**Hasor 开发指南**

**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](#_Toc381113837)

[1.1 内容引导 7](#_Toc381113838)

[1.2 开源协议 8](#_Toc381113839)

[1.3 内置组建及其授权协议 8](#_Toc381113840)

[1.4 获取和贡献 9](#_Toc381113841)

[1.5 类库引用 9](#_Toc381113842)

[1.6 约定优于配置(COC原则) 9](#_Toc381113843)

[第二章 起步 10](#_Toc381113844)

[2.1 选用IDE 10](#_Toc381113845)

[2.2 创建项目 10](#_Toc381113846)

[2.2.1 创建Hasor项目 10](#_Toc381113847)

[2.2.2 Maven创建Hasor项目 11](#_Toc381113848)

[2.2.3 创建Web项目 12](#_Toc381113849)

[2.3 启动Hasor 13](#_Toc381113850)

[2.4 关于Bean 13](#_Toc381113851)

[2.5 依赖注入(JSR-330) 14](#_Toc381113852)

[2.5.1 概念 14](#_Toc381113853)

[2.5.2 传统注入方式 15](#_Toc381113854)

[2.5.3 Guice与JSR-330 16](#_Toc381113855)

[2.5.4 构造方法注入 16](#_Toc381113856)

[2.5.5 属性方式注入 17](#_Toc381113857)

[2.5.6 字段方式注入 17](#_Toc381113858)

[2.5.7 单例Bean 17](#_Toc381113859)

[2.6 Aop 18](#_Toc381113860)

[2.6.1 概念 18](#_Toc381113861)

[2.6.2 静态代理方式 18](#_Toc381113862)

[2.6.3 动态代理方式 19](#_Toc381113863)

[2.7.1 方法级拦截器 20](#_Toc381113864)

[2.7.2 类级拦截器 20](#_Toc381113865)

[2.7.3 全局拦截器 20](#_Toc381113866)

[2.7.4 拦截范围 21](#_Toc381113867)

[2.8 事件的抛出和监听(Event) 22](#_Toc381113868)

[2.8.1 抛出和监听 22](#_Toc381113869)

[2.8.2 同步事件 22](#_Toc381113870)

[2.8.3 异步事件 23](#_Toc381113871)

[2.9 一个模块依赖另外一个模块 23](#_Toc381113872)

[2.9.1 强依赖 23](#_Toc381113873)

[2.9.2 弱依赖 24](#_Toc381113874)

[2.9.3 依赖反制 25](#_Toc381113875)

[2.10 使用配置文件 25](#_Toc381113876)

[2.12 Web开发 27](#_Toc381113877)

[2.12.1 HttpServlet 27](#_Toc381113878)

[2.12.2 Filter 27](#_Toc381113879)

[2.12.3 Session监听器(HttpSessionListener) 28](#_Toc381113880)

[2.12.4 Servlet启动监听器(ServletContextListener) 28](#_Toc381113881)

[2.12.5 截获服务器异常 29](#_Toc381113882)

[2.13 Web-MVC 29](#_Toc381113883)

[2.13.1 Action 30](#_Toc381113884)

[2.13.2 获取Request和Response 30](#_Toc381113885)

[2.13.3 RESTful映射 30](#_Toc381113886)

[2.13.4 返回Json数据 32](#_Toc381113887)

[2.13.5 Action结果处理 32](#_Toc381113888)

[2.14 打包Web资源(WebJars) 33](#_Toc381113889)

[2.14.1 Jar包中的Web资源。 34](#_Toc381113890)

[2.14.2 Zip压缩包中的Web资源。 34](#_Toc381113891)

[2.14.3 指定的目录中加载资源Web资源。 34](#_Toc381113892)

[第三章 架构 36](#_Toc381113893)

[3.1 技术选型 36](#_Toc381113894)

[3.2 总体架构 36](#_Toc381113895)

[3.3 分层设计 36](#_Toc381113896)

[3.4 生命周期 36](#_Toc381113897)

[3.4.1 init阶段 36](#_Toc381113898)

[3.4.2 start阶段 36](#_Toc381113899)

[3.4.3 stop阶段 36](#_Toc381113900)

