

JFinal 手册

版本： 1.8

作者： 詹波

日期： 2014-05-23

<http://www.jfinal.com>

目录

第一章 快速上手	6
1.1 创建项目	6
1.2 放入 JFinal 库文件	9
1.3 修改 web.xml	9
1.4 添加 java 文件	9
1.5 启动项目	10
1.6 开启浏览器看效果	11
第二章 JFinalConfig.....	12
2.1 概述	12
2.2 configConstant(Constants me).....	12
2.3 configRoute(Routes me).....	12
2.4 configPlugin (Plugins me)	15
2.5 configInterceptor (Interceptors me)	15
2.6 configHandler (Handlers me)	15
2.7 afterJFinalStart()与 beforeJFinalStop().....	16
第三章 Controller.....	17
3.1 概述	17
3.2 Action.....	17
3.3 getPara 系列方法.....	17
3.4 getFile 文件上传	18
3.5 setAttr 方法.....	18
3.6 render 系列方法	19
第四章 Interceptor.....	21
4.1 概述	21
4.2 Interceptor 应用.....	21
4.3 Interceptor 配置.....	21
4.4 ActionInvocation.....	22
4.5 InterceptorStack	23

第五章 ActiveRecord.....	24
5.1 概述.....	24
5.2 ActiveRecordPlugin.....	24
5.3 Model.....	24
5.4 JFinal 独创 Db + Record 模式	26
5.5 声明式事务	27
5.6 Cache	28
5.7 Dialect 多数据库支持	28
5.8 表关联操作	28
5.9 Oracle 支持.....	29
5.10 多数据源支持	31
第六章 EhCachePlugin	33
6.1 概述.....	33
6.2 EhCachePlugin	33
6.3 CacheInterceptor	33
6.4 EvictInterceptor	34
6.5 CacheKit.....	34
6.6 ehcache.xml 简介	35
第七章 SpringPlugin.....	36
7.1 概述.....	36
7.2 SpringPlugin.....	36
7.3 locInterceptor.....	36
第八章 Validator	38
8.1 概述.....	38
8.2 Validator	38
8.3 Validator 配置	38
第九章 FreeMarker 基础.....	39
9.1 概述.....	39
9.2 FreeMarker 示例.....	39
9.3 在 JFinal 中扩展	39

第十章 JFinal 架构及扩展	40
10.1 概述	40
10.2 架构	40

摘要

JFinal 是基于 Java 语言的极速 WEB + ORM 开发框架，其核心设计目标是开发迅速、代码量少、学习简单、功能强大、轻量级、易扩展、Restful。在拥有 Java 语言所有优势的同时再拥有 ruby、python、php 等动态语言的开发效率！为您节约更多时间，去陪恋人、家人和朋友 :)

JFinal 有如下主要特点：

- MVC 架构，设计精巧，使用简单
- 遵循 COC 原则，零配置，无 xml
- 独创 Db + Record 模式，灵活便利
- ActiveRecord 支持，使数据库开发极致快速
- 自动加载修改后的 java 文件，开发过程中无需重启 web server
- AOP 支持，拦截器配置灵活，功能强大
- Plugin 体系结构，扩展性强
- 多视图支持，支持 FreeMarker、JSP、Velocity
- 强大的 Validator 后端校验功能
- 功能齐全，拥有 struts2 绝大部分核心功能
- 体积小仅 218K，且无第三方依赖

JFinal 官方网站：<http://www.jfinal.com>

JFinal QQ 群: 222478625、196337924、326297041

JFinal 官方微信:

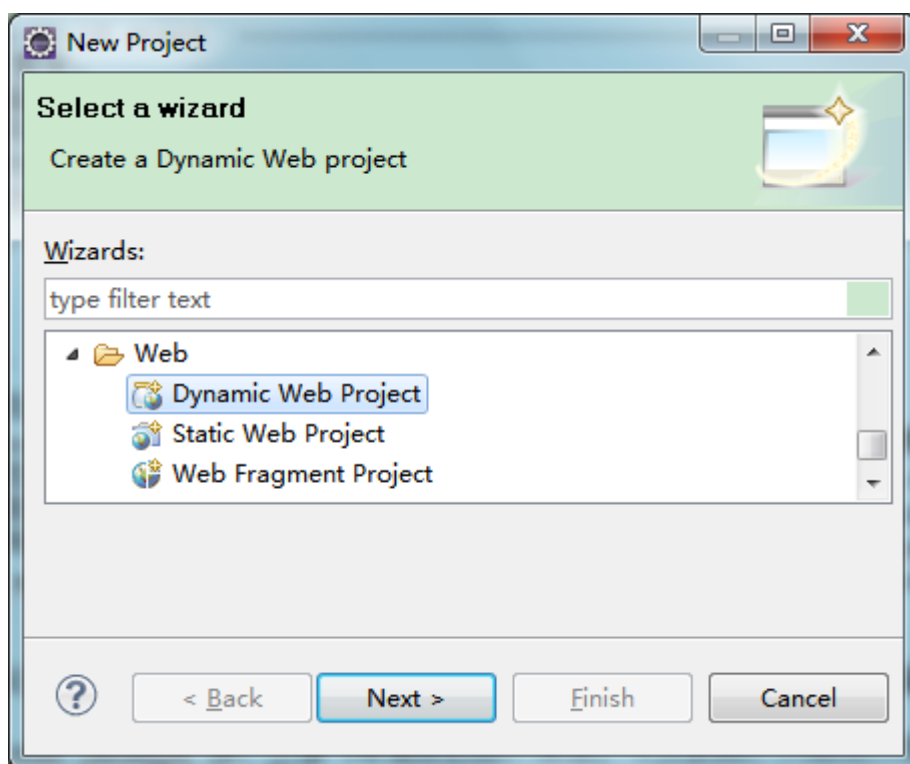


第一章 快速上手

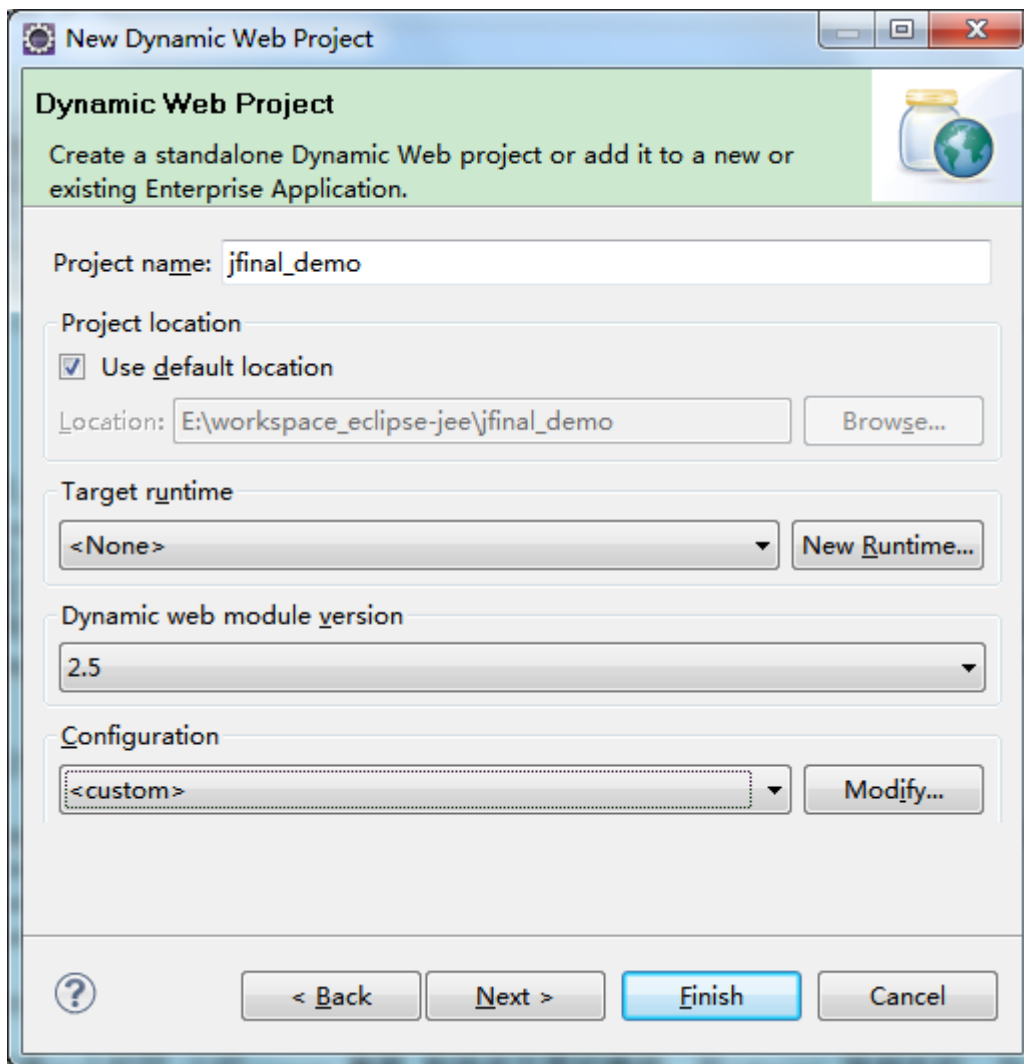
1.1 创建项目

JFinal 推荐使用 Eclipse IDE for Java EE Developers 做为开发环境。下载链接：
<http://www.eclipse.org/downloads/download.php?file=/technology/epp/downloads/release/indigo/SR2/eclipse-jee-indigo-SR2-win32.zip>

- 创建 Dynamic Web Project

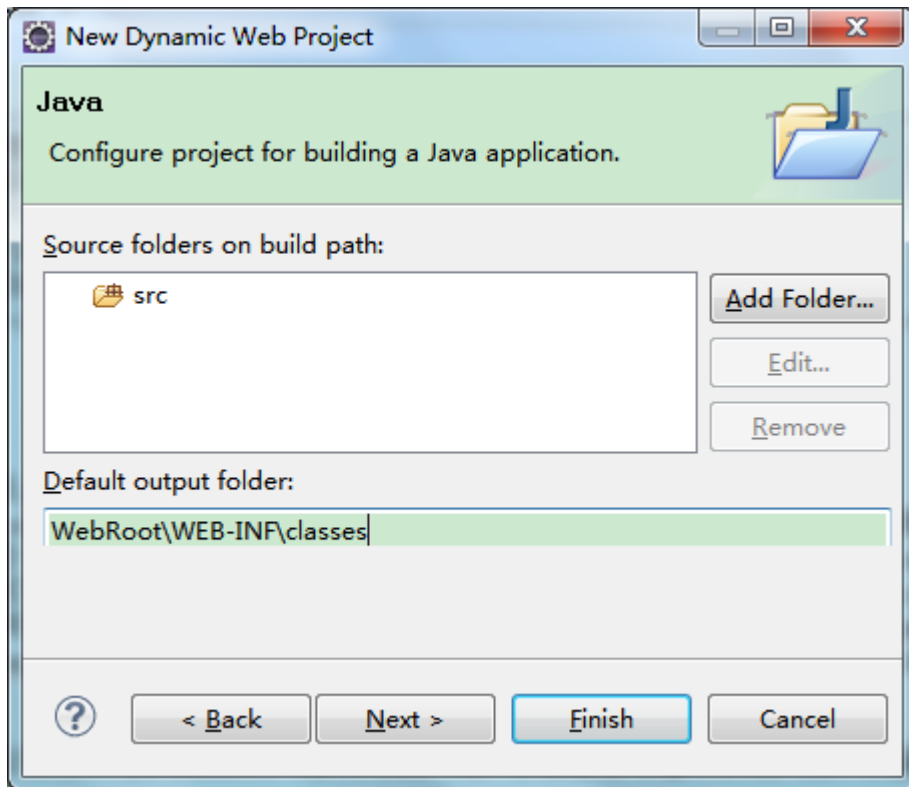


- 填入项目基本信息



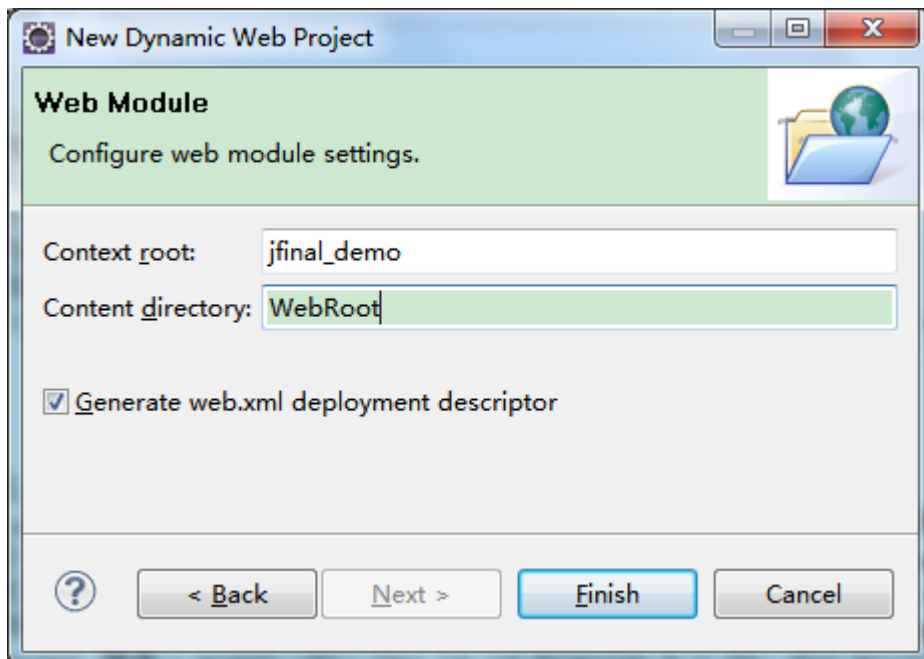
注意：Target runtime 一定要选择<None>

- 修改 Default Output Folder，推荐输入 WebRoot\WEB-INF\classes



特别注意：此处的 Default out folder 必须要与 WebRoot\WEB-INF\classes 目录完全一致才可以使用 JFinal 集成的 Jetty 来启动项目。

- 修改 Content directory，推荐输入 WebRoot



注意：此处也可以使用默认值 WebContent，但上一步中的 WebRoot\WEB-INF\classes 则需要改成 WebContent\WEB-INF\classes 才能对应上。

1.2 放入 JFinal 库文件

将 jfinal-xxx.jar 与 jetty-server-8.1.8.jar 拷贝至项目 WEB-INF\lib 下即可。注意：jetty-server-8.1.8.jar 是开发时使用的运行环境，生产环境不需要此文件。

1.3 修改 web.xml

将如下内容添加至 web.xml

```
<filter>
  <filter-name>jfinal</filter-name>
  <filter-class>com.jfinal.core.JFinalFilter</filter-class>
  <init-param>
    <param-name>configClass</param-name>
    <param-value>demo.DemoConfig</param-value>
  </init-param>
</filter>
<filter-mapping>
  <filter-name>jfinal</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

1.4 添加 java 文件

在项目 src 目录下创建 demo 包，并在 demo 包下创建 DemoConfig 文件， 内容如下：

