**Hasor 使用手册**

**手册版本：0.0.2**

**软件版本：0.0.3**

**作者：赵永春(zyc@hasor.net)**

**日期：2013-09-24**

目录

[第一章 介绍 6](#_Toc377371806)

[1.1 内容引导 7](#_Toc377371807)

[1.2 开源协议 8](#_Toc377371808)

[1.3 内置组建及其授权协议 8](#_Toc377371809)

[1.4 获取和贡献 9](#_Toc377371810)

[1.5 类库引用 9](#_Toc377371811)

[1.6 约定优于配置(COC原则) 9](#_Toc377371812)

[第二章 起步 10](#_Toc377371813)

[2.1 选用IDE 10](#_Toc377371814)

[2.2 创建项目 10](#_Toc377371815)

[2.3 启动Hasor 11](#_Toc377371816)

[2.4 第一个模块(HelloWord) 11](#_Toc377371817)

[2.5 Bean 11](#_Toc377371818)

[2.5.1 注册Bean 11](#_Toc377371819)

[2.5.2 获取Bean 12](#_Toc377371820)

[2.5.3 单例Bean 12](#_Toc377371821)

[2.5.4 容器之外的Bean 12](#_Toc377371822)

[2.6 依赖注入(IoC) 12](#_Toc377371823)

[2.6.1 字段方式注入 12](#_Toc377371824)

[2.6.2 属性方式注入 13](#_Toc377371825)

[2.6.3 构造方法注入 13](#_Toc377371826)

[2.7 Aop拦截器(MethodInterceptor) 13](#_Toc377371827)

[2.7.1 方法级拦截器 14](#_Toc377371828)

[2.7.2 类级拦截器 14](#_Toc377371829)

[2.7.3 全局拦截器 14](#_Toc377371830)

[2.7.4 拦截范围 15](#_Toc377371831)

[2.8 事件的抛出和监听(Event) 15](#_Toc377371832)

[2.8.1 抛出和监听 16](#_Toc377371833)

[2.8.2 同步事件 16](#_Toc377371834)

[2.8.3 异步事件 16](#_Toc377371835)

[2.9 一个模块依赖另外一个模块 16](#_Toc377371836)

[2.9.1 强依赖 17](#_Toc377371837)

[2.9.2 弱依赖 18](#_Toc377371838)

[2.9.3 依赖反制 18](#_Toc377371839)

[2.10 使用配置文件 19](#_Toc377371840)

[2.11 Web项目 21](#_Toc377371841)

[2.12 Web开发 23](#_Toc377371842)

[2.12.1 HttpServlet 23](#_Toc377371843)

[2.12.2 Filter 23](#_Toc377371844)

[2.12.3 Session监听器(HttpSessionListener) 24](#_Toc377371845)

[2.12.4 Servlet启动监听器(ServletContextListener) 24](#_Toc377371846)

[2.12.5 截获服务器异常 24](#_Toc377371847)

[2.13 Web-MVC 25](#_Toc377371848)

[2.13.1 Action 25](#_Toc377371849)

[2.13.2 获取Request和Response 26](#_Toc377371850)

[2.13.3 RESTful映射 26](#_Toc377371851)

[2.13.4 返回Json数据 27](#_Toc377371852)

[2.13.5 Action结果处理 28](#_Toc377371853)

[2.14 打包Web资源(WebJars) 28](#_Toc377371854)

[2.14.1 Jar包中的Web资源。 29](#_Toc377371855)

[2.14.2 Zip压缩包中的Web资源。 29](#_Toc377371856)

[2.14.3 指定的目录中加载资源Web资源。 30](#_Toc377371857)

[（未完待续） 30](#_Toc377371858)

[第三章 架构 31](#_Toc377371859)

[3.1 技术选型 31](#_Toc377371860)

[3.2 总体架构 31](#_Toc377371861)

[3.3 分层设计 31](#_Toc377371862)

[3.4 生命周期 31](#_Toc377371863)

[3.4.1 init阶段 31](#_Toc377371864)

[3.4.2 start阶段 31](#_Toc377371865)

[3.4.3 stop阶段 31](#_Toc377371866)

[3.5 模块&插件 31](#_Toc377371867)

[3.5.1 运行状态 31](#_Toc377371868)

[3.5.2 依赖 31](#_Toc377371869)

[3.5.3 插件 31](#_Toc377371870)

[3.6 事件 31](#_Toc377371871)

[3.7 环境变量 31](#_Toc377371872)

[3.8 Xml解析 31](#_Toc377371873)

[3.9 Web支持 31](#_Toc377371874)

[3.10 JDBC支持 31](#_Toc377371875)

[第四章 核心技术 31](#_Toc377371876)

[4.1 Core部分 31](#_Toc377371877)

[4.1.1 Bean 31](#_Toc377371878)

[4.1.2 IoC(JSR-330) 31](#_Toc377371879)

[4.1.3 Aop 31](#_Toc377371880)

[4.1.4 Event 31](#_Toc377371881)

[4.1.5 Plugin 31](#_Toc377371882)

[4.1.6 Module 31](#_Toc377371883)

[4.1.7 Guice 31](#_Toc377371884)

[4.1.8 Cache 31](#_Toc377371885)

[4.1.9 读取配置文件 31](#_Toc377371886)

[4.1.10 配置文件监听器 31](#_Toc377371887)

[4.1.11 解析Xml文件 31](#_Toc377371888)

[4.2 Web部分 31](#_Toc377371889)

[4.2.1 Controller 31](#_Toc377371890)

[4.2.2 Restful 31](#_Toc377371891)

[4.2.3 Result 31](#_Toc377371892)

[4.2.4 Servlet3.0 31](#_Toc377371893)

[4.2.5 Request请求资源 31](#_Toc377371894)

[4.2.6 Hasor JSP标签库 31](#_Toc377371895)

[4.3 JDBC部分 31](#_Toc377371896)

[4.3.1 增/删/改/查 32](#_Toc377371897)

[4.3.2 参数化SQL 32](#_Toc377371898)

[4.3.3 单值查询 32](#_Toc377371899)

[4.3.4 调用存储过程 32](#_Toc377371900)

[4.3.5 事务控制 32](#_Toc377371901)

[4.3.6 事务传播行为 32](#_Toc377371902)

[4.3.7 多数据源 32](#_Toc377371903)

[4.3.8 多数据源的事务控制 32](#_Toc377371904)

[第五章 内核开发 32](#_Toc377371905)

[5.1 Core部分 32](#_Toc377371906)

[5.1.1 启动内核 32](#_Toc377371907)

[5.1.2 添加模块 32](#_Toc377371908)

[5.1.3 注册Bean 32](#_Toc377371909)

[5.1.4 注册Aop 32](#_Toc377371910)

[5.1.5 ApiBinder 32](#_Toc377371911)

[5.1.6 扫描类路径 32](#_Toc377371912)

[5.1.7 类型绑定与获取 32](#_Toc377371913)

[5.1.8 事件 32](#_Toc377371914)

[5.1.9 获取AppContext 32](#_Toc377371915)

[5.1.10 环境变量 32](#_Toc377371916)

[5.1.11 创建Guice 32](#_Toc377371917)

[5.2.12 读取配置文件 32](#_Toc377371918)

[5.2.13 解析Xml文件 32](#_Toc377371919)

[5.2 Web部分 32](#_Toc377371920)

[5.2.1 启动Web支持内核 32](#_Toc377371921)

[5.2.2 注册HttpServlet 32](#_Toc377371922)

[5.2.3 注册Filter 32](#_Toc377371923)

[5.2.4 注册ServletContextListener 32](#_Toc377371924)

[5.2.5 WebApiBinder 32](#_Toc377371925)

[5.2.6 获取ServletContext 32](#_Toc377371926)