[3.5 模块&插件 36](#_Toc381113901)

[3.5.1 运行状态 36](#_Toc381113902)

[3.5.2 依赖 36](#_Toc381113903)

[3.5.3 插件 36](#_Toc381113904)

[3.6 事件 36](#_Toc381113905)

[3.7 环境变量 36](#_Toc381113906)

[3.8 Xml解析 36](#_Toc381113907)

[3.9 Web支持 36](#_Toc381113908)

[3.10 JDBC支持 36](#_Toc381113909)

[第四章 核心技术 36](#_Toc381113910)

[4.1 Core部分 36](#_Toc381113911)

[4.1.1 Bean 36](#_Toc381113912)

[4.1.2 IoC(JSR-330) 36](#_Toc381113913)

[4.1.3 Aop 36](#_Toc381113914)

[4.1.4 Event 36](#_Toc381113915)

[4.1.5 Plugin 36](#_Toc381113916)

[4.1.6 Module 36](#_Toc381113917)

[4.1.7 Guice 36](#_Toc381113918)

[4.1.8 Cache 36](#_Toc381113919)

[4.1.9 读取配置文件 36](#_Toc381113920)

[4.1.10 配置文件监听器 36](#_Toc381113921)

[4.1.11 解析Xml文件 36](#_Toc381113922)

[4.2 Web部分 36](#_Toc381113923)

[4.2.1 Controller 36](#_Toc381113924)

[4.2.2 Restful 36](#_Toc381113925)

[4.2.3 Result 36](#_Toc381113926)

[4.2.4 Servlet3.0 36](#_Toc381113927)

[4.2.5 Request请求资源 36](#_Toc381113928)

[4.2.6 Hasor JSP标签库 36](#_Toc381113929)

[4.3 JDBC部分 36](#_Toc381113930)

[4.3.1 增/删/改/查 37](#_Toc381113931)

[4.3.2 参数化SQL 37](#_Toc381113932)

[4.3.3 单值查询 37](#_Toc381113933)

[4.3.4 调用存储过程 37](#_Toc381113934)

[4.3.5 事务控制 37](#_Toc381113935)

[4.3.6 事务传播行为 37](#_Toc381113936)

[4.3.7 多数据源 37](#_Toc381113937)

[4.3.8 多数据源的事务控制 37](#_Toc381113938)

[第五章 内核开发 37](#_Toc381113939)

[5.1 Core部分 37](#_Toc381113940)

[5.1.1 启动内核 37](#_Toc381113941)

[5.1.2 添加模块 37](#_Toc381113942)

[5.1.3 注册Bean 37](#_Toc381113943)

[5.1.4 注册Aop 37](#_Toc381113944)

[5.1.5 ApiBinder 37](#_Toc381113945)

[5.1.6 扫描类路径 37](#_Toc381113946)

[5.1.7 类型绑定与获取 37](#_Toc381113947)

[5.1.8 事件 37](#_Toc381113948)

[5.1.9 获取AppContext 37](#_Toc381113949)

[5.1.10 环境变量 37](#_Toc381113950)

[5.1.11 创建Guice 37](#_Toc381113951)

[5.2.12 读取配置文件 37](#_Toc381113952)

[5.2.13 解析Xml文件 37](#_Toc381113953)

[5.2 Web部分 37](#_Toc381113954)

[5.2.1 启动Web支持内核 37](#_Toc381113955)

[5.2.2 注册HttpServlet 37](#_Toc381113956)

[5.2.3 注册Filter 37](#_Toc381113957)

[5.2.4 注册ServletContextListener 37](#_Toc381113958)

[5.2.5 WebApiBinder 37](#_Toc381113959)

[5.2.6 获取ServletContext 37](#_Toc381113960)

[5.2.7 获取Request/Response 37](#_Toc381113961)

[5.3 JDBC部分 37](#_Toc381113962)

[5.3.1 脱离Hasor使用JDBC 37](#_Toc381113963)

[5.3.2 DataSourceHelper 37](#_Toc381113964)

[第六章 配置文件详解 37](#_Toc381113965)

[6.1 Core部分 37](#_Toc381113966)

