Hasor 权威指南

手册版本：0.0.2

软件版本：0.0.10

作者：赵永春(zyc@hasor.net)

日期：2015-01-08

目录

第一章　介绍

Hasor 是一款开源的轻量级 Java 应用程序开发框架，它的核心目标是提供一个简单、切必要的开发环境给开发者，开发者可以在此基础上构建出更加完善的应用程序。它由多个不同功能的软件包组合而成，您可以根据需要去选择它们。Hasor 通过各种不同功能的扩展插件来加速您的开发效率。

本质上 Hasor 与 Struts、Hibernate 等单层框架不同。它是由一个及其微小的核心和强有力的外围插件扩展组合而成，也就是所谓的“**微内核 + 插件**”这种方式。

Hasor 将应用程序的启动分为 “**init、start**” 两个阶段。插件在 init 阶段可以为应用程序的正式执行创造必要的条件，同时开发者也可以在这一过程中初始化程序的必要环境。最后在 **start** 阶段开启程序的正式旅程。

与 Spring 不同的是 Hasor 更加注重框架在使用中的“**编程性**”让开发者更多的可以控制到框架的内部。而非通过“**Xml、注解**”等方式提供的扩展方式，相比这类框架 Hasor 会显得更加简单和灵活。

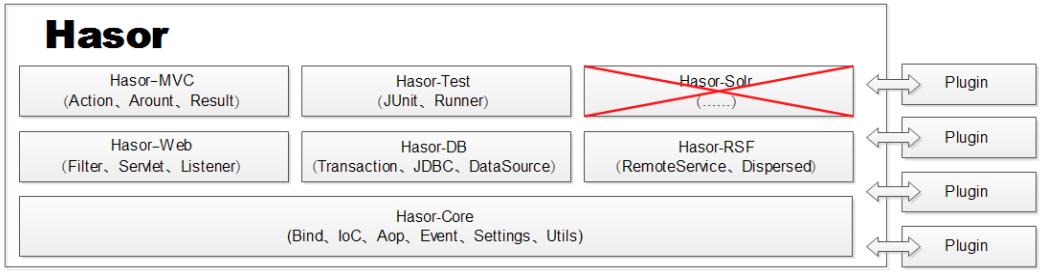
特点：

- **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：** 身材较小而内心强大（500 多 K 的体积内除了框架之外还携带了大量工具）
- **灵活性：** Hasor 更加注重编程性而非声明式配置，因此它允许你更大范围的控制整个应用程序乃至框架，这要比基于配置的框架显得更加简单灵活。
- **零配置：** 真正的零配置，完全不需要配置文件。当然如果你需要配置文件存放一些程序配置 Hasor 还为你准备一个配置文件工具。

设计思想：

“微内核 + 插件”是 Hasor 的主体设计思想，Binder 是 Hasor 提供的统一扩展 API。Hasor 的所有扩展程序全部以 Module 插件的形式安插到启动过程中，顺序由开发者决定。通过插件丰富 Hasor 功能。从而协调整个应用程序的运行，作为插件是可以随时被剔除的。

HASOR 体系架构：



Hasor 框架本质上是围绕在 Hasor-Core 项目提供的扩展体系上建立的一系列技术整合。从架构上也可以看出这个特征，因此这也是 Hasor 框架名称的由来。

被整合进来的技术通常是以“Hasor-XXX”的方式来命名。当然也有一些 Hasor 原生的技术例如：Hasor-RSF、Hasor-MVC 等。这些原生技术不一定要必须跑在 Hasor-Core 之上，但是要求必须能够提供一种与 Hasor-Core 整合的方式。

主要模块：

HASOR-CORE----*目前最新版本：0.0.10*

Hasor 的核心软件包，几乎所有 Hasor 扩展模块都会依赖到它。包含工具和 Hasor 两个部分，它是整个 Hasor 蓝图的基础。该软件包提供了：配置文件解析、事件、容器、IoC/Aop 等核心功能。

HASOR-WEB----*目前最新版本：0.0.8*

Hasor-Web 是参照 guice-servlet 进行的仿制，其中 Hasor 在仿制过程中做了大量改进优化。这使得 Hasor-Web 具有了更多很多优越的特性。同时它也是 Hasor 作为 Web 上开发框架的一个基石。几乎后续所有的 Web 模块都会依赖到它。开发者使用它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 入口。

