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

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。Hasor 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。
- 约定优于配置(COC)：可以完全不需要配置 Hasor 就进行系统开发，配置文件功能的可以完全留给业务系统使用。

设计思想：

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十章都是在详细讲解 Hasor 各个部件的功能和实现机制以及原理。

第十一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*)邮箱联系到我也可以通过(*https://github.com/zycgit/hasor/pulls*)递交您的代码。

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

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

1.5 类库引用

依赖：Hasor依赖以下4个软件包：

slf4j-api-1.7.5.jar	*(负责输出Hasor产生的日志)*
guice-3.0.jar	*(Guice3.0，负责提供DI相关的支持)*
aopalliance-1.0.jar	*(Aop联盟API，Guice依赖)*
javax.inject-1.jar	*(JSR-330标准，Guice依赖)*

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

log4j-1.2.17.jar	*(log4j库文件)*
slf4j-log4j12-1.7.2.jar	*(slf4j-log4j的适配器)*
log4j.xml	*(参考的log4j配置文件)*

1.6 约定优于配置(COC原则)

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

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

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

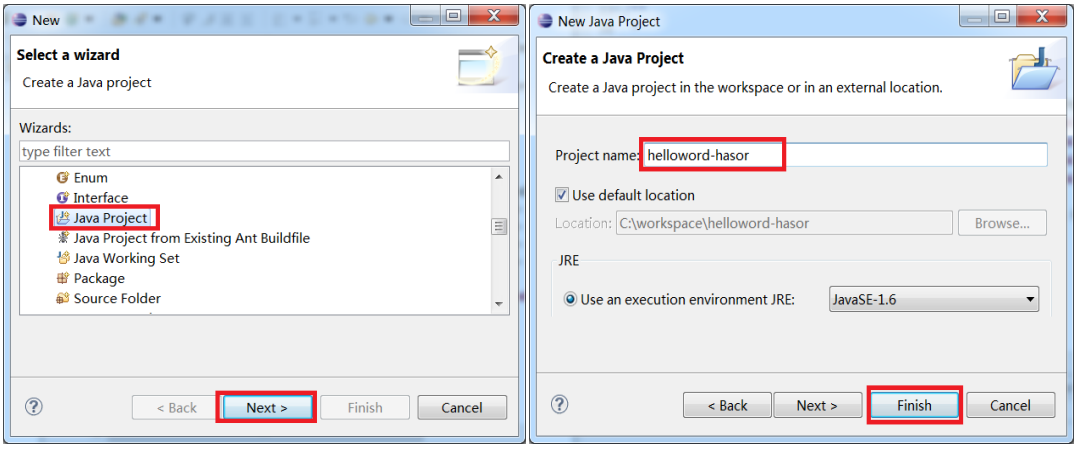
第二章　使用 Hasor

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。

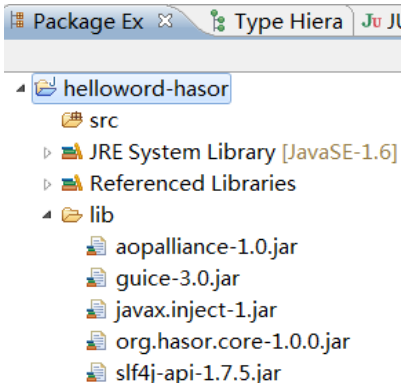


图 1：

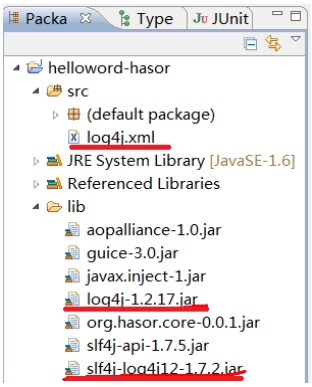


图 2：

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

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，必须调用
    }
}
```

在控制台中可以得到 *2 个警告*和 *1 个错误*，它们看上去应该是这样的：

