

JFinal 手册

版本： 3.0

作者： 詹波

日期： 2017-01-21

<http://www.jfinal.com>

目录

第 0 章 升级到 3.0.....	7
0.1 configEngine	7
0.2 RenderFactory	7
0.3 baseViewPath	8
0.4 其它	9
第一章 快速上手	10
1.1 创建项目	10
1.2 放入 JFinal 库文件	12
1.3 修改 web.xml	12
1.4 添加 java 文件	12
1.5 启动项目	13
1.6 开启浏览器看效果	14
1.7 Maven 下开发	14
1.8 IDEA 下开发	15
1.9 特别声明	15
第二章 JFinalConfig.....	16
2.1 概述	16
2.2 configConstant(Constants me)	16
2.3 configRoute(Routes me)	16
2.4 configEngine(Engine me)	19
2.5 configPlugin (Plugins me)	20
2.6 configInterceptor (Interceptors me)	20
2.7 configHandler (Handlers me)	20
2.8 afterJFinalStart()与 beforeJFinalStop().....	21
2.9 PropKit	21
第三章 Controller.....	22
3.1 概述	22
3.2 Action.....	22
3.3 getPara 系列方法.....	22
3.4 getModel 与 getBean 系列方法	23
3.5 setAttr 方法.....	24

3.6 getFile 文件上传	25
3.7 renderFile 文件下载	25
3.8 session 操作方法	25
3.9 render 系列方法	25
第四章 AOP	28
4.1 概述	28
4.2 Interceptor	28
4.3 Before	29
4.4 Clear	30
4.5 Interceptor 的触发	31
4.6 Duang、Enhancer	32
4.7 Inject 拦截器	33
4.7 Routes 级别拦截器	33
第五章 ActiveRecord	35
5.1 概述	35
5.2 ActiveRecordPlugin	35
5.3 Model	35
5.4 JFinal 独创 Db + Record 模式	37
5.5 声明式事务	38
5.6 Cache	39
5.7 Dialect 多数据库支持	39
5.8 表关联操作	39
5.9 复合主键	40
5.10 Oracle 支持	41
5.11 Sql 管理与动态生成	43
5.12 多数据源支持	46
5.13 任意环境下使用 ActiveRecord	47
5.14 Generator 与 JavaBean	48
第六章 Template Engine	51
6.1 概述	51
6.2 configEngine	51
6.3 表达式	52
6.3.1 与 java 规则基本相同的表达式	52

6.3.2 属性取值表达式扩展	53
6.3.3 静态属性访问	53
6.3.4 静态方法调用	53
6.3.5 空合并安全取值调用操作符	54
6.3.6 单引号字符串	54
6.3.7 相等与不等比较表达式增强	55
6.3.8 布尔表达式增强	55
6.3.9 范围数组定义表达式	56
6.3.10 Map 定义表达式	56
6.3.11 逗号表达式	57
6.3.12 从 java 中去除的运算符	57
6.4 指令	57
6.4.1 输出指令#()	57
6.4.2 if 指令	58
6.4.3 for 指令	59
6.4.4 set 指令	61
6.4.5 include 指令	62
6.4.6 define 指令	63
6.4.7 模板函数调用	64
6.4.8 指令扩展	66
6.4.9 通过普通 java 类扩展	67
6.4.10 通过共享对象扩展	68
6.4.11 注释	68
6.4.12 非解析块	69
6.5 任意环境下使用 Engine	69
6.5.1 基本用法	69
6.5.2 进阶用法	69
6.5.3 Engine 对象管理	70
第七章 EhCachePlugin	72
7.1 概述	72
7.2 EhCachePlugin	72
7.3 CacheInterceptor	72
7.4 EvictInterceptor	73

7.5 CacheKit	73
7.6 ehcache.xml 简介	74
第八章 RedisPlugin	75
8.1 概述	75
8.2 RedisPlugin	75
8.3 Redis 与 Cache	75
8.4 非 web 环境使用 RedisPlugin	76
第九章 Cron4jPlugin	77
9.1 概述	77
9.2 Cron4jPlugin	77
9.3 使用外部配置文件	78
9.4 高级用法	79
第十章 Validator	80
10.1 概述	80
10.2 Validator	80
10.3 Validator 配置	80
第十一章 国际化	82
11.1 概述	82
11.2 I18n 与 Res	82
11.3 I18nInterceptor	83
第十二章 JFinal 架构及扩展	85
12.1 概述	85
12.2 架构	85

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，支持零配置，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
- 极简、高性能 Template Engine，十分钟内掌握基本用法
- 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 390K，且无第三方依赖

强烈建议加入 JFinal 俱乐部，获取 JFinal 最佳实践项目源代码 jfinal-club，以最快的速度、最轻松的方式掌握最简洁的用法，省去看文档的时间：<http://www.jfinal.com/club>

JFinal 官方 QQ 群: 540853725、576124753

JFinal 官方微信:



第 0 章 升级到 3.0

0.1 configEngine

JFinal 3.0 新增了模板引擎模块，继承 JFinalConfig 的实现类中需要添加 public void configEngine(Engine me)方法，以便对模板引擎进行配置。以下是示例代码：

```
public void configEngine(Engine me) {  
    me.addSharedFunction("/_view/common/__layout.html");  
    me.addSharedFunction("/_view/common/__paginate.html");  
    me.addSharedFunction("/_view/_admin/common/__admin_layout.html");  
}
```

项目升级如果不使用 Template Engine 该方法可以留空。

JFinal 3.0 默认 ViewType 为 ViewType.JFINAL_TEMPLATE，如果老项目使用的是 Freemarker 模板，并且不希望改变模板类型，需要在 configConstant 方法中通过 me.setViewType(ViewType.FREE_MARKER)进行指定，以前已经指定过 ViewType 的则不必理会。

0.2 RenderFactory

JFinal 3.0 对 render 模块做了全面重构，抽取出了 IRenderFactory 接口，而原来的 RenderFactory 成为了接口的默认实现类，去除了原来的 IMainRenderFactory、IErrorRenderFactory、IXmlRenderFactory 三个接口，所有对 render 的扩展与定制全部都可以通过继承 RenderFactory 来实现，3.0 版本的 render 模块可对所有 render 进行切换与定制，并且扩展方式完全一致。如果老项目对 IMainRenderFactory 做过扩展，只需要照如下方式进行升级：

```
public class MyRenderFactory extends RenderFactory {  
    public Render getRender(String view) {  
        return new MyRender(view);  
    }  
}
```

同理，如果以前对 IErrorRenderFactory 或者 IXmlRenderFactory 做过扩展的，只需要在上面的 MyRenderFactory 类中加上 getErrorRender(...) 与 getXmlRender(...) 方法即可。扩展完

以后在 `configConstant` 中进行如下配置：

```
public void configConstant(Constants me) {  
    me.setRenderFactory(new MyRenderFactory());  
}
```


JFinal 3.0 对所有 render 扩展，采取了完全一致的扩展方式，学习成本更低，使用更方便，升级也很方便。此外，原来 `RenderFactory` 类中的 `me()` 已经被取消，老项目对此有依赖的只需要将 `RenderFactory.me()` 直接改为 `RenderManager.me().getRenderFactory()` 即可。

0.3 baseViewPath

`baseViewPath` 设置由原来的 `configConstant(...)` 方法中转移到了 `Routes` 对象中，并且可以对不同的 `Routes` 对象分别设置，如下是示例：

```
/**  
 * 前台路由  
 */  
public class FrontRoutes extends Routes {  
  
    public void config() {  
        setBaseViewPath("/_view");  
  
        add("/", IndexController.class, "/index");  
        add("/share", ShareController.class);  
        add("/feedback", FeedbackController.class);  
        add("/project", ProjectController.class);  
        add("/login", LoginController.class);  
        add("/reg", RegController.class);  
        add("/donate", DonateController.class);  
        add("/upload", UploadController.class);  
        add("/download", DownloadController.class);  
    }  
}
```

从 `configConstant(...)` 转移到 `configRoute(...)` 中的好处是可以分别对不同的 `Routes` 进行设置，不同模块的 `baseViewPath` 很可能不相同，从而可以减少冗余代码。上面的代码示例是用于 `Routes` 拆分后的情况，如果你的应用并没有对 `Routes` 进行拆分，只需要在 `configRoute` 中如下配置即可：



```
public void configRoute(Routes me) {  
    me.setBaseViewPath("_view");  
    me.add("/", IndexController.class);  
}
```

0.4 其它

Ret.put(...).put(...)这种链式用法，需要改成 Ret.set(...).set(...)，因为 Ret 改为继承自 HashMap，为了避免与 HashMap.put(...)相冲突。Ret.get(...)方法返回泛型值的场景改为 Ret.getAs(...)。

configConstant(...) 的 Constants 参数中的 setFreeMarkerExtension、setVelocityExtension 方法统一改为使用 setViewExtension 方法。setMainRenderFactory、setErrorRenderFactory 被 setRenderFactory 取代。

renderXml(...)方法依赖的 XmlRender 由原来 Freemarker 语法实现改成了由 JFinal Template Engine 实现，用到 renderXml(...)的项目需要修改模板内容。

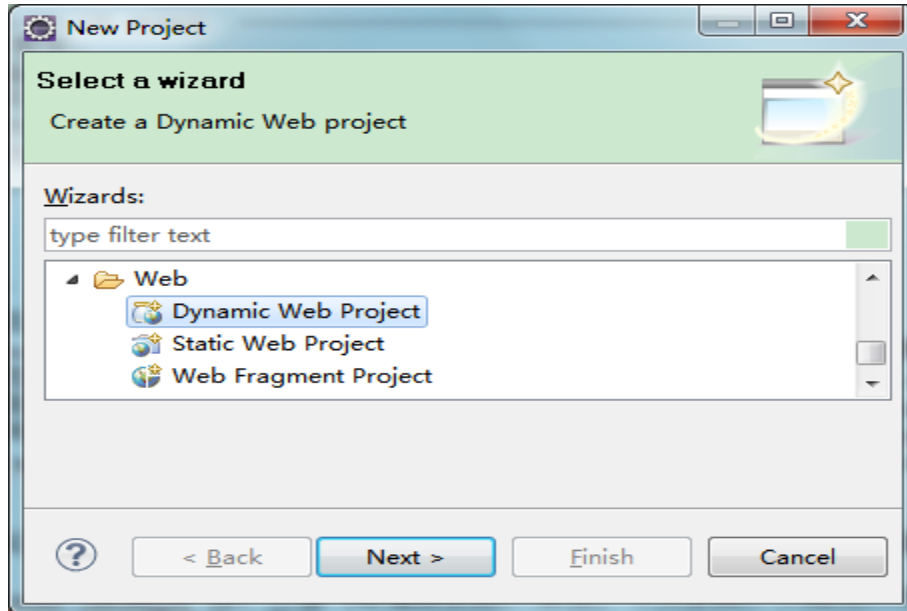
强烈建议加入 JFinal 俱乐部，获取 JFinal 最佳实践项目源代码 jfinal-club，以最快的速度、最轻松的方式掌握最简洁的用法，省去看文档的时间：<http://www.jfinal.com/club>

