也可以单独枚举 Keys 和 Values 这两个属性。这里不再举例说明。

分割字符串

通过 THashMap 上的类方法 Split，可以直接将一个字符串分割为 THashMap 对象。

Split 有八个参数，这八个参数的含义分别是：

Str 表示待分割的字符串。

ItemSeparator 表示各项之间的分割符，默认值是逗号";", 它可以是单个字符，也可以是字符串。

KeyValueSeparator 表示键值对之间的分割符，默认值是"=", 它可以是单个字符，也可以是字符串。

Limit 表示最多分割为几项，默认值是 0，表示没有限制。

TrimKey 表示是否去掉分割后 Key 的首尾空格，默认为 false。

TrimValue 表示是否去掉分割后 Value 的首位空格，默认为 false。

SkipEmptyKey 表示是否忽略空 Key，默认为 false。

SkipEmptyValue 表示是否忽略空 Value，默认为 false。

下面我们来看例子：

```
procedure SplitDemo;
  procedure PrintMap(Map: IMap);
  var
    I: Integer;
  begin
    for I := 0 to Map.Count - 1 do Writeln(Map.Keys[I] + ':' + Map.Values[I]);
    Writeln('--------');
  end;
var
  S: String;
begin
  S := 'Mon = Monday; Tue = Tuesday; Wed = Wednesday; Thu = Thursday; Fri = ; = Saturday';
  PrintMap(THashMap.Split(S));
  PrintMap(THashMap.Split(S, '; '));
  PrintMap(THashMap.Split(S, ';', ':'));
  PrintMap(THashMap.Split(S, ';', '=', 2));
  PrintMap(THashMap.Split(S, ';', '=', 0, true));
  PrintMap(THashMap.Split(S, ';', '=', 0, true, true));
  PrintMap(THashMap.Split(S, ';', '=', 0, true, true, true));
  PrintMap(THashMap.Split(S, ';', '=', 0, true, true, true, true));
end;
```

该程序的运行结果为：

```
Mon : Monday
 Tue : Tuesday
 Wed : Wednesday
 Thu : Thursday
 Fri :
 : Saturday
--------
Mon : Monday
Tue : Tuesday
Wed : Wednesday
Thu : Thursday
Fri :
: Saturday
--------
Mon = Monday:
 Tue = Tuesday:
```

```
 Wed = Wednesday:
 Thu = Thursday:
 Fri = :
 = Saturday:
--------
Mon : Monday
 Tue : Tuesday; Wed = Wednesday; Thu = Thursday; Fri = ; = Saturday
--------
Mon: Monday
Tue: Tuesday
Wed: Wednesday
Thu: Thursday
Fri:
: Saturday
--------
Mon:Monday
Tue:Tuesday
Wed:Wednesday
Thu:Thursday
Fri:
:Saturday
--------
Mon:Monday
Tue:Tuesday
Wed:Wednesday
Thu:Thursday
Fri:
--------
Mon:Monday
Tue:Tuesday
Wed:Wednesday
Thu:Thursday
--------
```

这八个参数的作用和区别通过上面例子的运行结果可以很好的反应出来（注意开头空格、空行、和结尾冒号的区别）。

因为 Split 是类方法，在 C++ Builder 中无法直接调用它，所以我们提供了一个帮助函数 MapSplit，它实现的是相同的功能，而且它在 C++ Builder 中也是可用的。

上面程序的 C++ Builder 写法如下：

```
void PrintMap(_di_IMap Map) {
   for (int i = 0; i < Map->Count; i++) {
       cout << VarToStr(Map->Keys->Item[i]) << ':' << VarToStr(Map->Values->Item[i]) << end
l;
   }
```

```
    cout << "--------" << endl;
}

void SplitDemo() {
    AnsiString S = "Mon = Monday; Tue = Tuesday; Wed = Wednesday; Thu = Thursday; Fri = ; = S
aturday";
    PrintMap(MapSplit(__classid(THashMap), S));
    PrintMap(MapSplit(__classid(THashMap), S, "; "));
    PrintMap(MapSplit(__classid(THashMap), S, ';', ':'));
    PrintMap(MapSplit(__classid(THashMap), S, ';', '=', 2));
    PrintMap(MapSplit(__classid(THashMap), S, ';', '=', 0, true));
    PrintMap(MapSplit(__classid(THashMap), S, ';', '=', 0, true, true));
    PrintMap(MapSplit(__classid(THashMap), S, ';', '=', 0, true, true, true));
    PrintMap(MapSplit(__classid(THashMap), S, ';', '=', 0, true, true, true, true));
}
```

运行结果跟 Delphi 和 FreePascal 是完全一致的，这里不再重复。

连接字符串

如果您的 IMap 对象中所有的 Key 和 Value 都是字符串，或者都可以转换为字符串类型，那么您可以通过 Join 方法，将这些数据以指定的方式连接成一个字符串。Join 方法有四个参数，这四个参数的含义分别是：

ItemGlue 表示各项之间的连接符，默认值是“;”

KeyValueGlue 表示 Key 和 Value 之间的连接符，默认值是“=”

LeftPad 表示在连接后的整个字符串左面需要补加的内容，默认值为空字符串。

RightPad 表示在连接后的整个字符串右面需要补加的内容，默认值为空字符串。

下面我们来看例子：

```
procedure JoinDemo;
var
  S: String;
  Map: IMap;
begin
  S := '1=Monday,2=Tuesday,3=Wednesday,4=Thursday,5=Friday,6=Saturday,7=Sunday';
  Map := THashMap.Split(S, ',');
  Writeln(Map.Join);
  Writeln(Map.Join('; '));
  Writeln(Map.Join('; ', ':'));
  Writeln(Map.Join('"', "'", ' = ', '"', '"'));
end;
```

运行结果如下：

```
1=Monday;2=Tuesday;3=Wednesday;4=Thursday;5=Friday;6=Saturday;7=Sunday
```

```
1=Monday; 2=Tuesday; 3=Wednesday; 4=Thursday; 5=Friday; 6=Saturday; 7=Sunday
1:Monday; 2:Tuesday; 3:Wednesday; 4:Thursday; 5:Friday; 6:Saturday; 7:Sunday
"1 = Monday", "2 = Tuesday", "3 = Wednesday", "4 = Thursday", "5 = Friday", "6 = Saturday",
"7 = Sunday"
```