```
08/17 20:34:53 [main] INFO: HasorSettings:loadNsProp ->> find 'ns.prop' at
'jar:file:/C:/workspa……hasor.core-0.0.1.jar!/META-INF
08/17 20:34:53 [main] INFO: HasorSettings:loadStaticConfig ->> load
'jar:file:/C:/workspace/helloword-hasor/lib/org.hasor.core-0.0.1.jar!/static-c
onfig.xml'
08/17 20:34:53 [main] WARN: HasorSettings:loadMainConfig ->> cannot load the root
configuration file 'hasor-config.xml'
08/17 20:34:53 [main] INFO: StandardInitContext:initContext ->> loadPackages :
[ org.hasor.context.* , org.hasor.servlet.* ]
08/17 20:34:53 [main] WARN: HasorSettings:start ->> Can't get to mainConfig
'hasor-config.xml'.
08/17 20:34:54 [main] INFO: AnnoAppContext:loadModule ->> find HasorModule :
[ org.hasor.servlet.anno.support.ServletAnnoSupportModule ,
org.hasor.context.anno.support.AnnoSupportModule ]
08/17 20:34:54 [main] INFO: DefaultAppContext:onInit ->> init Event on :
org.hasor.context.anno.support.AnnoSupportModule
08/17 20:34:54 [main] ERROR: DefaultAppContext.java:229 - onInit ->>
ServletAnnoSupportModule is not init! Hasor context does not support the web module.
08/17 20:34:54 [main] INFO: MasterModule:configure ->> init modules finish.
```

【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。*<第五章：配置文件(5.3 节-主配置文件)>*

第二条警告：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，如果没有定义主配置文件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)>*

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。*<第六章：Web 支持(6.1 节-简介)>*

2.3 第一个模块(HelloWord)

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

```
@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

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

```
public class CustomBean {
    public void foo() {
        System.out.println("invoke CustomBean.foo");
    }
}
```

2.4.1 注册 Bean

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

方式一： 使用@Bean 注解将 Bean 注册到 Hasor 容器中。

```
@Bean("myBean")/*@Bean注解还可以定义多个名字*/
public class CustomBean {
……
```

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

```
@Module
public class FirstModule extends AbstractHasorModule {
    public void init(ApiBinder binder) {
        apiBinder.newBean("myBean").bindType(CustomBean.class);
    }
}
```

2.4.2 获取 Bean

```
CustomBean b1 = context.getBean("myBean");//根据名字获取
CustomBean b2 = context.getInstance(CustomBean.class);//根据类型
```

2.4.3 单例 Bean

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

```
@Bean("myBean")
@Singleton/*声明单例*/
public class CustomBean {
……
```

2.4.4 容器之外的 Bean

通过和注册 Bean 的一些比较可以有一个比较深刻的印象：

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

Bean 相关的介绍可以查看<第四章：环境支持(4.5 节-Bean 服务(@Bean))>以了解更多信息。

2.5 依赖注入(IoC)

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

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

2.5.1 字段方式注入

```
public class TestBean {
  @Inject//将CustomBean注入到TestBean被注入的Bean可以不具备get/set
  private CustomBean customBean = null;
  public void callFoo() { this.customBean.foo(); }
}
```

2.5.2 属性方式注入

```
public class TestBean {
    private CustomBean customBean = null;
    @Inject
    public void setCustomBean(CustomBean customBean){
        this.customBean = customBean;
    }
}
```

2.5.3 构造方法注入

```
public class TestBean2 {
    private CustomBean customBean = null;
    @Inject/*标记到构造方法上*/
    public TestBean2(CustomBean customBean) {
        this.customBean = customBean;
    }
    public void callFoo() { this.customBean.foo(); }
}
```