[5.2.7 获取Request/Response 32](#_Toc377371927)

[5.3 JDBC部分 32](#_Toc377371928)

[5.3.1 脱离Hasor使用JDBC 32](#_Toc377371929)

[5.3.2 DataSourceHelper 32](#_Toc377371930)

[第六章 配置文件详解 32](#_Toc377371931)

[6.1 Core部分 32](#_Toc377371932)

[6.2 Web部分 32](#_Toc377371933)

[6.3 JDBC部分 32](#_Toc377371934)

[第七章 API参考手册 33](#_Toc377371935)

[第八章 附录 33](#_Toc377371936)

[8.1 约定 33](#_Toc377371937)

# 第一章 介绍

Hasor是一款开源的Java应用开发框架。它围绕Guice为核心创建一系列的插件组合而成。使用Hasor会加速软件软件开发，并降低开发成本。目前Hasor包含了：Core、Web、JDBC三个主要的软件包，总共大约14个插件。其中Hasor的内核只有不到50个类。

Hasor的本质与Struts、Hibernate等单层框架不同，它可以提供一个以统一、高效的、友好的方式构造整个应用程序。并且可以将这些单层框架建立起一个连贯的体系，可以说Hasor是一个搭建开发环境的框架。

特点：

* **微内核****：**Hasor有着不到50个类文件组成的微内核，我们可以看到的大部分功能都是在内核基础上由不同插件提供的。
* **轻量化：**利用Guice3.0强大的DI支持使得Hasor的运行效率很高，而且具有很小的身材(算上依赖才5个Jar不到2MB)。
* **兼容性：**由于Hasor仅仅是一个轻量化容器，这又使得它可以很方便的和任何框架整合到一起。例如：通过Hasor-Web模块可以完美支持Web开发。
* **注解化：**Hasor的绝大部分插件都提供注解配置，因此可以说Hasor的开发是基于注解化的。
* **容 器：**Hasor包含并管理每个模块对象的配置和生命周期。
* **约定优于配置(COC)：**可以完全不需要配置Hasor就进行系统开发，配置文件功能的可以完全留给业务系统使用。

设计思想：

“微内核+插件”是Hasor的主体设计思想，Hasor将内核称为Context。创建Context 就是在创建Guice。通过启动的不同阶段提供生命周期支持。围绕生命周期Hasor提供统一插件接口。从而通过插件丰富Hasor功能，而作为插件是可以随时被剔除的。

主要模块：

Hasor-Core

Hasor的核心软件包，几乎所有Hasor扩展模块都会依赖到它。该软件包中包含了：生命周期管理、配置文件解析、事件、容器、IoC/Aop、Bean。并且通过Guice提供JSR-330标准的兼容。此外它还包括了六个插件，通过插件可以简化您在开发中的工作量。

Hasor-Web

Hasor 为支持 Web 特性而建立的外壳，分为核心和插件两个部分。使用 Hasor-Web 可以通过编码形式动态注册Servlet/Filter ，Hasor-Web为它们建立了统一的 Dispatcher。

同时它具备 Action、Restful、拦截器、Servlet3.0等常见Web开发技术。它不光是Hasor对Web的补充，更是一个全面的Web开发框架。

Hasor-JDBC

该模块是SpringJDBC轻量化的重构，它只有不到100个类就完成了SpringJDBC几乎所有功能。它的80%以上的代码来自SpringJDBC。而且Hasor-JDBC的关键部分可以脱离Hasor独立使用。

## 1.1 内容引导

**第一章：介绍**

用以说明介绍Hasor框架的设计目标、目的以及设计思想。同时说明了Hasor所使用的开源协议、类库以及如何贡献和索取Hasor代码。

**第二章：起步**

从使用角度指导读者一步一步搭建开发环境，并通过例子教会读者使用Hasor进行项目开发。本章内容将会逐步深入，在本章的最后一段会介绍如何开发一个简单的Hasor插件。

掌握了本章内容后您就可以使用Hasor在实际项目中运用了。如果您对实现原理感兴趣可以继续阅读第三章内容。

**第三章：架构**

本章将会从架构层面介绍Hasor的设计思想以及内核的分层设计，想要了解Hasor实现原理的朋友推荐阅读本章内容。

**第四章：核心技术**

本章将重点介绍Hasor的核心接口的使用，在本章中您会重温部分第二章内容。同时本章将会介绍给您，如何通过其它方式实现第二章中的功能。阅读本章有助于您了解Hasor。如果您正在基于Hasor开发插件那么本章将会是最佳的选择。

**第五章：微内核开发**

Hasor是“微内核+插件”这种形式的，Hasor的内核只有不到50个类。且Hasor的内核对运行环境没有要求。您可以基于这50个类的微内核来打造属于自己的开发环境，这对于一些需要打造专有开发环境的企业而言是一个不错的选择。阅读本章将指导您如何基于Hasor内核制定一款专属开发环境。

**第六章：配置文件详解**

Hasor是遵循COC原则设计的，因此几乎所有配置都做了默认设置。正因为如此您在使用Hasor开发项目时几乎不需要编写任何配置文件“零配置”。但是当您需要更改默认设置时可以参考本章内容。

**第七章：API参考手册**

本章将一一详细讲解Hasor所有接口的方法如何使用，同时本章还会提供一个常用API速查表。

**第八章：附录**

其它可能需要本手册携带的信息会在该章节给出。

## 1.2 开源协议

作为开源发布Hasor使用是Apache License 2.0协议。

## 1.3 内置组建及其授权协议

More

More是我在2008年之后构建的第一款开源框架，当时以失败告终。而后More的大部分代码都被拆除或者改造。目前保留下来的只有ClassCode、Xml以及一部分位于util包中的工具类。

目前util包中的工具类大部分也已被Apsche的commons-lang、commons-beans项目中的代码所替代。该项目受Apache License 2.0协议保护。

ASM 3.0

ASM是一款字节码框架，使用它可以动态的创建或修改java类文件。配合ClassLoader可以装载修改之后的类Hibernate、Spring都曾使用过它。该框架的部分完成代码位于org.more.asm软件包中，Hasor并没有使用到这个软件包。并且ClassCode组建作为More项目保留组建而存在。软件地址：(<http://www.objectweb.org/asm>)。

Ognl

Ognl是一款表达式解析引擎代码位于org.more.ognl软件包中，Hasor并没有使用到这个软件包。软件地址：(<http://commons.apache.org/proper/commons-ognl/>)

这部分内容是受Apache License 2.0协议保护。

apache-commons-lang 2.6、apache-commons-beans 1.7

org.more.convert软件包的内容是来源于apache-commons-beans-1.7，org.more.util的大部分代码是来源于apache-commons-lang-2.6。Hasor并不引用这两个软件包，但是由于org.more中包含了相关代码因此在这里需要加以说明并且列出其授权信息。这部分内容是受Apache License 2.0协议保护。

org.more.json软件包

该软件包是来源于org.eclipse.jetty.util\_8.1.3.v20120522.jar，位于Eclipse eclipse-sdk-4.2.2-win32软件中，属于Eclipse 4.2.2的一部分。Hasor-mvc项目依赖这部分功能实现Json数据转换。该代码受到受Apache License 2.0协议以及Eclipse Public License协议共同保护。

--开源协议--

Apache License 2.0协议： (<http://www.apache.org/licenses/LICENSE-2.0>)

Eclipse Public License协议： (<http://www.eclipse.org/legal/epl-v10.html>)

ASM3.0协议： (<http://asm.ow2.org/license.html>)

## 1.4 获取和贡献

目前Hasor的代码托管于Github(<https://github.com/zycgit/hasor>)，您可以通过Git客户端获取到Hasor的最新代码。可以在(<http://msysgit.github.io/>)上获取到最新的Git客户端。当您对Hasor有一个更好的改进或想法，可以通过([zyc@hasor.net](mailto:zyc@hasor.net))邮箱联系到我也可以通过(<https://github.com/zycgit/hasor/pulls>)递交您的代码。

当您发现Hasor的漏洞和不足可以通过Email联系我。或者在OSChina上发表技术问答。Issues(<https://github.com/zycgit/hasor/issues>)也是一个发表问题的渠道。或者加入QQ群(293401803)我会为你解答疑问。

项目主页为：<http://www.hasor.net/> [尚在建设中...]