第一章 快速上手

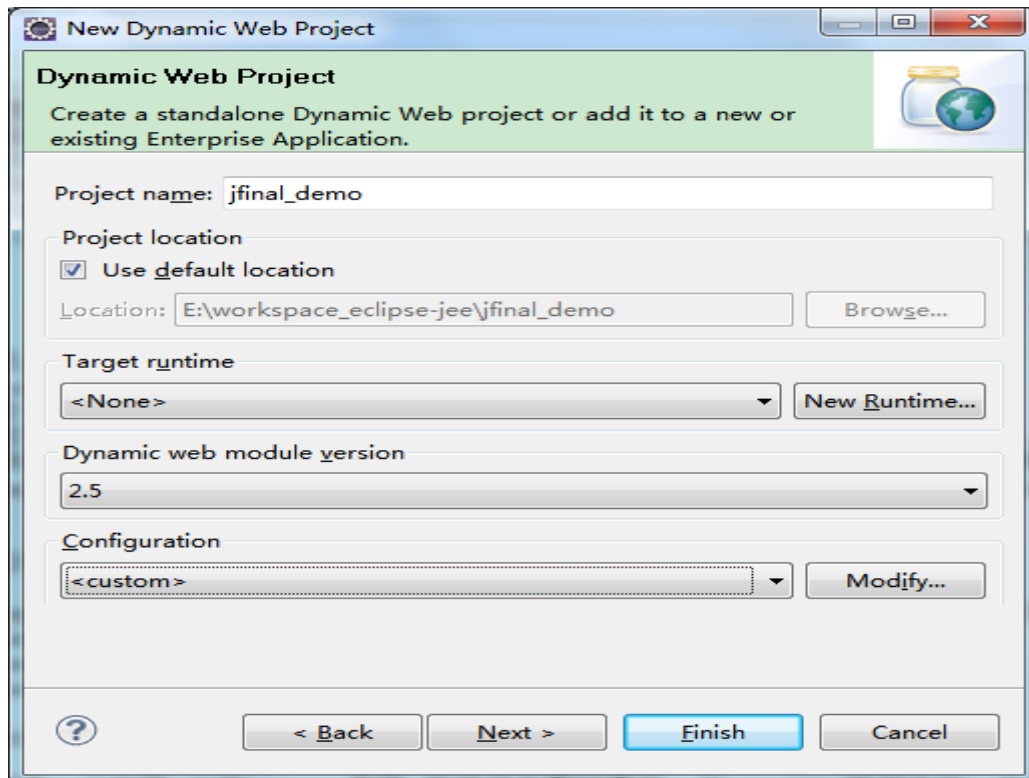
1.1 创建项目

以最常用的 Eclipse 为例，推荐使用 Eclipse IDE for Java EE Developers 版本。

- 创建 Dynamic Web Project

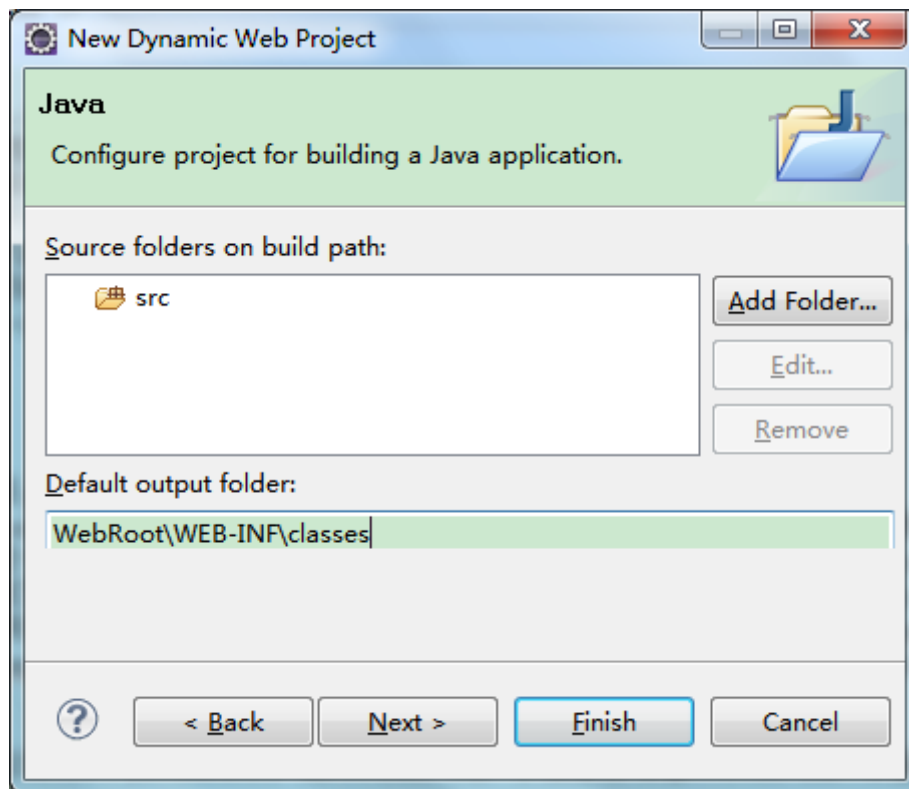


- 填入项目基本信息



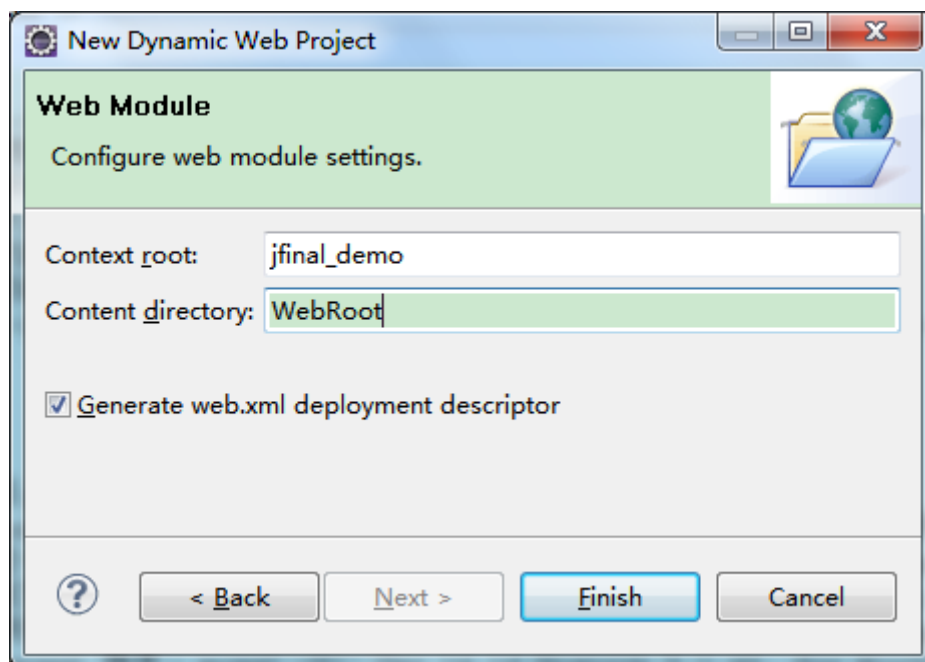
注意：Target runtime 一定要选择<None>

- 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 configEngine(Engine me) {}
    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
<http://www.jfinal.com>

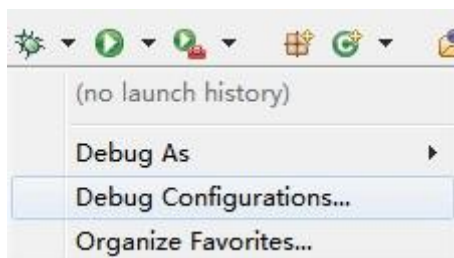
签内的配置相一致(在本例中该配置为 demo.DemoConfig)。

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

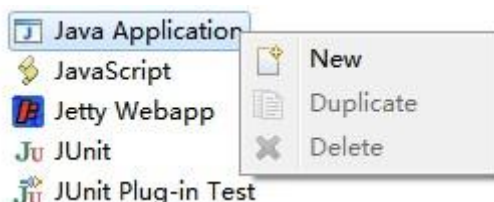
```
package demo;
import com.jfinal.core.Controller;
public class HelloController extends Controller {
    public void index() {
        renderText("Hello JFinal World.");
    }
}
```

1.5 启动项目

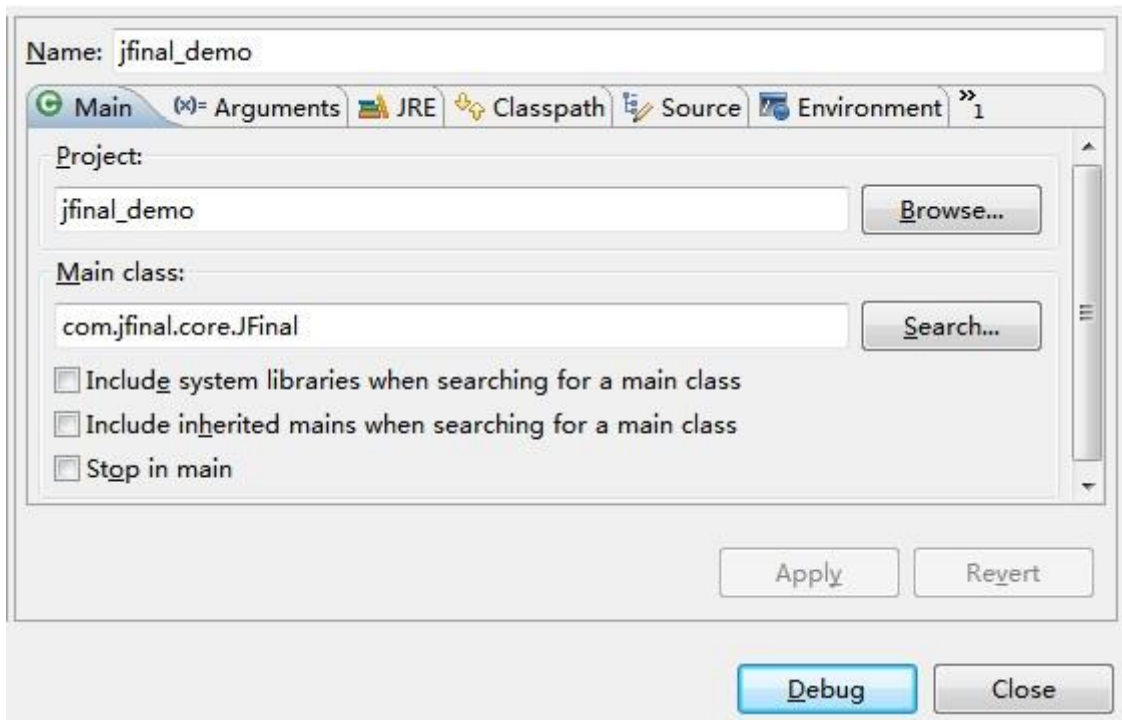
创建启动项如下图所示：



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：



上面的启动配置也可以使用一个任意的 `main` 方法代替。在任意一个类文件中添加一个 `main` 启动集成的 `jetty` 如下图所示：

```
public static void main(String[] args) {  
    JFinal.start("src/main/webapp", 80, "/");  
}
```

上图中的四个参数与前面介绍中的一样，注意根据自己的项目结构进行更改。

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 示例可在 JFinal 官方网站下载: <http://www.jfinal.com>

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`。

1.7 Maven 下开发

如果使用 Maven 管理 jar 包依赖，只需要引入如下 dependency:

```
<dependency>
  <groupId>com.jfinal</groupId>
  <artifactId>jfinal</artifactId>
  <version>3.0</version>
</dependency>
```

Maven 下启动 JFinal 与前面介绍的非 maven 方式基本相同，唯一的区别是在创建 Java Application 启动项时，在 Arguments 标签中的 Program arguments 中输入：src/main/webapp 80 / 5 参数用来指定 web 项目的根目录。如下示例代码是 main 方法启动方式：

```
public static void main(String[] args) {
    JFinal.start("src/main/webapp", 80, "/", 5);
}
```

1.8 IDEA 下开发

在 IDEA 之下使用 JFinal 整合的 Jetty 启动方式并不支持热加载，所以启动方式需要去掉最后一个参数，变成如下代码参数：

```
public static void main(String[] args) {
    JFinal.start("src/main/webapp", 80, "/");
}
```

特别注意：如果不去掉最后一个参数将无法在 IDEA 下使用 jfinal 整合的 jetty 启动

在 IDEA 下开发，建议创建一个 main 方法并且使用 JFinal.start("src/main/webapp", 80, "/") 的方式，使用 Shift + F9 的快捷键启动，在修改代码后，再使用 Ctrl + F5 的方式重启，此方式比用传统的 maven jetty plugin 要快速，注意使用 Ctrl + F5 重启前需要使用 Alt + 5 将焦点转向 debug 窗。IDEA 下开发尽量使用快捷键，避免使用鼠标，将极大提升开发率。

1.9 特别声明

JFinal 项目是符合 java web 规范的普通项目，所以开发者原有的项目启动和部署知识全部有效，不需要特殊对待 JFinal 项目。

因此，本章介绍的所有启动方式仅仅针对于 JFinal 内部提供的 jetty 整合方式。当碰到启动问题时如果不是在使用 jfinal 整合的 jetty 在启动，那么决然与 jfinal 无关，从网上查找 java web 启动的知识即可解决。

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 子类需要实现六个抽象方法，如下所示：

```
public class DemoConfig extends JFinalConfig {  
    public void configConstant(Constants me) {}  
    public void configRoute(Routes me) {}  
    public void configEngine(Engine 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，如下代码配置了 JFinal 运行在开发模式：

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

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 URL、Controller、Method 以及请求所携带的参数。

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.setBaseViewPath("/view");
    me.addInterceptor(new FrontInterceptor());
    me.add("/hello", HelloController.class);
}
```

`Routes.setBaseViewPath(baseViewPath)`方法用于为该 `Routes` 内部的所有 `Controller` 设置视图渲染时的基础路径，该基础路径与 `Routes.add(..., viewType)`方法传入的 `viewPath` 以及 `Controller.render(view)` 方法传入的 `view` 参数联合组成最终的视图路径，规则如下：

finalView = baseViewPath + viewPath + view

注意：当 `view` 以 “/” 字符打头时表示绝对路径，`baseViewPath` 与 `viewPath` 将被忽略。

`Routes` 类中添加路由的方法有两个：

```
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() {
        setBaseViewPath("/view/front");
        add("/", IndexController.class);
        add("/blog", BlogController.class);
    }
}
```

```
public class AdminRoutes extends Routes {
    public void config() {
        setBaseViewPath("/view/admin");
        addInterceptor(new AdminInterceptor());
        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 configEngine(Engine 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。

FrontRoutes 与 AdminRoutes 中分别使用 setBaseViewPath(...) 设置了各自 Controller.render(view)时使用的 baseViewPath。

AdminRoutes 还通过 addInterceptor(new AdminInterceptor())添加了 Routes 级别的拦截器，该拦截器将拦截 AdminRoutes 中添加的所有 Controller，相当于业务层的 inject 拦截器，会在 class 拦截器之前被调用。这种用法可以避免在后台管理这样的模块中的所有 class 上使用 @Before(AdminInterceptor.class)，减少代码冗余。

2.4 configEngine(Engine me)

此方法用来配置 Template Engine，以下是代码示例：

```

public void configEngine(Engine me) {
    me.addSharedFunction("/_view/common/__layout.html");
    me.addSharedFunction("/_view/common/_paginate.html");
    me.addSharedFunction("/_view/_admin/common/__admin_layout.html");
}

```

上面的方法向模板引擎中添加了三个定义了共享函数的模板文件，更详细的介绍详见后续章节。

2.5 configPlugin (Plugins me)

此方法用来配置 JFinal 的 Plugin, 如下代码配置了 Druid 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");
    DruidPlugin dp = new DruidPlugin(getProperty("jdbcUrl"),
        getProperty("user"), getProperty("password"));
    me.add(dp);
    ActiveRecordPlugin arp = new ActiveRecordPlugin(dp);
    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。

2.6 configInterceptor (Interceptors me)

此方法用来配置 JFinal 的全局拦截器, 全局拦截器将拦截所有 action 请求, 除非使用 @Clear 在 Controller 中清除, 如下代码配置了名为 AuthInterceptor 的拦截器。

```
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、Inject、Class、Method 四个层次, 其中以上代码配置粒度为全局。Inject、Class 与 Method 级的 Interceptor 配置将在后续章节中详细介绍。

2.7 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, 并对应用拥有完全的控制权, 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());
}
```

2.8 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。

2.9 PropKit

PropKit 工具类用来操作外部配置文件。PropKit 可以极度方便地在系统任意时空使用，如下是示例代码：

```
public class AppConfig extends JFinalConfig {
    public void configConstant(Constants me) {
        // 第一次使用use加载的配置将成为主配置，可以通过PropKit.get(...)直接取值
        PropKit.use("a_little_config.txt");
        me.setDevMode(PropKit.getBoolean("devMode"));
    }

    public void configPlugin(Plugins me) {
        // 非第一次使用use加载的配置，需要通过每次使用use来指定配置文件名再来取值
        String redisHost = PropKit.use("redis_config.txt").get("host");
        int redisPort = PropKit.use("redis_config.txt").getInt("port");
        RedisPlugin rp = new RedisPlugin("myRedis", redisHost, redisPort);
        me.add(rp);

        // 非第一次使用 use加载的配置，也可以先得到一个Prop对象，再通过该对象来获取值
        Prop p = PropKit.use("db_config.txt");
        DruidPlugin dp = new DruidPlugin(p.get("jdbcUrl"), p.get("user")...);
        me.add(dp);
    }
}
```

如上代码所示，PropKit 可同时加载多个配置文件，第一个被加载的配置文件可以使用 PropKit.get(...)方法直接操作，非第一个被加载的配置文件则需要使用 PropKit.use(...).get(...)来操作。PropKit 的使用并不限于在 YourJFinalConfig 中，可以在项目的任何地方使用。此外 PropKit.use(...)方法在加载配置文件内容以后会将数据缓存在内存之中，可以通过 PropKit.useless(...)将缓存的内容进行清除。

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、getModel 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者 url 中间号挂参的域值。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 getModel 与 getBean 系列方法

getModel 用来接收页面表单域传递过来的 model 对象，表单域名称以“modelName.attrName”方式命名，getModel 使用的 attrName 必须与数据表字段名完全一样。除了 getModel 以外，还提供了一个 getBean 方法用于支持传统的 Java Bean，此外 **getBean 在也可以用于使用 jfinal 生成器生成了 getter、setter 方法的 Model，此时在页面中可以使用与 setter 方法相一致的属性名，而非数据表字段名。** 以下是一个简单的示例：

```
// 定义Model, 在此为Blog
public class Blog extends Model<Blog> {
    public static final Blog me = new Blog();
}

// 在页面表单中采用modelName.attrName形式为作为表单域的name
<form action="/blog/save" method="post">
    <input name="blog.title" type="text">
    <input name="blog.content" type="text">
    <input value="提交" type="submit">
</form>

public class BlogController extends Controller {
    public void save() {
        // 页面的modelName正好是Blog类名的首字母小写
        Blog blog = getModel(Blog.class);