上面程序的 C++ Builder 写法如下：

```
void JoinDemo() {
    AnsiString S = "1=Monday,2=Tuesday,3=Wednesday,4=Thursday,5=Friday,6=Saturday,7=Sunday"
;
    _di_IMap Map = MapSplit(__classid(THashMap), S, ',');
    cout << Map->Join() << endl;
    cout << Map->Join("; ") << endl;
    cout << Map->Join("; ", ':') << endl;
    cout << Map->Join("\", \"", " = ", "\"", "\"") << endl;
}
```

运行结果跟 Delphi 和 FreePascal 是完全一致的，这里不再重复。

THashedMap 类

该类继承自 THashMap。它与 THashMap 唯一的不同就是它的 Values 也是以 THashedList 方式存储的。这使得在该类的对象上执行 ContainsValue 方法、Key 属性操作的时间复杂度为 O(1)，而 THashMap 上执行这两个操作的时间复杂度为 O(n)。

它与 THashMap 的另一点区别是，当 Map 对象中含有重复值时，执行 Key 属性操作，THashedMap 返回的是最后插入的 Key，这与 THashMap 正好相反。

TCaseInsensitiveHashMap 类

该类继承自 THashMap。它与 THashMap 行为类似，唯一的不同就是在通过 Key 来查询 Value 时，该类的对象是不区分大小写的，而 THashMap 是大小写敏感的。

TCaseInsensitiveHashedMap 类

该类继承自 THashMap。它与 THashedMap 行为类似，唯一的不同就是在通过 Key 来查询 Value 时，该类的对象是不区分大小写的，而 THashedMap 是大小写敏感的。

下面这个例子很好的说明了这四种 Map 的区别：

```
procedure QueryDemo;
  procedure PrintQueryMap(Map: IMap);
  var
    I: Integer;
  begin
    for I := 0 to Map.Count - 1 do Writeln(Map.Keys[I] + ':' + Map.Values[I]);
    Writeln(Map.Count);
```

```
    Writeln(Map['Abc']);
    Writeln(Map.Key['1']);
    Writeln('--------');
  end;
var
  S: String;
begin
    S := 'abc=2;Abc=1;AB=1';
    PrintQueryMap(THashMap.Split(S));
    PrintQueryMap(THashedMap.Split(S));
    PrintQueryMap(TCaseInsensitiveHashMap.Split(S));
    PrintQueryMap(TCaseInsensitiveHashedMap.Split(S));
end;
```

运行结果如下：

```
abc:2
Abc:1
AB:1
3
1
Abc
--------
abc:2
Abc:1
AB:1
3
1
AB
--------
abc:1
AB:1
2
1
abc
--------
abc:1
AB:1
2
1
AB
--------
```

上面程序的 C++ Builder 写法如下：

```
void PrintQueryMap(_di_IMap Map) {
    for (int i = 0; i < Map->Count; i++) {
        cout << VarToStr(Map->Keys->Item[i]) << ':' << VarToStr(Map->Values->Item[i]) << end
l;
    }
    cout << Map->Count << endl;
    cout << VarToStr(Map->Value["Abc"]) << endl;
    cout << VarToStr(Map->Key["1"]) << endl;
    cout << "--------" << endl;
}

void QueryDemo() {
    AnsiString S = "abc=2;Abc=1;AB=1";
    PrintQueryMap(MapSplit(__classid(THashMap), S));
    PrintQueryMap(MapSplit(__classid(THashedMap), S));
    PrintQueryMap(MapSplit(__classid(TCaseInsensitiveHashMap), S));
    PrintQueryMap(MapSplit(__classid(TCaseInsensitiveHashedMap), S));
}
```

运行结果跟 Delphi 和 FreePascal 是完全一致的，这里不再重复。

可序列化自定义类型

Hprose for Pascal 的可序列化自定义类型非常简单，直接从 TPersistent 继承的类都是可序列化的类，不过只有被标记为 published 属性会被序列化，另外如果属性的 stored 特征被标记为 false 的话，也不会被序列化。可序列化的属性必须是前面介绍的那些可序列化基本类型、IList、IMap 和可序列化自定义类型。另外，枚举、集合类型也支持作为属性被序列化，但是结构体不能作为可序列化类型的属性类型。

除了从 TPersistent 继承以外，直接从 TObject 继承的类也可以成为可序列化自定义类型，只需要在其定义的前后加上{$M+}{$M-}标记即可，这样这个类就有了 RTTI 信息，也就可以支持序列化了。例如：

```
type

  TSex = (Unknown, Male, Female, InterSex);
  {$M+}
  TUser = class
  private
    FName: string;
    FSex: TSex;
    FBirthday: TDateTime;
    FAge: Integer;
    FMarried: Boolean;
  published
    property Name: string read FName write FName;
    property Sex: TSex read FSex write FSex;
```

```
    property Birthday: TDateTime read FBirthday write FBirthday;
    property Age: Integer read FAge write FAge;
    property Married: Boolean read FMarried write FMarried;
  end;
  {$M-}
```

这个 TUser 类就是可序列化自定义类型。

不过还有一点需要注意，你需要在定义该类的单元的 initialization 段中加入

```
  RegisterClass(TUser, 'User');
```

这条语句才可以让它真正成为可以与其它语言交互可序列化自定义类型。其中'User'是 TUser 类的别名，这个名字是需要跟其它语言中定义的类名一致的。

另外，如果是跟 Java、.NET 这些语言交互，还要注意一点，Java 和.NET 的类常常会放在名称空间中，例如 my.namespace.User，如果要跟这个类交互的话，可以这样定义别名：

```
  RegisterClass(TUser, 'my_namespace_User');
```

