**Hasor 使用手册**

**手册版本：1.0.0**

**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](#_Toc364858449)

[1.1 概述 6](#_Toc364858450)

[1.2 开源协议 6](#_Toc364858451)

[1.3 内置组建及其授权协议 6](#_Toc364858452)

[1.4 获取和贡献 7](#_Toc364858453)

[1.5 类库引用 7](#_Toc364858454)

[第二章 使用Hasor 8](#_Toc364858455)

[2.1 创建项目 8](#_Toc364858456)

[2.2 启动Hasor 9](#_Toc364858457)

[2.3 第一个模块(HelloWord) 10](#_Toc364858458)

[2.4 Bean 10](#_Toc364858459)

[1)注册Bean 10](#_Toc364858460)

[2)获取Bean 10](#_Toc364858461)

[3)单例Bean 11](#_Toc364858462)

[4)比较注册Bean非注册Bean 11](#_Toc364858463)

[2.5 依赖注入(IoC) 11](#_Toc364858464)

[1)属性注入 11](#_Toc364858465)

[2)构造方法注入 12](#_Toc364858466)

[3)非注册Bean的依赖注入 12](#_Toc364858467)

[2.6 Aop拦截器(MethodInterceptor) 12](#_Toc364858468)

[1)方法级 13](#_Toc364858469)

[2)类级 13](#_Toc364858470)

[3)全局 13](#_Toc364858471)

[4)拦截范围 14](#_Toc364858472)

[2.7 事件的抛出和监听(Event) 15](#_Toc364858473)

[1)抛出和监听 15](#_Toc364858474)

[2)同步事件 15](#_Toc364858475)

[3)异步事件 15](#_Toc364858476)

[2.8 计时器(Timer) 16](#_Toc364858477)

[2.9 一个模块依赖另外一个模块 16](#_Toc364858478)

[2.10 使用配置文件 16](#_Toc364858479)

[2.11 创建Web项目 16](#_Toc364858480)

[2.12 Filter 16](#_Toc364858481)

[2.13 HttpServlet 16](#_Toc364858482)

[2.14 Session监听器 16](#_Toc364858483)

[2.15 截获服务器异常 16](#_Toc364858484)

[2.16 Action 16](#_Toc364858485)

[2.17 RESTful映射 16](#_Toc364858486)

[2.18 获取Request和Response 16](#_Toc364858487)

[2.19 返回Json数据 16](#_Toc364858488)

[2.20 Action结果处理 16](#_Toc364858489)

[2.21 把Web资源打入Jar包 16](#_Toc364858490)

[第三章 模块 16](#_Toc364858491)

[3.1 定义 16](#_Toc364858492)

[3.2 模块生命周期(HasorModule) 16](#_Toc364858493)

[3.3 模块运行状态(ModuleInfo) 16](#_Toc364858494)

[3.4 模块依赖(Dependency) 16](#_Toc364858495)

[3.5 一个简单的模块示例(Simple) 16](#_Toc364858496)

[3.6 复杂模块Demo演示 16](#_Toc364858497)

[第四章 环境支持 16](#_Toc364858498)

[4.1 环境(Context) 16](#_Toc364858499)

[4.1.1 StandardInitContext 16](#_Toc364858500)

[4.1.2 DefaultAppContext 16](#_Toc364858501)

[4.1.3 AnnoAppContext 16](#_Toc364858502)

[4.1.4 AnnoWebAppContext 16](#_Toc364858503)

[4.2 类路径扫描范围 16](#_Toc364858504)

[4.2.1 设置扫描范围 16](#_Toc364858505)

[4.3 事件(EventManager) 16](#_Toc364858506)

[4.3.1 系统生命周期事件 16](#_Toc364858507)

[4.3.2 自定义事件 16](#_Toc364858508)

[4.3.3 计时器和Timer事件 16](#_Toc364858509)

[4.3.4 只生效一次的事件监听器 16](#_Toc364858510)

[4.3.5 同步事件 16](#_Toc364858511)

[4.3.6 异步事件 16](#_Toc364858512)

[4.3.7 理解事件中的事件 16](#_Toc364858513)

[4.4 Binder 16](#_Toc364858514)

[4.4.1 ApiBinder 17](#_Toc364858515)

[4.4.2 WebApiBinder 17](#_Toc364858516)

