



用户手册

1.3

(Python 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 Python	6
安装方法	6
创建 Hprose 的 Hello 服务器	6
创建 Hprose 的 Hello 客户端	7
第二章 类型映射	9
本章提要	9
基本类型	10
值类型	10
引用类型	10
基本类型的映射	11
序列化类型映射	11
反序列化类型映射	12
容器类型	12
列表类型	12
字典类型	13
对象类型	13
通过 HproseClassManager 来注册自定义类型	14
第三章 服务器	15
本章提要	15
常用服务器介绍	16
mod_wsgi 方式	16
uWSGI 方式	17

在 Ubuntu 上的安装与发布	17
在 Mac OS X 上的安装与发布	18
eurasia 方式	19
发布服务	19
发布函数	20
发布方法	20
别名机制	21
发布对象	22
发布类	23
迷失的方法	23
服务器开关	24
隐藏发布列表	24
调试开关	24
P3P 开关	24
跨域开关	24
服务器事件	25
onBeforeInvoke 事件	25
onAfterInvoke 事件	25
onSendHeader 事件	25
onSendError 事件	25
会话管理	25
WSGI 中间件	26
第四章 客户端	27
本章提要	27
同步调用	28
直接通过远程方法名进行远程调用	28
通过代理对象进行远程调用	28
通过 Invoke 方法进行远程调用	29
非引用参数传递	29
引用参数传递	30
异步调用	30
非引用参数传递	30
引用参数传递	31
异常处理	32
同步调用中的异常处理	32
异步调用中的异常处理	32
超时设置	33
HTTP 参数设置	33

代理服务器	33
HTTP 标头	33
持久连接	34
调用结果返回模式	34
Serialized 模式	34
Raw 模式	34
RawWithEndTag 模式	34

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 Python 进行分布式程序开发的全过程。

本章提要

- 安装 Hprose for Python
- 创建 Hprose 的 Hello 服务器
- 创建 Hprose 的 Hello 客户端

安装 Hprose for Python

Hprose for Python 支持 Python 2.3、2.4、2.5、2.6、2.7、3.0、3.1。

服务器端可以不依赖于 Web 服务器独立运行，也可以配合任何支持 WSGI 的 Web 服务器运行。

安装方法

如果您安装的是 Python 2.3~2.7 版本，进入 py23 目录，执行：

```
python setup.py install
```

即可，如果你同时安装有多个版本的 Python，则需要指明具体版本的路径，例如：

```
/usr/bin/python2.5 setup.py install  
/usr/bin/python2.6 setup.py install
```

你也可以直接安装相应版本的 egg 包，例如：

```
ez_setup hprose-1.0-py2.5.egg
```

如果您安装的是 Python 3.0~3.1 版本，进入 py3k 目录，执行同样操作即可。

创建 Hprose 的 Hello 服务器

创建 Python 的 Hprose 的服务器非常简单。下面我们以 Hprose 自带的独立服务器为例来进行讲解：

```
#!/usr/local/bin/python2.6  
# encoding: utf-8  
  
from hprose.httpserver import HproseHttpServer  
  
def hello(name):  
    return 'Hello' + ' ' + name;  
  
server = HproseHttpServer(port=3030)  
server.addFunction(hello)  
server.start()
```

上面这段就是独立服务器端的代码，是不是相当的简单啊？

第一句

```
#!/usr/local/bin/python2.6
```

不是必须的，但在 linux/unix 的主机上，比较有用。

第二句

```
# encoding: utf-8
```

算是必须的，否则你的程序无法正确传递中文。

下面的代码很简单，就不需要我多做解释了。我们来看看效果吧，首先，把上面代码保存为 `helloserver.py`，然后在命令行中输入：

```
python helloserver.py
```

如果看到如下的输出字样：

```
Serving on port :3030...
```

说明服务已经启动了。现在你可以在同一台机器上打开浏览器，输入：`http://127.0.0.1:3030/`，然后回车，看到浏览器中有如下输出：

```
Fa1{s5"hello"}z
```

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

Python 的 Hprose 客户端创建更加容易，下面我们来创建一个 `helloclient.py`，内容如下：

```
#!/usr/local/bin/python2.6
# encoding: utf-8
from hprose.httpclient import HproseHttpClient
client = HproseHttpClient('http://127.0.0.1:3030/')
print client.Hello('World!')
```