就是把“.”换成“_”就可以啦。还需要注意，类的别名是区分大小写的。但是属性名不区分大小写，尽管如此，属性名仍然建议按照 Delphi 方式来定义。

在 C++ Builder 中不方便直接定义 Delphi 的类，但是可以直接使用 Delphi 中的自定义可序列化类型的单元。

在 Free Pascal 中自定义可序列化类型的声明和使用方式跟 Delphi 完全相同。

其它帮助方法

VarEquals 函数

判断两个 Variant 对象是否相等。

VarRef 函数

返回一个 Variant 数据的 Variant 引用。

VarUnref 函数

返回一个 Variant 引用所指向的 Variant 数据。

VarIsList 函数

通过 VarIsList 函数可以判断一个 Variant 包装的数据是否是一个 IList 对象。

VarIsMap 函数

通过 VarIsMap 函数可以判断一个 Variant 包装的数据是否是一个 IMap 对象。

VarIsObj 函数

通过 VarIsObj 函数可以判断一个 Variant 包装的数据是否是一个对象，通过两个参数的重载，还可以判断是否是指定类的对象。

VarToList 函数

将 Variant 包装的数据转换为一个 IList 对象。只有当用 VarIsList 函数判断为 true 时，该函数才会返回结果，否则抛出异常。

VarToMap 函数

将 Variant 包装的数据转换为一个 IMap 对象。只有当用 VarIsMap 函数判断为 true 时，该函数才会返回结果，否则抛出异常。

VarToObj 函数

将 Variant 包装的数据转换为一个对象，只有当用 VarIsObj 函数判断为 true 时，该函数才会返回结果，否则抛出异常。该函数的三个参数重载的版本不抛出异常，而是当转换失败时返回 false。

ObjToVar 函数

将一个对象包装成 Variant 类型的数据。这样就可以将它作为元素放入 IList 中，也可以作为 Key 或 Value 放入 IMap 中，还可以通过 Hprose 在远程调用中进行传递了。

注意：没有 ListToVar 和 MapToVar 函数，因为 IList 和 IMap 对象本身就可以直接赋值给 Variant 变量。

第三章 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 Pascal 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（$-2^{31} \sim 2^{31}-1$）。
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。

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

Pascal 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 Pascal 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

Pascal 类型	Hprose 类型
Integer，Shortint，Smallint，Longint，Byte，Word，非 Boolean 类型的枚举属性，集合属性，Variant（varByte，varWord，varShortInt，varSmallint，varInteger）	整型
Cardinal，Longword，Int64，UInt64，QWord，纯数字字符串，Variant（varLongWord，varInt64，varUInt64，varQWord，纯数字的 varString，varUString，varOleStr）	长整型
Real，Real48，Single，Double，Extended，Comp，Currency，Variant（varSingle，varDouble，varCurrency）	浮点型
NaN	非数
Infinity	正无穷大
NegInfinity	负无穷大
True，Variant（varBoolean 的 True 值）	布尔真
False，Variant（varBoolean 的 False 值）	布尔假
nil，Variant（varNull，varEmpty）	空
array of Byte，Variant（一维 varByte 数组）	二进制型（或空串）
Char ,WideChar ,PChar ,PWideChar ,ShortString ,AnsiString ,WideString，UnicodeString，Variant（varString，varUString，varOleString）	字符串型（或空串、字符）
只包含日期部分的 TDateTime，Variant（varDate）	日期型
只包含时间部分的 TDateTime，Variant（varDate）	时间型
包含日期和时间部分的 TDateTime，Variant（varDate）	日期时间型

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	Pascal 类型
整型	Variant (varInteger)
长整型	Variant (varString)
浮点型	Variant (varDouble)
非数	Variant (varDouble 的 NaN 值)
正无穷大	Variant (varDouble 的 Infinity 值)
负无穷大	Variant (varDouble 的 NegInfinity 值)
布尔真	Variant (varBoolean 的 True 值)
布尔假	Variant (varBoolean 的 False 值)
空	Null
空串	''
二进制型	Variant (一维字节数组)
字符/字符串型	Variant (varOleStr)
日期型	Variant (varDate)
时间型	Variant (varDate)
日期时间型	Variant (varDate)

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	Pascal 类型 (安全)	Pascal 类型 (非安全)
整型	Variant (varInteger，varSingle，varDouble，varCurrency，varInt64，varString，varUString，varOleStr)	Variant (varByte，varShortint，varWord，varSmallint，varLongWord，varUInt64，varQWord，varBoolean，varDate)

Hprose 类型	Pascal 类型（安全）	Pascal 类型（非安全）
长整型	Variant（varString，varUString，varOleStr）	Variant（varInteger，varByte，varShortint，varWord，varSmallint，varLongWord，varSingle，varDouble，varCurrency，varInt64，varUInt64，varQWord，varBoolean，varDate）
浮点型	Variant（varDouble，varString，vaarUString，varOleStr）	Variant（varSingle，varCurrency，varInteger ,varByte ,varShortint ,varWord，varSmallint，varLongWord，varInt64，varUInt64，varQWord，varBoolean，varDate）
非数	Variant（varDouble 和 varSingle 的 NaN 值，varString，varUString，varOleStr）	无
正无穷大	Variant（varDouble 和 varSingle 的 Infinity 值，varString，varUString，varOleStr）	无
负无穷大	Variant（varDouble 和 varSingle 的 NegInfinity 值，varString，varUString，varOleStr）	无
布尔真	Variant（varBoolean，varSingle，varDouble，varCurrency，varInteger，varByte，varShortint，varWord，varSmallint，varLongWord，varInt64，varUInt64，varQWord，varString，varUString，varOleStr）	无
布尔假	Variant（varBoolean，varSingle，varDouble，varCurrency，varInteger，varByte，varShortint，varWord，varSmallint，varLongWord，varInt64，varUInt64，varQWord，varString，varUString，varOleStr）	无
空	任意类型	无
空串	任意类型	无

Hprose 类型	Pascal 类型（安全）	Pascal 类型（非安全）
二进制型	Variant（一维字节数组）	无
字符/字符串型	Variant（varOleStr）	无
日期型	Variant（varDate）	无
时间型	Variant（varDate）	无
日期时间型	Variant（varDate）	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 Pascal 类型的映射关系。