        // 如果表单域的名称为 "otherName.title"可加上一个参数来获取
        blog = getModel(Blog.class, "otherName");
    }
}
```

上面代码中，表单域采用了“blog.title”、“blog.content”作为表单域的 name 属性，“blog”是类文件名称“Blog”的首字母变小写，“title”是 blog 数据库表的 title 字段，如果希望表单域使用任意的 modelName，只需要在 getModel 时多添加一个参数来指定，例如：
getModel(Blog.class, "otherName")。

如果希望传参时避免使用 modelName 前缀，可以使用空串作为 modelName 来实现：getModel(Blog.class, ""); 这对开发纯 API 项目非常有用。

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 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。同样的道理在 Interceptor、Validator 中也需要先调用 getFile。

文件默认上传至项目根路径下的 upload 子路径之下, 该路径称为文件上传基础路径。可以在 JFinalConfig.configConstant(Constants me)方法中通过 me.setBaseUploadPath(baseUploadPath) 设置文件上传基础路径, 该路径参数接受以"/"打头或者以 windows 磁盘盘符打头的绝对路径, 即可将基础路径指向项目根径之外, 方便单机多实例部署。当该路径参数设置为相对路径时, 则是以项目根为基础相对路径。

3.7 renderFile 文件下载

Controller 提供了 renderFile 系列方法支持文件下载。

文件默认下载路径为项目根路径下的 download 子路径之下, 该路径称为文件下载基础路径。可以在 JFinalConfig.configConstant(Constants me) 方法中通过 me.setBaseDownloadPath(baseDownloadPath) 设置文件下载基础路径, 该路径参数接受以"/"打头或者以 windows 磁盘盘符打头的绝对路径, 即可将基础路径指向项目根径之外, 方便单机多实例部署。当该路径参数设置为相对路径时, 则是以项目根为基础相对路径。

3.8 session 操作方法

通过 setSessionAttr(key, value)可以向 session 中存放数据, getSessionAttr(key)可以从 session 中读取数据。还可以通过 getSession()得到 session 对象从而使用全面的 session API。

3.9 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: JFinal Template、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：JFINAL_TEMPLATE、FreeMarker、JSP、Velocity，不进行配置时的缺省配置为 JFINAL_TEMPLATE。

此外，还可以通过 constants.setRenderFactory(IRenderFactory)来设置 Controller 中所有 render 系列方法所使用的 Render 实现类。

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
renderTemplate("test.html")	渲染名为 test.html 的视图，且视图类型为 JFinal Template。
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。
renderJson("{\"age\":18}")	直接渲染 json 字符串。
renderJson(new String[]{"user", "blog"})	仅将 setAttr("user", user)与 setAttr("blog", blog)设置的属性转换成 json 并渲染。使用 setAttr 设置的其它属性并不转换为 json。
renderFile("test.zip");	渲染名为 test.zip 的文件，一般用于文件下载

renderText("Hello JFinal")	渲染纯文本内容"Hello JFinal"。
renderHtml("Hello Html")	渲染 Html 内容"Hello Html"。
renderError (404 , "test.html")	渲染名为 test.html 的文件，且状态为 404。
renderError (500 , "test.html")	渲染名为 test.html 的文件，且状态为 500。
renderNull()	不渲染，即不向客户端返回数据。
render(new XmlRender())	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().forIE())` 或者 `render(new JsonRender(params).forIE())`。这种情况只出现在 IE 浏览器 ajax 文件上传，其它普通 ajax 请求不必理会。

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 AOP

4.1 概述

传统 AOP 实现需要引入大量繁杂而多余的概念，例如：Aspect、Advice、Joinpoint、Pointcut、Introduction、Weaving、Around 等等，并且需要引入 IOC 容器并配合大量的 XML 或者 annotation 来进行组件装配。

传统 AOP 不但学习成本极高，开发效率极低，开发体验极差，而且还影响系统性能，尤其是在开发阶段造成项目启动缓慢，极大影响开发效率。

JFinal 采用极速化的 AOP 设计，专注 AOP 最核心的目标，将概念减少到极致，仅有三个概念：Interceptor、Before、Clear，并且无需引入 IOC 也无需使用啰嗦的 XML。

4.2 Interceptor

Interceptor 可以对方法进行拦截，并提供机会在方法的前后添加切面代码，实现 AOP 的核心目标。Interceptor 接口仅仅定了一个方法 `void intercept(Invocation inv)`。以下是简单的示例：

```
public class DemoInterceptor implements Interceptor {
    public void intercept(Invocation inv) {
        System.out.println("Before method invoking");
        inv.invoke();
        System.out.println("After method invoking");
    }
}
```

以上代码中的 `DemoInterceptor` 将拦截目标方法，并且在目标方法调用前后向控制台输出文本。`inv.invoke()` 这一行代码是对目标方法的调用，在这一行代码的前后插入切面代码可以很方便地实现 AOP。

注意：必须调用 `inv.invoke()` 方法，才能将当前调用传递到后续的 `Interceptor` 与 `Action`。

`Invocation` 作为 `Interceptor` 接口 `intercept` 方法中的唯一参数，提供了很多便利的方法在拦截器中使用。以下为 `Invocation` 中的方法：

方法	描述
void invoke()	传递本次调用，调用剩下的拦截器与目标方法
Controller getController()	获取 Action 调用的 Controller 对象（仅用于控制层拦截）
String getActionKey()	获取 Action 调用的 action key 值（仅用于控制层拦截）
String getControllerKey()	获取 Action 调用的 controller key 值（仅用于控制层拦截）
String getViewPath()	获取 Action 调用的视图路径（仅用于控制层拦截）
<T> T getTarget()	获取被拦截方法所属的对象
Method getMethod()	获取被拦截方法的 Method 对象
String getMethodName()	获取被拦截方法的方法名
Object[] getArgs()	获取被拦截方法的所有参数值
Object getArg(int)	获取被拦截方法指定序号的参数值
<T> T getReturnValue()	获取被拦截方法的返回值
void setArg(int)	设置被拦截方法指定序号的参数值
void setReturnValue(Object)	设置被拦截方法的返回值
boolean isActionInvocation()	判断是否为 Action 调用，也即是否为控制层拦截

4.3 Before

Before 注解用来对拦截器进行配置，该注解可配置 Class、Method 级别的拦截器，以下是代码示例：

```
// 配置一个Class级别的拦截器，她将拦截本类中的所有方法
@Before(AaaInter.class)
public class BlogController extends Controller {

    // 配置多个Method级别的拦截器，仅拦截本方法
    @Before({BbbInter.class, CccInter.class})
    public void index() {
    }

    // 未配置Method级别拦截器，但会被Class级别拦截器AaaInter所拦截
    public void show() {
    }
}
```

如上代码所示，Before 可以将拦截器配置为 Class 级别与 Method 级别，前者将拦截本类中所有方法，后者仅拦截本方法。此外 Before 可以同时配置多个拦截器，只需在大括号内用逗号将多个拦截器进行分隔即可。

除了 Class 与 Method 级别的拦截器以外，JFinal 还支持**全局拦截器**以及 Inject 拦截器(Inject 拦截将在后面介绍)，全局拦截器分为控制层全局拦截器与业务层全局拦截器，前者拦截控制层所有 Action 方法，后者拦截业务层所有方法。

全局拦截器需要在 YourJFinalConfig 进行配置，以下是配置示例：

```
public class AppConfig extends JFinalConfig {
    public void configInterceptor(Interceptors me) {
        // 添加控制层全局拦截器
        me.addGlobalActionInterceptor(new GlobalActionInterceptor());

        // 添加业务层全局拦截器
        me.addGlobalServiceInterceptor(new GlobalServiceInterceptor());

        // 为兼容老版本保留的方法，功能与addGlobalActionInterceptor完全一样
        me.add(new GlobalActionInterceptor());
    }
}
```

当某个 Method 被多个级别的拦截器所拦截，拦截器各级别执行的次序依次为：Global、Inject、Class、Method，如果同级中有多个拦截器，那么同级中的执行次序是：配置在前面的先执行。

4.4 Clear

拦截器从上到下依次分为 Global、Inject、Class、Method 四个层次，Clear 用于清除**自身所处层次以上层**的拦截器。

Clear 声明在 Method 层时将针对 Global、Inject、Class 进行清除。Clear 声明在 Class 层时将针对 Global、Inject 进行清除。Clear 注解携带参数时清除目标层中指定的拦截器。

Clear 用法记忆技巧：

- 共有 Global、Inject、Class、Method 四层拦截器
- 清除只针对 Clear 本身所处层的向上所有层，本层与下层不清除
- 不带参数时清除所有拦截器，带参时清除参数指定的拦截器

在某些应用场景之下，需要移除 Global 或 Class 拦截器。例如某个后台管理系统，配置了一个全局的权限拦截器，但是其登录 action 就必须清除掉她，否则无法完成登录操作，以下是

代码示例：

```
// login方法需要移除该权限拦截器才能正常登录
@Before(AuthInterceptor.class)
public class UserController extends Controller {
    // AuthInterceptor 已被Clear清除掉，不会被其拦截
    @Clear
    public void login() {
    }

    // 此方法将被AuthInterceptor拦截
    public void show() {
    }
}
```

Clear 注解带有参数时，能清除指定的拦截器，以下是一个更加全面的示例：

```
@Before(AAA.class)
public class UserController extends Controller {
    @Clear
    @Before(BBB.class)
    public void login() {
        // Global、Class级别的拦截器将被清除，但本方法上声明的BBB不受影响
    }

    @Clear({AAA.class, CCC.class}) // 清除指定的拦截器AAA与CCC
    @Before(CCC.class)
    public void show() {
        // 虽然Clear注解中指定清除CCC，但她无法被清除，因为清除操作只针对于本层以上的各层
    }
}
```

4.5 Interceptor 的触发

JFinal 中的 AOP 被划分为控制层 AOP 以及业务层 AOP，严格来说业务层 AOP 并非仅限于在业务层使用，因为 JFinal AOP 可以应用于其它任何地方。

控制层拦截器的触发，只需发起 action 请求即可。业务层拦截器的触发需要先使用 enhance 方法对目标对象进行增强，然后调用目标方法即可。以下是业务层 AOP 使用的例子：

```

// 定义需要使用AOP的业务层类
public class OrderService {
    // 配置事务拦截器
    @Before(Tx.class)
    public void payment(int orderId, int userId) {
        // service code here
    }
}

// 定义控制器，控制器提供了enhance系列方法可对目标进行AOP增强
public class OrderController extends Controller {
    public void payment() {
        // 使用 enhance方法对业务层进行增强，使其具有AOP能力
        OrderService service = enhance(OrderService.class);

        // 调用payment方法时将会触发拦截器
        service.payment(getParaToInt("orderId"), getParaToInt("userId"));
    }
}

```

以上代码中 OrderService 是业务层类，其中的 payment 方法之上配置了 Tx 事务拦截器，OrderController 是控制器，在其中使用了 enhance 方法对 OrderService 进行了增强，随后调用其 payment 方法便可触发 Tx 拦截器。简言之，业务层 AOP 的触发相对于控制层仅需多调用一次 enhance 方法即可，而 Interceptor、Before、Clear 的使用方法完全一样。

4.6 Duang、Enhancer

Duang、Enhancer 用来对目标进行增强，让其拥有 AOP 的能力。以下是代码示例：

```

public class TestMain{
    public void main(String[] args) {
        // 使用Duang.duang方法在任何地方对目标进行增强
        OrderService service = Duang.duang(OrderService.class);
        // 调用payment方法时将会触发拦截器
        service.payment(...);

        // 使用Enhancer.enhance方法在任何地方对目标进行增强
        OrderService service = Enhancer.enhance(OrderService.class);
    }
}

```

Duang.duang()、Enhancer.enhance()与 Controller.enhance()方法在功能上完全一样，她们

除了支持类增强以外，还支持对象增强，例如 `duang(new OrderService())` 以对象为参数的用法，功能本质上是一样的，在此不再赘述。

使用 `Duang`、`Enhancer` 类可以对任意目标在任何地方增强，所以 `JFinal` 的 AOP 可以应用于非 web 项目，只需要引入 `jfinal.jar` 包，然后使用 `Enhancer.enhance()` 或 `Duang.duang()` 即可极速使用 `JFinal` 的 AOP 功能。

4.7 Inject 拦截器

`Inject` 拦截器是指在使用 `enhance` 或 `duang` 方法增强时使用参数传入的拦截器。`Inject` 可以对目标完全无侵入地应用 AOP。

假如需要增强的目标在 `jar` 包之中，无法使用 `Before` 注解对其配置拦截器，此时使用 `Inject` 拦截器可以对 `jar` 包中的目标进行增强。如下是 `Inject` 拦截器示例：

```
public void injectDemo() {  
    // 为enhance方法传入的拦截器称为Inject拦截器，下面代码中的Tx称为Inject拦截器  
    OrderService service = Enhancer.enhance(OrderService.class, Tx.class);  
    service.payment(...);  
}
```

如上代码中 `Enhance.enhance()` 方法的第二个参数 `Tx.class` 被称之为 `Inject` 拦截器，使用此方法便可完全无侵入地对目标进行 AOP 增强。

`Inject` 拦截器与前面谈到的 `Global`、`Class`、`Method` 级别拦截器是同一层次上的概念。与 `Class` 级拦截器一样，`Inject` 拦截器将拦截被增强目标中的所有方法。`Inject` 拦截器可以被认为就是 `Class` 级拦截器，只不过执行次序在 `Class` 级拦截器之前而已。

4.7 Routes 级别拦截器

`Routes` 级别拦截器是指在 `Routes` 中添加的拦截器，如下是示例：

```
/**  
 * 后端路由  
 */  
public class AdminRoutes extends Routes {  
    public void config() {  
        addInterceptor(new AdminAuthInterceptor());  
        add("/admin", IndexAdminController.class, "/index");  
        add("/admin/project", ProjectAdminController.class, "/project");  
        add("/admin/share", ShareAdminController.class, "/share");  
    }  
}
```

以上的 `addInterceptor(new AdminAuthInterceptor())` 向 `AdminRoutes` 中添加了拦截器, 该拦截器 `AdminAuthInterceptor` 将拦截添加在 `AdminRoutes` 中的所有 `Controller`, 例如, 在本例中, 该拦截器将拦截 `IndexAdminController`、`ProjectAdminController`、`ShareAdminController` 中所有的 `action` 方法。

`Routes` 级别的拦截器在本质上与 `Inject` 拦截器完全相同, 同样也是 `class` 级别拦截器之前被调用。

此设计可以避免在类似后台管理这样模块的 `Controller` 类上不断重复使用 `@Before(AdminAuthInterceptor.class)`, 减少代码冗余。

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) {
        DruidPlugin dp = new DruidPlugin("jdbc:mysql://localhost/db_name",
            "userName", "password");
        me.add(dp);
        ActiveRecordPlugin arp = new ActiveRecordPlugin(dp);
        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：DruidPlugin 与 ActiveRecordPlugin，前者是 druid 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().dao();
}
```

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.findByIdLoadColumns (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.findByIdLoadColumns (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);

// 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);
```

```
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，声明式事务需要使用 ActiveRecordPlugin 提供的拦截器来实现，拦截器的配置方法见 Interceptor 有关章节。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");
    // 转出操作
    Db.update("update account set cash = cash - ? where id = ?",
        transAmount, fromAccountId);
    // 转入操作
    Db.update("update account set cash = cash + ? where id = ?",
        transAmount, toAccountId);
}
```

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 TxByActionKeys、TxByActionKeyRegex、TxByMethods、TxByMethodRegex，分别支持 actionKeys、actionKey 正则、actionMethods、actionMethod 正则声明式事务，以下是示例代码：

