



用户手册

1.3

（Ruby版）

目录

[前言 1](#_Toc322941023)

[本章提要 1](#_Toc322941024)

[欢迎使用Hprose 2](#_Toc322941025)

[体例 3](#_Toc322941026)

[菜单描述 3](#_Toc322941027)

[屏幕截图 3](#_Toc322941028)

[代码范例 3](#_Toc322941029)

[运行结果 3](#_Toc322941030)

[获取帮助 3](#_Toc322941031)

[电子文档 3](#_Toc322941032)

[在线支持 3](#_Toc322941033)

[联系我们 4](#_Toc322941034)

[第一章 快速入门 5](#_Toc322941035)

[本章提要 5](#_Toc322941036)

[安装Hprose for Ruby 6](#_Toc322941037)

[安装方法 6](#_Toc322941038)

[创建Hprose的Hello服务器 6](#_Toc322941039)

[创建Hprose的Hello客户端 7](#_Toc322941040)

[第二章 类型映射 9](#_Toc322941041)

[本章提要 9](#_Toc322941042)

[基本类型 10](#_Toc322941043)

[值类型 10](#_Toc322941044)

[引用类型 10](#_Toc322941045)

[基本类型的映射 11](#_Toc322941046)

[序列化类型映射 11](#_Toc322941047)

[反序列化类型映射 11](#_Toc322941048)

[容器类型 12](#_Toc322941049)

[列表类型 12](#_Toc322941050)

[字典类型 12](#_Toc322941051)

[对象类型 13](#_Toc322941052)

[通过HproseClassManager来注册自定义类型 14](#_Toc322941053)

[第三章 服务器 15](#_Toc322941054)

[本章提要 15](#_Toc322941055)

[服务发布方式 16](#_Toc322941056)

[Rackup方式 16](#_Toc322941057)

[独立服务器方式 16](#_Toc322941058)

[Rails on Rack方式 17](#_Toc322941059)

[发布服务 19](#_Toc322941060)

[发布函数 19](#_Toc322941061)

[发布语句块 20](#_Toc322941062)

[发布方法 20](#_Toc322941063)

[别名机制 21](#_Toc322941064)

[发布对象 22](#_Toc322941065)

[发布类 23](#_Toc322941066)

[迷失的方法 23](#_Toc322941067)

[服务器开关 23](#_Toc322941068)

[隐藏发布列表 23](#_Toc322941069)

[调试开关 24](#_Toc322941070)

[P3P开关 24](#_Toc322941071)

[跨域开关 24](#_Toc322941072)

[服务器事件 24](#_Toc322941073)

[on\_before\_invoke事件 25](#_Toc322941074)

[on\_after\_invoke事件 25](#_Toc322941075)

[on\_send\_header事件 25](#_Toc322941076)

[on\_send\_error事件 25](#_Toc322941077)

[会话管理 25](#_Toc322941078)

[Rack中间件 25](#_Toc322941079)

[第四章 客户端 28](#_Toc322941080)

[本章提要 28](#_Toc322941081)

[同步调用 29](#_Toc322941082)

[直接通过远程方法名进行远程调用 29](#_Toc322941083)

[通过代理对象进行远程调用 30](#_Toc322941084)

[通过Invoke方法进行远程调用 31](#_Toc322941085)

[非引用参数传递 31](#_Toc322941086)

[引用参数传递 31](#_Toc322941087)

[异步调用 32](#_Toc322941088)

[非引用参数传递 32](#_Toc322941089)

[引用参数传递 33](#_Toc322941090)

[异常处理 33](#_Toc322941091)

[同步调用中的异常处理 33](#_Toc322941092)

[异步调用中的异常处理 34](#_Toc322941093)

[超时设置 35](#_Toc322941094)

[HTTP参数设置 35](#_Toc322941095)

[代理服务器 35](#_Toc322941096)

[HTTP标头 35](#_Toc322941097)

[持久连接 35](#_Toc322941098)

[调用结果返回模式 35](#_Toc322941099)

[Serialized模式 36](#_Toc322941100)

[Raw模式 36](#_Toc322941101)

[RawWithEndTag模式 36](#_Toc322941102)

