**Hasor 使用手册**

**手册版本：0.0.1**

**软件版本：0.0.1**

**作者：赵永春(zyc@hasor.net)**

**日期：2013-08-16**

目录

[第一章 介绍 5](#_Toc365205722)

[1.1 概述 6](#_Toc365205723)

[1.2 开源协议 6](#_Toc365205724)

[1.3 内置组建及其授权协议 6](#_Toc365205725)

[1.4 获取和贡献 7](#_Toc365205726)

[1.5 类库引用 7](#_Toc365205727)

[第二章 使用Hasor 8](#_Toc365205728)

[2.1 创建项目 8](#_Toc365205729)

[2.2 启动Hasor 9](#_Toc365205730)

[2.3 第一个模块(HelloWord) 10](#_Toc365205731)

[2.4 Bean 10](#_Toc365205732)

[2.4.1 注册Bean 10](#_Toc365205733)

[2.4.2 获取Bean 10](#_Toc365205734)

[2.4.3 单例Bean 11](#_Toc365205735)

[2.4.4 比较注册Bean非注册Bean 11](#_Toc365205736)

[2.5 依赖注入(IoC) 11](#_Toc365205737)

[2.5.1 属性注入 11](#_Toc365205738)

[2.5.2 构造方法注入 12](#_Toc365205739)

[2.5.3 非注册Bean的依赖注入 12](#_Toc365205740)

[2.6 Aop拦截器(MethodInterceptor) 12](#_Toc365205741)

[2.6.1 方法级 13](#_Toc365205742)

[2.6.2 类级 13](#_Toc365205743)

[2.6.3 全局 13](#_Toc365205744)

[2.6.4 拦截范围 14](#_Toc365205745)

[2.7 事件的抛出和监听(Event) 15](#_Toc365205746)

[2.7.1 抛出和监听 15](#_Toc365205747)

[2.7.2 同步事件 15](#_Toc365205748)

[2.7.3 异步事件 15](#_Toc365205749)

[2.8 计时器(Timer) 16](#_Toc365205750)

[2.9 一个模块依赖另外一个模块 16](#_Toc365205751)

[2.9.1 强依赖 16](#_Toc365205752)

[2.9.2 弱依赖 17](#_Toc365205753)

[2.9.3 依赖反制 18](#_Toc365205754)

[2.10 使用配置文件 19](#_Toc365205755)

[2.11 创建Web项目 21](#_Toc365205756)

[2.12 Web开发 22](#_Toc365205757)

[2.12.1 HttpServlet 22](#_Toc365205758)

[2.12.2 Filter 23](#_Toc365205759)

[2.12.3 Session监听器(HttpSessionListener) 23](#_Toc365205760)

[2.12.4 Servlet启动监听器(ServletContextListener) 24](#_Toc365205761)

[2.12.5 截获服务器异常 24](#_Toc365205762)

[2.13 Web-MVC 25](#_Toc365205763)

[2.13.1 Action 25](#_Toc365205764)

[2.13.2 RESTful映射 26](#_Toc365205765)

[2.13.3 获取Request和Response 27](#_Toc365205766)

[2.13.4 返回Json数据 27](#_Toc365205767)

[2.13.5 Action结果处理 27](#_Toc365205768)

[2.14 把Web资源打入Jar包 28](#_Toc365205769)

[第三章 模块 29](#_Toc365205770)

[3.1 定义 29](#_Toc365205771)

[3.2 生命周期(HasorModule) 29](#_Toc365205772)

[3.3 模块运行状态(ModuleInfo) 29](#_Toc365205773)

[3.4 模块依赖(Dependency) 29](#_Toc365205774)

[3.4.1 强依赖 29](#_Toc365205775)

[3.4.2 弱依赖 29](#_Toc365205776)

[3.4.3 依赖反制 29](#_Toc365205777)

[3.4.4 依赖算法 29](#_Toc365205778)

[3.5 一个简单的模块示例(Simple) 29](#_Toc365205779)

[3.6 复杂模块Demo演示 29](#_Toc365205780)

[第四章 环境支持 29](#_Toc365205781)

[4.1 环境(Context) 29](#_Toc365205782)

[4.1.1 StandardInitContext 29](#_Toc365205783)

[4.1.2 DefaultAppContext 29](#_Toc365205784)

[4.1.3 AnnoAppContext 29](#_Toc365205785)

[4.1.4 AnnoWebAppContext 29](#_Toc365205786)

[4.2 类路径扫描范围 29](#_Toc365205787)

[4.2.1 设置扫描范围 29](#_Toc365205788)

[4.3 事件(EventManager) 29](#_Toc365205789)

[4.3.1 系统生命周期事件 29](#_Toc365205790)

[4.3.2 自定义事件 29](#_Toc365205791)

[4.3.3 计时器(Timer) 29](#_Toc365205792)

[4.3.4 只生效一次的事件监听器 29](#_Toc365205793)

[4.3.5 同步事件 29](#_Toc365205794)

[4.3.6 异步事件 29](#_Toc365205795)