然后在命令行中输入：

```
python helloclient.py
```

如果看到下面的结果：

```
Hello World!
```

就说明客户端创建成功了！

请注意，上面运行客户端之前必须要先把服务器运行起来，否则你可能会看到类似如下的错误输出：

```
Traceback (most recent call last):
  File "helloclient.py", line 5, in <module>
    print client.Hello('World!')
  File "/Library/Frameworks/Python.framework/Versions/2.6/lib/python2.6/site-packages/hprose-1.0-py2.6.egg/hprose/client.py", line 42, in __call__
```

```
return self.__invoke(self.__name, args, callback, byRef)
File "/Library/Frameworks/Python.framework/Versions/2.6/lib/python2.6/site-packages/hprose-1.0-py2.6.egg/hprose/client.py", line 79, in invoke
    return self.__invoke(name, args, byRef)
File "/Library/Frameworks/Python.framework/Versions/2.6/lib/python2.6/site-packages/hprose-1.0-py2.6.egg/hprose/client.py", line 128, in __invoke
    self._sendData(context)
File "/Library/Frameworks/Python.framework/Versions/2.6/lib/python2.6/site-packages/hprose-1.0-py2.6.egg/hprose/httpclient.py", line 320, in _sendData
    data = self.__post(request)
File "/Library/Frameworks/Python.framework/Versions/2.6/lib/python2.6/site-packages/hprose-1.0-py2.6.egg/hprose/httpclient.py", line 357, in __post
    httpclient.request('POST', path, data, header)
File "/Library/Frameworks/Python.framework/Versions/2.6/lib/python2.6/httplib.py", line 910, in request
    self._send_request(method, url, body, headers)
File "/Library/Frameworks/Python.framework/Versions/2.6/lib/python2.6/httplib.py", line 947, in _send_request
    self.endheaders()
File "/Library/Frameworks/Python.framework/Versions/2.6/lib/python2.6/httplib.py", line 904, in endheaders
    self._send_output()
File "/Library/Frameworks/Python.framework/Versions/2.6/lib/python2.6/httplib.py", line 776, in _send_output
    self.send(msg)
File "/Library/Frameworks/Python.framework/Versions/2.6/lib/python2.6/httplib.py", line 735, in send
    self.connect()
File "/Library/Frameworks/Python.framework/Versions/2.6/lib/python2.6/httplib.py", line 716, in connect
    self.timeout)
File "/Library/Frameworks/Python.framework/Versions/2.6/lib/python2.6/socket.py", line 514, in create_connection
    raise error, msg
socket.error: [Errno 61] Connection refused
```

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

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 Python 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

Python 类型与 Hprose 类型的映射关系不是一一对应的。在序列化和反序列化过程中可能会有一种 Python 类型对应多种 Hprose 类型的情况出现（当然条件会有不同）。我们下面以列表的形式来说明。

序列化类型映射

Python 类型	Hprose 类型
int , py3k 中是 int 的 -2147483648 ~ 2147483647 范围之内的数字（含边界）	整型
long , py3k 中是 int 的 -2147483648 ~ 2147483647 范围之外的数字	长整型
float	浮点型
fpconst.NaN	非数
fpconst.PosInf	正无穷大
fpconst.NegInf	负无穷大
True	布尔真
False	布尔假
None	空
str , py3k 中是 bytes 和 bytearray	二进制型（或空串）
单字元的 unicode	字符
str、unicode	字符串型（或空串）
date	日期型
time	时间型
datetime	日期时间型

这里二进制型和字符串型需要做一下说明，在 python 2.3~2.6 中（Hprose for py23 版本），Hprose 提供了一个 Unicode 开关，它在 hprose.io 包中，默认值为 False，你可以这样修改它的默认值：

```
import hprose.io
hprose.io.Unicode = True
```

默认状态下，str 为 utf-8 编码的字符串时，str 作为字符串型序列化，否则作为二进制型序列化。

当 hprose.io.Unicode 为 True 时，只有 unicode 类型作为字符串型序列化，str 的值不管是否是 utf-8 编码，都全部按照二进制型序列化。

默认情况下，程序兼容性会更好。当修改 hprose.io.Unicode 为 True 后，程序性能会更好，但程序中不能使用 str 来表示字符串，否则就会有兼容性问题。

因为 Python 3.x 严格区分字符串和字节数组，所以在 Hprose for py3k 中，不存在这个开关。

反序列化类型映射

Hprose 类型	Python 类型
整型	int
长整型	long、py3k 是 int
浮点型	float
非数	fpconst.NaN
正无穷大	fpconst.PosInf
负无穷大	fpconst.NegInf
布尔真	True
布尔假	False
空	None
空串	""或 u""
二进制型	str，py3k 是 bytes
字符/字符串型	str 或 unicode，py3k 是 str
日期型	date
时间型	time
日期时间型	datetime

反序列化时，python 2.3~2.6 也会根据 hprose.io.Unicode 开关的取值来做出不同的动作。

默认情况下，空串和字符串型都是按照 utf-8 编码的 str 类型来反序列化的。当 hprose.io.Unicode 为 True 时，空串和字符串型都按照 unicode 类型来反序列化。

在反序列化时，不管是性能还是兼容性来说，默认情况下都比选择开启 hprose.io.Unicode 开关的情况下要好。

所以，通常情况下，强烈推荐不要改变 hprose.io.Unicode 的默认值，除非您的程序中的字符串全部是按照 unicode 来处理的。

容器类型

Hprose 中的容器类型包括列表类型和字典类型两种。它们分别对应于 Python 的 list (tuple) 和 dict 类型。

列表类型

Python 中的 list 和 tuple 类型，都被映射为 Hprose 列表类型。例如：

```
a = [1, 2, 3, 4, 5]
```

```
fruit = ('apple', 'banana', 'cherry')
```

都被序列化为 Hprose 的列表类型，但是反序列化时，都被反序列化为 list。

字典类型

Python 中的 dict 类型被映射为 Hprose 字典类型，例如：

```
a = { 1: 'one', 2: 'two', 3: 'three' }
info = { 'version': 4,
          'OS': 'Linux',
          'lang': 'English',
          'short_tags': True }
```