# 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
* 报告漏洞与需求

## 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](mailto:manager@hprfc.com?subject=市场及大型项目合作) |
| 产品购买及项目定制： | [sales@hprfc.com](mailto:sales@hprfc.com?subject=产品购买及项目定制) |
| 技术支持： | [support@hprfc.com](mailto:support@hprfc.com?subject=技术支持) |
| 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 Ruby进行分布式程序开发的全过程。

## 本章提要

* 安装Hprose for Ruby
* 创建Hprose的Hello服务器
* 创建Hprose的Hello客户端

## 安装Hprose for Ruby

Hprose for Ruby支持Ruby 1.8.5及其更高版本。

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Rack的Web服务器运行。

### 安装方法

如果您已经安装了Ruby，进入hprose for ruby的目录，如果其中没有hprose-1.2.0.gem的话，您可以执行：

ruby gem.spec

生成hprose-1.2.0.gem。

然后在该目录下执行：

gem install hprose

即可安装成功。

## 创建Hprose的Hello服务器

创建Ruby的Hprose的服务器非常简单。Hprose for Ruby服务器是标准的Rack应用程序，所以你需要先安装rack，安装很简单，只需要执行：

gem install rack

就安装成功了。

然后创建服务发布程序，你只需几行代码就可以创建它：

require "rubygems"

require "hproseserver"

def hello(name)

'Hello ' + name + '!'

end

app = HproseHttpService.new()

app.add(method(:hello))

run app

将上面这段代码保存为helloservice.ru，注意扩展名是ru，不是rb。

然后执行：

rackup helloservice.ru

现在你可以在同一台机器上打开浏览器，输入：http://127.0.0.1:9292/，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

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

#!/usr/bin/ruby

require "rubygems"

require "hproseclient"

client = HproseHttpClient.new('http://localhost:9292/')

puts client.hello('World')

然后在命令行中输入：

ruby helloclient.rb

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：

C:/Ruby/lib/ruby/1.8/net/http.rb:560:in `initialize': 由于目标计算机积极拒绝，无法连接。 - connect(2) (Errno::ECONNREFUSED)

from C:/Ruby/lib/ruby/1.8/net/http.rb:560:in `open'

from C:/Ruby/lib/ruby/1.8/net/http.rb:560:in `connect'

from C:/Ruby/lib/ruby/1.8/timeout.rb:53:in `timeout'

from C:/Ruby/lib/ruby/1.8/timeout.rb:93:in `timeout'

from C:/Ruby/lib/ruby/1.8/net/http.rb:560:in `connect'

from C:/Ruby/lib/ruby/1.8/net/http.rb:553:in `do\_start'

from C:/Ruby/lib/ruby/1.8/net/http.rb:548:in `start'

from C:/Ruby/lib/ruby/gems/1.8/gems/hprose-1.0.0/lib/hprose/httpclient.rb:84:in `\_post'

from C:/Ruby/lib/ruby/gems/1.8/gems/hprose-1.0.0/lib/hprose/httpclient.rb:70:in `send\_data'

from C:/Ruby/lib/ruby/gems/1.8/gems/hprose-1.0.0/lib/hprose/client.rb:99:in `\_invoke'

from C:/Ruby/lib/ruby/gems/1.8/gems/hprose-1.0.0/lib/hprose/client.rb:67:in `invoke'

from C:/Ruby/lib/ruby/gems/1.8/gems/hprose-1.0.0/lib/hprose/client.rb:121:in `method\_missing'

from helloclient.rb:4

到这里，您应该已经掌握Hprose for Ruby的基本用法啦。接下来，就让我们一起对Hprose for Ruby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 Ruby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（-231～231-1）。 |
| 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

Ruby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Ruby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

| **Ruby类型** | **Hprose类型** |
| --- | --- |
| Fixnum中-2147483648～2147483647范围之内的数字（含边界） | 整型 |
| Fixnum中-2147483648～2147483647范围之外的数字，Bignum | 长整型 |
| Float | 浮点型 |
| 0.0/0.0 | 非数 |
| 1.0/0.0 | 正无穷大 |
| -1.0/0.0 | 负无穷大 |
| true | 布尔真 |
| false | 布尔假 |
| nil | 空 |
| 非utf8编码的String | 二进制型（或空串） |
| utf8编码的单字元String | 字符 |
| Symbol，utf8编码的多字元String | 字符串型（或空串） |
| Time | 日期/时间/日期时间型 |