```
package demo;
import com.jfinal.config.*;
public class DemoConfig extends JFinalConfig {
    public void configConstant(Constants me) {
        me.setDevMode(true);
    }
    public void configRoute(Routes me) {
        me.add("/hello", HelloController.class);
    }
    public void configPlugin(Plugins me) {}
    public void configInterceptor(Interceptors me) {}
    public void configHandler(Handlers me) {}
}
```

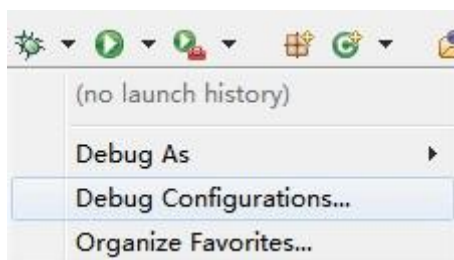
注意: DemoConfig.java 文件所在的包以及自身文件名必须与 web.xml 中的 param-value 标签内的配置相一致(在本例中该配置为 demo.DemoConfig)。

在 demo 包下创建 HelloController 类文件, 内容如下:

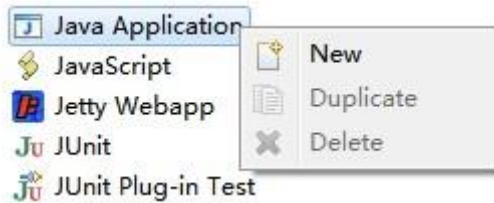
```
package demo;
import com.jfinal.core.Controller;
public class HelloController extends Controller {
    public void index() {
        renderText("Hello JFinal World.");
    }
}
```

1.5 启动项目

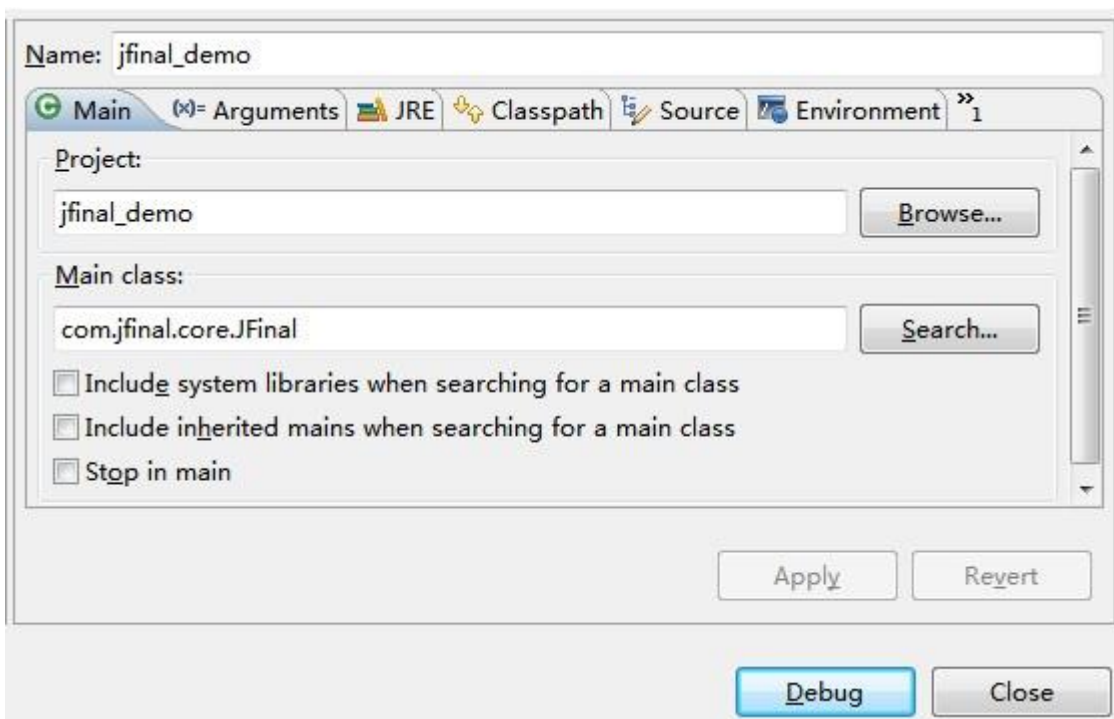
创建启动项如下图所示:



鼠标右键点击 Java Application 并选择 New 菜单项，新建 Java Application 启动项，如下图所示：



在右侧窗口中的 Main class 输入框中填入：com.jfinal.core.JFinal 并点击 Debug 按钮启动项目，如下图所示：



1.6 开启浏览器看效果

打开浏览器在地址栏中输入：<http://localhost/hello>，输出内容为 Hello JFinal World 证明项目框架搭建完成。如需完整 demo 示例请在此处下载：

<http://code.google.com/p/jfinal/downloads/list>

注意：在 tomcat 下开发或运行项目时，需要先删除 jetty-server-xxx.jar 这个包，否则会引起冲突。Tomcat 启动项目需要使用 TOMCAT-HOME/bin/startup.bat 或 startup.sh 而不能使用上面介绍的启动方式，因为上面的启动方式需要用到 jetty-server-xxx.jar。

第二章 JFinalConfig

2.1 概述

基于 JFinal 的 web 项目需要创建一个继承自 JFinalConfig 类的子类，该类用于对整个 web 项目进行配置。

JFinalConfig 子类需要实现五个抽象方法，如下所示：

```
public class DemoConfig extends JFinalConfig {  
    public void configConstant(Constants me) {}  
    public void configRoute(Routes me) {}  
    public void configPlugin(Plugins me) {}  
    public void configInterceptor(Interceptors me) {}  
    public void configHandler(Handlers me) {}  
}
```

2.2 configConstant(Constants me)

此方法用来配置 JFinal 常量值，如开发模式常量 devMode 的配置，默认视图类型 ViewType 的配置，如下代码配置了 JFinal 运行在开发模式下且默认视图类型为 JSP：

```
public void configConstant(Constants me) {  
    me.setDevMode(true);  
    me.setViewType(ViewType.JSP);  
}
```

在开发模式下，JFinal 会对每次请求输出报告，如输出本次请求的 Controller、Method 以及请求所携带的参数。JFinal 支持 JSP、FreeMarker、Velocity 三种常用视图。

2.3 configRoute(Routes me)

此方法用来配置 JFinal 访问路由，如下代码配置了将”/hello”映射到 HelloController 这个控制器，通过以下的配置，http://localhost/hello 将访问 HelloController.index() 方法，而 http://localhost/hello/methodName 将访问到 HelloController.methodName()方法。

```
public void configRoute(Routes me) {  
    me.add("/hello", HelloController.class);  
}
```

Routes 类主要有如下两个方法：

```
public Routes add(String controllerKey, Class<? extends Controller>
    controllerClass, String viewPath)
public Routes add(String controllerKey, Class<? extends Controller>
    controllerClass)
```

第一个参数 `controllerKey` 是指访问某个 `Controller` 所需要的一个字符串，该字符串唯一对应一个 `Controller`，`controllerKey` 仅能定位到 `Controller`。第二个参数 `controllerClass` 是该 `controllerKey` 所对应到的 `Controller`。第三个参数 `viewPath` 是指该 `Controller` 返回的视图的相对路径(该参数具体细节将在 `Controller` 相关章节中给出)。当 `viewPath` 未指定时默认值为 `controllerKey`。

JFinal 路由规则如下表：

url 组成	访问目标
<code>controllerKey</code>	<code>YourController.index()</code>
<code>controllerKey/method</code>	<code>YourController.method()</code>
<code>controllerKey/method/v0-v1</code>	<code>YourController.method()</code> ，所带 url 参数值为：v0-v1
<code>controllerKey/v0-v1</code>	<code>YourController.index()</code> ，所带 url 参数值为：v0-v1

从表中可以看出，JFinal 访问一个确切的 Action(Action 定义见 3.2 节)需要使用 `controllerKey` 与 `method` 来精确定位，当 `method` 省略时默认值为 `index`。`urlPara` 是为了能在 url 中携带参数值，`urlPara` 可以在一次请求中同时携带多个值，JFinal 默认使用减号“-”来分隔多个值（可通过 `constants.setUrlParaSeparator(String)` 设置分隔符），在 `Controller` 中可以通过 `getPara(int index)` 分别取出这些值。`controllerKey`、`method`、`urlPara` 这三部分必须使用正斜杠“/”分隔。

注意，`controllerKey` 自身也可以包含正斜杠“/”，如“`/admin/article`”，这样实质上实现了 struts2 的 namespace 功能。

JFinal 在以上路由规则之外还提供了 `ActionKey` 注解，可以打破原有规则，以下是代码示例：

