



HPROSE

用户手册

1.2

(Java 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
升级.....	3
联系我们.....	4
报告漏洞与需求.....	4
第一章 快速入门.....	5
本章提要.....	5
安装 Hprose for Java.....	6
安装方法	6
创建 Hprose 的 Hello 服务器	6
创建 Hprose 的 Hello 客户端	10
通过 invoke 方法动态调用	10
通过接口方式调用	12
第二章 类型映射.....	14
本章提要.....	14
基本类型.....	15
值类型.....	15
引用类型	15
基本类型的映射.....	16
序列化类型映射	16
反序列化默认类型映射	16
反序列化有效类型映射	17
容器类型.....	19
列表类型	19
序列化类型映射	19
反序列化类型映射	19
字典类型	19
序列化类型映射	19
反序列化类型映射	19
对象类型.....	19

通过 ClassManager 来注册自定义类型	19
第三章 服务器.....	21
本章提要	21
直接使用 HproseServlet 发布服务	22
发布实例方法	22
注册自定义类型	28
隐藏发布列表	28
调试开关	29
对象序列化模式	29
P3P 开关	30
跨域开关	30
服务器事件	31
事件配置	31
onBeforeInvoke 事件	31
onAfterInvoke 事件	31
onSendHeader 事件	31
onSendError 事件	32
存取环境上下文	32
发布静态方法	32
自己编写 Servlet 发布 Hprose 服务	33
扩展 HproseServlet	33
HproseMethods 类型	33
setGlobalMethods 方法	34
使用 HproseHttpService 来构建 Servlet	35
按请求发布方法	35
按会话发布方法	36
自己编写 JSP 发布 Hprose 服务	37
第四章 客户端.....	39
本章提要	39
同步调用	40
通过 invoke 方法进行同步调用	40
带名称空间（别名前缀）方法	40
可变的参数和结果类型	40
引用参数传递	41
自定义类型的传输	43
通过代理接口进行同步调用	44
接口定义	44
带名称空间（别名前缀）方法	45
可变的参数和结果类型	45
泛型参数和结果	46
自定义类型	47
异步调用	48
通过 invoke 方法进行异步调用	48

通过代理接口进行异步调用	49
异常处理.....	53
同步调用异常处理	53
异步调用异常处理	53
超时设置.....	53
HTTP 参数设置	53
代理服务器	53
持久连接	54
HTTP 标头	54
第五章 其它平台.....	55
本章提要.....	55
Hprose for GAE	56
Hprose for Android	56
Hprose for JavaME.....	56
CDC 环境.....	56
CLDC 1.0 环境.....	56
CLDC 1.1 环境.....	57
附录 API 参考	58
本章提要.....	58
hprose.common 包	59
HproseCallback 接口	59
handler 方法	59
HproseErrorEvent 接口.....	59
handler 方法	59
HproseMethod 类	59
method 字段	59
obj 字段.....	59
paramTypes 字段.....	59
HproseMethods 类.....	60
构造方法	60
addInstanceMethods 方法	60
addMethod 方法	60
addMethods 方法.....	62
addMissingMethod 方法.....	62
addStaticMethods 方法.....	63
get 方法.....	63
getAllNames 方法	63
getCount 方法	63
HproseException 异常	64
hprose.io 包	64
HproseMode 类	64
FieldMode 静态字段	64
PropertyMode 静态字段.....	64

HproseTags 类	64
TagArgument 静态字段	64
TagBytes 静态字段	64
TagCall 静态字段	64
TagClass 静态字段	65
TagClosebrace 静态字段	65
TagDate 静态字段	65
TagDouble 静态字段	65
TagEmpty 静态字段	65
TagEnd 静态字段	65
TagError 静态字段	65
TagFalse 静态字段	65
TagFunctions 静态字段	66
TagInfinity 静态字段	66
TagInteger 静态字段	66
TagList 静态字段	66
TagLong 静态字段	66
TagMap 静态字段	66
TagNaN 静态字段	66
TagNeg 静态字段	66
TagNull 静态字段	67
TagObject 静态字段	67
TagOpenbrace 静态字段	67
TagPoint 静态字段	67
TagPos 静态字段	67
TagQuote 静态字段	67
TagRef 静态字段	67
TagResult 静态字段	67
TagSemicolon 静态字段	68
TagString 静态字段	68
TagTime 静态字段	68
TagTrue 静态字段	68
TagUTC 静态字段	68
HproseReader 类	68
stream 字段	68
构造方法	68
checkTag 方法	69
checkTags 方法	69
readArray 方法	69
readBigInteger 方法	69
readBoolean 方法	69
readBytes 方法	69
readDate 方法	70
readDateTime 方法	70

readDouble 方法.....	70
readEmpty 方法.....	71
readEnum 方法.....	71
readInfinity 方法.....	71
readInteger 方法	71
readList 方法	71
readLong 方法.....	72
readMap 方法.....	72
readNaN 方法.....	72
readNull 方法	73
readObject 方法	73
readString 方法	73
readTime 方法.....	74
readUntil 方法	74
unserialize 方法	74
HproseWriter 类	74
stream 字段	74
构造方法	75
serialize 方法.....	75
writeArray 方法	75
writeBoolean 方法.....	77
writeBytes 方法	77
writeCollection 方法	78
writeDate 方法	78
writeDouble 方法.....	79
writeEmpty 方法.....	79
writeEnum 方法.....	79
writeInfinity 方法.....	79
writeInteger 方法.....	79
writeList 方法	79
writeLong 方法.....	80
writeMap 方法	80
writeNaN 方法.....	80
writeNull 方法.....	80
writeObject 方法.....	80
writeString 方法.....	80
writeTime 方法	81
HproseFormatter 类.....	81
serialize 方法.....	81
unserialize 方法	82
hprose.client 包	83
HproseClient 类	83
onError 事件	83
invoke 方法	83

useService 方法	84
HproseHttpClient 类	85
构造方法	85
getHeader 方法	85
setHeader 方法	85
isKeepAlive 方法	85
setKeepAlive 方法	85
getKeepAliveTimeout 方法	86
setKeepAliveTimeout 方法	86
getProxyHost 方法	86
getProxyPass 方法	86
getProxyPort 方法	86
getProxyUser 方法	86
setProxyHost 方法	86
setProxyPass 方法	86
setProxyPort 方法	87
setProxyUser 方法	87
useService 方法	87
hprose.server 包	87
HproseServiceEvent 接口	87
onAfterInvoke 方法	87
onBeforeInvoke 方法	87
onSendError 方法	88
onSendHeader 方法	88
HttpContext 类	88
构造方法	88
getApplication 方法	88
getConfig 方法	88
getRequest 方法	88
getResponse 方法	88
getSession 方法	89
getSession 方法	89
HproseHttpMethods 类	89
HproseService 类	89
add 方法	89
addMissingMethod 方法	92
getEvent 方法	92
setEvent 方法	92
getGlobalMethods 方法	92
setGlobalMethods 方法	93
getMode 方法	93
setMode 方法	93
isDebugEnabled 方法	93
setDebugEnabled 方法	93

HproseHttpService 类.....	93
getCurrentContext 方法.....	93
handle 方法.....	93
isEnabled 方法.....	94
setEnabled 方法.....	94
isP3pEnabled 方法	94
setP3pEnabled 方法	94
HproseServlet 类	94

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 C++, .NET, Java, Delphi, Objective-C, ActionScript, JavaScript, ASP, PHP, Python, Ruby, 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, 分布式的电信级应用系统。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 的商业化版本, 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 是商业开源的, 您可以在购买相应的授权以后即可获得源代码的阅读或修改权, 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。

升级

每 1-3 个月我们就会推出一个小版本更新, 所有拥有升级许可证的正版用户都可以免费升级。每半年到一年我们会发布一个 Hprose 的全新版本。所有正版用户都可以凭借授权许可证享受优惠升级。

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	潍坊高远速通软件有限公司
公司地址	潍坊市高新区玉清东街以北中心次干道以西高新大厦（邮编 261061）
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	+86-0536-2999015（周一至周五，北京时间早上 9 点到下午 5 点）

报告漏洞与需求

如果您在 Hprose 中发现了漏洞或需要一些新的功能，请告知我们。最好的方法是通过我们公司网站的反馈系统进行反馈。我们的反馈系统专门用于收集所有有用的信息，它能帮助我们捕获缺陷和了解用户对新功能的需求。您每报告一个缺陷或功能需求后，都将会相应的收到一封电子邮件，其中含有跟踪号码，通过它，您可以及时了解报告的处理状态。当我们完成对报告的处理，我们会发送电子邮件告知您处理结果及原因（例如：缺陷将在下次发布中被解决、新特性将在下次发布时加入进来这并不是缺陷等等）。

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 Java 进行分布式程序开发的全过程。

本章提要

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

安装 Hprose for Java

Hprose for Java 分为 javaSE/javaEE 版本和 javaME 版本。

Hprose for JavaSE/JavaEE 版本支持 JDK 1.2 及其更高版本。

Hprose for JavaSE/JavaEE 版本支持 Google 云计算平台 GAE (Google App Engine) for Java。

Hprose for JavaSE 版本支持 Android 应用开发。

Hprose for JavaME 版本支持 CDC, CLDC 的所有版本。

Hprose for JavaME 的 CLDC 1.1 版本还提供了带有部分 JavaSE 功能的扩展版本。:

Hprose for JavaEE 版本支持 Apache Tomcat、Glassfish、Web Logic、Web Sphere 或其它任何一款可以运行 Java Servlet 的应用服务器。

安装方法

直接将 Hprose.jar 或 HproseClient.jar 放入开发环境的运行库目录下，并添加到运行库路径中即可。或者直接将源代码放入开发环境的源代码目录下即可。如果您使用的是 Hprose.jar 或者是包含了服务器部分的全部源代码，请将 Servlet 规范的运行库也添加到运行库路径中。

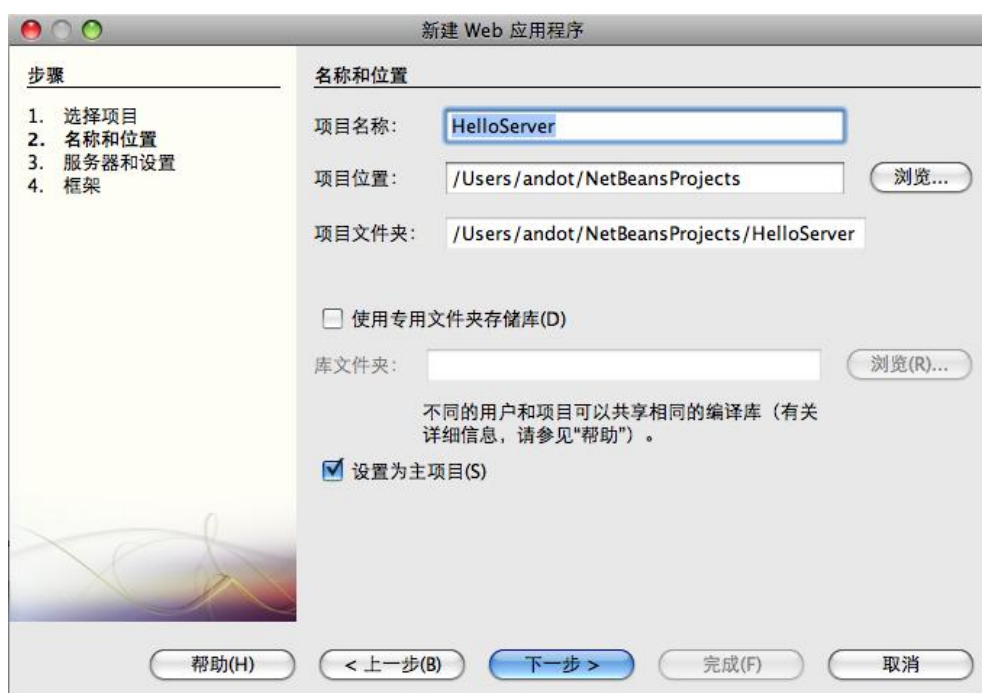
创建 Hprose 的 Hello 服务器

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

首先启动 Netbeans 开发环境，打开菜单的“文件→新建项目”，选择“Java Web→Web 应用程序”：



然后选择下一步，将项目名称改为 HelloServer：

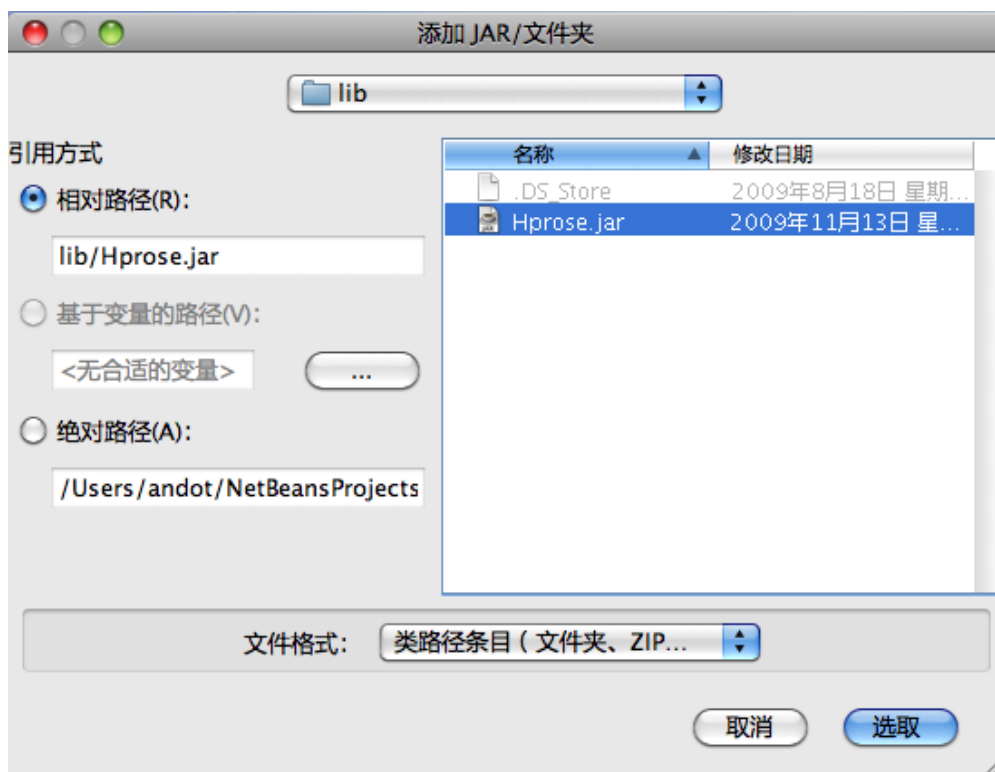


再点下一步，选择好服务器后，点完成。

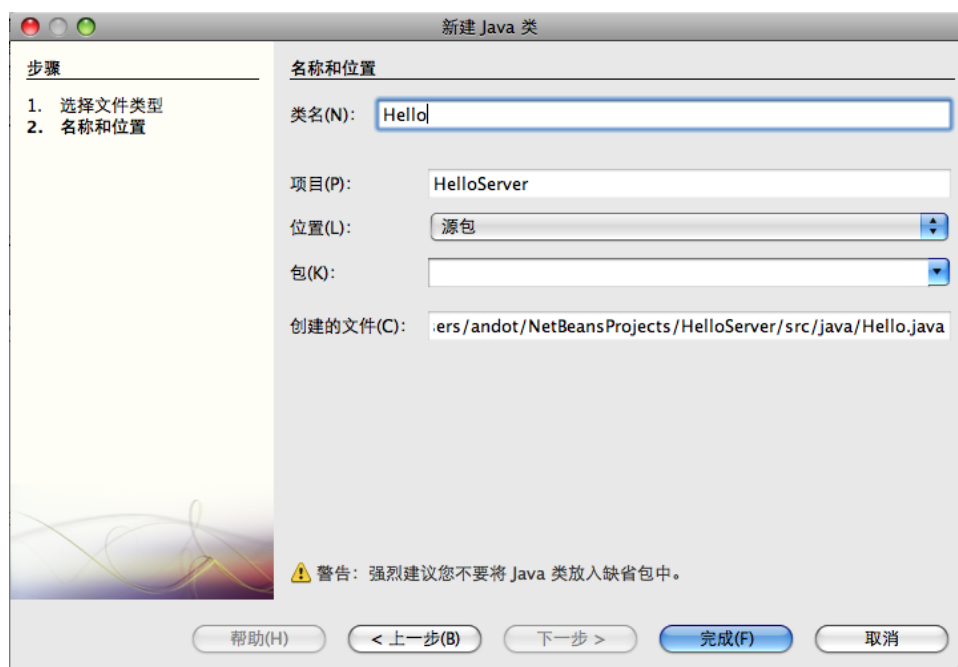


下面我们该把 Hprose.jar 放到该项目中来了。在 Netbeans 创建的 HelloServer 目录下，新建一个 lib 目录，然后将 Hprose.jar 复制到其中。

接下来在项目中，打开 HelloServer 下“库”的右键菜单，点击“添加 JAR/文件夹...”，选择 Hprose.jar，将其添加入库中。



之后在“源包”上打开右键菜单，选“新建→Java 类...”，将类名改为 Hello，包名任意，也可以不填写，只要跟后面配置 HproseServlet 时保持一致就可以啦。