#### 反序列化类型映射

| **Hprose类型** | **Ruby类型** |
| --- | --- |
| 整型 | Fixnum |
| 长整型 | Fixnum，Bignum |
| 浮点型 | Float |
| 非数 | 0.0/0.0 |
| 正无穷大 | 1.0/0.0 |
| 负无穷大 | -1.0/0.0 |
| 布尔真 | true |
| 布尔假 | false |
| 空 | nil |
| 空串 | "" |
| 二进制/字符/字符串型 | String |
| 日期/时间/日期时间型 | Time |

## 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Ruby的Array和Hash类型。

### 列表类型

Ruby中的Array类型，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。

另外，Range、MatchData类型也被序列化为Hprose的列表类型。

但是反序列化时，Hprose的列表类型只被反序列化为Ruby的Array类型。

### 字典类型

Ruby中的Hash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

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

例如：

class User

attr\_accessor :name, :age, :birthday, :sex, :married

end

user = User.new()

user.name = 'Tom'

user.age = 28

user.birthday = Time.local(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。

类可以嵌套定义，嵌套定义的所有类名一起构成类的全名，在类的全名中，用于分割类名的双冒号分隔符会被替换为下滑线分隔符。Hprose在多语言交互时，对象类型是通过全名来识别的。

例如：

class My

class Package

class User

attr\_accessor :name, :age, :birthday, :sex, :married

end

end

end

这个My::Package::User类，当我们在跟PHP交互时，它是与PHP中的My\_Package\_User类对应的。

所以上面的类也可以这样定义：

class My\_Package\_User

attr\_accessor :name, :age, :birthday, :sex, :married

end

效果是相同的。

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My.Package.User，My\_Package.User，My.Package\_User或者My\_Package\_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 Ruby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：

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

## 本章提要

* 服务发布方式
* 发布服务
* 服务器开关
* 服务器事件
* 会话管理
* Rack中间件

## 服务发布方式

Hprose for Ruby服务从本质上来讲是一个Rack应用程序，因此所有Rack支持的方式，它都适用。Hprose for Ruby服务之所有以Rack方式来实现有两个原因：

首先，Rack是Ruby在Web开发中的一个标准，它与WSGI在Python中的地位和作用是相当的。因此，采用这种方式来实现，将可以最大程度的支持最多的服务器。

其次，采用Rack方式在运行效率上相当高效，因此它可以更充分的发挥服务器的性能。

下面我们来介绍几种最主要的服务发布方式。

### Rackup方式

在前面快速入门一章里，我们介绍了如何以Rackup方式来发布服务，我们使用的是默认的Webrick服务器，但是在实际部署服务时，我们几乎从来不用这个服务器，这个服务器一般仅用于开发期间的调试。

另外常用服务器还有mongrel，thin等。这些服务器的启动方式基本上是相同的，都可以通过rackup来启动，例如：

rackup -s mongrel helloservice.ru

不过前提是，你要安装有这些服务器，安装这些服务器很简单，例如安装mongrel只需要执行：

gem install mongrel

就可以啦。

不推荐在Windows下使用thin，它在Windows下运行rack应用很不稳定。但是在Linux下，thin表现的很好，因此如果您使用Linux作为服务器，则推荐使用thin。

rackup有许多参数，可以通过这些参数来设置采用何种服务器，以及监听的主机名、端口号，服务运行模式等许多选项。

### 独立服务器方式

rackup方式的好处是，一个ru文件可以通过多种服务器来启动它，只要在使用rackup时指定不同的服务器参数即可。但是有时候我们就是确定只用某一个服务器，这时候你可以直接以rb文件方式来写一个独立运行的服务器，例如下面这个例子：

require "rubygems"

require "ebb"

require "hproseserver"

def hello(name)

'Hello ' + name + '!'

end

app = HproseHttpService.new()

app.add(:hello)

Ebb.start\_server(app)