[4.4.3 Binder和Guice 17](#_Toc364858517)

[4.4.4 Binder的有效范围 17](#_Toc364858518)

[4.5 Bean服务(@Bean) 17](#_Toc364858519)

[4.6 Aop拦截器服务(@Before) 17](#_Toc364858520)

[4.7 启动和销毁 17](#_Toc364858521)

[第五章 配置文件 17](#_Toc364858522)

[5.1 简介 17](#_Toc364858523)

[5.2 独立性&随意性(Settings) 17](#_Toc364858524)

[5.3 主配置文件(hasor-config.xml) 17](#_Toc364858525)

[5.4 静态配置文件(static-config.xml) 17](#_Toc364858526)

[5.5 配置名称映射文件(config-mapping.properties) 17](#_Toc364858527)

[5.6 配置文件监听器 17](#_Toc364858528)

[5.7 命名空间(NameSpace) 17](#_Toc364858529)

[5.8 配置文件简单操作(Simple) 17](#_Toc364858530)

[5.9 自定义Xml配置文件解析器 17](#_Toc364858531)

[第六章 Web支持 17](#_Toc364858532)

[6.1 简介 17](#_Toc364858533)

[6.2 Web模块(WebApiBinder) 17](#_Toc364858534)

[6.3 Filter过滤器(@WebFilter) 17](#_Toc364858535)

[6.4 HttpServlet(@WebServlet) 17](#_Toc364858536)

[6.5 Session监听器(@WebSessionListener) 17](#_Toc364858537)

[6.6 Servlet异常截获(@WebError) 17](#_Toc364858538)

[第七章 Guice 17](#_Toc364858539)

[第八章 架构及扩展 17](#_Toc364858540)

[8.1 概述 17](#_Toc364858541)

[8.2 Settings体系 17](#_Toc364858542)

[8.3 Module体系 17](#_Toc364858543)

[8.4 Context体系 17](#_Toc364858544)

[8.5 Binder概念 17](#_Toc364858545)

[第九章 核心技术 17](#_Toc364858546)

[8.1 17](#_Toc364858547)

[8.2 17](#_Toc364858548)

[第十章 Controller模块(Hasor-MVC) 18](#_Toc364858549)

[8.1 18](#_Toc364858550)

[第十一章 Resource模块 (Hasor-MVC) 18](#_Toc364858551)

[9.1 18](#_Toc364858552)

# 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：

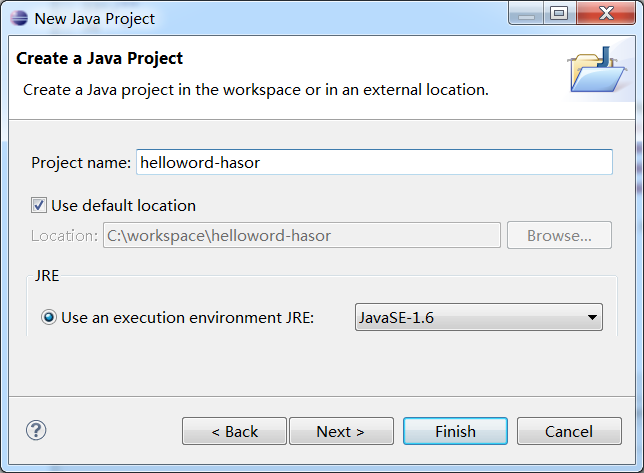
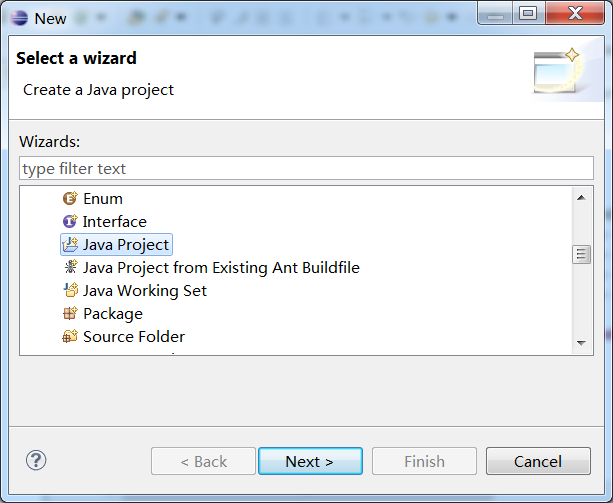
(<http://eclipse.org/downloads/packages/eclipse-ide-java-developers/keplerr>)