[4.3.7 理解事件中的事件 29](#_Toc365205796)

[4.4 Binder 29](#_Toc365205797)

[4.4.1 ApiBinder 29](#_Toc365205798)

[4.4.2 WebApiBinder 29](#_Toc365205799)

[4.4.3 Binder和Guice 29](#_Toc365205800)

[4.4.4 Binder的有效范围 29](#_Toc365205801)

[4.5 Bean服务(@Bean) 29](#_Toc365205802)

[4.6 Aop拦截器服务(@Before) 29](#_Toc365205803)

[4.7 启动和销毁 29](#_Toc365205804)

[4.8 环境变量 29](#_Toc365205805)

[第五章 配置文件 29](#_Toc365205806)

[5.1 简介 29](#_Toc365205807)

[5.2 独立性&随意性(Settings) 29](#_Toc365205808)

[5.2.1 与其他配置文件混杂 30](#_Toc365205809)

[5.3 主配置文件(hasor-config.xml) 30](#_Toc365205810)

[5.4 静态配置文件(static-config.xml) 30](#_Toc365205811)

[5.5 配置名称映射文件(config-mapping.properties) 30](#_Toc365205812)

[5.6 配置文件监听器 30](#_Toc365205813)

[5.7 命名空间(NameSpace) 30](#_Toc365205814)

[5.8 配置文件简单操作(Simple) 30](#_Toc365205815)

[5.9 自定义Xml配置文件解析器 30](#_Toc365205816)

[第六章 Web支持 30](#_Toc365205817)

[6.1 简介 30](#_Toc365205818)

[6.2 Web模块 30](#_Toc365205819)

[6.3 HttpServlet(@WebServlet) 30](#_Toc365205820)

[6.4 Filter过滤器(@WebFilter) 30](#_Toc365205821)

[6.5 监听器 30](#_Toc365205822)

[6.5.1 Session监听器(@WebSessionListener) 30](#_Toc365205823)

[6.5.2 Context监听器(@WebContextListener) 30](#_Toc365205824)

[6.6 Servlet异常截获(@WebError) 30](#_Toc365205825)

[第七章 Guice 30](#_Toc365205826)

[第八章 约定 30](#_Toc365205827)

[第九章 架构及扩展 30](#_Toc365205828)

[9.1 概述 30](#_Toc365205829)

[9.2 Settings体系 30](#_Toc365205830)

[9.3 Module体系 30](#_Toc365205831)

[9.4 Context体系 30](#_Toc365205832)

[9.5 Binder概念 30](#_Toc365205833)

[第十章 核心技术 30](#_Toc365205834)

[第十一章 Hasor-MVC软件包 30](#_Toc365205835)

[11.1 概述 30](#_Toc365205836)

[11.2 Controller模块 30](#_Toc365205837)

[11.2.1 介绍 31](#_Toc365205838)

[11.2.2 Action 31](#_Toc365205839)

[11.2.3 Action规则 31](#_Toc365205840)

[11.2.4 AbstractController抽象类 31](#_Toc365205841)

[11.2.5 RESTful支持 31](#_Toc365205842)

[11.2.6 Http动词与RESTful 31](#_Toc365205843)

[11.2.7 参数映射 31](#_Toc365205844)

[11.2.8 Action返回值处理 31](#_Toc365205845)

[11.2.9 Action返回值处理扩展 31](#_Toc365205846)

[11.3 Resource模块 31](#_Toc365205847)

[11.4 配置文件详解 31](#_Toc365205848)

# 第一章 介绍

Hasor是一款开源框架。它是为了解决企业模块化开发中复杂性而创建的。Hasor遵循简单的依赖、单一职责，在开发多模块企业项目中更加有调理。然而Hasor的用途不仅仅限于多模块项目开发。从简单性、松耦合性的角度而言，任何Java应用都可以从中受益。Hasor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。这一点与Spring比较相似。Hasor目前包含多个可选的子模块。

特点：

* 清晰：在Hasor体系中每一个模块都被封装到一个jar或者classpath路径中。
* 简单：少量的代码开发关键的部件，Hasor在开发上提供了强有力的粘合作用。此外，由于Hasor对外开发都是以接口形式提供开发者避免接触到大量无用的API。
* 容器：Hasor包含并管理每个模块对象的配置和生命周期。
* 轻量：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)。
* 友好：Hasor的所有功能仅在几个核心API接口和注解上实现。
* 兼容：对Web情况下进行了特殊制定，在开发Web项目时候可以得到更加友好的API支持。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。

设计思想：

模块：

Hasor-Core

核心包，所有Hasor模块都必须依赖它。对模块提供生命周期管理；以Settings接口形式提供配置文件获取服务；对主配置文件提供检测修改的支持；提供事件服务；提供环境变量操作接口；提供了Timer支持；提供了IoC/Aop。

Web支持下，可以注解声明Filter、HttpServlet、Session监听器、Aop拦截器；还提供了Servlet异常拦截器。通过Guice支持JSR-330。

Hasor-MVC

一个专门用于Web开发的模块，它提供了一个用于MVC模式下开发的请求控制器，通过它可以定义Action，并且可以将这个Action映射为RESTful，其API部分实现了JSR-311。

还有一个类路径资源装载器，用以加载位于Jar包中的资源。

## 1.1 概述

第一章是用以说明Hasor这个项目的功能、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。

第二章是引导读者使用Hasor进行项目开发，其中会包含丰富的Demo和API讲解。这一章节不会深入介绍Hasor部内部实现机制和原理。但是在第二章上会有许多知识点链接到第三章.