Hello 类的代码如下：

```
public class Hello {
    public String sayHello(String name) {
        return "Hello " + name + "!";
    }
}
```


好了，下面我们只要再配置一下 Servlet，就可以发布这个服务啦。

配置方法如下：

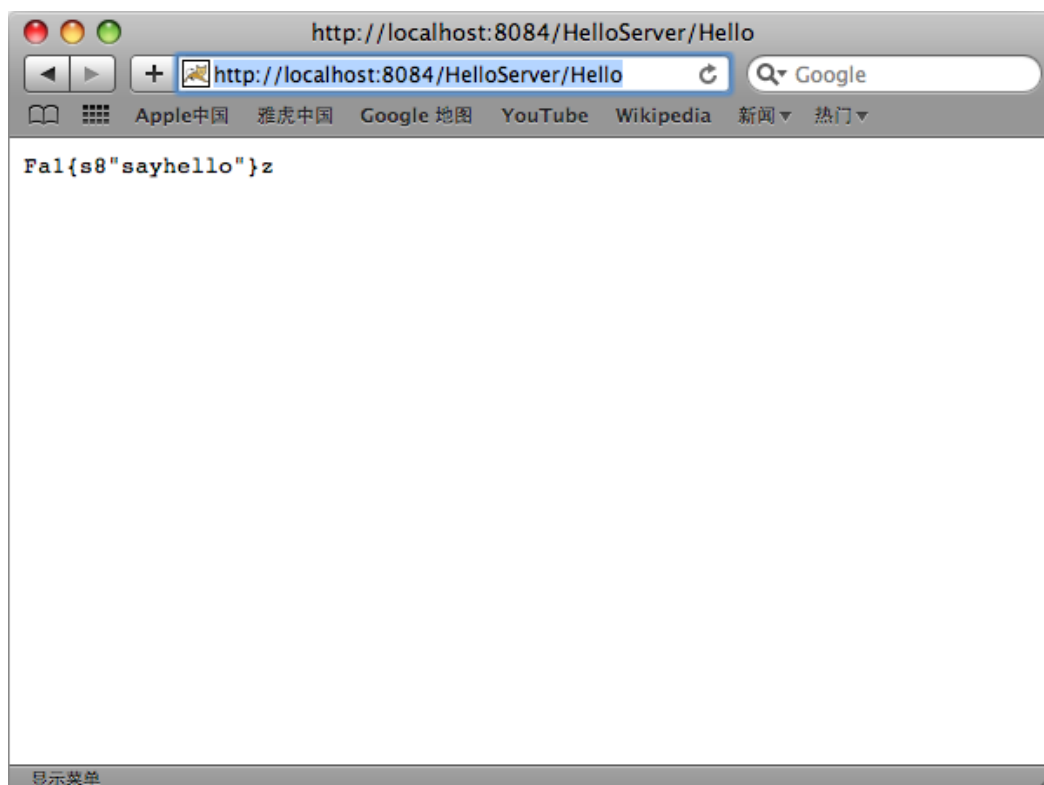
1. 打开 “HelloServer→配置文件→web.xml”。
2. 打开 “Servlet” 选单，点击 “添加 Servlet 元素...” 按钮，设置 Servlet 名称为 HelloServlet，Servlet 类为 hprose.server.HproseServlet，URL 模式为 /Hello。
3. 在初始化参数部分，点 “添加(A)...” 按钮，设置参数名为 class，参数值为 Hello，这里的 Hello 对应我们刚才创建的 Hello 类。如果您在创建 Hello 类时包含有包名，这里也应该写包含了包名的全名。

配置完成后如下图所示：



这样我们的服务器端就创建好了，是不是相当的简单啊？

好了我们来看看效果吧，打开 “HelloServer” 的右键菜单，选择 “运行”，之后您会看到浏览器窗口被打开，不过默认页面并不是我们的 Servlet 页面，没关系，我们直接在浏览器中更改一下 URL，改为：
http://localhost:8084/HelloServer/Hello，然后回车，如果看到如下页面就表示我们的服务发布成功啦。



接下来我们来看一下客户端如何创建吧。

创建 Hprose 的 Hello 客户端

客户端我们以 Java 控制台程序为例，开发环境仍然为 Netbeans。

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

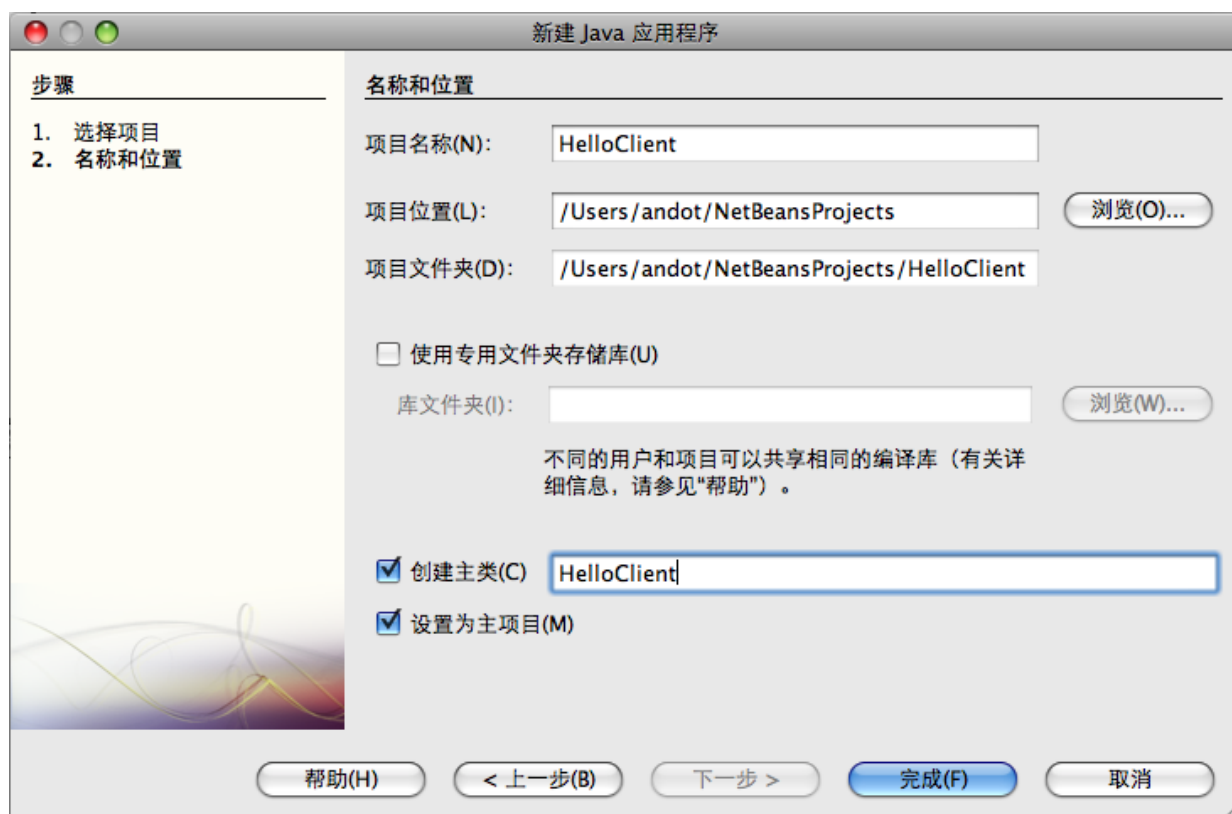
通过 invoke 方法动态调用

首先我们先来看看如何使用 invoke 方法来动态调用服务。

在 Netbeans 中， 打开菜单的“文件→新建项目”，选择“Java→Java 应用程序”。

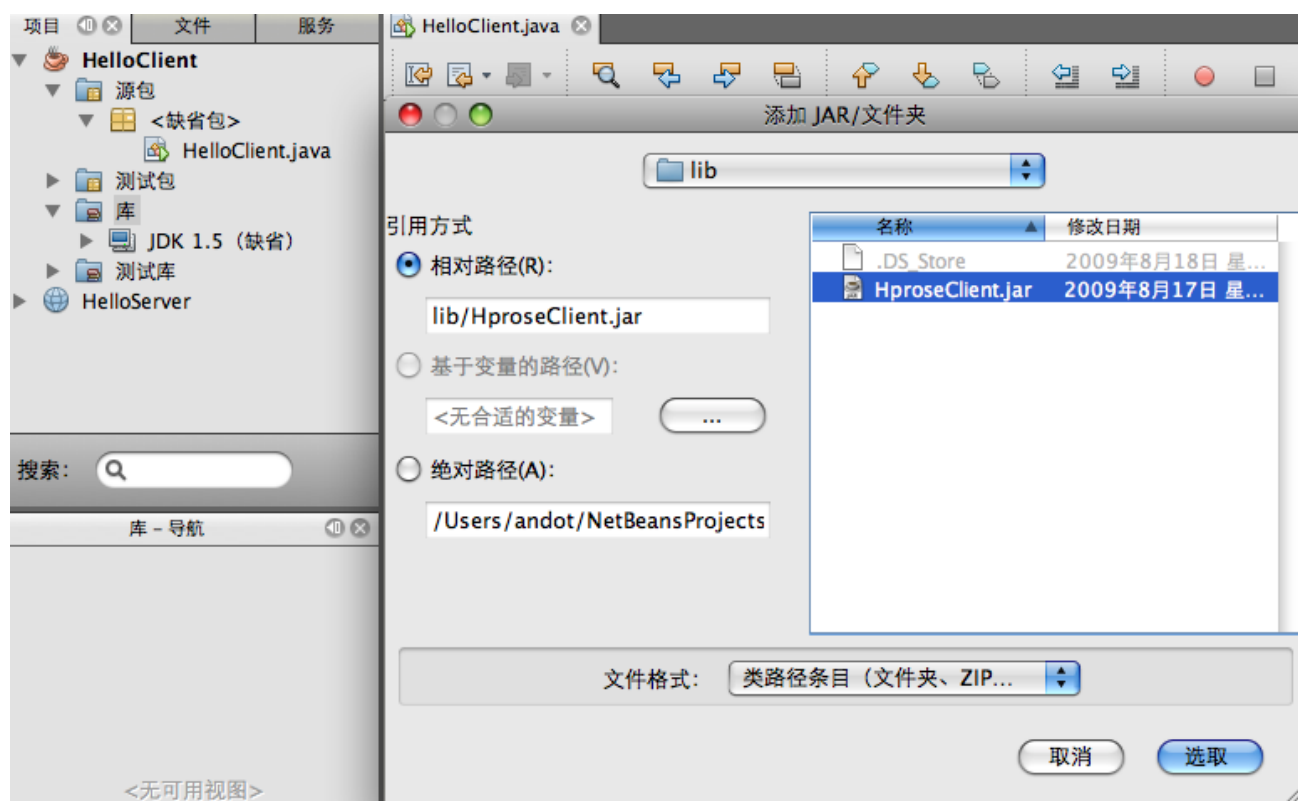


然后选择下一步，将项目名称改为 HelloClient，并将主类名也改为 HelloClient，然后点击完成即可。



下面我们该把 HproseClient.jar 放到该项目中来了。在 Netbeans 创建的 HelloClient 目录下，新建一个 lib 目录，然后将 HproseClient.jar 复制到其中。

接下来在项目中，打开 HelloClient 下“库”的右键菜单，点击“添加 JAR/文件夹...”，选择 HproseClient.jar，将其添加入库中。



接下来开始编写 HelloClient 类的代码：

```
import hprose.client.HproseHttpClient;
import java.io.IOException;
public class HelloClient {
    public static void main(String[] args) throws IOException {
        HproseHttpClient client = new HproseHttpClient();
        client.useService("http://localhost:8084/HelloServer/Hello");
        String result = (String) client.invoke("sayHello", new Object[] { "Hprose" });
        System.out.println(result);
    }
}
```

最后我们来运行一下看看结果吧，如果没有操作错误的话，您应该可以看到如下的输出结果：

```
Hello Hprose!
```

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

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

通过接口方式调用

我们用与上面同样的方式来创建项目 HelloClient2，然后打开 HelloClient2.java，编辑其代码如下：

```
import hprose.client.HproseHttpClient;
import java.io.IOException;
interface IHello {
    String sayHello(String name);
}
public class HelloClient2 {
    public static void main(String[] args) throws IOException {
        HproseHttpClient client = new HproseHttpClient();
        client.useService("http://localhost:8084/HelloServer/Hello");
        IHello hello = (IHello) client.useService(IHello.class);
        String result = hello.sayHello("Hprose");
        System.out.println(result);
    }
}
```

现在代码虽然多了，但是在调用时，却方便了不少。

如果您曾经用过 RMI 或者其它的远程调用工具的话，您可能会惊讶的发现，这里的 IHello 接口在服务器端并没有实现，只是有相同的方法，但在客户端仍然可以直接通过接口进行调用。Hprose 就是这样灵活，

您甚至可以定义不同于服务器实现的客户端接口，只要参数和结果类型是相容或者可以转换的类型，就可以正常的进行调用。所以在 Hprose 中，您不但可以通过接口方式来调用 Java 服务器提供的服务，同样可以调用非 Java 服务器提供的服务。



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

第二章 类型映射

类型映射是 Hprose 的基础，正是因为 Hprose 设计有良好的类型映射机制，才使得多语言互通得以实现。本章将对 Hprose for Java 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，仅支持单字节字符。
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

Java 类型与 Hprose 类型的映射关系不是——对应的。在序列化过程中可能会有多种 Java 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

Java 类型	Hprose 类型
byte, short, int, Byte, Short, Integer, Enum	整型
long, Long, BigInteger	长整型
float, double, Float, Double	浮点型
Float.NaN, Double.NaN	非数
Float.POSITIVE_INFINITY, Double.POSITIVE_INFINITY	正无穷大
Float.NEGATIVE_INFINITY, Double.NEGATIVE_INFINITY	负无穷大
true, Boolean.TRUE	布尔真
false, Boolean.FALSE	布尔假
null	空
byte[]	二进制型（或空串）
char, Character, char[], 单字符 String, 单字符 StringBuffer	字符
char[], String, StringBuffer, BigDecimal	字符串型（或空串）
java.sql.Date	日期型
java.sql.Time	时间型
java.util.Date, java.util.Calendar	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	Java 类型
整型	Integer
长整型	BigInteger
浮点型	Double

Hprose 类型	Java 类型
非数	Double.NaN
正无穷大	Double.POSITIVE_INFINITY
负无穷大	Double.NEGATIVE_INFINITY
布尔真	Boolean.TRUE
布尔假	Boolean.FALSE
空	null
空串	""
二进制型	byte[]
字符	Character
字符串型	String
日期型	java.util.Calendar
时间型	java.util.Calendar
日期时间型	java.util.Calendar

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	Java 类型（安全）	Java 类型（非安全）
整型	Integer, Long, Float, Double, BigInteger, BigDecimal, String, Character	Byte, Short, Boolean, Enum, java.util.Date, java.util.Calendar, java.sql.Date, java.sql.Time, java.sql.Timestamp
长整型	BigInteger, BigDecimal, String	Byte, Short, Integer, Long, Float, Double, Boolean, Enum, java.util.Date, java.util.Calendar, java.sql.Date, java.sql.Time, java.sql.Timestamp, Character
浮点型	Double, BigDecimal, String	Byte, Short, Integer, Long, Float, BigInteger, Boolean, Enum, java.util.Date, java.util.Calendar, java.sql.Date, java.sql.Time, java.sql.Timestamp, Character
非数	Float.NaN, Double.NaN, String	无

Hprose 类型	Java 类型 (安全)	Java 类型 (非安全)
正无穷大	Float.POSITIVE_INFINITY, Double.POSITIVE_INFINITY, String	无
负无穷大	Float.NEGATIVE_INFINITY, Double.NEGATIVE_INFINITY, String	无
布尔真	Boolean.TRUE, Byte, Short, Integer, Long, Float, Double, BigInteger, BigDecimal, String, Character	无
布尔假	Boolean.FALSE, Byte, Short, Integer, Long, Float, Double, BigInteger, BigDecimal, String, Character	无
空	任意类型	无
空串	byte[0], char[0], "", StringBuffer	无
二进制型	byte[]	String
字符	String, char[]	Byte, Short, Integer, Long, Float, Double, BigInteger, BigDecimal, Enum, java.util.Calendar, java.util.GregorianCalendar, java.util.Date, java.sql.Date, java.sql.Timestamp
字符串型	String, StringBuffer, char[], byte[]	Byte, Short, Integer, Long, Float, Double, BigInteger, BigDecimal, Character
日期型	java.util.Calendar, java.util.GregorianCalendar, java.util.Date, java.sql.Date, java.sql.Timestamp, Long	java.sql.Time
时间型	java.util.Calendar, java.util.GregorianCalendar, java.util.Date, java.sql.Time, java.sql.Timestamp, Long	java.sql.Date
日期时间型	java.util.Calendar, java.util.GregorianCalendar, java.util.Date, java.sql.Timestamp, Long	java.sql.Date, java.sql.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。下面我们来分别介绍它们与 Java 类型的映射关系。

列表类型

序列化类型映射

除 `byte[]`, `char[]` 以外的所有其它数组类型 and 所有实现了 `Collection` 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 Java 的 `ArrayList` 类型。有效类型为：

- 所有数组类型
- 所有实现了 `Collection` 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 `Map` 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 Java 的 `HashMap` 类型。有效类型为：