```
public class UserController extends Controller {
    @ActionKey("/login")
    public void login() {
        render("login.html");
    }
}
```

假定 UserController 的 controllerKey 值为“/user”，在使用了@ActionKey(“/login”)注解以后，actionKey 由原来的“/user/login”变为了“/login”。该注解还可以让 actionKey 中使用减号或数字等字符，如“/user/123-456”。

如果 JFinal 默认路由规则不能满足需求，开发者还可以根据需要使用 Handler 定制更加个性化的路由，大体思路就是在 Handler 中改变第一个参数 String target 的值。

JFinal 路由还可以进行拆分配置，这对大规模团队开发特别有用，以下是代码示例：

```
public class FrontRoutes extends Routes {
    public void config() {
        add("/", IndexController.class);
        add("/blog", BlogController.class);
    }
}
```

```
public class AdminRoutes extends Routes {
    public void config() {
        add("/admin", AdminController.class);
        add("/admin/user", UserController.class);
    }
}
```

```
public class MyJFinalConfig extends JFinalConfig {
    public void configRoute(Routes me) {
        me.add(new FrontRoutes()); // 前端路由
        me.add(new AdminRoutes()); // 后端路由
    }

    public void configConstant(Constants me) {}
    public void configPlugin(Plugins me) {}
    public void configInterceptor(Interceptors me) {}
    public void configHandler(Handlers me) {}
}
```

如上三段代码，FrontRoutes 类中配置了系统前端路由，AdminRoutes 配置了系统后端路由，MyJFinalConfig.configRoute(...)方法将拆分后的这两个路由合并起来。使用这种拆分配置不仅可以让 MyJFinalConfig 文件更简洁，而且有利于大规模团队开发，避免多人同时修改

MyJFinalConfig 时的版本冲突。

2.4 configPlugin (Plugins me)

此方法用来配置 JFinal 的 Plugin, 如下代码配置了 C3p0 数据库连接池插件与 ActiveRecord 数据库访问插件。通过以下的配置, 可以在应用中使用 ActiveRecord 非常方便地操作数据库。

```
public void configPlugin(Plugins me) {
    loadPropertyFile("your_app_config.txt");
    C3p0Plugin c3p0Plugin = new C3p0Plugin(getProperty("jdbcUrl"),
        getProperty("user"), getProperty("password"));
    me.add(c3p0Plugin);
    ActiveRecordPlugin arp = new ActiveRecordPlugin(c3p0Plugin);
    me.add(arp);
    arp.addMapping("user", User.class);
}
```

JFinal 插件架构是其主要扩展方式之一, 可以方便地创建插件并应用到项目中去。

2.5 configInterceptor (Interceptors me)

此方法用来配置 JFinal 的 Interceptor, 如下代码配置了名为 AuthInterceptor 的拦截器, 在此处配置的拦截器将会对所有的请求进行拦截, 除非使用 @ClearInterceptor 在 Controller 中清除。

```
public void configInterceptor(Interceptors me) {
    me.add(new AuthInterceptor());
}
```

JFinal 的 Interceptor 非常类似于 Struts2, 但使用起来更方便, Interceptor 配置粒度分为 Global、Controller、Action 三个层次, 其中以上代码配置粒度为全局。Controller 与 Action 级的 Interceptor 配置将在后续章节中详细介绍。

2.6 configHandler (Handlers me)

此方法用来配置 JFinal 的 Handler, 如下代码配置了名为 ResourceHandler 的处理器, Handler 可以接管所有 web 请求, 并对应用拥有完的控制权, 可以很方便地实现更高层的功能性扩展。

```
public void configHandler(Handlers me) {  
    me.add(new ResourceHandler());  
}
```

2.7 afterJFinalStart()与 beforeJFinalStop()

JFinalConfig 中的 afterJFinalStart()与 beforeJFinalStop()方法供开发者在 JFinalConfig 继承类中覆盖。JFinal 会在系统启动完成后回调 afterJFinalStart()方法，会在系统关闭前回调 beforeJFinalStop()方法。这两个方法可以很方便地在项目启动后与关闭前让开发者有机会进行额外操作，如在系统启动后创建调度线程或在系统关闭前写回缓存。

第三章 Controller

3.1 概述

Controller 是 JFinal 核心类之一，该类作为 MVC 模式中的控制器。基于 JFinal 的 Web 应用的控制器需要继承该类。Controller 是定义 Action 方法的地点，是组织 Action 的一种方式，一个 Controller 可以包含多个 Action。Controller 是线程安全的。

3.2 Action

Controller 以及在其中定义的 public 无参方法称为一个 Action。Action 是请求的最小单位。Action 方法必须在 Controller 中声明，该方法必须是 public 可见性且没有形参。

```
public class HelloController extends Controller {  
    public void index() {  
        renderText("此方法是一个action");  
    }  
    public void test() {  
        renderText("此方法是一个action");  
    }  
}
```

以上代码中定义了两个 Action: HelloController.index()、HelloController.test()。在 Controller 中提供了 getPara 系列方法 setAttr 方法以及 render 系列方法供 Action 使用。

3.3 getPara 系列方法

Controller 提供了 getPara 系列方法用来从请求中获取参数。getPara 系列方法分为两种类型。第一种类型为第一个形参为 String 的 getPara 系列方法。该系列方法是对 HttpServletRequest.getParameter(String name) 的封装，这类方法都是转调了 HttpServletRequest.getParameter(String name)。第二种类型为第一个形参为 int 或无形参的 getPara 系列方法。该系列方法是去获取 urlPara 中所带的参数值。getParaMap 与 getParaNames 分别对应 HttpServletRequest 的 getParameterMap 与 getParameterNames。

记忆技巧：第一个参数为 String 类型的将获取表单或 url 中问号挂参的域值。第一个参数为 int

或无参数的将获取 urlPara 中的参数值。

getPara 使用例子：

方法调用	返回值
getPara("title")	返回页面表单域名为“title”参数值
getParaToInt("age")	返回页面表单域名为“age”的参数值并转为 int 型
getPara(0)	返回 url 请求中的 urlPara 参数的第一个值，如 http://localhost/controllerKey/method/v0-v1-v2 这个请求将返回“v0”
getParaToInt(1)	返回 url 请求中的 urlPara 参数的第二个值并转换成 int 型，如 http://localhost/controllerKey/method/2-5-9 这个请求将返回 5
getParaToInt(2)	如 http://localhost/controllerKey/method/2-5-N8 这个请求将返回 -8。 注意：约定字母 N 与 n 可以表示负号，这对 urlParaSeparator 为“-”时非常有用。
getPara()	返回 url 请求中的 urlPara 参数的整体值，如 http://localhost/controllerKey/method/v0-v1-v2 这个请求将返回“v0-v1-v2”

3.4 getFile 文件上传

Controller 提供了 getFile 系列方法支持文件上传。**特别注意：**如果客户端请求为 multipart request（form 表单使用了 enctype="multipart/form-data"），那么必须先调用 getFile 系列方法才能使 getPara 系列方法正常工作，因为 multipart request 需要通过 getFile 系列方法解析请求体中的数据，包括参数。

3.5 setAttr 方法

setAttr(String, Object)转调了 HttpServletRequest.setAttribute(String, Object)，该方法可以将各种数据传递给 View 并在 View 中显示出来。

3.6 render 系列方法

render 系列方法将渲染不同类型的视图并返回给客户端。JFinal 目前支持的视图类型有：FreeMarker、JSP、Velocity、JSON、File、Text、Html 等等。除了 JFinal 支持的视图型以外，还可以通过继承 Render 抽象类来无限扩展视图类型。

通常情况下使用 Controller.render(String)方法来渲染视图，使用 Controller.render(String)时的视图类型由 JFinalConfig.configConstant(Constants constants) 配置中的 constants.setViewType(ViewType)来决定，该设置方法支持的 ViewType 有：FreeMarker、JSP、Velocity，不进行配置时的缺省配置为 FreeMarker。

假设在 JFinalConfig.configRoute(Routes routes) 中有如下 Controller 映射配置：
routes.add("/user", UserController.class, "/path"), render(String view)使用例子：

方法调用	描述
render("test.html")	渲染名为 test.html 的视图，该视图的全路径为"/path/test.html"
render("/other_path/test.html")	渲染名为 test.html 的视图，该视图的全路径为"/other_path/test.html"，即当参数以"/"开头时将采用绝对路径。

其它 render 方法使用例子：

方法调用	描述
renderFreeMarker("test.html")	渲染名为 test.html 的视图，且视图类型为 FreeMarker。
renderJsp("test.html")	渲染名为 test.html 的视图，且视图类型为 Jsp。
renderVelocity("test.html")	渲染名为 test.html 的视图，且视图类型为 Velocity。
renderJson()	将所有通过 Controller.setAttr(String, Object)设置的变量转换成 json 数据并渲染。
renderJson("users", userList)	以"users"为根，仅将 userList 中的数据转换成 json 数据并渲染。
renderJson(user)	将 user 对象转换成 json 数据并渲染。

<code>renderJson("{\"age\":18}")</code>	直接渲染 json 字符串。
<code>renderJson(new String[]{"user", "blog"})</code>	仅将 <code>setAttr("user", user)</code> 与 <code>setAttr("blog", blog)</code> 设置的属性转换成 json 并渲染。使用 <code>setAttr</code> 设置的其它属性并不转换为 json。
<code>renderFile("test.zip");</code>	渲染名为 test.zip 的文件，一般用于文件下载
<code>renderText("Hello JFinal")</code>	渲染纯文本内容"Hello JFinal"。
<code>renderHtml("Hello Html")</code>	渲染 Html 内容"Hello Html"。
<code>renderError (404 , "test.html")</code>	渲染名为 test.html 的文件，且状态为 404。
<code>renderError (500 , "test.html")</code>	渲染名为 test.html 的文件，且状态为 500。
<code>renderNull()</code>	不渲染，即不向客户端返回数据。
<code>render(new XmlRender())</code>	使用自定义的 XmlRender 来渲染。

注意：

1: IE 不支持 contentType 为 application/json,在 ajax 上传文件完成后返回 json 时 IE 提示下载文件,解决办法是使用: `render(new JsonRender(params).forIE())`。

2: 除 `renderError` 方法以外，在调用 `render` 系列的方法后程序并不会立即返回，如果需要立即返回需要使用 `return` 语句。在一个 action 中多次调用 `render` 方法只有最后一次有效。

第四章 Interceptor

4.1 概述

Interceptor 是 JFinal AOP 的实现方式，通过实现 Interceptor 接口以及使用 @Before annotation 可以细粒度地进行配置。Interceptor 并非线程安全的，线程安全拦截器需要继承 PrototypeInterceptor 来实现。

4.2 Interceptor 应用

Interceptor 接口仅仅定了一个方法 `void intercept(ActionInvocation ai)`。如下代码将在被拦截器的 Action 前后各输出一行文本到控制台：