列表类型

序列化类型映射

除一维字节数组以外的所有其它 Variant 数组类型，动态数组类型和所有实现了 IList 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 Pascal 的 TArrayList 类的 IList 接口。有效类型为：

- Variant 数组类型
- 所有实现了 IList 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 IMap 接口的类型均映射为 Hprose 字典类型。

反序列化类型映射

Hprose 字典类型默认映射为 Pascal 的 THashMap 类的 IMap 接口。有效类型为：

- 所有实现了 IMap 接口的可实例化类型
- 所有拥有与字典中 Key 所对应的属性相同的自定义可序列化可实例化类型

对象类型

Pascal 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：

- Pascal 中自定义的可序列化对象类型
- TCaseInsensitiveHashMap 的 IMap 接口（当上述类型定义不存在时）

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 Pascal 客户端，在本章中您将深入的了解 Hprose for Pascal 客户端的更多细节。

本章提要

- 同步调用
- 异步调用
- 异常处理
- 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。

直接使用 THproseIdHttpClient（或 THproseSynaHttpClient）上的 Invoke 方法就可以进行同步调用。Invoke 方法有多个重载，其中同步调用有两个重载方法，一个是支持引用参数传递的，另一个是不支持引用参数传递的。通常我们最常使用的是不支持引用参数传递的重载，因为它使用更方便，效率也更高。

在下面的例子中，我们以调用官方网站（地址为：http://www.hprose.com/example/）提供的服务为例来进行说明讲解。

可变的参数和结果类型

下面我们来看对 sum 方法的调用：

```
procedure SumDemo;
var
  Client: THproseIdHttpClient;
begin
  Client := THproseIdHttpClient.Create(nil);
  Client.UseService('http://www.hprose.com/example/');
  Writeln(Client.Invoke('sum', [1, 2, 3, 4, 5]));
  Writeln(Client.Invoke('sum', [1.2, 3.4, 4.5]));
  Writeln(Client.Invoke('sum', ['123', '45']));
  Client.Free;
end;
```

下面看运行结果：

```
15
9.1
168
```

这个在服务器端的 sum 函数是使用 php 语言定义并发布的，所以是一个弱类型参数的函数，并且参数个数也是任意多个。但是我们的客户端根本不需要知道也不需要在乎服务器端是什么语言，只要按照正确的方式调用就可以了。

这里我们传递了整数参数，浮点数参数，数字字符串参数，然后服务器端都进行了正确的类型转换和运算，并且返回了结果。

客户端得到的结果类型也是随服务器实际返回的结果类型而确定的。

引用参数传递

下面我们来继续看对 swapKeyAndValue 方法的调用：

```
procedure SwapKeyAndValueDemo;
var
  Client: THproseIdHttpClient;
  Map1, Map2: IMap;
  Args: TVariants;
begin
  Client := THproseIdHttpClient.Create(nil);
  Client.UseService('http://www.hprose.com/example/');
  Map1 := THashMap.Split('Mon=1;Tue=2;Wed=3;Thu=4;Fri=5;Sat=6;Sun=7');
  SetLength(Args, 1);
  Args[0] := Map1;
  Client.Invoke('swapKeyAndValue', Args);
  Map2 := VarToMap(Args[0]);
  Writeln(Map1.Join);
  Writeln(Map2.Join);
  Client.Free;
end;
```

运行结果：

```
Mon=1;Tue=2;Wed=3;Thu=4;Fri=5;Sat=6;Sun=7
1=Mon;2=Tue;3=Wed;4=Thu;5=Fri;6=Sat;7=Sun
```

上面的调用演示了引用参数传递。在对 swapKeyAndValue 进行调用时，我们使用了 Invoke 的引用参数传递的重载版本，所以调用后，参数 Args[0]的值发生了变化，但是需要注意的是，原始的 Map1 并没有改变，改变的是参数数组 Args 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：

```
procedure GetUserListDemo;
var
  Client: THproseIdHttpClient;
  UserList: IList;
  User: TUser;
  I: Integer;
begin
  Client := THproseIdHttpClient.Create(nil);
  Client.UseService('http://www.hprose.com/example/');
  UserList := VarToList(Client.Invoke('getUserList', []));
```

```
  for I := 0 to UserList.Count - 1 do begin
    VarToObj(UserList[I], TUser, User);
    Writeln('Name: ' + User.Name + ', ' +
           'Sex: ' + GetEnumName(TypeInfo(TSex),Ord(USer.Sex)) + ', ' +
           'Birthday: ' + DateToStr(User.Birthday) + ', ' +
           'Age: ' + IntToStr(User.Age) + ', ' +
           'Married: ' + BoolToStr(User.Married, true));
    User.Free;
  end;
  Client.Free;
end;
```

上面程序中的 TUser 的定义请参见第三章可序列化自定义类型一节。运行结果：

```
Name: Amy, Sex: Female, Birthday: 1983-12-3, Age: 26, Married: True
Name: Bob, Sex: Male, Birthday: 1989-6-12, Age: 20, Married: False
Name: Chris, Sex: Unknown, Birthday: 1980-3-8, Age: 29, Married: True
Name: Alex, Sex: InterSex, Birthday: 1992-6-14, Age: 27, Married: False
```

上例中，返回结果是一个包含 TUser 对象的 IList 的 Variant 包装，因此我们先用 VarToList 将 Variant 对象解包装，然后遍历 IList 各个元素时，因为每个元素也都是 Variant 包装的 TUser 对象，因此我们用 VarToObj 来解包装，最后我们就可以打印出 User 对象的各个属性的数据啦，因为 User 对象的生存周期不是自动管理的，因此在使用之后需要手工调用 Free 方法进行释放，而 UserList 对象因为是一个 IList 接口，所以它的生存周期是自动管理的，不需要手动释放。