都被映射为 Hprose 字典类型。

对象类型

Python 中自定义类的对象实例在序列化时被映射为 Hprose 对象类型。自定义类中的字段名，映射为 Hprose 对象类型中的属性名，自定义类中的字段值，映射为 Hprose 对象类型中的属性值，类中所有的字段值必须为上述可序列化类型。例如：

```
class User:
    pass

user = User()
user.name = 'Tom'
user.age = 28
user.birthday = date(1982, 2, 23)
user.sex = 1
user.married = True
```

当反序列化一个对象时，如果该对象所在的类未定义，该类将会被自动定义。

类所在的模块名和类名一起构成类的全名，在类的全名中，模块名部分的点分隔符会被替换为下滑线分隔符，模块和类名之间的也是通过下划线连接。

Hprose 在多语言交互时，对象类型是通过全名来识别的。

假设我们在 Python 中定义一个类 User，它所属模块为 hprose.data，那么当我们在跟 PHP 交互时，它是与 PHP 中的 hprose_data_User 类对应。

而当跟 java、ActionScript 或 C# 这类支持包或名字空间管理的语言进行交互时，它就跟 hprose.data.User，hprose_data.User，hprose.data_User 或者 hprose_data_User 这四个类相对应了。

所以我们应该保证这四个类只有一个被定义。如果有多个被定义，结果也只会跟其中某一个建立对应关系，至于是哪一个，这与查找方式有关，不是固定的。所以，应该避免同时定义多个这样的类。

通过 HproseClassManager 来注册自定义类型

在 Hprose 1.2 for Python 中，通过 HproseClassManager 的 register 方法可以让你不改变类名定义就可以与其它语言进行交互。

例如您有一个命名为 User 的类，希望传递给 Java，Java 中与之对应的类是 my.package.User，那么可以这样做：

```
from hprose.io import HproseClassManager  
HproseClassManager.register(User, "my_package_User")
```

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 Python 服务器，在本章中您将深入的了解 Hprose for Python 服务器的更多细节。

本章提要

- 常用服务器介绍
- 发布服务
- 服务器开关
- 服务器事件
- 会话管理
- WSGI 中间件

常用服务器介绍

在前面快速入门一章里，我们介绍了如何使用 Hprose for Python 自带的独立服务器发布服务，但是在实际部署服务时，我们几乎从来不用这种方式，这种方式一般仅用于开发期间的调试。

Hprose for Python 服务可以以标准的 WSGI 应用程序发布，可以搭配所有支持 WSGI 的服务器运行，我们下面就来介绍几个最常用的服务器以及如何在上面配置发布 Hprose for Python 服务。

mod_wsgi 方式

mod_wsgi 是 Apache 的 wsgi 模块，它的官方网站是：<http://code.google.com/p/modwsgi/>。

下面我们以 ubuntu 9.10 为例，来介绍 mod_wsgi 的安装与 Hprose for Python 的服务发布。

安装 mod_wsgi 非常简单，只需要执行：

```
apt-get install mod_wsgi
```

即可完成安装。

然后在 /var/www 目录下，新建 hprose_wsgi.py：

```
#!/usr/bin/python
# encoding: utf-8

from hprose.httpserver import HproseHttpService

def hello(name):
    return 'Hello' + ' ' + name;

application = HproseHttpService()
application.addFunction(hello)
```

上面的程序跟之前的 Hello 服务器的例子很相似，只是把 HproseHttpServer 换成了 HproseHttpService。另外，这里的 application 这个名字是固定的，不能改变。application 不需要执行 start 方法。

然后让我们配置发布该服务：

修改 Apache 的站点配置文件，在其中加入：

```
WSGIScriptAlias /hprose /var/www/hprose_wsgi.py
<Directory /var/www/>
    Order allow,deny
    allow from all
</Directory>
```

然后重新启动 Apache 服务器就可以啦。

现在打开浏览器，在地址栏中输入：<http://localhost/hprose/>，如果浏览器中显示如下内容：

```
Fa1{s5"hello"}z
```

就说明服务已经发布成功啦。

Hprose 在 mod_wsgi 的优化模式下可以运行的更快，开启 mod_wsgi 优化的方法很简单，打开 /etc/apache2/mods-enabled/wsgi.conf，然后将其中的：

```
#WSGI PythonOptimize 0
```

改为

```
WSGI PythonOptimize 2
```

即可。

uWSGI 方式

uWSGI 是一个用纯 C 语言编写的快速的 wsgi 服务器，可以与 apache2，nginx，cherokee 和 lighttpd 这些 Web 服务器方便的集成。下面我们以 cherokee 为例，来介绍如何使用 uWSGI 模块发布。

下面来分别介绍在 Ubuntu 和 Mac OS X 上的安装与发布。

在 Ubuntu 上的安装与发布

首先安装 uWSGI，首先从官方网站下载最新版本：<http://projects.unbit.it/uwsgi/>。

执行如下命令安装编译依赖：

```
apt-get install python2.6-dev
apt-get install libxml2-dev
```

解压缩 uWSGI 后，进入解压后的目录，执行：

```
make -f Makefile.Py26
sudo cp uwsgi26 /usr/local/bin/uwsgi
```

这样 uWSGI 安装完成，接下来安装 cherokee。

Ubuntu 9.10 的官方软件库中的 cherokee 比较老，不包含 uWSGI 模块，所以我们需要从 cherokee 的软件库中安装：

```
sudo add-apt-repository ppa:cherokee-webserver/ppa
sudo apt-get update
sudo apt-get install cherokee
```