## 1.5 类库引用

依赖：Hasor依赖以下4个软件包：

slf4j-api-1.7.5.jar (负责输出Hasor产生的日志)

guice-3.0.jar (Guice3.0，负责提供DI相关的支持)

aopalliance-1.0.jar (Aop联盟API，Guice依赖)

javax.inject-1.jar (JSR-330标准，Guice依赖)

日志：如果使用log4j作为日志组建那么需要加入以下两个jar文件，log4j.xml是参考：

log4j-1.2.17.jar (log4j库文件)

slf4j-log4j12-1.7.2.jar (slf4j-log4j的适配器)

log4j.xml (参考的log4j配置文件)

## 1.6 约定优于配置(COC原则)

约定优于配置(Convention Over Configuration)是一个简单的概念。系统，类库，框架应该假定合理的默认值，而非要求提供不必要的配置。流行的框架如 Ruby on Rails2 和 EJB3 已经开始坚持这些原则，以对像原始的EJB 2.1规范那样的框架的配置复杂度做出反应。一个约定优于配置的例子就像EJB3持久化，将一个特殊的Bean持久化，你所需要做的只是将这个类标注为@Entity。 框架将会假定表名和列名是基于类名和属性名。系统也提供了一些钩子，当有需要的时候你可以重写这些名字，但是，在大部分情况下，你会发现使用框架提供的默认值会让你的项目运行的更快。

在Hasor不鼓吹“零配置”、“零注解”、“零Xml”，但是Hasor会把最简的开发体验作为首要准则。在使用Hasor开发项目时你会很少接触到配置。大多数都只是标记一个注解了事。只有当上述方式没有办法达到目的的情况下Hasor才会借助配置文件加以配置说明。

使用Hasor作为开发框架的时候可能会发现，你甚至都不需要对Hasor进行任何配置就可以进行开发工作。

# 第二章 起步

## 2.1 选用IDE

Hasor推荐你使用Eclipse Standard或者Eclipse IDE for Java EE Developers作为开发环境。前者是Eclipse的基本版，后者为Web开发版本。您也可以根据自己的喜好制定一款Eclipse开发环境。

Eclipse Standard 4.3下载地址：

(<http://eclipse.org/downloads/packages/eclipse-standard-43/keplerr>)

Eclipse IDE for Java Developers下载地址：

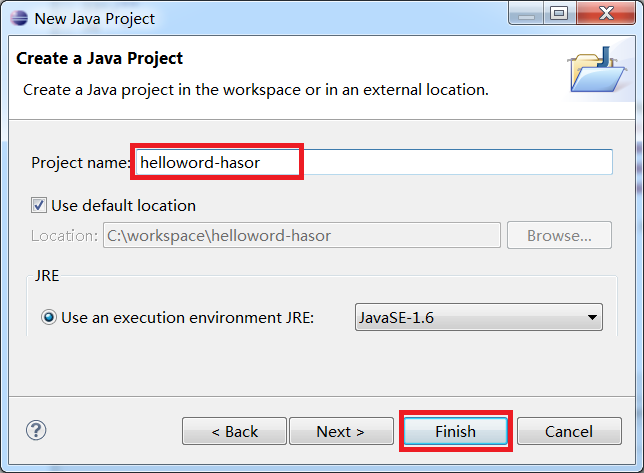
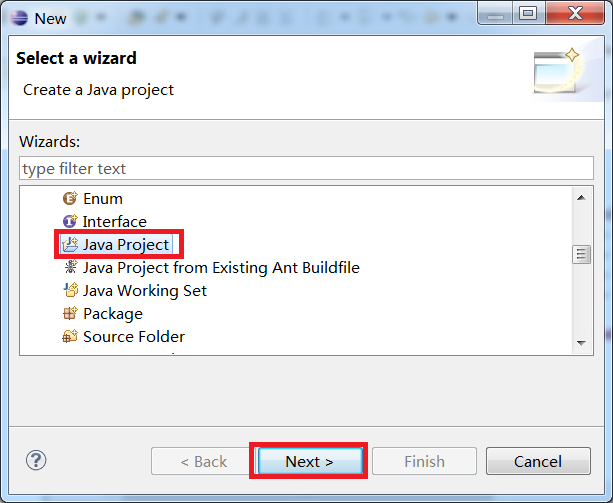
(<http://eclipse.org/downloads/packages/eclipse-ide-java-developers/keplerr>)

Eclipse IDE for Java EE Developers下载地址：

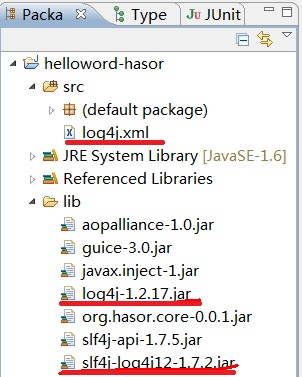
(<http://eclipse.org/downloads/packages/eclipse-ide-java-ee-developers/keplerr>)

## 2.2 创建项目

Step 1：创建项目



Step 2：加入Hasor的Jar包，并引入类路径，图1。

图1： 图2：

Step 3：使用log4j处理Hasor输出日志，加入下面三个文件。如图2。

Log4j.xml (参考的log4j配置文件)

log4j-1.2.17.jar (log4j库文件)

slf4j-log4j12-1.7.2.jar (slf4j-log4j的适配器)

## 2.3 启动Hasor

启动Hasor：

**import** java.io.IOException;

**import** net.hasor.core.context.AnnoStandardAppContext;

**public** **class** HelloHasor {

**public** **static** **void** main(String[] args) **throws** IOException {

//创建Hasor环境对象

AnnoStandardAppContext context = **new** AnnoStandardAppContext ();

context.start();//启动Hasor容器，必须调用

}

}

## 2.4 第一个模块(HelloWord)

下面代码展示了如何编写一个Hasor模块。当Hasor容器启动时模块的init方法会被调用。下面这个模块会在控制台打印一条消息。[<第三章：模块(3.1节-定义)>](#_3.1__定义)

@Module/\*声明模块\*/

**public** **class** FirstModule **extends** AbstractHasorModule {

**public** **void** init(ApiBinder binder) {

System.*out*.println("this is first module.");

}

}

再次启动项目即可看到控制台打印的“this is first module.”