2.6 Aop 拦截器(MethodInterceptor)

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

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

```
public class AopInterceptor_1 implements MethodInterceptor {
    public Object invoke(MethodInvocation arg0) throws Throwable {
        System.out.println("aop1:" + arg0.getMethod());
        return arg0.proceed();//拦截器，用在方法级别
    }
}
public class AopInterceptor_2 implements MethodInterceptor {
    public Object invoke(MethodInvocation arg0) throws Throwable {
        System.out.println("aop2:" + arg0.getMethod());
        return arg0.proceed();//拦截器，用在类级别
    }
}
public class AopInterceptor_3 implements MethodInterceptor {
    public Object invoke(MethodInvocation arg0) throws Throwable {
        System.out.println("aop3:" + arg0.getMethod());
        return arg0.proceed();//拦截器，用在全局
    }
}
```

2.6.1 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; }
}
```

2.6.2 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。

2.6.3 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(
                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：

```
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)

2.6.4 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(),……);
```

2.7 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))>*

2.7.1 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");
```

2.7.2 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
```

2.7.3 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。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
```

2.8 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。

2.8.1 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)*>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】

```
08/22 13:16:47 [main] INFO: ModuleReactor:process ->> build dependency tree for ModuleInfo.
08/22 13:16:47 [main] INFO: ModuleReactor:process ->> dependence Tree
0.-->  Mode2 (class org.test.dep.force.Mode2)
1.-->    Mode1 (class org.test.dep.force.Mode1)
08/22 13:16:47 [main] INFO: ModuleReactor:process ->> startup sequence.
0.-->Mode1 (class org.test.dep.force.Mode1)
1.-->Mode2 (class org.test.dep.force.Mode2)
08/22 13:16:47 [main] INFO: MasterModule:configure ->> send init sign...
Mode1  init!
08/22 13:16:47 [main] INFO: DefaultAppContext:onInit ->> init Event on : org.test.dep.force.Mode1
Mode2  init!
08/22 13:16:47 [main] INFO: DefaultAppContext:onInit ->> init Event on : org.test.dep.force.Mode2
08/22 13:16:47 [main] INFO: MasterModule:configure ->> init modules finish.
08/22 13:16:47 [main] INFO: DefaultAppContext:start ->> send start sign.
08/22 13:16:47 [main] INFO: DefaultAppContext:onStart ->> start Event on : org.test.dep.force.Mode1
08/22 13:16:47 [main] INFO: DefaultAppContext:onStart ->> start Event on : org.test.dep.force.Mode2
08/22 13:16:47 [main] INFO: DefaultAppContext:printModState ->> Modules State List:
0.-->[Running] Mode1 (class org.test.dep.force.Mode1)
1.-->[Running] Mode2 (class org.test.dep.force.Mode2)
08/22 13:16:47 [main] INFO: DefaultAppContext:start ->> hasor started!
```

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。

```
08/22 13:36:15 [main] INFO: ModuleReactor:process ->> dependence Tree
0.-->  Mode2 (class org.test.dep.force.Mode2)
1.-->    Mode1 (class org.test.dep.force.Mode1)
08/22 13:36:15 [main] INFO: ModuleReactor:process ->> startup sequence.
0.-->Mode1 (class org.test.dep.force.Mode1)
1.-->Mode2 (class org.test.dep.force.Mode2)
08/22 13:36:15 [main] ERROR: DefaultAppContext.java:227 - onInit ->> Mode1 is not init! this is my Error
08/22 13:36:15 [main] INFO: MasterModule:configure ->> send init sign...
08/22 13:36:15 [main] INFO: MasterModule:configure ->> init modules finish.
08/22 13:36:15 [main] INFO: DefaultAppContext:start ->> send start sign.
08/22 13:36:15 [main] INFO: DefaultAppContext:printModState ->> Modules State List:
0.-->[Stopped] Mode1 (class org.test.dep.force.Mode1)
1.-->[Stopped] Mode2 (class org.test.dep.force.Mode2)
08/22 13:36:15 [main] INFO: DefaultAppContext:start ->> hasor started!
```