HASOR-DB----*目前最新版本：0.0.3*

一个轻量化的数据库操作框架，该框架主要目的是为 Hasor 提供基于 JDBC 接口的数据库访问功能。前身是 Hasor-JDBC 项目，该项目中包含了： Transaction 、 JDBC 、DataSource 、 ORM 四个部分。这四个组建又互相成为一个体系。整个 Hasor-DB ，可以被独立使用。其中 Transaction 和 JDBC 是两个最重要的部件。它们的设计参考了 SpringJDB 可以说 Hasor-DB 是缩小版的 SpringJDBC ，拥有 SpringJDBC 绝大部分功能。

HASOR-MVC----*目前最新版本：0.0.2*

一个轻量化的 MVC 框架，它的被分为两个部分一个部分可以用于非 Web 下的 MVC 模式开发，而另一个重要的部分就是 Web 下的 MVC 开发。它是注解化的开发框架，开发者需要通过@MappingTo 来定义控制器，而 MappingTo 的表达式中可以配置请求参数。

Hasor-MVC 的 Web 部分还提供了 Restful、Strategy、Result 等支持，通过这些扩展可以使 Web 开发更加简单轻松。

HASOR-RSF----*目前最新版本：0.0.1*

该模块是 Hasor 提供的一个高性能分布式远程服务框架，它基于 Netty 构建。RSF 的设计吸收了淘宝 HSF 框架的精华，其关键部分小巧而精悍。去中心化是 RSF 的一大特点，使用 RSF 可以最大限度的避免服务注册中心，这在一些较小规模的项目中使用起来更加方便。当然 RSF 也会有自己的服务注册中心，目前该项目刚刚起步。

HASOR-TEST----*目前最新版本：0.0.3*

Hasor 基于 Junit4 的一个小型测试框架，目前功能还比较弱。

HASOR-QUICK----*目前最新版本：0.0.1*

提供了一组语法糖或者各类方面开发的小工具，让基于 Hasor 的开发更加快捷。

HASOR-SEARCH ----*目前最新版本：0.0.1*

该项目是基于 Sorl4 构建的搜索引擎，目前只是将 Solr 的 solr4j 接口封装为 Search 的几个核心接口，远程调用通过 RSF 实现。目前功能还比较弱，它的目的是使开发者在客户端调用远程搜索服务时就像使用本地服务一样畅快淋漓。

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。还有各个章节的内容引导，同时说明了 Hasor 所使用的开源协议、类库以及如何索取和贡献代码。

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，跟随本章的内容读者最后会完成一个简单的项目。

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以后的内容。

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。同时本章将会给您介绍 Hasor 开发接口背后的故事。阅读本章有助于您深入了解 Hasor。如果您想深入 Hasor 或者制定 Hasor 一个专属的开发框架，那么本章和第五章将会是最佳的选择。

第五章：扩展

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开发 Hasor 扩展模块，在本章您会更加深入的接触到 Hasor 的配置体系和扩展体系。

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会用到配置文件，即使您用代码的方式没有使用配置文件。那么本章的内容也会有助于您了解到 Hasor 都有什么配置，以及这些配置的作用。其实更多的会是 Hasor 组建的配置。

第七章：API 参考手册

本章将一一详细讲解 Hasor 所有接口的方法功能和作用。同时本章还会提供一个常用 API 速查表，当然 Hasor 更建议你使用离线版本的 HasorAPI。

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。在 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 4.0

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 的 Aop 是由 util 包的 classcode 工具提供的，该工具依赖了 asm4.0 框架。软件地址：(*http://www.objectweb.org/asm*)。

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。

--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 获取源码

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*

Git@OSC ： *http://git.oschina.net/zycgit/hasor*

Hasor 的项目主页是(*http://www.hasor.net/*)