## 2.5 Bean

Hasor定义任何类都可以被视为Bean，有名字的Bean被称为注册Bean。

**public** **class** CustomBean {

**public** **void** foo() {

System.*out*.println("invoke CustomBean.foo");

}

}

### 2.5.1 注册Bean

注册Bean是一种具有名字的JavaBean。同时它还会被Hasor容器主动管理。

方式一：使用@Bean注解将Bean注册到Hasor容器中。

@Bean("myBean")/\*@Bean注解还可以定义多个名字\*/

**public** **class** CustomBean {

……

方式二：在模块的init方法中使用ApiBinder接口以代码的方式注册Bean。如下代码：

@Module

**public** **class** FirstModule **extends** AbstractHasorModule {

**public** **void** init(ApiBinder binder) {

apiBinder.newBean("myBean").bindType(CustomBean.**class**);

}

}

### 2.5.2 获取Bean

CustomBean b1 = context.getBean("myBean");//根据名字获取

CustomBean b2 = context.getInstance(CustomBean.**class**);//根据类型

### 2.5.3 单例Bean

通过JSR-330标准API在类上标记“javax.inject.Singleton”注解将Bean声明为单例。

@Bean("myBean")

@Singleton/\*声明单例\*/

**public** **class** CustomBean {

……

### 2.5.4 容器之外的Bean

通过和注册Bean的一些比较可以有一个比较深刻的印象：

|  |  |  |
| --- | --- | --- |
| 项目 | 注册Bean | 非注册Bean |
| 具有名称 | 是 | 否 |
| 根据名称创建实例 | 是 | 否 |
| 根据类型创建实例 | 是 | 是 |
| Singleton单例 | 是 | 是 |
| Aop拦截器 | 是 | 是 |
| IoC依赖注入 | 是 | 是 |
| 容器主动管理 | 是 | 否 |

Bean相关的介绍可以查看[<第四章：环境支持(4.5节-Bean服务(@Bean))>](#_4.5__Bean服务(@Bean)_1)以了解更多信息。

## 2.6 依赖注入(IoC)

Hasor通过小巧的Google Guice3.0作为其DI容器以支持依赖注入的需要。有关Guice的更多信息查看[<第七章：Guice>](#_Guice)。

同时由于Google Guice3.0是JSR-330标准的实现。这就意味着任何根据JSR-330依赖注入标准所编写的Bean在Hasor下都可以很好的运行。目前Spring在3.1版本中也加入了JSR-330标准的支持。

### 2.6.1 字段方式注入

**public** **class** TestBean {

@Inject//将CustomBean注入到TestBean被注入的Bean可以不具备get/set

**private** CustomBean customBean = **null**;

**public** **void** callFoo() { **this**.customBean.foo(); }

}

### 2.6.2 属性方式注入

**public** **class** TestBean {

**private** CustomBean customBean = **null**;

@Inject

**public** **void** setCustomBean(CustomBean customBean){

**this**.customBean = customBean;

}

}

### 2.6.3 构造方法注入

**public** **class** TestBean2 {

**private** CustomBean customBean = **null**;

@Inject/\*标记到构造方法上\*/

**public** TestBean2(CustomBean customBean) {

**this**.customBean = customBean;

}

**public** **void** callFoo() { **this**.customBean.foo(); }

}

## 2.7 Aop拦截器(MethodInterceptor)

提示：Haosr不直接提供线程安全的拦截器，拦截器的线程安全由拦截器本身实现决定。

下面是三个拦截器的实现：MethodInterceptor接口是来源于Aop联盟aopalliance-1.0.jar。

**public** **class** AopInterceptor\_1 **implements** MethodInterceptor {

**public** Object invoke(MethodInvocation arg0) **throws** Throwable {

System.*out*.println("aop1:" + arg0.getMethod());

**return** arg0.proceed();//拦截器，用在方法级别

}

}

**public** **class** AopInterceptor\_2 **implements** MethodInterceptor {

**public** Object invoke(MethodInvocation arg0) **throws** Throwable {

System.*out*.println("aop2:" + arg0.getMethod());

**return** arg0.proceed();//拦截器，用在类级别

}

}

**public** **class** AopInterceptor\_3 **implements** MethodInterceptor {

**public** Object invoke(MethodInvocation arg0) **throws** Throwable {

System.*out*.println("aop3:" + arg0.getMethod());

**return** arg0.proceed();//拦截器，用在全局

}

}

### 2.7.1 方法级拦截器

@Bean("jobBean")

**public** **class** JobBean {

@Before(AopInterceptor\_1.**class**)// @Before注解声明方法需要拦截器

**public** String println(String msg) { **return** "println->"+msg; }

}

### 2.7.2 类级拦截器

@Bean("jobBean")

@Before(AopInterceptor\_2.**class**)//类级别拦截器

**public** **class** JobBean {

@Before(AopInterceptor\_1.**class**)//方法级别拦截器

**public** String println(String msg) {

**return** "println->" + msg;

}

**public** String foo(String msg) {

**return** "foo->" + msg;

}

}

注：类级拦截器鈺方法级拦截器生命方式上仅仅是位置不一样。

### 2.7.3 全局拦截器

提示：全局拦截器的定义是在模块的init方法中通过ApiBinder接口注册。

下面是一段简单的完整代码：[<第四章：环境支持(4.6节-Aop拦截器服务(@Before))>](#_4.6__Aop拦截器服务(@Before))

@Module

**public** **class** AopModule **extends** AbstractHasorModule {

//在@Bean注解处理之前注册全局拦截器，全局拦截器的优先级顺序会混乱

**public** **void** configuration(ModuleSettings info) {

info.afterMe(AnnoSupportModule.**class**);//全局->类级->方法级

}

**public** **void** init(ApiBinder apiBinder) {

//任意类的任意方法

apiBinder.getGuiceBinder().bindInterceptor(

Matchers.*any*(), Matchers.*any*(), **new** AopInterceptor\_3());

/\* 注意：如果应用程序中定义了多个拦截器。

\* 任意匹配会导致其他拦截其被AopInterceptor\_3代理

\* 可以使用下面这段代码在任意类中排除拦截器 \*/

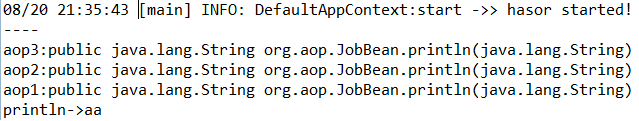
Matcher matcher = Matchers.*not*(

Matchers.*subclassesOf*(MethodInterceptor.**class**));

}

}

获取被代理的JobBean对象并调用foo方法查看运行结果：



上面输出的日志可以看出三个拦截器都已生效，并且按照下面这个顺序执行：

(全局->类级->方法级)

### 2.7.4 拦截范围

提示：在[<第四章：环境支持(4.6节-Aop拦截器服务(@Before))>](#_4.6__Aop拦截器服务(@Before))章节会有更加详细的介绍。

/\*语法\*/

apiBinder.getGuiceBinder().bindInterceptor(

<要代理的类筛选器>, <要代理的方法筛选器>, <拦截器对象>);

//任意类的任意方法

apiBinder.getGuiceBinder().bindInterceptor(

Matchers.*any*(), Matchers.*any*(), **new** AopInterceptor\_3());

//org.test包中的任意类的任意方法(不包含子包)

apiBinder.getGuiceBinder().bindInterceptor(

Matchers.*inPackage*(Package.*getPackage*("org.test")),

Matchers.*any*(), **new** AopInterceptor\_3());

//org.test包中的任意类的任意方法(包含子包)

apiBinder.getGuiceBinder().bindInterceptor(

Matchers.*inSubpackage*("org.test"),Matchers.*any*(),

**new** AopInterceptor\_3());

//标记了Bean注解的类

apiBinder.getGuiceBinder().bindInterceptor(

Matchers.*annotatedWith*(Bean.**class**), Matchers.*any*(),

**new** AopInterceptor\_3());

//自定义拦截器

**public** **class** CustomMatcher **extends** AbstractMatcher<Class<?>> {

**public** **boolean** matches(Class<?> t) {

**return** **false**;//做你要做的事，返回true false就可以了

}

}

//注册自定义拦截器

…….bindInterceptor(**new** CustomMatcher(),Matchers.*any*(),……);

## 2.8 事件的抛出和监听(Event)

使用事件可以为程序的模块划清界限，明确了通知者和接受者之间的关系。同时事件还可以增加程序的可维护性和重用性。[<第四章：环境支持(4.3节-事件(EventManager))>](#_4.3__事件(EventManager))

### 2.8.1 抛出和监听

下面代码定义了一个“HelloEvent”事件的监听器，当收到事件时打印第一个事件参数。

@EventListener("HelloEvent")