在第二章有兴趣的读者可以跟随链接深入了解Hasor内部实现机制和各部分功能原理。这些内容都在第二章以后的内容中出现。从第二章开始一直到第七章都是在详细讲解Hasor各个部件的功能和实现机制以及原理。

第八章之后会根据不用模块分别讲解各自模块下的功能。

## 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。

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)

# 第二章 使用Hasor

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：

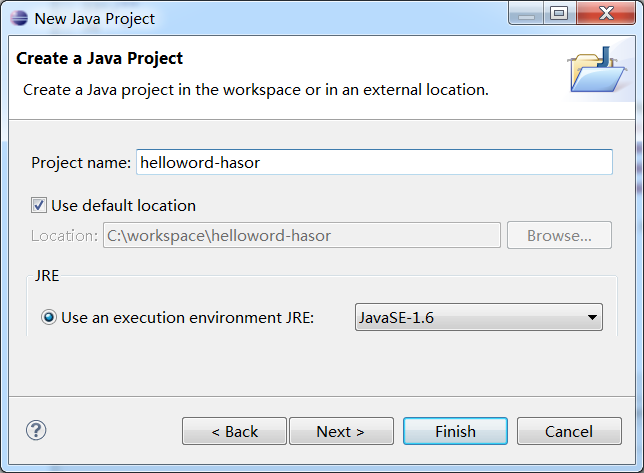
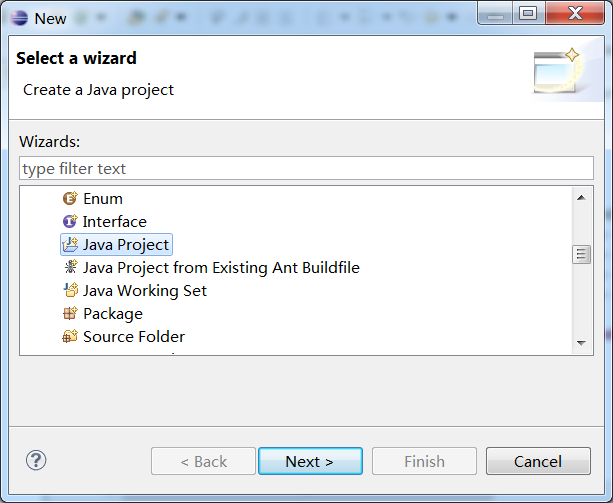
(<http://eclipse.org/downloads/packages/eclipse-ide-java-developers/keplerr>)

Eclipse IDE for Java EE Developers下载地址：

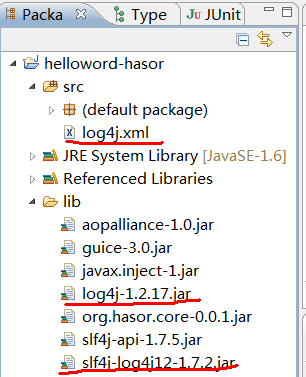
(<http://eclipse.org/downloads/packages/eclipse-ide-java-ee-developers/keplerr>)

## 2.1 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)

## 2.2 启动Hasor

执行如下代码启动Hasor：

**import** java.io.IOException;

**import** org.hasor.context.anno.context.AnnoAppContext;

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

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

//创建Hasor环境对象

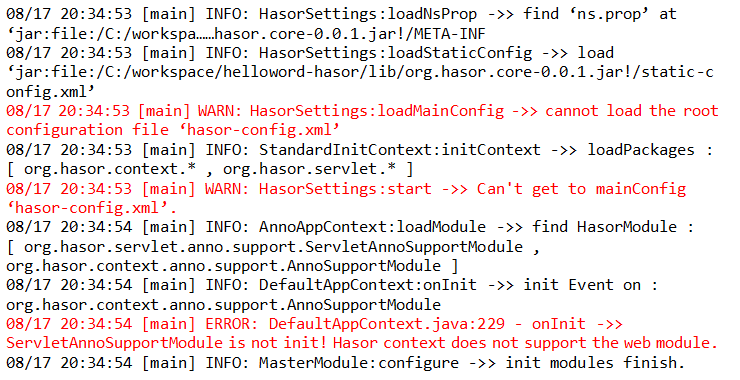
AnnoAppContext context = **new** AnnoAppContext();

context.start();//启动Hasor容器

}

}

在控制台中你应当能得到2个警告信息和1个错误信息，它们看上去应该是这样的：



*(上面日志信息略有删减)*

【解释】

第一条警告：是说，Hasor在启动时没有找到默认的主配置文件‘hasor-config.xml’。你可以不必关心它，**使用Hasor可以不定义任何配置文件**。这条警告消息仅仅是告诉开发者，如果你定义了配置文件Hasor并没有找到它。根据这条消息你可以判断Hasor是否加载了配置文件。[<第二章：快速上手(2.9节-使用配置文件)>](#_2.9__使用配置文件)、[<第五章：配置文件(5.3节-主配置文件)>](#_5.3__主配置文件(hasor-config.xml))