```
import com.jfinal.aop.Interceptor;
import com.jfinal.core.ActionInvocation;
public class DemoInterceptor implements Interceptor {
    public void intercept(ActionInvocation ai) {
        System.out.println("Before action invoking");
        ai.invoke();
        System.out.println("After action invoking");
    }
}
```

如上所示，JFinal Interceptor 可以很方便地在 action 调用前后插入切面代码来实现 AOP。

4.3 Interceptor 配置

拦截器配置共分三个级别：Global 级、Controller 级、Action 级。Global 级的拦截器将对所有 Action 进行拦截，Controller 级拦截器将对 Controller 中所有 Action 进行拦截，Action 级拦截器仅对该 Action 进行拦截。

Global 级拦截器在 `JFinalConfig.configInterceptor(Interceptors me)` 进行配置。Controller 级拦截器使用 @Before annotation 放置在 Controller 类定义前进行配置。Action 级拦截器使用 @Before annotation 放置在 Action 定义前进行配置。

以下为 Controller 与 Action 级拦截器配置示例：

```
@Before(DemoInterceptor.class) // 配置一个Controller级别的拦截器
public class HelloController extends Controller {
    @Before(AaaInterceptor.class)
    public void index() {
        renderText("配置一个action级别的拦截器");
    }
    @Before({AaaInterceptor.class, BbbInterceptor.class})
    public void test() {
        renderText("配置多个action级别的拦截器");
    }
    @ClearInterceptor
    public void login() {
        renderText("清除上一级别(Controller级)的拦截器");
    }
    @ClearInterceptor(ClearLayer.ALL)
    public void clearAllLayers() {
        renderText("清除所有级别(Global级与Controller级)的拦截器");
    }
}
```

ClearInterceptor 默认仅清除上一级别的拦截器。Action 级的上一级为 Controller 级，Controller 级的上一级为 Global 级。如果需删除多级拦截器，可以为 ClearInterceptor 加入参数 ClearLayer.ALL。

以下为 Global 级拦截器配置示例：

```
public class DemoConfig extends JFinalConfig {
    public void configInterceptor(Interceptors me) {
        // 此处配置的拦截器将拦截所有Action
        me.add(new AaaInterceptor());
    }
}
```

4.4 ActionInvocation

ActionInvocation 作为 Interceptor 接口 intercept 方法中的唯一参数，提供了很多便利的方法以便在拦截器中使用。

以上为 ActionInvocation 中的方法：

方法	描述
void invoke()	继续本次调用，将调用剩下的拦截器与 Action
Controller getController()	获取本次 Action 调用所属的 Controller
String getActionKey()	获取本次 Action 调用的 action key 值
String getControllerKey()	获取本次 Action 调用的 controller key 值
Method getMethod()	获取本次 Action 调用的 Method 对象
String getMethodName()	获取本次 Action 调用的 Method 对象的名称
String getViewPath()	获取本次 Action 调用的视图路径

4.5 InterceptorStack

InterceptorStack 可将多个拦截器组合成一个拦截器，当需要对某一组拦截器重复配置使用的时候可以减少配置代码量。示例代码如下：

```
public class ManagerInterceptor extends InterceptorStack {  
    public void config() {  
        addInterceptors(new AuthInterceptor(), new DemoInterceptor());  
    }  
}
```

以上代码将两个拦截器组合为了一个拦截器，组合后的拦截器与普通拦截器配置方法完全一样，仍然为：@Before(ManagerInterceptor.class)

第五章 ActiveRecord

5.1 概述

ActiveRecord 是 JFinal 最核心的组成部分之一，通过 ActiveRecord 来操作数据库，将极大地减少代码量，极大地提升开发效率。

5.2 ActiveRecordPlugin

ActiveRecord 是作为 JFinal 的 Plugin 而存在的，所以使用时需要在 JFinalConfig 中配置 ActiveRecordPlugin。

以下是 Plugin 配置示例代码

```
public class DemoConfig extends JFinalConfig {
    public void configPlugin(Plugins me) {
        C3p0Plugin cp = new C3p0Plugin("jdbc:mysql://localhost/db_name",
            "userName", "password");
        me.add(cp);
        ActiveRecordPlugin arp = new ActiveRecordPlugin(cp);
        me.add(arp);
        arp.addMapping("user", User.class);
        arp.addMapping("article", "article_id", Article.class);
    }
}
```

以上代码配置了两个插件：C3p0Plugin 与 ActiveRecordPlugin，前者是 c3p0 数据源插件，后者是 ActiveRecord 支持插件。ActiveRecord 中定义了 addMapping(String tableName, Class<? extends Model> modelClass) 方法，该方法建立了数据库表名到 Model 的映射关系。

另外，以上代码中 arp.addMapping("user", User.class)，表的主键名为默认为“id”，如果主键名称为 “user_id” 则需要手动指定，如：arp.addMapping("user", "user_id", User.class)。

5.3 Model

Model 是 ActiveRecord 中最重要的组件之一，它充当 MVC 模式中的 Model 部分。以下是 Model 定义示例代码：