第二个方框中可以看到两个模块都没有启动。

2.8.2 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)》*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】

```
08/22 13:52:13 [main] INFO: ModuleReactor:process ->> dependence Tree
0.-->  Mode2 (class org.test.dep.before.Mode2)
1.-->    Mode1 (class org.test.dep.before.Mode1)
08/22 13:52:13 [main] INFO: ModuleReactor:process ->> startup sequence.
0.-->Mode1 (class org.test.dep.before.Mode1)
1.-->Mode2 (class org.test.dep.before.Mode2)
08/22 13:52:13 [main] INFO: MasterModule:configure ->> send init sign...
08/22 13:52:13 [main] ERROR: DefaultAppContext.java:227 - onInit ->> Mode1 is not init! this is my Error
Mode2  init!
08/22 13:52:13 [main] INFO: DefaultAppContext:onInit ->> init Event on : org.test.dep.before.Mode2
08/22 13:52:13 [main] INFO: MasterModule:configure ->> init modules finish.
08/22 13:52:14 [main] INFO: DefaultAppContext:start ->> send start sign.
08/22 13:52:14 [main] INFO: DefaultAppContext:onStart ->> start Event on : org.test.dep.before.Mode2
08/22 13:52:14 [main] INFO: DefaultAppContext:printModState ->> Modules State List:
0.-->[Stopped] Mode1 (class org.test.dep.before.Mode1)
1.-->[Running] Mode2 (class org.test.dep.before.Mode2)
```

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。

2.8.3 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)*>会有更多详细的介绍。

```java
public class MyMode extends AbstractHasorModule {
    public void configuration(ModuleSettings info) {
        info.afterMe(JFinalMode.class);//反转依赖
    }
    ……
}
```

2.9 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
<?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
<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：