Eclipse IDE for Java EE Developers下载地址：

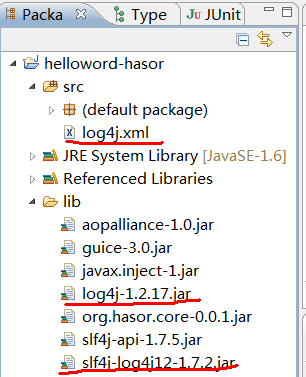
(<http://eclipse.org/downloads/packages/eclipse-ide-java-ee-developers/keplerr>)

## 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

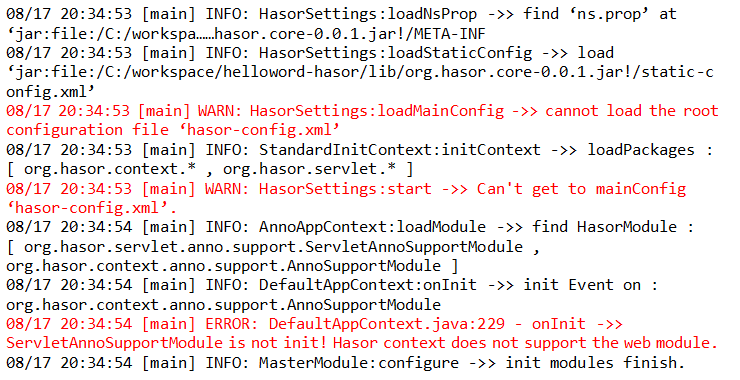
AnnoAppContext context = **new** AnnoAppContext();

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");

}

}

### 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)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**);

### 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。

### 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。

### 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)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(); }

}

### 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

}

}

### 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)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。

### 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());

}

}

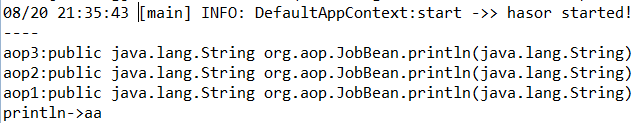
获取被代理的JobBean对象并调用foo方法：

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)

### 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))

### 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)同步事件

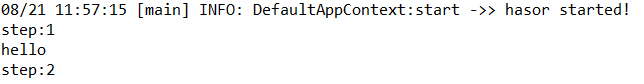
同步事件是指，当事件引发之后。等待所有事件监听器执行完毕返回在继续执行后面的代码。例如下面代码：[<第四章：环境支持(4.3.5节-同步事件)>](#_4.3.5__同步事件)

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

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

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

执行结果:



### 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)

## 2.9 一个模块依赖另外一个模块

## 2.10 使用配置文件

## 2.11 创建Web项目

## 2.12 Filter

## 2.13 HttpServlet

## 2.14 Session监听器

## 2.15 截获服务器异常

## 2.16 Action

## 2.17 RESTful映射

## 2.18 获取Request和Response

## 2.19 返回Json数据

## 2.20 Action结果处理

## 2.21 把Web资源打入Jar包

# 第三章 模块

## 3.1 定义

## 3.2 模块生命周期(HasorModule)

## 3.3 模块运行状态(ModuleInfo)

## 3.4 模块依赖(Dependency)

## 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 计时器和Timer事件

### 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

# 第五章 配置文件

## 5.1 简介

## 5.2 独立性&随意性(Settings)

## 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(WebApiBinder)

## 6.3 Filter过滤器(@WebFilter)

## 6.4 HttpServlet(@WebServlet)

## 6.5 Session监听器(@WebSessionListener)

## 6.6 Servlet异常截获(@WebError)

# 第七章 Guice

# 第八章 架构及扩展

## 8.1 概述

## 8.2 Settings体系

## 8.3 Module体系

## 8.4 Context体系

## 8.5 Binder概念

# 第九章 核心技术

## 8.1

## 8.2

# 第十章 Controller模块(Hasor-MVC)

## 8.1

# 第十一章 Resource模块 (Hasor-MVC)

## 9.1

开发者可以摘除More代码使用Apsche的commons-lang、commons-beans软件包作为替换，但需要改写一些源程序。如果开发者这样做