它使用了Ebb作为发布服务器。Ebb是Linux下使用的一个高性能的rack服务器。在Linux下可以直接通过gem install ebb来安装它，但如果你使用的是ubuntu，需要注意一定要先用apt-get安装好ruby-dev这个包。

之后直接执行该程序即可启动服务器了。

独立服务器方式与rackup方式有一点区别，例如上面发布的hello方法，在独立服务器中，它是属于顶级对象的，也可以说它相当于一个函数，因此可以直接通过add方法后面跟方法名:hello（或者字符串"hello"）。而在rackup方式下，在ru文件中定义的hello方法实际上属于Rack::Builder的一个实例，因此如果直接在add方法后面跟方法名，将无法发布它，因此只能通过method来获取到hello方法然后再用add来发布它。

独立服务器方式是速度最快的，如果您追求的是速度的极致，那么您可以采用这种方式。

独立服务器的代码需要为不同的服务器单独编写，虽然主要语句没有差别，但在服务器启动的语句上还是有少许差别的，因此这种方式在实际应用中并不多见。

更多被广泛应用的是下面这种Rails on Rack方式。

### Rails on Rack方式

Ruby能有今天这么火热，大半归功于Rails的成功。因此大部分的Ruby应用实际上都是Rails的应用，所以如果能在Rails上直接发布Hprose服务，就再好不过了。

要实现这个愿望是没有问题，在Rails 2.3之后，Rails也实现了对Rack的支持。通过创建Rails Metal应用程序，即可实现对Hprose服务的发布。

虽然采用Rails on Rack方式发布Hprose服务的效率赶不上单独的Rackup和独立服务器方式。但因为Rails Metal应用程序要比普通的Rails Action快很多倍，因此在Rails上发布Hprose服务，比普通的Rails应用还是高效的多，因为目前大部分的Rails应用还是基于Rails Action来实现的。

下面我们来详细看一下如何在Rails上发布一个Hprose服务。

首先第一步是安装Rails，通过gem很容易安装，在Windows命令行或Linux/Unix终端上键入：

gem install rails

然后回车，即可完成安装。

然后通过rails命令来创建一个Rails应用，例如：

rails exam

之后将会显示：

create

create app/controllers

create app/helpers

create app/models

create app/views/layouts

create config/environments

create config/initializers

create config/locales

…

这样exam目录就建好了，下面我们来建一个metal应用：

进入exam目录，执行：

ruby script\generate metal helloservice

或

ruby script/generate metal helloservice

上面的命令取决于你使用的是Windows还是Linux/Unix。

然后我们进入app目录下的metal目录，编辑rails自动生成的helloservice.rb：

# Allow the metal piece to run in isolation

require(File.dirname(\_\_FILE\_\_) + "/../../config/environment") unless defined?(Rails)

require "rubygems"

require 'hproseserver'

def hello(name)

'Hello ' + name

end

class Helloservice

def self.call(env)

if env["PATH\_INFO"] =~ /^\/helloservice/

app = HproseHttpService.new()

app.add(:hello)

app.call(env)

else

[404, {"Content-Type" => "text/html"}, ["Not Found"]]

end

end

end

然后回到exam目录下，执行：

ruby script\server

或

ruby script/server

之后，如果显示类似如下内容：

=> Booting Mongrel

=> Rails 2.3.5 application starting on http://0.0.0.0:3000

=> Call with -d to detach

=> Ctrl-C to shutdown server

就说明Rails启动成功了。

现在打开浏览器，在地址栏中输入：http://localhost:3000/helloservice/，如果浏览器中显示如下内容：

Fa1{s5"hello"}z

就说明Hprose服务已经发布成功啦。

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Rackup方式为例来给大家讲解。

## 发布服务

现在您已经了解了各种发布服务的方式，下面在来看看如何发布过程、lambda表达式、语句块和方法吧。

### 发布函数

Ruby中实际上没有函数，所有看似函数的定义实际上都是方法。我们这里把属于顶级对象（Object）的方法称为函数。通过add\_function方法你可以对函数进行发布，例如：

app.add\_function(:hello)

app.add\_function("hello")

app.add\_function(method(:hello))