**public** **class** CustomEvent **implements** HasorEventListener{

**public** **void** onEvent(String event, Object[] ps) **throws** Throwable{

System.*out*.println(ps[0]);

}

}

所用下面这段代码引发事件并传递一个参数。

context.getEventManager().doSyncEvent("HelloEvent", "hello");

### 2.8.2 同步事件

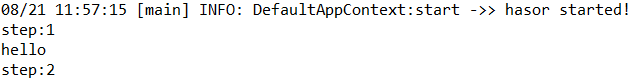
同步事件是指，当事件引发之后。需要等待所有事件监听器处理完毕才能继续执行引发事件后面的代码。在[<第四章：环境支持(4.3.4节-同步事件)>](#_4.3.4__同步事件) 会有更加详细的介绍。

System.*out*.println("step:1");

context.getEventManager().doSyncEvent("HelloEvent", "hello");

System.*out*.println("step:2");

下面是执行结果:



### 2.8.3 异步事件

异步事件是指，当事件引发之后，事件管理器会使用其他线程分发事件给事件监听器。事件引发程序可以不受阻塞的方式继续后面的程序执行。在[<第四章：环境支持(4.3.5节-异步事件)>](#_4.3.6__异步事件)会有更加详细的介绍。

System.*out*.println("step:1");

context.getEventManager().doAsynEventIgnoreThrow(

"HelloEvent", "hello");

System.*out*.println("step:2");

下面是执行结果:



## 2.9 一个模块依赖另外一个模块

在Hasor中依赖被分为三种情况（强依赖/弱依赖/依赖反制），使用这三种方式配置依赖可以灵活的根据需要来制定我们模块的启动顺序。下面是三种依赖的详细描述。

### 2.9.1 强依赖

是一种A依赖B的强制关系。就好比先有父亲后有孩子一样，父亲不存在孩子不可能存在。在[<第三章：模块依赖(3.4.1节-强依赖)>](#_3.4.1__强依赖) 会有更多详细的介绍。

在Hasor中这种强制依赖具体表现在被依赖的B模块只有正常启动A模块才能进行启动，否则A不会进入启动过程。

**public** **class** Mode1 **extends** AbstractHasorModule {

**public** **void** init(ApiBinder apiBinder) {

System.*out*.println("Mode1 init!");

}

}

**public** **class** Mode2 **extends** AbstractHasorModule {

**public** **void** configuration(ModuleSettings info) {

info.followTarget(Mode1.**class**);//Mode2强制依赖Mode1

}

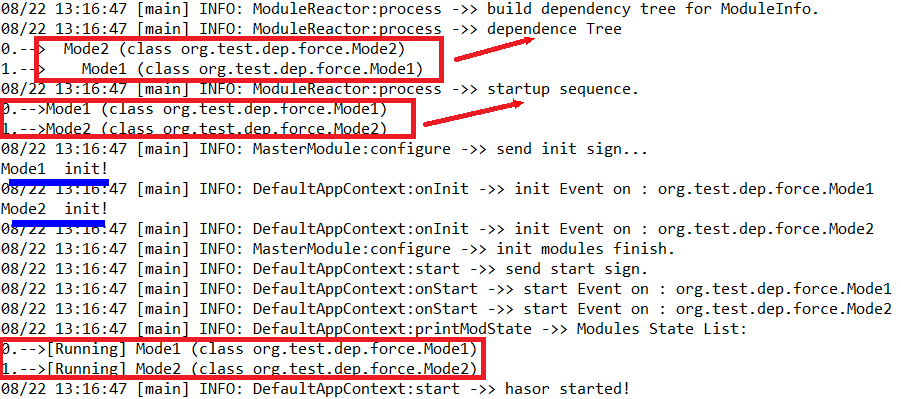
**public** **void** init(ApiBinder apiBinder) {

System.*out*.println("Mode2 init!");

}

}

【启动日志分析】



日志中第一个方框中的内容展示了Hasor中模块的依赖情况（依赖树）。第二个方框表示了模块的启动顺序。随后可以看到Hasor分别启动了这两个模块。在最后一个方框中表示了Hasor整个初始化启动完成之后各个模块的运行状态。

稍微修改一下程序，在Mode1模块的init方法中抛出一个异常再次执行程序可以得到下面这段日志信息。



第二个方框中可以看到两个模块都没有启动。

### 2.9.2 弱依赖

是一种A依赖B的关系，在这种依赖关系中只强调顺序不强调必然联系。这好比领导和员工，领导没了员工照样可以工作的道理。在[<第三章：模块依赖(3.4.2节-弱依赖)>](#_3.4.2__弱依赖)会有更多详细的介绍。

在Hasor中这种依赖表现在被依赖的B模块无论是否正常启动，A模块都会在B模块之后执行启动动作。修改上面Mode2例子中的代码：

**public** **class** Mode2 **extends** AbstractHasorModule {

**public** **void** configuration(ModuleSettings info) {

info.beforeMe(Mode1.**class**); //Mode2弱依赖Mode1

}

为了凸显这种弱依赖关系，被依赖的模块Mode1使用下面这段代码抛出一个异常：

**public** **class** Mode1 **extends** AbstractHasorModule {

**public** **void** init(ApiBinder apiBinder) {

**throw** **new** RuntimeException("this is my Error");

}

}

【启动日志分析】



在日志中可以看到Mode1初始化出现错误并得到了一个Error日志，日志中列出了异常信息。在第二个方框中看到Mode1虽然失败了，但是并没有影响到Mode2。

### 2.9.3 依赖反制

依赖反制可以将模块依赖关系颠倒。它相当于使用A依赖B的代码声明出B依赖A。它的具体应用场景可以被理解为当你想让自己的模块在引入的第三方模块包之前启动加载时。就可以使用依赖反制要求第三方模块依赖自己。

在[<第三章：依赖反制(3.4.3节-依赖反制)>](#_3.4.3__依赖反制)会有更多详细的介绍。

**public** **class** MyMode **extends** AbstractHasorModule {

**public** **void** configuration(ModuleSettings info) {

info.afterMe(JFinalMode.**class**);//反转依赖

}

……

}

## 2.10 使用配置文件

本小节主要讲解如何读取自定义配置文件。在[<第五章：配置文件>](#_第五章__配置文件)可以了解到更多内容。Hasor的配置文件需要放到ClassPath中且文件名为“hasor-config.xml”（全小写）下面这段Xml片段是配置文件的基本定义：

<?xml version=*"1.0"* encoding=*"UTF-8"* ?>

<config xmlns=*"http://project.hasor.net/hasor/schema/main"*>

</config>

现有如下配置文件：

<config xmlns=*"http://project.hasor.net/hasor/schema/main"*>

<myProject name=*"HelloWord"*>项目描述信息......</myProject>

<userInfo id=*"001"* name=*"哈库纳"* age=*"27"*>

<address>

<name>北京市海淀区...</name>

</address>

</userInfo>

</config>

使用下面这段代码读取部分配置文件的内容，无需编写自定义配置文件解析器。

**public** **static** **void** main(String[] args) **throws** IOException {

DefaultAppContext context = **new** DefaultAppContext();

Settings setting = context.getSettings();

System.*out*.println(setting.getString("myProject.name"));

System.*out*.println(setting.getString("myProject"));//项目信息

System.*out*.println(setting.getInteger("userInfo.age"));//age

}

执行结果会打印出：“HelloWord”、“项目描述信息......”、“27”

规则：

Hasor将标签所在Xpath路径用“属性.属性.属性”的方式进行Key/Value映射。这种映射的好处是减少了开发人员对Xml解析操作。

根据上面的Xml文件映射结果可以用如下表进行表示：

|  |  |
| --- | --- |
| key | Value |
| myProject | HelloWord |
| myProject.name | 项目描述信息...... |
| userInfo.id | 001 |
| userInfo.name | 哈库纳 |
| userInfo.age | 27 |
| userInfo.address.name | 北京市海淀区... |