```
public void configInterceptor(Interceptors me) {
    me.add(new TxByMethodRegex("(.*save.*|.*update.*)"));
    me.add(new TxByMethods("save", "update"));

    me.add(new TxByActionKeyRegex("/trans.*"));
    me.add(new TxByActionKeys("/tx/save", "/tx/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，TxByMethods 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，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 复合主键

JFinal ActiveRecord 从 2.0 版本开始, 采用极简设计支持复合主键, 对于 Model 来说需要在映射时指定复合主键名称, 以下是具体例子:

```
ActiveRecordPlugin arp = new ActiveRecordPlugin(druidPlugin);
// 多数据源的配置仅仅是如下第二个参数指定一次复合主键名称
arp.addMapping("user_role", "userId, roleId", UserRole.class);

//同时指定复合主键值即可查找记录
UserRole.dao.findById(123, 456);

//同时指定复合主键值即可删除记录
UserRole.dao.deleteById(123, 456);
```


如上代码所示，对于 **Model** 来说，只需要在添加 **Model** 映射时指定复合主键名称即可开始使用复合主键，在后续的操作中 **JFinal** 会对复合主键支持的个数进行检测，当复合主键数量不正确时会报异常，尤其是复合主键数量不够时能够确保数据安全。复合主键不限定只能有两个，可以是数据库支持下的任意多个。

对于 **Db + Record** 模式来说，复合主键的使用不需要配置，直接用即可：

```
Db.findById("user_role", "roleId, userId", 123, 456);  
Db.deleteById("user_role", "roleId, userId", 123, 456);
```

5.10 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) {
        DruidPlugin dp = new DruidPlugin(.....);
        //配置Oracle驱动
        dp.setDriverClass("oracle.jdbc.driver.OracleDriver");
        me.add(dp);
        ActiveRecordPlugin arp = new ActiveRecordPlugin(dp);
        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 + ".nextval")`就可以了。特别

注意这里的“.nextval”后缀一定要是小写，OracleDialect 对该值的大小写敏感。

注意：Oracle 下分页排序 Sql 语句必须满足 2 个条件：

- Sql 语句中必须有排序条件；
- 排序条件如果没有唯一性，那么必须在后边跟上一个唯一性的条件，比如主键

相关博文：<http://database.51cto.com/art/201010/231533.htm>

相关反馈：<http://www.jfinal.com/feedback/64#replyContent>

5.11 Sql 管理与动态生成

JFinal 3.0 利用自带的 Template Engine 极为简洁的实现了 Sql 管理功能，可以将 sql 保存在外部配置文件之中，然后通过如下方式引入到 ActiveRecordPlugin 之中：

```
ActiveRecordPlugin arp = new ActiveRecordPlugin(druidPlugin);
arp.setBaseSqlTemplatePath(PathKit.getRootClassPath());
arp.addSqlTemplate("demo.sql");
_MappingKit.mapping(arp);
me.add(arp);
```

如上图所示，第二行代码 `arp.setBaseSqlTemplatePath(...)` 设置 sql 文件存放的基础路径，注意上例代码将基础路径设置为了 classpath 的根，可以将 sql 文件放在 maven 项目下的 resources 之下，编译器会自动将其编译至 classpath 之下，该路径可按喜好自由设置。

第三行代码通过 `arp.addSqlTemplate(...)` 即可添加外部 sql 文件，可以通过多次调用 `addSqlTemplate` 来添加任意多个外部 sql 文件，并且对于不同的 ActiveRecordPlugin 对象都是彼此独立配置的，非常有利于多数据源下对 sql 进行合理的划分与管理。

外部 sql 文件可以使用 template engine 的所有功能进行 sql 的动态生成，从而解决了在 java 代码中拼接复杂 sql 时的可读性差以及维护性差的问题，sql 模块提供了 `#namespace`、`#sql`、`#p` 这三个专用指令管理 sql，如下是代码示例：

```
#sql("findPrettyGirl")
select * from girl where age > ? and age < ? and weight < 50
#end
```

以上通过 `#sql` 指令定义了一条 sql 语句，其中的参数“findPrettyGirl”为获取 sql 使用的 key 值，在 java 代码中的获取方式如下：

```
find(getSql("findPrettyGirl"), 16, 23);
```

通过调用 Model 或者 Db 的 `getSql(key)` 方法，可以获取到 sql 模板中的 sql 语句。此外还提供了 `#p` 指令用于辅助生成用于查询的 sql 与 para 两个参数，如下所示：

```
#sql("findPrettyGirl")
  select * from girl where age > #p(age) and weight < #p(weight)
#end
```

注意：使用 `#p` 指令后，java 代码必须使用 `getSqlPara(...)` 方法才可以协同工作。以上 sql 定义中，使用了 `#p(age)` 与 `#p(weight)` 指令对 **参数名** 与 **参数位置** 同时进行了指定，在 java 代码中只需要传入 `age`、`weight` 这两个参数就可以同时生成 sql 与其 para，如下所示：

```
JMap cond = JMap.create("age", 18).set("weight", 50);
SqlPara sp = getSqlPara("findPrettyGirl", cond);
find(sp);
```

如上所示，将 `age`、`weight` 包装成 Map 并传给 `getSqlPara(...)` 即可自动生成 `SqlPara` 对象，该对象中仅仅只是封装了 `String sql` 与 `Object[] paras` 而已，该 `paras` 对象的次序与 sql 模板中 `#p(para)` 的次序完全一致，在使用时无需再关心 para 次序问题，直接 `find(sqlPara)` 即可，当然，也可以这样来用：`find(sqlPara.getSql(), sqlPara.getPara())`。

此外，还提供了 `#namespace` 指令为 sql 语句指定命名空间，如下所示：

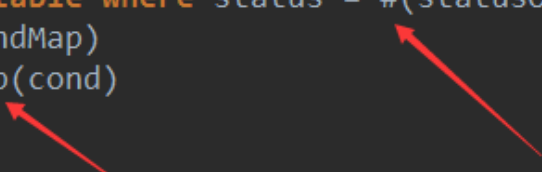
```
#namespace("japan")
  #sql("findPrettyGirl")
    select * from girl where age > ? and age < ? and weight < 50
  #end
#end
```

上面代码指定了 namespace 为 "japan"，在使用的时候，只需要在 key 前面添加 namespace 值前缀 + 句点符 + key 即可：

```
getSql("japan.findPrettyGirl");
```

以上展示了 sql 管理提供的三个专用指令：`#namespace`、`#sql`、`#p`，此外，还可以使用 JFinal Template Engine 中所有指令，生成复杂条件的 sql 语句，以下是相对复杂的示例：

```
#sql("find")
  select * from table where status = #(statusOk)
  #for(cond : condMap)
    and cond = #p(cond)
  #end
#end
```



以上代码中，`#(statusOk)`是 jfinal 模板引擎的一个输出指令，在生成 sql 时会从传入的 map 参数容器中取出名为 `statusOk` 的变量并输出该值进行 sql 拼装，而 `#p(cond)`则是前面例子中的 sql 管理专用指令，不再赘述。最后 `#for` 指令是对条件进行迭代，拼接 sql 语句。

更加高级的用法，可以用 jfinal 模板引擎的 `#define` 指令将常用的 sql 定义成通用的模板函数，然后重用该模板函数，下面是利用 id 数组删除数据的示例：

```
### 定义模板函数 deleteByIdList
#define deleteByIdList(table, idList)
  delete from #(table) where id in (
    #for (id : idList)
      #(for.index > 0 ? ", " : "") #(id)
    #end
  )
#end

### 调用上面定义的模板函数
#sql("deleteUsers")
  #@deleteByIdList("user", idList)
#end
```



如上 sql 模板先是用 template engine 所提供的 `#define` 指令定义模板函数 `deleteByIdList`，然后下方的 `#sql` 指令中对其调用，可以避免重复代码，在 java 中可使用如下代码来生成 sql：

```
List idList = Arrays.asList(1, 2, 3);
getSql("deleteUsers", JMap.create("idList", idList));
```

注意上面例子中的 `JMap` 是 JFinal 提供的用户体验更好的 Map 实现，使用任意的 Map 都可以，不限定为 `JMap`。总之，利用 sql 模块专用的三个指令再结合模板引擎已有指令自由组合，可非常简洁地实现极为强大的 sql 管理功能。

特别注意：sql 管理模块使用的模板引擎并非在 `configEngine(Engine me)`配置，因此在配置 `shared method`、`directive` 扩展时需要使用 `arp.getSqlKit().getEngine()`，然后对 `Engine` 进行配置。

5.12 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
    DruidPlugin dsMysql = new DruidPlugin(...);
    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
    DruidPlugin dsOracle = new DruidPlugin(...);
    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。假如希望同一个 Model 能够切换到不同的数据源上使用，也极度方便，这种用法非常适合不同数据源中的 table 拥有相同表结构的情况，开发者希望用同一个 Model 来操作这些相同表结构的 table，以下是示例代码：

```
public void multiDsModel() {  
    // 默认使用arp.addMapping(...)时关联起来的数据源  
    Blog blog = Blog.dao.findById(123);  
  
    // 只需调用一次use方法即可切换到另一数据源上去  
    blog.use("backupDatabase").save();  
}
```

上例中的代码，blog.use(“backupDatabase”)方法切换数据源到 backupDatabase 并直接将数据保存起来。

特别注意：只有在同一个 Model 希望对应到多个数据源的 table 时才需要使用 use 方法，如果同一个 Model 唯一对应一个数据源的一个 table，那么数据源的切换是自动的，无需使用 use 方法。

对于 Db + Record 的使用，数据源的切换需要使用 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 ActiveRecordPlugin 实例将会成为主数据源，可以省略 configName。最先创建的 ActiveRecordPlugin 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。

5.13 任意环境下使用 ActiveRecord

ActiveRecordPlugin 可以独立于 java web 环境运行在任何普通的 java 程序中，使用方式极度简单，相对于 web 项目只需要手动调用一下其 start() 方法即可立即使用。以下是代码示例：

```

public class ActiveRecordTest {
    public static void main(String[] args) {
        DruidPlugin dp = new DruidPlugin("localhost", "userName", "password");
        ActiveRecordPlugin arp = new ActiveRecordPlugin(dp);
        arp.addMapping("blog", Blog.class);

        // 与web环境唯一的不同是要手动调用一次相关插件的start()方法
        dp.start();
        arp.start();

        // 通过上面简单的几行代码，即可立即开始使用
        new Blog().set("title", "title").set("content", "cxt text").save();
        Blog.dao.findById(123);
    }
}

```

注意：ActiveRecordPlugin 所依赖的其它插件也必须手动调用一下 start()方法，如上例中的 dp.start()。

5.14 Generator 与 JavaBean

ActiveRecord 模块提供了 ModelGenerator、BaseModelGenerator、MappingKitGernator、DataDictionaryGenerator，可分别生成 Model、BaseModel、MappingKit、DataDictionary 四类文件。可根据数据表自动化生成这四类文件。

生成后的 Model 继承自 BaseModel 而非继承自 Model，BaseModel 中拥有 getter、setter 方法遵守传统 java bean 规范，Model 继承自 BaseModel 即完成了 JavaBean 与 Model 合体，拥有了传统 JavaBean 所有的优势，并且所有的 getter、setter 方法完全无需人工干预，数据表有任何变动一键重新生成即可。

使用时通常只需配置 Generator 的四个参数即可：baseModelPackageName、baseModelOutputDir、modelPackageName、modelOutputDir。四个参数分别表示 baseMode 的包名，baseModel 的输出路径，modle 的包名，model 的输出路径，以下是示例代码：


```
// base model 所使用的包名
String baseModelPkg = "model.base";
// base model 文件保存路径
String baseModelDir = PathKit.getWebRootPath() + "../src/model/base";

// model 所使用的包名
String modelPkg = "model";
// model 文件保存路径
String modelDir = baseModelDir + "../";

Generator gernerator = new Generator(dataSource, baseModelPkg, baseModelDir,
                                     modelPkg, modelDir);
gernerator.generate();
```

可在 JFinal 官网下载源码直接用于项目：<http://www.jfinal.com>

相关生成文件

BaseModel 是用于被最终的 Model 继承的基类，所有的 getter、setter 方法都将生成在此文件内，这样就保障了最终 Model 的清爽与干净，BaseModel 不需要人工维护，在数据库有任何变化时重新生成一次即可。

MappingKit 用于生成 table 到 Model 的映射关系，并且会生成主键/复合主键的配置，也即无需在 configPlugin(Plugins me)方法中书写任何样板式的映射代码。

DataDictionary 是指生成的数据字典，会生成数据表所有字段的名称、类型、长度、备注、是否主键等信息。

Model 与 Bean 合体后主要优势

- 充分利用海量的针对于 Bean 设计的第三方工具，例如 jackson、freemarker
- 快速响应数据库表变动，极速重构，提升开发效率，提升代码质量
- 拥有 IDE 代码提示不用记忆数据表字段名，消除记忆负担，避免手写字段名出现手误
- BaseModel 设计令 Model 中依然保持清爽，在表结构变化时极速重构关联代码
- 自动化 table 至 Model 映射
- 自动化主键、复合主键名称识别与映射
- MappingKit 承载映射代码，JFinalConfig 保持干净清爽
- 有利于分布式场景和无数据源时使用 Model
- 新设计避免了以往自动扫描映射设计的若干缺点：引入新概念(如注解)增加学习成本、性能低、jar 包扫描可靠性与安全性低

Model 与 Bean 合体后注意事项

- 合体后 JSP 模板输出 Bean 中的数据将依赖其 getter 方法，输出的变量名即为 getter 方法去掉“get”前缀字符后剩下的字符首字母变小写，如果希望 JSP 仍然使用之前的输出方式，可以在系统启动时调用一下 `ModelRecordElResolver.setResolveBeanAsModel(true);`
- Controller 之中的 `getModel()` 需要表单域名称对应于数据表字段名，而 `getBean()` 则依赖于 setter 方法，表单域名对应于 setter 方法去掉“set”前缀字符后剩下的字符串字母变小写。
- 许多类似于 jackson、fastjson 的第三方工具依赖于 Bean 的 getter 方法进行操作，所以只有合体后才可以使用 jackson、fastjson
- JFinalJson 将 Model 转换为 json 数据时，json 的 keyName 是原始的数据表字段名，而 jackson、fastjson 这类依赖于 getter 方法转化成的 json 的 keyName 是数据表字段名转换而成的驼峰命名
- 建议 mysql 数据表的字段名直接使用驼峰命名，这样可以令 json 的 keyName 完全一致，也可以使 JSP 在页面中取值时使用完全一致的属性名。注意：mysql 数据表的名称仍然使用下划线命名方式并使用小写字母，方便在 linux 与 windows 系统之间移植。
- 总之，合体后的 Bean 在使用时要清楚使用的是其 BaseModel 中的 getter、setter 方法还是其 Model 中的 `get(String attrName)` 方法

第六章 Template Engine

6.1 概述

JFinal Template Engine 采用独创的 DKFF (Dynamic Key Feature Forward)词法分析算法以及独创的 DLRD (Double Layer Recursive Descent)语法分析算法,极大减少了代码量,降低了学习成本,并提升了用户体验。