```
public class User extends Model<User> {  
    public static final User dao = new User();  
}
```

以上代码中的 User 通过继承 Model, 便立即拥有的众多方便的操作数据库的方法。在 User 中声明的 dao 静态对象是为了方便查询操作而定义的, 该对象并不是必须的。基于 ActiveRecord 的 Model 无需定义属性, 无需定义 getter、setter 方法, 无需 XML 配置, 无需 Annotation 配置, 极大降低了代码量。

以下为 Model 的一些常见用法:

```
// 创建name属性为James,age属性为25的User对象并添加到数据库  
new User().set("name", "James").set("age", 25).save();  
  
// 删除id值为25的User  
User.dao.deleteById(25);  
  
// 查询id值为25的User将其name属性改为James并更新到数据库  
User.dao.findById(25).set("name", "James").update();  
  
// 查询id值为25的user, 且仅仅取name与age两个字段的值  
User user = User.dao.findById(25, "name, age");  
  
// 获取user的name属性  
String userName = user.getStr("name");  
  
// 获取user的age属性  
Integer userAge = user.getInt("age");  
  
// 查询所有年龄大于18岁的user  
List<User> users = User.dao.find("select * from user where age>18");  
  
// 分页查询年龄大于18的user, 当前页号为1, 每页10个user  
Page<User> userPage = User.dao.paginate(1, 10, "select *", "from user  
where age > ?", 18);
```

特别注意: User 中定义的 `public static final User dao` 对象是全局共享的, 只能用于数据库查询, 不能用于数据承载对象。数据承载需要使用 `new User().set(...)` 来实现。

5.4 JFinal 独创 Db + Record 模式

Db 类及其配套的 Record 类，提供了在 Model 类之外更为丰富的数据库操作功能。使用 Db 与 Record 类时，无需对数据库表进行映射，Record 相当于一个通用的 Model。以下为 Db + Record 模式的一些常见用法：

```
// 创建name属性为James,age属性为25的record对象并添加到数据库
Record user = new Record().set("name", "James").set("age", 25);
Db.save("user", user);

// 删除id值为25的user表中的记录
Db.deleteById("user", 25);

// 查询id值为25的Record将其name属性改为James并更新到数据库
user = Db.findById("user", 25).set("name", "James");
Db.update("user", user);

// 查询id值为25的user, 且仅仅取name与age两个字段的值
user = Db.findById("user", 25, "name, age");

// 获取user的name属性
String userName = user.getStr("name");

// 获取user的age属性
Integer userAge = user.getInt("age");

// 查询所有年龄大于18岁的user
List<Record> users = Db.find("select * from user where age > 18");

// 分页查询年龄大于18的user,当前页号为1,每页10个user
Page<Record> userPage = Db.paginate(1, 10, "select *", "from user where age > ?", 18);
```

以下为事务处理示例：

```
boolean succeed = Db.tx(new IAtom() {
    public boolean run() throws SQLException {
        int count = Db.update("update account set cash = cash - ? where id = ?", 100, 123);
        int count2 = Db.update("update account set cash = cash + ? where id = ?", 100, 456);
        return count == 1 && count2 == 1;
    }
});
```

以上两次数据库更新操作在一个事务中执行，如果执行过程中发生异常或者 `invoke()` 方法返回 `false`，则自动回滚事务。

5.5 声明式事务

ActiveRecord 支持声明式事务处理，以下代码是声明式事务示例：

```
// 本例仅为示例，并未严格考虑账户状态等业务逻辑
@Before(Tx.class)
public void trans_demo() {
    // 获取转账金额
    Integer transAmount = getParaToInt("transAmount");
    // 获取转出账户id
    Integer fromAccountId = getParaToInt("fromAccountId");
    // 获取转入账户id
    Integer toAccountId = getParaToInt("toAccountId");
    // 获取转出账户对象
    Account fromAccount = Account.dao.findById(fromAccountId);
    // 获取转入账户对象
    Account toAccount = Account.dao.findById(toAccountId);
    // 转出操作
    fromAccount.set("cash", fromAccount.getInt("cash") -
transAmount).update();
    // 转入操作
    toAccount.set("cash", toAccount.getInt("cash") +
transAmount).update();
}
```

以上代码中，仅声明了一个 Tx 拦截器即为 action 添加了事务支持。除此之外 ActiveRecord 还配备了 `TxByRegex`、`TxByActionKeys`、`TxByActionMethods`，分别支持 `regex`(正则)、`actionKeys`、`actionMethods` 声明式事务，以下是示例代码：

```
public void configInterceptor(Interceptors me) {
    me.add(new TxByRegex(".*save.*"));
    me.add(new TxByActionKeys("/cash/trans", "/other"));
    me.add(new TxByActionMethods("save", "update"));
}
```

上例中的 `TxByRegex` 拦截器可通过传入正则表达式对 action 进行拦截，当 `actionKey` 被正

则匹配上将开启事务。TxByActionKeys 可以对指定的 actionKey 进行拦截并开启事务，TxByActionMethods 可以对指定的 method 进行拦截并开启事务。

注意：MySQL 数据库表必须设置为 InnoDB 引擎时才支持事务，MyISAM 并不支持事务。

5.6 Cache

ActiveRecord 可以使用缓存以大大提高性能，以下代码是 Cache 使用示例：

```
public void list() {
    List<Blog> blogList = Blog.dao.findByCache("cacheName", "key",
        "select * from blog");
    setAttr("blogList", blogList).render("list.html");
}
```

上例 findByCache 方法中的 cacheName 需要在 ehcache.xml 中配置如：<cache name="cacheName" ...>。此外 Model.paginateByCache(...)、Db.findByCache(...)、Db.paginateByCache(...)方法都提供了 cache 支持。在使用时，只需传入 cacheName、key 以及在 ehccache.xml 中配置相对应的 cacheName 就可以了。

5.7 Dialect 多数据库支持

目前 ActiveRecordPlugin 提供了 MysqlDialect、OracleDialect、AnsiSqlDialect 实现类。MysqlDialect 与 OracleDialect 分别实现对 Mysql 与 Oracle 的支持，AnsiSqlDialect 实现对遵守 ANSI SQL 数据库的支持。以下是数据库 Dialect 的配置代码：

```
public class DemoConfig extends JFinalConfig {
    public void configPlugin(Plugins me) {
        ActiveRecordPlugin arp = new ActiveRecordPlugin(...);
        me.add(arp);
        // 配置Postgresql方言
        arp.setDialect(new PostgresqlDialect());
    }
}
```

5.8 表关联操作

JFinal ActiveRecord 天然支持表关联操作，并不需要学习新的东西，JFinal 称此为无招胜

有招。表关联操作主要有两种方式：一是直接使用 sql 得到关联数据；二是在 Model 中添加获取关联数据的方法。

假定现有两张数据库表：user、blog，并且 user 到 blog 是一对多关系，blog 表中使用 user_id 关联到 user 表。如下代码演示使用第一种方式得到 user_name：

```
public void relation() {
    String sql = "select b.*, u.user_name from blog b inner
        join user u on b.user_id=u.id where b.id=?";
    Blog blog = Blog.dao.findFirst(sql, 123);
    String name = blog.getStr("user_name");
}
```

以下代码演示第二种方式在 Blog 中获取相关联的 User 以及在 User 中获取相关联的 Blog：

```
public class Blog extends Model<Blog>{
    public static final Blog dao = new Blog();

    public User getUser() {
        return User.dao.findById(get("user_id"));
    }
}

public class User extends Model<User>{
    public static final User dao = new User();

    public List<Blog> getBlogs() {
        return Blog.dao.find("select * from blog where user_id=?",
            get("id"));
    }
}
```

5.9 Oracle 支持

Oracle 数据库具有一定的特殊性，JFinal 针对这些特殊性进行了一些额外的支持以方便广大的 Oracle 使用者。以下是一个完整的 Oracle 配置示例：