规则限制：

1.当标签子元素和标签属性重名时，会发生属性值覆盖问题。

2.当存在多个相同标签配置不同内容时会发生属性值丢失覆盖问题。

3.不支持带命名空间的属性。

4.Xml配置文件标签在转换成key值时忽略大小写。

5.根节点不参与key值转换（无法突破该限制）。

使用DOM方式获取Xml内容：

当你遇到上述规则限制时可以使用DOM方式获取你想要的内容。例如：在下面这段Xml配置文件中获取用户001的姓名。

<?xml version=*"1.0"* encoding=*"UTF-8"* ?>

<config xmlns=*"http://project.hasor.net/hasor/schema/main"*>

<userInfo id=*"001"*>

<name>哈库纳</name>

<age>27</age>

<address>北京市海淀区...</address>

</userInfo>

<userInfo id=*"002"*>

<name>阿狸</name>

<age>30</age>

<address>青海...</address>

</userInfo>

</config>

下面是DOM方式读取userInfo节点的例子代码：

/\*虽然根节点不参与Key/Value转换但是可以获取到它\*/

XmlProperty xmlNode = setting.getXmlProperty("config");

**for** (XmlProperty node : xmlNode.getChildren()) {

**if** ("userInfo".equals(node.getName())) {

**if** ("001".equals(node.getAttributeMap().get("id"))) {

System.*out*.println(node);

}

}

}

使用Hasor配置文件的约定：

请不要使用下面这些名字作为第一级标签名这些都是Hasor保留的：

hasor ：这个标签是Hasor核心配置。

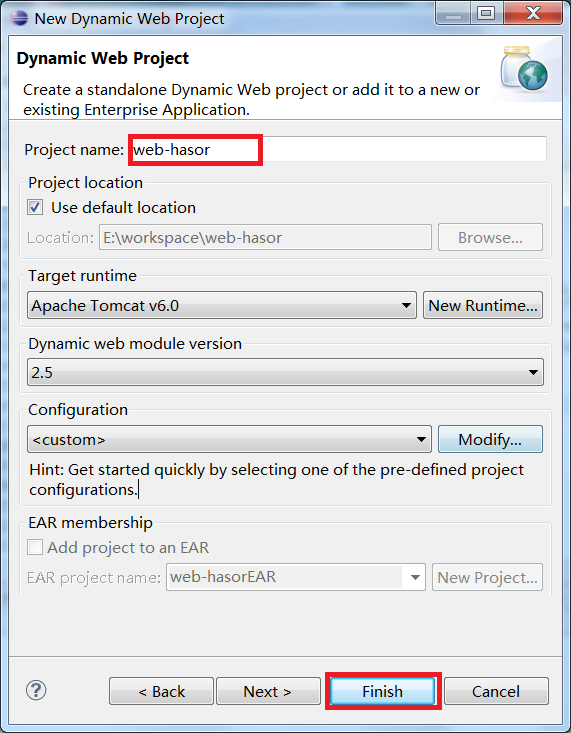
environmentVar ：这个标签是用来配置Hasor环境变量

hasor-<其他字符> ：hasor不同的模块保留区域，例如：Hasor-MVC模块是“hasor-mvc”。

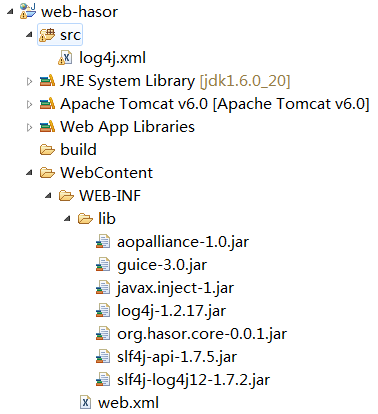
## 2.11 Web项目

Step 1：创建Dynamic Web Project项目

目前Hasor对Web工程没有特殊配置要求。新建一个Web项目输入项目名称点击Finish完成即可，您也可以按照项目需要进行特殊制定。



Step 2：在Web工程中加入Hasor的Jar包到lib目录，下面图中除了Hasor必须依赖的类库，除此之外还增加了Log4j相关日志的配置。



Step 3：修改web.xml配置文件加入如下内容即完成配置。

注意：Hasor在默认情况下会将请求编码设置为“**UTF-8**”响应编码也被设置为“**UTF-8**”。

<listener>

<listener-class>

org.hasor.servlet.startup.RuntimeListener

</listener-class>

</listener>

<filter>

<filter-name>runtime</filter-name>

<filter-class>

org.hasor.servlet.startup.RuntimeFilter

</filter-class>

</filter>

<filter-mapping>

<filter-name>runtime</filter-name>

<url-pattern>/\*</url-pattern>

</filter-mapping>

## 2.12 Web开发

### 2.12.1 HttpServlet

通过@WebServlet注解可以注册HttpServlet，这种方式可以避免我们配置web.xml。在[<第六章：Web支持(6.3节-HttpServlet)>](#_6.3__HttpServlet(@WebServlet))可以了解更多详细信息。

**import** org.hasor.servlet.anno.WebServlet;

@WebServlet({ "hasor.ex" })//声明Servlet

**public** **class** HelloServlet **extends** HttpServlet {

@Override

**protected** **void** service(HttpServletRequest arg0,

HttpServletResponse arg1)

**throws** ServletException, IOException {

Writer w = arg1.getWriter();

w.write("Hello Word");

w.flush();

}

}

启动程序输入Servlet地址“http://127.0.0.1:8080/<your project name>/hasor.ex”



### 2.12.2 Filter

通过@WebFilter注解可以注册Filter。在[<第六章：Web支持(6.4节-Filter)>](#_6.4__Filter过滤器(@WebFilter))可以了解更多详细信息。

**import** org.hasor.servlet.anno.WebFilter;

@WebFilter("/\*")//声明Filter

**public** **class** HelloFilter **implements** Filter {

**public** **void** doFilter(ServletRequest arg0, ServletResponse arg1,

FilterChain arg2) **throws** IOException, ServletException {

Writer w = arg1.getWriter();

w.write("Hello Word<br/>");

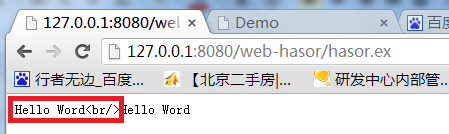
arg2.doFilter(arg0, arg1);

}

……

}

再次启动程序重新访问Servlet“http://127.0.0.1:8080/<your project name>/hasor.ex”



### 2.12.3 Session监听器(HttpSessionListener)

通过@WebSessionListener注解标记在HttpSessionListene接口实现类上完成监听器注册。在[<第六章：Web支持(6.5.1节-Session监听器)>](#_6.5.1__Session监听器(@WebSessionListen)会有更多详细介绍。

@WebSessionListener//声明HttpSessionListener

**public** **class** SessionLinser **implements** HttpSessionListener {

**public** **void** sessionCreated(HttpSessionEvent arg0) {

String sessionID = arg0.getSession().getId();

System.*out*.println("create session:" + sessionID);

}

**public** **void** sessionDestroyed(HttpSessionEvent arg0) {

String sessionID = arg0.getSession().getId();

System.*out*.println("destroyed session:" + sessionID);

}

}

创建一个index.jsp文件放到网站跟目录（jsp会主动创建HttpSession）

访问index.jsp文件你会在控制台得到类似下面这样的输出：



### 2.12.4 Servlet启动监听器(ServletContextListener)

通过@WebContextListener注解标记在ServletContextListener接口实现类上完成监听器注册。在[<第六章：Web支持(6.5.2节-Context监听器)>](#_6.5.2__Context监听器(@WebContextListen)会有更多详细介绍。