[6.2 Web部分 37](#_Toc381113967)

[6.3 JDBC部分 37](#_Toc381113968)

[第七章 API参考手册 38](#_Toc381113969)

[第八章 附录 38](#_Toc381113970)

[8.1 约定 38](#_Toc381113971)

# 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。基于 Hasor 的开发可以真正达到“零配置文件”。

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

Hasor的代码使用Git作为其版本管理器，您可以在(<http://msysgit.github.io/>)上获取到最新的Git客户端。同时您可以从下面两个地址中得到Hasor的最新源码。

Github：<https://github.com/zycgit/hasor>（国外）

Git@OSC：<http://git.oschina.net/zycgit/hasor>（国内）

当您发现Hasor的漏洞和不足，或者您对Hasor有一个更好的改进或想法，可以通过([zyc@byshell.org](mailto:zyc@byshell.org?subject=您好%20Hasor%20我有一个问题.)、[zyc@hasor.net](mailto:zyc@hasor.net?subject=您好%20Hasor%20我有一个问题.))邮箱联系到我。

Hasor的项目主页为：<http://www.hasor.net/>。目前Hasor依托OSChina作为社区平台和大家交流，这是我在OSChina上的博客地址：<http://my.oschina.net/u/1166271/blog>。

## 1.5 类库引用

Hasor会依赖以下4个软件包，在第三章会有专门的章节用来讲解Hasor为何依赖它们。

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)

如果您打算使用log4j作为日志组建那么需要加入以下两个jar文件，在附录中我添加了log4j.xml配置文件的参考配置内容。

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。

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。您会发现它就像是个保姆一样在照顾着你。

# 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。下面给出目前最新的Eclipse发行版供您选择，Hasor对您的开发环境没有要求。

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

我们推荐您使用Maven管理项目，因为Maven可以有效的帮助您管理软件包的各种依赖。当然如果您习惯于手动配置整个项目，本手册稍后也会有专门的章节介绍给您如何在一般项目中引入Hasor。

通常作为一个maven项目需要满足一些规则，本手册并不会讲解为何一定要满足这些规则才可以使用maven构建项目。这些内容已经超出了手册讨论的范围，有关Maven更多的知识您可以从<Maven 权威指南>这本书中得到答案，值得庆幸的是这本书在网上有免费的PDF版本可以下载。

### 2.2.1 创建Hasor项目

按照您的需要在Eclipse上新建一个项目，然后引入Hasor的jar包。

|  |  |  |
| --- | --- | --- |
| **Jar包** | **作用** | **大小** |
| hasor-core-0.0.7.jar | Hasor核心软件包 | ~700KB |
| hasor-web-0.0.5.jar | Web开发扩展包 | ~150KB |
| hasor-jdbc-0.0.2.jar | JDBC开发扩展包 | ~110KB |

以下是Hasor项目的基础依赖，无论您使用上面那个jar包都需要加入它们。

|  |  |  |
| --- | --- | --- |
| **Jar包** | **作用** | **大小** |
| slf4j-api-1.7.5.jar | 负责输出Hasor产生的日志 | 26KB |
| guice-3.0.jar | Guice3.0，负责提供DI相关的支持 | 694KB |
| aopalliance-1.0.jar | Aop联盟API，Guice依赖 | 5KB |
| javax.inject-1.jar | JSR-330标准，Guice依赖 | 3KB |

有人可能会问Hasor-Core为什么那么大？这是由于Hasor-Core中存在一些第三方的工具包Hasor为了避免引入更多的依赖。将用到的工具包的源代码加入到Hasor的软件包中。引入的工具包，占了Hasor约500KB的体积。

### 2.2.2 Maven创建Hasor项目

首先假定您已经在本地配置好了Maven环境，您可以使用下面这个命令创建一个一般的Maven项目。

mvn archetype:create -DgroupId=org.myhasor.app -DartifactId=myproject

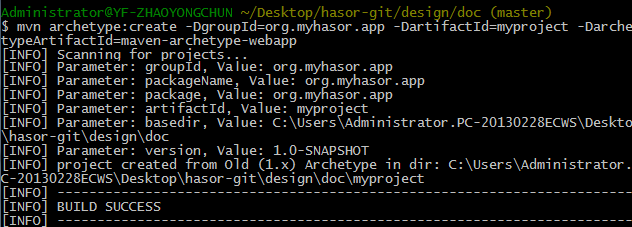
当您执行完上面这个命令之后可以得到一个名为“myproject”的文件夹。这就是Maven为我们创建好的一般Java项目。