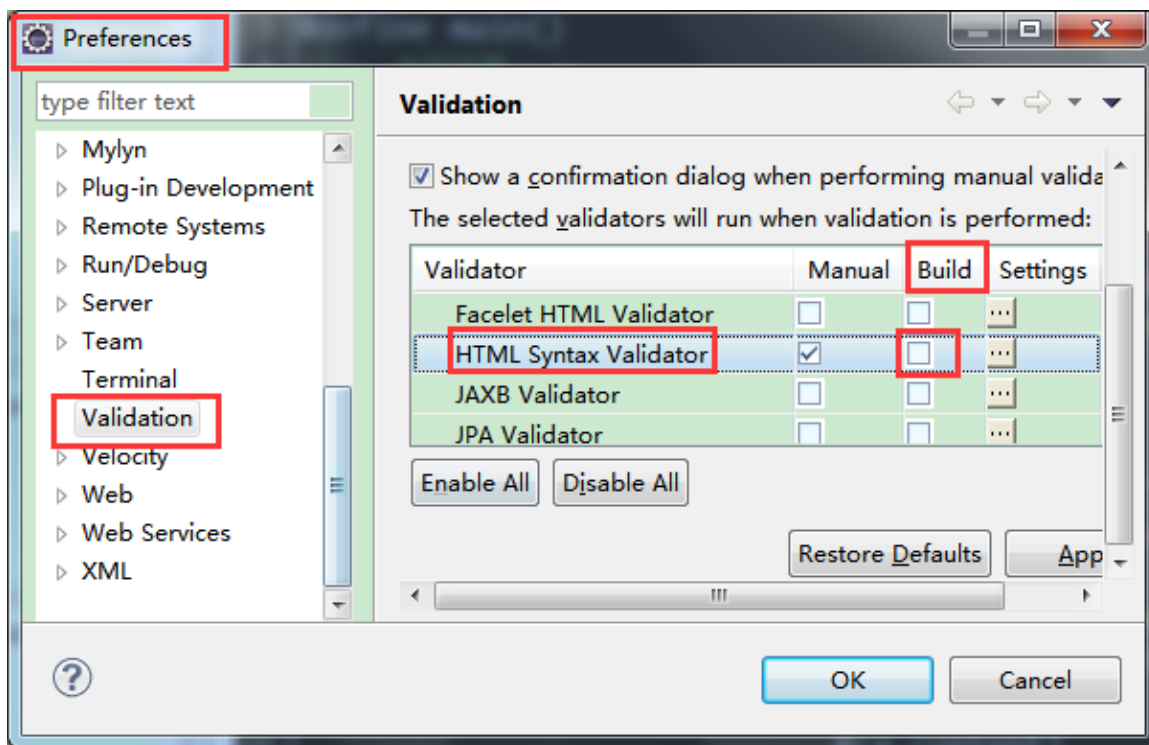
6.2 configEngine

如果在 web 环境下使用,可以通过 JFinalConfig 中的 configEngine(Engine me)抽象方法对其进行配置,如下是代码示例:

```
public void configEngine(Engine me) {  
    me.addSharedFunction("/_view/common/__layout.html");  
    me.addSharedFunction("/_view/common/_paginate.html");  
    me.addSharedFunction("/_view/_admin/common/__admin_layout.html");  
}
```

以上代码添加了三个共享函数模板文件,这三个文件中使用了#define 指令定义了一些常用的 template function。此外还可以通过 addDirective 添加用户自定义指令、通过 addShardObject 添加共享对象、通过 addSharedMethod 添加 java 类中定义的共享方法。后续将做更详细的介绍。

注意: 在 Eclipse 下开发时,需要将 Validation 配置中的 Html Syntax Validator 中的自动验证去除勾选,因为 eclipse 无法识别 JFinal Template Engine 使用的指令,从而会在指令下方显示黄色波浪线,影响美观。后续会提供相关插件给予支持,具体的配置方式见下图



6.3 表达式

JFinal Template Engine 表达式规则设计在总体上符合 java 表达式规则，仅仅针对模板引擎的特征进行极其少量的符合直觉的有利于开发体验的扩展。

6.3.1 与 java 规则基本相同的表达式

- 算术运算： + - * / % ++ --
- 比较运算： > >= < <= == != (基本用法相同，后面会介绍增强部分)
- 逻辑运算： ! && ||
- 三元表达式： ? :
- Null 值常量: null
- 字符串常量: "jfinal club"
- 布尔常量: true false
- 数字常量: 123 456F 789L 0.1D 0.2E10
- 数组存取: array[i](Map 被增强为额外支持 map[key]的方式取值, key 必须为 String 型)
- 属性取值: object.field(Map 被增强为额外支持 map.key 的方式取值)
- 方法调用: object.method(p1, p2..., pn) (支持可变参数)
- 逗号表达式: 123, 1>2, null, "abc", 3+6 (逗号表达式的值为最后一个表达式的值)

6.3.2 属性取值表达式扩展

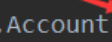
由于模板引擎的属性取值表达式极为常用，所以对其在用户体验上进行了符合直觉的扩展，field 表达式取值优先次序，以 user.name 为例：

- 假如 user.getName() 存在，则优先调用
- 假如 user 为 Model 子类，则调用 user.get("name")
- 假如 user 为 Record，则调用 user.get("name")
- 假如 user 为 Map，则调用 user.get("name")
- 假如 user 具有 public 修饰过的 name 属性，则取 user.name 属性值

6.3.3 静态属性访问

在模板中通常要访问 java 代码中定义的静态变量、静态常量，以下是代码示例：

```
#if(x.status == com.jfinal.club.common.model.Account::STATUS_LOCK_ID)
    <span>(已锁定)</span>
#end
```



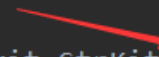
如上所示，通过**类名加双冒号再加静态属性名**即为静态属性访问表达式，上例中静态属性在 java 代码中是一个 int 数值，通过这种方式可以避免在模板中使用具体的常量值，从而有利于代码重构。

注意，这里的属性必须是 public static 修饰过的才可以被访问。此外，这里的静态属性并非要求为 final 修饰，非 final 属性还可以通过赋值表达式改变其值，如：
#set(com.jfinal.Account::STATUS = 123)。

6.3.4 静态方法调用

JFinal Template Engine 可以以非常简单的方式调用静态方法，以下是代码示例：

```
#if(com.jfinal.club.common.kit.StrKit::notBlank(title))
    .....
#end
```



使用方式与前面的静态属性访问保持一致，仅仅是将静态属性名换成静态方法名，并且后面多一对小括号与参数：**类名 + :: + 方法名(参数)**。静态方法调用支持可变参数。与静态属性相同，被调用的方法需要使用 public static 修饰才可访问。

如果觉得类名前方的包名书写很麻烦，可以使用后续即将介绍的 `me.addSharedMethod(...)` 方法将类中的方法添加为共享方法，调用的时候直接使用方法名即可，**连类名都不再需要**。

6.3.5 空合并安全取值调用操作符

JFinal Template Engine 引入了 swift 与 C#语言中的空合操作符，并在其基础之上进行了极为自然的扩展，该表达式符号为两个**紧靠**的问号：`??`。代码示例：

```
seoTitle ?? "JFinal 社区"  
object.field ??  
object.method() ??
```

以上第一行代码的功能与 swift 语言功能完全一样，也即在 `seoTitle` 值为 `null` 时整个表达式取后面表达式的值。而第二行代码表示对 `object.field` 进行空安全(Null Safe)属性取值，即在 `object` 为 `null` 时表达式不报异常，并且值为 `null`。

第三行代码与第二行代码类似，仅仅是属性取值变成了方法调用，并称之为空安全(Null Safe)方法调用，表达式在 `object` 为 `null` 时不报异常，其值也为 `null`。

当然，空合并与空安全可以极为自然地混合使用，如下是示例：

```
object.field ?? "默认值"  
object.method() ?? value
```

以上代码中，第一行代码表示左侧 `null safe` 属性取值为 `null` 时，整个表达式的值为后方的字符串中的值，而第二行代码表示值为 `null` 时整个表达式取 `value` 这个变量中的值。

特别注意：`??` 操作符的优先级高于数学计算运算符：`+`、`-`、`*`、`/`、`%`，低于单目运算符：`!`、`++`、`--`。强制改变优先级使用小括号即可。

例子：`a.b ?? && expr` 表达式中，其 `a.b ??` 为一个整体被求值，因为 `??` 优先级高于数学计算运算符，而数学计算运算符又高于 `&&` 运算符，进而推导出 `??` 优先级高于 `&&`

6.3.6 单引号字符串

针对 Template Engine 经常用于 html 的应用场景，添加了单引号字符串支持，以下是代码示例：

```
<a href="/" class="#(menu == 'index' ? 'current' : 'normal')">
  首页
</a>
```

以上代码中的三元表达式中有三处使用了单引号字符串，好处是可以与最外层的双引号协同工作，也可以反过来，最外层用单引号字符串，而内层表达式用双引号字符串。

这个设计非常有利于在模板文件中已有的双引号或单引号内容之中书写字符串表达式。

6.3.7 相等与不等比较表达式增强

相等不等表达式 `==` 与 `!=` 会对优先对左右表达式进行 `left.equals(right)` 比较操作，所以可以对字符串进行直接比较，如下所示：

```
#if(nickName == "james")
  ...
#end
```

6.3.8 布尔表达式增强

逻辑表达式在原有 java 基础之下进行了增强，吸收了 javascript 部分优点，可以减少代码输入量，具体规则自上而下优先应用如下列表：

- null 返回 false
- boolean 类型，原值返回
- Map、Connection(List 被包括在内) 返回 `size() > 0`
- 数组，返回 `length > 0`
- String、StringBuilder、StringBuffer 等继承自 CharSequence 类的对象，返回 `length > 0`
- Number 类型，返回 `intValue() != 0`
- Iterator 返回 `hasNext()` 值
- 其它返回 true

以上规则可以减少模板中的代码量，以下是示例：

```
#if(loginAccount && loginAccount.id == x.accountId)
  <a onclick="deleteReply(this, '/feedback/deleteReply?id=#(x.id)');">删除</a>
#end
```

以上代码中红色箭头所指的 `loginAccount` 表达式实质上代替了 java 表达式的

loginAccount != null 这种写法，减少了代码量。当然，上述表达式如果使用 ?? 运算符，还可以更加简单顺滑：if (loginAccount.id ?? == x.accountId)

6.3.9 范围数组定义表达式

直接举例：

```
#for(x : [1..10])  
  #(x)  
#end
```

上图红色方框加的部分定义了一个范围数组，其值为从 1 到 10 的整数数组，该表达式通常用于在开发前端页面时，模拟迭代输出多条静态数据，而又不必从后端读取数据。

此外，还支持递减的范围数组，例如：[10..1] 将定义一个从 10 到 1 的整数数组。上例中的 #for 指令与 #() 输出指令后续会详细介绍。

6.3.10 Map 定义表达式

直接举例：

```
#set(map = {k1 : 123, "k2" : "abc", 'k3' : object})  
#(map.k1)  
#(map.k2)  
#(map["k1"])  
#(map["k2"])
```

如上图所示，map 的定义使用一对大括号，每个元素以 key : value 的形式定义，多个元素之间用逗号分隔。

key 只允许是字符串或者合法的 java 变量名标识符，**注意：k1 使用的标识符而非 String 类型只是为了书写时的便利，与字符串是等价的，并不会对标识符进行表达式求值。**

上图中通过 #set 指令将定义的变量赋值给了 map 变量，第二与第三行中以 object.field 的方式进行取值，第四第五行以 array[i] 的方式进行取值。

特别注意：取值下标的值必须为 String 类型，原因是前面所讲的 Map 定义表达式的 key 只允许是字符串，如果下标为表达式，那么对该表达式求值后的结果必须为 String。

6.3.11 逗号表达式

将多个表达式使用逗号分隔开来组合而成的表达式称为逗号表达式，逗号表达式整值求值的结果为最后一个表达式的值。例如：1+2, 3*4 这个逗号表达式的值为 12。

6.3.12 从 java 中去除的运算符

针对模板引擎的应用场景，去除了位运算符，避免开发者在模板引擎中表述过于复杂，保持模板引擎的应用初衷，同时也可以提升性能。

6.4 指令

JFinal Template Engine 一如既往地坚持极简设计，核心只有#if、#for、#set、#include、#define、#(...)这六个指令，便实现了传统模板引擎几乎所有的功能，用户如果有任意一门程序语言基础，学习成本几乎为零。

如果官方提供的指令无法满足需求，还可以极其简单地在模板语言的层面对指令进行扩展，在 `com.jfinal.template.ext.directive` 包下面就有五个扩展指令，Active Record 的 sql 模块也针对 sql 管理功能扩展了三个指令，参考这些扩展指令的代码，便可无师自通，极为简单。

注意，JFinal 模板引擎指令的扩展是在词法分析、语法分析的层面进行扩展，与传统模板引擎的自定义标签类的扩展完全不是一个级别，前者可以极为全面和自由的利用模板引擎的基础设施，在更加基础的层面以极为简单直接的代码实现千变万化的功能。参考 Active Record 的 sql 管理模块，则可知其强大与便利。

6.4.1 输出指令#()

与几乎所有模板引擎不同，JFinal Template Engine 消灭了插值指令这个原本独立的概念，而是将其当成是所有指令中的一员，仅仅是指令名称省略了而已。因此，该指令的定界符与普通指令一样为小括号，从而不必像其它模板引擎一样引入额外的如大括号般的定界符。

#(...)输出指令的使用极为简单，只需要为该指令传入前面 6.4 节中介绍的任何表达式即可，指令会将这些表达式的求值结果进行输出，特别注意，当表达式的值为 null 时没有任何输出，更不会报异常。所以，对于 #(value) 这类输出不需要对 value 进行 null 值判断，如下是代码示例：

```
#(value)
#(object.field)
#(object.field ??)
#(a > b ? x : y)
#(seoTitle ?? "JFinal 俱乐部")
#(obj.method(), null)
```

如上图所示，只需要对输出指令传入表达式即可。注意上例中第一行代码 `value` 参数可以为 `null`，而第二行代码中的 `object` 为 `null` 时将会报异常，此时需要使用第三行代码中的空合安全取值调用运算符：`object.field??`

此外，注意上图最后一行代码中的输出指令参数为一个逗号表达式，逗号表达式的整体求值结果为最后一个表达式的值，而输出指令对于 `null` 值不做输出，所以这行代码相当于是仅仅调用了 `object.method()` 方法去实现某些操作。

输出指令可以自由定制，只需要实现 `IOOutputDirectiveFactory` 接口，然后在 `configEngine(Engine me)` 方法中，通过 `me.setOutputDirectiveFactory(...)` 切换即可。

6.4.2 if 指令

直接举例：

```
#if(cond)
...
#end
```

如上图所示，`if` 指令需要一个 `cond` 表达式作为参数，并且以 `#end` 为结尾符，`cond` 可以为 6.3 章节中介绍的所有表达式，包括逗号表达式，当 `cond` 求值为 `true` 时，执行 `if` 分支之中的代码。

`if` 指令必然支持 `#elseif` 与 `#else` 分支块结构，以下是示例：

```
#if(c1)
...
#elseif(c2)
...
#elseif(c3)
...
#else
...
#end
```

由于#elseif、#else 用法与 java 语法完全一样，在此不在赘述。

6.4.3 for 指令

JFinal Template Engine 对 for 指令进行了极为人性化的扩展，可以对任意类型数据进行迭代输出，包括支持 null 值迭代。以下是代码示例：

```
#for(x : list)
    #(x.field)
#end

#for(x : map)
    #(x.key)
    #(x.value)
#end
```