这样，cherokee 就安装好了，接下来执行：

```
sudo cherokee-admin -u
```

然后打开浏览器，输入：<http://127.0.0.1:9090/>，之后回车。如果看到 cherokee 的管理界面，就说明 cherokee 安装成功了。

然后在 /var/www 目录下，新建 hprose_wsgi.py：

```
#!/usr/bin/python
# encoding: utf-8

from hprose.httpserver import HproseHttpService

def hello(name):
    return 'Hello' + ' ' + name;

application = HproseHttpService()
application.addFunction(hello)
```

这个文件跟前面介绍的 mod_wsgi 时的文件是一模一样的，这里不多做解释。

接下来，在该目录下新建 uwsgi.xml：

```
<uwsgi>
<pythonpath>/var/www/</pythonpath>
<app mountpoint="/hprose">
    <script>hprose_wsgi</script>
</app>
</uwsgi>
```

这些准备工作完成后，就可以在 cherokee 上配置 uWSGI 了。

在 cherokee 管理界面上点击“虚拟主机设置→default→扩展设置→Wizards→Platforms->uWSGI run Wizard”。然后在“服务器配置文件”框中输入：

```
/var/www/uwsgi.xml
```

然后点“Submit”，再点保存重启，就配置完成啦。

现在打开浏览器，在地址栏中输入：<http://localhost/hprose/>，如果浏览器中显示如下内容：

```
Fa1{s5"hello"}z
```

就说明服务已经发布成功啦。

在 Mac OS X 上的安装与发布

在 Mac OS X 上你需要首先安装 XCode 开发环境。之后你可以通过下载源码方式或者通过 homebrew 方式来安装 uwsgi 和 cherokee。我们推荐您使用 homebrew 方式来安装。

homebrew 是 Mac OS X 上新兴的软件包管理工具，官方地址是：<http://mxcl.github.com/homebrew/>。

安装好 homebrew 之后，只需要执行：

```
brew install uwsgi
brew install cherokee
```

即可安装好这两个软件。

配置与在 Ubuntu 上配置类似，只是在最后还需要修改一下 cherokee 中的 uWSGI 启动参数，因为默认情况下 uWSGI 启动可能找不到 Python 安装路径。

具体配置方法如下，在完成与 Ubuntu 同样的配置之后，打开“虚拟主机设置→default→扩展设置→/hprose→处理→uWSGI 1”，然后在“解释器”输入框中，最后添加上

```
-H /System/Library/Frameworks/Python.framework/Versions/2.6
```

再点保存重启，就配置完成啦。

如果你使用的是 2.5 或者是自行安装的 Python 版本，这里替换成相应的路径即可。

eurasia 方式

eurasia 是国产的高性能 python 服务器，它也支持 WSGI 方式发布 Hprose 服务。eurasia 目前处于高速发展阶段，所以各个版本之间的接口差别比较大。下面的代码是使用的 eurasia 官方网站的最新 3.1.0-pre_alpha2 版本接口编写的，与官方 3.0 版本文档记载的方式有所不同，但原理一样的：

```
#!/usr/local/bin/python2.6
# encoding: utf-8

from hprose.httpserver import HproseHttpService
from eurasia30.web import WSGIServer

def hello(name):
    return 'Hello' + ' ' + name;

app = HproseHttpService()
app.addFunction(hello)

server = WSGIServer(app, '127.0.0.1:3030')
server.run()
```

eurasia 在 Stackless Python 下运行效果会好一点，所以我们可以从 <http://www.stackless.com/> 下载相应的 Stackless Python 安装。上面的例子中，第一行的路径就是在 Mac OS X 下 Stackless Python 安装后的默认路径。

上面这个例子跟前面的纯 WSGI 应用的例子很类似，不同之处是，最后需要创建一个 WSGIServer，并将 HproseHttpService 的应用实例传入，然后通过 run 方法来启动 WSGIServer。

这个程序的启动方法跟独立服务器是一样的，它也是以独立服务器方式运行的。

上面介绍的这几种方式各有优点，大家可以根据实际情况选择适合自己的部署方式。下面我们在介绍 Hprose 具体用法的时候，仍然以自带的独立服务器为例来给大家讲解。

发布服务

前面我们在服务器介绍中已经了解了在各种服务器上发布服务的方法，但那只是最基本的用法。下面我们来详细介绍一下关于 Hprose for Python 服务发布的更多内容。

发布函数

前面我们都是以发布一个函数为例来讲解的，这里我们主要谈一下哪些函数可以作为 Hprose 服务发布。

实际上大部分函数都是可以作为 Hprose 服务发布的，甚至包括 Python 中的内置的函数。但如果参数或结果中包含有不可序列化的类型（比如 open，file 等）或具有输出性质的函数（比如 print），那么这种函数就不能够发布。

你可以同时发布多个函数，不论是你自定义的，还是 Python 内置的都可以。例如：

```
#!/usr/bin/python
# encoding: utf-8

from os import uname
from time import time
from hprose.httpserver import HproseHttpService
from wsgiref.simple_server import make_server

application = HproseHttpService()
application.addFunctions((uname, time))

server = make_server('0.0.0.0', 3030, application)
try:
    server.serve_forever()
except KeyboardInterrupt:
    exit()
```