QQ 交流群：193943114

1.5 环境要求

外部依赖

Hasor 要求 JDK 运行版本为 1.6，除此之外没有任何第三方依赖。只有 Hasor 的模块才会产生第三方依赖。

日志输出

如果您打算使用 log4j 作为日志组建那么需要加入下面几个依赖 jar 文件，在附录中添加了 *log4j.xml*、*logger.properties* 配置文件的参考配置内容。

slf4j-api-1.7.5.jar	(slf4j 的 API)
jul-to-slf4j-1.7.2.jar	(jdk-logger 到 slf4j 的适配器)
slf4j-log4j12-1.7.2.jar	(slf4j-log4j 的适配器)
log4j-1.2.17.jar	(log4j 库文件)
logger.properties	(jdk-logger 的配置。不需要配置 JDK 启动参数 Hasor 会自动装载它)
log4j.xml	(参考的 log4j 配置文件)

```
#logger.properties 内容
handlers = org.slf4j.bridge.SLF4JBridgeHandler
```

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。大多数都只是约定俗成的方式，当然 Hasor 也允许您自己建立一套专有的约定标准。

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。您也可以使用 Idea 进行开发，下面以 Eclipse 为例，因为它是免费的。

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　项目依赖

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-0.0.10.jar	（必选）Hasor 核心软件包	~498 KB
hasor-web-0.0.8.jar	（可选）Web 基础框架	~66 KB
hasor-db-0.0.3.jar	（可选）JDBC 开发框架	~104 KB
hasor-mvc-0.0.2.jar	（可选）MVC 框架	~63 KB
hasor-test-0.0.3.jar	（可选）测试框架	~19 KB
hasor-rsf-0.0.1.jar	（可选）分布式服务框架	~455 KB
hasor-quick-0.0.1.jar	（可选）快速开发工具包，建议选择	~46 KB
hasor-search-client-0.0.1.jar	（可选）Hasor-Search 的客户端程序	~61 KB

提示 1：如果引用 RSF，需要引入 Netty 的 *netty-all-4.0.23.Final.jar* 依赖包。
提示 2：如果引用 Test，需要引入 Junit 的 *junit-4.8.2.jar* 依赖包。

2.2.2　Maven 创建项目

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：

```
Administrator@YF-ZHAOYONGCHUN ~/Desktop/hasor-git/design/doc (master)
$ mvn archetype:create -DgroupId=org.myhasor.app -DartifactId=myproject -Darche
typeArtifactId=maven-archetype-webapp
[INFO] Scanning for projects...
[INFO] Parameter: groupId, Value: org.myhasor.app
[INFO] Parameter: packageName, Value: org.myhasor.app
[INFO] Parameter: package, Value: org.myhasor.app
[INFO] Parameter: artifactId, Value: myproject
[INFO] Parameter: basedir, Value: C:\Users\Administrator.PC-20130228ECWS\Deskto
\hasor-git\design\doc
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] project created from Old (1.x) Archetype in dir: C:\Users\Administrator.
C-20130228ECWS\Desktop\hasor-git\design\doc\myproject
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
```

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>0.0.10</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。它会自动依赖相应版本的 Hasor-Core 项目。

```
<dependency>
    <groupId>net.hasor</groupId>
    <artifactId>hasor-web</artifactId>
    <version>0.0.8</version>
</dependency>
```

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，那么还需要追加“*-Dwtpversion=2.0*”参数，例如下面这个命令：

```
mvn eclipse:eclipse -Dwtpversion=2.0
```

如果您想看到 Hasor 的源代码和 JavaDocs，那么可以使用下面这个命令：

```
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 配置 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，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“demo-Test”和“demo-Web”两个示例项目中找到，了解示列程序将会有助于您更快的上手学习 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 net.hasor.core.AppContext;
import net.hasor.core.Hasor;
public class StartTest {
    public void startTest() throws Exception {
        //创建 Hasor 容器。
        AppContext appContext = Hasor.createAppContext();
    }
}
```

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 appContext = Hasor.createAppContext("my-config.xml");
```

下面是“*my-config.xml*”配置文件中需要的最少内容：

```
<?xml version="1.0" encoding="UTF-8"?>
<config xmlns="http://project.hasor.net/hasor/schema/main">

</config>
```

提示：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。

但是建议您，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 约定的默认项目配置命名空间。

2.4 关于 Bean

2.4.1 一般用法

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。

下面这段代码片段展示了如何注册一个 Bean。

```
AppContext appContext = Hasor.createAppContext(new Module() {
    public void loadModule(ApiBinder apiBinder) throws Throwable {
        apiBinder.bindType(PojoBean.class); /*绑定类型到Hasor*/
    }
});
```

需要提醒大家的是原生 Hasor-Core 获取 Bean 必须通过 Bean 绑定的类型才能获取。例如像如下代码：