第二条警告：也是因为没有定义主配置文件引起的。Hasor会动态的监听配置文件变化，因此当没有主配置文件定义时候，负责监听主配置文件的监听程序会报告这一条警告消息。[<第五章：配置文件(5.6节-配置文件监听器)>](#_5.6__配置文件监听器)

第三条Error：由于我们的第一个例子并非是Web程序，因此Hasor内部的一个Web支持模块不满足启动条件。这对于控制台例子而言可以不必关心它。[<第六章：Web支持(6.1节-简介)>](#_5.6__配置文件监听器)

## 2.3 第一个模块(HelloWord)

下面定义了一个模块。当启动时Hasor会调用模块的init方法初始化方法打印一条消息。任何继承自AbstractHasorModule抽象类或者实现了HasorModule接口的类都可以作为Hasor的模块。[<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.”

## 2.4 Bean

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

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

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

}

}

### 2.4.1 注册Bean

在Hasor中任何类都可以被视为Bean。Hasor将其区分为注册Bean和非注册Bean。注册Bean不同于非注册Bean的最大特征是Bean是否具备名字。注册的Bean可以同时具备多个名字，并且支持根据名字从Hasor中获取实例对象。下面是注册Bean的两种方式。

方式一：在模块的init阶段使用ApiBinder的方法注册Bean，并且命名为“myBean”。这种方法比较隐蔽不会受到classpath包扫描范围设置的影响。[<第四章：环境支持(4.2节-类路径扫描范围)>](#_4.2__类路径扫描范围_1)

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

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

}

方式二：最简单的注册方式！使用@Bean注解将Bean注册到Hasor容器中。AnnoAppContext会启用注解扫描。[<第四章：环境支持(4.1.3节- AnnoAppContext)>](#_4.1.3__AnnoAppContext)

@Bean("myBean")

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

……

### 2.4.2 获取Bean

对于注册到Hasor中的Bean可以使用下面这个方法获取Bean对象。

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

对于没有注册到Hasor的Bean可以使用下面这个方法借助Hasor创建实例对象。

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

### 2.4.3 单例Bean

代码形式：如果使用代码形式注册的Bean可以采用下面这个代码将其声明为单例。

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

.asEagerSingleton();

注解形式：使用JSR-330标准将Bean声明为单例。

**import** javax.inject.Singleton;//JSR-330，来源于javax.inject-1.jar

**import** org.hasor.context.anno.Bean;

@Bean("myBean")

@Singleton/\*声明单例\*/

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

……

提示：以上两种方式可以重叠使用。

### 2.4.4 比较注册Bean非注册Bean

下面是一些比较：

|  |  |  |
| --- | --- | --- |
| 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 | 是 | 是 |
| IoC/Aop | 是 | 是 |
| 被注册Bean注入 | 是 | 是 |
| 被非注册Bean注入 | 是 | 是 |

更多的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))以了解更多信息。

## 2.5 依赖注入(IoC)

Hasor具有强大的依赖注入功能，这是由于Hasor使用小巧的Google Guice3.0作为其DI支持[<第七章：Guice>](#_Guice)。同时Google Guice3.0又是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。下面详细介绍如何在Hasor中使用JSR-330实现IoC。

### 2.5.1 属性注入

@Bean("testBean")

**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();

}

}

### 2.5.2 构造方法注入

下面这个例子是使用构造方法将CustomBean注入到TestBean中：

**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(); }

}

### 2.5.3 非注册Bean的依赖注入

TestBean2是一个非注册Bean所以使用Context.getInstance(Class)方法获取Bean对象：

AnnoAppContext context = **new** AnnoAppContext();

context.start();

TestBean2 bean = context.getInstance(TestBean2.**class**);

bean.callFoo();

## 2.6 Aop拦截器(MethodInterceptor)

在Hasor中可以使用代码方式(4小段:会详细讲解使用代码方式配置拦截器)和注解方式进行Aop编程。首先编写拦截器，该拦截器会在方法调用之前打印一条消息。MethodInterceptor接口是来源于Aop联盟aopalliance-1.0.jar包。详细信息参见[<第七章：Guice>](#_第七章__Guice)。

PS：如果想要一个线程安全的拦截器请在MethodInterceptor接口的invoke方法上加上同步关键字。下面定义了三个拦截器(非线程安全)，代码如下：

**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

}

}

### 2.6.1 方法级

@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;