同步调用的 Invoke 方法最常用方式就是指使用前两个参数，后面还有 2 个参数用于指定返回结果类型，但一般不需要指定。

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。

简单的说，如果我们是在 Delphi 或 Lazarus 的 GUI 程序中使用 Hprose 客户端，我们就应该使用异步调用。

虽然您也可以使用多线程加同步调用来完成异步调用，但您不必这样做。因为直接使用 Hprose 提供的异步调用方式，将更加简单。

简单参数传递

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。Pascal 有过程类型，回调方法的过程类型定义如下：

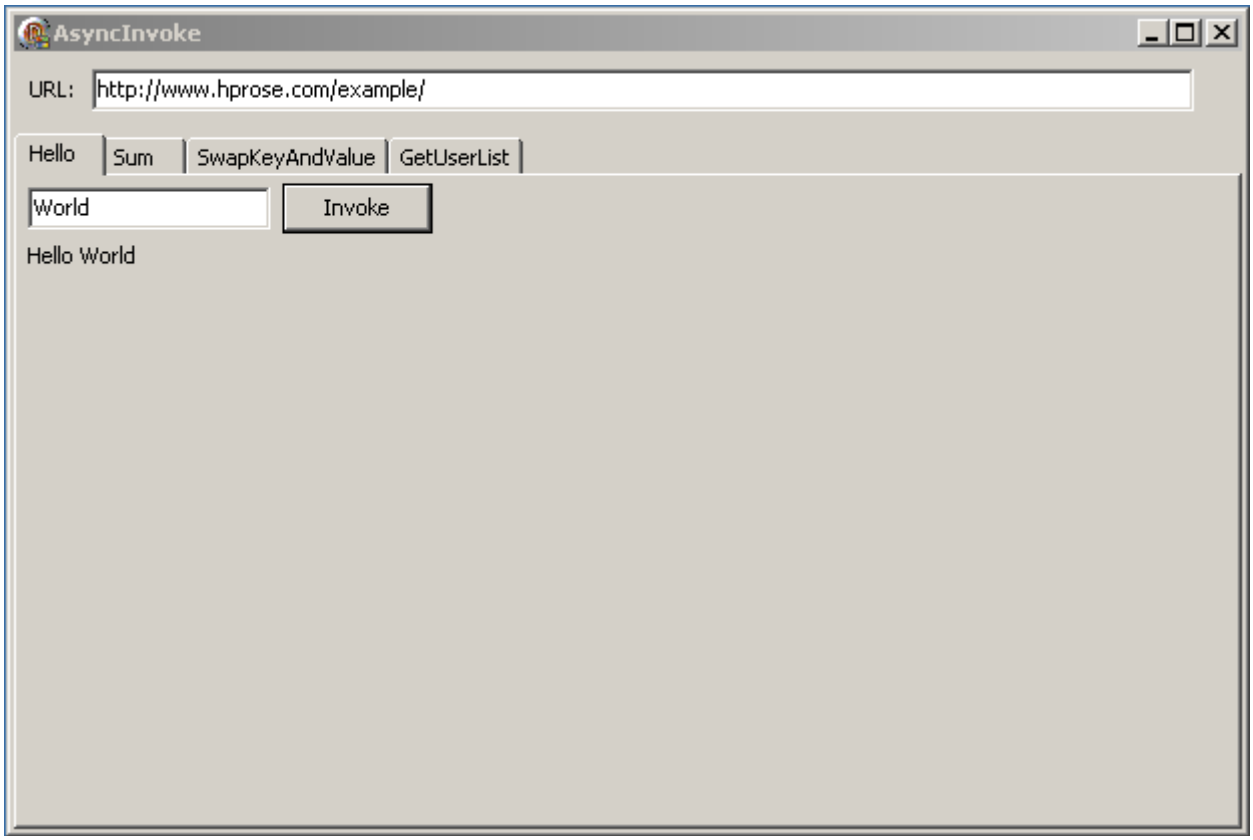
```
THproseCallback1 = procedure(Result: Variant) of object;
THproseCallback2 = procedure(Result: Variant; const Args: TVariants) of object;
```

其中，THproseCallback1 对应非引用参数传递的异步调用，THproseCallback2 对应引用参数传递的异步调用。

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。下面这个简单的例子很好的说明了如何进行异步调用：

```
procedure THproseTestForm.HelloCallback(Result: Variant);
begin
  HelloResult.Caption := Result;
end;
procedure THproseTestForm.HelloInvokeBtnClick(Sender: TObject);
begin
  HproseClient.Invoke('Hello', [HelloEdit.Text], HelloCallback, varString);
end;
```

上面这个整个程序的核心部分代码，其中 HelloResult 是一个 TLabel 控件，HelloEdit 是一个 TEdit 控件。这个程序的运行结果为：



复杂参数传递

同步调用可以到的事情，异步调用都可以做到。比如引用参数传递，容器类型和自定义类型传输，以及在调用中指定要返回的结果类型。下面我们来看一下上图中完整的程序代码：

```
unit MainForm;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, Tabs, ComCtrls, StdCtrls, HproseClient, HproseCommon, HproseIdHttpClient,
  Grids, ValEdit, TypInfo;

type

  TSex = (Unknown, Male, Female, InterSex);
  {$M+}
  TUser = class
  private
    FName: string;
    FSex: TSex;
    FBirthday: TDateTime;
    FAge: Integer;
    FMarried: Boolean;
  published
    property Name: string read FName write FName;
    property Sex: TSex read FSex write FSex;
    property Birthday: TDateTime read FBirthday write FBirthday;
    property Age: Integer read FAge write FAge;
    property Married: Boolean read FMarried write FMarried;
  end;
  {$M-}

  TAsyncInvokeForm = class(TForm)
    PageControl1: TPageControl;
    TabSheet1: TTabSheet;
    TabSheet2: TTabSheet;
    TabSheet3: TTabSheet;
    GetUserList: TTabSheet;
    URLEdit: TEdit;
    UrlLabel: TLabel;
    HelloEdit: TEdit;
    HelloInvokeBtn: TButton;
    SumArgs: TListBox;
```