上面这个例子，我们通过 addFunctions 方法，同时发布了 python 内置的 uname 和 time 这两个函数。

并且我们直接通过 Python 内置的 wsgi 服务器发布了服务，还没有使用 Hprose 内置的 HproseHttpServer。实际上 HproseHttpServer 本身也是通过 Python 内置的 wsgi 服务器发布服务的。

发布方法

Hprose for Python 也支持发布类方法、静态方法和对象实例方法，如下例：

```
#!/usr/bin/python
# encoding: utf-8
from hprose.httpserver import HproseHttpService, HproseHttpServer
class Example1:
    def foo():
        return 'foo'
    foo = staticmethod(foo)
    def bar(cls):
        return cls.__name__ + '.bar'
    bar = classmethod(bar)
    def foobar(self):
```

```

        return self.__class__.__name__ + '.foobar'

exam1 = Example1()

application = HproseHttpService()
application.addMethod('foobar', exam1)
application.addMethods(('foo', 'bar'), Example1)

server = HproseHttpServer('0.0.0.0', 3030, application)
server.start()

```

这里 foo 是一个静态方法，bar 是一个类方法，所以添加时第二个参数是类名。而 foobar 是一个实例方法，所以第二个参数是 Example1 的一个实例对象 exam1。

现在你可能会有这样的疑问，如果要同时发布两个不同类中的同名方法的话，会不会有冲突呢？如何来避免冲突呢？

别名机制

确实会遇到这种情况，就是当发布的方法同名时，后添加的方法会将前面添加到方法给覆盖掉，在调用时，你永远不可能调用到先添加的同名方法。不过 Hprose 提供了一种别名机制，可以解决这个问题。要用自然语言来解释这个别名机制的话，不如直接看代码示例更直接一些：

```

#!/usr/bin/python
# encoding: utf-8

from hprose.httpserver import HproseHttpService, HproseHttpServer

def hello(name):
    return 'Hello ' + name;

class Example1:
    def foo():
        return 'foo'
    foo = staticmethod(foo)
    def bar(cls):
        return cls.__name__ + '.bar'
    bar = classmethod(bar)
    def foobar(self):
        return self.__class__.__name__ + '.foobar'

class Example2:
    def foo():
        return 'foo, too'
    foo = staticmethod(foo)
    def bar(cls):

```

```

        return cls.__name__ + '.bar'
    bar = classmethod(bar)
    def foobar(self):
        return self.__class__.__name__ + '.foobar'

exam1 = Example1()
exam2 = Example2()

application = HproseHttpService()
application.addFunction(hello, 'hi')
application.addMethod('foobar', exam1, 'exam1_foobar')
application.addMethod('foobar', exam2, 'exam2_foobar')
application.addMethods(('foo', 'bar'), Example1, ('exam1_foo', 'exam1_bar'))
application.addMethods(('foo', 'bar'), Example2, 'exam2')

server = HproseHttpServer('0.0.0.0', 3030, application)
server.start()

```

从上面这个例子，我们就会发现不论是函数还是方法都可以通过别名来发布，只需要在最后再加一个别名参数就可以了。

同时添加多个方法（或函数）时，别名也应该是多个，并且个数要跟方法（或函数）名个数相同，且一一对应。但是也可以只指定一个别名前缀，别名前缀会跟前面的方法名自动以下划线相连组成别名。

最后要注意的一点是，通过别名发布的功能（或函数）在调用时如果用原方法（或函数）名调用是调用不到的，也就是说只能用别名来调用。

发布对象

除了向上面通过 addMethod 和 addMethods 发布方法以外，Hprose 可以让您更方便的发布一个对象上的方法，那就是使用 addInstanceMethods。addInstanceMethods 用来发布指定对象上的指定类层次上声明的所有实例方法。它有三个参数，其中后两个是可选参数。如果您在使用 addInstanceMethods 方法时，不指定类层次（或者指定为 NULL），则发布这个对象所在类上声明的所有实例方法。这个方法也支持指定名称空间（别名前缀）。

例如：

```

#!/usr/bin/python
# encoding: utf-8

from hprose.httpserver import HproseHttpService, HproseHttpServer

class Example1:
    def foo(self):
        return 'foo'
    def bar(self):
        return 'bar'

```

```

class Example2:
    def foo(self):
        return 'foo, too'
    def bar(self):
        return 'bar, too'

exam1 = Example1()
exam2 = Example2()

application = HproseHttpService()
application.addInstanceMethods(exam1, None, 'exam1')
application.addInstanceMethods(exam2, None, 'exam2')

server = HproseHttpServer('0.0.0.0', 3030, application)
server.start()

```

这样 exam1 中 foo、bar 会以 exam1_foo、exam1_bar 的别名发布 ,exam2 中的 foo、bar 会以 exam2_foo、exam2_bar 的别名发布。

发布类

跟 addInstanceMethods 方法类似 ,使用 addClassMethods 可以让您更方便的发布一个类上的类方法 , 使用 addStaticMethods 可以让您更方便的发布一个类上的静态方法。addClassMethods 有三个参数 , 其中后两个是可选参数。第一个参数是方法的发布类 , 第二个参数为方法的执行类 , 第三个参数为名称空间 (别名前缀)。addStaticMethods 有两个参数 , 其中后一个是可选参数。第一个参数是方法的发布类 , 第二个参数为名称空间 (别名前缀)。