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

对象类型

Java 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：

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

通过 ClassManager 来注册自定义类型

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

通过 `hprose.io.ClassManager` 的 `register` 方法就可以轻松实现这个需求。例如您有一个 `my.package.User` 的类，希望传递给 PHP，但是在 PHP 中与之对应的类是 `User`，那么可以这样做：

```
ClassManager.register(my.package.User, 'User');
```



Hprose 所支持的 Java 自定义可序列化对象类型仅是 Java 可序列化类型的一个子集。Hprose 对 Java 对象类型的序列化分为两种，一种是按属性序列化，一种是按字段序列化。这两种方式下都需要在定义类时实现 `java.io.Serializable` 接口。按属性序列化时，只有 `get(is)` 和 `set` 存取方法都定义的属性会被序列化。按字段序列化时，没有标明 `transient` 的字段都会被序列化。

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

本章提要

- 直接使用 HproseServlet 发布服务
- 自己编写 Servlet 发布 Hprose 服务
- 自己编写 JSP 发布 Hprose 服务

直接使用 HproseServlet 发布服务

因为在快速入门里面我们已经详细通过图解方式介绍了通过直接使用 HproseServlet 发布服务的整个过程，这里就不再通过图解方式介绍了，下面我们更多关注的是代码部分。

通过 HproseServlet 发布服务很简单，直接通过配置方式就可以，如果要发布的类是现成的，您不需要编写一行代码就可以完成发布。

发布的方法可以是静态方法，也可以是实例方法。但必须是 public 方法。您还可以同时发布多个类中的方法。下面先介绍如何发布类中的实例方法。

发布实例方法

在前面快速入门一章里，您已经看过传输简单类型的例子了。所以下面将以传输复杂类型为例，来介绍发布多个类中的实例方法，其中还包括了继承、覆盖、重载方法发布等内容。

先来看第一个类：

```
package hprose.exam;
import java.util.HashMap;
import java.util.Map;
public class Exam1 {
    protected String id;
    public Exam1() {
        id = "Exam1";
    }
    public String getID() {
        return id;
    }
    public int sum(int[] nums) {
        int sum = 0;
        for (int i = 0, n = nums.length; i < n; i++) {
            sum += nums[i];
        }
        return sum;
    }
    public Map<String, String> swapKeyAndValue(Map<String, String> strmap) {
        Map.Entry[] entrys = strmap.entrySet().toArray(new Map.Entry[strmap.size()]);
        strmap.clear();
        for (Map.Entry<String, String> entry: entrys) {
            strmap.put(entry.getValue(), entry.getKey());
        }
        return strmap;
    }
}
```

上面类中，有 3 个可以发布的方法 `getID`，`sum` 和 `swapKeyAndValue`。

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

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

`swapKeyAndValue` 方法的参数是一个存储字符串键值对的 `Map`，其结果为键值对调后的 `Map`。虽然您在这里会发现可以用泛型的 `Map`，但实际上 Java 的泛型只能在编译期进行检查，在运行时并不能得到泛型信息，因此，实际上并不能保证远程调用时接收到的 `Map` 中的键值对一定能转换为泛型中所标注的类型，当传输的类型与标注的类型不同时，就会发生运行时错误。因此当您标注泛型时，对于基本类型只能标注为反序列化默认类型映射中的类型，对于容器类型您可以标注为 `ArrayList`、`List`、`HashMap`、`Map` 或 `Collection`，当然还可以标注为自定义可序列化类型。

例如，下面这个方法在本地调用时只要参数类型一致，肯定是没有问题的。但是在使用 `Hprose` 进行远程调用时，即使客户端指定的参数是一致的，也仍然会返回错误。

```
public Map<Short, Short> swapKeyAndValue(Map<Short, Short> strmap) {
    Map.Entry[] entrys = strmap.entrySet().toArray(new Map.Entry[strmap.size()]);
    strmap.clear();
    for (Map.Entry<Short, Short> entry: entrys) {
        strmap.put(entry.getValue(), entry.getKey());
    }
    return strmap;
}
```

因为 Java 的 `Short` 类型在序列化时会转化为 `Hprose` 整型传输，而 `Hprose` 整型的反序列化默认类型映射为 Java 的 `Integer` 类型。`Integer` 类型和 `Short` 类型永远不会匹配，发生错误就成为了必然。因此，您应该避免在您发布的方法中使用上述的泛型定义。

另一种在使用泛型时常见的错误是，泛型参数为自定义可序列化类型，客户端和服务端当中泛型参数类型的名字相同，类型中结构定义也相同，但是客户端和服务端中的这个类型却属于两个不同的包。这种情况下，这两个类型会被判定为不同的类型（实际上也确实是不同的类型），这种情况下的调用也会发生错误。这个问题如果暂时看不明白也没关系，我们在后面也会举例说明。

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

```
package hprose.exam;
public enum Sex {
    Unknown, Male, Female, InterSex
}
```

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

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

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

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

下面我们继续回到正题上来。

Java 中当然不支持对枚举进行位运算,不过在其它语言(例如 C#)中枚举类型是可以进行位运算的,因此,合理的安排枚举类型的顺序有时候会方便与其它语言的交互。毕竟使用 Hprose 后,就可以在多语言之间进行互通了,因此我们在编程时应当考虑到这一点。

下面我们来看 User 类的定义吧。

```
package hprose.exam;
import java.sql.Date;
public class User implements java.io.Serializable {
    private String name;
    private Sex sex;
    private Date birthday;
    private int age;
    private boolean married;
    public User() {
    }
    public User(String name, Sex sex, Date birthday, int age, boolean married) {
        this.name = name;
        this.sex = sex;
        this.birthday = birthday;
        this.age = age;
        this.married = married;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public Sex getSex() {
        return sex;
    }
    public void setSex(Sex sex) {
```



```

        this.sex = sex;
    }
    public Date getBirthday() {
        return birthday;
    }
    public void setBirthday(Date birthday) {
        this.birthday = birthday;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
    public boolean isMarried() {
        return married;
    }
    public void setMarried(boolean married) {
        this.married = married;
    }
}

```

该类的定义有以下几点需要注意：

1. 必须要实现 `java.io.Serializable` 接口。
2. 必须要有一个无参构造方法。
3. 字段定义为私有，属性定义为共有。
4. 字段名和属性名应一一对应。

按照这种方式定义的类，既可以按照属性序列化传输，也可以按照字段序列化传输。

下面我们来看第二个要发布的类如何定义：

```

package hprose.exam;
import java.sql.Date;
import java.util.ArrayList;
import java.util.List;
public class Exam2 extends Exam1 {
    public Exam2() {
        id = "Exam2";
    }
    public List<User> getUserList() {
        ArrayList<User> userList = new ArrayList<User>();
        userList.add(new User("Amy", Sex.Female, Date.valueOf("1983-12-03"), 26, true));
        userList.add(new User("Bob", Sex.Male, Date.valueOf("1989-06-12"), 20, false));
        userList.add(new User("Chris", Sex.Unknown, Date.valueOf("1980-03-08"), 29, true));
    }
}

```

```

        userList.add(new User("Alex", Sex.InterSex, Date.valueOf("1992-06-14"), 17, false));
        return userList;
    }
}

```

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

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

好了，我现在来看如何发布它们吧。我们先来看最简单的发布一个类：

1. 打开“HproseExamServer→配置文件→web.xml”。

2. 打开“Servlet”选单，点击“添加 Servlet 元素...”按钮，设置 Servlet 名称为 HproseExamServlet，Servlet 类为 hprose.server.HproseServlet，URL 模式为/Methods。

3. 在初始化参数部分，点“添加(A)...”按钮，设置参数名为 class，参数值为 hprose.exam.Exam2，这里的 hprose.exam.Exam2 对应上面创建的 hprose.exam.Exam2 类。

配置后的 web.xml 内容如下：

```

<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://java.sun.com/xml/ns/
javaee http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
    <servlet>
        <servlet-name>HproseExamServlet</servlet-name>
        <servlet-class>hprose.server.HproseServlet</servlet-class>
        <init-param>
            <param-name>class</param-name>
            <param-value>hprose.exam.Exam2</param-value>
        </init-param>
    </servlet>
    <servlet-mapping>
        <servlet-name>HproseExamServlet</servlet-name>
        <url-pattern>/Methods</url-pattern>
    </servlet-mapping>
    <session-config>
        <session-timeout>
            30
        </session-timeout>
    </session-config>
</web-app>

```

现在就配置好了对 hprose.exam.Exam2 的发布，但是运行结果可能会出乎您的预料，当您打开浏览器浏览服务页面时，您会发现只有 getUserList 这一个方法被发布了，从 Exam1 中继承来的方法都没有被发布，这是正确的吗？

是的，这是正确的，这是因为所有的类都是从祖先类 Object 继承下来的，如果将继承来的 public 方法

都发布的话，势必会将从 Object 继承来的方法也一起发布，这肯定不是用户所期望的结果。另一个原因，用户定义的类的层次可能比较深，而其定义的基类上的方法可能并不想被一起发布。因此，默认情况下，只发布直接在该类中声明的 public 方法，而不会发布继承来的方法。

那如果想发布继承来的方法可以吗？当然可以。您可以这样发布。将上面配置文件中：

```
<init-param>
  <param-name>class</param-name>
  <param-value>hprose.exam.Exam2</param-value>
</init-param>
```

部分改为：

```
<init-param>
  <param-name>class</param-name>
  <param-value>hprose.exam.Exam2|hprose.exam.Exam1</param-value>
</init-param>
```

其中，“|”用来分隔参数，第一参数为创建对象的类名，第二个参数为第一个参数的祖先类的类名。

经过上面的配置修改后，您就会发现 Exam2 上的声明的方法和继承自 Exam1 的方法都被发布了。

那如果还要发布一个 Exam1 对象上的方法怎么办呢？您可以通过“,”来分隔要发布的不同的对象的类，但是您会发现不能简单的将配置修改为：

```
<init-param>
  <param-name>class</param-name>
  <param-value>hprose.exam.Exam1, hprose.exam.Exam2|hprose.exam.Exam1</param-value>
</init-param>
```

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

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

正确的修改如下：

```
<init-param>
  <param-name>class</param-name>
  <param-value>hprose.exam.Exam1|ex1,hprose.exam.Exam2|hprose.exam.Exam1|ex2</param-value>
</init-param>
```

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

从上面的配置您还可以发现，第二个参数可以空缺，空缺时表示发布的方法就是在创建对象的类上直接声明的方法。

经过上面的配置后，运行您的服务器，并打开浏览器浏览发布页，应该可以看到如下的发布列表：

```
Fa7{s7"ex1_sum"s19"ex1_swapkeyandvalue"s15"ex2_getuserlist"s7"ex2_sum"s9"ex1_getid"s9"ex2_getid"s19"ex2_swapkeyandvalue"}z
```

这就表示服务发布成功了。

注册自定义类型

在前一章中我们提到过使用 ClassManager 类来注册我们的自定义类型，以便于和其它语言交互。但是如果采用配置 Servlet 的方式发布服务，其中发布的服务又包含了需要注册的自定义类型，这时该怎么办呢？

在 Hprose 1.2 for Java 之前的版本中，如果希望通过配置 Servlet 的方式发布这种带有需要注册自定义类型的服务，只能在该类型的静态块中调用 ClassManager 的 register 方法来注册自己，这样才可以保证当该类型被使用时，会以注册的类型名去查找它，但是这种方式是紧密耦合的，所以最好不要采用这种方式，更好的做法是通过自己编写 Servlet 的方式来发布这种服务。这样便不需要对已有的类型进行任何修改了。

Hprose 1.2 for Java 中特别考虑到了这种情况，因此提供了一种通过配置的方式来注册自定义类型的方法。例如，如果需要注册一个 hprose.exam.User 的类，注册的名称为 User 的话，那么您可以这样配置：

```
<init-param>
  <param-name>type</param-name>
  <param-value>hprose.exam.User|User</param-value>
</init-param>
```

如果有多个类需要注册，则用逗号分隔。例如：除了注册上面的 User 之外，还需要注册一个 Person 的类的话，那么可以这样配置：

```
<init-param>
  <param-name>type</param-name>
  <param-value>hprose.exam.User|User,hprose.exam.Person|Person</param-value>
</init-param>
```

有了这种方式之后，您就再也不必为发布带有自定义类型的服务而犯愁了。

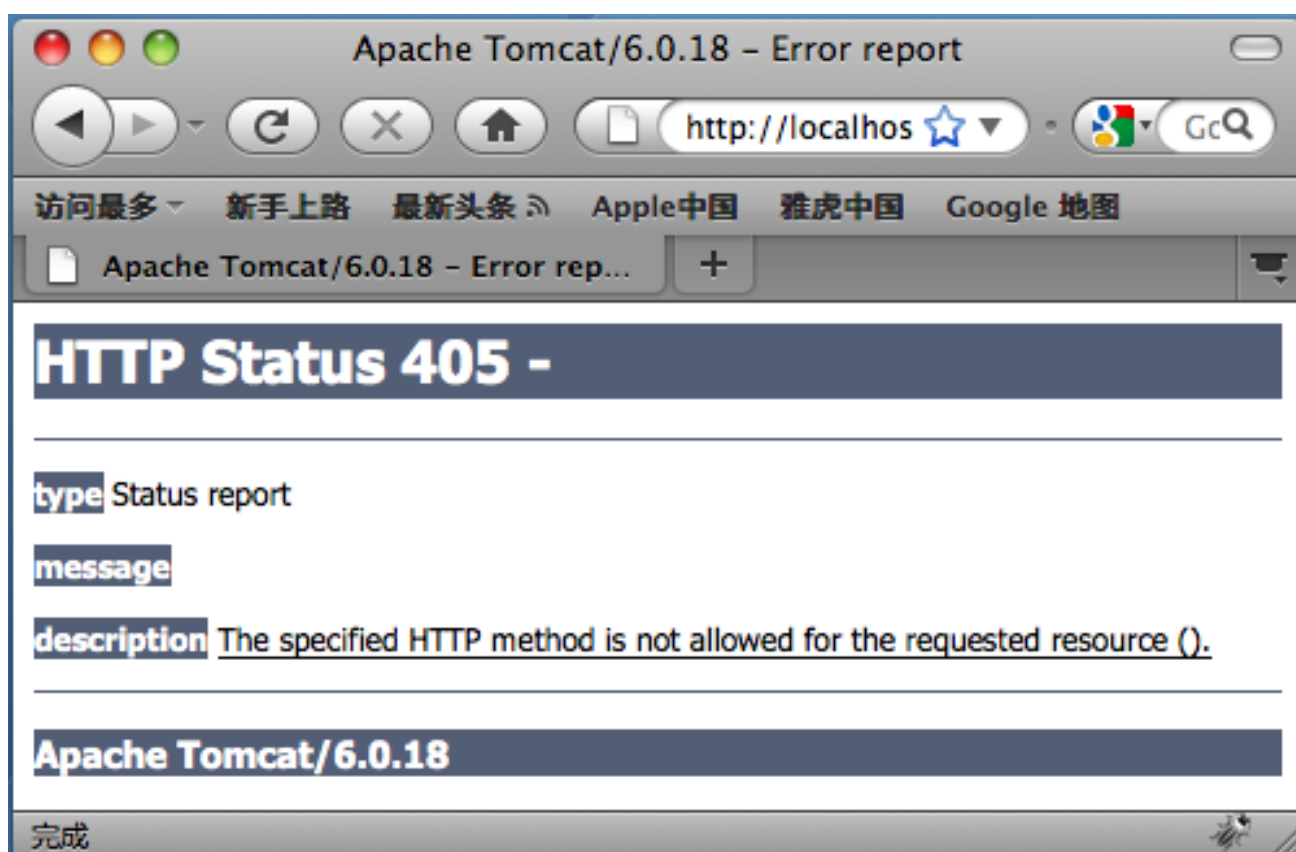
隐藏发布列表

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，只需要将初始化参数 get 设置为 false 即可，配置如下：

```
<init-param>
  <param-name>get</param-name>
  <param-value>>false</param-value>
</init-param>
```

之后再打开浏览器您可能会看到如下页面：



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。

配置方法与隐藏发布列表类似，只需要将初始化参数 debug 设置为 true 即可。

```
<init-param>
  <param-name>debug</param-name>
  <param-value>true</param-value>
</init-param>
```

对象序列化模式

前面我们发布的 getUserList 方法传递的是自定义可序列化对象列表。在介绍自定义可序列化对象上我们也多次提到有两种序列化模式：字段模式和属性模式。那么如何来配置使用哪种模式呢？

同样，通过设置初始化参数就可以完成配置，例如如果要设置成属性模式（默认是字段模式），只需要加入以下配置信息即可：

```
<init-param>
  <param-name>mode</param-name>

  <param-value>propertyMode</param-value>
</init-param>
```

设置为属性模式传输时，属性名首字母自动以小写表示，其它部分大小写不变，这样不但可以跟字段序列化模式统一，跟其它语言对象中的字段或属性命名也统一。

P3P 开关

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

```
<init-param>
  <param-name>p3p</param-name>
  <param-value>true</param-value>
</init-param>
```

跨域开关

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

```
<init-param>
  <param-name>crossDomain</param-name>
  <param-value>true</param-value>
</init-param>
```