```

public class DemoConfig extends JFinalConfig {
    public void configPlugin(Plugins me) {
        C3p0Plugin cp = new C3p0Plugin(.....);
        //配置Oracle驱动
        cp.setDriverClass("oracle.jdbc.driver.OracleDriver");
        me.add(cp);
        ActiveRecordPlugin arp = new ActiveRecordPlugin(cp);
        me.add(arp);
        // 配置Oracle方言
        arp.setDialect(new OracleDialect());
        // 配置属性名(字段名)大小写不敏感容器工厂
        arp.setContainerFactory(new CaseInsensitiveContainerFactory());
        arp.addMapping("user", "user_id", User.class);
    }
}

```

由于 Oracle 数据库会自动将属性名(字段名)转换成大写,所以需要手动指定主键名为大写,如: `arp.addMapping("user", "ID", User.class)`。如果想让 ActiveRecord 对属性名(字段名)的大小写不敏感可以通过设置 `CaseInsensitiveContainerFactory` 来达到,有了这个设置,则 `arp.addMapping("user", "ID", User.class)`不再需要了。

另外, Oracle 并未直接支持自增主键, JFinal 为此提供了便捷的解决方案。要让 Oracle 支持自动主键主要分为两步: 一是创建序列, 二是在 model 中使用这个序列, 具体办法如下:

1: 通过如下办法创建序列, 本例中序列名为: MY_SEQ

```

CREATE SEQUENCE MY_SEQ
INCREMENT BY 1
MINVALUE 1
MAXVALUE 9999999999999999
START WITH 1
CACHE 20;

```

2: 在 `YourModel.set(...)`中使用上面创建的序列

```

// 创建User并使用序列
User user = new User().set("id", "my_seq.nextval").set("age", 18);
user.save();
// 获取id值
Integer id = user.get("id");

```

序列的使用很简单，只需要 `yourModel.set(主键名, 序列名)`就可以了。

5.10 多数据源支持

`ActiveRecordPlugin` 可同时支持多数据源、多方言、多缓存、多事务级别等特性，对每个 `ActiveRecordPlugin` 可进行彼此独立的配置。简言之 `JFinal` 可以同时使用多数据源，并且可以针对这多个数据源配置独立的方言、缓存、事务级别等。

当使用多数据源时，只需要对每个 `ActiveRecordPlugin` 指定一个 `configName` 即可，如下是代码示例：

```
public void configPlugin(Plugins me) {
    // mysql 数据源
    C3p0Plugin dsMysql = new C3p0Plugin(...);
    me.add(dsMysql);

    // mysql ActiveRecordPlugin 实例，并指定configName为 mysql
    ActiveRecordPlugin arpMysql = new ActiveRecordPlugin("mysql", dsMysql);
    me.add(arpMysql);
    arpMysql.setCache(new EhCache());
    arpMysql.addMapping("user", User.class);

    // oracle 数据源
    C3p0Plugin dsOracle = new C3p0Plugin(...);
    me.add(dsOracle);

    // oracle ActiveRecordPlugin 实例，并指定configName为 oracle
    ActiveRecordPlugin arpOracle = new ActiveRecordPlugin("oracle", dsOracle);
    me.add(arpOracle);
    arpOracle.setDialect(new OracleDialect());
    arpOracle.setTransactionLevel(8);
    arpOracle.addMapping("blog", Blog.class);
}
```

以上代码创建了两个 `ActiveRecordPlugin` 实例 `arpMysql` 与 `arpOracle`，特别注意创建实例的同时指定其 `configName` 分别为 `mysql` 与 `oracle`。`arpMysql` 与 `arpOracle` 分别映射了不同的 `Model`，配置了不同的方言。

对于 `Model` 的使用，不同的 `Model` 会自动找到其所属的 `ActiveRecordPlugin` 实例以及相关配置进行数据库操作。对于 `Db + Record` 的使用，需要使用 `Db.use(cfgName)`方法得到数据库

操作对象，然后就可以进行数据库操作了，以下是代码示例：

```
// 查询 dsMysql数据源中的 user
List<Record> users = Db.use("mysql").find("select * from user");
// 查询 dsOracle数据源中的 blog
List<Record> blogs = Db.use("oracle").find("select * from blog");
```

以上两行代码，分别通过 `configName` 为 `mysql`、`oracle` 得到各自的数据库操作对象，然后就可以如同单数据完全一样的方式来使用数据库操作 API 了。简言之，对于 `Db + Record` 来说，多数据源相比单数据源仅需多调用一下 `Db.use(configName)`，随后的 API 使用方式完全一样。

注意最先创建的 `ActiveRecrodPlugin` 实例将会成为主数据源，可以省略 `configName`。最先创建的 `ActiveRecrodPlugin` 实例中的配置将默认成为主配置，此外还可以通过设置 `configName` 为 `DbKit.MAIN_CONFIG_NAME` 常量来设置主配置。

第六章 EhCachePlugin

6.1 概述

EhCachePlugin 是 JFinal 集成的缓存插件，通过使用 EhCachePlugin 可以提高系统的并发访问速度。

6.2 EhCachePlugin

EhCachePlugin 是作为 JFinal 的 Plugin 而存在的，所以使用时需要在 JFinalConfig 中配置 EhCachePlugin，以下是 Plugin 配置示例代码：

```
public class DemoConfig extends JFinalConfig {  
    public void configPlugin(Plugins me) {  
        me.add(new EhCachePlugin());  
    }  
}
```

6.3 CacheInterceptor

CacheInterceptor 可以将 action 所需数据全部缓存起来，下次请求到来时如果 cache 存在则直接使用数据并 render，而不会去调用 action。此用法可使 action 完全不受 cache 相关代码所污染，即插即用，以下是示例代码：

```
@Before(CacheInterceptor.class)  
public void list() {  
    List<Blog> blogList = Blog.dao.find("select * from blog");  
    User user = User.dao.findById(getParaToInt());  
    setAttr("blogList", blogList);  
    setAttr("user", user);  
    render("blog.html");  
}
```

上例中的用法将使用 actionKey 作为 cacheName，在使用之前需要在 ehcache.xml 中配置以 actionKey 命名的 cache 如：<cache name="/blog/list" ...>，注意 actionKey 作为 cacheName 配置时斜杠"/"不能省略。此外 CacheInterceptor 还可以与 CacheName 注解配合使用，以此来取代

默认的 `actionKey` 作为 `actionName`，以下是示例代码：

```
@Before(CacheInterceptor.class)
@CacheName("blogList")
public void list() {
    List<Blog> blogList = Blog.dao.find("select * from blog");
    setAttr("blogList", blogList);
    render("blog.html");
}
```

以上用法需要在 `ehcache.xml` 中配置名为 `blogList` 的 `cache` 如：`<cache name="blogList" ...>`。

6.4 EvictInterceptor

`EvictInterceptor` 可以根据 `CacheName` 注解自动清除缓存。以下是示例代码：

```
@Before(EvictInterceptor.class)
@CacheName("blogList")
public void update() {
    getModel(Blog.class).update();
    redirect("blog.html");
}
```

上例中的用法将清除 `cacheName` 为 `blogList` 的缓存数据，与其配合的 `CacheInterceptor` 会自动更新 `cacheName` 为 `blogList` 的缓存数据。

6.5 CacheKit

`CacheKit` 是缓存操作工具类，以下是示例代码：

```
public void list() {
    List<Blog> blogList = CacheKit.get("blog", "blogList");
    if (blogList == null) {
        blogList = Blog.dao.find("select * from blog");
        CacheKit.put("blog", "blogList", blogList);
    }
    setAttr("blogList", blogList);
    render("blog.html");
}
```

`CacheKit` 中最重要的两个方法是 `get(String cacheName, Object key)` 与 `put(String cacheName,`

Object key, Object value)。get 方法是从 cache 中取数据，put 方法是将数据放入 cache。参数 cacheName 与 ehcache.xml 中的<cache name="blog" ...>name 属性值对应；参数 key 是指取值用到的 key；参数 value 是被缓存的数据。

以下代码是 CacheKit 中重载的 CacheKit.get(String, String, IDataLoader)方法使用示例：