如果您是创建一个Web类型的项目，可以使用下面这个命令。这两个命令创建出来的maven项目目录结构上稍微有一些不同。

mvn archetype:create -DgroupId=org.myhasor.app -DartifactId=myproject

-DarchetypeArtifactId=maven-archetype-webapp

以Web类型项目为例，您大致可以在控制台得到下面这样的反馈：



当您看到“**BUILD SUCCESS**”字样之后就表明您的Maven项目已经建好了，然后进入这个目录打开“pom.xml”配置文件。找到“<dependencies>”标签，在它的后面加入下面这段依赖配置。

<dependency>

<groupId>net.hasor</groupId>

<artifactId>hasor-core</artifactId>

<version>0.0.7</version>

</dependency>

如果您创建的是Web项目，我建议您使用下面这个依赖配置。

<dependency>

<groupId>net.hasor</groupId>

<artifactId>hasor-web</artifactId>

<version>0.0.5</version>

</dependency>

如果您还需要Hasor-JDBC方面的支持，还需要追加以下依赖配置：

<dependency>

<groupId>net.hasor</groupId>

<artifactId>hasor-jdbc</artifactId>

<version>0.0.2</version>

</dependency>

提示：截止目前(2014-01-15)上面的配置是最新版本的配置，您可以到Hasor的官方网站获取最新版本。

当您配置好“pom.xml”之后需要在“pom.xml”所在的目录下执行下面这个命令以更新项目的依赖并将其转换为Eclipse项目。

mvn eclipse:eclipse

假如为Web类型项目，那么需要使用下面这个命令：

mvn eclipse:eclipse -Dwtpversion=2.0

如果您想看到Hasor的源代码和JavaDocs，那么可以对应使用下面这组命令：

mvn eclipse:eclipse -DdownloadSources -DdownloadJavadocs

mvn eclipse:eclipse -Dwtpversion=2.0 -DdownloadSources -DdownloadJavadocs

最后，您将项目导入Eclipse即可。

### 2.2.3 创建Web项目

只有当您引入了Hasor-Web软件包时，才会考虑web.xml配置文件。Hasor-Web为了保证Servlet2.5的兼容性，并没有直接依赖Servlet3.0相关API，因此使用Hasor-Web时候必须配置web.xml。

下面这段配置向您展示了如何在Web程序中配置Hasor：

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

<web-app …… version=*"2.4"*>

<listener>

<listener-class>

net.hasor.web.startup.RuntimeListener

</listener-class>

</listener>

<filter>

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

<filter-class>

net.hasor.web.startup.RuntimeFilter

</filter-class>

</filter>

<filter-mapping>

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

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

</filter-mapping>

</web-app>

“RuntimeListener”监听器的作用是启动Hasor容器。“RuntimeFilter”拦截器是所有Web请求的入口，因此它的拦截路径需要被配成“/\*”。

如果您最初就是在使用Hasor开发项目,那么很高兴您的web.xml配置文件在将来也就只有这些内容了。即使您需要整合Struts2或者SpringMVC，您也不必需要再去理会“web.xml”

## 2.3 启动Hasor

本书中所有例子的讲解都可以在“examples”目录中示列程序中找到，了解示列程序将会有助于您学习Hasor。

下面这段代码是在控制台中运行的，它展示了如何创建并启动Hasor上下文：

**import** java.io.IOException;

**import** net.hasor.core.AppContext;

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

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

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

//创建Hasor环境对象

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

context.start();//启动Hasor容器，必须调用

}

}

Hasor提供配置文件。如果您想使用配置文件初始化Hasor容器。您就需要将配置文件放置到classpath下，并起名为“hasor-config.xml”Hasor默认会寻找它。当然您可以使用下面这样的代码给定配置文件的名称：