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

另一种跨域方案同时适用于以上三种客户端，那就是通过设置跨域策略文件的方式。您只需要将 crossdomain.xml 文件放在网站发布的根目录上即可。

最后，Hprose 还提供了专门适用于 SilverLight 的跨域方案，那就是通过设置客户端访问策略文件的方式。做法跟 crossdomain.xml 类似，只需要将 clientaccesspolicy.xml 文件放在网站发布的根目录上即可。

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

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

服务器事件

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

Hprose 服务器提供了 4 个事件, 它们被定义为 HproseServiceEvent 接口的四个方法:

```
package hprose.server;

public interface HproseServiceEvent {
    void onBeforeInvoke(String name, Object[] args, boolean byRef);
    void onAfterInvoke(String name, Object[] args, boolean byRef, Object result);
    void onSendHeader();
    void onSendError(String error);
}
```

您只需要实现此接口就可以完成事件的定义啦。

事件配置

如果您已经定义好了事件类(这里我们假设事件类的类名为 hprose.exam.Event), 那么只需要在 Servlet 的初始化参数中配置以下参数即可:

```
<init-param>
    <param-name>event</param-name>
    <param-value>hprose.exam.Event</param-value>
</init-param>
```

那这个事件类该如何定义呢? 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, 其中 name 为客户端所调用的方法名, args 为方法的参数, byRef 表示是否是引用参数传递的调用。

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

onAfterInvoke 事件

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

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

onSendHeader 事件

当服务器返回响应头部时, onSendHeader 事件会被触发。

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

onSendError 事件

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

您可以在该事件中作日志记录，但该事件中不应再抛出任何异常。

存取环境上下文

在上面介绍的服务器事件或者服务器发布的方法中，您可能会需要存取环境上下文，例如 Request，Response，Session，Application，那么有什么方便的方法来获取它们吗？

Hprose 提供了 HttpContext 这个帮助类来表示在 http 环境下的上下文信息，通过 HproseHttpService 类的 getCurrentContext 静态方法，您可以获取到当前的环境上下文对应的 HttpContext 对象。然后再通过该对象上的 getRequest、getResponse、getSession、getConfig、getApplication 方法就可以获取到您需要的对象了。

HttpContext 帮助类可以在服务器事件中使用，也可以在服务器发布的方法中使用。另外，在服务器发布的方法中还有另外一种方式来获取环境上下文，那就是将方法的最后一个参数定义为相应环境上下文类型。

例如要获取 Session 对象，只需要将最后一个参数定义为 HttpSession 类型即可：

```
public String getSessionID(HttpSession session) {  
    return session.getId();  
}
```

客户端在调用该方法时，不需要代入任何参数，服务器会自动传递 session 参数给该方法。与上面方法等价的使用 HttpContext 帮助类实现的方法写法如下：

```
public String getSessionID() {  
    HttpContext context = HproseHttpService.getCurrentContext();  
    HttpSession session = context.getSession();  
    return session.getId();  
}
```

通过对比，您会发现直接通过参数传递方式的写法更简洁一些。通过参数方式可以传递 HttpContext 本身和它所包含的所有环境上下文类型，但是通过参数传递方式只能传递一个环境上下文参数。

发布静态方法

发布静态方法跟发布实例方法类似，只不过您在定义类时，需要把方法都定义成 static public 方法。之后在配置时，添加参数名为 staticClass 的初始化参数。staticClass 的值就是静态方法所在的类名，可以指定多个，使用“,”分隔。同样，也可以为静态方法的类指定名称空间（别名前缀），用“|”分隔。

假设我们有 hprose.exam.Exam3 和 hprose.exam.Exam4 两个类中的静态方法要发布，并且希望给 hprose.exam.Exam3 指定名称空间的话，可以这样配置：

```
<init-param>
```



```
<param-name>staticClass</param-name>
<param-value>hprose.exam.Exam3|ex3,hprose.exam.Exam4</param-value>

</init-param>
```

静态方法和实例方法的发布可以在同一个 Servlet 中配置。如果有同名方法，就是用名称空间（别名前缀）来区分。否则同名静态方法会覆盖同名实例方法的发布。

好了，直接使用 HproseServlet 发布服务到这里我们就全部介绍完了。如果您觉得它已经可以满足您的全部需求，那么您就可以直接跳过下面几节进入下一章了。如果它还不能完全满足您的需求，那么接下来的几节可能对您就非常重要了，下面几节将让您更加灵活的控制服务的发布。

自己编写 Servlet 发布 Hprose 服务

如果您对直接使用 HproseServlet 发布服务还有什么不满的话，当然可以自己编写一个 Servlet 来发布 Hprose 服务。而且您会发现这并不是什么难事，因为 Hprose 已经为您提供了很好的基础，您只需要对 HproseServlet 做一下扩展，或者直接使用 HproseHttpService 来构建自己的 Servlet 即可。

扩展 HproseServlet

我们先来介绍最简单的方式，那就是以 HproseServlet 为基类，通过创建子类的方式来扩展 HproseServlet。使用这种方式，您可以更灵活的控制方法的发布。

在前面直接使用 HproseServlet 发布服务时，您会发现我们只能以类为单位来发布方法，而不能单独发布类或对象上的某一个或几个方法，另外，如果我们想对某一个方法指定别名，而不是对整个对象或类上所有方法增加名称空间（别名前缀）的话，也是做不到的。

上面这些需求您都可以通过扩展 HproseServlet 来实现。

创建 HproseServlet 的子类非常容易，HproseServlet 提供了多个可以覆盖的方法，其中最重要是：setGlobalMethods。

该方法的参数类型是 HproseMethods 类型的,下面我们先来对 HproseMethods 类型做一下介绍。

HproseMethods 类型

HproseMethods 类型用于表示发布方法的集合。

它有以下几个方法：

1. addMethod
2. addMethods
3. addInstanceMethods
4. addStaticMethods
5. addMissingMethod

下面我们对这几个方法分别做一下介绍。

addMethod 方法

addMethod 用来控制单个方法的发布，它有十种重载形式。通过它可以发布任意对象上的任意 public

实例方法，或任意类上的任意 public 静态方法，并且可以为每个方法都指定别名。当您发布的方法具有相同个数参数的重载时，使用 `addMethod` 是唯一可行的方法。

addMethods 方法

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

addInstanceMethods 方法

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

addStaticMethods 方法

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

addMissingMethod 方法

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

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

用 `addMissingMethod` 发布的方法参数应为以下形式：

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

第一个参数表示客户端调用时指定的方法名，方法名在传入该方法时全部是小写的。

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

对于以上方法的每个重载形式的详细介绍请参见附录：Hprose for Java API 参考。

现在，您对 `HproseMethods` 的以上方法应该有一定程度的认识了，下面我们再结合 `HproseServlet` 的 `setGlobalMethods` 方法来看一下如何实际应用它们。

setGlobalMethods 方法

`setGlobalMethods` 方法的定义形式如下：

```
protected void setGlobalMethods(HproseMethods methods) {  
}
```

实际上它是在 `HproseServlet` 的 `init` 方法最后被调用的，其参数为全局发布方法的集合。当我们需要在

全局发布方法时，就可以覆盖这个方法来实现。

例如我们只想要发布一个 Exam2 对象上的 getID 方法，通过直接配置 HproseServlet 并不能得到我们想要的效果，那么通过覆盖 setGlobalMethods，我们就可以实现这个功能了。

下面看我们自己定义的 Servlet：

```
package hprose.exam;
import hprose.server.HproseServlet;
import hprose.common.HproseMethods;
public class MyHproseServlet extends HproseServlet {
    protected void setGlobalMethods(HproseMethods methods) {
        Exam2 exam2 = new Exam2();
        methods.addMethod("getID", exam2);
    }
}
```

是不是很简单，只需要 9 行代码，就完成我们自己的 HproseServlet 了。

接下来我们看看如何使用 HproseHttpService 来构建 Servlet。

使用 HproseHttpService 来构建 Servlet

通常我们是不需要使用 HproseHttpService 来构建 Servlet 的，因为通过配置和扩展 HproseServlet 已经几乎可以满足所有要求了。

但您仍然有理由来使用 HproseHttpService 构建 Servlet，例如：您可能希望发布服务能够与 Spring 集成；或者您可能不喜欢配置文件，而希望通过使用注解（annotation）方式来发布服务。当然，您还可能希望不在全局范围内发布方法，而是为每个请求或每个会话来创建对象并发布其上的方法。

在这种情况下，使用 HproseHttpService 来构建 Servlet 将是个好主意。

但是我们并不打算在这里介绍如何编写与 Spring 集成发布服务的 Servlet，也没有打算介绍如何编写一个使用注解（annotation）方式发布服务的 Servlet。因为这将花费不少篇幅，并且大部分代码脱离了我们重点介绍的内容。

我们这里主要来讲解在使用 HproseHttpService 来构建自己的 Servlet 时，需要注意的一些问题，以及如何为每个请求或每个会话来创建对象并发布其上的方法。

按请求发布方法

这里我们还是以发布 Exam2 上面的方法为例，来说明如何编写 Servlet：

```
package hprose.exam;
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import hprose.server.HttpContext;
import hprose.server.HproseHttpMethods;
```

```
import hprose.server.HproseHttpService;

public class MyHproseServlet2 extends HttpServlet {
    private HproseHttpService service = new HproseHttpService();

    protected void processRequest(HttpServletRequest request, HttpServletResponse
response)
        throws ServletException, IOException {
        Exam2 exam2 = new Exam2();
        HproseHttpMethods methods = new HproseHttpMethods();
        methods.addInstanceMethods(exam2);
        methods.addInstanceMethods(exam2, Exam1.class);
        service.handle(new HttpContext(request, response, this.getServletConfig(),this.getServletContext()),
            methods);
    }

    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        processRequest(request, response);
    }

    protected void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        processRequest(request, response);
    }

    public String getServletInfo() {
        return "Hprose Servlet 1.0";
    }
}
```

您完全可以把上面的例子作为模版来改写成您自己的 Servlet。

在这个例子中，关键的方法就是 processRequest 方法。

首先创建 HproseHttpMethods 类的对象，之后通过该对象上的 addInstanceMethods 方法发布 Exam2 对象上的方法，最后通过 HproseHttpService 对象的 handle 方法来处理请求。handle 方法的第一个参数是 HttpContext 类型，这里需要通过创建 HttpContext 对象实例来初始化当前上下文，第二个参数就是我们要发布的方法集合。

按请求发布方法时，发布的方法不必是线程安全的，按会话或者全局发布的方法应该保证方法本身是线程安全的。按请求发布方法的情况不多，但是按会话发布方法的情况却可能经常会遇到，下面我们就来说明一下如何按会话来发布方法。

按会话发布方法

因为按会话发布和按请求发布的主要区别在 processRequest 方法上，所以我们下面的例子中，只写这一个方法。

```
protected void processRequest(HttpServletRequest request, HttpServletResponse
response)
    throws ServletException, IOException {
```

```

HttpSession session = request.getSession();
HproseHttpMethods methods = (HproseHttpMethods) session.getAttribute("exam2");
if (methods == null) {
    Exam2 exam2 = new Exam2();
    methods = new HproseHttpMethods();
    methods.addInstanceMethods(exam2);
    methods.addInstanceMethods(exam2, Exam1.class);
    session.setAttribute("exam2", methods);
}
service.handle(new HttpContext(request, response, this.getServletConfig(), this.getServletContext()),
    methods);
}

```

在这个例子中我们首先获取了 session 对象。之后，我们查询 session 中是否已经包含了发布的方法集合，如果有就直接使用它，否则通过 new 运算符创建一个新的，在添加完发布的方法后，将其以 exam2 为名放入 session 中。之后就是发布 exam2 上的方法了。

这个例子很好懂，这里就不再多作解释啦。

HproseHttpService 还有许多方法和属性，可以直接参见 HproseServlet 源码来了解它们的用法。

自己编写 JSP 发布 Hprose 服务

使用 JSP 来发布 Hprose 很简单，也是使用 HproseHttpService 来实现，但这是最不常用的方式，当然也是不推荐的方式。因为使用 JSP 方式相对于 Servlet 来说效率较低。

所以，下面我们仅用一个简单的例子来结束本章，不再对例子作详细说明，因为有了前面的基础，相信您一定很容易看懂下面的例子：

```

<%@page contentType="text/plain" pageEncoding="UTF-8"%>
<%@page import="hprose.server.*"%>
<%@page import="hprose.exam.*"%>
<jsp:useBean id="service" scope="application" class="hprose.server.HproseHttpService" />
<jsp:setProperty name="service" property="debugEnabled" value="true" />
<%
    Exam2 exam2 = new Exam2();
    HproseHttpMethods methods = new HproseHttpMethods();
    methods.addInstanceMethods(exam2);
    methods.addInstanceMethods(exam2, Exam1.class);
    service.handle(new HttpContext(request, response, config, application), methods);
%>

```

在某些服务器上用 JSP 方式发布服务，可能会出现一下错误信息：

```
java.lang.IllegalStateException: getOutputStream() has already been called for this response
```

解决方法只需要在页面最后，加入以下语句即可：

```
out.clear();  
out = pageContext.pushBody();
```

另外要注意保存 JSP 文件时请使用 UTF-8 编码，并去掉 BOM。

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

本章提要

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

同步调用

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

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

直接使用 HproseHttpClient 上的 invoke 方法或者采用代理接口方式都可以进行同步调用，但是只有通过 invoke 方法才能进行引用参数传递。

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

通过 invoke 方法进行同步调用

通过 invoke 方法调用是最直接、最基本的方式，所以我们先来介绍它。

带名称空间（别名前缀）方法

先来看调用最简单的例子：

```
package hprose.exam;
import hprose.client.HproseHttpClient;
import java.io.IOException;
public class ClientExam1 {
    public static void main(String[] args) throws IOException {
        HproseHttpClient client = new HproseHttpClient();
        client.useService("http://localhost:8084/HproseExamServer/Methods");
        System.out.println(client.invoke("ex1_getId"));
        System.out.println(client.invoke("ex2_getId"));
    }
}
```

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

```
Exam1
Exam2
```

从结果中我们可以很清楚的分辨出它们确实是调用了两个不同类的对象的方法。

可变的参数和结果类型

下面我们再来看对 sum 方法的调用：

```
package hprose.exam;
import hprose.client.HproseHttpClient;
```



```

import java.io.IOException;
import java.util.ArrayList;
public class ClientExam2 {
    public static void main(String[] args) throws IOException {
        HproseHttpClient client = new HproseHttpClient();
        client.useService("http://localhost:8084/HproseExamServer/Methods");
        System.out.println(client.invoke("ex1_sum", new Object[] { new int[] {1,2,3,4,5} }));
        System.out.println(client.invoke("ex1_sum", new Object[] { new short[] {6,7,8,9,10} }));
        System.out.println(client.invoke("ex1_sum", new Object[] { new long[] {11,12,13,14,15} }));
        System.out.println(client.invoke("ex1_sum", new Object[] { new double[] {16,17,18,19,20} }));
        System.out.println(client.invoke("ex1_sum", new Object[] { new String[]
{"21","22","23","24","25"} }));
        ArrayList<Integer> intList = new ArrayList<>();
        intList.add(26);
        intList.add(27);
        intList.add(28);
        intList.add(29);
        intList.add(30);
        System.out.println(client.invoke("ex2_sum", new Object[] { intList,
double.class}));
    }
}

```

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

下面看运行结果：

```

15
40
65
90
115
140.0

```

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

sum 的结果默认以 Integer 类型返回，但是您也可以指定其它类型作为返回值类型，只要该类型是反序列化有效类型即可。例如上例中，最后一个调用 ex2_sum 的例子，我们指定了 double 类型为返回值类型，所以得到的结果是 140.0。

引用参数传递

下面我们来继续看对 swapKeyAndValue 方法的调用：

```

package hprose.exam;
import hprose.client.HproseHttpClient;
import java.io.IOException;
import java.util.HashMap;
public class ClientExam3 {
    public static void main(String[] args) throws IOException {
        HproseHttpClient client = new HproseHttpClient();
        client.useService("http://localhost:8084/HproseExamServer/Methods");
        HashMap map = new HashMap();
        map.put("January", "Jan");
        map.put("February", "Feb");
        map.put("March", "Mar");
        map.put("April", "Apr");
        map.put("May", "May");
        map.put("June", "Jun");
        map.put("July", "Jul");
        map.put("August", "Aug");
        map.put("September", "Sep");
        map.put("October", "Oct");
        map.put("November", "Nov");
        map.put("December", "Dec");
        Object[] arguments = new Object[] { map };
        HashMap result = (HashMap)client.invoke("ex1_swapKeyAndValue", arguments);
        System.out.println(map);
        System.out.println(result);
        System.out.println(arguments[0]);
        result = (HashMap)client.invoke("ex2_swapKeyAndValue", arguments, true);
        System.out.println(map);
        System.out.println(result);
        System.out.println(arguments[0]);
    }
}

```

运行结果：

```

{October=Oct, January=Jan, April=Apr, February=Feb, August=Aug, June=Jun, November=Nov, July=Jul, May=May,
December=Dec, March=Mar, September=Sep}
{Sep=September, Feb=February, Mar=March, Apr=April, Oct=October, Jan=January, May=May, Nov=November,
Dec=December, Jul=July, Aug=August, Jun=June}
{October=Oct, January=Jan, April=Apr, February=Feb, August=Aug, June=Jun, November=Nov, July=Jul, May=May,
December=Dec, March=Mar, September=Sep}
{October=Oct, January=Jan, April=Apr, February=Feb, August=Aug, June=Jun, November=Nov, July=Jul, May=May,
December=Dec, March=Mar, September=Sep}