}

}

### 2.6.2 类级

@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。

### 2.6.3 全局

在模块的init过程使用ApiBinder接口注册全局Aop拦截器，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注解被处理之前初始化Aop模块，作用是提前注册全局拦截器\*/

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

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

}

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

/\*排除所有拦截器\*/

/\*在这个例子中配置成任意类型任意方法会导致\*/

/\*AopInterceptor\_1、AopInterceptor\_2拦截器对象的方法被\*/

/\*AopInterceptor\_3代理，造成AopInterceptor\_3被多次调用。\*/

Matcher m = Matchers.*not*(

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

apiBinder.getGuiceBinder().bindInterceptor(

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

}

}

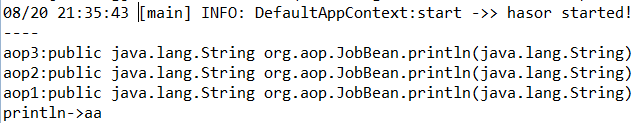
获取被代理的JobBean对象并调用foo方法：

JobBean job = context.getBean("jobBean");

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

job.println("aa");//System.out.println(job.foo("abc"));

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拦截器：

(全局->类级->方法级)

### 2.6.4 拦截范围

下面实例代码列出了部分常用的拦截器范围在[<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如何使用拦截器。

//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

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

## 2.7 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。同时使用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))

### 2.7.1 抛出和监听

下面这段代码展示了如何编写一个事件监听器，该监听器监听“HelloEvent”事件，并且打印接收到的事件参数中第一个参数。[<第四章：环境支持(4.3.2节-自定义事件)>](#_4.3.2__自定义事件)

@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。在引发事件时，传递一个参数。

AnnoAppContext context = **new** AnnoAppContext();

context.start();

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

### 2.7.2 同步事件

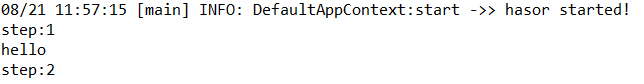
同步事件是指，当事件引发之后。等待所有事件监听器执行完毕返回在继续执行后面的代码。例如下面代码：[<第四章：环境支持(4.3.5节-同步事件)>](#_4.3.5__同步事件)

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

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

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

执行结果:



### 2.7.3 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。调用程序可以在第一时间继续执行后续代码。例如下面代码：[<第四章：环境支持(4.3.6节-异步事件)>](#_4.3.6__异步事件)

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

context.getEventManager().doAsynEventIgnoreThrow(

"HelloEvent", "hello");

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

执行结果:



## 2.8 计时器(Timer)

计时器的定义目前采用的是Hasor事件接口“HasorEventListener”下面定义了一个计时器：

**public** **class** TimerEventListener **implements** HasorEventListener {

**public** **int** i = 0;

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

System.*out*.println(event + ":" + (i++));

}

}

下面这段代码是一段完整的使用Timer计时器的例子：

**public** **class** TimerEventTest {

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

AnnoAppContext context = **new** AnnoAppContext();

context.start();//

context.getEventManager().addTimer(

"TestTimer", **new** TimerEventListener());

Thread.*sleep*(5000);

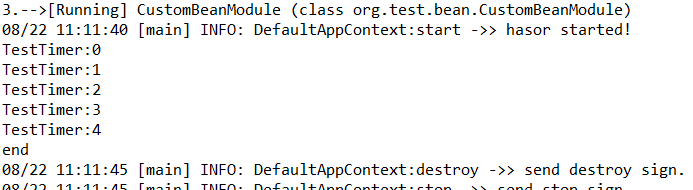
System.*out*.println("end");

context.destroy();//注意，必须销毁

}

}

下面是执行结果：



注意事项：

1.Timer计时器的调度时间间隔默认1秒，这个数值需要在配置文件中配置。

2.目前不同的Timer不可以分别设置不同的触发时间。

3.使用Timer的程序必须使用context.destroy()方法销毁容器，否则Timer线程不会被终止。[<第四章：环境支持(4.3.3节-计时器(Timer))>](#_4.3.3__计时器(Timer))

## 

## 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。

### 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。[<第三章：模块依赖(3.4.1节-强依赖)>](#_3.4.1__强依赖)

下面使用例子来说明这个问题：

**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

}

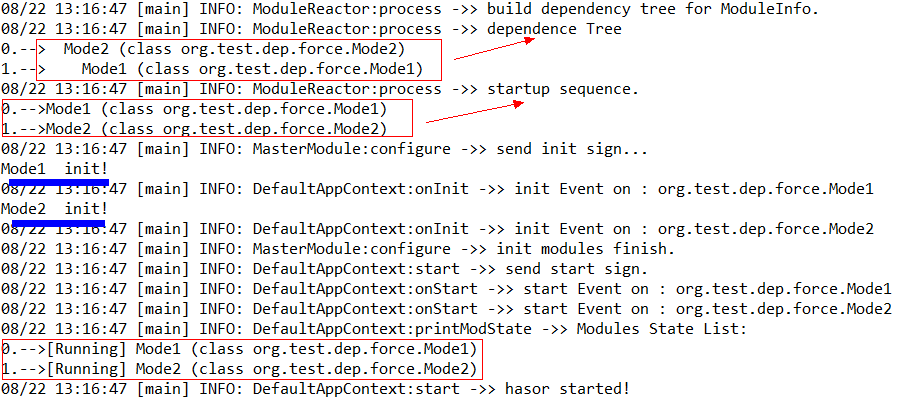
**public** **void** init(ApiBinder apiBinder) {