这三个写法是一样的，当然你还可以向前面的例子中那样把add\_function简写成add，但我们推荐您使用add\_function这种完整写法，因为它更加清晰也更加高效。

上面三种写法中最保险的是最后一种，因为在Rackup方式下，只有最后一种可以工作，因为在Rackup方式下，在ru文件中定义的方法都不属于顶级对象。

通过add\_function方法，你还可以发布过程（proc）和lambda表达式，例如：

app.add\_function(proc { |name| 'Hello ' + name + '!' }, "hello")

app.add\_function(Proc.new { |name| 'Hello ' + name + '!' }, "hello")

app.add\_function(lambda { |name| 'Hello ' + name + '!' }, "hello")

上面三种写法是一样的。但是发布过程和lambda表达式时，一定要注意最后要跟方法别名。例如下面的写法是错误的：

hello = lambda { |name| 'Hello ' + name + '!' }

app.add\_function(hello)

但是这样写是正确的：

hello = lambda { |name| 'Hello ' + name + '!' }

app.add\_function(hello, "hello")

你也可以给发布的方法提供别名，例如：

app.add\_function(method(:hello), "hi")

但这时，在客户端只能以别名hi来调用该方法，而不能用hello这个方法名来调用它了。

### 发布语句块

Hprose for Ruby提供了一个add\_block方法专门用来发布语句块，这是一个相当好用的功能，你不需要事先创建一个方法，也不会向发布一个lambda表达式或者proc那样看上去很复杂。发布语句块的语法看上去相当自然，例如用add\_block来发布一个hello方法是这样写的：

app.add\_block("hello") { |name|

'Hello ' + name + '!'

}

看上去跟定义一个方法差不多吧。

### 发布方法

Hprose for Ruby 也支持发布类方法和对象实例方法，如下例：

require "rubygems"

require "hproseserver"

class Example1

def foo

'foo'

end

def bar

'bar'

end

def Example1.foobar

'foobar'

end

end

exam1 = Example1.new()

app = HproseHttpService.new()

app.add\_methods([:foo, :bar], exam1)

app.add\_method(:foobar, Example1)

run app

这里foo，bar是对象实例方法，它们都属于对象exam1。而foobar是一个类方法，它属于Example1。

不管是类方法还是实例方法，都可以使用add\_method一个一个添加，也可以使用add\_methods成批添加。

另外，前面我们举的例子中：

app.add\_function(method(:hello))

还可以写成：

app.add\_method(:hello, self)

这两种写法是等价的。

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

有时会遇到这种情况，就是发布的方法出现同名时，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：

require "rubygems"

require "hproseserver"

class Example1

def foo

'foo'

end

def bar

'bar'

end

def Example1.foobar

'foobar'

end

end

exam1 = Example1.new()

class Example2

def foo

'foo, too'

end

def bar

'bar, too'

end

def Example2.foobar

'foobar, too'

end

end

exam2 = Example2.new()

app = HproseHttpService.new()

app.add\_methods([:foo, :bar], exam1, "exam1")

app.add\_method(:foobar, Example1, "exam1\_foobar")

app.add\_methods([:foo, :bar], exam2, "exam2")

app.add\_method(:foobar, Example2, "exam2\_foobar")

run app

从上面这个例子中，我们发现在发布单个方法时，需要指定完整的别名，而在批量发布方法时，只需要添加别名前缀就可以啦。

而实际上，同时添加多个方法（或函数）时，别名也可以是多个，但个数要跟方法（或函数）名个数相同，且一一对应。但一般只指定一个别名前缀即可，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，通过别名发布的方法在调用时如果用原方法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add\_method和add\_methods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add\_instance\_methods。add\_instance\_methods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add\_instance\_methods方法时，不指定类层次（或者指定为nil），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）。

例如上面例子中的：

app.add\_methods([:foo, :bar], exam1, "exam1")

app.add\_methods([:foo, :bar], exam2, "exam2")

可以替换为：

app.add\_instance\_methods(exam1, nil, "exam1")

app.add\_instance\_methods(exam2, nil, "exam2")