```

```
{Sep=September, Feb=February, Mar=March, Apr=April, Oct=October, Jan=January, May=May, Nov=November,
Dec=December, Jul=July, Aug=August, Jun=June}
{Sep=September, Feb=February, Mar=March, Apr=April, Oct=October, Jan=January, May=May, Nov=November,
Dec=December, Jul=July, Aug=August, Jun=June}
```

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

自定义类型的传输

最后我们来看一下对 `getUserList` 方法的调用：

```
package hprose.exam;
import hprose.client.HproseHttpClient;
import java.io.IOException;
import java.util.List;
public class ClientExam4 {
    public static void main(String[] args) throws IOException {
        HproseHttpClient client = new HproseHttpClient();
        client.useService("http://localhost:8084/HproseExamServer/Methods");
        List<User> userList = (List<User>)client.invoke("ex2_getUserList");
        for (User user : userList) {
            System.out.printf("name: %s, ", user.getName());
            System.out.printf("age: %d, ", user.getAge());
            System.out.printf("sex: %s, ", user.getSex());
            System.out.printf("birthday: %s, ", user.getBirthday());
            System.out.printf("married: %s.", user.isMarried());
            System.out.println();
        }
        System.out.println();
        User[] users = (User[])client.invoke("ex2_getUserList", User[].class);
        for (User user : users) {
            System.out.printf("name: %s, ", user.getName());
            System.out.printf("age: %d, ", user.getAge());
            System.out.printf("sex: %s, ", user.getSex());
            System.out.printf("birthday: %s, ", user.getBirthday());
            System.out.printf("married: %s.", user.isMarried());
            System.out.println();
        }
    }
}
```

运行结果：

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

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

上例中，返回结果默认是 ArrayList，所以我们用 List 接收结果没有问题，另外，为了后面 for 循环时不用手动转型，我们使用的是泛型的 List，但是要注意，泛型中所标明的 User 类必须与服务器端的 User 类定义完全一致，包括名称空间（包名），否则在运行时会发生类型转换错误。

不过可以通过指定 User 数组类型作为返回结果，这种情况下，客户端的 User 类可以与服务器的 User 类名空间不同（甚至类名不同也可以），只需要其内部序列化的字段和属性相同就可以啦。通常，如果在不需要对返回的结果作增删的时候，指定数组类型作为返回结果更高效和方便一些。

通过代理接口进行同步调用

看完通过 invoke 进行同步调用的方式后，再来看一下通过接口进行同步调用的方式。通过接口方式进行同步调用更加直观，方便，但是不支持动态调用和引用参数传递。

接口定义

为了调用上面的方法，我们需要先定义接口，下面是接口的定义：

```
package hprose.exam;
import java.util.List;
import java.util.Map;
public interface IExam1 {
    String getId();
    int sum(int[] nums);
    int sum(short[] nums);
    int sum(long[] nums);
    int sum(double[] nums);
    int sum(String[] nums);
    double sum(List nums);
    Map<String, String> swapKeyAndValue(Map<String, String> strmap);
}
```

这个是与 Exam1 对应的接口，其中除了跟 Exam1 声明相同的方法签名以外，还有另外一些重载的方法，它们都能在调用过程中自动转换为正确的类型进行调用。

```
package hprose.exam;
```

```
public interface IExam2 {
    User[] getUserList();
}
```

这个接口与 Exam2 对应，返回类型我们改成了 User[] 类型。

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

带名称空间（别名前缀）方法

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

```
package hprose.exam;
import hprose.client.HproseHttpClient;
import java.io.IOException;
public class ClientExam5 {
    public static void main(String[] args) throws IOException {
        HproseHttpClient client = new HproseHttpClient();
        client.useService("http://localhost:8084/HproseExamServer/Methods");
        IExam1 exam1 = (IExam1) client.useService(IExam1.class, "ex1");
        IExam1 exam2 = (IExam1) client.useService(IExam1.class, "ex2");
        System.out.println(exam1.getId());
        System.out.println(exam2.getId());
    }
}
```

在这个例子中，exam1 和 exam2 都是 IExam1 的接口对象，但是它们被指定了不同的名称空间（别名前缀），因此在后面对它们上面的 getId 方法调用时，调用的就是两个不同的方法了。

运行结果如下：

```
Exam1
Exam2
```

可变的参数和结果类型

下面我们再来看对 sum 方法的调用：

```
package hprose.exam;
import hprose.client.HproseHttpClient;
import java.io.IOException;
import java.util.ArrayList;
public class ClientExam6 {
    public static void main(String[] args) throws IOException {
        HproseHttpClient client = new HproseHttpClient();
```

```

    client.useService("http://localhost:8084/HproseExamServer/Methods");
    IExam1 exam = (IExam1) client.useService(IExam1.class, "ex1");
    System.out.println(exam.sum(new int[] {1,2,3,4,5}));
    System.out.println(exam.sum(new short[] {6,7,8,9,10}));
    System.out.println(exam.sum(new long[] {11,12,13,14,15}));
    System.out.println(exam.sum(new double[] {16,17,18,19,20}));
    System.out.println(exam.sum(new String[] {"21","22","23","24","25"}));
    ArrayList intList = new ArrayList();
    intList.add(26);
    intList.add(27);
    intList.add(28);
    intList.add(29);
    intList.add(30);
    System.out.println(exam.sum(intList));
}
}

```

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

泛型参数和结果

下面继续来看对 `swapKeyAndValue` 方法的调用：

```

package hprose.exam;
import hprose.client.HproseHttpClient;
import java.io.IOException;
import java.util.HashMap;
import java.util.Map;
public class ClientExam7 {
    public static void main(String[] args) throws IOException {
        HproseHttpClient client = new HproseHttpClient();
        client.useService("http://localhost:8084/HproseExamServer/Methods");
        IExam1 exam = (IExam1) client.useService(IExam1.class, "ex1");
        Map<String, String> map = new HashMap<String, String>();
        map.put("January", "Jan");
        map.put("February", "Feb");
        map.put("March", "Mar");
        map.put("April", "Apr");
        map.put("May", "May");
        map.put("June", "Jun");
        map.put("July", "Jul");
        map.put("August", "Aug");
        map.put("September", "Sep");
    }
}

```

```

        map.put("October", "Oct");
        map.put("November", "Nov");
        map.put("December", "Dec");
        Map<String, String> map2 = exam.swapKeyAndValue(map);
        System.out.println(map);
        System.out.println(map2);
    }
}

```

运行结果如下：

```

{October=Oct, January=Jan, April=Apr, February=Feb, August=Aug, June=Jun, November=Nov, July=Jul, May=May,
December=Dec, March=Mar, September=Sep}
{Sep=September, Feb=February, Mar=March, Apr=April, Oct=October, Jan=January, May=May, Nov=November,
Dec=December, Jul=July, Aug=August, Jun=June}

```

泛型要注意的问题在上一章介绍本方法的发布时已经做过说明啦，这里就不再重复了。

自定义类型

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

```

package hprose.exam;
import hprose.client.HproseHttpClient;
import java.io.IOException;
public class ClientExam8 {
    public static void main(String[] args) throws IOException {
        HproseHttpClient client = new HproseHttpClient();
        client.useService("http://localhost:8084/HproseExamServer/Methods");
        IExam2 exam2 = (IExam2) client.useService(IExam2.class, "ex2");
        User[] users = exam2.getUserList();
        for (User user : users) {
            System.out.printf("name: %s, ", user.getName());
            System.out.printf("age: %d, ", user.getAge());
            System.out.printf("sex: %s, ", user.getSex());
            System.out.printf("birthday: %s, ", user.getBirthday());
            System.out.printf("married: %s.", user.isMarried());
            System.out.println();
        }
    }
}

```

运行结果如下：

```

name: Amy, age: 26, sex: Female, birthday: 1983-12-03, married: true.

```

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

同样,这个例子已经很好的说明了 Hprose 使用的易用性和灵活性,不用多做解释相信您也已经看懂了,所以这里就不再多作解释啦。

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。

虽然您也可以使用多线程加同步调用来完成异步调用(实际上 Hprose 的异步调用也是如此实现的),但您不必这样做。您可以直接使用 Hprose 提供的异步调用方式,这将更加简单。

通过 invoke 方法进行异步调用

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

因为 java 本身不能向 C/C++ 那样传递函数或方法指针,也不能向 C# 那样传递方法委托,所以在 java 中 invoke 的回调方法是以接口 HproseCallback 来表示的,它只有一个方法:

```
void handler(Object result, Object[] arguments);
```

其中,第一个参数表示返回结果,第二个参数表示调用参数。如果您不是在进行引用参数传递的调用,那么第二个参数的参数值,跟您调用时的参数值是一致的。

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,其中的参将会自动被传入。

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

```
package hprose.exam;
import hprose.client.HproseHttpClient;
import hprose.common.HproseCallback;
import java.io.IOException;
public class ClientExam9 {
    public static void main(String[] args) throws IOException {
        HproseHttpClient client = new HproseHttpClient();
        client.useService("http://localhost:8084/HproseExamServer/Methods");
        client.invoke("ex1_getId", new HproseCallback() {
```



```

        public void handler(Object result, Object[] args) {
            System.out.println(result);
        }
    });
    client.invoke("ex2_getId", new HproseCallback() {
        public void handler(Object result, Object[] args) {
            System.out.println(result);
        }
    });
    System.in.read();
}
}

```

这个例子展示了同时进行 ex1_getId 和 ex2_getId 这两个方法的异步调用。

这里都是通过创建匿名类来实现回调方法的，您也可以使用命名类来定义，但通常使用匿名类来定义回调方法会更灵活，更直观。

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

这个程序的执行结果为：

```

Exam1
Exam2

```

但也可能为：

```

Exam2
Exam1

```

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

通过代理接口进行异步调用

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

```

package hprose.exam;
import hprose.client.HproseHttpClient;
import hprose.common.HproseCallback;
import java.io.IOException;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

```

```
interface ITest {
    Map<String, String> swapKeyAndValue(Map<String, String> strmap, HproseCallback callback);
    void swapKeyAndValue(Map<String, String> strmap, HproseCallback callback, boolean byRef);
    void getUserList(HproseCallback callback, Class returnType);
}

public class ClientExam10 {
    public static void main(String[] args) throws IOException {
        HproseHttpClient client = new HproseHttpClient();
        client.useService("http://localhost:8084/HproseExamServer/Methods");
        final ITest test = (ITest) client.useService(ITest.class, "ex2");
        Map<String, String> map = new HashMap<String, String>();
        map.put("January", "Jan");
        map.put("February", "Feb");
        map.put("March", "Mar");
        map.put("April", "Apr");
        map.put("May", "May");
        map.put("June", "Jun");
        map.put("July", "Jul");
        map.put("August", "Aug");
        map.put("September", "Sep");
        map.put("October", "Oct");
        map.put("November", "Nov");
        map.put("December", "Dec");
        test.swapKeyAndValue(map, new HproseCallback() {
            public void handler(Object result, Object[] args) {
                synchronized (test) {
                    Map<String, String> map = (Map<String, String>)args[0];
                    Map<String, String> map2 = (Map<String, String>)result;
                    System.out.println("byVal:");
                    System.out.println(map);
                    System.out.println(map2);
                    System.out.println();
                }
            }
        });
        test.swapKeyAndValue(map, new HproseCallback() {
            public void handler(Object result, Object[] args) {
                synchronized (test) {
                    Map<String, String> map = (Map<String, String>)args[0];
                    Map<String, String> map2 = (Map<String, String>)result;
                    System.out.println("byRef:");
                    System.out.println(map);
                    System.out.println(map2);
                    System.out.println();
                }
            }
        });
    }
}
```

```

        }
    }
    }, true);
    test.getUserList(new HproseCallback() {
        public void handler(Object result, Object[] args) {
            List<User> users = (List<User>)result;
            for (User user : users) {
                System.out.printf("name: %s, ", user.getName());
                System.out.printf("age: %d, ", user.getAge());
                System.out.printf("sex: %s, ", user.getSex());
                System.out.printf("birthday: %s, ", user.getBirthday());
                System.out.printf("married: %s.", user.isMarried());
                System.out.println();
            }
            System.out.println();
        }
    }, null);
    test.getUserList(new HproseCallback() {
        public void handler(Object result, Object[] args) {
            User[] users = (User[])result;
            for (User user : users) {
                System.out.printf("name: %s, ", user.getName());
                System.out.printf("age: %d, ", user.getAge());
                System.out.printf("sex: %s, ", user.getSex());
                System.out.printf("birthday: %s, ", user.getBirthday());
                System.out.printf("married: %s.", user.isMarried());
                System.out.println();
            }
            System.out.println();
        }
    }, User[].class);
    System.in.read();
}
}

```

这个程序比较长，我们分段来分析。

首先来看接口定义，异步调用的接口方法中，都含有一个 `HproseCallback` 参数，该参数在所有的参数之后定义，在它之后的还有两个可选参数，一个表示返回值类型，另一个表示是否进行引用参数传递。这两个参数可以同时出现，同时出现时，返回值类型应该在引用参数传递参数之前。

接口中您可以有两种方式来定义返回值类型。

一种是以方法签名方式来直接表示返回值类型，不过这种方式千万不要与同步调用方式混淆，实际上在异步调用过程中不可能通过方法的返回值来直接得到调用的结果。如果返回值声明为原生数据类型，则永远返回该类型所对应的零值（例如 `int` 返回 0，`boolean` 返回 `false`），如果为引用类型，则永远返回 `null`。而具体的结果是在回调方法中通过第一个参数返回的。

通过方法签名方式来标记返回值是您可以认为是静态方式，因为它在调用过程中不能变化。另一种方法就是通过向 `HproseCallback` 参数之后声明返回值类型，这种是动态方式，它可以在调用过程中来指定具体的返回值类型。

第一个调用，我们采用的是静态方式，直接将返回值类型声明为 `Map<String,String>`。

第二个调用返回值类型声明为 `void` 类型，并且没有在后面跟随返回类型参数，它表示将按照默认反序列化类型返回数据。

后面两个调用的是同一个方法，该方法返回值声明为 `void` 类型，但是后面有跟随返回类型参数。

前一个带入的返回类型为 `null`，这表明将按照默认反序列化类型返回数据，因为这个方法返回的是元素为自定义类型（`User`）的列表类型，所以它所对应的默认反序列化类型是元素为自定义类型（`User`）的 `ArrayList`，当然在回调方法中我们就可以通过 `List<User>` 泛型接口类型来存取它了。

而后一个调用带入的返回类型为 `User[]`，所以，在回调方法中我们可以通过 `User[]` 类型来存取它。

关于引用参数传递，我们可以看前两个调用，第一个没有设置引用参数传递参数，它是值传递，第二个设置了引用参数传递，则它的参数在调用后会被改变。

您可能还发现我们在前两个调用的回调方法中作了同步，原因是这两个方法本身也是异步执行的，所以如果不作同步处理，则它们的结果可能会交替输出，这样我们就不能分辨究竟是不是引用参数传递啦。

后面两个调用没有作同步处理，它们的结果就可能会交替输出，甚至会在前两个方法中间输出内容。

下面是该程序运行的一种可能的结果：

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

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

{October=Oct, January=Jan, April=Apr, February=Feb, August=Aug, June=Jun, November=Nov, July=Jul, May=May,
December=Dec, March=Mar, September=Sep}
{Sep=September, Feb=February, Mar=March, Apr=April, Oct=October, Jan=January, May=May, Nov=November,
Dec=December, Jul=July, Aug=August, Jun=June}

byRef:
{Sep=September, Feb=February, Mar=March, Apr=April, Oct=October, Jan=January, May=May, Nov=November,
Dec=December, Jul=July, Aug=August, Jun=June}
{Sep=September, Feb=February, Mar=March, Apr=April, Oct=October, Jan=January, May=May, Nov=November,
Dec=December, Jul=July, Aug=August, Jun=June}
```

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

同步调用下发生的异常将被直接抛出，使用 try...catch 语句块即可捕获异常，通常服务器端调用返回的异常是 HproseException 类型。而如果通讯发生错误，一般为 IOException 类型。但是在调用过程中也可能抛出其它类型的异常，为了保险，您可以使用 catch 捕获 Throwable 类型来处理全部可能发生的异常。

另外，如果您采用代理接口方式调用，需要注意，您捕捉到的异常是被包装后的，使用 getCause 方法来获取真正包含出错信息的异常。

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 hprose.common.HproseErrorEvent，它是一个接口类型，使用非常简单，例如：