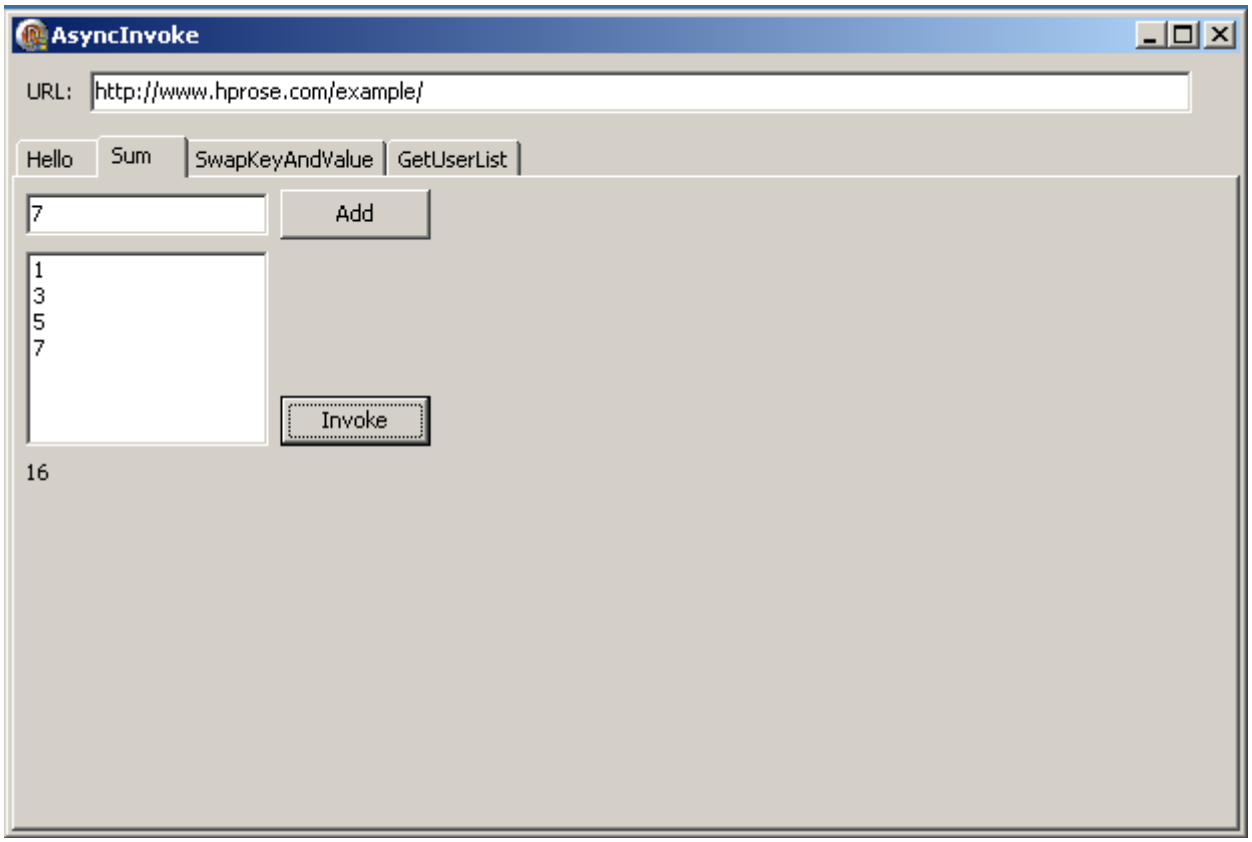
```
    SumEdit: TEdit;
    AddBtn: TButton;
    SumInvokeBtn: TButton;
    HelloResult: TLabel;
    SumResult: TLabel;
    HproseClient: THproseIdHttpClient;
    ValueListEditor: TValueListEditor;
    SwapInvokeBtn: TButton;
    UserInvokeBtn: TButton;
    UsersMemo: TMemo;
    procedure HproseClientError(const Name: string; const Error: Exception);
    procedure HelloInvokeBtnClick(Sender: TObject);
    procedure AddBtnClick(Sender: TObject);
    procedure SumInvokeBtnClick(Sender: TObject);
    procedure ValueListEditorDblClick(Sender: TObject);
    procedure SwapInvokeBtnClick(Sender: TObject);
    procedure UserInvokeBtnClick(Sender: TObject);
  private
    function SetURL(): Boolean;
    procedure HelloCallback(Result: Variant);
    procedure SumCallback(Result: Variant; const Args: TVariants);
    procedure SwapCallback(Result: Variant);
    procedure UserCallback(Result: Variant);
  public
    { Public declarations }
  end;

var
  AsyncInvokeForm: TAsyncInvokeForm;

implementation

{$R *.dfm}

function TAsyncInvokeForm.SetURL(): Boolean;
begin
  Result := URLEdit.Text <> '';
  if URLEdit.Text <> '' then
    HproseClient.UseService(URLEdit.Text)
  else
    ShowMessage('fill the hprose server url first.');
end;

procedure TAsyncInvokeForm.SumInvokeBtnClick(Sender: TObject);
var
```