迷失的方法

当客户端调用一个服务器端没有发布的方法时 , 默认情况下 , 服务器端会抛出错误。但是如果你希望对客户端调用的不存在的方法在服务器端做特殊处理的话 , 你可以通过 addMissingFunction 方法来实现。

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。

使用 addMissingFunction 发布的方法可以是实例方法、类方法、静态方法、函数或者 lambda 表达式 , 但是只能发布一个。如果多次调用 addMissingFunction 方法 , 将只有最后一次发布的有效。

用 addMissingFunction 发布的方法参数应该为两个 :

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 的列表长度为 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 的列表。

除了可直接使用 addMissingFunction 来处理迷失的方法以外 , 你还可以通过 addFunction 或 addMethod 发布一个别名为星号 (*) 的方法。效果是一样的。

服务器开关

隐藏发布列表

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都更加灵活。

如果您不希望用户直接通过浏览器就可以查看发布列表的话，您可以禁止服务器接收 GET 请求。方法很简单，只需要在调用 start 方法之前调用 setGetEnabled 方法，将参数设置为 False 即可。

好了，现在通过 GET 方式访问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。

开启方法很简单，只需要在调用 start 方法之前调用 setDebugEnabled 方法，将参数设置为 True 即可。

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。开启方法与上面的开关类似，只需要在调用 start 方法之前调用 setP3PEnabled 方法，将参数设置为 True 即可。

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将 crossdomain 属性设置为 true 即可。当你在使用 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。这个只需要将 crossdomain.xml 放在服务器发布的根目录上即可。对于 WSGI 应用程序来说，使用后面将要介绍的 UrlMapMiddleware 中间件就可以实现。

对于 SilverLight 客户端来说，还支持 clientaccesspolicy.xml 这个客户端访问策略文件，它的设置方法跟 crossdomain.xml 是一样的，都是放在服务器发布的根目录上。

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。

HproseHttpService 提供了四个事件，它们分别是 `onBeforeInvoke`、`onAfterInvoke`、`onSendHeader` 和 `onSendError`。下面我们就来对这四个事件分别做一下介绍。

onBeforeInvoke 事件

当服务器端发布的方法被调用前，`onBeforeInvoke` 事件被触发，它有四个参数，他们从左到右的顺序分别是 `environ`，`name`，`args` 和 `byRef`。其中 `environ` 是环境变量集合，`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，其中前四个参数与 `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，该事件有两个参数。按从左到右的顺序分别是 `environ` 和 `header`。其中 `environ` 是环境变量集合，`header` 是返回的相应头部集合。

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，该事件有两个参数 `environ` 和 `error`。其中 `environ` 是环境变量集合，`error` 是错误对象。

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。

会话管理

Hprose for Python 支持 Session 管理，在创建 `HproseHttpService` 对象时，需要以 `Session` 在环境变量中的键值名作为参数，并在创建 `HproseHttpService` 对象后，将你所使用的 WSGI Session 中间件应用于其上即可。

已经通过测试的被 Hprose for Python 支持的 WSGI Session 中间件有：

- Breaker : <http://pypi.python.org/pypi/Beaker>
- wsgistate : <http://pypi.python.org/pypi/wsgistate>

在发布方法中使用 session 很简单，只需要将方法的最后一个参数定义为 session 即可。

WSGI 中间件

因为 HproseHttpService 的实例是一个标准的 WSGI 应用程序，因此你可以在它之上应用各种 WSGI 中间件，例如压缩输出，路径映射等。

Hprose for Python 中自带一个路径映射的中间件，它的使用方法如下：

```
#!/usr/bin/python
# encoding: utf-8

from hprose.httpserver import HproseHttpService, HproseHttpServer, UrlMapMiddleware

class Example1:
    def foo(self):
        return 'foo'
    def bar(self):
        return 'bar'

class Example2:
    def foo(self):
        return 'foo, too'
    def bar(self):
        return 'bar, too'

exam1 = Example1()
exam2 = Example2()

app1 = HproseHttpService()
app1.addInstanceMethods(exam1)

app2 = HproseHttpService()
app2.addInstanceMethods(exam2)

application = UrlMapMiddleware([('exam1', app1), ('exam2', app2)])
server = HproseHttpServer('0.0.0.0', 3030, application)
server.start()
```

这样，你就可以使用不同的路径发布不同的服务了，非常方便。

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 Python 客户端，在本章中您将深入的了解 Hprose for Python 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

直接通过远程方法名进行远程调用

在快速入门一章中，我们已经见识过这种方式的调用了，这里再来具一个例子来进行说明：

```
#!/usr/bin/python
# encoding: utf-8

from hprose.httpclient import HproseHttpClient
client = HproseHttpClient("http://www.hprose.com/example/")
print(client.sum(1, 2, 3, 4, 5))
print(client.Sum(6.0, 7.0, 8.0))
users = client.getUserList()
print(users)
for user in users:
    print(user.__dict__)
```

这个例子的运行结果是：

```
15
21.0
[<__main__.User object at 0x02607530>, <__main__.User object at 0x026071D0>, <__main__.User
object at 0x02607370>, <__main__.User object at 0x026074D0>]
{'age': 26, 'birthday': datetime.date(1983, 12, 3), 'married': True, 'name': 'Amy', 'sex':
2}
{'age': 20, 'birthday': datetime.date(1989, 6, 12), 'married': False, 'name': 'Bob', 'sex':
1}
{'age': 29, 'birthday': datetime.date(1980, 3, 8), 'married': True, 'name': 'Chris', 'sex':
0}
{'age': 27, 'birthday': datetime.date(1992, 6, 14), 'married': False, 'name': 'Alex', 'sex':
3}
```