AppContext app = **new** AnnoStandardAppContext("myHasor-config.xml");

下面是hasor-config.xml配置文件中需要的最小内容：

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

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

</config>

其实，任何带有命名空间描述的xml文件都可以作为Hasor的配置文件。

建议您，使用上面这个配置文件中使用的命名空间，因为这个命名空间是Hasor约定的默认配置空间。

## 2.4 关于Bean

在Hasor中任何类都可以被视为Bean，但是通常我们所指的Bean包括了：Pojo、Service、Action、DTO、BO、VO等等。

在Hasor中Bean被分为“**有名字的**”和“**匿名的**”，它们的区别仅仅是鉴于是否有名字。下面这段代码向大家展示了如何将一个普通类声明为具有名字的Bean：

@Bean("myBean")/\*@Bean注解是用来起名字的\*/

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

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

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

}

}

一个具有名字的Bean，对我们而言最大的好处就是可以通过名字获取到它，例如：

CustomBean myBean = context.getBean("myBean");

当然您也可以通过Hasor获取没有名字的Bean：

CustomBean myBean = context.getInstance(CustomBean.**class**);

## 2.5 依赖注入(JSR-330)

### 2.5.1 概念

“依赖注入(DI)”有时候也被称为“控制反转(IoC)”本质上它们是同一个概念。具体是指，当某个类调用另外一个类的时候通常需要调用者来创建被调用者。但在控制反转的情况下调用者不在主动创建被调用者，而是改为由容器注入，因此而得名。

这里的“创建”强调的是调用者的主动性。而依赖注入则不在需要调用者主动创建被调用者。

举个例子通常情况下调用者（ClassA），会先创建好被调用者（FunBean），然后在调用方法callFoo中调用被调用者（FunBean）的foo方法：

**public** **class** ClassA {

**private** FunBean funBean = **new** FunBean();

**public** **void** callFoo() {

**this**.funBean.foo();

}

}

**public** **class** FunBean {

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

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

}

}

使用了依赖注入的情况恰恰相反，调用者（ClassA）事先并不知道要创建哪个被调用者（FunBean）。ClassA调用的是被注入进来的FunBean，通常我们会为需要依赖注入的对象留有set方法，在调用callFoo方法之前是需要先将funBean对象通过setFunBean方法设置进来的。例如：

**public** **class** ClassA {

**private** FunBean funBean = **null**;

**public** **void** setFunBean(FunBean funBean) {

**this**.funBean = funBean;

}

**public** **void** callFoo() {

**this**.funBean.foo();

}

}

**public** **class** FunBean {

……

### 2.5.2 传统注入方式

严格意义上来说注入的形式分为两种，它们是“**构造方法注入**”和“**set属性注入**”。我们经常听到有第三种注入方式叫“**接口注入**”。其实它只是“set属性注入”的一种接口表现形式。

**A.构造方法注入：**是指被注入的对象通过构造方法传入，例如下面代码：

**public** **class** ClassA {

**private** FunBean funBean = **null**;

**public** ClassA(FunBean funBean) {

**this**.funBean = funBean;

}

**public** **void** callFoo() {

**this**.funBean.foo();

}

}

**B.set属性注入：**是指被注入的对象通过其get/set读写属性方法注入进来，例如：

**public** **class** ClassA {

**private** FunBean funBean = **null**;

**public** **void** setFunBean(FunBean funBean) {

**this**.funBean = funBean;

}

**public** **void** callFoo() {

**this**.funBean.foo();

}

}

**C.接口注入：**是指通过某个接口的set属性方法来注入，大家可以看到其本质还是set属性注入。只不过调用者（ClassA），需要实现某个注入接口。

**public** **interface** IClassA {

**public** **void** setFunBean(FunBean funBean);

}

**public** **class** ClassA **implements** IClassA{

**private** FunBean funBean = **null**;

**public** **void** setFunBean(FunBean funBean) {

**this**.funBean = funBean;

}

**public** **void** callFoo() {

**this**.funBean.foo();

}

}

### 2.5.3 Guice与JSR-330

JSR-330相关的API是由“**javax.inject.\***”软件包提供的一组标准API。通过注解作为其表现形式。Hasor使用Google旗下的开源DI容器Guice作为其JSR-330的标准支持组件。

Guice是Google开发的一个轻量级，基于Java5（主要运用泛型与注释特性）的依赖注入框架。Guice非常小而且快。Guice是类型安全的，它能够对构造函数，属性，方法（包含任意个参数的任意方法，而不仅仅是setter方法）进行注入。

Guice还具有一些可选的特性比如：自定义scopes，传递依赖，静态属性注入，与Spring集成和AOP联盟方法注入等。

对于DI框架来说，性能是很重要的，Guice比Spring快这是主流说法，在Guice的官方网站上您可以看到它宣称比Spring快1000倍！

Hasor选用Guice是由于它的开发接口十分灵活，比起Spring而言Guice更适合作为一个内嵌DI工具来使用。由于Guice是JSR-330标准的实现，这也就使得Hasor也具备了支持JSR-330标准的能力。在下面几个小节会讲解如何使用JSR-330标准将其注入到需要的类上。

我们先假定有一个被调用者（PojoBean），下面是PojoBean类的源代码：

**public** **class** PojoBean {

**private** String uuid = UUID.*randomUUID*().toString();

**private** String name = "马三";

**private** String address = "北京马连洼街道办...";

**public** String getUuid() {

**return** uuid;

}

**public** **void** setUuid(String uuid) {

**this**.uuid = uuid;

}

……

}

### 2.5.4 构造方法注入

**public** **class** ConstructorInject {

**private** PojoBean userBean;

@javax.inject.Inject/\*依赖注入\*/

**public** ConstructorInject(PojoBean userBean) {

**this**.userBean = userBean;

}

**public** String getUserName() {

**return** **this**.userBean.getName();

}

}

### 2.5.5 属性方式注入

**public** **class** MethodInject {

**private** PojoBean userBean;

@javax.inject.Inject/\*依赖注入\*/

**public** **void** setUserBean(PojoBean userBean) {

**this**.userBean = userBean;

}

**public** String getUserName() {

**return** **this**.userBean.getName();

}

}

### 2.5.6 字段方式注入

字段注入是DI容器对“set属性注入”的一种改进.这种改进使得被注入的对象不在需要实现一个set方法,DI容器会主动的将要注入的对象赋值到给定的字段上。

**public** **class** FieldInject {

@javax.inject.Inject/\*依赖注入\*/

**private** PojoBean userBean;

**public** String getUserName() {

**return** **this**.userBean.getName();

}

}

### 2.5.7 单例Bean

单例，通常是指整个应用程序范围内某个类型对象只有一个。Hasor使用AppContext接口表示一个应用程序，在一个AppContext内Hasor通过下面这样的代码可以保证单例：

@javax.inject.Singleton/\*声明单例\*/

**public** **class** SingletonBean {

**private** **long** time = 0;

**public** SingletonBean() {

time = System.*currentTimeMillis*();

}

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

System.*out*.println("create at time:" + time);

}

}

## 2.6 Aop

### 2.6.1 概念

“面向切面编程”也被称为“Aop”，是目前非常活跃的一个开发思想。利用AOP可以对业务逻辑的各个部分进行隔离，从而使得业务逻辑各部分之间的耦合度降低，提高程序的可重用性，同时提高了开发的效率。

Aop编程的目的是将例如日志记录、性能统计、安全控制、事务、异常处理等代码从业务逻辑代码中划分出来。

比方说我有一个查询用户信息的接口，现在要为这个接口添加记录的功能。每当执行一次查询都记录下查询消耗时间。如果我要实现这个功能，一般情况下需要在接口实现类的每一个方法前后都要安插代码来收集数据。如果这样做的话会比较繁琐，但是通过Aop的方式就显得非常优雅。

实现Aop编程模型分为（**静态代理**、**动态代理**）两种方式，其中静态代理多以代理模式（Proxy Pattern）的形式出现。而动态代理则花样繁多，常见的有：Java 原生的Propxy、 CGLib、JBossAOP、等。

### 2.6.2 静态代理方式

假设有一个工厂，工厂里的工人上下班每次都需要打卡。那么这个工厂的工人可以抽象为Worker接口、工作可以被抽象成为doWork方法。一个对象化的工人就构建出来了如下：

**public** **interface** Worker {

**public** **void** doWork();

}

打卡分为上班打卡和下班打卡，为此抽象一个打卡机，并将上下班打卡使用beforeWork和afterWork方法表示。如下：

**public** **class** Machine {

**public** **void** beforeWork() { …… }

**public** **void** afterWork() { …… }

}

工厂规定每个员工只要来到工厂就视为上班打卡、当离开工厂就被认为下班打卡。为了人性化考勤，公司使用了一种现代化的技术可以让员工不必自己动手去打卡，犹如配备了一名贴身小秘书。

其实不难看出这项新技术仅仅是围绕着工人（Worker）在工作（doWork）前后实现了自动打卡。下面是这个技术的抽象：

**class** WorkerProxy **implements** Worker {

**private** Machine machine;

**private** Worker targetWorker;

**public** **void** doWork() {

**this**.machine.beforeWork();

**this**.targetWorker.doWork();

**this**.machine.afterWork();

}

}

### 2.6.3 动态代理方式

### 2.6.4 动态代理方式

### 2.6.3 动态代理方式

### 2.6.3 动态代理方式

Aop联盟(Aopalliance)

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 \*/

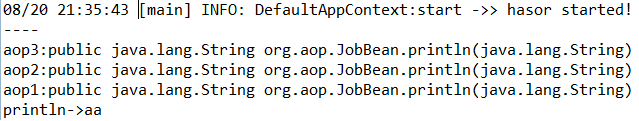
Matcher matcher = Matchers.*not*(

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

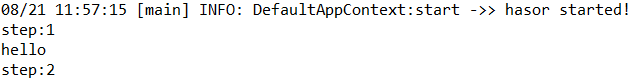
同步事件是指，当事件引发之后。需要等待所有事件监听器处理完毕才能继续执行引发事件后面的代码。在[<第四章：环境支持(4.3.4节-同步事件)>](#_4.3.4__同步事件) 会有更加详细的介绍。

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

}

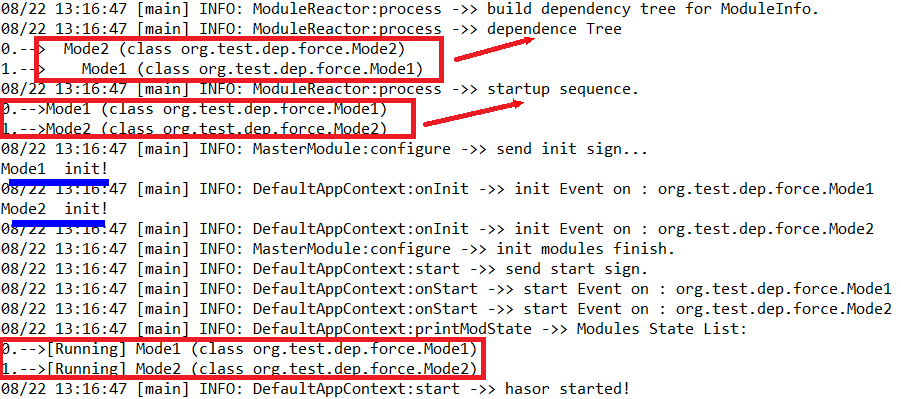
**public** **void** init(ApiBinder apiBinder) {

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”。

## 2.12 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/>");

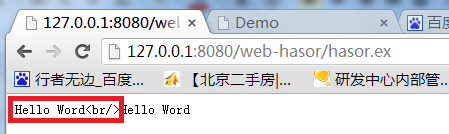
arg2.doFilter(arg0, arg1);

}

……

}

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.");

}

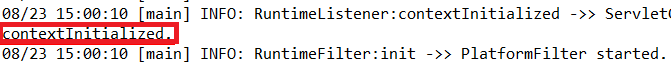
**public** **void** contextInitialized(ServletContextEvent arg0) {

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>

# 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

8.2 参考log4j.xml配置文件