```
PojoBean myBean = appContext.getInstance(PojoBean.class);
```

对于同一个类型的不同绑定还可以给予一个名字例如：

```
AppContext appContext = Hasor.createAppContext(new Module() {
    public void loadModule(ApiBinder apiBinder) throws Throwable {
        apiBinder.bindType(String.class)
            .nameWith("ModuleA").toInstance("this String form A");
        apiBinder.bindType(String.class)
            .nameWith("ModuleB").toInstance("this String form B");
    }
});
```

对于这样的绑定使用如下代码获取到我们需要的那个 Bean 对象。

```
appContext.findBindingBean("ModuleA", String.class);
```

2.4.2 Quick 提供的 Bean 插件

上面的代码或许你会感觉到声明一个 Bean 要写那么多代码，很麻烦。于是 Hasor 通过 Quick 插件提供了一组专门针对 Bean 的简化操作 API 以帮你解决问题。

下面代码展示了如何声明一个 Bean：

```
@Bean()/*@Bean注解是用来起名字的*/
public class CustomBean {
    public void foo() {
        System.out.println("invoke CustomBean.foo");
    }
}
```

下面这段代码展示了如何获取这个 Bean：

```
AppContext appContext = Hasor.createAppContext();
Beans beans = appContext.getInstance(Beans.class);
MyBean myBean = beans.getBean("myBean");
```

注意的是 Quick 是 Hasor 的一个子项目，上面的简化 API 并不是和核心包提供的。如果要使用 Quick，我们要做的只是在工程中引入如下依赖就可以了：

```
<dependency>
    <groupId>net.hasor</groupId>
    <artifactId>hasor-quick</artifactId>
    <version>0.0.1</version>
</dependency>
```

2.5 依赖注入

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 编码方式

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 Hasor 的方式

目前 Hasor 的依赖注入还比较弱，需要依赖注入的 Bean 需要实现“*InjectMembers*”接口。依赖注入过程在该接口中完成，该接口会提供一个“*doInject*”方法并将“*AppContext*”传递进来供类初始化成员变量使用。例如下面代码：

```
public class UserBean implements InjectMembers {
    private String       userID   = "1234";
    private String       userName = "测试用户";
    private TypeBean userType = null;
    public void doInject(AppContext appContext) {
        this.userType = appContext.getInstance(TypeBean.class);
    }
  ……
}
```

提示：在 Hasor 的未来版本中将会提供包括自动注入在内的高级注入功能。

2.5.4 单例

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 可以通过单例保证唯一性。下面是 Hasor 原生方式，通过代码“*asEagerSingleton()*”来声明 Bean 的唯一性。

```
AppContext appContext = Hasor.createAppContext(new Module() {
    public void loadModule(ApiBinder apiBinder) throws Throwable {
        apiBinder.bindType(PojoBean.class).asEagerSingleton();
    }
});
```

如果是使用的 Quite 中的 Bean，声明单例更简单如下：

```
@Bean(singleton = true)
public class MyBeanSingleton { …… }
```

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

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

在静态代理中所有类型都是衡定的。在程序执行时，代理类（WorkerProxy）的 class 文件已经预先存在。在动态代理中这却恰恰相反的，代理类不会预先存在，当需要它的时候通过一些专门的类库创建这个代理程序。

比方说一个程序中有多种不同的 Servies 类。我们要打印出调用每个业务方法所占用的时间。如果使用静态代理方式会发现，程序中根本不存在衡定的“doWorker”方法。

虽然不存在衡定的“doWorker”方法，但是调用行为是存在的。而且可以将其行为抽象出来这就是 Aop 中的“**切面**”，负责执行这个切面的类就叫“**拦截器**”。下面这个代码展示了如何用 Java 的原生支持实现动态代理。

```
ClassLoader lod = Thread.currentThread().getContextClassLoader();
Class<?>[] faceSet = new Class[] { TestBean2_Face.class };
//
Object proxy = Proxy.newProxyInstance(//
        lod, faceSet, new JavaInvocationHandler());
//
TestBean2_Face face = (TestBean2_Face) proxy;
System.out.println(face.toString());
```

上面例子中拦截器就是：“JavaInvocationHandler”类。