@WebContextListener//声明ServletContextListener

**public** **class** ContextLinser **implements** ServletContextListener {

**public** **void** contextDestroyed(ServletContextEvent arg0) {

System.*out*.println("contextDestroyed.");

}

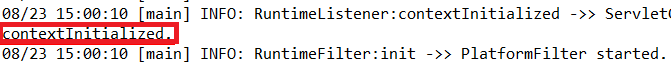
**public** **void** contextInitialized(ServletContextEvent arg0) {

System.*out*.println("contextInitialized.");

}

}

当Web工程启动时就会调用上面的监听器通知程序启动。下面是控制台输出：



### 2.12.5 截获服务器异常

该服务是用来拦截意外的Servlet异常抛出。该功能支持根据异常类型绑定不同的处理程序。详细内容详见：[<第六章：Web支持(6.6节-Servlet异常截获)>](#_6.6__Servlet异常截获(@WebError))。

提示：关于404，该功能不支持拦截诸如404状态。这是由于404并不代表出现服务器异常。

下面代码中包含了一个抛出异常的Servlet和一个异常拦截器：

**import** org.hasor.servlet.anno.WebError;

//异常拦截器

@WebError(ServletException.**class**)

**public** **class** WebError\_500 **implements** WebErrorHook {

**public** **void** doError(ServletRequest request,

ServletResponse response, Throwable error) **throws** Throwable {

System.*out*.println(error.getMessage());

Writer w = response.getWriter();

w.write("Error Msg:" + error.getMessage());

w.flush();

}

}

//抛出异常的Servlet

@WebServlet("err.ex")

**public** **class** ErrServlet **extends** HttpServlet {

**protected** **void** service(

HttpServletRequest req, HttpServletResponse res

) **throws** ServletException, IOException {

**throw** **new** ServletException("ee");

}

}

## 2.13 Web-MVC

使用Web-MVC功能需要加入“org.hasor.mvc-0.0.1.jar”软件包，该软件包提供了Controller、Resource两个模块。在[<第十一章：Hasor-MVC软件包>](#_第十一章__Controller模块(Hasor-MVC))有会对该软件包详细介绍。

Controller： MVC开发模型的支持、提供Action控制器和RESTful映射。

Resource： 允许Web程序获应用程序从Jar包或其它目录中响应Web请求操作。

### 2.13.1 Action

下面定义了一个简单的Action，在[<第十一章：Hasor-MVC软件包(11.2.2节-Action)>](#_11.2.2__Action)会有详细介绍。在浏览器输入如下地址：http://localhost:8080/<projectName>/abc/123/print.do查看执行结果。

@Controller("/abc/123")//命名空间

**public** **class** FirstAction {

/\*print是action名字，代表print.do\*/

**public** **void** print() {

System.*out*.println("Hello Action!");

}

}

限制和约定：

1.Action类中所有共有方法都可以被访问。（私有、受保护）不会被暴露。

2.不支持方法重载。

3.Action扩展名为“\*.do”。

### 2.13.2 获取Request和Response

在Hasor-MVC中获取Request和Response可以使用AbstractController抽象类。

@Controller("/abc/123")

**public** **class** ReqResAction **extends** AbstractController {

**public** **void** print() {

HttpServletRequest req = **this**.getRequest();

HttpServletResponse res = **this**.getResponse();

System.*out*.println("Hello Action!");

}

}

AbstractController作为Action的基类在Action上提供了很多有趣的工具方法。详细的介绍参看[<第十一章：Hasor-MVC软件包(11.2.4节-AbstractController抽象类)>](#_11.2.4__AbstractController抽象类)。

### 2.13.3 RESTful映射

在Hasor中RESTful风格的实现参照了JSR-311标准。下面这段代码是一个简单的RESTful映射。在[<第十一章：Hasor-MVC软件包(11.2.5节-RESTful支持)>](#_11.2.5__RESTful支持)会有更多的介绍内容。

@Path("/user/{uid}/")

**public** **void** userInfo( @PathParam("uid") String uid) {

System.*out*.println("user :" + uid);

}

在浏览器输入如下地址：http://localhost:8080/<projectName>/user/123你会看到控制台打印出“user :123”。下面这段代码展示了其他参数获取的方式（部分）。

@Path("/user/{uid}/")

**public** **void** userInfo(

@PathParam("uid") String uid,//@Path中声明的参数。

@HeaderParam("User-Agent") String userAgent,//Heade请求头

@QueryParam("age") **int** age,//请求地址“?”之后的参数。

@QueryParam("ns") String[] ns) {//同名参数数组

System.*out*.println(String.*format*("user %s age=%s by:%s",

uid, age, userAgent));

}

【注解说明】

@AttributeParam 其功能相当于request.getAttribute(xxx)。

@CookieParam 其功能相当于从Cookie中获取内容。

@HeaderParam 其功能相当于request.getHeader(xxx)。

@InjectParam 其功能相当于context.getInstance(XXX.class)。

@PathParam 用于获取在@Path注解中用“{...}”括起来的内容。

@QueryParam 用于获取URL请求地址“?”后面的参数。

【动词限定】

在RESTful中每一个资源会因不同的Http请求动作被解释为不同的操作。这一点可以在Action方法上标记动词注解以达到目的。有关RESTful动词信息参见：[<第十一章：Hasor-MVC软件包(11.2.6节-Http动词与RESTful)>](#_11.2.6__Http动词与RESTful)。

例如:下面两个方法都映射到同一个RESTful地址，但是分别由不同的动词进行标记。

@Controller()//标记是一个控制器，不需要配置名字空间。

**public** **class** UserAction **extends** AbstractController {

@Get //接受Get类型请求【动词Get】

@Path("/userMag/{uid}")//请求样例：/userMag/AA-BB-CC?name=ABC

**public** Object getUserObject(@PathParam("uid") String userID) {

System.*out*.println(String.*format*("get user %s.", userID));

HashMap mapData = **new** HashMap();

mapData.put("userID", userID);

mapData.put("name", "用户名称");

**return** mapData;

}

@Post //接受Pust类型请求【动词Post】

@Path("/userMag/{uid}")//请求样例：/userMag/AA-BB-CC?name=ABC

**public** **void** updateUser(@PathParam("uid") String userID) {

String name = **this**.getPara("name");

System.*out*.println(

String.*format*("update user %s new Name is %s",

userID, name));

}

}

【可用的动词注解】

@Any ：接收任何动词类型的请求。

@Get ：只接收GET请求。

@Head ：只接收HEAD请求。

@Options ：只接收OPTIONS请求。

@Post ：只接收POST请求。

@Put ：只接收PUT请求。

提示：在Hasor中动词可以通过@HttpMethod注解自定义。

### 2.13.4 返回Json数据

在Hasor中由Action返回一个JSON数据只需要在Action方法上标记一个@Json注解。并且将要回写到客户端的数据对象return即可，Action控制器会执行JSON序列化。例如：

@Controller("/abc/123")

**public** **class** ReqResAction {

@Json()

**public** Object getData() { **return** ……; }

}

### 2.13.5 Action结果处理

本节内容是[<2.12.4节-返回Json数据>](#_2.12.4__返回Json数据)内容的延伸。@Json注解是Action结果处理器提供的功能之一。除了@Json注解之外Action结果处理器还提供了其他几种返回值处理方式。并且开发者可以自定义结果处理逻辑。在[<第十一章：Hasor-MVC软件包(11.2.9节-Action返回值处理扩展)>](#_11.2.9__Action返回值处理扩展)中会有更加详细的说明。