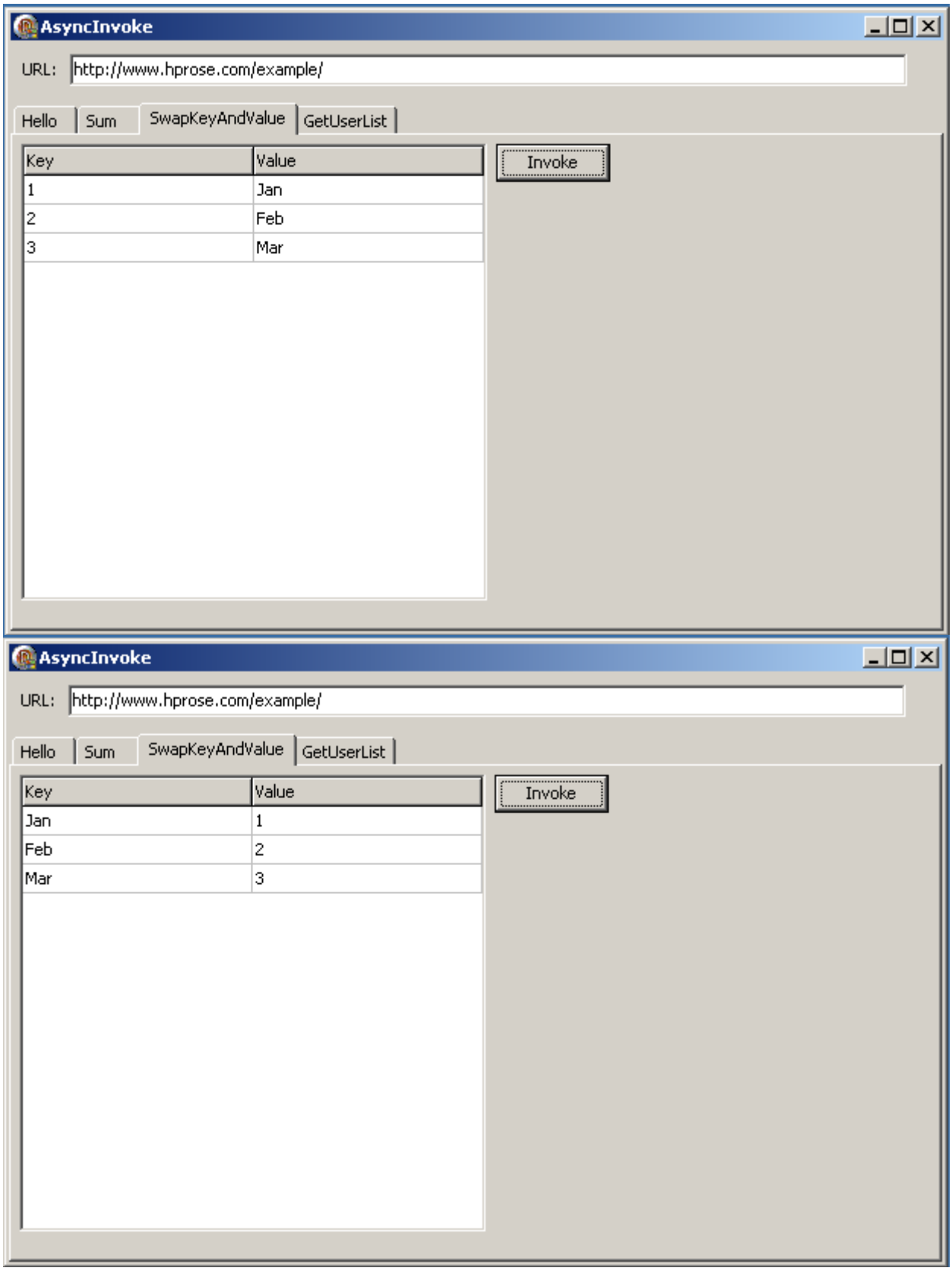
```
  Args: TVariants;
  I: Integer;
begin
  if SetURL() then begin
    SetLength(Args, SumArgs.Count);
    for I := 0 to SumArgs.Count - 1 do
      Args[I] := StrToFloat(SumArgs.Items.Strings[I]);
    HproseClient.Invoke('Sum', Args, SumCallback);
  end;
end;

procedure TAsyncInvokeForm.SwapInvokeBtnClick(Sender: TObject);
var
  Map: IMap;
  I: Integer;
begin
  if SetURL() then begin
    Map := THashMap.Create as IMap;
    for I := 1 to ValueListEditor.RowCount - 1 do
      Map[ValueListEditor.Cells[0, I]] := ValueListEditor.Cells[1, I];
    HproseClient.Invoke('swapKeyAndValue', [Map], SwapCallback);
  end;
end;

procedure TAsyncInvokeForm.UserInvokeBtnClick(Sender: TObject);
begin
  if SetURL() then
    HproseClient.Invoke('getUserList', [], UserCallback);
end;

procedure TAsyncInvokeForm.ValueListEditorDblClick(Sender: TObject);
begin
  ValueListEditor.InsertRow('', '', true);
end;

procedure TAsyncInvokeForm.HproseClientError(const Name: string;
  const Error: Exception);
begin
  ShowMessage(Name + ':' + Error.Message);
end;

procedure TAsyncInvokeForm.HelloInvokeBtnClick(Sender: TObject);
begin
  if SetURL() then
    HproseClient.Invoke('Hello', [HelloEdit.Text], HelloCallback, varString);
```