```
class JavaInvocationHandler implements InvocationHandler {
    public Object invoke(Object proxy, Method method, Object[] args)
                                                    throws Throwable {
        return null; // TODO Auto-generated method stub
    }
}
```

由此可见在 Java 中实现一个动态代理还算很简单的,但是有的时候我们想把所有 Bean 都管理起来。并且按照自己的意愿来对其进行动态代理，在这种要求下我们不得不自己去开发一套 Bean 管理程序，或者使用更为成熟的框架例如：Spring、Guice、或者您也可以使用 Hasor 进行 Bean 的管理。

2.6.4 Hasor 的方式

Hasor 的 Aop 声明使用方式和 JDK 自带的很相似，但是由于 Hasor 具有 Bean 管理的功能，因此 Hasor 很容易在一批 Bean 上使用动态代理功能。这将会大大减少重复代码的开发，而且 Hasor 还可以通过匹配器让您自己框选符合条件的 Bean，不同于 Spring 的是 Hasor 提供的 Api 更加注重编程性而非配置声明。

```
public class SimpleInterceptor implements MethodInterceptor {
    public Object invoke(MethodInvocation invocation) throws
Throwable {
        try {
            System.out.println("before...");
            Object returnData = invocation.proceed();
            System.out.println("after...");
            return returnData;
        } catch (Exception e) {
```

2.7 Aop

后面需要重新编写

```
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());
```

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：类级拦截器钰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))>*

```
@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(
```

获取被代理的 JobBean 对象并调用 foo 方法查看运行结果：

```
08/20 21:35:43 [main] INFO: DefaultAppContext:start ->> hasor started!
----
aop3:public java.lang.String org.aop.JobBean.println(java.lang.String)
aop2:public java.lang.String org.aop.JobBean.println(java.lang.String)
aop1:public java.lang.String org.aop.JobBean.println(java.lang.String)
println->aa
```

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））>*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))>*

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)>*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：

```
08/21 11:57:15 [main] INFO: DefaultAppContext:start ->> hasor started!
step:1
hello
step:2
```

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)>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：

```
08/21 14:22:18 [main] INFO: DefaultAppContext:start ->> hasor started!
step:1
step:2
hello
```

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>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
```

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
<?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：

```java
/*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

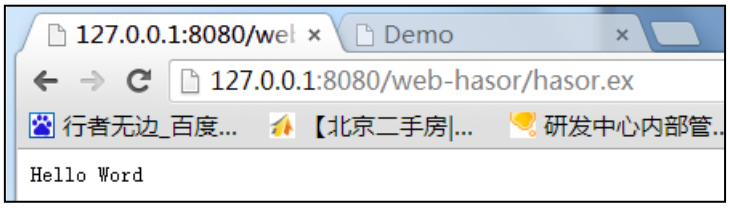
2.12.1　HttpServlet

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)*>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”*

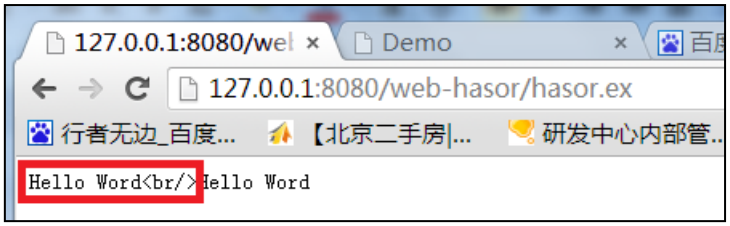


2.12.2 Filter

通过@WebFilter 注解可以注册 Filter。在*<第六章：Web 支持(6.4 节-Filter)>*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)>*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：

```
信息: Server startup in 1698 ms
create session:82956BAC80D064CA8A128ED56D99B859
```

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)>*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：

```
08/23 15:00:10 [main] INFO: RuntimeListener:contextInitialized ->> Servlet
contextInitialized.
08/23 15:00:10 [main] INFO: RuntimeFilter:init ->> PlatformFilter started.
```

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)>*。

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>*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)>*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)>。

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)>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,
```

【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)>*。

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(
```

【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>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)>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)>*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

第五章 内核开发

第六章 配置文件详解

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