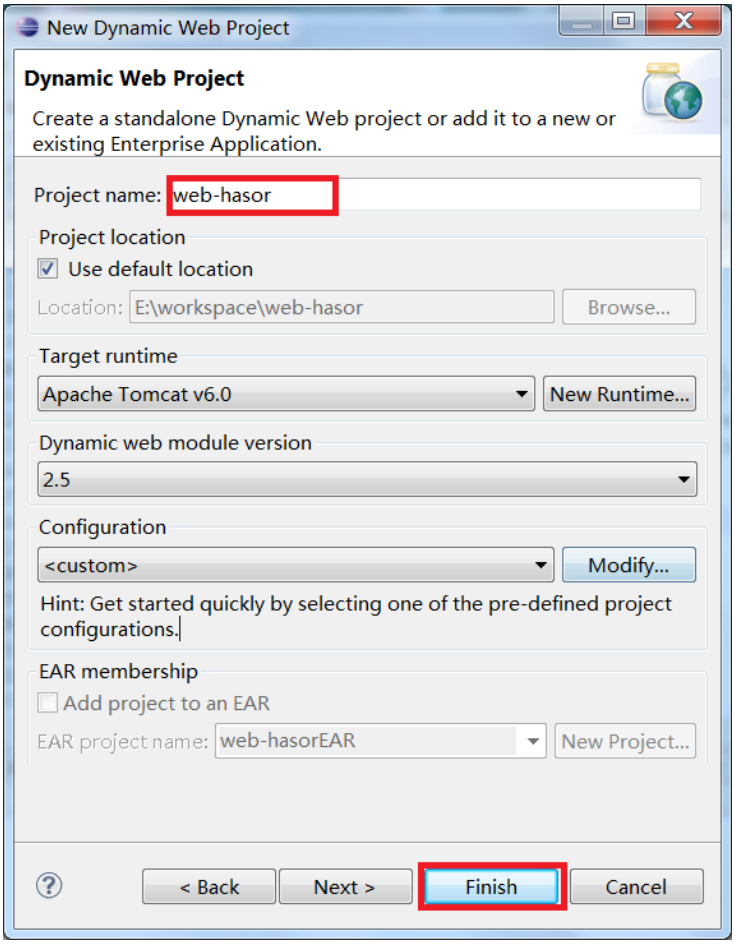
请不要使用下面这些名字作为第一级标签名这些都是 Hasor 保留的：

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”。

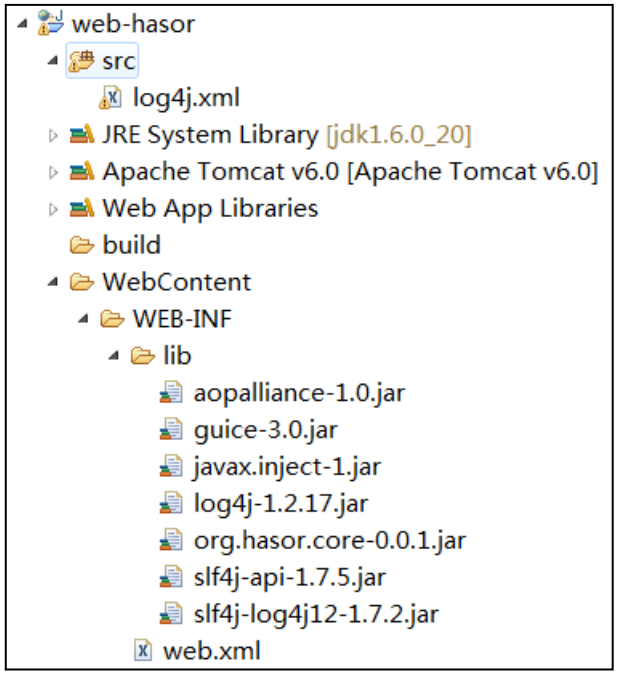
2.10 Web 项目

Step 1：创建 Dynamic Web Project 项目

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



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



Step 3：修改 web.xml 配置文件加入如下内容即完成配置。

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

```
<listener>
    <listener-class>
        org.hasor.servlet.startup.RuntimeListener
    </listener-class>
</listener>
<filter>
    <filter-name>runtime</filter-name>
    <filter-class>
        org.hasor.servlet.startup.RuntimeFilter
    </filter-class>
</filter>
<filter-mapping>
    <filter-name>runtime</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

2.11 Web 开发

2.11.1 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();
    }
}
```

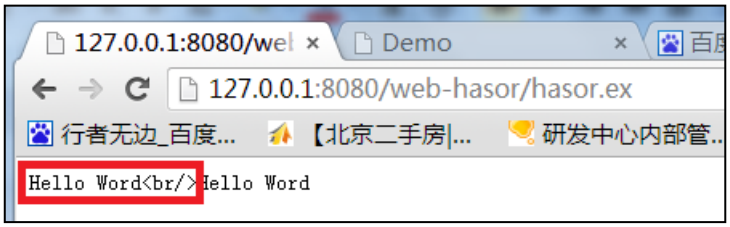
启动程序输入 Servlet 地址 *“http://127.0.0.1:8080/<your project name>/hasor.ex”*

2.11.2 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”*



2.11.3 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
```

2.11.4 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.
```

2.11.5 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：

```java
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");
    }
}
```

2.12 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。

2.12.1 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。

```java
@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”。

2.12.2 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)>。

2.12.3 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, 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)>*。

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。

2.12.4 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 ……; }
}
```

2.12.5 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);
    }
}
```

2.13 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。

2.13.1 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*”。

2.13.2 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。

2.13.3 指定的目录中加载资源 Web 资源。

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

```
<config xmlns="http://project.hasor.net/hasor/schema/main">
   <hasor-mvc>
      <resourceLoader enable="true">
         <!-- 使用绝对路径配置的资源文件包 -->
         <pathLoader>C:\bizData\icons\</pathLoader>
         <!-- 使用WEB-INF目录下的pic.zip作为资源文件包-->
         <pathLoader>%HASOR_WEBROOT%/WEB-INF/picDir</pathLoader>
      </resourceLoader>
   </hasor-mvc>
</config>
```

（未完待续）

……