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

}

}

分析启动日志：



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，在这种依赖关系中强调依赖顺序，不强调必然关系。好比领导和员工，领导没了员工照样存在。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。[<第三章：模块依赖(3.4.2节-弱依赖)>](#_3.4.2__弱依赖)

修改上面例子中的代码：

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

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

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

}

}

分析启动日志：



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。这就是弱依赖的作用：**强调关系不强调必然联系**。

### 2.9.3 依赖反制

在Hasor中使用依赖需要在模块中写明代码。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，在不修改第三方模块代码的情况下将你的模块加到目标模块的依赖树中。[<第三章：依赖反制(3.4.3节-依赖反制)>](#_3.4.3__依赖反制)

我们假定JFinalMode模块依赖HibernateMode，现在不修改这两个第三方模块的前提下让JFinal模块依赖我们的模块。

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

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

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

}

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

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

}

}

这样一来依赖顺序就变为JFinalMode依赖：HibernateMode、MyMode。

## 2.10 使用配置文件

在Hasor中配置文件被分为主配置文件和静态配置文件。有关配置文件更多的信息可以通过[<第五章：配置文件>](#_第五章__配置文件)进行详细了解。本小节主要介绍主配置文件。[<第五章：配置文件(5.3节-主配置文件)>](#_5.3__主配置文件(hasor-config.xml))

Hasor的配置文件需要放到ClassPath中文件名为“hasor-config.xml” 全小写文件名：

<?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：

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

<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</name>

<age>27</age>

<address>

<name>北京市海淀区...</name>

<zip>100000</zip>

</address>

</userInfo>

</config>

**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... |
| userInfo.address.zip | 100000 |

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）。

突破限制：

先有如下配置文件，根据限制无法正确取到用户001的姓名。接下来展示了一段代码用以突破这种限制取得想要的内容。

<?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>

下面的代码是利用Settings接口获取Xml文档根节点并以DOM的方式操作：

/\*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);

}

}

}

约定：

Hasor的配置文件比较随意，它只要求一个命名空间。即使是这样在使用Hasor之前您最好还是遵照Hasor约定建立一个属于这个项目或模块的标签，然后将配置信息都放到这个标签中。这样做的好处是可以最大程度上避免和其他第三方模块所使用的配置重叠。

有关约定的资料在[<第八章：约定>](#_第八章__约定)有详细介绍。

提示：

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。

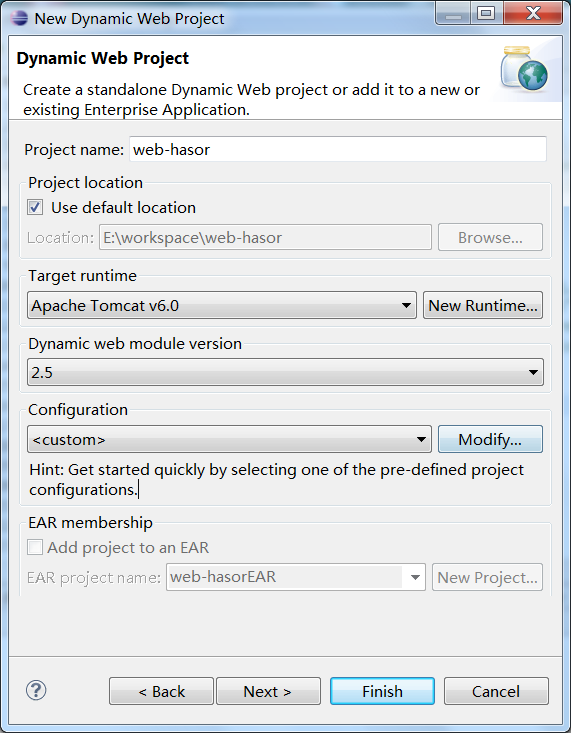
environmentVar ：这个标签是用来配置Hasor环境变量

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

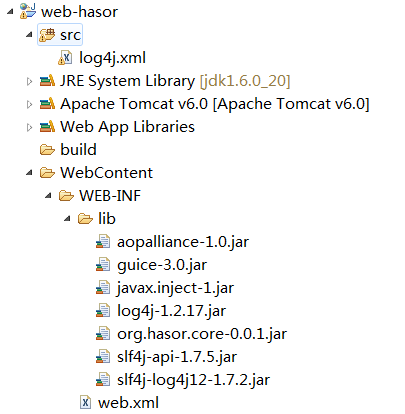
## 2.11 创建Web项目

Step 1：创建Dynamic Web Project项目

输入项目名称点击Finish完成即可，如您对项目有特殊制定，可以按照项目需要进行特殊配置。目前Hasor对Web工程没有特殊配置要求。



Step 2：加入Hasor的Jar包，并引入lib路径.下面图中除了Hasor必须依赖的Jar之外还增加了Log4j相关日志的配置.