```
HproseHttpClient client = new HproseHttpClient();
client.onError = new HproseErrorEvent() {
    public void handler(String name, Throwable ex) {
        System.out.println(name + ":" + ex.toString());
    }
};
```

该接口只有一个方法 handler，第一个参数表示发生错误的远程方法名，第二个参数为抛出并被捕获的异常对象。

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

HTTP 参数设置

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

代理服务器

默认情况下，代理服务器是被禁用的。可以通过 setProxyHost、setProxyPort 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 null，表示禁用代理服务器。ProxyPort 为端口号，用整数表示，默认是 80。

如果您所使用的代理服务器需要身份验证，可以通过 `setProxyUser` 和 `setProxyPass` 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 `null`，表示不需要身份验证。

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`，可以通过它们的 `set` 方法（`setKeepAlive` 和 `setKeepAliveTimeout`）来对它们进行设置。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。

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`
- `Connection`
- `Keep-Alive`
- `Host`

因为这些标头有特别意义，客户端会自动设定这些值。

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

第五章 其它平台

Hprose for Java 除了支持 JavaSE 和 JavaEE 平台之外,同样也支持 Google App Engine、Android 平台,同时还提供了单独的 JavaME 平台的实现。本章我们主要来了解一下在这些平台上使用 Hprose for Java 开发需要注意的一些问题。

本章提要

- Hprose for GAE
- Hprose for Android
- Hprose for JavaME

Hprose for GAE

Google App Engine 是 Google 提供的云计算平台，它支持使用 Java 来进行开发。

Hprose for Java 可以很好的工作于 Google App Engine 平台之上，不论是服务器还是客户端都可以完美支持。

Hprose for Java 服务器端在 GAE 上运行部署可以采用直接配置 HproseServlet 的方式，也可以采用自己编写 Servlet 的方式发布 Hprose 服务。这一点跟在其它 JavaEE 服务器上发布 Hprose 服务没有什么区别。

Hprose for Java 客户端在 GAE 上仅支持同步调用，不支持异步调用，因为 GAE 禁止使用线程类。

Hprose for Java 客户端在 GAE 上不能调用其自身应用发布的服务，因为 GAE 为防止访问循环死锁，限制自我访问。

Hprose for Java 客户端上不能够设置代理服务器和 KeepAlive 属性，因为 GAE 不支持。

Hprose for Java 客户端在 GAE 上只能访问标准的 http 和 https 端口发布的 Hprose。因为 GAE 限制不支持对其它端口的访问。

Hprose for Android

Hprose for Java 客户端可以直接运行于 Android 平台之上，跟在普通的 JavaSE 平台上开发没有任何区别。但在调用时推荐采用异步方式，因为同步调用方式对于界面操作不友好。另外，如果您遇到：

```
java.net.SocketException: Permission denied (maybe missing INTERNET permission)
```

这个错误，只需要在 AndroidManifest.xml 中，需要进行如下配置即可：

```
<uses-permission android:name="android.permission.INTERNET" />
```

Hprose for JavaME

因为 JavaME 平台的特殊性，Hprose 专门为 JavaME 提供了单独版本。并且对不同的 JavaME 环境提供了不同的版本。目前的版本提供了对 CDC、CLDC 1.0 和 CLDC 1.1 这三种不同环境的支持。

CDC 环境

对于 CDC 环境来说，Hprose for JavaME 的用法跟 JavaSE 环境下基本相同，仅仅在类型上缺少 java.sql.Date， java.sql.Time， java.sql.Timestamp 这三种类型的支持，因为在 CDC 环境下没有这三种类型。但是这并不妨碍跟 Java 或其它语言的交互，因为 java.util.Date， java.util.Calendar 这两种类型都可以接收或发送时间型、日期型和日期时间型的数据。

在 CDC 环境下可以设置 KeepAlive 属性，但是不支持代理服务器设置。

CLDC 1.0 环境

大多数老式的支持 JavaME 的手机支持的都是 CLDC 1.0。

在该配置下，不支持浮点数运算。如果接受其它语言传来的浮点数，默认映射为 String 类型，如果手动指定为其它整数类型，可能会损失精度，或者结果完全不对（当浮点数以科学计数法表示时）。所以如果一定要接受浮点数，推荐用默认映射。

Hprose 长整型默认也是映射为 String 类型，因为 CLDC 1.0 不支持 BigInteger 类型。您也可以手动指定为其它整数类型，但当超出类型范围时，会出错。

日期时间处理上跟 CDC 环境下一样，使用 java.util.Date, java.util.Calendar 这两种类型来接收或发送时间型、日期型和日期时间型的数据。

容器类型处理上，Hprose 列表类型默认映射为 Vector 类型，Hprose 类型映射为 Hashtable 类型。

因为 CLDC 1.0 不支持 Java 可序列化对象，因此 Hprose 可序列化对象类需要实现 hprose.io.Serializable 接口，该接口定义如下：

```
package hprose.io;

public interface Serializable {
    String[] getPropertyNames();
    Class getPropertyType(String name);
    Object getProperty(String name);
    void setProperty(String name, Object value);
}
```

getPropertyNames 方法的作用是返回所有需要序列化的属性名。

getPropertyType 方法的作用是返回指定属性的类型。

getProperty 方法的作用是返回指定属性的值。

setProperty 方法的作用是设置指定属性的值。

这四个方法都需要用户自己来实现。

当然，您也可以不定义类，而使用 Hashtable 类型来接受服务器端传来的对象。

在 CLDC 1.0 环境下进行远程调用时，仅支持 invoke 方式，不支持代理接口方式。

在 CLDC 1.0 环境下可以设置 KeepAlive 属性，但是不支持代理服务器设置。

CLDC 1.1 环境

Hprose 为 CLDC 1.1 提供了两个版本的实现，一个普通版本，另一个是扩展版本。

普通版本跟 CLDC 1.0 版本基本一样，唯一明显不同是 CLDC 1.1 版本支持浮点数类型的传输。

扩展版本提供了跟 JavaSE 1.4 兼容的 BigInteger 和 BigDecimal 实现，并且包名与 JavaSE 中一致。

扩展版本还提供了跟 JavaSE 1.4 兼容的绝大部分容器类型，包括 ArrayList、HashMap、LinkedList、TreeMap 等。包名也是与 JavaSE 中一致。

因此，您很容易将 JavaSE 程序通过 Hprose 移植到 CLDC 1.1 平台上。不过因为扩展版本提供了这些兼容实现，所以生成的 jar 文件将会比普通版本大一些，如果您的运行环境对程序大小限制很严格，则不适合使用扩展版本。

附录 API 参考

本附录是 Hprose for Java 的主要类及其方法的 API 列表参考，方便用户在使用时可以直接查找，对于仅用于 Hprose for Java 内部使用的类及其方法本附录不做列举。

本章提要

- hprose.common 包
- hprose.io 包
- hprose.client 包
- hprose.server 包

hprose.common 包

HproseCallback 接口

该接口用于表示异步回调委托。

handler 方法

```
void handler(Object result, Object[] arguments)
```

该方法在异步调用执行后被调用，result 为异步调用的结果，arguments 为异步调用的参数。

HproseErrorEvent 接口

该接口用于表示异步错误事件。

handler 方法

```
void handler(String name, Throwable error)
```

该方法在异步调用发生错误时被调用，name 为发生错误时所执行的远程方法名，error 为发生错误时所抛出的异常对象。

HproseMethod 类

该类用于表示发布的方法。该类在内部使用，用户一般不会直接使用它，因此为了高效处理，下面的字段都是可以直接存取的，而没有与之对应的属性。

method 字段

```
java.lang.reflect.Method method
```

发布的方法。

obj 字段

```
java.lang.Object obj
```

方法绑定的对象。方法为静态方法时，该字段为 null。

paramTypes 字段

```
java.lang.Class[] paramTypes
```

方法的参数类型列表。

HproseMethods 类

该类用于表示发布方法的集合。

构造方法

```
HproseMethods()
```

该构造方法用于创建一个 HproseMethods 对象。但是请不要直接使用该构造方法，而应该使用它的子类的构造方法来创建该类的对象实例，除非您能够确认您所发布的方法中没有使用环境上下文参数。

addInstanceMethods 方法

```
void addInstanceMethods(Object obj)
```

添加 obj 对象上绑定的 obj 所在类直接声明的所有 public 实例方法到发布方法集合。

如果 obj 上方法有重载的情况，则所有参数个数不相同的方法都会被添加。如果有两个或多个参数个数也相同的方法，则只有其中一个方法会被添加。因此，用此方法发布的对象中应避免出现参数个数相同的重载情况。

```
void addInstanceMethods(Object obj, Class type)
```

添加 obj 对象上绑定的 type 类上直接声明的所有 public 实例方法到发布方法集合。

使用该方法请注意上面提到的重载冲突的情况。

```
void addInstanceMethods(Object obj, Class type, String aliasPrefix)
```

添加 obj 对象上绑定的 type 类上直接声明的所有 public 实例方法到发布方法集合，并设置这些方法的别名前缀为 aliasPrefix。

使用该方法请注意上面提到的重载冲突的情况。

```
void addInstanceMethods(Object obj, String aliasPrefix)
```

添加 obj 对象上绑定的 obj 所在类直接声明的所有 public 实例方法到发布方法集合，并设置这些方法的别名前缀为 aliasPrefix。

使用该方法请注意上面提到的重载冲突的情况。

addMethod 方法

```
void addMethod(java.lang.reflect.Method method, Object obj, String aliasName)
```

添加 method 方法到发布方法集合，将其绑定于对象 obj，并设置其别名为 aliasName。

```
void addMethod(String methodName, Class type)
```

添加 type 类上方法名为 methodName 的方法到发布方法集合。

如果在 type 类上有多个方法名为 methodName 的方法，则所有参数个数不相同的方法都会被添加。如果有两个或多个参数个数也相同的方法，则只有其中一个方法会被添加。因此，用此方法发布的对象中应避免出现参数个数相同的重载情况。

```
void addMethod(String methodName, Class type, Class[] paramTypes)
```

添加 type 类上方法名为 methodName，参数类型为 paramTypes 的方法到发布方法集合。

使用该方法可以避免上面所提到的重载冲突的情况。

```
void addMethod(String methodName, Class type, Class[] paramTypes, String aliasName)
```

添加 type 类上方法名为 methodName，参数类型为 paramTypes 的方法到发布方法集合，并设置其别名为 aliasName。

使用该方法可以避免上面所提到的重载冲突的情况。

```
void addMethod(String methodName, Class type, String aliasName)
```

添加 type 类上方法名为 methodName 的方法到发布方法集合，并设置其别名为 aliasName。

使用该方法请注意上面提到的重载冲突的情况。

```
void addMethod(String methodName, Object obj)
```

添加 obj 对象上方法名为 methodName 的方法到发布方法集合。

如果在 obj 对象所在类上有多个方法名为 methodName 的方法，则所有参数个数不相同的方法都会被添加。如果有两个或多个参数个数也相同的方法，则只有其中一个方法会被添加。因此，用此方法发布的对象中应避免出现参数个数相同的重载情况。

```
void addMethod(String methodName, Object obj, Class[] paramTypes)
```

添加 obj 对象上方法名为 methodName，参数类型为 paramTypes 的方法到发布方法集合。

使用该方法可以避免上面所提到的重载冲突的情况。

```
void addMethod(String methodName, Object obj, Class[] paramTypes, String aliasName)
```

添加 obj 对象上方法名为 methodName，参数类型为 paramTypes 的方法到发布方法集合，并设置其别名为 aliasName。

使用该方法可以避免上面所提到的重载冲突的情况。

```
void addMethod(String methodName, Object obj, String aliasName)
```

添加 obj 对象上方法名为 methodName 的方法到发布方法集合，并设置其别名为 aliasName。

使用该方法请注意上面提到的重载冲突的情况。

addMethods 方法

```
void addMethods(String[] methodNames, Class type)
```

添加 type 类上方法名为 methodNames 的一组方法到发布方法集合。

如果所发布的方法在 type 类上有重载，则所有参数个数不相同的方法都会被添加。如果有两个或多个参数个数也相同的方法，则只有其中一个方法会被添加。因此，用此方法发布的对象中应避免出现参数个数相同的重载情况。

```
void addMethods(String[] methodNames, Class type, String aliasPrefix)
```

添加 type 类上方法名为 methodNames 的一组方法到发布方法集合，并设置这些方法的别名前缀为 aliasPrefix。

使用该方法请注意上面提到的重载冲突的情况。

```
void addMethods(String[] methodNames, Class type, String[] aliasNames)
```

添加 type 类上方法名为 methodNames 的一组方法到发布方法集合，并设置这些方法的别名为 aliasNames。

使用该方法请注意上面提到的重载冲突的情况。

```
void addMethods(String[] methodNames, Object obj)
```

添加 obj 对象上方法名为 methodNames 的一组方法到发布方法集合。

如果所发布的方法在 obj 对象上有重载，则所有参数个数不相同的方法都会被添加。如果有两个或多个参数个数也相同的方法，则只有其中一个方法会被添加。因此，用此方法发布的对象中应避免出现参数个数相同的重载情况。

```
void addMethods(String[] methodNames, Object obj, String aliasPrefix)
```

添加 obj 对象上方法名为 methodNames 的一组方法到发布方法集合，并设置这些方法的别名前缀为 aliasPrefix。

使用该方法请注意上面提到的重载冲突的情况。

```
void addMethods(String[] methodNames, Object obj, String[] aliasNames)
```

添加 obj 对象上方法名为 methodNames 的一组方法到发布方法集合，并设置这些方法的别名为 aliasNames。

使用该方法请注意上面提到的重载冲突的情况。

addMissingMethod 方法

```
void addMissingMethod(String methodName, Class type)
```

添加 type 类上名称为 methodName 的方法为默认方法。该方法对参数有明确限制，参数为 2 个，按从

左到右的顺序分别为 String，Object[]类型。所以不会有重载冲突问题。

当客户端调用服务器端的方法时，如果服务器端在发布的方法集合中找不到客户端调用的方法，则会执行该方法。客户端调用的方法名会作为该方法的第一个参数被传入，客户端调用的方法参数会作为该方法的第二个参数被传入。

```
void addMissingMethod(String methodName, Object obj)
```

添加 obj 对象上名称为 methodName 的方法为默认方法。该方法对参数有明确限制，参数为 2 个，按从左到右的顺序分别为 String，Object[]类型。所以不会有重载冲突问题。

用该方法发布的方法功能跟上面的功能一致。

以上两个方法只有最后一次添加的有效。

addStaticMethods 方法

```
void addStaticMethods(Class type)
```

添加 type 类上直接声明的所有 public 静态方法到发布方法集合。

如果 type 类上方法有重载的情况，则所有参数个数不相同的方法都会被添加。如果有两个或多个参数个数也相同的方法，则只有其中一个方法会被添加。因此，用此方法发布的对象中应避免出现参数个数相同的重载情况。

```
void addStaticMethods(Class type, String aliasPrefix)
```

添加 type 类上直接声明的所有 public 静态方法到发布方法集合，并设置这些方法的别名前缀为 aliasPrefix。

使用该方法请注意上面提到的重载冲突的情况。

get 方法

```
HproseMethod get(String aliasName, int paramCount)
```

通过别名和参数个数来获取发布的方法，如果方法不存在则返回 null。

getAllNames 方法

```
java.util.Set getAllNames()
```

获取所有发布方法名（别名）的集合。该集合为只读的。

getCount 方法

```
int getCount()
```

获取发布的方法数量。重载方法是按照多个统计的，而不是只算一个。

HproseException 异常

该异常继承自 java.io.IOException，当 Hprose 内部输入输出发生错误时，引发该异常。

hprose.io 包

HproseMode 类

该类是一个枚举类，但为了兼容 Java 1.2，因此该类并没有采用 Java 1.5 的 enum 方式定义，该类有两个值，它们以静态字段表示。该类用于表示序列化模式。

FieldMode 静态字段

```
static HproseMode FieldMode
```

字段模式，该模式表示只序列化类中的实例字段，transient 的字段不序列化。

PropertyMode 静态字段

```
static HproseMode PropertyMode
```

属性模式，该模式表示只序列化类中的可读写属性。

HproseTags 类

该类中定义了序列化标记。该类为静态类，不必实例化。用户一般不需要使用该类，该类在 Hprose 内部使用。

TagArgument 静态字段

```
static int TagArgument
```

参数标记。

TagBytes 静态字段

```
static int TagBytes
```

二进制串标记。

TagCall 静态字段

```
static int TagCall
```

调用标记。

TagClass 静态字段

```
static int TagClass
```

类标记。

TagClosebrace 静态字段

```
static int TagClosebrace
```

右花括号标记。

TagDate 静态字段

```
static int TagDate
```

日期标记。

TagDouble 静态字段

```
static int TagDouble
```

浮点数标记。

TagEmpty 静态字段

```
static int TagEmpty
```

空串标记。

TagEnd 静态字段

```
static int TagEnd
```

结束标记。

TagError 静态字段

```
static int TagError
```

错误标记。

TagFalse 静态字段

```
static int TagFalse
```

假值标记。

TagFunctions 静态字段

```
static int TagFunctions
```

发布函数列表标记。

TagInfinity 静态字段

```
static int TagInfinity
```

无穷大标记。

TagInteger 静态字段

```
static int TagInteger
```

整数标记。

TagList 静态字段

```
static int TagList
```

列表标记。

TagLong 静态字段

```
static int TagLong
```

长整型标记。

TagMap 静态字段

```
static int TagMap
```

表标记。

TagNaN 静态字段

```
static int TagNaN
```

非数标记。

TagNeg 静态字段

```
static int TagNeg
```

负号标记。

TagNull 静态字段

```
static int TagNull
```

空标记。

TagObject 静态字段

```
static int TagObject
```

对象标记。

TagOpenbrace 静态字段

```
static int TagOpenbrace
```

左花括号标记。

TagPoint 静态字段

```
static int TagPoint
```

点标记。

TagPos 静态字段

```
static int TagPos
```

正号标记。

TagQuote 静态字段

```
static int TagQuote
```

引号标记。

TagRef 静态字段

```
static int TagRef
```

引用标记。

TagResult 静态字段

```
static int TagResult
```

结果标记。

TagSemicolon 静态字段

```
static int TagSemicolon
```

分号标记。

TagString 静态字段

```
static int TagString
```

字符串标记。

TagTime 静态字段

```
static int TagTime
```

时间标记。

TagTrue 静态字段

```
static int TagTrue
```

真值标记。

TagUTC 静态字段

```
static int TagUTC
```

世界标准时间标记。

HproseReader 类

用于反序列化的类。

stream 字段

```
java.io.InputStream stream
```

数据源流。

构造方法

```
HproseReader(java.io.InputStream stream)
```

参数 stream 是数据源流。

```
HproseReader(java.io.InputStream stream, HproseMode mode)
```

参数 stream 是数据源流, mode 是为序列化模式。

checkTag 方法