【内置结果处理器】

@Json 返回值可以是任意类型，使用这种方式可以将Action的返回值序列化成为JSON数据 并响应给客户端。

@Forword 返回一个字符串，当Action调用处理完毕之后处理一个服务端转发操作。

@Include 返回一个字符串，当Action调用处理完毕之后处理一个服务端包含操作。

@Redirect 返回一个字符串，当Action调用处理完毕之后处理一个客户端重定向操作。

【自定义结果处理器】

下面的代码来自于@Json的实现。

@Retention(RetentionPolicy.*RUNTIME*)

@Target({ ElementType.*METHOD* })

**public** **@interface** Json {}

@ControllerResultDefine(Json.**class**)

**public** **class** JsonResultPro **implements** ControllerResultProcess {

/\*接口ControllerResultProcess的实现方法，介于篇幅故省略参数\*/

**public** **void** process(……) **throws** ServletException, IOException {

String jsonData = JSON.*toString*(result);

**if** (response.isCommitted() == **false**)

response.getWriter().write(jsonData);

}

}

## 2.14 打包Web资源(WebJars)

现在Web前端使用了越来越多的JS或CSS等静态资源文件，如jQuery、Backbone.js和Bootstrap等等。平时webapp目录下的这些资源在开发过程中每次打包、测试都需要等待很长时间，而且不便于统一管理。为此我们更喜欢将它们打入一个Jar包，然后凭借一些小工具让Web程序支持它们。Resource模块就是这样一种工具。该功能默认是关闭的，需要通过“hasor-config.xml”配置才可开启该功能。

<config xmlns=*"http://project.hasor.net/hasor/schema/main"*>

<hasor-mvc>

<resourceLoader enable=*"true"/*>

</hasor-mvc>

<environmentVar>

<!-- 设置工作路径，jar中的资源文件会根据这个路径存放 -->

<HASOR\_WORK\_HOME>c:\hasor-work</HASOR\_WORK\_HOME>

</environmentVar>

</config>

【补充说明】

该功能目前可以支持“js,css,gif,ico,jpg,jpeg,png,swf,swc,flv,mp3,wav,avi”文件格式，你也可以通过配置contentTypes属性用来重写这个配置，例如：

<config xmlns=*"http://project.hasor.net/hasor/schema/main"*>

<hasor-mvc>

<resourceLoader enable=*"true"* contentTypes=*"js,css"/*>

</hasor-mvc>

……

</config>

上面这段配置文件重写了contentTypes属性之后Resource模块将只负责处理所有“js、css”类型资源的Web请求响应工作。

在[<第十一章：Hasor-MVC软件包(11.3.节-Resource模块)>](#_11.3__Resource模块)会有Resource模块更详细的功能说明。

### 2.14.1 Jar包中的Web资源。

当完成上述配置工作之后在类路径中增加“/META-INF/webapp”目录。所有位于ClassPath中Web资源都保存这里，这样Hasor的Resource模块才能找到它们。

例如资源：“/META-INF/webapp/images/me.png”的访问地址为：

“http://localhost:8080/<projectName>/images/me.png”。

### 2.14.2 Zip压缩包中的Web资源。

当项目拥有众多部署版本，不同版本之间仅有几个资源文件不同时，(2.13.1节)所提供的功能就无法满足你的部署要求。

为此Resource模块支持从一个外部Zip格式的压缩包作为Web资源提供源。以满足这种场景下Web资源文件管理的需求。（Zip文件可以存放到任意可访问的目录上）

下面这段配置展示了如何将“C:\bizData\icons.zip”压缩包中的压缩文件。作为Web资源。

<config xmlns=*"http://project.hasor.net/hasor/schema/main"*>

<hasor-mvc>

<resourceLoader enable=*"true"*>

<!-- 使用绝对路径配置的资源文件包 -->

<zipLoader>C:\bizData\icons.zip</zipLoader>

<!-- 使用WEB-INF目录下的pic.zip作为资源文件包-->

<zipLoader>%HASOR\_WEBROOT%/WEB-INF/pic.zip</zipLoader>

</resourceLoader>

</hasor-mvc>

</config>

启动项目在浏览器中输入“http://localhost:8080/<projectName>/static/images/icon1.png”。

提示1：icons.zip压缩包中需要事先存有“/static/images/icon1.png”文件。

提示2：“%HASOR\_WEBROOT%”是Web环境下Hasor的一个环境变量，相当于这段代码：“ServletContext.getRealPath("/")”。

提示3：当配置两个以上Loader时候按照Loader配置的先后顺序处理冲突的资源。

### 2.14.3 指定的目录中加载资源Web资源。

与(2.13.2节-Zip压缩包中的Web资源)提供的功能是一致的。不同的是这种方式是将一个目录作为“资源压缩包”配置方式如下：

<config xmlns=*"http://project.hasor.net/hasor/schema/main"*>

<hasor-mvc>

<resourceLoader enable=*"true"*>

<!-- 使用绝对路径配置的资源文件包 -->

<pathLoader>C:\bizData\icons\</pathLoader>

<!-- 使用WEB-INF目录下的pic.zip作为资源文件包-->

<pathLoader>%HASOR\_WEBROOT%/WEB-INF/picDir</pathLoader>

</resourceLoader>

</hasor-mvc>

</config>

# （未完待续）

……

# 第三章 架构

## 3.1 技术选型

## 3.2 总体架构

## 3.3 分层设计

## 3.4 生命周期

### 3.4.1 init阶段

### 3.4.2 start阶段

### 3.4.3 stop阶段

## 3.5 模块&插件

### 3.5.1 运行状态

### 3.5.2 依赖

### 3.5.3 插件

## 3.6 事件

## 3.7 环境变量

## 3.8 Xml解析

## 3.9 Web支持

## 3.10 JDBC支持

# 第四章 核心技术

## 4.1 Core部分

### 4.1.1 Bean

### 4.1.2 IoC(JSR-330)

### 4.1.3 Aop

### 4.1.4 Event

### 4.1.5 Plugin

### 4.1.6 Module

### 4.1.7 Guice

### 4.1.8 Cache

### 4.1.9 读取配置文件

### 4.1.10 配置文件监听器

### 4.1.11 解析Xml文件

## 4.2 Web部分

### 4.2.1 Controller

### 4.2.2 Restful

### 4.2.3 Result

### 4.2.4 Servlet3.0

### 4.2.5 Request请求资源

### 4.2.6 Hasor JSP标签库

## 4.3 JDBC部分

### 4.3.1 增/删/改/查

### 4.3.2 参数化SQL

### 4.3.3 单值查询

### 4.3.4 调用存储过程

### 4.3.5 事务控制

### 4.3.6 事务传播行为

### 4.3.7 多数据源

### 4.3.8 多数据源的事务控制

# 第五章 内核开发

## 5.1 Core部分

### 5.1.1 启动内核

### 5.1.2 添加模块

### 5.1.3 注册Bean

### 5.1.4 注册Aop

### 5.1.5 ApiBinder

### 5.1.6 扫描类路径

### 5.1.7 类型绑定与获取

### 5.1.8 事件

### 5.1.9 获取AppContext

### 5.1.10 环境变量

### 5.1.11 创建Guice

### 5.2.12 读取配置文件

### 5.2.13 解析Xml文件

## 5.2 Web部分

### 5.2.1 启动Web支持内核

### 5.2.2 注册HttpServlet

### 5.2.3 注册Filter

### 5.2.4 注册ServletContextListener

### 5.2.5 WebApiBinder

### 5.2.6 获取ServletContext

### 5.2.7 获取Request/Response

## 5.3 JDBC部分

### 5.3.1 脱离Hasor使用JDBC

### 5.3.2 DataSourceHelper

# 第六章 配置文件详解

## 6.1 Core部分

## 6.2 Web部分

## 6.3 JDBC部分

# 第七章 API参考手册

# 第八章 附录

## 8.1 约定