如果项目还有其他第三方依赖也同时加入lib目录即可。秉承Hasor简化配置的思想，在演示基本Web程序开发时仍然不使用配置文件。

Step 3：修改web.xml配置文件加入如下内容即完成配置。

<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>

**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”。**

## 2.12 Web开发

### 2.12.1 HttpServlet

在Hasor中定义HttpServlet是及其容易的：下面代码展示了定义一个Servlet

**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();

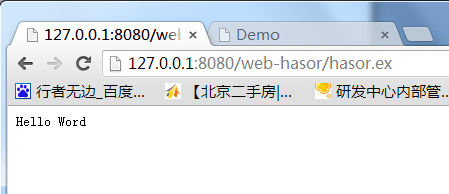
w.write("Hello Word");

w.flush();

}

}

启动Web工程输入Servlet地址你会看到如下内容：“http://127.0.0.1:8080/<your project name>/hasor.ex” [<第六章：Web支持(6.3节-HttpServlet)>](#_6.3__HttpServlet(@WebServlet))



### 2.12.2 Filter

在Hasor中定义Filter也及其容易：下面代码展示了定义一个Filter

**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/>");

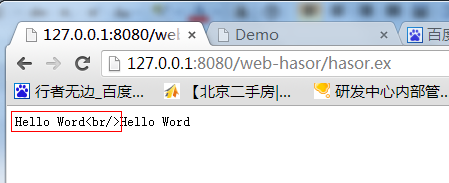
arg2.doFilter(arg0, arg1);

}

……

}

再次启动Web工程重新访问Servlet你会看到如下内容：“http://127.0.0.1:8080/<your project name>/hasor.ex” [<第六章：Web支持(6.4节-Filter)>](#_6.4__Filter过滤器(@WebFilter))



### 2.12.3 Session监听器(HttpSessionListener)

为了测试Session监听器，创建一个index.jsp文件放到网站跟目录。当访问这个JSP时，jsp会创建一个Session。这正好满足了我们的要求

然后在Web工程中增加如下代码： [<第六章：Web支持(6.5.1节-Session监听器)>](#_6.5.1__Session监听器(@WebSessionListen)

@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);

}

}

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)

在Hasor中还可以注册ServletContextListener接口，更多信息可以参看[<第六章：Web支持(6.5.2节-Context监听器)>](#_6.5.2__Context监听器(@WebContextListen)

下面是示例代码：

@WebContextListener//声明ServletContextListener

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

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

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

}

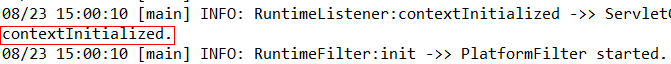
**public** **void** contextInitialized(ServletContextEvent arg0) {

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

}

}

启动程序会在控制台得到类似下面这样的输出：



### 2.12.5 截获服务器异常

该服务是用来拦截意外的异常抛出，支持根据抛出的异常类型绑定处理程序。当异常处理器也出现异常抛出时该截获服务支持二次截获。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))为了让拦截器起到作用首先在Servlet中故意抛出一条异常信息：

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

@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();

}

}

@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，在Web-MVC功能支持中分为Controller、Resource两大部分。Controller：提供Action控制器，RESTful支持。Resource提供了允许Web程序获取资源可以来源于某个Jar包或目录。[<第十一章：Hasor-MVC软件包>](#_第十一章__Controller模块(Hasor-MVC))有详细介绍。

### 2.13.1 Action

新建一个类输入如下代码：

@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!");

}

}

在浏览器输入如下地址：http://localhost:8080/<projectName>/abc/123/print.do你会看到控制台打印出HelloAction。[<第十一章：Hasor-MVC软件包(11.2.2节-Action)>](#_11.2.2__Action)

规则：

1.Action类中所有共有方法都可以被访问。（私有、受保护）不会被暴露成Action。

2.可以继承AbstractController抽象类，以获得更多的功能支持。

3.Action不支持重载方法。

4.Action扩展名“\*.do”可以通过hasor-config.xml配置文件修改。

5.Action支持单例、支持Ioc/Aop，可以使用@Before注解。

### 2.13.2 RESTful映射

在Hasor中RESTful风格映射参照了JSR-311标准，在未来版本中Hasor会强化JSR-311标准的实现。下面这段代码是一个简单的RESTful映射。[<第十一章：Hasor-MVC软件包(11.2.5节-RESTful支持)>](#_11.2.5__RESTful支持)会有更多的信息。

@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）。

@Any/\*Http动词，如果为Any则可以忽略不写。\*/

/\*下面配置的是，内容类型，可以不写，默认为“\*/\*”可通过配置文件配置\*/

@Produces("text/html")

@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));

}

【参数】：[<第十一章：Hasor-MVC软件包(11.2.7节-参数映射)>](#_11.2.7__参数映射)