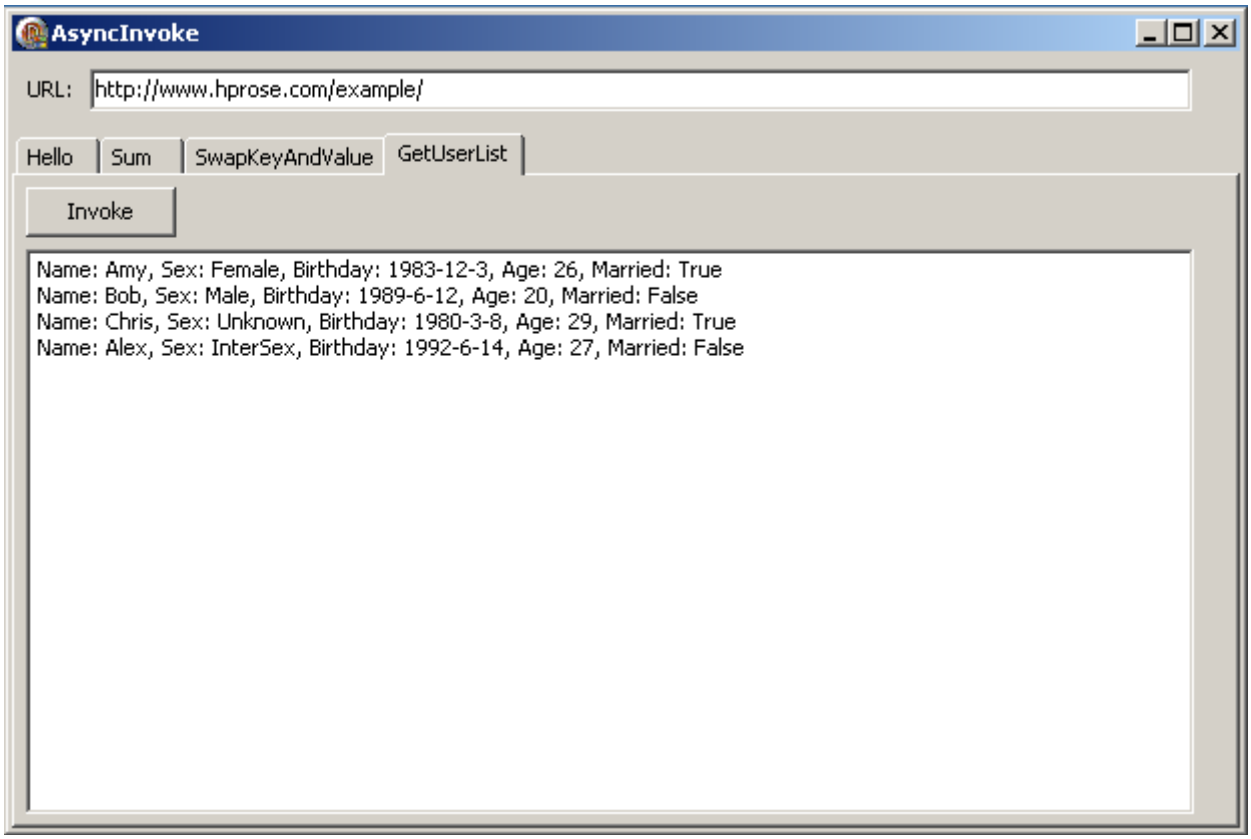
```
end;

procedure TAsyncInvokeForm.AddBtnClick(Sender: TObject);
begin
  try
    StrToFloat(SumEdit.Text);
    SumArgs.Items.Append(SumEdit.Text);
  except
  end;
end;

procedure TAsyncInvokeForm.HelloCallback(Result: Variant);
begin
  HelloResult.Caption := Result;
end;

procedure TAsyncInvokeForm.SumCallback(Result: Variant; const Args: TVariants);
var
  I: Integer;
begin
  SumResult.Caption := Result;
  for I := 0 to Length(Args) - 1 do ShowMessage(Args[I]);

end;

procedure TAsyncInvokeForm.SwapCallback(Result: Variant);
var
  Map: IMap;
  I: Integer;
begin
  Map := VarToMap(Result);
  for I := 0 to Map.Count - 1 do begin
    ValueListEditor.Cells[0, I + 1] := '';
    ValueListEditor.Cells[1, I + 1] := '';
  end;
  for I := 0 to Map.Count - 1 do begin
    ValueListEditor.Cells[0, I + 1] := Map.Keys[I];
    ValueListEditor.Cells[1, I + 1] := Map.Values[I];
  end;
end;

procedure TAsyncInvokeForm.UserCallback(Result: Variant);
var
  List: IList;
  I: Integer;
```

```
  User: TUser;
begin
  List := VarToList(Result);
  for I := 0 to List.Count - 1 do begin
    VarToObj(List[I], TUser, User);
    UsersMemo.Lines.Append('Name: ' + User.Name + ', ' +
                           'Sex: ' + GetEnumName(TypeInfo(TSex),Ord(USer.Sex)) + ', ' +
                           'Birthday: ' + DateToStr(User.Birthday) + ', ' +
                           'Age: ' + IntToStr(User.Age) + ', ' +
                           'Married: ' + BoolToStr(User.Married, true));
    User.Free;
  end;
end;

initialization
  RegisterClass(TUser, 'User');
end.
```

这个程序虽然较长，但是很容易理解。下面是运行的一些截图：




AsyncInvoke
URL: http://www.hprose.com/example/
Hello
Sum
SwapKeyAndValue
GetUserList
Key
Value
1
Jan
2
Feb
3
Mar
Invoke
AsyncInvoke
URL: http://www.hprose.com/example/
Hello
Sum
SwapKeyAndValue
GetUserList
Key
Value
Jan
1
Feb
2
Mar
3
Invoke



关于异步调用就这么多内容，您只要理清了思路，掌握并灵活运用它并不难。下面我们再来看一下如何在客户端处理远程调用中发生的异常吧。

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...except 语句块即可捕获异常，通常服务器端调用返回的异常是 EHproseException 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 except 捕获 Exception 类型来处理全部可能发生的异常。

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。

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 THproseErrorEvent，它是一个方法类型，使用非常简单，例如：

```
procedure TAsyncInvokeForm.HproseClientError(const Name: string; const Error: Exception);
begin
  ShowMessage(Name + ':' + Error.Message);
end;
```

然后将其设置到 Hprose 客户端的 OnError 方法上即可。Hprose 1.3 还可以将这个方法直接作为参数

传入 Invoke 调用，这样可以对每个调用指定不同的错误处理。

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。

超时设置

Hprose 1.2 for Pascal 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。

注意：Indy8 版本不支持该属性。

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 ProxyHost、ProxyPort 来设置 http 代理服务器的地址和端口。ProxyHost 可以是 IP 或者域名，用字符串来表示，默认值为空字符串，表示禁用代理服务器。ProxyPort 为端口号，用整数表示，默认是 8080。

如果您所使用的代理服务器需要身份验证，可以通过 ProxyUser 和 ProxyPass 来设置用于代理服务器上验证的用户名和密码。这两个属性的默认值也为空字符串，表示不需要身份验证。

HTTP 标头

有时候您可能需要设置特殊的 http 标头，只需要通过设置 Headers 属性即可。

标头名不可以为以下值：

- Context-Type
- Context-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 Headers 熟悉来设置 Cookie 标头，将会影响 Session 的正常工作。

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（注意：只有 Syna 客户端支持）。

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。

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 Delphi 中 ,序列化的结果以 TMemoryStream 的 Varient 包装类型返回。用户可以用 VarToObj 将其转化会 TMemoryStream 类型后，通过 HproseReader 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。