上图代码中展示了 for 指令迭代输出。第一个 for 指令是对 list 进行迭代输出，用法与 java 语法完全一样，第二个 for 指令是对 map 进行迭代，取值方式为 item.key 与 item.value。

注意：当被迭代的目标为 null 时，不需要做 null 值判断，for 指令会直接跳过 null 值，不进行迭代。

for 指令还支持对其状态进行获取，代码示例：

```

#for(x : listAaa)
  #(for.index)
  #(x.field)

  #for(x : listBbb)
    #(for.outer.index)
    #(for.index)
    #(x.field)
  #end
#end

```

以上代码中的`#(for.index)`、`#(for.outer.index)`是对 `for` 指令当前状态值进行获取，前者是获取当前 `for` 指令迭代的下标值(从 0 开始的整数)，后者是内层 `for` 指令获取上一层 `for` 指令的状态。这里注意 `for.outer` 这个固定的用法，专门用于在内层 `for` 指令中引用上层 `for` 指令状态。

注意：`for` 指令嵌套时，各自拥有自己的变量名作用域，规则与 `java` 语言一致，例如上例中的两个`#(x.field)`处在不同的 `for` 指令作用域内，会正确获取到所属作用域的变量值。

`for` 指令支持的所有状态值如下图：

```

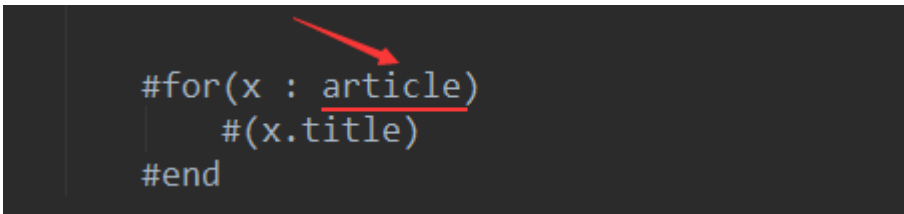
#for(x : listAaa)
  #(for.size)      被迭代对象的size值
  #(for.index)     从0开始的下标值
  #(for.count)     从1开始的记数值
  #(for.first)     是否为第一次迭代
  #(for.last)      是否为最后一次迭代
  #(for.odd)       是否为奇数次迭代
  #(for.even)      是否为偶数次迭代
  #(for.outer)     引用上层for指令状态
#end

```

具体用法已在图中用中文进行了说明，在此不再赘述。

除了 `Map`、`List` 以外，`for` 指令还支持 `Collection`、`Iterator`、`array` 普通数组、`Iterable`、`Enumeration`、`null` 值的迭代，用法在形式上与前面的 `List` 迭代完全相同，都是`#for(id : target)`的形式，对于 `null` 值，`for` 指令会直接跳过不迭代。

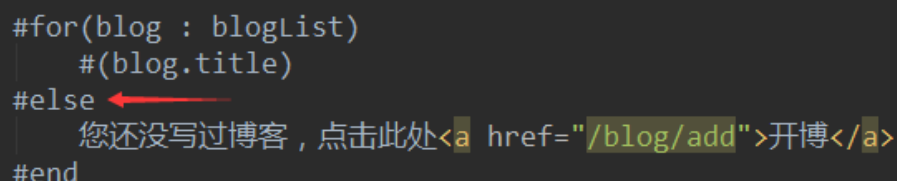
此外，`for` 指令还支持对任意类型进行迭代，此时仅仅是对该对象进行一次性迭代，如下所示：



```
#for(x : article)
  #(x.title)
#end
```

上图中的 `article` 为一个普通的 java 对象，而非集合类型对象，`for` 循环会对该对象进行一次性迭代操作，`for` 表达式中的 `x` 即为 `article` 对象本身，所以可以使用 `#(x.title)` 进行输出。

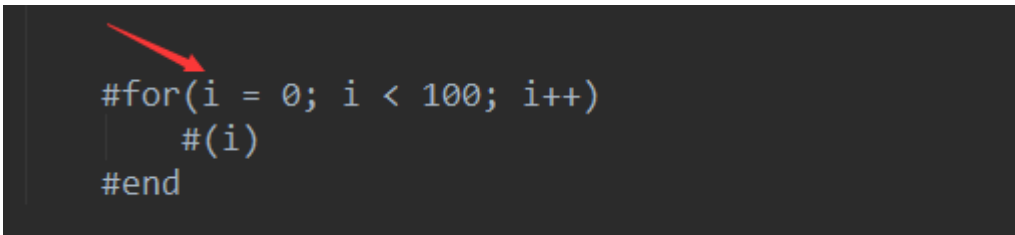
for 指令还支持 `#else` 分支语句，在 `for` 指令迭代次数为 0 时，将执行 `#else` 分支内部的语句，如下是示例：



```
#for(blog : blogList)
  #(blog.title)
#else
  您还没写过博客，点击此处
```

以上代码中，当 `blogList.size()` 为 0 或者 `blogList` 为 `null` 值时，也即迭代次数为 0 时，会执行 `#else` 分支，这种场景在 web 项目中极为常见。

最后，除了上面介绍的 `for` 指令迭代用法以外，还支持更常规的 `for` 语句形式，以下是代码示例：



```
#for(i = 0; i < 100; i++)
  #(i)
#end
```

与 java 语法基本一样，唯一的不同是变量声明不需要类型，直接用赋值语句即可，JFinal Template Engine 中的变量是动态弱类型。

注意：以上这种形式的 `for` 语句，比前面的 `for` 迭代少了 `for.size` 与 `for.last` 两个状态，只支持如下几个状态：`for.index`、`for.count`、`for.first`、`for.odd`、`for.even`、`for.outer`

`for` 指令还支持 **`continue`**、**`break`** 指令，用法与 java 完全一致，在此不再赘述。

6.4.4 set 指令

`set` 指令用于声明变量同时对其赋值，也可以是为已存在的变量进行赋值操作。`set` 指令只接受赋值表达式，以及用逗号分隔的赋值表达式列表，如下是代码示例：

```
#set(x = 123)
#set(a = 1, b = 2, c = a + b)
#set(array[0] = 123)
#set(map.key = 456)
#{x} #{a} #{array[0]} #{map.key}
```

以上代码中，第一行代码最为简单为 x 赋值为 123，第二行代码是一个赋值表达式列表，会从左到右依次执行赋值操作，如果等号右边出现表达式，将会对表达式求值以后再赋值。最后一行代码是输出上述赋值以后各变量的值，其他所有指令也可以像输出指令一样进行变量的访问。

请注意，`#for`、`#include`、`#define` 这三个指令会开启新的变量名作用域，`#set` 指令会首先在本作用域中查找变量是否存在，如果存在则对本作用域中的变量进行操作，否则继续向上层作用域查找，找到则操作，如果找不到，则将变量定义在顶层作用域中，这样设计非常有利于在模板中传递变量的值。

当需要明确需要在本层作用域赋值时，需要使用 **`#setLocal` 指令**，该指令所需参数与用法与 `#set` 指令完全一样，只不过作用域被指定为当前作用域。`#setLocal` 指令通常用于 `#define`、`#include` 指令之内，用于实现通用功能，从而希望其中的变量名不会与上层作用域发生命名上的冲突。

6.4.5 include 指令

`include` 指令用于将外部模板内容包含进来，被包含的内容会被解析成为当前模板中的一部分进行使用，如下是代码示例：

```
#include("_sidebar.html")
```

`include` 指令只接受一个 `String` 型参数，当不以 `"/"` 字符打头时被包含的模板与当前模板处于同一个目录以内，当以 `"/"` 打头时被包含模板将以 `baseTemplatePath` 为相对路径去找文件，`baseTemplatePath` 可以在 `configEngine(Engine me)` 中通过 `me.setBaseTemplatePath(...)` 进行设置，建议永远设置为：`PathKit.getWebRootPath()`，JFinal 已默认设置成了该值，不需要再干预。

通常设置 `baseTemplatePath` 的场景是单独将 `Engine` 模块用于非 web 项目。

6.4.6 define 指令

define 指令是模板引擎主要的扩展方式之一，define 指令可以定义**模板函数(Template Function)**。通过 define 指令，可以将需要被重用的模板片段定义成一个个 template function，在调用的时候可以通过传入参数实现千变万化的功能。

在此给出使用 define 指令实现的 layout 功能，首先创建一个 layout.html 文件，其中的代码如下：

```
#define layout()  
<html>  
  <head>  
    <title>JFinal俱乐部</title>  
  </head>  
  <body>  
    #@content()  
  </body>  
</html>  
#end
```

以上代码中通过#define layout()定义了一个名称为 layout 的模板函数，定义以#end 结尾，其中的#@content()表示调用一个名为 content 的模板函数。

特别注意：模板函数的调用比指令调用多一个@字符，是为了与指令调用区分开来。

接下来再创建一个模板文件，如下所示：

```
#include("layout.html")  
#@layout()  
#define content()  
<div>  
  这里是模板内容部分，相当于传统模板引擎的nested的部分  
</div>  
#end
```

上图中的第一行代码表示将前面创建的模板文件 layout.html 包含进来，第二行代码表示调用 layout.html 中定义的 layout 模板函数，而这个模板函数中又调用了 content 这个模板函数，该 content 函数已被定义在当前文件中，简单将这个过程理解为函数定义与函数调用就可以了。注意，上例实现 layout 功能的模板函数、模板文件名称可以任意取，不必像 velocity、freemarker 需要记住 nested、layoutContent 这样无聊的概念。

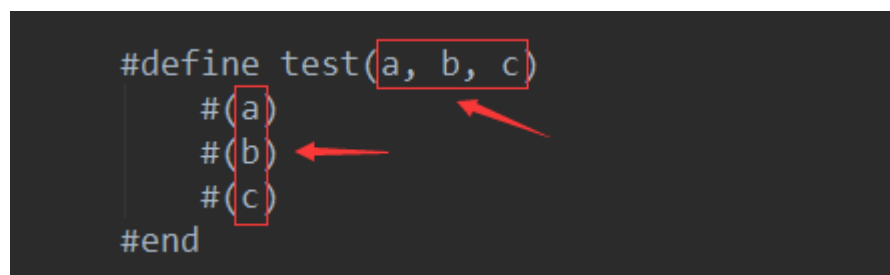
通常作为 layout 的模板文件会在很多模板中被使用，那么每次使用时都需要#include 指令进行包含，本质上是一种代码冗余，可以在 configEngine(Engine me)方法中，通过

`me.addSharedFunction("layout.html")`方法，将该模板中定义的所有模板函数设置为共享的，那么就可以省掉`#include(...)`，通过此方法可以将所有常用的模板函数全部定义成类似于共享库这样的集合，极大提高重用度、减少代码量、提升开发效率。

JFinal Template Engine 彻底消灭掉了 `layout`、`nested`、`macro` 这些无需有的概念，极大降低了学习成本，并且极大提升了扩展能力。模板引擎本质是一门程序语言，任何可用于生产环境的语言可以像呼吸空气一样自由地去实现 `layout` 这类功能。

此外，模板函数必然支持形参，用法与 `java` 规则基本相同，唯一不同的是不需要指定参数类型，只需要参数名称即可，如下是代码示例：

```
#define test(a, b, c)
    #(a)
    #(b)
    #(c)
#end
```



以上代码中的模板函数 `test`，有 `a`、`b`、`c` 三个形参，在函数体内仅简单对这三个变量进行了输出，注意形参必须是合法的 `java` 标识符，形参的作用域为该模板函数之内符合绝大多数程序语言习惯，以下是调用该模板函数的例子代码：

```
#@test(123, "abc", user.name)
```

以上代码中，第一个参数传入的整型 `123`，第二个是字符串，第三个是一个 `field` 取值表达式，从例子可以看出，实参可以是任意表达式，在调用时模板引擎会对表达式求值，并逐一赋值给模板函数的形参。

注意：形参与实参数量要相同，如果实参偶尔有更多不确定的参数要传递进去，可以在调用模板函数代码之前使用 `#set` 指令将值传递进去，在模板函数内部可用空合安全取值调用表达式进行适当控制，具体用法参考 `jfinal-club` 项目中的 `_paginate.html` 中的 `append` 变量的用法。

`define` 还支持 **return** 指令，可以在模板函数中返回，但不支持返回值。

6.4.7 模板函数调用

调用 `define` 定义的模板函数的格式为：`#@name(p1, p2..., pn)`，模板函数调用比指令调用多一个 `@` 字符，多出的 `@` 字符用来与指令调用区别开来。

此外，模板函数还支持安全调用，格式为：`#@name?(p1, p2..., pn)`，安全调用只需在模板

函数名后面添加一个问号即可。安全调用是指当模板函数未定义或者未找到时不做任何操作。

安全调用适合用于一些模板中可有可无的内容部分，以下是一个典型应用示例：

```
#define layout()
<!DOCTYPE html>
<html lang="zh-CN" xml:lang="zh-CN">
  <head>
    <link rel="stylesheet" type="text/css" href="/assets/css/jfinal-com.css">
    #@css?()
  </head>
  <body>
    <div class="jf-body-box clearfix">
      #@main()
    </div>
    <script type="text/javascript" src="/assets/js/jquery.min.js"></script>
    #@js?()
  </body>
</html>
#end
```

以上代码示例定义了一个 web 应用的 layout 模板，注意看其中的两处红色箭头指向的就是模板函数安全调用，该 layout 模板中引入了一个 css 与一个 js 资源文件，这两个文件对于所有模板都是必须的，但有些模块除了需要这两个资源文件以外，还需要额外的资源文件，那么就可以通过#define css()与#define js()这两模板函数为该 layout 提供额外的资源，如下是代码示例：

```
##@layout()   ### 调用layout.html中定义的模板函数layout()

#define main()
| 这里是body中的内容块
#end

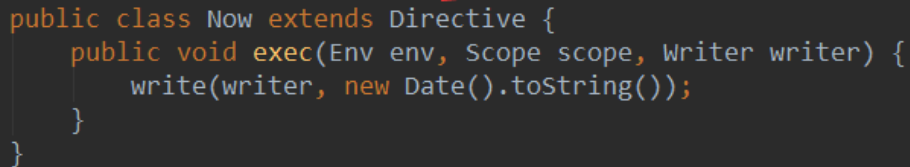
#define css()
| 在这里可以引入额外的css文件
#end

#define js()
| 在这里可以引入额外的js文件
#end
```

以上代码中先是通过#@layout()调用了前面定义过的 layout()这个模板函数，而这个模板函数中又分别调用了#@main()、#@css?()、#@js?()这三个模板函数，其中后两个是安全调用，所以对于不需要额外的 css、js 文件的模板，则不需要定义这两个方法，安全调用在调用不存在的模板函数时会直接跳过。

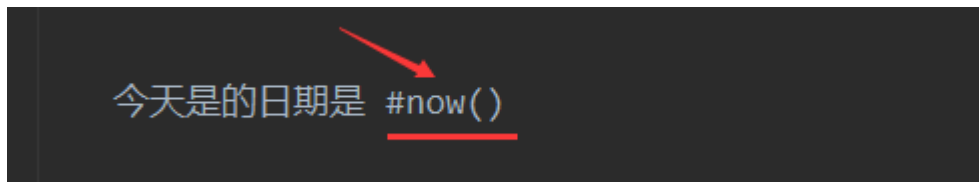
6.4.8 指令扩展

由于采用独创的 DKFF 和 DLRD 算法, JFinal Template Engine 可以极其便利地在语言层面对指令进行扩展, 而代码量少到不可想象的地步, 学习成本无限逼近于 0。以下是一个代码示例:



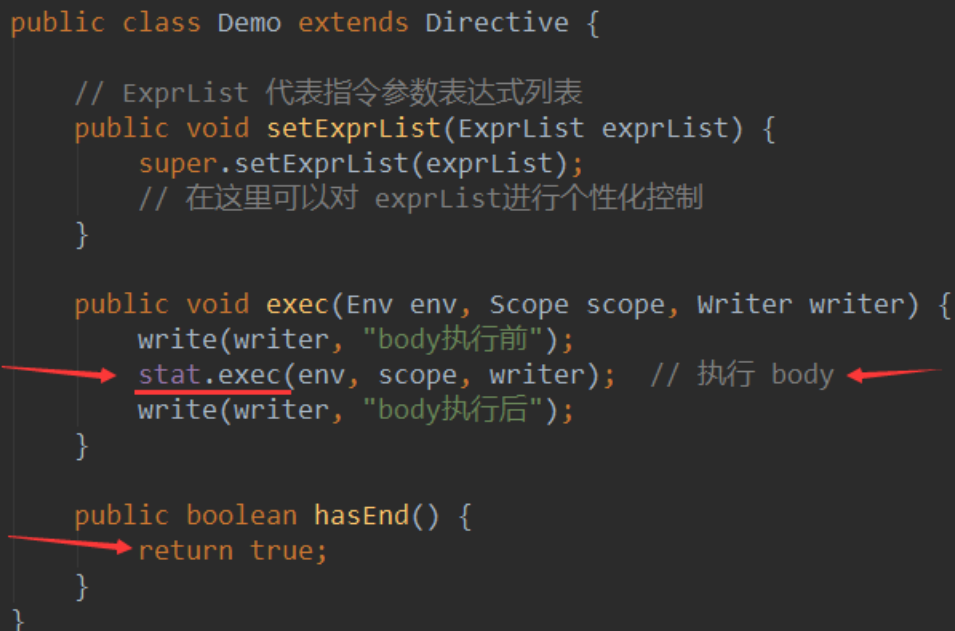
```
public class Now extends Directive {  
    public void exec(Env env, Scope scope, Writer writer) {  
        write(writer, new Date().toString());  
    }  
}
```

以上代码中, 通过继承 Directive 并实现 exec 方法, 三行代码即实现一个 #now 指令, 可以向模板中输出当前日期, 在使用前只需通过 me.addDirective("now", new Now()) 添加到模板引擎中即可。以下是在模板中使用该指令的例子:



今天是的日期是 #now()

除了支持上述无 #end 块, 也即无指令 body 的指令外, JFinal Template Engine 还直接支持包含 #end 与 body 的指令, 以下是示例:



```
public class Demo extends Directive {  
    // ExprList 代表指令参数表达式列表  
    public void setExprList(ExprList exprList) {  
        super.setExprList(exprList);  
        // 在这里可以对 exprList 进行个性化控制  
    }  
  
    public void exec(Env env, Scope scope, Writer writer) {  
        write(writer, "body执行前");  
        stat.exec(env, scope, writer); // 执行 body  
        write(writer, "body执行后");  
    }  
  
    public boolean hasEnd() {  
        return true;  
    }  
}
```

如上所示, Demo 继承 Directive 覆盖掉父类中的 hasEnd 方法, 并返回 true, 表示该扩展指令具有 #end 结尾符。上例中 public void exec 方法中的三行代码, 其中 stat.exec(...) 表示执行

指令 `body` 中的代码，而该方法前后的 `write(...)` 方法分别输出一个字符串，最终的输出结果详见后面的使用示例。此外通过覆盖父类的 `setExprList(...)` 方法可以对指令的参数进行控制，该方法并不是必须的。

通过 `me.addDirective("demo", new Demo())` 添加到引擎以后，就可以像如下代码示例中使用：

```
#demo()  
    这里是 demo body 的内容  
#end
```

最后的输出结果如下：

```
body 执行前  
这里是 demo body 的内容  
body 执行后
```

上例中的 `#demo` 指令 `body` 中包含一串字符，将被 `Demo.exec(...)` 方法中的 `stat.exec(...)` 所执行，而 `stat.exec(...)` 前后的 `write(...)` 两个方法调用产生的结果与 `body` 产生的结果生成了最终的结果。

6.4.9 通过普通 java 类扩展

JFinal Template Engine 可以极其简单的直接使用任意的 java 类，并且被使用的 java 类无需实现任何接口也无需继承任何抽象类，完全无耦合。以下代码以 JFinal 之中的 `com.jfinal.kit.StrKit` 类为例：

```
public void configEngine(Engine me) {  
    me.addSharedMethod(new com.jfinal.kit.StrKit());  
}
```

以上代码已将 `StrKit` 类中所有的 `public` 方法添加为 `shared method`，添加完成以后便可以直接在模板中使用，以下是代码示例：

```
#if(isBlank(nickName))  
    .....  
#end
```

上例中的 `isBlank` 方法就来自于 `StrKit` 类，这种扩展方式简单、便捷、无耦合。

6.4.10 通过共享对象扩展

通过使用 `addSharedObject` 方法，将某个具体对象添加为共享对象，可以全局进行使用，以下是代码示例：

```
public void configEngine(Engine me) {  
    me.addSharedObject("RESOURCE_HOST", "http://res.jfinal.com");  
    me.addSharedObject("sk", new com.jfinal.kit.StrKit());  
}
```

以上代码中的第二行，添加了一个名为 `RESOURCE_HOST` 的共享对象，而第三行代码添加了一个名为 `sk` 的共享对象，以下是在模板中的使用例子：

```
  
#if(sk.isBlank(title))  
    ...  
#end
```

以上代码第一行中使用输出指令输出了 `RESOURCE_HOST` 这个共享变量，对于大型 web 应用系统，通过这种方式可以很方便地规划资源文件所在的服务器。以上第二行代码调用了名为 `sk` 这个共享变量的 `isBlank` 方法，使用方式符合开发者直觉。

注意：由于对象被全局共享，所以需要注意线程安全问题，尽量只共享常量以及无状态对象。

6.4.11 注释

JFinal Template Engine 支持单行与多行注释，以下是代码示例：

```
### 这里是单行注释  
  
#--  
    这里是多行注释的第一行  
    这里是多行注释的第二行  
--#
```

如上所示，单行注释使用三个 `#` 字符，多行注释以 `#--` 打头，以 `--#` 结尾。与传统模板引擎不同，这里的单行注释采用三个字符，主要是为了减少与文本内容相冲突的可能性，模板内容是极为自由化的内容，如果使用三个字符，冲突的概率降低一个数量级。

6.4.12 非解析块

非解析块是指不被解析，而仅仅当成纯文本的内容区块，如下所示：

```
#[[
    #(value)
    #for(x : list)
        #(x.name)
    #end
]]#
```

如上所示，非解析块以#[[打头，以]]#结尾，中间被包裹的内容虽然是指令，但仍然被当成是纯文本。无论是单行注释、多行注释，还是非解析块，都是以三个字符开头，目的都是为了降低与纯文本内容的冲突概率。

注意：用于注释、非解析块的三个控制字符之间不能有空格

6.5 任意环境下使用 Engine

JFinal Template Engine 的使用不限于 web，可以使用在任何 java 开发环境中，使用方式极为简单。

6.5.1 基本用法

直接举例：

```
Engine.use().getTemplate("demo.html").renderToString(JMap.create("k", "v"));
```

一行代码搞定模板引擎在任意环境下的使用，将极简贯彻到底。上例中的 use()方法将从 Engine 中获取默认存在的 main Engine 对象，然后通过 getTemplate 获取 Template 对象，最后再使用 renderToString 将模板渲染到 String 之中。

6.5.2 进阶用法

直接举例：

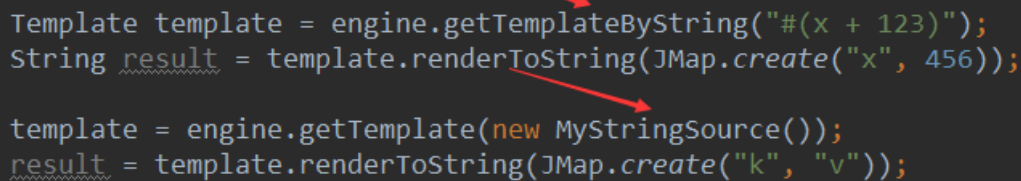
```
Engine engine = Engine.create("myEngine");
engine.setBaseTemplatePath(PathKit.getWebRootPath());
Template template = engine.getTemplate("wxAppMsg.txt");
String wxAppMsg = template.renderToString(JMap.create("toUser", "james"));

engine = Engine.use("myEngine");
```

上例第一行代码创建了名为”myEngine”的 Engine 对象，第二行代码设置引擎查找模板文件的基础中径，第三行代码利用 wxAppMsg.txt 这个模板创建一个 Template 对象，第三行代码使用该对象渲染内容到 String 对象中，从而生成了微信小程序消息内容。注意，最后一行代码使用 use 方法获取到了第一行代码创建的 engine 对象，意味着使用正确的 engineName 可以在任何地方获取到之前创建的 engine 对象，极为方便。

除了可以将模板渲染到 String 中以外，还可以渲染到任意的 Writer 之中，只需要用一下 Template.render(Map data, java.io.Writer writer)方法即可实现，例如：Writer 接口如果指向文件，那么就将其内容渲染到文件之中，甚至可以实现 Writer 接口将内容渲染到 socket 套接字中。

除了外部模板文件可以作为模板内容的来源以外，还可以通过 String 数据或者 IStringSource 接口实现类作为模板数据的来源，以下是代码示例：



```
Template template = engine.getTemplateByString("#(x + 123)");
String result = template.renderToString(JMap.create("x", 456));

template = engine.getTemplate(new MyStringSource());
result = template.renderToString(JMap.create("k", "v"));
```

上例代码第一行通过 getTemplateByString 来获取 Template 对象，而非从外部模板文件来获取，这种用法非常适合模板内容非常简短的情况，避免了创建外部模板文件，例如：非常适合用于替换 JDK 中的 String.format(...)方法。

上例中的第三行代码，传入的参数是 new MyStringSource()，MyStringSource 类是 IStringSource 接口的实现类，通过该接口可以实现通过任意方式来获取模板内容，例如，通过网络 socket 连接来获取，IStringSource 接口用法极其简单，在此不再赘述。

6.5.3 Engine 对象管理

Engine 对象的创建方式有两种，一种是通过 Engine.create(...)方法，另一种是直接使用 new Engine(...)语句，前者创建的对象是在 Engine 模块管辖之内，可以通过 Engine.use(...)获取到，

而后者创建的对象脱离了 Engine 模块管辖，无法通过 Engine.use(...)获取到，开发者需要自行管理。

JFinal 的 render 模块以及 activerecord 模块使用 new Engine(...)创建实例，无法通过 Engine.use(...)获取到，前者可以通过 RenderManager.me().getEngine()获取到，后者可以通过 DbKit.getConfig().getSqlKit().getEngine()或者 ActiveRecordPlugin.getEngine()获取到。

Engine 对象管理的设计，允许在同一个应用程序中独立且自由地使用多个 Engine 实例，JFinal 自身的 render、activerecord 对 Engine 的独立使用就是典型的例子。这个设计还使开发者不仅可以利用官方提供的 create、use 方法来创建和使用 Engine 对象，而且还可以通过 new Engine(...)的方式来创建更为自由、游离的对象自行去管理，有利于实现更为独立的模块功能。

强烈建议加入 JFinal 俱乐部，获取极为全面的 jfinal 最佳实践项目源代码 jfinal-club，项目中包含大量的**模板引擎使用实例**，可以用最快的速度，几乎零成本的轻松方式，掌握 JFinal Template Engine 最简洁的用法，省去看文档的时间：**<http://www.jfinal.com/club>**

第七章 EhCachePlugin

7.1 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。

7.2 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());  
    }  
}
```

7.3 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" ...>。

7.4 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。

7.5 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。

7.6 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。

第八章 RedisPlugin

8.1 概述

RedisPlugin 是支持 Redis 的极速化插件。使用 RedisPlugin 可以极度方便的使用 redis，该插件不仅提供了丰富的 API，而且还同时支持多 redis 服务端。Redis 拥有超高的性能，丰富的数据结构，天然支持数据持久化，是目前应用非常广泛的 nosql 数据库。对于 redis 的有效应用可极大提升系统性能，节省硬件成本。

8.2 RedisPlugin

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

```
public class DemoConfig extends JFinalConfig {
    public void configPlugin(Plugins me) {
        // 用于缓存bbs模块的redis服务
        RedisPlugin bbsRedis = new RedisPlugin("bbs", "localhost");
        me.add(bbsRedis);

        // 用于缓存news模块的redis服务
        RedisPlugin newsRedis = new RedisPlugin("news", "192.168.3.9");
        me.add(newsRedis);
    }
}
```

以上代码创建了两个 RedisPlugin 对象，分别为 bbsRedis 和 newsRedis。最先创建的 RedisPlugin 对象所持有的 Cache 对象将成为主缓存对象，主缓存对象可通过 Redis.use()直接获取，否则需要提供 cacheName 参数才能获取，例如：Redis.use("news")

8.3 Redis 与 Cache

Redis 与 Cache 联合起来可以非常方便地使用 Redis 服务，Redis 对象通过 use()方法来获取到 Cache 对象，Cache 对象提供了丰富的 API 用于使用 Redis 服务，下面是具体使用示例：

```

public void redisDemo() {
    // 获取名称为bbs的Redis Cache对象
    Cache bbsCache = Redis.use("bbs");
    bbsCache.set("key", "value");
    bbsCache.get("key");

    // 获取名称为news的Redis Cache对象
    Cache newsCache = Redis.use("news");
    newsCache.set("k", "v");
    newsCache.get("k");

    // 最先创建的Cache将成为主Cache，所以可以省去cacheName参数来获取
    bbsCache = Redis.use();    // 主缓存可以省去cacheName参数
    bbsCache.set("jfinal", "awesome");
}

```

以上代码中通过”bbs”、”news”做为 use 方法的参数分别获取到了两个 Cache 对象，使用这两个对象即可操作其所对应的 Redis 服务端。

通常情况下只会创建一个 RedisPlugin 连接一个 redis 服务端，使用 Redis.use().set(key,value) 即可。

注意：使用 incr、incrBy、decr、decrBy 方法操作的计数器，需要使用 getCounter(key) 进行读取而不能使用 get(key)，否则会抛反序列化异常

8.4 非 web 环境使用 RedisPlugin

RedisPlugin 也可以在非 web 环境下使用，只需引入 jfinal.jar 然后多调用一下 redisPlugin.start()即可，以下是代码示例：

```

public class RedisTest {
    public static void main(String[] args) {
        RedisPlugin rp = new RedisPlugin("myRedis", "localhost");
        // 与web下唯一区别是需要这里调用一次start()方法
        rp.start();