从这个例子中，我们可以看出通过远程方法名进行调用时，远程方法名是不区分大小写的，所以不论是写 sum 还是 Sum 都可以正确调用，如果远程方法返回结果中包含有某个类的对象，而该类并没有在客户端明确定义的话，Hprose 会自动帮你生成这个类的定义（例如上例中的 User 类），并返回这个类的对象。

通过代理对象进行远程调用

如果服务发布方法时使用了别名前缀，那么客户端每次调用时，都需要写带了别名前缀的完整方法名，这样多少有些不便。不过 Hprose 提供了更简单的方式来调用带有别名前缀的方法，那就是通过代理对象。

下面这个例子是通过代理对象来调用“Hprose for Python 服务器”→“发布服务”→“发布对象”这一小节中发布的方法：

```

#!/usr/bin/python
# encoding: utf-8

from hprose.httpclient import HproseHttpClient
client = HproseHttpClient("http://localhost:3030/")
exam1 = client.exam1
exam2 = client.exam2
print(exam1.foo())
print(exam1.bar())
print(exam2.foo())
print(exam2.bar())

```

这个例子中，通过 `client.exam1` 得到了 `exam1` 这个代理对象，通过 `client.exam2` 得到了 `exam2` 这个代理对象，之后就可以直接在 `exam1` 和 `exam2` 上调用它们的 `foo()` 和 `bar()` 方法了。运行结果如下：

```

foo
bar
foo, too
bar, too

```

通过 `Invoke` 方法进行远程调用

非引用参数传递

上面介绍的通过远程方法名进行远程调用的例子就是非引用参数传递，下面是用 `invoke` 方法重写的代码：

```

#!/usr/bin/python
# encoding: utf-8

from hprose.httpclient import HproseHttpClient
client = HproseHttpClient("http://www.hprose.com/example/")
print(client.invoke('sum', (1, 2, 3, 4, 5)))
print(client.invoke('Sum', [6.0, 7.0, 8.0]))
users = client.invoke('getUserList')
print(users)
for user in users:
    print(user.__dict__)

```

运行结果与上面例子的运行结果完全相同。但是我们发现用 `invoke` 方法并不方便，因为当有参数时，必须要把参数单独放入一个元组或列表中才可以进行传递。所以通常我们无需直接使用 `invoke` 方法，除非我们需要动态调用。另外，还有一种情况下，你会用到 `invoke` 方法，那就是在进行引用参数传递时。

引用参数传递

下面这个例子很好的说明了如何进行引用参数传递：

```
#!/usr/bin/python
# encoding: utf-8

from hprose.httpclient import HproseHttpClient
client = HproseHttpClient("http://www.hprose.com/example/")
args = [{"Mon": 1, "Tue": 2, "Wed": 3, "Thu": 4, "Fri": 5, "Sat": 6, "Sun": 7}]
print(args)
result = client.invoke("swapKeyAndValue", args, byRef = True)
print(args)
print(result)
```

上面程序的运行结果如下：

```
[{"Wed": 3, "Sun": 7, "Fri": 5, "Tue": 2, "Mon": 1, "Thu": 4, "Sat": 6}
[{"1": "Mon", 2: "Tue", 3: "Wed", 4: "Thu", 5: "Fri", 6: "Sat", 7: "Sun"}]
{1: "Mon", 2: "Tue", 3: "Wed", 4: "Thu", 5: "Fri", 6: "Sat", 7: "Sun"}
```

我们看到运行前后，args 中的值已经改变了。

这里有一点要注意，当参数本身是数组时，该数组应该作为参数数组的第一个元素传递，例如上例中的 args 是：

```
[{"Mon": 1, "Tue": 2, "Wed": 3, "Thu": 4, "Fri": 5, "Sat": 6, "Sun": 7}]
```

而不能只写：

```
{"Mon": 1, "Tue": 2, "Wed": 3, "Thu": 4, "Fri": 5, "Sat": 6, "Sun": 7}
```

否则程序将会出错，或者在调用中陷入等待状态，这样的错误有时候不容易被找到，因此一定要注意这一点。

异步调用

非引用参数传递

异步调用是通过指定回调函数的方式来实现的，看下面这个例子：

```
#!/usr/bin/python
# encoding: utf-8

from sys import stdout
from hprose.httpclient import HproseHttpClient
```

```

client = HproseHttpClient("http://www.hprose.com/example/")
client.sum(1, 2, 3, 4, 5, callback=lambda result: stdout.write(str(result) + "\r\n"))
def printResult(result, args):
    print(result, args)
client.Sum(6.0, 7.0, 8.0, callback=printResult)
def printUsers(users):
    print(users)
    for user in users:
        print(user.__dict__)
users = client.getUserList(callback=printUsers)
raw_input()