@AttributeParam

可以标记在任意类型参数上，其功能相当于request.getAttribute(xxx)。

@CookieParam

可以标记在任意类型参数上，其功能相当于从Cookie中获取内容。

@HeaderParam

可以标记在任意类型参数上，其功能相当于request.getHeader(xxx)。

@InjectParam

可以标记在任意类型参数上，其功能相当于context.getInstance(XXX.class)。

@PathParam

可以标记在任意类型参数上，用于获取在@Path注解中用“{...}”括起来的内容。

@QueryParam

可以标记在任意类型参数上，用于获取URL请求地址“?”后面的参数。

【Http动词】：[<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)

@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。

### 2.13.3 获取Request和Response

在Hasor-MVC中获取Request和Response可以使用两种方式，通过继承抽象控制器类可以使用提供的受保护方法getRequest()和getResponse()获取。

@Controller("/abc/123")

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

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

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

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

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

}

}

下面这段代码也可以帮助你获取到Request和Response，但是这种方式只能在请求响应过程中起到作用。

HttpProvider provider = HttpProvider.*getProvider*();

HttpServletRequest req = provider.getRequest();

HttpServletResponse res = provider.getResponse();

更多支持：[<第十一章：Hasor-MVC软件包(11.2.4节-AbstractController抽象类)>](#_11.2.4__AbstractController抽象类)

### 2.13.4 返回Json数据

@Controller("/abc/123")

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

@Json()

**public** Object getData() {

**return** ……;

}

}

在Hasor中由Action返回一个JSON数据是十分容易的一件事，你只要在Action方法中将要回写到客户端的JSON数据对象作为方法返回值，然后在方法上标记一个@Json注解即可。

### 2.13.5 Action结果处理

利用Hasor的返回值处理可以扩展Action后续操作下面列举了Hasor-MVC模块内内置的可用返回值操作。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)中你可以通过里面的知识自定义返回值处理。

@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。

## 2.14 把Web资源打入Jar包

# 第三章 模块

## 3.1 定义

## 3.2 生命周期(HasorModule)

## 3.3 模块运行状态(ModuleInfo)

## 3.4 模块依赖(Dependency)

### 3.4.1 强依赖

### 3.4.2 弱依赖

### 3.4.3 依赖反制

### 3.4.4 依赖算法

## 3.5 一个简单的模块示例(Simple)

## 3.6 复杂模块Demo演示

# 第四章 环境支持

## 4.1 环境(Context)

### 4.1.1 StandardInitContext

### 4.1.2 DefaultAppContext

### 4.1.3 AnnoAppContext

### 4.1.4 AnnoWebAppContext

## 4.2 类路径扫描范围

### 4.2.1 设置扫描范围

## 4.3 事件(EventManager)

### 4.3.1 系统生命周期事件

### 4.3.2 自定义事件

### 4.3.3 计时器(Timer)

### 4.3.4 只生效一次的事件监听器

### 4.3.5 同步事件

### 4.3.6 异步事件

### 4.3.7 理解事件中的事件

## 4.4 Binder

### 4.4.1 ApiBinder

### 4.4.2 WebApiBinder

### 4.4.3 Binder和Guice

### 4.4.4 Binder的有效范围

## 4.5 Bean服务(@Bean)

## 4.6 Aop拦截器服务(@Before)

## 4.7 启动和销毁

## 4.8 环境变量

# 第五章 配置文件

## 5.1 简介

## 5.2 独立性&随意性(Settings)

### 5.2.1 与其他配置文件混杂

## 5.3 主配置文件(hasor-config.xml)

## 5.4 静态配置文件(static-config.xml)

## 5.5 配置名称映射文件(config-mapping.properties)

## 5.6 配置文件监听器

## 5.7 命名空间(NameSpace)

## 5.8 配置文件简单操作(Simple)

## 5.9 自定义Xml配置文件解析器

# 第六章 Web支持

## 6.1 简介

## 6.2 Web模块

## 6.3 HttpServlet(@WebServlet)

## 6.4 Filter过滤器(@WebFilter)

## 6.5 监听器

### 6.5.1 Session监听器(@WebSessionListener)

### 6.5.2 Context监听器(@WebContextListener)

## 6.6 Servlet异常截获(@WebError)

# 第七章 Guice

# 第八章 约定

# 第九章 架构及扩展

## 9.1 概述

## 9.2 Settings体系

## 9.3 Module体系

## 9.4 Context体系

## 9.5 Binder概念

# 第十章 核心技术

# 第十一章 Hasor-MVC软件包

## 11.1 概述

## 11.2 Controller模块

### 11.2.1 介绍

### 11.2.2 Action

### 11.2.3 Action规则

### 11.2.4 AbstractController抽象类

### 11.2.5 RESTful支持

### 11.2.6 Http动词与RESTful

### 11.2.7 参数映射

### 11.2.8 Action返回值处理

### 11.2.9 Action返回值处理扩展

## 11.3 Resource模块

## 11.4 配置文件详解