```
void checkTag(int expectTag)
```

检查数据流中下一个字节是否是 expectTag。

checkTags 方法

```
int checkTags(String expectTags)
```

检查数据流中下一个字节是否是 expectTags 中的一个。

readArray 方法

```
void readArray(Class[] types, Object[] a, int count)
```

从数据流中读取元素类型分别为 types 元素个数为 count 的数组 a。

```
Object[] readArray(int count)
```

从数据流中读取元素个数为 count 的数据, 并返回。

readBigInteger 方法

```
java.math.BigInteger readBigInteger()
```

从数据流中读取一个大整数, 并在读取数据前先读取并判断标记。

```
java.math.BigInteger readBigInteger(boolean includeTag)
```

从数据流中读取一个大整数, includeTag 表示是否在读取数据前先读取并判断标记。

readBoolean 方法

```
boolean readBoolean()
```

从数据流中读取一个布尔值。

readBytes 方法

```
byte[] readBytes()
```

从数据流中读取一个二进制串到字节数组, 并在读取数据前先读取并判断标记。

```
byte[] readBytes(boolean includeTag)
```

从数据流中读取一个二进制串到字节数组，includeTag 表示是否在读取数据前先读取并判断标记。

readDate 方法

```
Object readDate()
```

从数据流中读取一个日期类型对象，并在读取数据前先读取并判断标记，结果以 Calendar 类型返回。

```
Object readDate(boolean includeTag)
```

从数据流中读取一个日期类型对象，includeTag 表示是否在读取数据前先读取并判断标记，结果以 Calendar 类型返回。

```
Object readDate(boolean includeTag, Class type)
```

从数据流中读取一个日期类型对象，includeTag 表示是否在读取数据前先读取并判断标记，结果以指定的 type 类型返回。

```
Object readDate(Class type)
```

从数据流中读取一个日期类型对象，并在读取数据前先读取并判断标记，结果以指定的 type 类型返回。

readDateTime 方法

```
Object readDateTime()
```

从数据流中读取一个日期或时间类型对象，并在读取数据前先读取并判断标记，结果以指定的 Calendar 类型返回。

```
Object readDateTime(Class type)
```

从数据流中读取一个日期或时间类型对象，并在读取数据前先读取并判断标记，结果以指定的 type 类型返回。

readDouble 方法

```
double readDouble()
```

从数据流中读取一个双精度浮点数类型数据，并在读取数据前先读取并判断标记。

```
double readDouble(boolean includeTag)
```

从数据流中读取一个双精度浮点数类型数据，includeTag 表示是否在读取数据前先读取并判断标记。

readEmpty 方法

```
Object readEmpty()
```

从数据流中读取一个空串类型数据，以空字符串返回。

readEnum 方法

```
Object readEnum(Class type)
```

从数据流中读取一个枚举类型数据，并在读取数据前先读取并判断标记，结果以指定的 type 类型返回。

```
Object readEnum(Class type, boolean includeTag)
```

从数据流中读取一个枚举类型数据，includeTag 表示是否在读取数据前先读取并判断标记，结果以指定的 type 类型返回。

readInfinity 方法

```
double readInfinity()
```

从数据流中读取一个无穷大数，并在读取数据前先读取并判断标记。

```
double readInfinity(boolean includeTag)
```

从数据流中读取一个无穷大数，includeTag 表示是否在读取数据前先读取并判断标记。

readInteger 方法

```
int readInteger()
```

从数据流中读取一个整型数字，并在读取数据前先读取并判断标记。

```
int readInteger(boolean includeTag)
```

从数据流中读取一个整型数字，includeTag 表示是否在读取数据前先读取并判断标记。

readList 方法

```
Object readList()
```

从数据流中读取一个列表类型对象，并在读取数据前先读取并判断标记，结果以 ArrayList 类型返回。

```
Object readList(boolean includeTag)
```

从数据流中读取一个列表类型对象，includeTag 表示是否在读取数据前先读取并判断标记，结果以 ArrayList 类型返回。

```
Object readList(boolean includeTag, Class type)
```

从数据流中读取一个列表类型对象, includeTag 表示是否在读取数据前先读取并判断标记, 结果以指定的 type 类型返回。

```
Object readList(Class type)
```

从数据流中读取一个列表类型对象, 并在读取数据前先读取并判断标记, 结果以指定的 type 类型返回。

readLong 方法

```
long readLong()
```

从数据流中读取一个长整型数字, 并在读取数据前先读取并判断标记。

```
long readLong(boolean includeTag)
```

从数据流中读取一个长整型数字, includeTag 表示是否在读取数据前先读取并判断标记。

readMap 方法

```
Object readMap()
```

从数据流中读取一个表类型对象, 并在读取数据前先读取并判断标记, 结果以 HashMap 类型返回。

```
Object readMap(boolean includeTag)
```

从数据流中读取一个表类型对象, includeTag 表示是否在读取数据前先读取并判断标记, 结果以 HashMap 类型返回。

```
Object readMap(boolean includeTag, Class type)
```

从数据流中读取一个表类型对象, includeTag 表示是否在读取数据前先读取并判断标记, 结果以指定的 type 类型返回。

```
Object readMap(Class type)
```

从数据流中读取一个表类型对象, 并在读取数据前先读取并判断标记, 结果以指定的 type 类型返回。

readNaN 方法

```
double readNaN()
```

从数据流中读取一个非数。

readNull 方法

```
Object readNull()
```

从数据流中读取一个 null。

readObject 方法

```
Object readObject()
```

从数据流中读取一个对象，并在读取数据前先读取并判断标记，结果以实际类型返回。

```
Object readObject(boolean includeTag)
```

从数据流中读取一个对象，includeTag 表示是否在读取数据前先读取并判断标记，结果以实际类型返回。

```
Object readObject(boolean includeTag, Class type)
```

从数据流中读取一个对象，includeTag 表示是否在读取数据前先读取并判断标记，结果以指定的 type 类型返回。

```
Object readObject(Class type)
```

从数据流中读取一个对象，并在读取数据前先读取并判断标记，结果以指定的 type 类型返回。

readString 方法

```
Object readString()
```

从数据流中读取一个字符串类型对象，并在读取数据前先读取并判断标记，结果以 String 类型返回。

```
Object readString(boolean includeTag)
```

从数据流中读取一个字符串类型对象，includeTag 表示是否在读取数据前先读取并判断标记，结果以 String 类型返回。

```
Object readString(boolean includeTag, Class type)
```

从数据流中读取一个字符串类型对象，includeTag 表示是否在读取数据前先读取并判断标记，结果以指定的 type 类型返回。

```
Object readString(Class type)
```

从数据流中读取一个字符串类型对象，并在读取数据前先读取并判断标记，结果以指定的 type 类型返回。

readTime 方法

```
Object readTime()
```

从数据流中读取一个时间类型对象，并在读取数据前先读取并判断标记，结果以 Calendar 类型返回。

```
Object readTime(boolean includeTag)
```

从数据流中读取一个时间类型对象，includeTag 表示是否在读取数据前先读取并判断标记，结果以 Calendar 类型返回。

```
Object readTime(boolean includeTag, Class type)
```

从数据流中读取一个时间类型对象，includeTag 表示是否在读取数据前先读取并判断标记，结果以指定的 type 类型返回。

```
Object readTime(Class type)
```

从数据流中读取一个时间类型对象，并在读取数据前先读取并判断标记，结果以指定的 type 类型返回。

readUntil 方法

```
String readUntil(int tag)
```

从数据流中读取字符串直到遇到标记 tag。

unserialize 方法

```
Object unserialize()
```

从数据流中反序列化当前数据并返回。

```
Object unserialize(Class type)
```

从数据流中以指定的 type 类型反序列化当前数据并返回。

HproseWriter 类

用于序列化的类。

stream 字段

```
java.io.OutputStream stream
```

数据源流。

构造方法

```
HproseWriter(java.io.OutputStream stream)
```

参数 stream 是数据源流。

```
HproseWriter(java.io.OutputStream stream, HproseMode mode)
```

参数 stream 是数据源流，mode 是为序列化模式。

serialize 方法

```
void serialize(Object obj)
```

将对象 obj 序列化到数据流中。

writeArray 方法

```
void writeArray(BigDecimal[] array)
```

将数组 array 序列化到数据流中。

```
void writeArray(BigDecimal[] array, boolean checkRef)
```

将数组 array 序列化到数据流中，checkRef 表示是否在序列化时检查引用。

```
void writeArray(BigInteger[] array)
```

将数组 array 序列化到数据流中。

```
void writeArray(BigInteger[] array, boolean checkRef)
```

将数组 array 序列化到数据流中，checkRef 表示是否在序列化时检查引用。

```
void writeArray(boolean[] array)
```

将数组 array 序列化到数据流中。

```
void writeArray(boolean[] array, boolean checkRef)
```

将数组 array 序列化到数据流中，checkRef 表示是否在序列化时检查引用。

```
void writeArray(byte[][] array)
```

将数组 array 序列化到数据流中。

```
void writeArray(byte[][] array, boolean checkRef)
```

将数组 array 序列化到数据流中，checkRef 表示是否在序列化时检查引用。

```
void writeArray(char[][] array)
```

将数组 array 序列化到数据流中。

```
void writeArray(char[][] array, boolean checkRef)
```

将数组 array 序列化到数据流中，checkRef 表示是否在序列化时检查引用。

```
void writeArray(double[] array)
```

将数组 array 序列化到数据流中。

```
void writeArray(double[] array, boolean checkRef)
```

将数组 array 序列化到数据流中，checkRef 表示是否在序列化时检查引用。

```
void writeArray(float[] array)
```

将数组 array 序列化到数据流中。

```
void writeArray(float[] array, boolean checkRef)
```

将数组 array 序列化到数据流中，checkRef 表示是否在序列化时检查引用。

```
void writeArray(int[] array)
```

将数组 array 序列化到数据流中。

```
void writeArray(int[] array, boolean checkRef)
```

将数组 array 序列化到数据流中，checkRef 表示是否在序列化时检查引用。

```
void writeArray(long[] array)
```

将数组 array 序列化到数据流中。

```
void writeArray(long[] array, boolean checkRef)
```

将数组 array 序列化到数据流中，checkRef 表示是否在序列化时检查引用。

```
void writeArray(Object array)
```

将数组 array 序列化到数据流中。

```
void writeArray(Object[] array)
```

将数组 array 序列化到数据流中。

```
void writeArray(Object[] array, boolean checkRef)
```

将数组 array 序列化到数据流中，checkRef 表示是否在序列化时检查引用。

```
void writeArray(Object array, boolean checkRef)
```

将数组 array 序列化到数据流中，checkRef 表示是否在序列化时检查引用。

```
void writeArray(short[] array)
```

将数组 array 序列化到数据流中。

```
void writeArray(short[] array, boolean checkRef)
```

将数组 array 序列化到数据流中，checkRef 表示是否在序列化时检查引用。

```
void writeArray(String[] array)
```

将数组 array 序列化到数据流中。

```
void writeArray(String[] array, boolean checkRef)
```

将数组 array 序列化到数据流中，checkRef 表示是否在序列化时检查引用。

```
void writeArray(StringBuffer[] array)
```

将数组 array 序列化到数据流中。

```
void writeArray(StringBuffer[] array, boolean checkRef)
```

将数组 array 序列化到数据流中，checkRef 表示是否在序列化时检查引用。

writeBoolean 方法

```
void writeBoolean(boolean b)
```

将布尔值 b 序列化到数据流中。

writeBytes 方法

```
void writeBytes(byte[] bytes)
```

将字节数组 bytes 序列化到数据流中。

```
void writeBytes(byte[] bytes, boolean checkRef)
```

将字节数组 `bytes` 序列化到数据流中，`checkRef` 表示是否在序列化时检查引用。

writeCollection 方法

```
void writeCollection(java.util.Collection collection)
```

将集合 `collection` 序列化到数据流中。

```
void writeCollection(java.util.Collection collection, boolean checkRef)
```

将集合 `collection` 序列化到数据流中，`checkRef` 表示是否在序列化时检查引用。

writeDate 方法

```
void writeDate(java.util.Calendar calendar)
```

将 `calendar` 序列化到数据流中。

```
void writeDate(java.util.Calendar calendar, boolean checkRef)
```

将 `calendar` 序列化到数据流中，`checkRef` 表示是否在序列化时检查引用。

```
void writeDate(java.sql.Date date)
```

将 `date` 序列化到数据流中。

```
void writeDate(java.util.Date date)
```

将 `date` 序列化到数据流中。

```
void writeDate(java.sql.Date date, boolean checkRef)
```

将 `date` 序列化到数据流中，`checkRef` 表示是否在序列化时检查引用。

```
void writeDate(java.util.Date date, boolean checkRef)
```

将 `date` 序列化到数据流中，`checkRef` 表示是否在序列化时检查引用。

```
void writeDate(java.sql.Time time)
```

将 `time` 序列化到数据流中。

```
void writeDate(java.sql.Time time, boolean checkRef)
```

将 `time` 序列化到数据流中，`checkRef` 表示是否在序列化时检查引用。

```
void writeDate(java.sql.Timestamp time)
```

将 time 序列化到数据流中。

```
void writeDate(java.sql.Timestamp time, boolean checkRef)
```

将 time 序列化到数据流中，checkRef 表示是否在序列化时检查引用。

writeDouble 方法

```
void writeDouble(double d)
```

将浮点数 d 序列化到数据流中。

writeEmpty 方法

```
void writeEmpty()
```

序列化空串到数据流中。

writeEnum 方法

```
void writeEnum(Object obj)
```

将枚举对象 obj 序列化到数据流中。

writeInfinity 方法

```
void writeInfinity(boolean positive)
```

序列化无穷大数到数据流中，positive 为 true 表示正无穷大，为 false 表示负无穷大。

writeInteger 方法

```
void writeInteger(int i)
```

将整数 i 序列化到数据流中。

writeList 方法

```
void writeList(java.util.List list)
```

将 list 序列化到数据流中。

```
void writeList(java.util.List list, boolean checkRef)
```

将 list 序列化到数据流中，checkRef 表示是否在序列化时检查引用。

writeLong 方法

```
void writeLong(java.math.BigInteger l)
```

将大整数 `l` 序列化到数据流中。

```
void writeLong(long l)
```

将长整型数 `l` 序列化到数据流中。

writeMap 方法

```
void writeMap(java.util.Map map)
```

将 `map` 序列化到数据流中。

```
void writeMap(java.util.Map map, boolean checkRef)
```

将 `map` 序列化到数据流中，`checkRef` 表示是否在序列化时检查引用。

writeNaN 方法

```
void writeNaN()
```

将非数序列化到数据流中。

writeNull 方法

```
void writeNull()
```

将 `null` 序列化到数据流中。

writeObject 方法

```
void writeObject(Object object)
```

将对象 `object` 序列化到数据流中。

```
void writeObject(Object object, boolean checkRef)
```

将对象 `object` 序列化到数据流中，`checkRef` 表示是否在序列化时检查引用。

writeString 方法

```
void writeString(char[] s)
```

将字符数组 `s` 序列化到数据流中。


```
void writeString(char[] s, boolean checkRef)
```

将字符数组 `s` 序列化到数据流中，`checkRef` 表示是否在序列化时检查引用。

```
void writeString(String s)
```

将字符串 `s` 序列化到数据流中。

```
void writeString(String s, boolean checkRef)
```

将字符串 `s` 序列化到数据流中，`checkRef` 表示是否在序列化时检查引用。

writeTime 方法

```
void writeTime(java.sql.Time time)
```

将 `time` 序列化到数据流中。

```
void writeTime(java.sql.Time time, boolean checkRef)
```

将 `time` 序列化到数据流中，`checkRef` 表示是否在序列化时检查引用。

HproseFormatter 类

序列化和反序列化帮助类。

serialize 方法

```
static java.io.ByteArrayOutputStream serialize(boolean b)
```

序列化布尔值 `b` 并返回序列化的数据流。

```
static java.io.ByteArrayOutputStream serialize(byte b)
```

序列化字节 `b` 并返回序列化的数据流。

```
static java.io.ByteArrayOutputStream serialize(char c)
```

序列化字符 `c` 并返回序列化的数据流。

```
static java.io.ByteArrayOutputStream serialize(double d)
```

序列化双精度浮点数 `d` 并返回序列化的数据流。

```
static java.io.ByteArrayOutputStream serialize(float f)
```

序列化单精度浮点数 `f` 并返回序列化的数据流。

```
static java.io.ByteArrayOutputStream serialize(int i)
```

序列化整型数 *i* 并返回序列化的数据流。