        Redis.use().set("key", "value");
        Redis.use().get("key");
    }
}

```

第九章 Cron4jPlugin


9.1 概述

Cron4jPlugin 是 JFinal 集成的任务调度插件，通过使用 Cron4jPlugin 可以使用通用的 cron 表达式极为便利的实现任务调度功能。

9.2 Cron4jPlugin

Cron4jPlugin 是作为 JFinal 的 Plugin 而存在的，所以使用时需要在 JFinalConfig 中配置，如下是代码示例：

```
Cron4jPlugin cp = new Cron4jPlugin();  
cp.addTask("* * * * *", new MyTask());  
me.add(cp);
```



如上所示创建插件、addTask 传入参数，并添加到 JFinal 即完成了基本配置，上图所示红色箭头所指的第一个字符串参数是用于任务调度的 cron 表达式，第二个参数是 Runnable 接口的一个实现类，Cron4jPlugin 会根据 cron 表达式调用 MyTask 中的 run 方法。

请注意，cron 表达最多只允许五部分，每部分用空格分隔开来，这五部分从左到右依次表示分、时、天、月、周，其具体规则如下：

- 分：从 0 到 59
- 时：从 0 到 23
- 天：从 1 到 31，字母 L 可以表示月的最后一天
- 月：从 1 到 12，可以别名：jan, "feb", "mar", "apr", "may", "jun", "jul", "aug", "sep", "oct", "nov" and "dec"
- 周：从 0 到 6, 0 表示周日, 6 表示周六, 可以使用别名： "sun", "mon", "tue", "wed", "thu", "fri" and "sat"

如上五部分的分、时、天、月、周又分别支持如下字符，其用法如下：

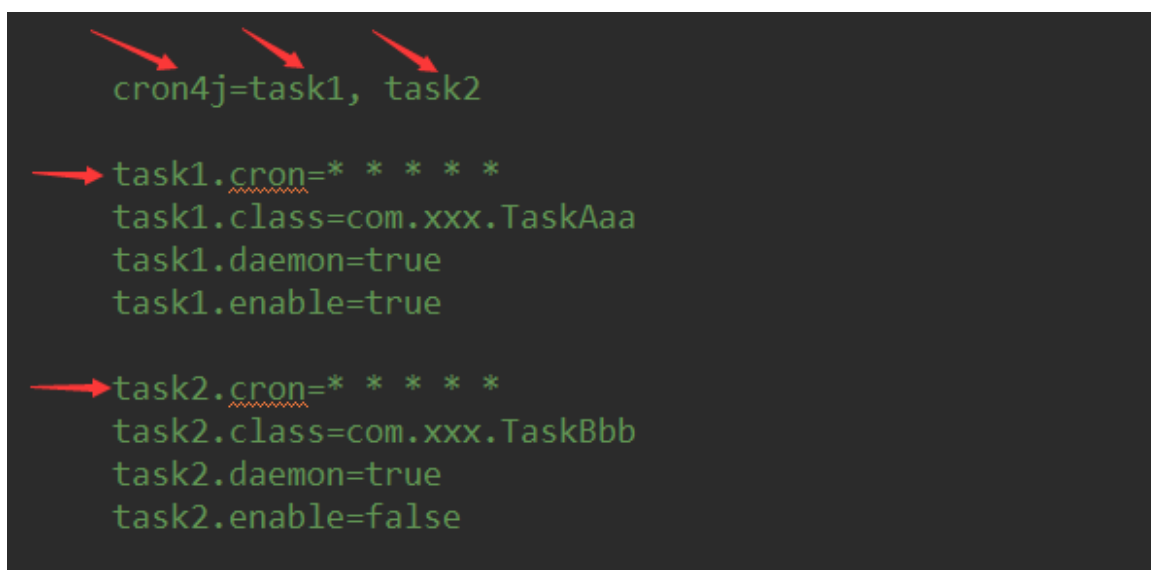
- 数字 n：表示一个具体的时间点，例如 5 * * * * 表示 5 分这个时间点时执行
- 逗号 ,：表示指定多个数值，例如 3,5 * * * * 表示 3 和 5 分这两个时间点执行

- 减号 -: 表示范围, 例如 1-3 * * * * 表示 1 分、2 分再到 3 分这三个时间点执行
- 星号 *: 表示每一个时间点, 例如 * * * * * 表示每分钟执行
- 除号 /: 表示指定一个值的增加幅度。例如 n/m 表示从 n 开始, 每次增加 m 的时间点执行

以上规则不是 JFinal 创造的, 是通用的 cron 表达式规则, 如果开发者本身具有这方面的知识, 用起来会得心应手。

9.3 使用外部配置文件

上一个示例仅展示了 java 硬编码式的配置, 更多的应用场景是使用外部配置文件, 灵活配置调度策略, 以便于随时改变调度策略, 如下是外部配置的代码示例:



```
cron4j=task1, task2

task1.cron=* * * * *
task1.class=com.xxx.TaskAaa
task1.daemon=true
task1.enable=true

task2.cron=* * * * *
task2.class=com.xxx.TaskBbb
task2.daemon=true
task2.enable=false
```

上图中的 cron4j 是所谓的配置名称: configName, 可以随便取名, 这个名称在创建 Cron4jPlugin 对象时会被用到, 如果创建 Cron4jPlugin 对象时不提供名称则默认值为“cron4j”。

上图中的 configName 后面紧跟着的是 task1、task2, 表示当前配置的两个 task 的名称, 这两个名称规定了后续的配置将以其打头, 例如后面的 task1.cron、task2.cron 都是以这两个 task 名称打头的。

上图中的 task1.cron 是指该 task 的 cron 表达式, task1.class 是指该 task 要调度的目标 java 类, 该 java 类需要实现 Runnable 接口, task1.daemon 是指被调度的任务线程是否为守护线程, task1.enable 是指该 task 是开启还是停用, 这个配置不是必须的, 可以省略, 省略时默认表示开启。同理 task2 的配置与 task1 的意义相同, 只是 taskName 不同。

总结一下: configName 指定 taskName, 多个 taskName 可以逗号分隔, 而每个 taskName 指定具体的 task, 每个具体的 task 有四项配置: cron、class、daemon、enable, 每个配置以 taskName

打头。

假定配置文件名为 config.txt，配置完成以后 Cron4jPlugin 的创建方式可以如下：

```
cp = new Cron4jPlugin("config.txt");
cp = new Cron4jPlugin("config.txt", "cron4j");

cp = new Cron4jPlugin(PropKit.use("config.txt"));
cp = new Cron4jPlugin(PropKit.use("config.txt"), "cron4j");

me.add(cp);
```

以上代码中，前四行代码是利用配置文件创建 Cron4jPlugin 对象的四种方式，第一行代码只传入了配置文件名，省去了 configName，那么默认值为“cron4j”。第二代码与第一行代码本质一样，只是指定了其 configName。第三与第四行代码与前两行代码类似，仅仅是利用 PropKit 对其进行了加载。

请注意：这里所说的 configName，就是前面示例中配置项 **cron4j**=task1, task2 中的“**cron4j**”，这个 configName 相当于就是 Cron4jPlugin 寻找的配置入口。

9.4 高级用法

除了可以对实现了 Runnable 接口的 java 类进行调度以外，还可以直接调度外部的应用程序，例如 windows 或 linux 下的某个可执行程序，如下是代码示例：

```
String[] command = { "C:\\tomcat\\bin\\catalina.bat", "start" };
String[] envs = { "CATALINA_HOME=C:\\tomcat", "JAVA_HOME=C:\\jdk5\\jdk5" };
File directory = "C:\\MyDirectory";
ProcessTask task = new ProcessTask(command, envs, directory);
cron4jPlugin.addTask(task);
me.add(cron4jPlugin);
```

如上所示，只需要创建一个 ProcessTask 对象，并让其指向某个应用程序，再通过 addTask 添加进来，就可以实现对其的调度，这种方式实现类似于每天半夜备份服务器数据库并打包成 zip 的功能，变得极为简单便捷。更加详细的用法，可以看一下 Cron4jPlugin.java 源代码中的注释。

第十章 Validator

10.1 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。

10.2 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。

以上代码 handleError 方法中的 keepXxx 方法用于将页面表单中的数据保持住并传递回页，以便于用户无需再重复输入已经通过验证的表单域，如果传递过来的是 model 对象，可以使用 keepModel 方法来保持住用户输入过的数据。

10.3 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)    // 配置方式与拦截器完全一样  
    public void login() {  
    }  
}
```

第十一章 国际化

11.1 概述

JFinal 为国际化提供了极速化的支持，国际化模块仅三个类文件，使用方式要比 spring 这类框架容易得多。

11.2 I18n 与 Res

I18n 对象可通过资源文件的 `baseName` 与 `locale` 参数获取到与之相对应的 Res 对象，Res 对象提供了 API 用来获取国际化数据。

以下给出具体使用步骤：

- 创建 `i18n_en_US.properties`、`i18n_zh_CN.properties` 资源文件，`i18n` 即为资源文件的 `baseName`，可以是任意名称，在此示例中使用“`i18n`”作为 `baseName`
- `i18n_en_US.properties` 文件中添加如下内容
`msg=Hello {0}, today is{1}.`
- `i18n_zh_CN.properties` 文件中添加如下内容
`msg=你好{0}, 今天是{1}.`
- 在 `YourJFinalConfig` 中使用 `me.setI18nDefaultBaseName("i18n")` 配置资源文件默认 `baseName`
- **特别注意**，java 国际化规范要求 `properties` 文件的编辑需要使用专用的编辑器，否则会出乱码，常用的有 Properties Editor，在此可以下载：<http://www.oschina.net/p/properties+editor>

以下是基于以上步骤以后的代码示例：

```
// 通过locale参数en_US得到对应的Res对象
Res resEn = I18n.use("en_US");
// 直接获取数据
String msgEn = resEn.get("msg");
// 获取数据并使用参数格式化
String msgEnFormat = resEn.format("msg", "james", new Date());

// 通过locale参数zh_CN得到对应的Res对象
Res resZh = I18n.use("zh_CN");
// 直接获取数据
String msgZh = resZh.get("msg");
// 获取数据并使用参数格式化
String msgZhFormat = resZh.format("msg", "詹波", new Date());

// 另外, I18n还可以加载未使用me.setI18nDefaultBaseName()配置过的资源文件, 唯一的不同是
// 需要指定baseName参数, 下面例子需要先创建otherRes_en_US.properties文件
Res otherRes = I18n.use("otherRes", "en_US");
otherRes.get("msg");
```

11.3 I18nInterceptor

I18nInterceptor 拦截器是针对 web 应用提供的一个国际化组件, 以下是在 freemarker 模板中使用的例子:

```
//先将I18nInterceptor配置成全局拦截器
public void configInterceptor(Interceptors me) {
    me.add(new I18nInterceptor());
}

// 然后在freemarker中即可通过_res对象来获取国际化数据
${_res.get("msg")}
```

以上代码通过配置了 I18nInterceptor 拦截 action 请求, 然后即可在 freemarker 模板文件中通过名为_res 对象来获取国际化数据, I18nInterceptor 的具体工作流程如下:

- 试图从请求中通过 controller.getPara(“_locale”)获取 locale 参数, 如果获取到则将其保存到

cookie 之中

- 如果 `controller.getPara("_locale")` 没有获取到参数值，则试图通过 `controller.getCookie("_locale")` 得到 locale 参数
- 如果以上两步仍然没有获取到 locale 参数值，则使用 `I18n.defaultLocale` 的值做为 locale 值来使用
- 使用前面三步中得到的 locale 值，通过 `I18n.use(locale)` 得到 Res 对象，并通过 `controller.setAttr("_res", res)` 将 Res 对象传递给页面使用
- 如果 `I18nInterceptor.isSwitchView` 为 true 值的话还会改变 render 的 view 值，实现整体模板文件的切换，详情可查看源码。

以上步骤 `I18nInterceptor` 中的变量名 `"_locale"`、`"_res"` 都可以在创建 `I18nInterceptor` 对象时进行指定，不指定时将使用默认值。还可以通过继承 `I18nInterceptor` 并且覆盖 `getLocalPara`、`getResName`、`getBaseName` 来定制更加个性化的功能。

在有些 web 系统中，页面需要国际化的文本过多，并且 css 以及 html 也因为国际化而大不相同，对于这种应用场景先直接制做多套同名称的国际化视图，并将这些视图以 locale 为子目录分类存放，最后使用 `I18nInterceptor` 拦截器根据 locale 动态切换视图，而不必对视图中的文本逐个进行国际化切换，只需将 `I18nInterceptor.isSwitchView` 设置为 true 即可，省时省力。

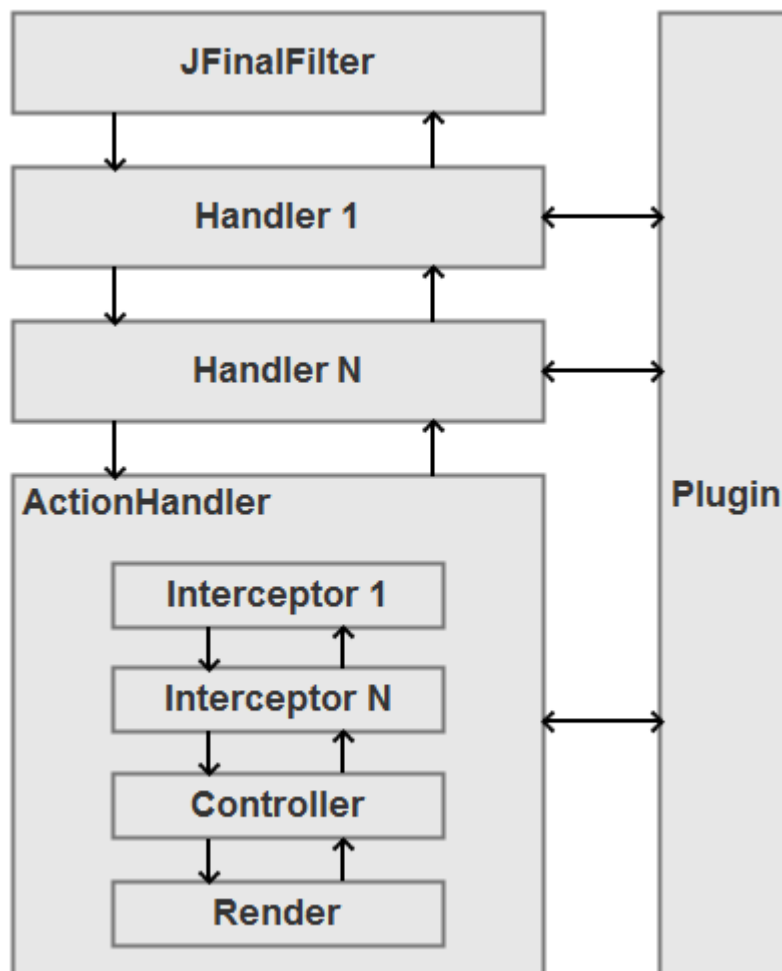
第十二章 JFinal 架构及扩展

12.1 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。

12.2 架构

JFinal 顶层架构图如下：



未完待续

JFinal 官方 QQ 群: 540853725、576124753

JFinal 官方微信:



强烈建议加入 JFinal 俱乐部，获取 JFinal 最佳实践项目源代码 jfinal-club，以最快的速度、最轻松的方式掌握最简洁的用法，省去看文档的时间：<http://www.jfinal.com/club>