效果是一样的。

### 发布类

跟add\_instance\_methods方法类似，使用add\_class\_methods可以让您更方便的发布一个类上的类方法。add\_class\_methods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）。

例如上面例子中的：

app.add\_method(:foobar, Example1, "exam1\_foobar")

app.add\_method(:foobar, Example2, "exam2\_foobar")

可以替换为：

app.add\_class\_methods(Example1, nil, "exam1")

app.add\_class\_methods(Example2, nil, "exam2")

### 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。但是如果你希望能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add\_missing\_function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方法中没有查找到时，将调用这个特定的方法。

使用add\_missing\_function发布的方法可以是实例方法、类方法、lambda表达式或者过程，但是只能发布一个。如果多次调用add\_missing\_function方法，将只有最后一次发布的有效。

用add\_missing\_function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。

除了可以直接使用add\_missing\_function来处理迷失的方法以外，你还可以通过add\_function、add\_method或者add\_block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，只需要将get属性设置为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，只需要将debug属性设置为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，只需要将p3p属性设置为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Rackup这种启动服务方式，使用Rack的URLMap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on\_before\_invoke、on\_after\_invoke、on\_send\_header和on\_send\_error。下面我们就来对这四个事件分别做一下介绍。

### on\_before\_invoke事件

当服务器端发布的方法被调用前，on\_before\_invoke事件被触发，它有四个参数，他们从左到右的顺序分别是env，name，args和byref。其中env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。

### on\_after\_invoke事件

当服务器发布的方法被成功调用后，on\_after\_invoke事件被触发，其中前四个参数与on\_before\_invoke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，on\_after\_invoke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。

### on\_send\_header事件

当服务器返回响应头部时，on\_send\_header事件会被触发，该事件有俩个参数。按从左到右的顺序分别是env和header。其中env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。

### on\_send\_error事件

当服务器端调用发生错误，或者在on\_before\_invoke、on\_after\_invoke事件中抛出异常时，该事件被触发，该事件有两个参数env和error。其中env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 Ruby支持Session管理，在创建HproseHttpService对象后，将你所使用的Rack Session中间件应用于其上即可。

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。客户端调用时无需也不应该指定该参数，该参数会自动代入。

## Rack中间件

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

例如使用Rack中自带一个路径映射的中间件，它的使用方法如下：

require "rubygems"

require "hproseserver"

class Example1

def foo

'foo'

end

def bar

'bar'

end

def Example1.foobar

'foobar'

end

end

exam1 = Example1.new()

class Example2

def foo

'foo, too'

end

def bar

'bar, too'

end

def Example2.foobar

'foobar, too'

end

end

exam2 = Example2.new()

map '/exam1' do

app = HproseHttpService.new()

app.add\_instance\_methods(exam1, nil, "exam1")

app.add\_class\_methods(Example1, nil, "exam1")

run app

end

map '/exam2' do

app = HproseHttpService.new()

app.add\_instance\_methods(exam2, nil, "exam2")

app.add\_class\_methods(Example2, nil, "exam2")

run app

end

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 Ruby客户端，在本章中您将深入的了解Hprose for Ruby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/ruby

require "rubygems"

require "hproseclient"

client = HproseHttpClient.new("http://www.hprose.com/example/")

puts client.sum(1, 2, 3, 4, 5)

puts client.Sum(6.0, 7.0, 8.0)

users = client.getUserList

users.each { |user|

puts user

puts "name: " + user.name

puts "age: " + user.age.to\_s

puts "birthday: " + user.birthday.to\_s

puts "sex: " + user.sex.to\_s

puts "married: " + user.married.to\_s

}

这个例子的运行结果是：

15

21.0

#<User:0x10112ec90>

name: Amy

age: 26

birthday: Sat Dec 03 00:00:00 +0800 1983

sex: 2

married: true

#<User:0x10112efb0>

name: Bob

age: 20

birthday: Mon Jun 12 00:00:00 +0900 1989

sex: 1

married: false

#<User:0x10112eba0>

name: Chris

age: 29

birthday: Sat Mar 08 00:00:00 +0830 1980

sex: 0

married: true

#<User:0x101128ed0>

name: Alex

age: 27

birthday: Sun Jun 14 00:00:00 +0800 1992

sex: 3

married: false

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 Ruby服务器”→“发布服务”→“别名机制”这一小节中发布的方法：

#!/usr/bin/ruby

require "rubygems"

require "hproseclient"

client = HproseHttpClient.new('http://localhost:9292/')

exam1 = client["exam1"]

puts exam1.foo

puts exam1.bar

puts exam1.foobar

exam2 = client[:exam2]

puts exam2.foo

puts exam2.bar

puts exam2.foobar

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、bar和foobar方法了。

"exam1"和:exam2在这里作用是一样的。运行结果如下：

foo

bar

foobar

foo, too

bar, too

foobar, too

另外，如果别名是多个下滑线分割的，例如：my\_exam1\_foo，可以通过client["my\_exam1"]获取代理对象，也可以client["my"]["exam1"]分两次来获取代理对象，这是等价的。

### 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/ruby

require "rubygems"

require "hproseclient"

client = HproseHttpClient.new("http://www.hprose.com/example/")

puts client.invoke("sum", [1, 2, 3, 4, 5])

puts client.invoke("Sum", [6.0, 7.0, 8.0])

users = client.invoke("getUserList")

users.each { |user|

puts user

puts "name: " + user.name

puts "age: " + user.age.to\_s

puts "birthday: " + user.birthday.to\_s

puts "sex: " + user.sex.to\_s

puts "married: " + user.married.to\_s

}

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数组中才可以进行传递。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/ruby

require "rubygems"

require "hproseclient"

client = HproseHttpClient.new("http://www.hprose.com/example/")

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

puts args

result = client.invoke("swapKeyAndValue", args, true)

puts args

上面程序的运行结果如下：

Wed3Sun7Thu4Mon1Tue2Sat6Fri5

5Fri6Sat1Mon7Sun2Tue3Wed4Thu

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Hash或数组时，该Hash或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/ruby

require "rubygems"

require "hproseclient"

client = HproseHttpClient.new("http://www.hprose.com/example/")

client.sum(1, 2, 3, 4, 5) { |result|

puts result

}

client.Sum(6.0, 7.0, 8.0) { |result, args|

puts result

puts args

}

client.getUserList { |users|

users.each { |user|

puts user

puts "name: " + user.name

puts "age: " + user.age.to\_s

puts "birthday: " + user.birthday.to\_s

puts "sex: " + user.sex.to\_s

puts "married: " + user.married.to\_s

}

}

gets

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。

回调方法由语句块构成，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。

### 引用参数传递

跟同步方式相同，在异步方式下，只能通过invoke方式进行引用参数传递的远程调用。例如：

#!/usr/bin/ruby

require "rubygems"

require "hproseclient"

client = HproseHttpClient.new("http://www.hprose.com/example/")

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

puts args

result = client.invoke("swapKeyAndValue", args, true) { |result, args|

puts args

}

gets

这个运行结果为：

Wed3Sun7Thu4Mon1Tue2Sat6Fri5

5Fri6Sat1Mon7Sun2Tue3Wed4Thu

同样，异步调用也支持通过代理对象进行远程调用，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 Ruby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begin...rescue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/ruby

require "rubygems"

require "hproseclient"

client = HproseHttpClient.new("http://www.hprose.com/example/")

begin

client.notexisted

rescue ::Exception => e

puts e

end

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 Ruby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/ruby

require "rubygems"

require "hproseclient"

client = HproseHttpClient.new("http://www.hprose.com/example/")

client.onerror = lambda { |name, error|

puts name

puts error

}

client.notexisted {}

gets

上面这个程序中，我们通过lambda表达式定义了onerror事件，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：onrrror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 Ruby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proxy属性来设置http代理服务器的地址和端口，它是一个只写属性。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，只需要设置header属性就可以啦，header是一个Hash类型的属性，虽然它本身是只读的，但是你可以设置它的元素。key表示标头名，value表示标头值，key和value都是字符串型。

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header属性来设置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keepalive\_timeout。通过keepalive设置为True时，表示开启持久连接特征。keepalive\_timeout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RawWithEndTag。

### Serialized模式

Serialized模式下，结果以序列化模式返回，在Ruby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。

### 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。