```
static java.io.ByteArrayOutputStream serialize(long l)
```

序列化长整型数 *l* 并返回序列化的数据流。

```
static java.io.ByteArrayOutputStream serialize(Object obj)
```

以字段模式序列化对象 *obj* 并返回序列化的数据流。

```
static java.io.ByteArrayOutputStream serialize(Object obj, HproseMode mode)
```

以指定模式 *mode* 序列化对象 *obj* 并返回序列化的数据流。

```
static java.io.ByteArrayOutputStream serialize(short s)
```

序列化短整型数 *s* 并返回序列化的数据流。

unserialize 方法

```
static Object unserialize(byte[] data)
```

从字节数组 *data* 以字段模式反序列化数据并返回。

```
static Object unserialize(byte[] data, HproseMode mode)
```

从字节数组 *data* 以指定模式 *mode* 反序列化数据并返回。

```
static Object unserialize(byte[] data, HproseMode mode, Class type)
```

从字节数组 *data* 以指定模式 *mode* 反序列化类型为 *type* 的数据并返回。

```
static Object unserialize(java.io.InputStream stream)
```

从数据流 *stream* 以字段模式反序列化数据并返回。

```
static Object unserialize(java.io.InputStream stream, HproseMode mode)
```

从数据流 *stream* 以指定模式 *mode* 反序列化数据并返回。

```
static Object unserialize(java.io.InputStream stream, HproseMode mode, Class type)
```

从数据流 *stream* 以指定模式 *mode* 反序列化类型为 *type* 的数据并返回。

hprose.client 包

HproseClient 类

Hprose 客户端基类，该类为抽象类。

onError 事件

```
HproseErrorEvent onError
```

该事件当异步调用出错时被执行。

invoke 方法

```
Object invoke(String functionName)
```

同步调用方法 `functionName`，并返回结果。

```
Object invoke(String functionName, Class returnType)
```

同步调用方法 `functionName`，并以 `returnType` 类型返回结果。

```
Object invoke(String functionName, Object[] arguments)
```

同步调用方法 `functionName`，参数为 `arguments`，并返回结果。

```
Object invoke(String functionName, Object[] arguments, boolean byRef)
```

同步调用方法 `functionName`，参数为 `arguments`，通过 `byRef` 来设置是否引用传递，并返回结果。

```
Object invoke(String functionName, Object[] arguments, Class returnType)
```

同步调用方法 `functionName`，参数为 `arguments`，并以 `returnType` 类型返回结果。

```
Object invoke(String functionName, Object[] arguments, Class returnType, boolean byRef)
```

同步调用方法 `functionName`，参数为 `arguments`，通过 `byRef` 来设置是否引用参数传递，并以 `returnType` 类型返回结果。

```
void invoke(String functionName, HproseCallback callback)
```

异步调用方法 `functionName`，当结果返回时回调 `callback`。

```
void invoke(String functionName, HproseCallback callback, Class returnType)
```

异步调用方法 `functionName`，当结果返回时回调 `callback`，结果类型为 `returnType`。

```
void invoke(String functionName, Object[] arguments, HproseCallback callback)
```

异步调用方法 `functionName`，参数为 `arguments`，当结果返回时回调 `callback`。

```
void invoke(String functionName, Object[] arguments, HproseCallback callback, boolean byRef)
```

异步调用方法 `functionName`，参数为 `arguments`，通过 `byRef` 来设置是否引用参数传递，当结果返回时回调 `callback`。

```
void invoke(String functionName, Object[] arguments, HproseCallback callback, Class returnType)
```

异步调用方法 `functionName`，参数为 `arguments`，当结果返回时回调 `callback`，结果类型为 `returnType`。

```
void invoke(String functionName, Object[] arguments, HproseCallback callback, Class returnType, boolean byRef)
```

异步调用方法 `functionName`，参数为 `arguments`，通过 `byRef` 来设置是否引用参数传递，结果返回时回调 `callback`，结果类型为 `returnType`。

useService 方法

```
Object useService(Class type)
```

返回接口类型为 `type` 的远程代理对象。

```
Object useService(Class[] types)
```

返回接口类型为 `types` 的远程代理对象。

```
Object useService(Class[] types, String ns)
```

返回接口类型为 `types`，名称空间为 `ns` 的远程代理对象。

```
Object useService(Class type, String ns)
```

返回接口类型为 `type`，名称空间为 `ns` 的远程代理对象。

```
Object useService(String uri, Class type)
```

返回服务地址为 `uri`，接口类型为 `type` 的远程代理对象。

```
Object useService(String uri, Class[] types)
```

返回服务地址为 `uri`，接口类型为 `types` 的远程代理对象。

```
Object useService(String uri, Class type, String ns)
```

返回服务地址为 `uri`，接口类型为 `type`，名称空间为 `ns` 的远程代理对象。

```
Object useService(String uri, Class[] types, String ns)
```

返回服务地址为 uri，接口类型为 types，名称空间为 ns 的远程代理对象。

HproseHttpClient 类

它是 HproseClient 的子类，它实现了 http 的 Hprose 客户端。

构造方法

```
HproseHttpClient()
```

创建一个 Hprose 的 http 客户端，传输采用字段模式序列化。

```
HproseHttpClient(HproseMode mode)
```

创建一个 Hprose 的 http 客户端，传输采用指定模式 mode 序列化。

```
HproseHttpClient(String uri)
```

创建一个 Hprose 的 http 客户端，并设置服务器的 uri，传输采用字段模式序列化。

```
HproseHttpClient(String uri, HproseMode mode)
```

创建一个 Hprose 的 http 客户端，并设置服务器的 uri，传输采用指定模式 mode 序列化。

getHeader 方法

```
String getHeader(String name)
```

获取 http 标头名为 name 的标头值。

setHeader 方法

```
void setHeader(String name, String value)
```

设置 http 标头，标头名为 name，标头值为 value。

isKeepAlive 方法

```
boolean isKeepAlive()
```

获取是否保持持久连接。

setKeepAlive 方法

```
void setKeepAlive(boolean keepAlive)
```

设置是否保持持久连接。

getKeepAliveTimeout 方法

```
int getKeepAliveTimeout()
```

获取持久连接超时值。

setKeepAliveTimeout 方法

```
void setKeepAliveTimeout(int keepAliveTimeout)
```

设置持久连接超时值。

getProxyHost 方法

```
String getProxyHost()
```

获取代理主机。

getProxyPass 方法

```
String getProxyPass()
```

获取代理密码。

getProxyPort 方法

```
int getProxyPort()
```

获取代理端口。

getProxyUser 方法

```
String getProxyUser()
```

获取代理用户名。

setProxyHost 方法

```
void setProxyHost(String proxyHost)
```

设置代理主机。

setProxyPass 方法

```
void setProxyPass(String proxyPass)
```

设置代理密码。

setProxyPort 方法

```
void setProxyPort(int proxyPort)
```

设置代理端口。

setProxyUser 方法

```
void setProxyUser(String proxyUser)
```

设置代理用户名。

useService 方法

```
boolean useService(String uri)
```

设置服务地址的 uri，如果地址格式正确返回 true，否则返回 false。

hprose.server 包

HproseServiceEvent 接口

onAfterInvoke 方法

```
void onAfterInvoke(String name, Object[] args, boolean byRef, Object result)
```

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

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

onBeforeInvoke 方法

```
void onBeforeInvoke(String name, Object[] args, boolean byRef)
```

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

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

onSendError 方法

```
void onSendError(String error)
```

当服务器端调用发生错误, 或者在 `onBeforeInvoke`、`onAfterInvoke` 事件中抛出异常时, 该事件被触发。您可以在该事件中作日志记录, 但该事件中不应再抛出任何异常。

onSendHeader 方法

```
void onSendHeader()
```

当服务器返回响应头部时, `onSendHeader` 事件会被触发。在该事件中, 您可以发送您自己的头信息, 例如设置 Cookie。该事件中不应抛出任何异常。

HttpContext 类

该类表示 http 上下文。

构造方法

```
HttpContext(javax.servlet.http.HttpServletRequest request, javax.servlet.http.HttpServletResponse response, javax.servlet.ServletConfig config, javax.servlet.ServletContext application)
```

创建 http 上下文。

getApplication 方法

```
javax.servlet.ServletContext getApplication()
```

获取全局应用对象。

getConfig 方法

```
javax.servlet.ServletConfig getConfig()
```

获取配置对象。

getRequest 方法

```
javax.servlet.http.HttpServletRequest getRequest()
```

获取请求对象。

getResponse 方法

```
javax.servlet.http.HttpServletResponse getResponse()
```


获取响应对象。

getSession 方法

```
javax.servlet.http.HttpSession getSession()
```

获取会话对象。

getSession 方法

```
javax.servlet.http.HttpSession getSession(boolean create)
```

获取会话对象，create 参数决定是否创建新会话。

HproseHttpMethods 类

HproseMethods 的子类。

HproseService 类

Hprose 服务器端基类，该类为抽象类。

add 方法

```
void add(Object obj)
```

添加 obj 对象上绑定的 obj 所在类直接声明的所有 public 实例方法到发布方法集合。

如果 obj 上方法有重载的情况，则所有参数个数不相同的方法都会被添加。如果有两个或多个参数个数也相同的方法，则只有其中一个方法会被添加。因此，用此方法发布的对象中应避免出现参数个数相同的重载情况。

```
void add(Object obj, Class type)
```

添加 obj 对象上绑定的 type 类上直接声明的所有 public 实例方法到发布方法集合。

使用该方法请注意上面提到的重载冲突的情况。

```
void add(Object obj, Class type, String aliasPrefix)
```

添加 obj 对象上绑定的 type 类上直接声明的所有 public 实例方法到发布方法集合，并设置这些方法的别名前缀为 aliasPrefix。

使用该方法请注意上面提到的重载冲突的情况。

```
void add(Object obj, String aliasPrefix)
```

添加 obj 对象上绑定的 obj 所在类直接声明的所有 public 实例方法到发布方法集合，并设置这些方法的别名前缀为 aliasPrefix。

使用该方法请注意上面提到的重载冲突的情况。

```
void add(java.lang.reflect.Method method, Object obj, String aliasName)
```

添加 method 方法到发布方法集合，将其绑定于对象 obj，并设置其别名为 aliasName。

```
void add(String methodName, Class type)
```

添加 type 类上方法名为 methodName 的方法到发布方法集合。

如果在 type 类上有多个方法名为 methodName 的方法，则所有参数个数不相同的方法都会被添加。如果有两个或多个参数个数也相同的方法，则只有其中一个方法会被添加。因此，用此方法发布的对象中应避免出现参数个数相同的重载情况。

```
void add(String methodName, Class type, Class[] paramTypes)
```

添加 type 类上方法名为 methodName，参数类型为 paramTypes 的方法到发布方法集合。

使用该方法可以避免上面所提到的重载冲突的情况。

```
void add(String methodName, Class type, Class[] paramTypes, String aliasName)
```

添加 type 类上方法名为 methodName，参数类型为 paramTypes 的方法到发布方法集合，并设置其别名为 aliasName。

使用该方法可以避免上面所提到的重载冲突的情况。

```
void add(String methodName, Class type, String aliasName)
```

添加 type 类上方法名为 methodName 的方法到发布方法集合，并设置其别名为 aliasName。

使用该方法请注意上面提到的重载冲突的情况。

```
void add(String methodName, Object obj)
```

添加 obj 对象上方法名为 methodName 的方法到发布方法集合。

如果在 obj 对象所在类上有多个方法名为 methodName 的方法，则所有参数个数不相同的方法都会被添加。如果有两个或多个参数个数也相同的方法，则只有其中一个方法会被添加。因此，用此方法发布的对象中应避免出现参数个数相同的重载情况。

```
void add(String methodName, Object obj, Class[] paramTypes)
```

添加 obj 对象上方法名为 methodName，参数类型为 paramTypes 的方法到发布方法集合。

使用该方法可以避免上面所提到的重载冲突的情况。

```
void add(String methodName, Object obj, Class[] paramTypes, String aliasName)
```

添加 obj 对象上方法名为 methodName，参数类型为 paramTypes 的方法到发布方法集合，并设置其别名为 aliasName。

使用该方法可以避免上面所提到的重载冲突的情况。

```
void add(String methodName, Object obj, String aliasName)
```

添加 obj 对象上方法名为 methodName 的方法到发布方法集合，并设置其别名为 aliasName。

使用该方法请注意上面提到的重载冲突的情况。

```
void add(String[] methodNames, Class type)
```

添加 type 类上方法名为 methodNames 的一组方法到发布方法集合。

如果所发布的方法在 type 类上有重载，则所有参数个数不相同的方法都会被添加。如果有两个或多个参数个数也相同的方法，则只有其中一个方法会被添加。因此，用此方法发布的对象中应避免出现参数个数相同的重载情况。

```
void add(String[] methodNames, Class type, String aliasPrefix)
```

添加 type 类上方法名为 methodNames 的一组方法到发布方法集合，并设置这些方法的别名前缀为 aliasPrefix。

使用该方法请注意上面提到的重载冲突的情况。

```
void add(String[] methodNames, Class type, String[] aliasNames)
```

添加 type 类上方法名为 methodNames 的一组方法到发布方法集合，并设置这些方法的别名为 aliasNames。

使用该方法请注意上面提到的重载冲突的情况。

```
void add(String[] methodNames, Object obj)
```

添加 obj 对象上方法名为 methodNames 的一组方法到发布方法集合。

如果所发布的方法在 obj 对象上有重载，则所有参数个数不相同的方法都会被添加。如果有两个或多个参数个数也相同的方法，则只有其中一个方法会被添加。因此，用此方法发布的对象中应避免出现参数个数相同的重载情况。

```
void add(String[] methodNames, Object obj, String aliasPrefix)
```

添加 obj 对象上方法名为 methodNames 的一组方法到发布方法集合，并设置这些方法的别名前缀为 aliasPrefix。

使用该方法请注意上面提到的重载冲突的情况。

```
void add(String[] methodNames, Object obj, String[] aliasNames)
```

添加 obj 对象上方法名为 methodNames 的一组方法到发布方法集合，并设置这些方法的别名为 aliasNames。

使用该方法请注意上面提到的重载冲突的情况。

```
void add(Class type)
```

添加 type 类上直接声明的所有 public 静态方法到发布方法集合。

如果 type 类上方法有重载的情况，则所有参数个数不相同的方法都会被添加。如果有两个或多个参数个数也相同的方法，则只有其中一个方法会被添加。因此，用此方法发布的对象中应避免出现参数个数相同的重载情况。

```
void add(Class type, String aliasPrefix)
```

添加 type 类上直接声明的所有 public 静态方法到发布方法集合，并设置这些方法的别名前缀为 aliasPrefix。

使用该方法请注意上面提到的重载冲突的情况。

addMissingMethod 方法

```
void addMissingMethod(String methodName, Class type)
```

添加 type 类上名称为 methodName 的方法为默认方法。该方法对参数有明确限制，参数为 2 个，按从左到右的顺序分别为 String，Object[]类型。所以不会有重载冲突问题。

当客户端调用服务器端的方法时，如果服务器端在发布的方法集合中找不到客户端调用的方法，则会执行该方法。客户端调用的方法名会作为该方法的第一个参数被传入，客户端调用的方法参数会作为该方法的第二个参数被传入。

```
void addMissingMethod(String methodName, Object obj)
```

添加 obj 对象上名称为 methodName 的方法为默认方法。该方法对参数有明确限制，参数为 2 个，按从左到右的顺序分别为 String，Object[]类型。所以不会有重载冲突问题。

用该方法发布的方法功能跟上面的功能一致。

以上两个方法只有最后一次添加的有效。

getEvent 方法

```
HproseServiceEvent getEvent()
```

获取事件。

setEvent 方法

```
void setEvent(HproseServiceEvent event)
```

设置事件。

getGlobalMethods 方法

```
HproseMethods getGlobalMethods()
```

获取全局方法集合。

setGlobalMethods 方法

```
void setGlobalMethods(HproseMethods methods)
```

设置全局方法集合。

getMode 方法

```
HproseMode getMode()
```

获取序列化模式。

setMode 方法

```
void setMode(HproseMode mode)
```

设置序列化模式。

isDebugEnabled 方法

```
boolean isDebugEnabled()
```

获取是否允许调试。

setDebugEnabled 方法

```
void setDebugEnabled(boolean enabled)
```

设置是否允许调试。

HproseHttpService 类

它是 HproseService 的子类，它实现了 http 的 Hprose 服务器端。

getCurrentContext 方法

```
static HttpContext getCurrentContext()
```

获取当前 http 上下文。

handle 方法

```
void handle(HttpContext httpContext)
```

执行处理请求，参数为 http 上下文。

```
void handle(HttpContext httpContext, HproseHttpMethods methods)
```

执行处理请求，参数为 http 上下文，methods 是服务器在当前请求中发布的方法。

isEnabled 方法

```
boolean isEnabled()
```

获取是否允许 GET 方法。

setEnabled 方法

```
void setEnabled(boolean enabled)
```

设置是否运行 GET 方法。

isP3pEnabled 方法

```
boolean isP3pEnabled()
```

获取是否允许 P3P 特征。

setP3pEnabled 方法

```
void setP3pEnabled(boolean enabled)
```

设置是否允许 P3P 特征。

HproseServlet 类

默认 Hprose 的 Servlet 类，其用法请参考手册正文内容。