```

它与前面同步调用的第一个例子是相同的，但是结果顺序不一定是按照调用的顺序输出的。

回调函数可以使用 lambda 表达式，也可以使用函数。不论是 lambda 表达式还是函数，参数个数可以是 0 个，也可以是 1 个或 2 个。如果参数个数是零个，在执行回调方法时，不会有结果传入，这通常用于调用没有结果的远程方法。如果参数个数是一个，在执行回调方法时，结果会作为参数来传入。如果参数个数是两个，则第一个参数是执行结果，第二个参数是调用参数。

使用 lambda 表达式时需要注意，在 lambda 表达式中不能出现像 print 这样的语句，只能使用表达式，因此在上面的例子中，在 lambda 表达式中输出时，我们用的是 sys.stdout.write。通常使用函数作为回调更为灵活方便一些。

引用参数传递

跟同步方式不同，在异步方式下，可以直接通过远程方法名进行引用参数传递的远程调用。例如：

```

#!/usr/bin/python
# encoding: utf-8

from sys import stdout
from hprose.httpclient import HproseHttpClient
client = HproseHttpClient("http://www.hprose.com/example/")
arg = {"Mon": 1, "Tue": 2, "Wed": 3, "Thu": 4, "Fri": 5, "Sat": 6, "Sun": 7}
print(arg)
def printResult(result, args):
    print(result)
    print(args)
client.swapKeyAndValue(arg, callback=printResult, byRef = True)
raw_input()

```

这个运行结果为：

```

{'Wed': 3, 'Sun': 7, 'Fri': 5, 'Tue': 2, 'Mon': 1, 'Thu': 4, 'Sat': 6}
{1: 'Mon', 2: 'Tue', 3: 'Wed', 4: 'Thu', 5: 'Fri', 6: 'Sat', 7: 'Sun'}
[{1: 'Mon', 2: 'Tue', 3: 'Wed', 4: 'Thu', 5: 'Fri', 6: 'Sat', 7: 'Sun'}]

```

这个跟前面介绍同步调用时的引用参数传递例子很相似，但是要注意，使用远程方法名进行参数传递跟使用 invoke 方法进行参数传递有一点区别，那就是使用远程方法名进行参数传递时，不需要将参数放入列表中。

同样，异步调用也支持通过代理对象和通过 invoke 方法进行远程调用，而且都支持引用参数传递，用法跟上面类似，结合同步调用和上面异步调用的例子，您应该可以做到无师自通了。

异常处理

同步调用中的异常处理

Hprose for Python 的客户端在同步调用过程中，如果服务器端发生错误，或者客户端本身发生错误，异常将在客户端被直接抛出，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 HproseException 类型。但是在调用过程中也可能抛出其它类型的异常。

例如，当调用不存在的方法时：

```
#!/usr/bin/python
# encoding: utf-8
from hprose.httpclient import HproseHttpClient
client = HproseHttpClient("http://www.hprose.com/example/")
try:
    client.notexisted()
except Exception, e:
    print e
```

运行结果如下：

```
Can't find this function notexisted().
file: /var/www/hprfc/website/example/hproseHttpServer.php
line: 157
trace: #0 /var/www/hprfc/website/example/hproseHttpServer.php(385): HproseHttpServer->doInvoke()
#1 /var/www/hprfc/website/example/hproseHttpServer.php(402): HproseHttpServer->handle()
#2 /var/www/hprfc/website/example/index.php(60): HproseHttpServer->start()
#3 {main}
```

异步调用中的异常处理

Hprose for Python 的客户端在异步调用过程中，如果服务器或客户端发生错误，异常不会被抛出，不会影响程序执行，但如果需要处理异常，可以通过 onError 事件来捕捉所有异步调用过程中发生的异常，例如：

```
#!/usr/bin/python
# encoding: utf-8
from hprose.httpclient import HproseHttpClient
```

```

client = HproseHttpClient("http://www.hprose.com/example/")
def errorhandler(name, error):
    print(name)
    print(error)
client.onError = errorhandler
client.notexisted(callback=lambda:True)
raw_input()

```

上面这个程序中，我们定义了 `errorhandler` 这个方法，它有两个参数，第一个参数表示发生错误时所调用的方法名，第二个参数是捕捉到的异常。上面的程序运行结果如下：

```

notexisted
Can't find this function notexisted().
file: /var/www/hprfc/website/example/hproseHttpServer.php
line: 157
trace: #0 /var/www/hprfc/website/example/hproseHttpServer.php(385): HproseHttpServer->doInvoke()
#1 /var/www/hprfc/website/example/hproseHttpServer.php(402): HproseHttpServer->handle()
#2 /var/www/hprfc/website/example/index.php(60): HproseHttpServer->start()
#3 {main}

```

注意：`onError` 事件只对异步调用有效，对同步调用无效。

超时设置

Hprose 1.2 for Python 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，单位为秒。当调用超过 `timeout` 的时间后，调用将被中止。

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、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 `setProxy` 来设置 http 代理服务器的地址和端口。当参数是两个时，第一个参数表示主机名，第二个参数表示端口号。当参数为一个时，参数是代理服务器的完整地址字符串，例如："http://10.54.1.39:8000"。

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
- Context-Length
- 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。

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。通过 keepAlive 设置为 True 时，表示开启持久连接特征。keepAliveTimeout 可以设置持久连接超时时间，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 RawWithTag。

Serialized 模式

Serialized 模式下，结果以序列化模式返回，在 Python 中，序列化的结果以 string 类型返回。用户可以通过 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。

RawWithTag 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 RawWithTag 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。