```
public void list() {
    List<Blog> blogList = CacheKit.get("blog", "blogList", new
IDataLoader() {
        public Object load() {
            return Blog.dao.find("select * from blog");
        }
    });
    setAttr("blogList", blogList);
    render("blog.html");
}
```

CacheKit.get 方法提供了一个 IDataLoader 接口，该接口中的 load()方法在缓存值不存在时才会被调用。该方法的具体操作流程是：首先以 cacheName=blog 以及 key=blogList 为参数去缓存取数据，如果缓存中数据存在就直接返回该数据，不存在则调用 IDataLoader.load()方法来获取数据。

6.6 ehcache.xml 简介

EhCache 的使用需要有 ehcache.xml 配置文件支持，该配置文件中配置了很多 cache 节点，每个 cache 节点会配置一个 name 属性，例如：<cache name="blog" ...>，该属性是 CacheKit 取值所必须的。其它配置项如 eternal、overflowToDisk、timeToIdleSeconds、timeToLiveSeconds 详见 EhCache 官方文档。

第七章 SpringPlugin

7.1 概述

SpringPlugin 是与 Spring 整合的插件, 通过使用 SpringPlugin 可以轻松与 Spring framework 整合。

7.2 SpringPlugin

SpringPlugin 是作为 JFinal 的 Plugin 而存在的, 所以使用时需要在 JFinalConfig 中配置 SpringPlugin, 以下是 Plugin 配置示例代码:

```
public class DemoConfig extends JFinalConfig {  
    public void configPlugin(Plugins me) {  
        me.add(new SpringPlugin());  
    }  
}
```

若创建 SpringPlugin 对象时未指定构造方法中的形参, SpringPlugin 将自动去 WebRoot/WEB-INF 目录下寻找 applicationContext.xml 作为配置文件进行初始化。您还可以通过另外两个构造方法指定配置文件或 ApplicationContext 对象。

7.3 IocInterceptor

IocInterceptor 拦截 action 并对其进行依赖注入, 以下是示例代码:

```
@Before(IocInterceptor.class)  
public class BlogController extends Controller {  
    private Integer id;  
    @Inject.BY_NAME  
    private BlogService blogService;  
    public void index() {  
        blogService.doService();  
        render("index.html");  
    }  
}
```

上例将对 `blogService` 按属性名称进行注入。注解`@Inject.BY_NAME` 按属性名进行注入，`@Inject.BY_TYPE` 按类型进行注入。不指定注入类型时将不会进行注入。

第八章 Validator

8.1 概述

Validator 是 JFinal 校验组件，在 Validator 类中提供了非常方便的校验方法，学习简单，使用方便。

8.2 Validator

Validator 自身实现了 Interceptor 接口，所以它也是一个拦截器，配置方式与拦截器完全一样。以下是 Validator 示例：

```
public class LoginValidator extends Validator {
    protected void validate(Controller c) {
        validateRequiredString("name", "nameMsg", "请输入用户名");
        validateRequiredString("pass", "passMsg", "请输入密码");
    }
    protected void handleError(Controller c) {
        c.keepPara("name");
        c.render("login.html");
    }
}
```

protected void validate(Controller c)方法中可以调用 validateXxx(...)系列方法进行后端校验，protected void handleError(Controller c)方法中可以调用 c.keepPara(...)方法将提交的值再传回页面以便保持原先输入的值，还可以调用 c.render(...)方法来返回相应的页面。注意 handleError(Controller c)只有在校验失败时才会调用。

8.3 Validator 配置

Validator 配置方式与拦截器完全一样，见如下代码：

```
public class UserController extends Controller {
    @Before(LoginValidator.class)    // 配置方式与拦截器一样
    public void login() {
    }
}
```

第九章 FreeMarker 基础

9.1 概述

JFinal 默认使用 FreeMarker 作为 View，为了使 eclipse jee 能正确识别 html，所以默认使用“.html”作为 FreeMarker 视图文件的扩展名(原为“.ftl”)。

如果需要使用 JSP 作为默认视图需要在 configConstant(Constants me)方法中进行配置，见如下配置：

```
public void configConstant(Constants me) {  
    me.setDevMode(true);  
    me.setViewType(ViewType.JSP);  
}
```

9.2 FreeMarker 示例

以下代码为 FreeMarker 经常使用的指令与插值：

```
<table>  
  <#list userList as user>  
    <tr>  
      <td>${user.name}</td>  
      <td>${user.age}</td>  
      <td>${user.email}</td>  
    </tr>  
  </#list>  
</table>
```

以上代码将 userList 中的 user 对象循环输出。

9.3 在 JFinal 中扩展

可以通过 FreeMarkerRender.getConfiguration().setSharedVariable(“myKit”, new MyKit()) 为 FreeMarker 设置共享工具类，在 view 中使用 \${myKit.method(para)}。

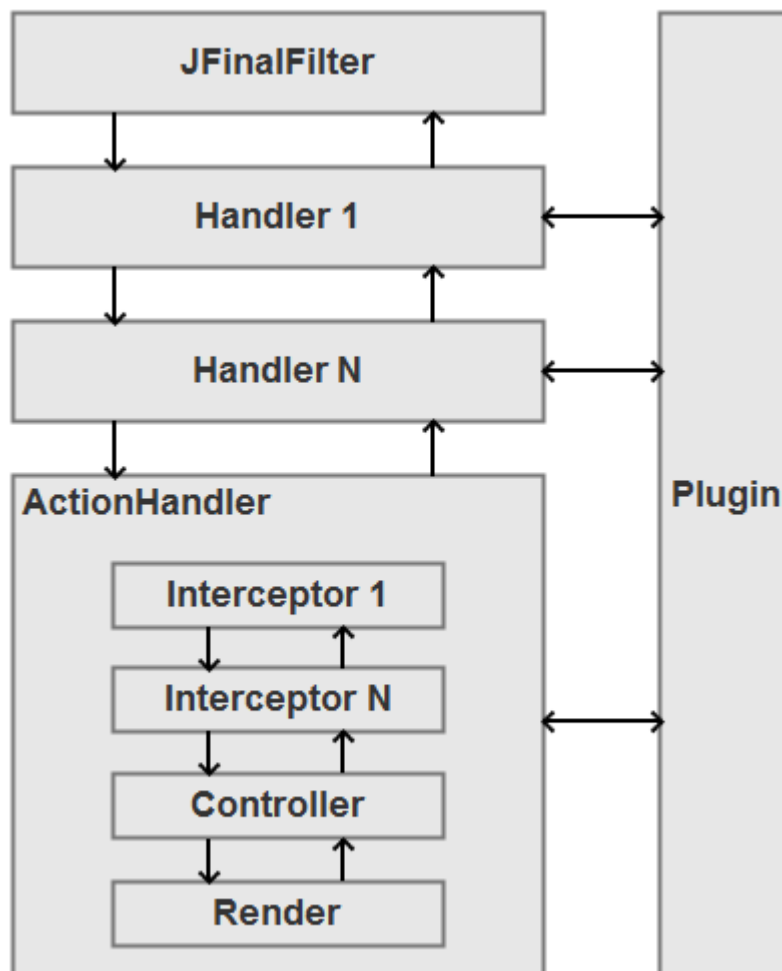
第十章 JFinal 架构及扩展

10.1 概述

JFinal 采用微内核全方位扩展架构，全方位是指其扩展方式在空间上的表现形式。JFinal 由 Handler、Interceptor、Controller、Render、Plugin 五大部分组成。本章将简单介绍此架构以及基于此架构所做的一些较为常用的扩展。

10.2 架构

JFinal 顶层架构图如下：



未完待续

JFinal 官方网站: <http://www.jfinal.com>

JFinal QQ 群: 222478625、196337924、326297041

JFinal 官方微信:

