1.1 Maven 基础

开发 jfinal 项目建议使用 maven，而不是使用传统手工的方式去管理 jar 包和构建项目。由于 maven 应用十分广泛，互联网已经有很多 maven 方面的资源，所以本小节只介绍 maven 使用的最基础的几个小点，了解这几个点上手使用 jfinal 已经够用。

1、下载

进入 maven 官网下载页面: http://maven.apache.org/download.cgi 点击 apache-maven-3.6.0-bin.zip 下载。建议最低下载 3.5.0 版本，高版本更加稳定。

2、安装

将 maven 解压到某个目录中，配置一下环境变量即完成安装，环境变量的配置与 JDK 的配置方式类似，配置两个环境变量即可。以下是 linux 系统下的配置示例：

```
export MAVEN_HOME=/Users/zhanbo/app/maven-3.5.2
export PATH=$PATH:$JAVA_HOME/bin:$MAVEN_HOME/bin
```

如上所示，将上述两行代码放在 /etc/profile 或者 ~/.bash_profile 文件之中即完成了 maven 的安装。windows 系统的环境变量配置参考这里：传送门

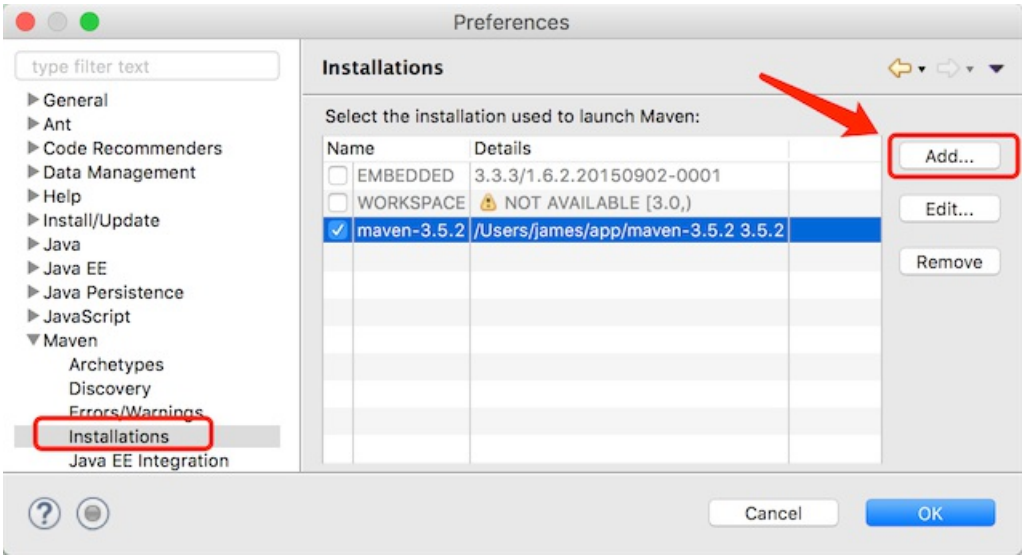
最后，打开控制台输入如下命令检查 maven 是否安装成功，安装成功会显示 maven 版本号：

```
mvn -v
```

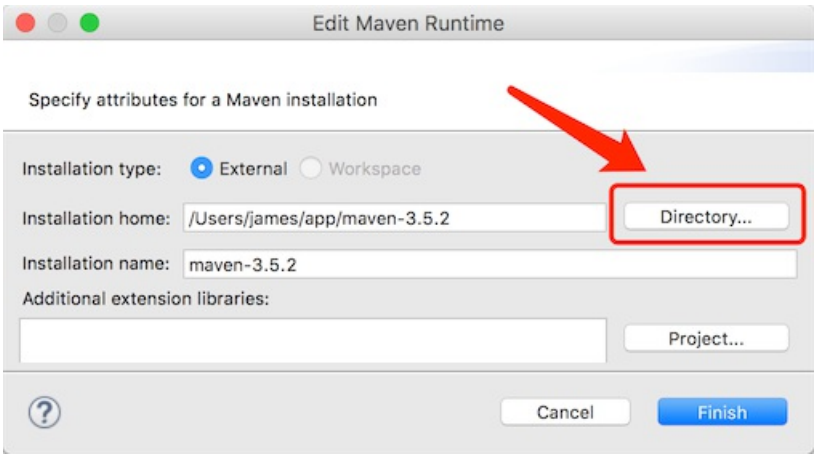
3、配置 eclipse 指向 maven

eclipse 本身自带一个嵌入的 maven，但嵌入的 maven 使用并不可靠，也不方便，例如在控制台无法使用 maven 的命令行进行操作。所以一定不要使用 eclipse 嵌入的 maven。以下以简要介绍一下配置方式

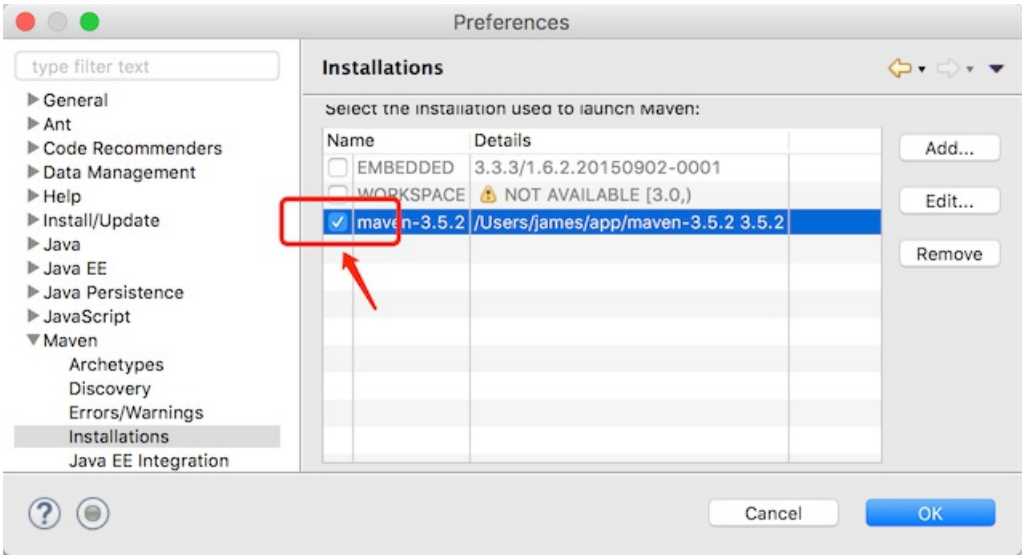
打开配置主窗口，点击左侧的 Maven 下的 Installatioins 子菜单



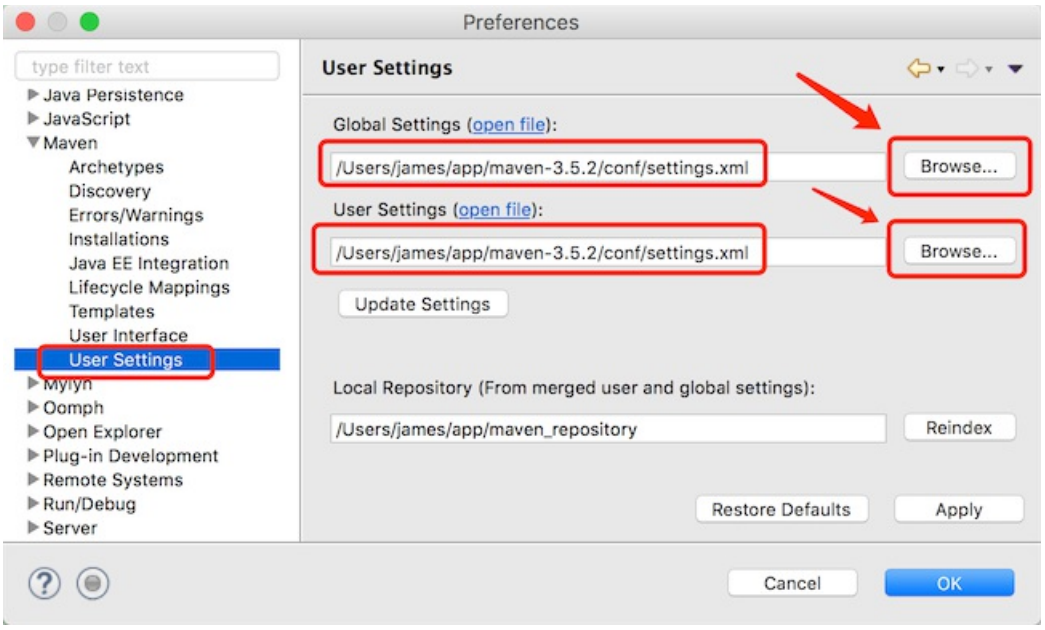
点击上图中的 add 按钮会弹出下面的窗口：



点击上图中的 Directory 选择 maven 解压到的那个目录，勾选刚刚添加的 maven，并去掉其它两个 maven 的勾选项，只保留刚刚安装的那个勾选



最后再点击左侧的 User Settings 菜单，然后分别点击右边两个的 Browe 按钮，为其配置好两个 settings.xml 文件即可，这两个文件在 maven 安装目录的 conf 子目录之下



点击 ok 按钮完成配置

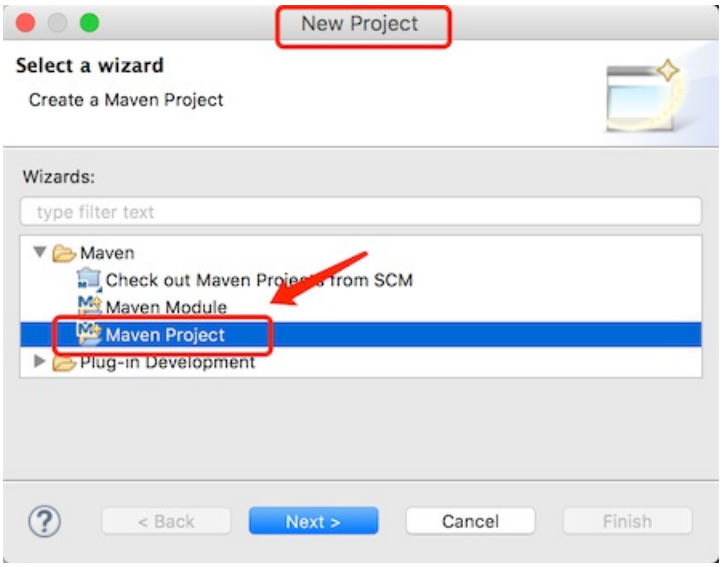
1.2 jfinal-undertow 下开发 >

1.2 jfinal-undertow 下开发

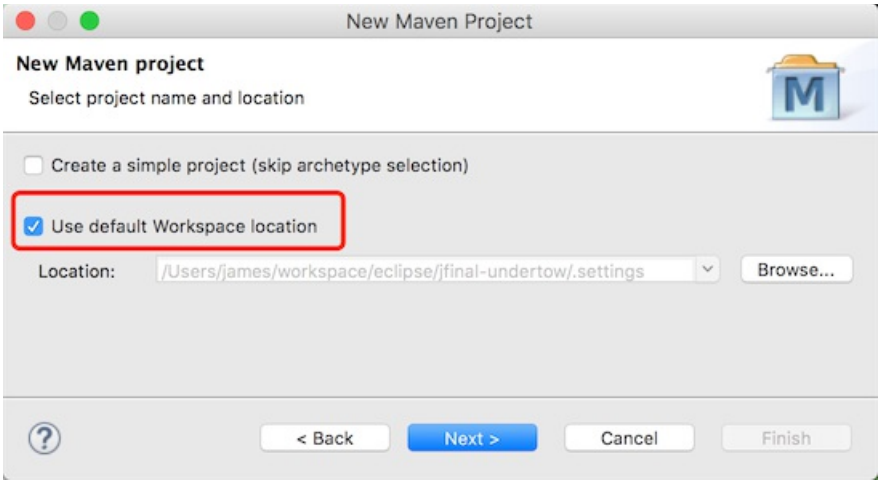
1、创建标准的 maven 项目

eclipse 主菜单选择 new，再选择 project

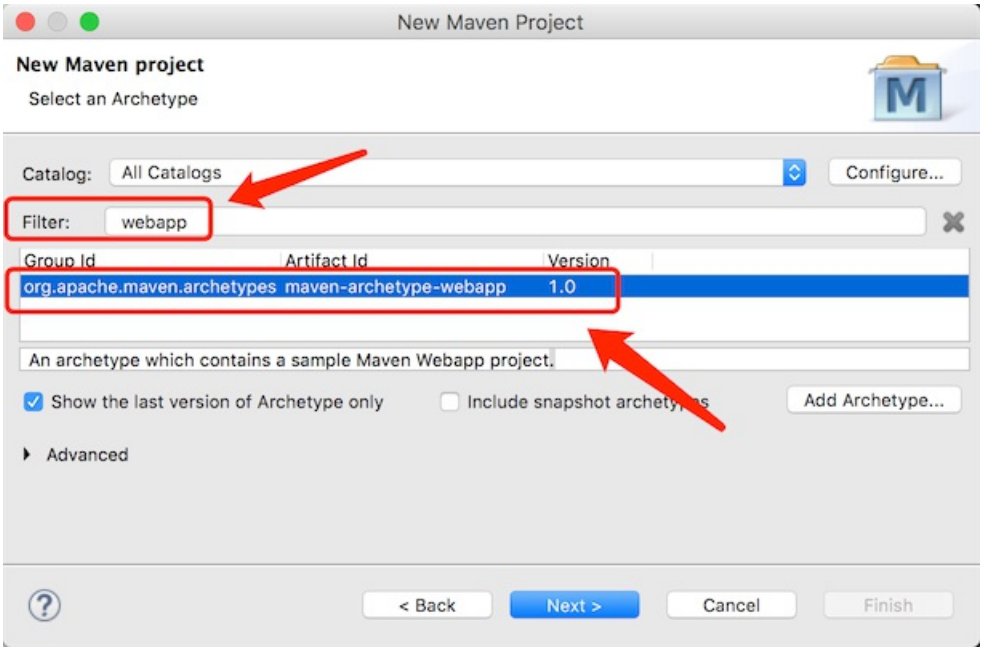
在弹出的窗口中选择 Maven Project，点击 next 按钮进入下一步



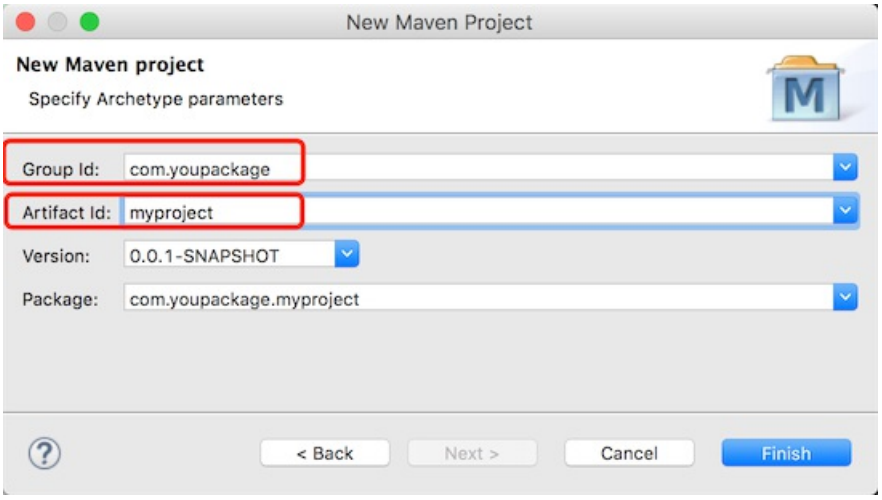
在接下来的窗口中勾选 Use default Wrokspace location 点击 next 进入下一步



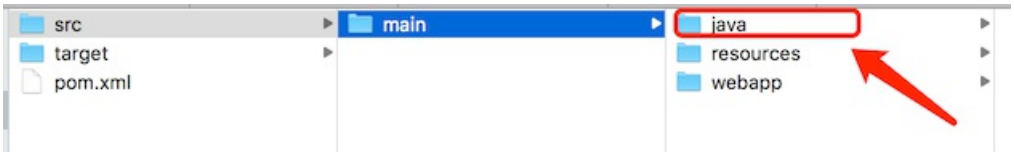
在接下来的窗口中的 Filter 栏输入 "webapp"，选择 org.apache.maven.archetypes maven-archetype-webapp 1.0，点击 next 进入下一步



在接下来的窗口中输入 Group Id 以及 Artifact Id，点击 finish 完成项目的创建



创建创建完成后，最终的目录结构如下



注意：在某些低版本的 eclipse 中，项目创建完成后，在 main 目录下面会**缺少 "java" 这个目录**，只需要在资源管理器里面手动创建该目录即可

2、添加 jfinal-undertow 依赖

打开 pom.xml 文件，在其中添加如下依赖

```
<dependency>
    <groupId>com.jfinal</groupId>
    <artifactId>jfinal-undertow</artifactId>
    <version>1.5</version>
</dependency>
```

如果需要 WebSocket 支持，再添加一个依赖，不开发 WebSocket 无需理会

```
<dependency>
    <groupId>io.undertow</groupId>
    <artifactId>undertow-websockets-jsr</artifactId>
    <version>2.0.17.Final</version>
</dependency>
```

3、添加 java 文件

在项目 src/main/java 目录下创建demo包，并在 demo 包下创建 DemoConfig 文件

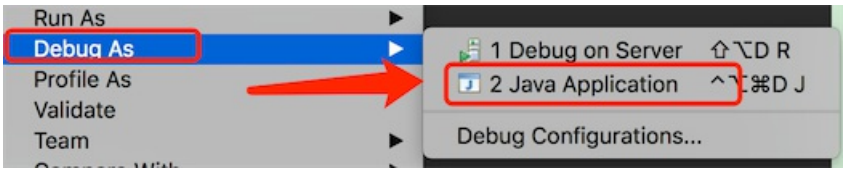
```
package demo;
import com.jfinal.config.*;

public class DemoConfig extends JFinalConfig {

    public static void main(String[] args) {
        UndertowServer.start(DemoConfig.class, 80, true);
    }

    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) {}
}
```

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.");
    }
}
```

4、启动项目

在 DemoConfig 类文件上点击鼠标右键，选择 Debug As，再选择 Java Applidation 即可运行



5、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

完整 demo示例 可在JFinal官方网站首页右侧下载：http://www.jfinal.com

< 1.1 Maven 基础
1.3 jfinal-undertow 下部署 >

1.3 jfinal-undertow 下部署

1、指定打包为类型为 jar

修改 pom.xml 文件，其中的 packaging 标签值要改成 jar

```
<packaging>jar</packaging>
```

2、添加 maven-jar-plugin 插件

```
<!--
 jar 包中的配置文件优先级高于 config 目录下的 "同名文件"
 因此，打包时需要排除掉 jar 包中来自 src/main/resources 目录的
 配置文件，否则部署时 config 目录中的同名配置文件不会生效
 -->
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-jar-plugin</artifactId>
    <version>2.6</version>
    <configuration>
        <excludes>
            <exclude>*.txt</exclude>
            <exclude>*.xml</exclude>
            <exclude>*.properties</exclude>
        </excludes>
    </configuration>
</plugin>
```

maven-jar-plugin 仅为了避免将配置文件打入 jar 包，如果是打成 fatjar 包则不需要添加此插件

3、添加 maven-assembly-plugin 插件

修改 pom.xml，在其中的 plugins 标签下面添加如下 maven-assembly-plugin 插件

```
<!--
    使用 mvn clean package 打包
    更多配置可参考官司方文档：http://maven.apache.org/plugins/maven-assembly-plugin/single-mojo.html
-->
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-assembly-plugin</artifactId>
  <version>3.1.0</version>
  <executions>
    <execution>
    <id>make-assembly</id>
    <phase>package</phase>
    <goals>
      <goal>single</goal>
    </goals>

    <configuration>
      <!-- 打包生成的文件名 -->
      <finalName>${project.artifactId}</finalName>
      <!-- jar 等压缩文件在被打包进入 zip、tar.gz 时是否压缩，设置为 false 可加快打包速度 -->
      <recompressZippedFiles>false</recompressZippedFiles>
      <!-- 打包生成的文件是否要追加 release.xml 中定义的 id 值 -->
      <appendAssemblyId>true</appendAssemblyId>
      <!-- 指向打包描述文件 package.xml -->
      <descriptors>
        <descriptor>package.xml</descriptor>
      </descriptors>
      <!-- 打包结果输出的基础目录 -->
      <outputDirectory>${project.build.directory}/</outputDirectory>
      </configuration>
      </execution>
    </executions>
```

```
</plugin>
```

maven-assembly-plugin 是 maven 官方提供的打包插件，功能十分完善，可以配置很多参数进行定制化构建，更详细的文件参考其官方文档：http://maven.apache.org/plugins/maven-assembly-plugin/single-mojo.html

4、添加 package.xml 文件

在项目根目录下面添加 package.xml，该文件是在上述 maven-assembly-plugin 在 descriptor 标签中指定的打包描述文件，内容如下：

```
<assembly xmlns="http://maven.apache.org/ASSEMBLY/2.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/ASSEMBLY/2.0.0 http://maven.apache.org/xsd/assembly-2.0.0.xsd">

  <!--
    assembly 打包配置更多配置可参考官司方文档：
    http://maven.apache.org/plugins/maven-assembly-plugin/assembly.html
  -->

  <id>release</id>

  <!--
      设置打包格式，可同时设置多种格式，常用格式有：dir、zip、tar、tar.gz
      dir 格式便于在本地测试打包结果
      zip 格式便于 windows 系统下解压运行
      tar、tar.gz 格式便于 linux 系统下解压运行
  -->
  <formats>
    <format>dir</format>
    <format>zip</format>
    <!-- <format>tar.gz</format> -->
  </formats>

  <!-- 打 zip 设置为 true 时，会在 zip 包中生成一个根目录，打 dir 时设置为 false 少层目录 -->
  <includeBaseDirectory>true</includeBaseDirectory>

  <fileSets>
    <!-- src/main/resources 全部 copy 到 config 目录下 -->
    <fileSet>
      <directory>${basedir}/src/main/resources</directory>
      <outputDirectory>config</outputDirectory>
    </fileSet>

    <!-- src/main/webapp 全部 copy 到 webapp 目录下 -->
    <fileSet>
      <directory>${basedir}/src/main/webapp</directory>
      <outputDirectory>webapp</outputDirectory>
    </fileSet>

    <!-- 项目根下面的脚本文件 copy 到根目录下 -->
    <fileSet>
      <directory>${basedir}</directory>
      <outputDirectory></outputDirectory>
      <!-- 脚本文件在 linux 下的权限设为 755，无需 chmod 可直接运行 -->
      <fileMode>755</fileMode>
      <includes>
        <include>*.sh</include>
        <include>*.bat</include>
      </includes>
    </fileSet>
  </fileSets>

  <!-- 依赖的 jar 包 copy 到 lib 目录下 -->
  <dependencySets>
    <dependencySet>
      <outputDirectory>lib</outputDirectory>
    </dependencySet>
  </dependencySets>

</assembly>
```

打包描述文件是 maven-assembly-plugin 的一部分，描述文件可以非常方便地控制打包的各种细节动作，更详细的文档参考：http://maven.apache.org/plugins/maven-assembly-plugin/assembly.html

5、在项目根目录下面添加启动脚本

注意：以下三个脚本文件在 jfinal 官网首页右侧下载的 jfinal demo 项目中已经提供，建议下载后 copy 到自己的项目中使用

windows 下面的启动脚本 start.bat 内容如下：

```
@echo off

rem ------------------------------------------------------------------
rem
rem 使用说明：
rem
rem 1: 该脚本用于别的项目时只需要修改 MAIN_CLASS 即可运行
rem
rem 2: JAVA_OPTS 可通过 -D 传入 undertow.port 与 undertow.host 这类参数覆盖
rem    配置文件中的相同值此外还有 undertow.resourcePath, undertow.ioThreads
rem    undertow.workerThreads 共五个参数可通过 -D 进行传入
rem
rem 3: JAVA_OPTS 可传入标准的 java 命令行参数,例如 -Xms256m -Xmx1024m 这类常用参数
rem
rem
rem ------------------------------------------------------------------

setlocal & pushd


rem 启动入口类,该脚本文件用于别的项目时要改这里
set MAIN_CLASS=com.jfinal.club.common.JFinalClubConfig

rem Java 命令行参数,根据需要开启下面的配置,改成自己需要的,注意等号前后不能有空格
rem set "JAVA_OPTS=-Xms256m -Xmx1024m -Dundertow.port=80 -Dundertow.host=0.0.0.0"
rem set "JAVA_OPTS=-Dundertow.port=80 -Dundertow.host=0.0.0.0"

set APP_BASE_PATH=%~dp0
set CP=%APP_BASE_PATH%config;%APP_BASE_PATH%lib\*
java -Xverify:none %JAVA_OPTS% -cp %CP% %MAIN_CLASS%


endlocal & popd
pause
```

如果项目的入口 main 方法有变动，修改一下上面内容中的 MAIN_CLASS 变量值即可，以下的 Linux 脚本也同样如此

Linux 下的启动脚本 start.sh 内容如下：

```
#!/bin/bash

# ------------------------------------------------------------------
#
# 使用说明：
#
# 1: 该脚本用于别的项目时只需要修改 MAIN_CLASS 即可运行
#
# 2: JAVA_OPTS 可通过 -D 传入 undertow.port 与 undertow.host 这类参数覆盖
#    配置文件中的相同值此外还有 undertow.resourcePath、undertow.ioThreads、
#    undertow.workerThreads 共五个参数可通过 -D 进行传入，该功能尽可能减少了
#    修改 undertow 配置文件的必要性
#
# 3: JAVA_OPTS 可传入标准的 java 命令行参数，例如 -Xms256m -Xmx1024m 这类常用参数
#
# 4: 脚本最后一部分给出了 4 种启动项目的命令行，根据注释中的提示自行选择合适的方式
#
# ------------------------------------------------------------------

# 启动入口类，该脚本文件用于别的项目时要改这里
```

```
MAIN_CLASS=com.jfinal.club.common.JFinalClubConfig

# Java 命令行参数，根据需要开启下面的配置，改成自己需要的，注意等号前后不能有空格
# JAVA_OPTS="-Xms256m -Xmx1024m -Dundertow.port=80 -Dundertow.host=0.0.0.0"
# JAVA_OPTS="-Dundertow.port=80 -Dundertow.host=0.0.0.0"

# 生成 class path 值
APP_BASE_PATH=$(cd `dirname $0`; pwd)
CP=${APP_BASE_PATH}/config:${APP_BASE_PATH}/lib/*

# 运行为后台进程，并在控制台输出信息
java -Xverify:none ${JAVA_OPTS} -cp ${CP} ${MAIN_CLASS} &

# 运行为后台进程，并且不在控制台输出信息
# nohup java -Xverify:none ${JAVA_OPTS} -cp ${CP} ${MAIN_CLASS} >/dev/null 2>&1 &

# 运行为后台进程，并且将信息输出到 output.log 文件
# nohup java -Xverify:none ${JAVA_OPTS} -cp ${CP} ${MAIN_CLASS} > output.log &

# 运行为非后台进程，多用于开发阶段，快捷键 ctrl + c 可停止服务
# java -Xverify:none ${JAVA_OPTS} -cp ${CP} ${MAIN_CLASS}
```

Linux 下的服务停止脚本 stop.sh 内容如下：

```
#!/bin/bash

# ----------------------------------------------------------------
#
# 使用说明：
#
# 1：MAIN_CLASS 必须要与配对的 start.sh 文件中的 MAIN_CLASS 完全相同
#
# 2：该脚本用于别的项目时只需要修改 MAIN_CLASS 即可使用
#
# 3：注意：如果有其它项目的 MAIN_CLASS 与本项目一样则不能使用本脚本关闭服务
#          同理同一个项目使用了不同端口启动的，也会拥有相同的 MAIN_CLASS 值
#          也不能使用本脚本关闭服务，这种情况使用下面的命令先查 pid 值：
#             ps aux | grep java
#
#          确认好 pid 以后，使用 kill pid 关闭服务
#
#          注意 kill 命令不要带 -9 这个参数，否则 jfinal 中的一些与服务关闭
#          有关的回调方法将不会被回调，例如 JFinalConfig.onStop()
#
# 4：如果不需要上述的 onStop() 回调，使用 kill -9 可加快关闭服务的速度
#
# ----------------------------------------------------------------

# 启动入口类，该脚本文件用于别的项目时要改这里
MAIN_CLASS=com.jfinal.club.common.JFinalClubConfig

# kill 命令不使用 -9 参数时，会回调 onStop() 方法，确定不需要此回调建议使用 -9 参数
kill `pgrep -f ${MAIN_CLASS}` 2>/dev/null

# 以下代码与上述代码等价
# kill $(pgrep -f ${MAIN_CLASS}) 2>/dev/null
```

如上述过程出现任何问题，请下载官网首页的 jfinal demo，里面有所有配置与脚本，可直接拿来使用

特别注意：linux、mac 下的脚本文件换行字符必须是 '\n'，而 windows 下必须是 "\r\n"，否则脚本无法执行，并会输出无法理解的错误信息，难以排错。如何查看脚本文件中的换行字符见文档：https://www.jfinal.comhttps://www.jfinal.com/doc/1-5

6、打包

打开命令行终端，cd 命令进入项目根目录，运行以下命令即可打包

```
mvn clean package
```

执行上述打包命令以后，在项目根下面的 target 目录下面会出现打好的 xxx.zip，解压该 zip 文件运行里面的 start 脚本即可

除了 zip 文件还会在 target 下面生成一个目录，运行该目录下面的 start 也可以运行项目，该目录下面的内容与 zip 文件中的内容是完全一样的。

7、部署

将上述打包命令生成的 zip 文件上传到服务器并解压即完成了部署工作，基于 jfinal-undertow 开发项目的最大优势就是不需要下载、安装、配置 tomcat 这类 server

8、fatjar 打包部署

fatjar 打包是指将项目中所有 class、所以资源以及所有 jar 包依赖全部打包到一个单独的 jar 包之中，打包好的独立 jar 包可以很方便复制、部署、运行，非常适合于做微服务项目开发，也很适合没有 web 资源或者 web 资源很少的项目

具体的用法可以在官网首页下载 jfinal-demo-for-maven，其中的 doc 目录下面有文档详细介绍了此方法，后续会在文档频道添加 fatjar 打包方法

9、jfinal-undertow 的主要优势

1：极速启动，启动速度比 tomcat 快 5 到 8 倍。jfinal.com 官网启动时间在 1.5 秒内

2：极简精妙的热部署设计，实现极速轻量级热部署，响应极为迅速，让开发体验再次提升一个档次

3：性能比 tomcat、jetty 高出很多，可代替 tomcat、jetty 用于生产环境

4：undertow 为嵌入式而生，可直接用于生产环境部署，部署时无需下载服务，无需配置服务，极其适合微服务开发、部署

5：告别 web.xml、告别 tomcat、告别 jetty，节省大量打包与部署时间。令开发、打包、部署成为一件开心的事

6：功能丰富，支持 classHotSwap、WebSocket、gzip 压缩、servlet、filter、sessionHotSwap 等功能

7：支持 fatjar 与 非 fatjar 打包模式，为 jfinal 下一步的微服务功能做好准备

8：开发、打包、部署一体化，整个过程无需对项目中的任何地方进行调整或修改，真正实现从极速开发到极速部署

9：以上仅为 jfinal-undertow 的部分功能，更多好用的功能如 fatjar 打包模式见 jfinal 官网文档

< 1.2 jfinal-undertow 下开发
1.4 jfinal-undertow 高级用法 >

1.4 jfinal-undertow 高级用法

一、基础配置

1、启用配置文件

在 src/main/resources 目录下面创建 undertow.txt 文件，该文件会被 jfinal undertow 自动加载并对 jfinal undertow 进行配置。

如果不想使用 "undertow.txt" 这个文件名，还可以通过 UndertowServer.create(AppConfig.class, "other.txt") 方法的第二个参数来指定自己喜欢的文件名。

配置文件创建好以后，就可以按照下面的文档来配置相应的功能了。

2、常用配置

```
# true 值支持热加载
undertow.devMode=true
undertow.port=80
undertow.host=0.0.0.0
```

开发时配置成 true 支持热加载，注意该 devMode 与 jfinal 项目中的 devMode 没有任何关系，注意区分。

3、web 资源加载路径配置

jfinal undertow 可以十分方便地从文件系统的目录以及 class path 或 jar 包中加载 web 静态资源，以下是配置示例：

```
undertow.resourcePath = src/main/webapp, classpath:static
```

如上所示 "src/main/webapp" 表示从项目根目录下的 "src/main/webapp" 下去加载 web 静态资源。 "classpath:static" 表示从 class path 以及 jar 包中的 static 路径下去加载 web 静态资源。

注意："classpath:static" 这种配置是 jfinal undertow 1.5 才添加的功能。

undertow.resourcePath 配置的另一个重点是，以 "classpath:" 为前缀的配置需要自行注意路径是否存在，尽可能只配置存在的路径。而不以 "classpath:" 打头的配置可以将开发与部署时的路径一起配置进来（逗号分隔开），jfinal undertow 会在运行时检测路径是否存在，存在才真正让其生效，从而很方便一次配置同时适用于开发、生产两种环境。

4、性能配置

```
# io 线程数与 worker 线程数
# undertow.ioThreads=
# undertow.workerThreads=
```

默认配置已充分考虑常用场景，如果没有压测数据作为指导，建议使用默认配置，不要添加这两项配置。

ioThreads 为 NIO 处理 io 请求的线程数量，在生产环境下建议的配置范围是 cpu 核心数的一倍到两倍。建议根据压测结果进行调整。

workerThreads 是处理请求的线程数量，在生产环境下可以使用默认配置，或者根据压测结果进行配置。在压测数据几乎最好的情况下尽可能选择更小的 workThreads 值。因为当性能达到某个高度时再增加 workThreads 值并不会带来明显的性能提升，但会明显增加系统的资源消耗。

5、开启 gzip 压缩

```
# gzip 压缩开关
undertow.gzip.enable=true
# 配置压缩级别，默认值 -1。 可配置 1 到 9。 1 拥有最快压缩速度，9 拥有最高压缩率
undertow.gzip.level=-1
# 触发压缩的最小内容长度
```

```
undertow.gzip.minLength=1024
```

开启 gzip 压缩可以降低网络流量，提升访问速度

6、配置 session

```
# session 过期时间，注意单位是秒
undertow.session.timeout=1800
# 支持 session 热加载，避免依赖于 session 的登录型项目反复登录，默认值为 true。仅用于 devMode，生产环境无影响
undertow.session.hotSwap=true
```

7、配置 https

```
# 是否开启 ssl
undertow.ssl.enable=false
# ssl 监听端口号，部署环境设置为 443
undertow.ssl.port=443
# 密钥库类型，建议使用 PKCS12
undertow.ssl.keyStoreType=PKCS12
# 密钥库文件
undertow.ssl.keyStore=demo.pfx
# 密钥库密码
undertow.ssl.keyStorePassword=123456
# 别名配置，一般不使用
undertow.ssl.keyAlias=demo
```

SSL 证书的获取见后面的第二小节

8、配置 http2

```
# ssl 开启时，是否开启 http2。检测该配置是否生效在 chrome 地址栏中输入：chrome://net-internals/#http2
undertow.http2.enable=true
```

开启 http2 可以大大加快访问速度，不必担心 https 比 http 慢这个事

9、配置 http 重定向到 https

```
# ssl 开启时，http 请求是否重定向到 https
undertow.http.toHttps=false
# ssl 开启时，http 请求跳转到 https 使用的状态码，默认值 302
undertow.http.toHttpsStatusCode=302
```

10、配置关闭 http

```
# ssl 开启时，是否关闭 http
undertow.http.disable=false
```

在启用 https 后，可以配置关闭 http，这样就只能访问 https 了。该配置项比较适合于小程序服务端。

对于一般的 web 项目不建议配置此项，而是配置 undertow.http.toHttps=true 将 http 重定向到 https

11、自由配置 Undertow

以上配置方式由 jfinal undertow 做了直接的支持，如果上面的配置仍然不能满足需求，还可以通过下面的方式自由配置 undertow：

```
UndertowServer.create(JFinalConfig.class)
    .onStart( builder -> {
        builder.setServerOption(UndertowOptions.参数名, 参数值);
     })
    .start();
```

如上所示，通过在 onStart 方法中使用 builder.setServerOption(...) 可以对 underow 进行更加深入的配置，还可以调用 builder 中的其它 API 进行其它类型的配置。上述的 UndertowOptions 中定义了很多 undertow 的配置名，查看其中的注释

文档可以知道还有很多有用的配置

12、添加 Filter、WebSocket、Servlet、Listener

使用 UndertowServer.configWeb(...) 可以很方便添加 Filter、WebSocket、Servlet、Listener 这些标准的 Java Web 组件：

```
UndertowServer.create(AppConfig.class)
     .configWeb( builder -> {
         // 配置 Filter
         builder.addFilter("myFilter", "com.abc.MyFilter");
         builder.addFilterUrlMapping("myFilter", "/*");
         builder.addFilterInitParam("myFilter", "key", "value");

         // 配置 Servlet
         builder.addServlet("myServlet", "com.abc.MyServlet");
         builder.addServletMapping("myServlet", "*.do");
         builder.addServletInitParam("myServlet", "key", "value");

         // 配置 Listener
         builder.addListener("com.abc.MyListener");

         // 配置 WebSocket，MyWebSocket 需使用 ServerEndpoint 注解
         builder.addWebSocketEndpoint("com.abc.MyWebSocket");
      })
     .start();
```

以上代码中的 MyWebSocket 需要使用 ServerEndpoint 注解标识其为一个 WebSocket 组件，例如：

```
@ServerEndpoint("/myapp.ws")
public class MyWebSocket {
    @OnMessage
    public void message(String message, Session session) {
        for (Session s : session.getOpenSessions()) {
            s.getAsyncRemote().sendText(message);
        }
    }
}
```

与上述 MyWebSocket 配合使用的例子 html 在这里可以下载：传送门

要修改一下该 html 中的 url 为： "ws://localhost:80/myapp.ws"，端口号适当调整。

注意：由于 JFinalFilter 会接管所有不带 "." 字符的 URL 请求所以 @ServerEndpoint 注解中的 URL 参数值建议以 ".ws" 结尾，否则请求会响应 404 找不到资源，例如：

```
@ServerEndpoint("/myapp.ws")
public class MyWebSocketEndpoint  {
    ......
}
```

当然，ServerEndpoint 中的 URL 不使用 ".ws" 结尾也是可以的，只需要参考 jfinal 的 UrlSkipHandler 做一个 Handler 跳过属于属于 WebSocket 的 URL 即可

二、SSL 证书

1、申请 SSL 证书

建议从阿里云或者腾迅云获取 SSL 证书，有免费版和收费版，阿里云 SSL 证书获取：SSL 证书获取传送门

2、下载合适的证书类型

SSL 证书申请下来以后，可以到控制台下载证书，如下图所示：

点击下载链接以后，在右侧下载 tomcat 类型的 SSL 证书

下载的证书里头，有一个 xxx.pfx 文件，该文件就是证书文件。此外还有一个 pfx-password.txt 文件，此文件里面放的是证书密码，将 xxx.pfx 放入项目的 src/main/resources 目录，在 undertow.txt 配置文件中添加如下几行配置：

```
# 是否开启 ssl
undertow.ssl.enable=true
# ssl 监听端口号，部署环境设置为 443
undertow.ssl.port=443
# 密钥库类型，建议使用 PKCS12
undertow.ssl.keyStoreType=PKCS12
# 密钥库文件
undertow.ssl.keyStore=demo.pfx
# 密钥库密码
undertow.ssl.keyStorePassword=123456
```

其中的 "demo.pfx" 即为前面下载的证书文件的文件名，"123456" 即为证书密码，这两个配置根据你下载到的实际内容进行修改。"PKCS12" 这个配置是证书类型，阿里云下载的 tomcat 型证书为 PKCS12，腾迅云可能是 "JKS"

3、启动项目

启动项目，通过 "https://申请证书时绑定的域名" 即可访问

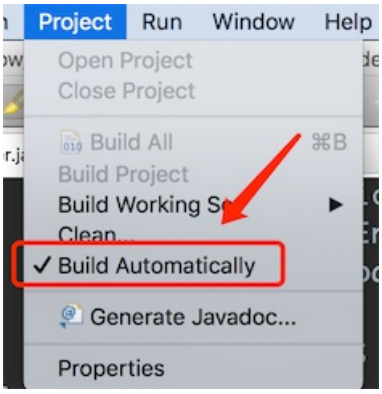
< 1.3 jfinal-undertow 下部署
1.5 jfinal-undertow 常见问题 >

1.5 jfinal-undertow 常见问题

1、IDEA 下支持热加载

jfinal undertow 是通过监听 target/classes 下面的 .class 文件被修改事件去触发的热加载。

eclipse 拥有自动编译功能，会在 java 源代码文件保存时自动编译并更新 target/classes 下的相关 .class 文件，所以 eclipse 下面默认就支持热加载。注意 eclipse 的自动编译菜单一定要勾选上(默认是勾选的)，如下图：



但 IDEA 默认并不支持自动编译，那么 target/classes 下面的 .class 文件无法及时更新，所以 IDEA 默认不支持热加载。因此需要想办法触发 IDEA 的编译动作，建议去网上找一些开启 IDEA 自动编译的配置方法。 下面的链接给出了一种通过快捷键触发编译的方法供参考：

https://my.oschina.net/fdblog/blog/172229

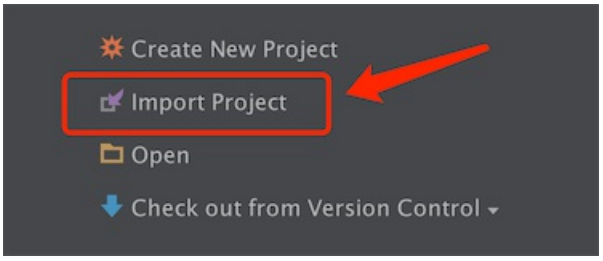
此外，jfinal 官网很久以前的文档也给出过一种方法：

https://www.jfinal.com/share/1357

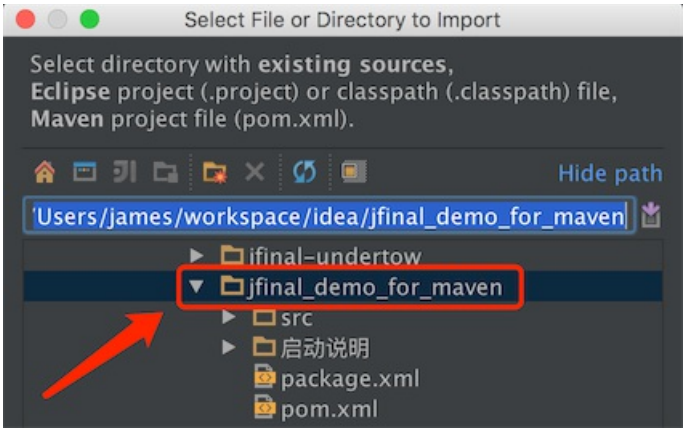
最后：别忘了 undertow.devMode 要配置成 true 才支持执加载，具体配置方法见前面两小节的文档。

2、IDEA 下模板文件路径不正确

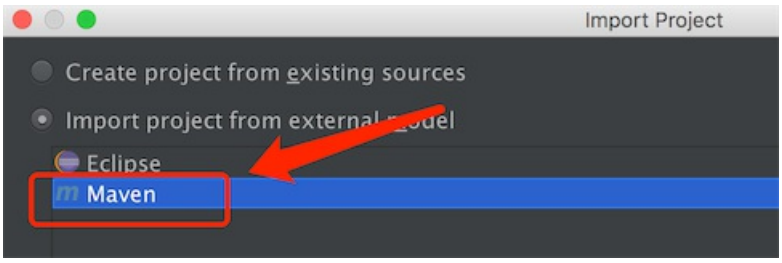
在 IDEA 下发时，如果是通过打开项目的目录，或者通过打开项目的 pom.xml 文件导入的 maven 项目将会找不到模板文件路径。正确的导入方法如下：



如下图所示，一定要选择 import project，而不能选择 Open，下一步是选择项目根目录：



再一步是选择 maven：



然后再一路点击 next 直到完成即可。

要点：如果第一步选择 Open 将是错误的做法。如果项目已经使用上述方式导入过一次，IDEA 生成了各种配置文件，那么选择 Open 打开项目的方式没有问题。

3、maven 多模块项目启动报错

如果 maven 的多 module 结构的项目在启动时出现找不到模板的异常，添加如下配置：

```
undertow.resourcePath=webapp, {修改为自己的项目名}/src/main/webapp, src/main/webapp
```

这里有相关问题的分享：http://www.jfinal.com/share/1285

4、类型转换异常、子类对象无法赋值到基类变量

如果两个类路径与类名完全一样的两个类出现类型转换异常，或者子类对象无法赋值给基类变量，可以通过配置 hotSwapClassPrefix 来解决。

假定出现转换异常的类为： "com.abc.UserService"，解决办法如下：

```
UndertowServer.create(MyApp.class)
    .addHotSwapClassPrefix("com.abc.")
    .start();
```

如上所示，将出现类型转换异常的类的包前缀通过 addHotSwapClassPrefix(...) 添加进去即可。原因是 jfinal undertow 默认只对 target/classes 以及 jfinal 自身进行热加载，所以当你的类文件在 jar 包并且需要被热加载的时候就需要通过上面的办法添加为被热加载。

一般情况下你项目中 target/classes 下的类文件正好才是需要被热加载的类，所以不会有问题。这个配置仅用于开发环境，部署环境没有热加载机制，所以完全没有影响。

此外，还可以通过配置文件来设置：

```
undertow.hotSwapClassPrefix=com.abc.
```

如果要添加多个，可以通过逗号分隔。

5、shiro 热加载问题

jfinal undertow 暂不支持 shiro 热加载，配置 undertow.devMode=false 可以使用，但不支持热加载

6、部署在外网服务器上无法访问问题

出于安全性考虑 jfinal undertow 的 undertow.host 默认配置为了 localhost，部分外网服务器上无法访问时，使用如下配置：

```
undertow.host=0.0.0.0
```

如果添加了上述配置还是无法访问，检查一下阿里云是否开启了相关的端口号（假定你用的阿里云）

7、脚本无法使用问题

jfinal 官方提供的 start.sh、stop.sh、restart.sh、start.bat 这几个脚本文件里面有较为详细的使用说明，只要按照说明去使用一般不会有问题。

这里要提醒一个比较奇葩的问题，假定你自己创建上述脚本文件，虽然是从 jfinal 官方 copy 过去的内容，然后死活就是无法使用，原因是脚本文件的换行符出了问题。 linux、mac 系统之下的脚本换行字符必须是 '\n'，而 windows 下必须是 "\r\n"。

查看脚本文件的换行字符的方法是：先用 eclipse 打开一个 java 源代码文件，然后在工具栏点击一下 "Show Whitespace Characters" 这个图标，然后再用 eclipse 打开脚本文件，在每行末尾处会显示出换行字符。显示一个字符的就是 '\n'，否则就是 '\r\n'。

注意，此问题与 jfinal 完全无关，是操作系统自己的限制。

重要：最近俱乐部同学发现一类新的脚本无法启动的原因，那就是 pom.xml 中的 jetty 依赖没有删除，从而引发异常：java.lang.NoClassDefFoundError: com/jfinal/config/JFinalConfig，删掉 jetty 依赖即可解决。

< 1.4 jfinal-undertow 高级用法
1.6 jetty-server 下开发 >

1.6 jetty-server 下开发

1、创建标准的 maven 项目

与 jfinal undertow 相关小节完全一样

2、添加 jetty server 与 jfinal 依赖

```
<dependency>
    <groupId>com.jfinal</groupId>
    <artifactId>jfinal</artifactId>
    <version>3.6</version>
</dependency>

<dependency>
    <groupId>com.jfinal</groupId>
    <artifactId>jetty-server</artifactId>
    <version>2019.1</version>
    <scope>provided</scope>
</dependency>

<!-- 下面的依赖仅在使用 JSP 时才需要 -->
<dependency>
    <groupId>org.eclipse.jetty</groupId>
    <artifactId>jetty-jsp</artifactId>
    <version>9.2.26.v20180806</version>
        <scope>provided</scope>
</dependency>
```

jetty-server 依赖的 scope 为 provided，仅用于开发阶段，部署时不需要，打包时也会自动跳过。

这里特别注意一下：如果是使用 IDEA 开发，scope 仍然得设置成为 compile，否则提示缺少 jar 包，在打包的时候记得要改回 provided，避免打进一些不需要的 jar 包

3、添加 java 文件

与 jfinal undertow 相关小节几乎一样，仅仅是 main 方法中的内容有所不同，如下：

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

4、启动项目

与 jfinal undertow 相关小节完全一样

5、开启浏览器看效果

与 jfinal undertow 相关小节完全一样

6、常见问题解决：

1：出现 NoClassDefFoundError 异常

原因之一是 maven 本地库下载的 jar 文件数据有错误。看一下异常中是哪个类文件抛出的 NoClassDefFoundError，在本地 maven 库中删掉其目录，让 maven 从中心库中重新下载一次即可，例如使用的 jfinal-3.6 版本，就删掉 maven repository 中的 /com/jfinal/jfinal 目录下面的 3.6 子目录。

原因之二是对同一个 jar 包，引入了多个不同版本，删掉其中多余的即可

以上问题本质上与 jfinal 无关，纯属 Java 开发时碰到的基础性异常

< 1.5 jfinal-undertow 常见问题
1.7 tomcat 下部署 >

1.7 tomcat 下部署

1、指定打包类型为 war

修改 pom.xml 文件，其中的 packaging 标签值要改成 war

```
<packaging>war</packaging>
```

2、删掉 jetty-server 依赖

修改 pom.xml 文件，将 jetty-server 有关依赖全部删除，否则部署到 tomcat 中会产生冲突

3、打包

控制台 cd 进入项目根目录执行下面命令打出 war 包

```
mvn clean package
```

4、部署

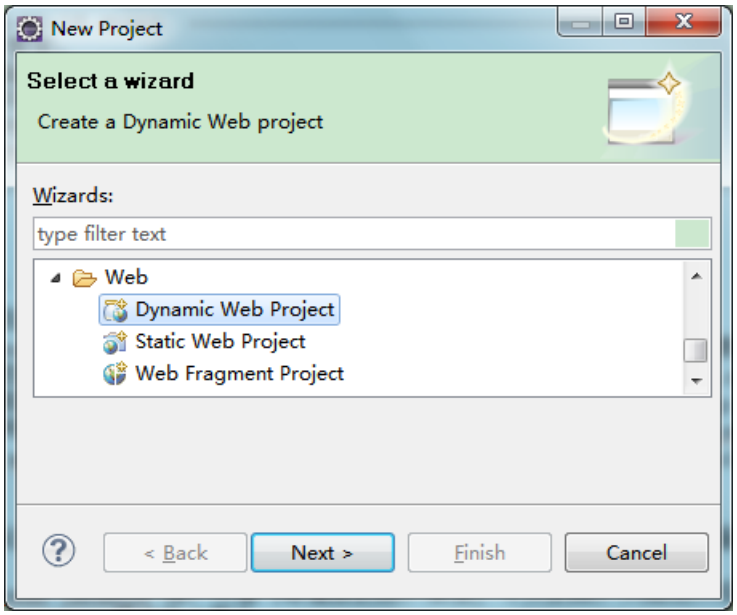
jfinal 开发的项目就是标准的 java web 项目，所以部署方式没有任何特殊的地方，有一些部署方面要注意的小技巧，见这篇博文：https://my.oschina.net/jfinal/blog/353062

< 1.6 jetty-server 下开发
1.8 非 maven 方式开发 >

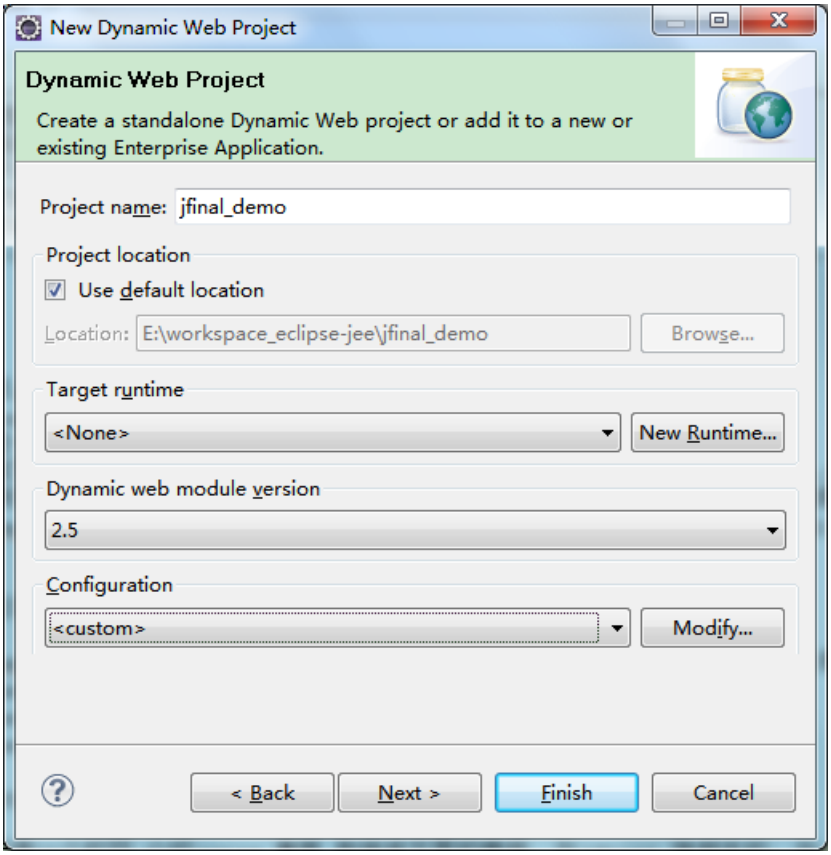
1.8 非 maven 方式开发

推荐使用Eclipse Mars这个版本，下载地址为：https://www.eclipse.org/mars/

1、创建Dynamic Web Project

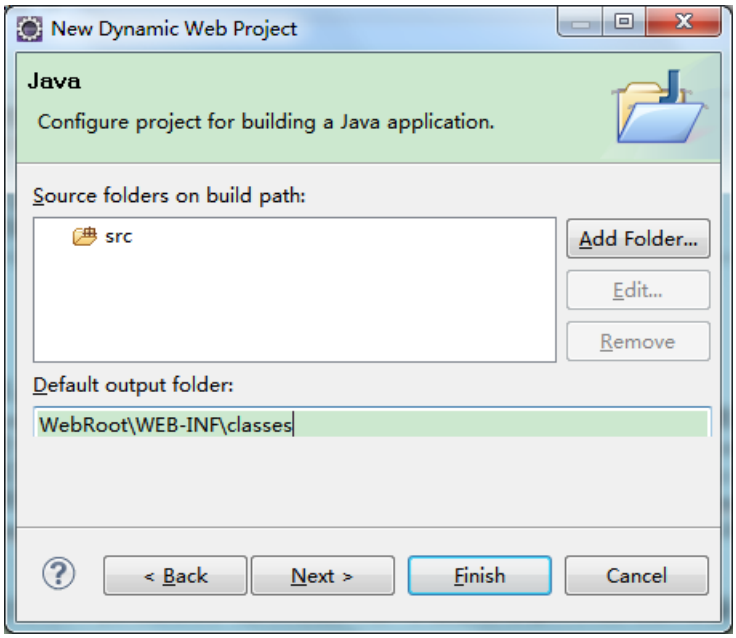


2、填入项目基本信息



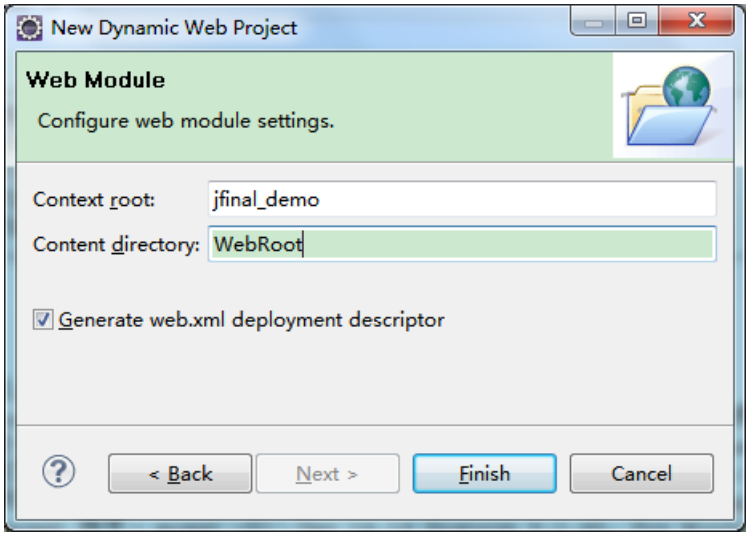
注意上图中：Target runtime 一定要选择<None>

3、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。

4、修改Content directory，推荐输入WebRoot



注意上图：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。

5、放入JFinal库文件

将 jfinal-3.6.jar 与 jetty-server-2019.1.jar 拷贝至项目WEB-INF\lib下即可。注意：jetty-server-2019.1.jar 是开发时使用的运行环境，生产环境不需要此文件。

所需要的 jar 包可以在 jfinal.com 首页下载 jfinal-3.6-all.zip 文件，该文件中包含了所需的常用 jar 包，以及 jar 包使用说

明。

6、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>
```

7、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) {}
}
```

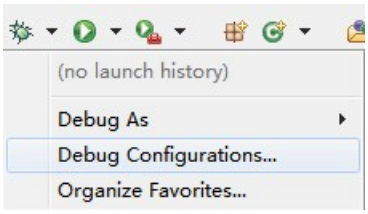
注意：DemoConfig.java文件所在的包以及自身文件名必须与web.xml中的param-value标签内的配置相一致(在本例中该配置为demo.DemoConfig)。

在demo包下创建HelloController类文件， 内容如下：

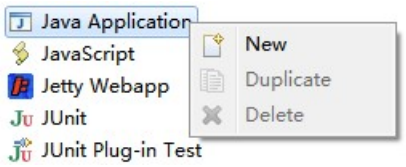
```
package demo;
import com.jfinal.core.Controller;
public class HelloController extends Controller {
    public void index() {
       renderText("Hello JFinal World.");
    }
}
```

8、启动项目

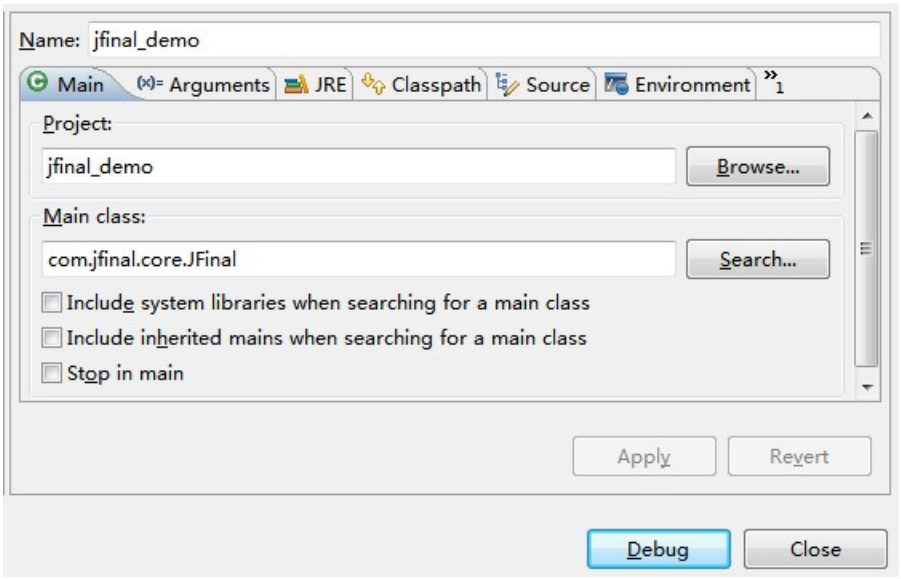
创建启动项如下图所示：



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) {
 // eclipse 下的启动方式
 JFinal.start("src/main/webapp", 80, "/", 5);
}
```

上面代码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。上面的第一个参数”src/main/webapp”适用于标准的maven项目，如果是非 maven 的老式 java web 项目，第一个参数通常是 ”WebRoot” 或 ”WebContent” 。

9、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。

强烈建议使用 maven 标准项目结构进行开发，本小节文档介绍的手动添加 jar 包的方式是比较古老的方式

< 1.7 tomcat 下部署
1.9 IDEA下开发 >

1.9 IDEA下开发

由于 jfinal-undertow 与 jetty-server 的热加载都是通过监控 class 文件是否被更新而触发的，但 IDEA 默认不支持自动编译，所以其 class 文件在开发过程中不会被更新。因此，在 IDEA 下默认不支持热加载，可以在网上找一找开启 IDEA 自动编译的配置来支持，下面给出几个这类资源：

https://my.oschina.net/fdblog/blog/172229

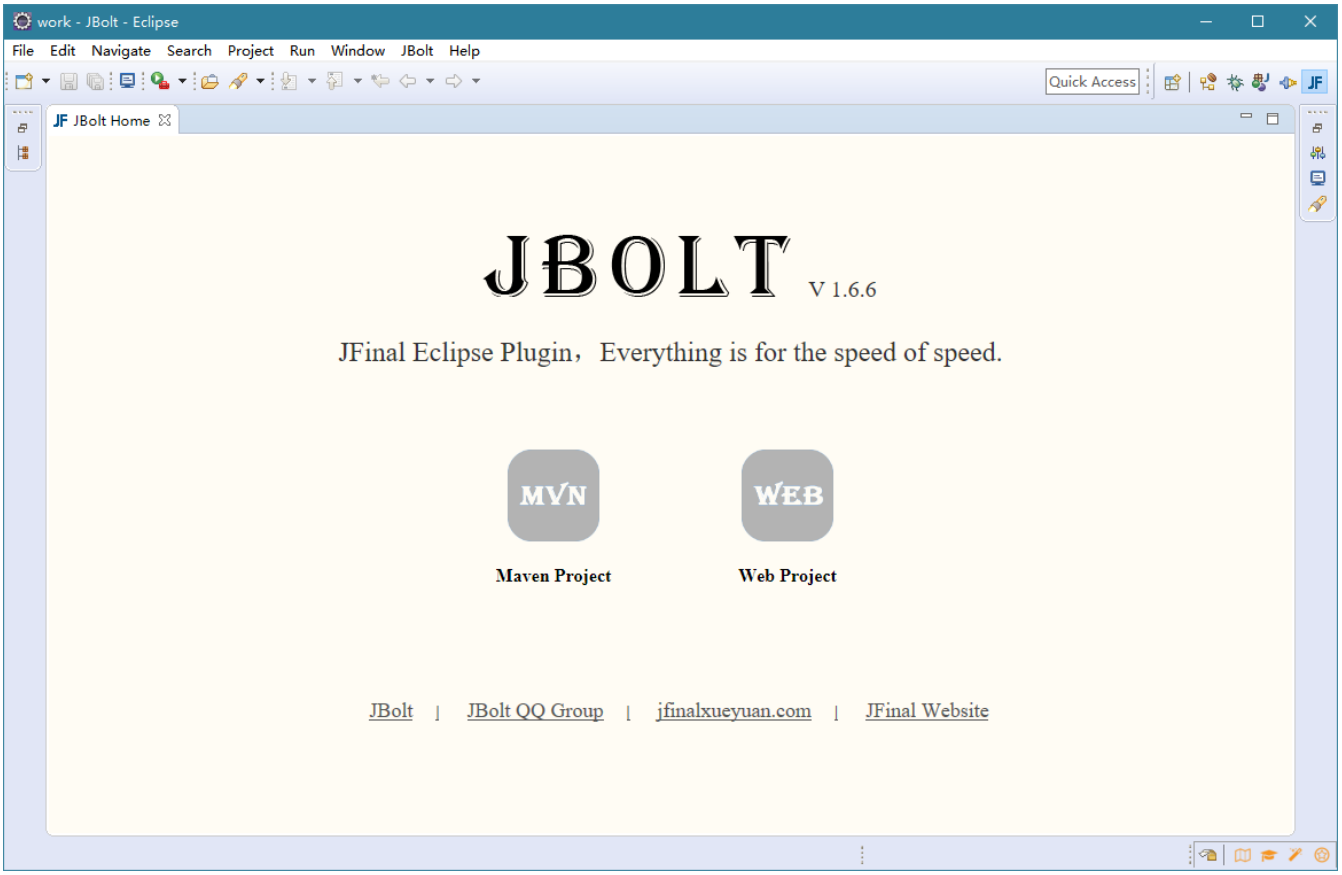
https://www.jfinal.com/share/1357

也可以使用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。

< 1.8 非 maven 方式开发
1.10 JBolt 插件下开发 >

1.10 JBolt 插件下开发

JBolt是专门为JFinal社区开发者定制的Eclipse开发环境下极速开发IDE插件，JFinal开发者必备利器。



JBolt官网：http://www.jbolt.cn

作者：山东小木

运行Eclipse平台版本：Eclipse Java EE 版本 4.6.3+（推荐下载离线包版Eclipse）

目前JBolt在Window、Linux桌面版、Mac上测试均可正常开发使用

安装使用教程（视频版）：点击进入观看视频教程

安装使用教程（图文版）：http://www.jfinal.com/share/1052

JBolt的能力：

总体来讲，JBolt为开发提供了可视化向导配置后一键生成开发环境基础项目代码、依赖类库自动导入、配置文件自动生成、生成后的项目可以一键启动访问的能力。

1、根据创建项目的向导配置信息，可以一键生成基于JFinal的项目开发环境，**支持普通动态Web工程和Maven工程的生成**，同时根据配置自动导入依赖的jar包、生成web.xml、数据库配置文件、jfinalConfig主配置类、IndexController.java、路由集合配置类、服务器配置信息等。

2、支持Mysql、Oracle、sql server、h2、sqlite、postgresql等主流数据库配置文件和Model、BaseModel代码的生成

3、JBolt提供了一个JFinal Enjoy 模板编辑器，可以同时编辑HTML JS CSS 和JFinal模板代码，模板指令提示和自动完成

4、JBolt提供了一个快速新增JFinal Controller、Inteceptor、validator、handler、Model向导

5、JBolt安装后开启提供了一个JFinal定制Eclipse视图，专门优化了常用操作和菜单的顺序位置，使用起来更清爽、更方便。

6、提供了JFinal-undertow、Jetty等服务器环境的自动创建、导入、生成配置文件等功能

7、普通工程和Maven工程里基于JFinal Undertow的项目都可以打包成JFinal定制的打包部署结构，Maven工程里使用的是assembly插件，普通成功里是专门为JFinal定制的JFinal Packaging Tool，一键秒级打包项目。

7、内置一键生成基于JFinal的Demo教学案例项目(待开发)

8、向导一键生成微信公众号、企业号、小程序的后端开发环境和配置信息(待开发)

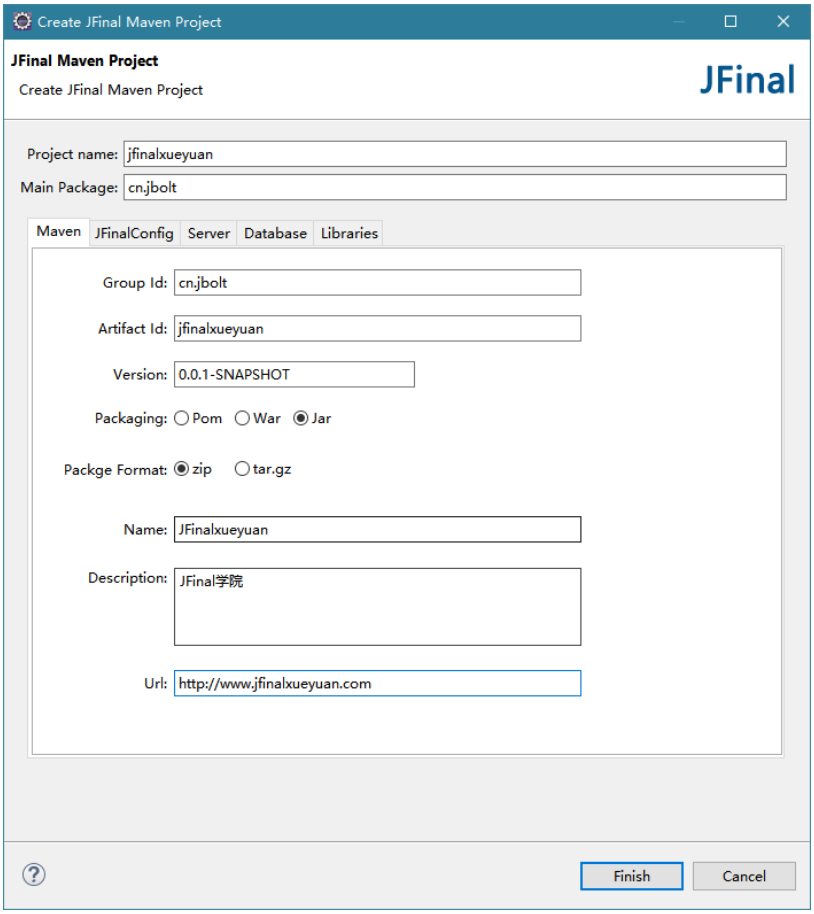
9、内置BaseController、BaseService封装常用 内置controller生成模板 service生成模板 一键实现增删改查分页(开发中)

10、JBolt提供的一套可以轻松二次开发后台管理系统（包含字典、全局配置、微信公众号管理、权限管理、用户管理、角色管理、登录注册等基础模块）（已开发了

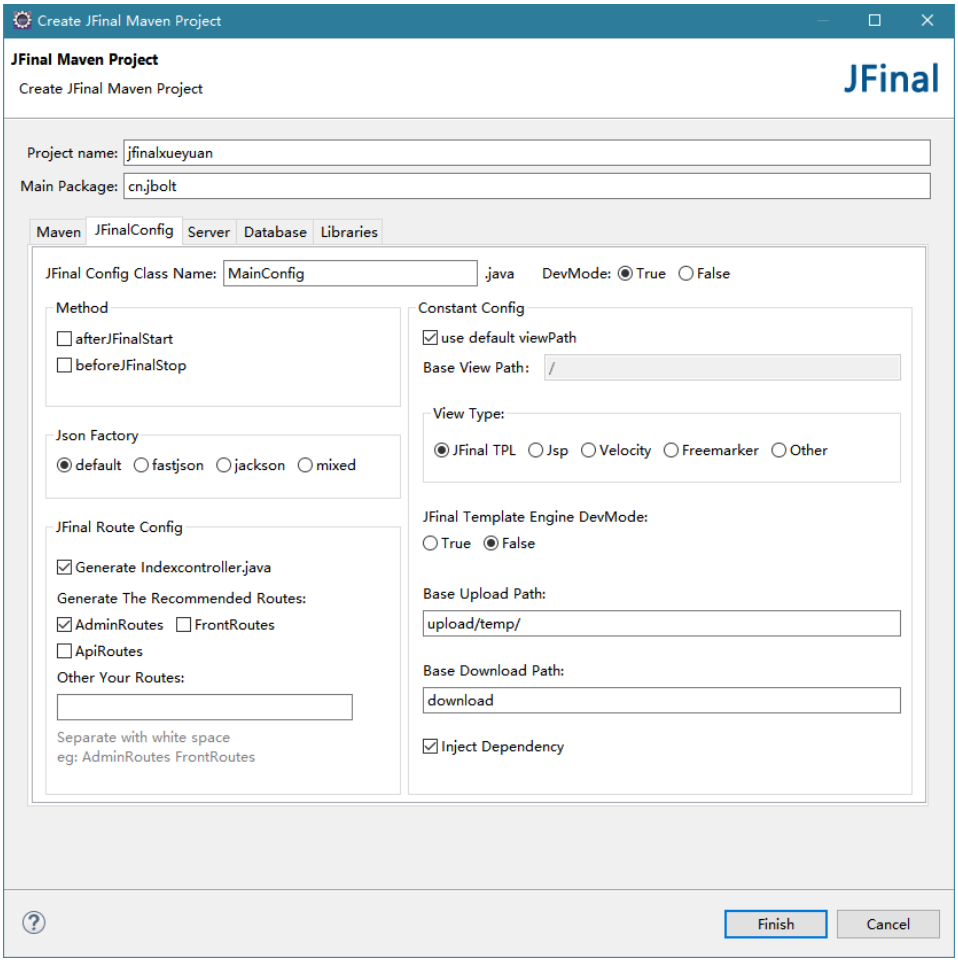
80%）

JBolt截图：

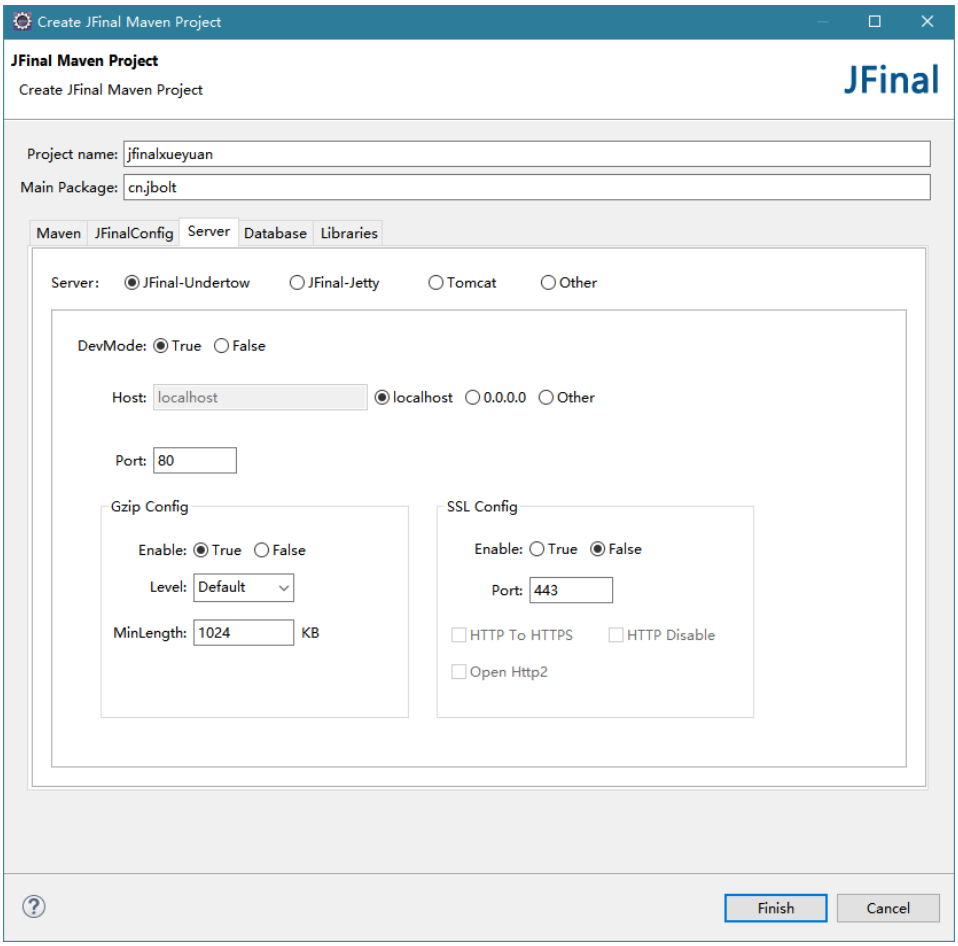
创建Maven项目的向导：



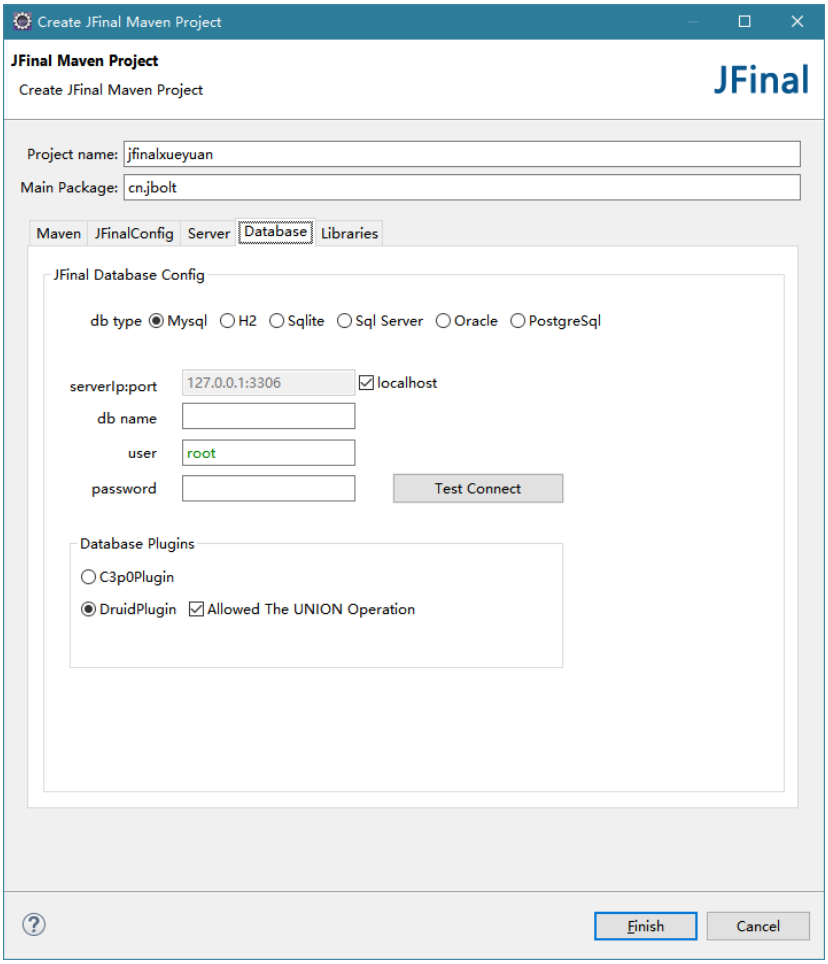
JFinalConfig的常见配置、路由配置等：



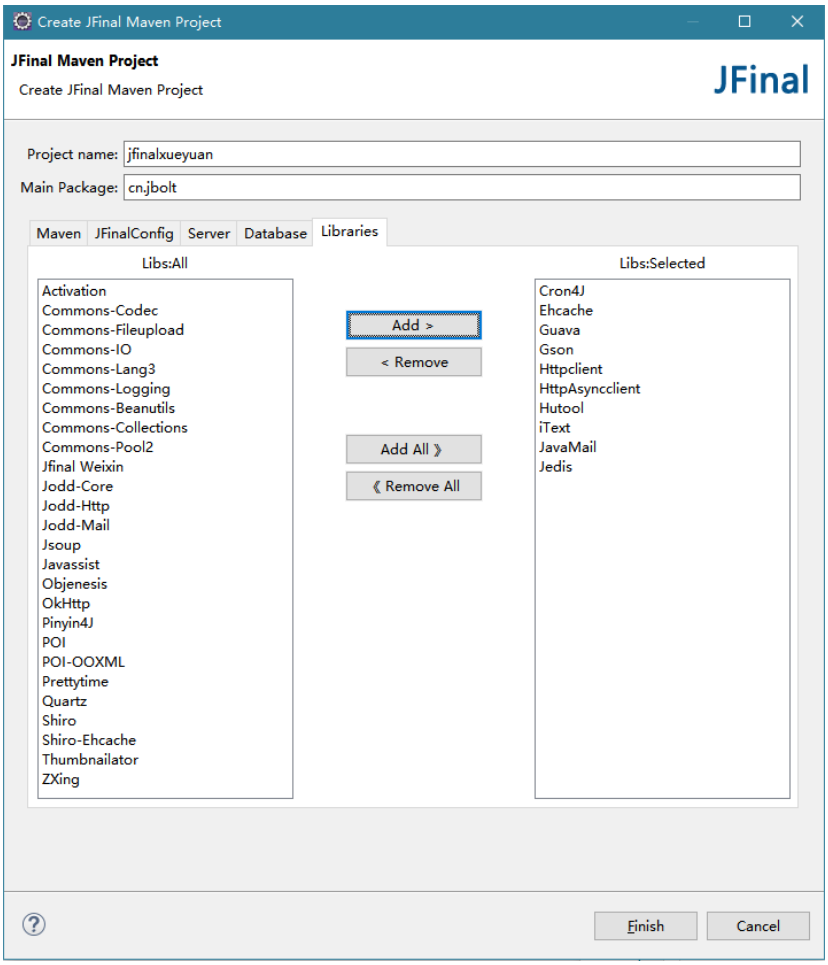
JFinal-Undertow Server的配置


Create JFinal Maven Project
JFinal Maven Project
Create JFinal Maven Project
JFinal
Project name: jfinalxueyuan
Main Package: cn.jbolt
Maven
JFinalConfig
Server
Database
Libraries
Server: JFinal-Undertow JFinal-Jetty Tomcat Other
DevMode: True False
Host: localhost localhost 0.0.0.0 Other
Port: 80
Gzip Config
Enable: True False
Level: Default
MinLength: 1024 KB
SSL Config
Enable: True False
Port: 443
HTTP To HTTPS
HTTP Disable
Open Http2
Finish
Cancel

数据库的配置：



第三方常用类库的自动导入配置：



还有更多功能正在开发，如果您有特殊需求，也可以联系我们，如果通用就更新到JBolt里 如果特殊，为您定制。

< 1.9 IDEA下开发
1.11 特别声明 >

1.11 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及部署方式 **仅仅针对于 JFinal 内部提供的 jfinal-undertow 以及 jetty-server 整合方式**。当碰到启动问题时如果并非在使用 jfinal 整合的 undertow、jetty，那么决然与 jfinal 无关，也不需要查看本章的文档，从网上查找 Java Web 启动与部署的知识即可解决。

如果不使用 JFinal 内部提供的 undertow、jetty 整合方式启动，那么可以去掉对相关的 jar 包依赖，maven 项目则可删掉相关 dependency 配置。

最后，如果部署没有使用 jfinal undertow，那么需要将 pom.xml 中的打包类型改为 war，否则会打出 jar 包：

```
<packaging>war</packaging>
```

同样，这个问题也与 jfinal 无关，是 maven 基础知识

< 1.10 JBolt 插件下开发
2.1 概述 >

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) {}
}
```

< 1.11 特别声明
2.2 configConstant(..) >

2.2 configConstant(..)

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。

< 2.1 概述
2.3 configRoute(..) >

2.3 configRoute(..)

此方法用来配置访问路由，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) {
    // 如果要将控制器超类中的 public 方法映射为 action 配置成 true，一般不用配置
    me.setMappingSuperClass(false);

    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(…, viewPath) 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 3.6 新增了一个配置方法：**setMappingSuperClass(boolean)**，默认值为 false。配置成为 true 时，你的 controller 的超类中的 public 方法也将会映射成 action。如果你的项目中使用了 jfinal weixin 项目的 MsgController，则需要将其配置成 true，因为 MsgController 中的 index() 需要被映射成 action 才能正常分发微信服务端的消息。

JFinal路由规则如下表：

url 组成	访问目标
controllerKey	YourController.index()
controllerKey/method	YourController.method()
controllerKey/method/v0-v1	YourController.method()，所带 url 参数值为：v0-v1
controllerKey/v0-v1	YourController.index()，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十分有用，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。

< 2.2 configConstant(..)

2.4 configEngine(..) >

2.4 configEngine(..)

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/layout.html");
}
```

上面的方法向模板引擎中添加了三个定义了 template function 的模板文件，更详细的介绍详见 Template Engine 那一章节的内容。

< 2.3 configRoute(..)
2.5 configPlugin(..) >

2.5 configPlugin(..)

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) {
    DruidPlugin dp = new DruidPlugin(jdbcUrl, userName, password);
    me.add(dp);

    ActiveRecordPlugin arp = new ActiveRecordPlugin(dp);
    arp.addMapping("user", User.class);
    me.add(arp);
}
```

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。

< 2.4 configEngine(..)
2.6 configInterceptor(..) >

2.6 configInterceptor(..)

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。

< 2.5 configPlugin(..)
2.7 configHandler(..) >

2.7 configHandler(..)

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());
}
```

< 2.6 configInterceptor(..)
2.8 onStart()、onStop() 回调配置 >

2.8 onStart()、onStop() 回调配置

在 JFinalConfig 继承类中可以添加 onStart() 与 onStop()，JFinal 会在系统启动完成之后以及系统关闭之前分别回调这两个方法：

```
// 系统启动完成后回调
public void onStart() {
}

// 系统关闭之前回调
public void onStop() {
}
```

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。

注意：jfinal 3.6 版本之前这两个方法名为：afterJFinalStart() 与 beforeJFinalStop()。为减少记忆成本、代码输入量以及输入手误的概率，jfinal 3.6 版本改为了目前更简短的方法名。老方法名仍然被保留，仍然可以使用，方便老项目升级到 jfinal 最新版本。

< 2.7 configHandler(..)
2.9 PropKit 读取配置 >

2.9 PropKit 读取配置

PropKit工具类用来读取外部键值对配置文件，PropKit可以极度方便地在系统任意时空使用，配置文件的格式如下：

```
userName=james
email=no-reply@jfinal.com
devMode=true
```

如下是 PropKit 代码示例：

```
PropKit.use("config.txt");
String userName = PropKit.get("userName");
String email = PropKit.get("email");

// Prop 配合用法
Prop p = PropKit.use("config.txt");
Boolean devMode = p.getBoolean("devMode");
```

如下是在项目中具体的使用示例：

```
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PorpKit.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。

< 2.8 onStart()、onStop() 回调配置
3.1 概述 >

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。

< 2.9 PropKit 读取配置
3.2 Action >

3.2 Action

在Controller之中定义的public方法称为Action。Action是请求的最小单位。Action方法必须在Controller中定义，且必须是public可见性。

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

以上代码中定义了两个Action：HelloController.index()、HelloController.test()。

Action可以有返回值，返回值可在拦截器中通过invocation.getReturnValue() 获取到，以便进行render控制。

< 3.1 概述
3.3 Action 参数注入 >

3.3 Action 参数注入

Action 参数注入是指为 action 方法传入参数，可以省去 getPara(...) 代码直接获得参数值，以下是代码示例：

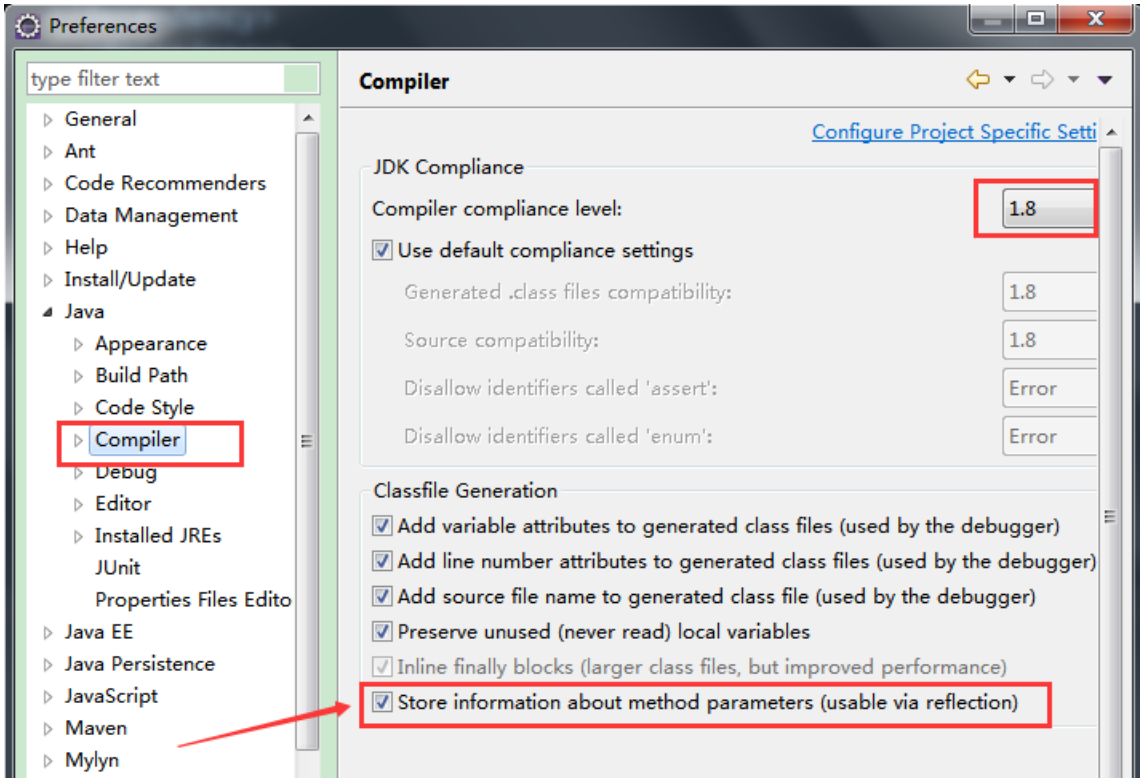
```
public class ProjectController extends Controller {
    public void index(Project project) {
       project.save();
       render("index.html");
    }
}
```

Action 参数注入可以代替 getPara、getBean、getModel 系列方法获取参数，使用 File、UploadFile 参数时可以代替 getFile 方法实现文件上传。这种传参方式还有一个好处是便于与 swagger 这类第三方无缝集成，生成API文档。

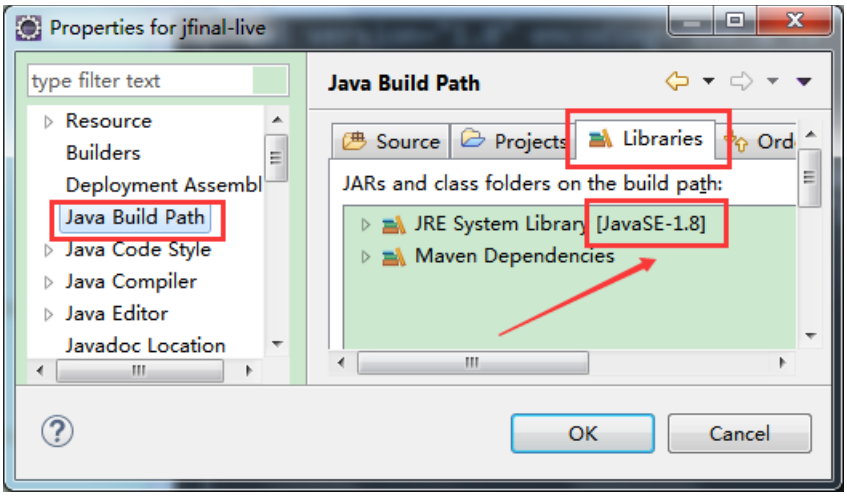
重要用法：如果 action 形参是一个 model 或者 bean，原先通过 getBean(User.class, "") 获取时第二个参数为空字符串或 null，那么与之等价的形参注入只需要用一下 @Para("") 注解即可：

public void action(**@Para("")**User user) { …. }

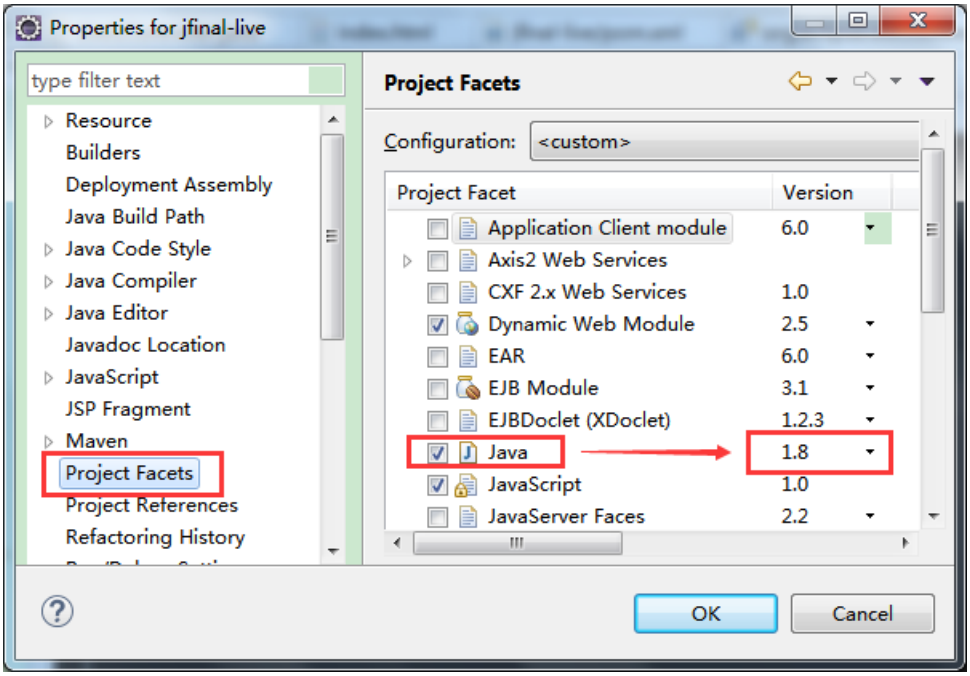
使用 Action 参数注入功能需要在开发工具中配置打开编译参数保留住方法参数名称。注意过于老旧的 eclipse 版本不支持 java 8 和该配置项，建议至少使用eclipse mars版本，以下是eclipse中的设置：



检查项目属性配置的Java Build Path菜单下的Libraries下的java版本是否为1.8：

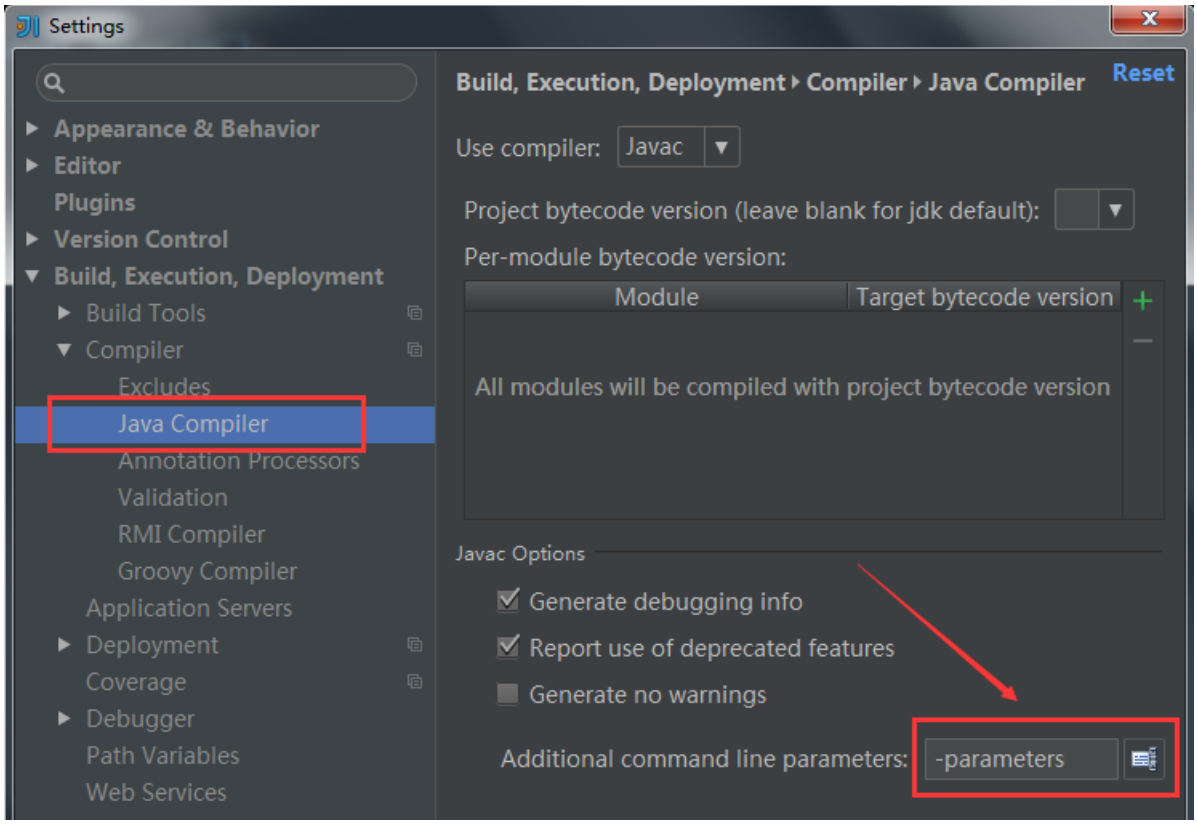


检查项目属性配置的Project Facets菜单下的 java版本配置确定是否为1.8：



注意：配置完成后，先将原有编译出来的class文件clear掉，并重新编译一次整个项目

如果使用IDEA，添加一个编译参数 -parameters即可，其配置方法如下：



如果要使用maven插件进行编译，为maven-compiler-plugin编译插件配置一个<compilerArgument>-parameters</compilerArgument> 属性即可：

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>3.6.1</version>
    <configuration>
        <source>1.8</source>
        <target>1.8</target>
        <encoding>UTF-8</encoding>
        <!-- java8 保留参数名编译参数 -->
        <compilerArgument>-parameters</compilerArgument>
    </configuration>
</plugin>
```

以上截图中的红色箭头指向部分是配置关键，以下 XML 配置内容与上面截图完全一样，提供出来便于复制使用：

```
<plugin>
   <groupId>org.apache.maven.plugins</groupId>
   <artifactId>maven-compiler-plugin</artifactId>
   <version>3.6.1</version>
   <configuration>
```

```
      <source>1.8</source>
      <target>1.8</target>
      <encoding>UTF-8</encoding>
      <!-- java8 保留参数名编译参数 -->
      <compilerArgument>-parameters</compilerArgument>
   </configuration>
</plugin>
```

最后要注意：jfinal 3.5 版直接支持 action 参数注入功能，但如果使用的 jfinal 3.2、3.3、3.4 这三个老版本希望支持该功能需要使用 jfinal 的 jfinal-java8 这个分支发行版，其 maven坐标如下：

```
<dependency>
    <groupId>com.jfinal</groupId>
    <artifactId>jfinal-java8</artifactId>
    <version>3.4</version>
</dependency>
```

如果用的是 jfinal 3.5 可以不必理会。

< 3.2 Action
3.4 getPara 系列方法 >

3.4 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"

jfinal 3.6 重要更新：jfinal 3.6 针对 getPara 系以及 getParaToXxx 系统方法添加了更简短的替代方法，以下是部分使用示例：

```
// 替代 getPara 的 get 用法
String title = get("title");

// 替代 getParaToInt 的 getInt 用法
Integer age = getInt("age");

// 替代 setAttr 的 set 用法
set("article", article);
```

jfinal 3.5 重要更新：jfinal 3.5 版本新增了 getRawData() 方法，可以很方便地从 http 请求 body 中获取 String 型的数据，通常这类数据是 json 或 XML 数据，例如：

```
String json = getRawData();
User user = FastJson.getJson().parse(json, User.class);
```

以上代码通过 getRawData() 获取到了客户端传过来的 String 型的 json 数据库。 getRawData() 方法可以在一次请求交互中多次反复调用，不会抛出异常。

这里要注意一个问题：通过 forwardAction(...) 转发到另一个 action 时，getRawData() 无法获取到数据，此时需要使用 setAttr("rawData", getRawData()) 将数据传递给 forward 到的目标 action，然后在目标 action 通过 getAttr("rawData") 获取。一般这种情况很少见。

< 3.3 Action 参数注入
3.5 getBean与getModel系列 >

3.5 getBean与getModel系列

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。

getBean方法用于支持传统Java Bean，包括支持使用jfinal生成器生成了getter、setter方法的Model，页面表单传参时使用与setter方法相一致的attrName，而非数据表字段名。

getModel与getBean区别在于前者使用数据库表字段名而后者使用与setter方法一致的属性名进行数据注入。建议优先使用getBean方法。

以下是一个简单的示例：

```
// 定义Model，在此为Blog
public class Blog extends Model<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。

如果希望在接收时跳过数据转换或者属性名错误异常可以传入true参：getBean(…, true)

< 3.4 getPara 系列方法
3.6 set / setAttr 方法 >

3.6 set / 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。

通过查看 jfinal 源码 Controller 可知 setAttr(String, Object) 方法在底层仅仅转调了底层的 HttpServletRequest 方法：

```
private HttpServletRequest request;

public Controller setAttr(String name, Object value) {
    request.setAttribute(name, value);
    return this;
}
```

jfinal 3.6 新增：为了进一步减少代码量、提升开发效率，jfinal 3.6 新增了 set 方法替代 setAttr，用法如下：

```
set("article", article);

// 链式用法
set("project", project).set("replyList", replyList).render("index.html");
```

jfinal 对于减少代码量、提升开发效率、降低学习成本的追求永不止步。

< 3.5 getBean与getModel系列
3.7 getFile文件上传 >

3.7 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的相对路径。

< 3.6 set / setAttr 方法
3.8 renderFile文件下载 >

3.8 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(path) 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的相对路径。

< 3.7 getFile文件上传
3.9 session操作方法 >

3.9 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。

```
public void login() {
   User user = loginService.login(...);
   if (user != null) {
      setSessionAttr("loginUser", user);
   }
}
```

为了便于项目在未来方便支持集群与分布式，已不建议使用 session 存放数据，建议将 session 范畴数据存放在数据库中。

< 3.8 renderFile文件下载
3.10 render系列方法 >

3.10 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、QrCode 二维码 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：

```
// 渲染名为test.html的视图，且视图类型为 JFinal Template
renderTemplate("test.html");

// 生成二维码
renderQrCode("content");

// 渲染名为test.html的视图，且视图类型为FreeMarker
renderFreeMarker("test.html");

// 渲染名为test.html的视图，且视图类型为Velocity
renderVelocity("test.html");

// 将所有setAttr(..)设置的变量转换成 json 并渲染
renderJson();

// 以 "users" 为根，仅将 userList 中的数据转换成 json 并渲染
renderJson("users", userList);

// 将user对象转换成 json 并渲染
renderJson(user);

// 直接渲染 json 字符串
renderJson("{\"age\":18}" );

// 仅将setAttr("user", user)与setAttr("blog", blog)设置的属性转换成json并渲染
renderJson(new  String[]{"user", "blog"});

// 渲染名为test.zip的文件，一般用于文件下载
renderFile("test.zip");

// 渲染纯文本内容 "Hello JFinal"
renderText("Hello JFinal");

// 渲染 Html 内容 "Hello Html"
renderHtml("Hello Html");

// 渲染名为 test.html 的文件，且状态为 404
renderError(404 , "test.html");

// 渲染名为 test.html 的文件，且状态为 500
renderError(500 , "test.html");

// 不渲染，即不向客户端返回数据
renderNull();

// 使用自定义的MyRender来渲染
render(new MyRender());
```

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。

< 3.9 session操作方法
4.1 概述 >

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、Poincut、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。

< 3.10 render系列方法
4.2 Interceptor >

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 接口仅仅定义了一个方法 public void intercept(Invocation inv)。以下是简单示例：

```
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。

常见错误：目前为止仍有很多同学忘了调用 inv.invoke() 方法，造成 controller 中的 action 不会被执行。在此再次强调一次，一定要调用一次 inv.invoke()，除非是刻意不去调用剩下的拦截器与 action，这种情况仍然需要使用 inv.getController().render()/renderJson() 调用一下相关的 render() 方法为客户端响应数据。

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

更正一下上面截图中倒数第三行的一处手误：setArg(int) 应该改为 setArg(int, Object)

< 4.1 概述
4.3 Before >

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，只需用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、Routes、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。

< 4.2 Interceptor
4.4 Clear >

4.4 Clear

拦截器从上到下依次分为Global、Routes、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、Routes、Class进行清除。Clear声明在Class层时将针对Global、Routes 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、Routes、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
  }
}
```

< 4.3 Before
4.5 Inject 依赖注入 >

4.5 Inject 依赖注入

使用 @Inject 注解可以向 Controller 以及 Interceptor 中注入依赖对象，使用注入功能需要如下配置：

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

配置完成以后就可以在控制器中使用了，例如：

```
public class AccountController {

   @Inject
   AccountService service;     // 此处会注入依赖对象

   public void index() {
       service.justDoIt();     // 调用被注入对象的方法
   }
}
```

@Inject 还可以用于拦截器的属性注入，例如：

```
public class MyInterceptor implements Interceptor {

    @Inject
    Service service;     // 此处会注入依赖对象

    public void intercept(Invocation inv) {
        service.justDoIt();     // 调用被注入对象的方法
        inv.invoke();
    }
}
```

如果需要创建的对象并不是 Controller 的属性，也不是 Interceptor 的属性，还可以使用 Aop.get(...) 方法进行依赖对象的创建以及注入，例如：

```
public class MyKit {

   static Service service = Aop.get(Service.class);

   public void doIt() {
      service.justDoIt();
   }
}
```

由于 MyKit 的创建并不是 jfinal 接管的，所以不能使用 @Inject 进行依赖注入。 而 Controller、Interceptor 的创建和组装是由 jfinal 接管的，所以可以使用 @Inject 注入依赖。

有了 Aop.get(...) 就可以在任何地方创建对象并且对创建的对象进行注入。此外还可以使用 Aop.inject(...) 仅仅向对象注入依赖但不创建对象。

@Inject 注解还支持指定注入的实现类，例如下面的代码，将为 Service 注入 MyService 对象：

```
@Inject(MyService.class)
Service service;
```

当被注入的对象是通过配置指定的，不能使用 @Inject(...) 指定参数的形式来指定被注入的类型时，还可以通过 Aop.addMapping(...) 事先添加映射，例如：

```
Aop.addMapping(Service.class, MyService.class);
```

通过上面的映射，下面的代码将会为 Service 注入 MyService

```
public class IndexController {

    @Inject
    Service service;

    public void index() {
        service.justDoIt();
    }
}
```

< 4.4 Clear
4.6 Aop 工具 >

4.6 Aop 工具

Aop 工具类可以创建对象并且对创建的对象进行依赖注入，例如：

```
Service service = Aop.get(Service.class);
```

以上代码会创建 Service 对象，如果 Service 中使用了 @Before 配置过拦截器，那么会生效，如果 Service 中的属性使用了 @Inject，则会被注入依赖对象。

Aop.inject(...) 方法相对于 Aop.get(...) 方法少一个对象创建功能。

此外，Aop 工具类还可以添加映射：

```
Aop.addMapping(Service.class, MyService.class);
```

通过上面的映射，下面的代码将会为 Serivce 创建 MyService 对象，而非 Service 对象：

```
Aop.get(Service.class);     // 这里获取到的是 MyService 对象

@Inject
Service service;            // 这里被注入的是 MyService 对象
```

Aop.addMapping(...) 的主要用途是为接口、抽象类指定被注入的具体实现类。

< 4.5 Inject 依赖注入
4.7 Routes级别拦截器 >

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() {
     // 此处配置 Routes 级别的拦截器，可配置多个
     addInterceptor(new AdminAuthInterceptor());

     add("/admin", IndexAdminController.class, "/index");
     add("/admin/project", ProjectAdminController.class, "/project");
     add("/admin/share", ShareAdminController.class, "/share");
  }
}
```

在上例中，AdminAuthInterceptor 将拦截IndexAdminController、ProjectAdminController、ShareAdminController 中所有的 action 方法。

Routes 拦截器在功能上通过一行代码，同时为多个 Controller 配置好相同的拦截器，减少了代码冗余。Routes 级别拦截器将在 Class 级别拦截器之前被调用。

< 4.6 Aop 工具
5.1 概述 >

5.1 概述

重大更新：自 jfinal 3.0 起，添加了 sql 管理模块，比 mybatis 使用 XML 管理 sql 的方案要爽得多，快速查看：http://www.jfinal.comhttps://www.jfinal.com/doc/5-13

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。

ActiveRecord 模式的核心是：一个 Model 对象唯一对应数据库表中的一条记录，而对应关系依靠的是数据库表的主键值。

因此，ActiveRecord 模式要求数据库表必须要有主键。当数据库表没有主键时，只能使用 Db + Record 模式来操作数据库。

< 4.7 Routes级别拦截器
5.2 ActiveRecordPlugin >

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ActiveRecrod支持插件。ActiveReceord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)。

重要：以上的 arp.addMapping(...) 映射配置，可以让 jfinal 生成器自动化完成，不再需要手动添加这类配置，具体用法见文档：https://www.jfinal.comhttps://www.jfinal.com/doc/5-4

< 5.1 概述
5.3 Model >

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.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.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。

< 5.2 ActiveRecordPlugin
5.4 Generator与JavaBean >

5.4 Generator与JavaBean

1、生成器的使用

ActiveRecord 模块的 com.jfinal.plugin.activerecord.generator 包下，提供了一个 Generator 工具类，可自动生成 Model、BaseModel、MappingKit、DataDictionary 四类文件。

生成后的 Model 与 java bean 合体，立即拥有了 getter、setter 方法，使之遵守传统的 java bean 规范，立即拥有了传统 JavaBean 所有的优势，开发过程中不再需要记忆字段名。

使用生成器通常只需配置Generator的四个参数即可，以下是具体使用示例：

```
// 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);
// 在 getter、setter 方法上生成字段备注内容
gernerator.setGenerateRemarks(true);
gernerator.generate();
```

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。

可在官网下载jfinal-demo项目，其中的生成器可直接用于项目：http://www.jfinal.com

2、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。

3、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

4、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

5、常见问题解决

- Sql Server数据库在使用生成器之时，会获取到系统自带的表，需要对这些表进行过滤，具体办法参考：http://www.jfinal.com/share/211

< 5.3 Model
5.5 独创Db + Record模式 >

5.5 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);
```

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run()方法返回false，则自动回滚事务。

< 5.4 Generator与JavaBean
5.6 paginate 分页 >

5.6 paginate 分页

1、常用 paginate

Model 与 Db 中提供了最常用的分页API：paginate(int pageNumber, int pageSize, String select, String sqlExceptSelect, Object... paras)

其中的参数含义分别为：当前页的页号、每页数据条数、sql语句的select部分、sql语句除了select以外的部分、查询参数。绝大多数情况下使用这个API即可。以下是使用示例：

```
dao.paginate(1, 10, "select *", "from girl where age > ? and weight < ?", 18, 50);
```

2、sql 最外层带 group by 的 paginate

API 原型：paginate(int pageNumber, int pageSize, boolean isGroupBySql, String select, String sqlExceptSelect, Object... paras)，相对于第一种仅仅多了一个boolean isGroupBySql参数，以下是代码示例：

```
dao.paginate(1, 10, true, "select *", "from girl where age > ? group by age", 18);
```

以上代码中 sql 的最外层有一个 group by age，所以第三个参数 isGroupBySql 要传入 true 值。

如果是嵌套型sql，但是 group by 不在最外层，那么第三个参数必须为 false，例如：select * from (select x from t group by y) as temp。

再次强调：isGroupBy 参数只有在最外层 sql 具有 group by 子句时才能为 true 值，嵌套 sql 中仅仅内层具有 group by 子句时仍然要使用 false。

3、paginateByFullSql

API 原型：paginateByFullSql(int pageNumber, int pageSize, String totalRowSql, String findSql, Object... paras)。

相对于其它 paginate API，将查询总行数与查询数据的两条sql独立出来，这样处理主要是应对具有复杂order by语句或者select中带有distinct的情况，只有在使用第一种paginate出现异常时才需要使用该API，以下是代码示例：

```
String from = "from girl where age > ?";
String totalRowSql = "select count(*) " + from;
String findSql = "select * " + from + " order by age";
dao.paginateByFullSql(1, 10, totalRowSql, findSql, 18);
```

上例代码中的order by子句并不复杂，所以仍然可以使用第一种API搞定。

重点：paginateByFullSql 最关键的地方是 **totalRwoSql、findSql 这两条 sql 要能够共用最后一个参数 Object... paras**，相当于 dao.find(totalRwoSql, paras) 与 dao.find(findSql, paras) 都要能正确执行，否则断然不能使用 paginateByFullSql。

当 paginate、paginateByFullSql 仍然无法满足业务需求时，可以通过使用 Model.find、Db.query 系列方法组合出自己想要的分页方法。jfinal 只为最常见场景提供支持。

4、使用SqlPara 参数的 paginate

API原型： paginate(int pageNumber, int pageSize, SqlPara sqlPara)，用于配合sql管理功能使用，将在sql管理功能那章做介绍。

5、常见问题解决

首先必须要介绍一下 paginate 底层的基本实现原理，才能有效呈现和解决问题，假定有如下分页代码：

```
paginate(1, 5, "select *", "from article where id > ? order by id", 10);
```

底层首先会利用上面第三个与第四个 String 参数生成如下 sql 去获取分页所需要的满足查询条件的所有记录数，也叫 totalRow：

```
"select count(*)" + "from article where id > 10"
```

注意看上面的 sql，第一部分的 "select count(*)" 是固定写死的，第二部分是根据用户的第四个参数，去除 order by id 而得到的。

去除 order by 子句这部分，一是因为很多数据库根本不支持 select count(*) 型 sql 带有 order by 子句，必须要去掉否则出错。二是因为 select count(*) 查询在有没有 order by 子句时结果是完全一样的，所以去除后有助于提升性能。

第一类错误： 如果用户分页的第二个参数不是 "select *"，而是里面带有一个或多个问号占位符，这种情况下最后面的 para 部分不仅仅只有上例中的 10 这一个参数

以下是这种问题的一个例子：

```
paginate(1, 5, "select (... where x = ?)", "from article where id=?", 8, 10)
```

注意看上面的例子中有两个问号占位符，对应的参数值是 8 和 10。但是生成的用于计算 totalRow 的代码如下：

```
queryLong("select count(*) from article where id=?", 8, 10);
```

因此，多出来一个参数 8 是多余的，从而造成异常。出现这种情况只需要在外层再套一个 sql 就可解决：

```
paginate(1, 5, "select *", "from (" + 原sql在此 + ") as t", 8, 10);
```

也就是将原来的 sql 外层再套一个 select * from (...) as t，让第三个参数中没有问号占位符而是一个 "select *"。这样处理就避免掉了第一个问号占位符被生成 totalRow 的 sql 吃掉了。

第二类错误： 如果 order by 子句使用了子查询，或者使用了函数调用，例如：

```
paginate(1, 5, "select *", "from ... order by concat(...)");
```

如上所示，order by 子句使用了 concat 函数，由于 jfinal 是使用了一个简单的正则来移除 order by 子句的，但是无法完全移除带有函数的 order by 子句，也就是移除不干净，结果就是 sql 是错误的。

jfinal 也曾使用复杂的正则来将 order by 子句移除干净，但性能低得无法忍受，最后只能取舍使用简单正则。 由于 order by 子句可以是极为复杂的嵌套 sql，所以要移除干净的代价就是性能的急剧下降，jfinal 也是不得以这样设计。

所以，解决第二类常见错误就是使用 paginateByFullSql 方法，这个方法让你的计算 totalRow 的 sql 与查询数据的 sql 完全手写，jfinal 也就不再需要处理 select 部分与移除 order by 这部分，全交由用户自己手写。

综上，只要了解了 paginate 在底层的工作机制，解决问题就很容易了。最终可以通过 paginateByFullSql 来稍多写点代码来解决。

< 5.5 独创Db + Record模式

5.7 数据库事务处理 >

5.7 数据库事务处理

1、Db.tx 事务

在 Db 工具类里面，提供了一个系列的 tx(...) 方法支持数据库事务，以下是 Java 8 的 lambda 语法使用示例：

```
Db.tx(() -> {
  Db.update("update t1 set f1 = ?", 123);
  Db.update("update t2 set f2 = ?", 456);
  return true;
});
```

以上代码中的两个 Db.update 数据库操作将开启事务，return true 提交事务，return false 则回滚事务。Db.tx(...) 做事务的好处是控制粒度更细，也即不必抛出异常即可回滚。

Db.tx 方法 **默认针对主数据源** 进行事务处理，如果希望对其它数据源开启事务，使用 **Db.use(configName).tx(...)** 即可。此外，Db.tx(...) 还支持指定事务级别：

```
Db.tx(Connection.TRANSACTION_SERIALIZABLE, () -> {
  Db.update(...);
  new User().setNickName("james").save();
  return true;
});
```

以上代码中的 Db.tx(...) 第一个参数传入了事务级别参数 Connection.TRANSACTION_SERIALIZABLE，该方法对于需要灵活控制事务级的场景十分方便实用。

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。

2、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。

当事务拦截器 Tx 配置在 Controller 层，并且希望使用 try catch 对其进行响应控制，在 jfinal 3.6 及更高版本中可以像下面这样使用：

```
@Before(Tx.class)
public void trans {
  try {
```

```
    service.justDoIt(...);
    render("ok.html");
  } catch (Exception e) {
    render("error.html");
    throw e;
  }
}
```

如上所示，只需要在 catch 块中直接使用 render(....) 就可以在异常发生时指定响应的模板。最后一定要**使用 throw e** 将异常向上抛出，处于上层的 Tx 拦截器才能感知异常并回滚事务。（注意：这个功能是 jfinal 3.6 才添加的）

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。

特别注意：声明式事务默认只针对主数据源进行回滚，如果希望针对 “非主数据源” 进行回滚，需要使用注解进行配置，以下是示例：

```
@TxConfig("otherConfigName")
@Before(Tx.class)
public void doIt() {
   ...
}
```

以上代码中的 @TxConfig 注解可以配置针对 otherConfigName 进行回滚。Tx 拦截器是通过捕捉到异常以后才回滚事务的，所以上述代码中的 doIt() 方法中如果有 try catch 块捕获到异常，必须再次抛出，才能让 Tx 拦截器感知并回滚事务。

3、使用技巧

建议优先使用 Db.tx(...) 做数据库事务，一是该方式可以让事务覆盖的代码量最小，性能会最好。二是该方式可以利用返回值来控制是否回滚事务，而 Tx 拦截器只能通过捕捉到异常来回滚事务。三是 Java 8 的 lambda 语法使其代码也很简洁。

Tx 事务拦截器在捕捉到异常后回滚事务，会再次抛向外抛出异常，所以在使用 Tx 拦截器来做事务处理时，通常需要再额外添加一个 ExceptionInterceptor，放在 Tx 拦截器之前去再次捕捉到 Tx 所抛出的异常，然后在这里做 renderJson/render 之类的动作向客户端展示不同的数据与页面。如果不添加这样的机制，会展示一个统一默认的 500 error 页面，无法满足所有需求。

综上，强烈建议优先使用 Db.tx(...) 做事务处理。

4、事务级别与性能

JDBC 默认的事务级别为：Connection.TRANSACTION_READ_COMMITTED。为了避免某些同学的应用场景下对事务级别要求较高，jfinal 的 ActiveRecordPlugin 默认使用的是 Connection.TRANSACTION_REPEATABLE_READ，但这在对某个表存在高并发锁争用时性能会下降，这时可以通过配置事务级别来提升性能：

```
public void configPlugin(Plugins me) {
    ActiveRecordPlugin arp = new ActiveRecordPlugin(...);
```

```
    arp.setTransactionLevel(Connection.TRANSACTION_REPEATABLE_READ);
    me.add(arp);
}
```

有一位同学就碰到了由事务级别引起的性能问题：

http://www.jfinal.com/feedback/4703?p=2#reply_start

< 5.6 paginate 分页
5.8 Cache 缓存 >

5.8 Cache 缓存

1、使用 Ehcache 缓存

ActiveRecord 可以使用缓存以大大提高性能，默认的缓存实现是 ehcache，使用时需要引入 ehcache 的 jar 包及其配置文件，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。

2、使用任意缓存实现

除了要把使用默认的 ehcache 实现以外，还可以通过实现 ICache 接口切换到任意的缓存实现上去，下面是个简单提示意性代码实现：

```
public class MyCache implements ICache {
  public <T>T get(String cacheName, Object key) {
  }

  public void put(String cacheName, Object key, Object value) {
  }

  public void remove(String cacheName, Object key) {
  }

  public void removeAll(String cacheName) {
  }
}
```

如上代码所示，MyCache 需要实现 ICache 中的四个抽象方法，然后通过下面的配置方式即可切换到自己的 cache 实现上去：

```
ActiveRecordPlugin arp = new ActiveRecordPlugin(...);
arp.setCache(new MyCache());
```

如上代码所示，通过调用 ActiveRecordPlugin.setCache(...) 便可切换 cache 实现。

< 5.7 数据库事务处理
5.9 Dialect多数据库支持 >

5.9 Dialect多数据库支持

目前ActiveRecordPlugin提供了MysqlDialect、OracleDialect、PostgresqlDialect、SqlServerDialect、Sqlite3Dialect、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 Cache 缓存
5.10 表关联操作 >

5.10 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 Dialect多数据库支持
5.11 复合主键 >

5.11 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 表关联操作
5.12 Oracle支持 >

5.12 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(……);
    me.add(dp);
    //配置Oracle驱动
    dp.setDriverClass("oracle.jdbc.driver.OracleDriver");

    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.addMaping(“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.addMaping(“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 复合主键
5.13 SQL 管理与动态生成 >

5.13 SQL 管理与动态生成

JFinal利用自带的Template Engine极为简洁的实现了Sql管理功能。一如既往的极简设计，仅有#sql、#para、#namespace三个指令，学习成本依然低到极致。

重要：除了以上三个 sql 管理专用指令以外，jfinal 模板引擎的所有指令和功能也可以用在 sql 管理，jfinal 模板引擎用法见第 6 章：http://www.jfinal.comhttps://www.jfinal.com/doc/6-1

1、基本配置

在ActiveRecordPlugin中使用sql管理功能示例代码如下：

```
ActiveRecordPlugin arp = new ActiveRecordPlugin(druidPlugin);
arp.getEngine().setSourceFactory(new ClassPathSourceFactory());
arp.addSqlTemplate("all.sql");
_MappingKit.mapping(arp);
me.add(arp);
```

如上例所示，arp.getEngine().setSourceFactory(new ClassPathSourceFactory()) 这行代码，设置了模板引擎将从 class path 或者 jar 包中读取 sql 文件。 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 class path 之下，进而可以被读取到。

第三行代码通过 arp.addSqlTemplate(…) 添加外部 sql 模板文件，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，有利于多数据源下对 sql 进行模块化管理。

如果希望在开发阶段可以对修改的sql文件实现热加载，可以配置arp.setDevMode(true)或者arp.getEngine().setDevMode(true)，如果不配置则默认使用configConstant中的me.setDevMode(…)配置。

特别注意：sql管理模块使用的模板引擎并非在 configEngine(Engine me)配置，因此在对其配置 shared method、directive 等扩展时需要使用 activeRecordPlugin.getEngine() 得到用于 sql 管理功能的 Engine 对象，然后对该 Engine 对象进行配置。

2、#sql 指令

通过#sql指令可以定义sql语句，如下是代码示例：

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

上例通过 #sql 指令在模板文件中定义了key值为“findGirl”的sql语句，在java 代码中的获取方式如下：

```
String sql = Db.getSql("findGirl");
Db.find(sql, 16, 23);
```

上例中第一行代码通过Db.getSql()方法获取到定义好的sql语句，第二行代码直接将sql用于数据库查询。

此外，还可以通过Model.getSql(key)方法来获取sql语句，功能与Db.getSql(key)基本一样，唯一不同的是为多数据源分别配置了sql模板的场景：

- Model.getSql()在自身所对应的ActiveRecordPlugin的sql模板中去取sql
- Db.getSql()在主ActiveRecordPlugin的sql模板中去取sql
- 可通过Db.use(…).getSql(…) 实现Model.getSql()相同功能

也就是说，多数据源之下，可为不同的ActiveRecordPlugin对象分别去配置sql模板，有利于模块化管理。

3、#para 指令

#para 指令用于生成 sql 中的问号占位符以及问号占位符所对应的参数值，两者分别保存在 SqlPara对象的 sql 和

paraList 属性之中。

#para指令支持两种用法，一种是传入 **int型常量参数** 的用法，如下示例展示的是int型常量参数的用法：

```
#sql("findGirl")
  select * from girl where age > #para(0) and weight < #para(1)
#end
```

上例代码中两个 #para 指令，传入了两个 int 型常量参数，所对应的java后端代码必须调用getSqlPara(String key, Object... paras)，如下是代码示例：

```
SqlPara sqlPara = Db.getSqlPara("findGirl", 18, 50);
Db.find(sqlPara);
```

以上第一行代码中的 18 与 50 这两个参数，分别被前面 #sql 指令中定义的 #para(0) 与 #para(1) 所使用。

Db.getSqlPara(String key, Object... paras) 方法的第二个参数 Object... paras，在传入实际参数时，下标值从 0 开始算起与 #para(int) 指令中使用的 int 型常量相对应。

#para 指令的另一种用法是传入除了 int 型常量以外的任意类型参数(注意：两种用法处在同一个 #sql 指令之中时只能选择其中一种)，如下是代码示例：

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

与上例模板文件配套的java代码如下所示：

```
Kv cond = Kv.by("age", 18).set("weight", 50);
SqlPara sp = Db.getSqlPara("findGirl", cond);
Db.find(sp);
```

上例代码获取到的SqlPara对象sp中封装的sql为：select * from girl where age > ? and weight < ?，封装的与sql问号占位符次序一致的参数列表值为：[18, 50]。

以上两个示例，获取到的SqlPara对象中的值完全一样。其中的sql值都为：select * from girl where age > **?** and weight < **?**，其中的参数列表值也都为 [18、50]。不同的是 #para 用法不同，以及它们对应的java代码传参方式不同，前者传入的是Object... paras参数，后者是Map data参数。

切记：#para 指令所到之处**永远是生成一个问号占位符**，并不是参数的值，参数值被生成在了SqlPara对象的paraList属性之中，通过sqlPara.getPara()可获取。如果想生成参数值用一下模板输出指令即可：#(value)

极其重要的通用技巧：如果某些数据库操作API不支持SqlPara参数，而只支持String sql和Object... paras这两个参数，可以这样来用：method(sqlPara.getSql(), sqlPara.getPara())。这样就可以让所有这类API都能用上Sql管理功能。

加餐：有些同学希望在 sql 文件中获取getSqlPara(String, Object... paras) 方法传入的paras参数，可以通过表达式 **_PARA_ARRAY_[index]** 来获取到下标为index的参数值。

由于经常有人问到 mysql 数据库 sql 语句的 like 子句用法，补充如下示例：

```
#sql("search")
  select * from article where title like concat('%', #para(title), '%')
#end
```

以上示例的like用法完全是 JDBC 决定的，JFinal仅仅是生成了如下sql而已：

select * from article where title like concat('%', ?, '%')，也就是仅仅将 #para(title) 替换生成为一个问号占位 "?" 而已。

4、#namespace 指令

#namespace 指令为 sql 语句指定命名空间，不同的命名空间可以让#sql指令使用相同的key值去定义sql，有利于模块化管理，如下所示：

```
#namespace("japan")
  #sql("findGirl")
    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namesapce值前缀 + 句点符 + key即可：

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

5、分页用法

Sql管理实现分页功能，在使用#sql定义sql时，与普通查询完全一样，不需要使用额外的指令，在java代码中使用getSqlPara得到SqlPara对象以后，直接扔给Db或者Model的paginate方法就打完收工了，以下是代码示例：

```
SqlPara sqlPara = Db.getSqlPara("findGirl", 18, 50);
Db.paginate(1, 10, sqlPara);
```

如以上代码所示，将sqlPara对象直接用于paginate方法即可，而#sql定义与普通的非分页sql定义完全相同。

传统而平庸的sql管理框架实现分页功能额外引入很多无聊的概念，例如需要引入page指令、orderBy指令、select指令或者各种tag等无聊的东西，徒增很多学习成本，浪费广大开发者的生命。

6、高级用法

除了#sql、#para、#namespace之外，还可以使用JFinal Template Engine中所有存在的指令，生成复杂条件的sql语句，以下是相对灵活的示例：

```
#sql("find")
  select * from girl
  #for(x : cond)
    #(for.first ? "where": "and") #(x.key) #para(x.value)
  #end
#end
```

以上代码#for指令对Map类型的cond参数进行迭代，动态生成自由的查询条件。上图中的三元表达式表示在第一次迭代时生成 where，后续则生成 and 。#(x.key) 输出参数 key 值，#para(x.value) 输出一个问号占位符，以及将参数 value 值输出到 SqlPara.paraList 中去。

以上sql 模板所对应的 java 代码如下：

```
Kv cond = Kv.by("age > ", 16).set("sex = ", "female");
SqlPara sp = Db.getSqlPara("find", Kv.by("cond", cond));
Db.find(sp);
```

以上第一行代码是 JFinal 独创的参数带有比较运算符的用法，可以同时生成sql查询条件名称、条件运算符号、参数列表，一石三鸟。甚至可以将此法用于 and or not再搭配一个 LinkedHashMap 生成更加灵活的逻辑组合条件sql。

更加好玩的用法，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，以便消除重复性 sql 代码。总之，利用 #sql、#para、#namespace 这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。

注意：以上例子中的Kv是 JFinal 提供的用户体验更好的 Map 实现，使用任意的 Map 实现都可以，不限定为Kv。

< 5.12 Oracle支持
5.14 多数据源支持 >

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

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

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

以上代码创建了创了两个ActiveRecordPlugin实例arpMysql与arpOrace，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ActiveRecrodPlugin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(configName)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。

< 5.13 SQL 管理与动态生成
5.15 独立使用ActiveRecord >

5.15 独立使用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 jfinal 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 多数据源支持
5.16 存储过程调用 >

5.16 存储过程调用

使用工具类 Db 可以很方便调用存储过程，以下是代码示例：

```
Db.execute((connection) -> {
    CallableStatement cs = connection.prepareCall(...);
    cs.setObject(1, ...);
    cs.setObject(2, ...);
    cs.execute();
    cs.close();
    return cs.getObject(1);
});
```

如上所示，使用 Db.execute(...) 可以很方便去调用存储过程，其中的 connection 是一个 Connection 类型的对象，该对象在使用完以后，不必 close()，因为 jfinal 在上层会默认帮你 close() 掉。

< 5.15 独立使用ActiveRecord
6.1 概述 >

6.1 概述

Enjoy 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。

与以往任何一款 java 模板引擎都有显著的不同，极简设计、独创算法、极爽开发体验，从根本上重新定义了模板引擎，这里是发布时的盛况，传送门：**JFinal 3.0 发布，重新定义模板引擎**

从 JFinal 3.0 到 JFinal 3.3，Enjoy 引擎进行了精细化的打磨，这里是与 Enjoy 引擎打磨有关的近几个版本发布时的盛况:

JFinal 3.0 发布，重新定义模板引擎

JFinal 3.1 发布，没有繁琐、没有复杂，只有妙不可言

JFinal 3.2 发布，星星之火已成燎原之势

JFinal 3.3 发布，天下武功，唯快不破

Enjoy 模板引擎专为 java 开发者打造，所以坚持两个核心设计理念：一是在模板中可以直接与 java 代码通畅地交互，二是尽可能沿用 java 语法规则，将学习成本降到极致。

因此，立即掌握 90% 的用法，只需要记住一句话：JFinal 模板引擎表达式与 Java 是直接打通的。

记住了上面这句话，就可以像下面这样愉快地使用模板引擎了：

```
// 算术运算
1 + 2 / 3 * 4
// 比较运算
1 > 2
// 逻辑运算
!a && b != c || d == e
// 三元表达式
a > 0 ? a : b
// 方法调用
"abcdef".substring(0, 3)
target.method(p1, p2, pn)
```

Enjoy 模板引擎核心概念只有指令与表达式这两个。而表达式是与 Java 直接打通的，所以没有学习成本，剩下来只有 #if、#for、#define、#set、#include、#(...) 六个指令需要了解，而这六个指令的学习成本又极低。

Enjoy 模板引擎发布一年多以来，得到了非常多用户的喜爱，反馈证明体验极好，强烈建议还没能尝试过的同学们试用。

< 5.16 存储过程调用
6.2 引擎配置 >

6.2 引擎配置

1、配置入口

Enjoy 引擎的配置入口统一在 Engine 类中。Engine 类中提供了一系列 setter 方法帮助引导配置，减少记忆负担，例如：

```
// 支持模板热加载
engine.setDevMode(true);

// 添加共享模板函数
engine.addSharedFunction("_layout.html");
```

2、配置多个 Engine 对象

由于 Enjoy 引擎被设计为多种用途，同一个项目中的不同模块可以使用不同的 Engine 对象，分别用于不同的用途，那么则需要分别配置，各个 engine 对象的配置彼此独立互不干扰。

例如 jfinal 中的 Controller.render(String) 以及 SQL 管理功能 Db.getSqlPara(...) 就分别使用了两个不同的 Engine 对象，所以这两个对象需要独立配置，并且配置的地点是完全不同的。

应用于 Controller.render(String) 的 Engine 对象的配置在 **configEngine(Engine me)** 中进行：

```
public void configEngine(Engine me) {
  // devMode 配置为 true，将支持模板实时热加载
  me.setDevMode(true);
}
```

应用于 SQL 管理的 Engine 对象的配置在 **configPlugin(Plugins me)** 中进行：

```
public void configPlugin(Plugins me) {
  ActiveRecordPlugin arp = new ActiveRecordPlugin(...);
  Engine engine = arp.getEngine();

  // 上面的代码获取到了用于 sql 管理功能的 Engine 对象，接着就可以开始配置了
  engine.setToClassPathSourceFactory();
  engine.addSharedMethod(new StrKit());

  me.add(arp);
}
```

常见错误：经常有同学在配置 SQL 管理的 Engine 对象时，是在 configEngine(Engine me) 中进行配置，造成配置错配的问题。

同理，自己创建出来的 Engine 对象也需要独立配置：

```
Engine engine = Engine.create("myEngine");
engine.setDevMode(true);
```

3、多 Engine 对象管理

使用 Engine.create(engineName) 创建的 engine 对象，可以在任意地方通过 Engine.use(engineName) 很方便获取，管理多

个 engine 对象十分方便，例如：

```
// 创建一个 Engine 对象并进行配置
Engine forEmail = Engine.create("forEmail");
forEmail.addSharedMethod(EmailKit.class);
forEmail.addSharedFunction("email-function.txt");

// 创建另一个 Engine 对象并进行配置
Engine forWeixin = Engine.create("forWeixin");
forWeixin.addSharedMethod(WeixinKit.class);
forWeixin.addSharedFunction("weixin-function.txt");
```

上面代码创建了两个 Engine 对象，并分别取名为 "forEmail" 与 "forWeixin"，随后就可以分别使用这两个 Engine 对象了：

```
// 使用名为 "forEmail" 的 engine 对象
String ret = Engine.use("forEmail").getTemplate("email-template.txt").renderToString(...);
System.out.print(ret);

// 使用名为 "forWeixin" 的 engine 对象
String ret = Engine.use("forWeixin").getTemplate("weixin-template.txt").renderToString(...);
System.out.print(ret);
```

如上代码所示，通过 Engine.use(...) 方法可以获取到 Engine.create(...) 创建的对象，既很方便独立配置，也很方便独立获取使用。

4、模板热加载配置

为了达到最高性能 Enjoy 引擎默认对模板解析结果进行缓存，所以模板被加载之后的修改不会生效，如果需要实时生效需要如下配置：

```
engine.setDevMode(true);
```

配置成 devMode 模式，开发环境下是提升开发效率必须的配置。

5、共享模板函数配置

如果模板中通过 #define 指令定义了 template function，并且希望这些 template function 可以在其它模板中直接调用的话，可以进行如下配置：

```
// 添加共享函数，随后可在任意地方调用这些共享函数
me.addSharedFunction("/view/common/layout.html");
```

以上代码添加了一个共享函数模板文件 layout.html，这个文件中使用了#define指令定义了template function。通过上面的配置，可以在任意地方直接调用 layout.html 里头的 template function。

6、从 class path 和 jar 包加载模板配置

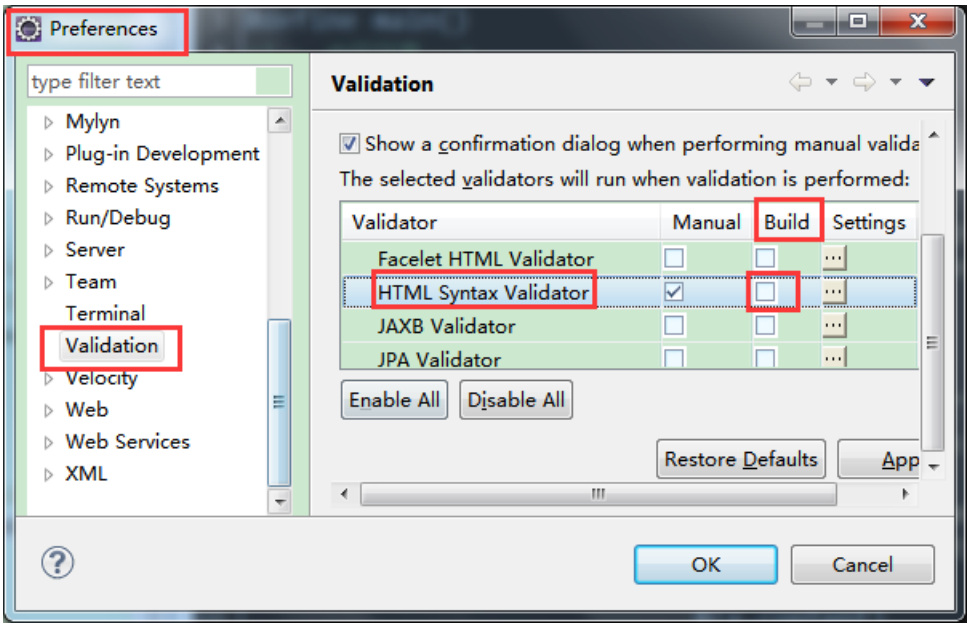
如果模板文件在项目的 class path 路径或者 jar 包之内，可以通过**me.setToClassPathSourceFactory()** 以及 me.setBaseTemplatePath(**null**) 来实现，以下是代码示例：

```
public void configEngine(Engine me) {
   me.setDevMode(true);

   me.setBaseTemplatePath(null);
   me.setToClassPathSourceFactory();

   me.addSharedFunction("/view/common/layout.html");
}
```

此外，还可以通过实现 ISourceFactory、ISource 扩展出从任何地方加载模板文件的功能，然后通过 me.setSourceFactory(...) 切换到自己的扩展上去。目前已有用户实现 DbSource 来从数据库加载模板的功能。

7、Eclipse 下开发

在 Eclipse 下开发时，可以将 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**。具体的配置方式见下图：



配置完成后注意**重启 eclipse 生效**，这个操作不影响使用，仅为了代码美观，关爱处女座完美主义者。

< 6.1 概述
6.3 表达式 >

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。

对于表达式的使用，**再次强调**一个关键点：表达式与Java是直接打通的。掌握了这个关键点立即就掌握了模板引擎90%的用法。如下是代码示例：

```
123 + "abc"
"abcd".substring(0, 1)
userList.get(0).getName()
```

以上代码第一、第二行，与Java表达式的用法完全一样。第三行代码中，假定userList中有User对象，并且User具有getName()方法，只要知道变量的类型，就可以像使用Java表达式一样调用对象的方法。

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)**
- 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)**

小技巧：如果从java端往map中传入一个key为中文的值，可以通过map["中文"] 的方式去访问到，而不能用 "map.中文" 访问。因为引擎会将之优先当成是object.field的访问形式，而目前引擎暂时还不支持中文作为变量名标识符。

2、属性访问

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：

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

此外，还支持数组的length长度访问：array.length，与java语言一样

最后，属性访问表达式还可以通过 FieldGetter 抽象类扩展，具体方法参考 com.jfinal.template.expr.ast.FieldGetters，这个类中已经给出了多个默认实现类，以下配置将支持 **user.girl** 表达式去调用 user 对象的 **boolean isGirl()** 方法：

```
Engine.addFieldGetterToLast(new com.jfinal.template.expr.ast.FieldGetters.IsMethodFieldGetter());
```

3、方法调用

模板引擎被设计成与 java 直接打通，可以在模板中直接调用对象上的任何public方法，使用规则与java中调用方式保持一致，以下代码示例：

```
#("ABCDE".substring(0, 3))
#(girl.getAge())
#(list.size())
#(map.get(key))
```

以上第一行代码调用了String对象上的substring(0, 3)方法输出值为 "ABC"。第二行代码在girl对象拥有getAge()方法时可调用。第三行代码假定map为一个Map类型时可调用其 get(...) 方法。

简单来说：模板表达式中可以直接调用对象所拥有的public方法，方法调用支持可变参数，例如支持这种方法被调用：obj.find(String sql, Object … args)。

对象方法调用与java直接打通式设计，学习成本为0、与java交互极其方便、并且立即拥有了非常强大的扩展机制。

4、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.demo.common.model.Account::STATUS_LOCK_ID)
   <span>(账号已锁定)</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。

由于静态属性访问需要包名前缀，代码显得比较长，在实际使用时如果多次用到同一个值，可以用 #set(STATUS_LOCK_ID = ...) 指令将常量值先赋给一个变量，可以节省一定的代码。

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。

5、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.kit.StrKit::isBlank(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。

此外，还可以调用静态属性上的方法，以下是代码示例：

```
(com.jfinal.MyKit::me).method(paras)
```

上面代码中需要先用一对小扩号将静态属性取值表达式扩起来，然后再去调用它的方法，小括号在此仅是为了改变表达式的优先级。

6、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&&

7、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。

8、相等与不等比较表达式增强

相等不等表达式 == 与 != 会对左右表达式进行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
```

注意：Controller.keepPara(...) 方法会将任何数据转换成String后传递到view层，所以原本可以用相等表达式比较的两个Integer型数据，在keepPara(...)后变得不可比较，因为变为了String与Integer型的比较。解决方法见本章的Extionsion Method小节。

9、布尔表达式增强

布尔表达式在原有java基础之下进行了增强，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
- String、StringBuilder等一切继承自 CharSequence 类的对象，返回 length > 0
- 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(user && user.id == x.userId)
  ...
#end
```

以上代码中的 user 表达式实质上代替了java表达式的 user != 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 (user.id ?? == x.userId)

10、范围数组定义表达式

直接举例：

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

上图中的表达式 [1..10] 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。

11、Map定义表达式

Map定义表达式的最实用场景是在调用方法或函数时提供极为灵活的参数传递方式，当方法或函数需要传递的参数名与数量不确定时极为有用，以下是基本用法：

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

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 只允许是合法的 java 变量名标识符或者 String 常量值（jfinal 3.4 起将支持 int、long、float、double、boolean、null 等等常量值），**注意：上例中使用了标识符 k1 而非 String 常量值 "k1" 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 k1 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map[key] 的方式进行取值，第六行则是与 java 表达式打通式的用法。

特别注意：上例代码如果使用 map[k1] 来取值，则会对 k1 标识符先求值，得到的是 null，也即map[k1] 相当于 map[null]，因此上述代码中使用了 map["k1"] 这样的形式来取值。

此外，map 取值还支持在定义的同时来取值，如下所示：

```
#({1:'自买', 2:'跟买'}.get(1))
#({1:'自买', 2:'跟买'}[2])

### 与双问号符联合使用支持默认值
#({1:'自买', 2:'跟买'}.get(999) ?? '其它')
```

上述 key 为 int 常量，自 jfinal 3.4 版本才开始支持。

12、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，逗号表达式整体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。

13、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。

14、表达式总结

以上各小节介绍的表达式用法，主要是在 java 表达式规则之上做的有利于开发体验的精心扩展，**你也可以先无视这些用法，而是直接当成是 java 表达式去使用，则可以免除掉上面的学习成本。**

上述这些在 java 表达式规则基础上做的精心扩展，一是基于模板引擎的实际使用场景而添加，例如单引号字符串。二是对过于啰嗦的 java 语法的改进，例如字符串的比较 str == "james" 取代 str.equals("james")，所以是十分值得和必要的。

< 6.2 引擎配置
6.4 指令 >

6.4 指令

Enjoy Template Engine一如既往地坚持极简设计，核心只有 #if、#for、#switch、#set、#include、#define、#(…) 这七个指令，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，Enjoy 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。

1、输出指令#( )

与几乎所有 java 模板引擎不同，Enjoy 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")
#(object.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，只需要继承 OutputDirectiveFactory 类并覆盖其中的 getOutputDirective 方法，然后在 configEngine(Engine me)方法中，通过 me. setOutputDirectiveFactory(…) 切换即可。

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 #else if 与 #else 分支块结构，以下是示例：

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

由于#else if、#else用法与java语法完全一样，在此不在赘述。（jf3.3版添加了对# else if风格的支持，也即else与if之间可以有空白字符）

3、#for 指令

Enjoy 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，不进行迭代。从而可以避免 if 判断，节省代码提高效率。

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示例：

```
#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
```

具体用法在上面代码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
```

上例中的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
   您还没有写过博客，点击此处<a href="/blog/add">开博</a>
#end
```

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，Enjoy 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。

4、#switch 指令（3.6 版本新增指令）

#switch 指令对标 java 语言的 switch 语句。基本用法一致，但做了少许提升用户体验的改进，用法如下：

```
#switch (month)
  #case (1, 3, 5, 7, 8, 10, 12)
    #(month) 月有 31 天
  #case (2)
    #(month) 月平年有28天，闰年有29天
  #default
    月份错误: #(month ?? "null")
#end
```

如上代码所示，#case 分支指令支持以逗号分隔的多个参数，这个功能就消解掉了 #break 指令的必要性，所以 enjoy 模板引擎是不需要 #break 指令的。

#case 指令参数还可以是任意表达式，例如：

```
#case (a, b, x + y, "abc", "123")
```

上述代码中用逗号分隔的表达式先会被求值，然后再逐一与 #switch(value) 指令中的 value 进行比较，只要有一个值与其相等则该 case 分支会被执行。

#case 支持逗号分隔的多参数，从而无需引入 #break 指令，不仅减少了代码量，而且避免了忘写 #break 指令时带来的错误隐患。还有一个与 java 语法有区别的地方是 #case、#default 指令都未使用冒号字符。

5、#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)  #(c)  #(array[0])  #(map.key)  #(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，其她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。

当需要明确指定在本层作用域赋值时，可以使用#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，用于实现模块化，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、#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 指令第一个参数必须为 String 常量，当以 ” /” 打头时将以 baseTemplatePath 为相对路径去找文件，否则将以使用 #include 指令的当前模板的路径为相对路径去找文件。

baseTemplatePath 可以在 configEngine(Engine me) 中通过 me.setBaseTemplatePath(…) 进行配置。

此外，include指令支持传入无限数量的赋值表达式，十分有利于模块化，例如：如下名为 ”_hot_list.html” 的模板文件用于展示热门项目、热门新闻等等列表：

```
<div class="hot-list">
  <h3>#(title)</h3>
  <ul>
    #for(x : list)
    <li>
      <a href="#(url)/#(x.id)">#(x.title)</a>
    </li>
    #end
  </ul>
</div>
```

上图中的 title、list、url 是该html片段需要的变量，使用include指令分别渲染“热门项目”与“热门新闻”的用法如下：

```
#include("_hot_list.html", title="热门项目", list=projectList, url="/project")
#include("_hot_list.html", title="热门新闻", list=newsList, url="/news")
```

上面两行代码中，为“_hot_list.html”中用到的三个变量title、list、url分别传入了不同的值，实现了对“_hot_list.html”的模块化重用。

7、#render 指令

render指令在使用上与include指令几乎一样，同样也支持无限量传入赋值表达式参数，主要有两点不同：

- render指令支持动态化模板参数，例如：#render(temp)，这里的temp可以是任意表达式，而#include指令只能使用字符串常量：#include(“abc.html”)
- render指令中#define定义的模板函数只在其子模板中有效，在父模板中无效，这样设计非常有利于模块化

引入 #render 指令的核心目的在于支持动态模板参数。

8、#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一个的 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() 表示调用另一个名为 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。

Enjoy Template Engine彻底消灭掉了layout、nested、macro这些无聊的概念，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。

9、模板函数调用与 #call 指令

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()
<html>
  <head>
    <link rel="stylesheet" type="text/css" href="/assets/css/jfinal.css">
    #@css?()
  </head>

  <body>
    <div class="content">
      #@main()
    </div>

    <script type="text/javascript" src="/assets/js/jfinal.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：#@css?() 与 #@js?() 就是模板函数安全调用。

上述模板中引入的 jfinal.css 与 jfinal.js 是两个必须的资源文件，对大部分模块已经满足需要，但对于有些模块，除了需要这两个必须的资源文件以外，还需要额外的资源文件，那么就可以通过#define css() 与 #define js() 来提供，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()
   这里可以引入额外的 css 内容
#end

#define js()
   这里可以引入额外的 js 内容
#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。

#call 指令是 jfinal 3.6 版本新增指令，使用 #call 指令，模板函数的名称与参数都可以动态指定，提升模板函数调用的灵活性，用法如下：

```
#call(funcName, p1, p2, ..., pn)
```

上述代码中的 funcName 为函数名，p1、p2、pn 为被调用函数所使用的参数。如果希望模板函数不存在时忽略其调用，添加常量值 true 在第一个参数位置即可：

```
#call(true, funcName, p1, p2, ..., pn)
```

10、#date 指令

date指令用于格式化输出日期型数据，包括Date、Timestamp等一切继承自Date类的对象的输出，使用方式极其简单：

```
#date(account.createAt)
#date(account.createAt, "yyyy-MM-dd HH:mm:ss")
```

上面的第一行代码只有一个参数，那么会按照默认日期格式进行输出，默认日期格式为："yyyy-MM-dd HH:mm"。上面第二行代码则会按第二个参数指定的格式进行输出。

如果希望改变默认输出格式，只需要通过engine.setDatePattern()进行配置即可。

11、#number 指令

number 指令用于格式化输出数字型数据，包括 Double、Float、Integer、Long、BigDecimal 等一切继承自Number类的对象的输出，使用方式依然极其简单：

```
#number(3.1415926, "#.##")
#number(0.9518, "#.##%")
#number(300000, "光速为每秒，### 公里。")
```

上面的 #number指令第一个参数为数字类型，第二个参数为String类型的pattern。Pattern参数的用法与JDK中DecimalFormat中pattern的用法完全一样。当不知道如何使用pattern时可以在搜索引擎中搜索关键字DecimalFormat，可以找到非常多的资料。

#number指令的两个参数可以是变量或者复杂表达式，上例参数中使用常量仅为了方便演示。

12、#escape 指令

escape 指令用于 html 安全转义输出，可以消除 XSS 攻击。escape 将类似于 html 形式的数据中的大于号、小于号这样的字符进行转义，例如将小于号转义成：< 将空格转义成

使用方式与输出指令类似：

```
#escape(blog.content)
```

13、指令扩展

由于采用独创的 DKFF 和 DLRD 算法，Enjoy 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 NowDirective 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", NowDirective.class) 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：

```
今天的日期是： #now()
```

除了支持上述无#end块，也即无指令body的指令外，Enjoy 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) {
    // 在这里可以对 exprList 进行个性化控制
    super.setExprList(exprList);
  }

  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;  // 返回 true 则该指令拥有 #end 结束标记
  }
}
```

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", Demo.class)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。

14、常见错误

Enjoy 模板引擎的使用过程中最常见的错误就是分不清 "表达式" 与 "非表达式"，所谓表达式是指模板函数调用、指令调用时**小括号里面**的所有东西，例如：

```
#directiveName(这里所有东西是表达式)

#@functionName(这里所有东西是表达式)
```

上例中的两行代码分别是调用指令与调用模板函数，小括号内的东西是表达式，而表达式的用法与 Java 几乎一样，该这么来用：

```
#directiveName( user.name )
```

最常见错误的用法如下：

```
#directiveName ( #(user.name) )
```

简单来说这种错误就是在该使用表达式的地方使用指令，**在表达式中永远不要出现字符 '#'，而是直接使用 java 表达式**。

< 6.3 表达式
6.5 注释 >

6.5 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，模板是极其自由化的内容，使用三个字符，冲突的概率降低一个数量级。

< 6.4 指令
6.6 原样输出 >

6.6 原样输出

原样输出是指不被解析，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，原样输出以 #[[三个字符打头，以]]# 三个字符结尾，中间被包裹的内容虽然是指令，但仍然被当成是纯文本，这非常有利于解决与前端javascript模板引擎的指令冲突问题。

无论是单行注释、多行注释，还是原样输出，都是以三个字符开头，目的都是为了降低与纯文本内容冲突的概率。

注意：用于注释、原样输出的三个控制字符之间不能有空格

< 6.5 注释
6.7 Shared Method 扩展 >

6.7 Shared Method 扩展

1、基本用法

Enjoy 模板引擎可以极其简单的直接使用任意的 java 类中的 public 方法，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

#if(notBlank(title))
   ...
#end
```

上例中的isBlank 和 notBlank 方法都来自于StrKit类，这种扩展方式简单、便捷、无耦合。

2、默认 Shared Method 配置扩展

Enjoy 模板引擎默认配置添加了 com.jfinal.template.ext.sharedmethod.SharedMethodLib 为 Shared Method，所以其中的方法可以直接使用不需要配置。里头有 isEmpty(...) 与 notEmpty(...) 两个方法可以使用。

isEmpty(...) 用来判断 Collection、Map、数组、Iterator、Iterable 类型对象中的元素个数是否为 0，其规如下：

- null 返回 true
- List、Set 等一切继承自 Collection 的，返回 isEmpty()
- Map 返回 isEmpty()
- 数组返回 length == 0
- Iterator 返回 ！hasNext()
- Iterable 返回 ！iterator().hasNext()

以下是代码示例：

```
#if ( isEmpty(list) )
    list 中的元素个数等于 0
#end

#if ( notEmpty(map) )
    map 中的元素个数大于 0
#end
```

如上所示，isEmpty(list) 判断 list 中的元素是否大于 0。notEmpty(...) 的功能与 isEmpty(...) 恰好相反，等价于！isEmpty(...)

< 6.6 原样输出
6.8 Shared Object扩展 >

6.8 Shared Object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：

```
<img src="#(RESOURCE_HOST)/img/girl.jpg" />
#if(sk.isBlank(title))
   ...
#end
```

以上代码第一行中使用输出指令输出了RESOUCE_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。

< 6.7 Shared Method 扩展
6.9 Extension Method扩展 >

6.9 Extension Method扩展

Extension Method 用于对已存在的类在其外部添加扩展方法，该功能类似于ruby语言中的mixin特性。

JFinal Template Engine 默认已经为String、Integer、Long、Float、Double、Short、Byte 这七个基本的 java 类型，添加了 toInt()、toLong()、toFloat()、toDouble()、toBoolean()、toShort()、toByte() 七个extension method。以下是使用示例：

```
#set(age = "18")
#if(age.toInt() >= 18)
   join to the party
#end
```

上例第一行代码中的age为String类型，由于String被添加了toInt()扩展方法，所以调用其toInt()方法后便可与后面的Integer型的值18进行比较运算。

Extension Method特性有两大应用场景，第一个应用场景是为java类库中的现存类进行功能性扩展，下面给出一个为Integer添加扩展方法的例子：

```
public class MyIntegerExt {
  public Integer square(Integer self) {
    return self * self;
  }

  public Double power(Integer self, Double exponent) {
    return Math.pow(self, exponent);
  }

  public Boolean isOdd(Integer self) {
    return self % 2 != 0;
  }
}
```

如上面代码所示，由于是对Integer进行扩展，所以上述三个扩展方法中的第一个参数必须是Integer类型，以便调用该方法时让这个参数承载调用者自身。其它参数可以是任意类型，例如上述power方法中的exponent参数。扩展方法至少要有一个参数。以下代码是对上述扩展方法进行配置：

```
Engine.addExtensionMethod(Integer.class, MyIntegerExt.class);
```

上述代码第一个参数Integer是被扩展的类，第二个参数MyIntegerExt是扩展类，通过上面简单的两步，即可在模板中使用：

```
#set(num = 123)
#(num.square())
#(num.power(10))
#(num.isOdd())
```

上述代码第二、三、四行调用了MyIntegerExt中的三个扩展方法，分别实现了对123的求平方、求10次方、判断是否为奇数的功能。

如上例所示，extension method可以在类自身以外的地方对其功能进行扩展，在模板中使用时的书写也变得非常方便。

Extension Method 的另一个重要的应用场景，是将不确定类型变得确定，例如Controller.keepPara()方法会将所有参数当成String的类型来keep住，所以表单中的Integer类型在keepPara()后变成了String型，引发模板中的类型不正确异常，以下是解决方案：

```
<select name="type">
  #for(x : list)
    <option value="#(x.type)" #if(x.type == type.toInt()) selected #end />
  #end
</select>
```

假定type为Integer类型，上面的 select域在表单提交后，如果在后端使用了keepPara()，那么再次渲染该模板时type会变为String类型，从而引发类型不正确异常，通过type.toInt()即可解决。当然，你也可以使用keepPara(Integer.class, “type”)

进行解决。

< 6.8 Shared Object扩展
6.10 Spring整合 >

6.10 Spring整合

1、maven 坐标

Spring 整合可以在pom.xml中配置 jfinal 坐标，也可以配置 Enjoy Template Engine 的独立发布版本坐标，其 maven 坐标如下：

```
<dependency>
  <groupId>com.jfinal</groupId>
  <artifactId>enjoy</artifactId>
  <version>3.6</version>
</dependency>
```

JFinal Template Engine 的独立发布版本 Enjoy 只有 207K 大小，并且无任何第三方依赖。

2、Spring MVC整合

在 Spring mvc下整合 Enjoy 非常简单，只需要配置一个 bean 即可，如下是具体配置方式：

```
<bean id="viewResolver" class="com.jfinal.template.ext.spring.JFinalViewResolver">
  <!-- 是否热加载模板文件 -->
  <property name="devMode" value="true"/>
  <!-- 配置shared function，多文件用逗号分隔 -->
  <property name="sharedFunction" value="/view/_layout.html, /view/_paginate.html"/>

  <!-- 是否支持以 #(session.value) 的方式访问 session -->
  <property name="sessionInView" value="true"/>
  <property name="prefix" value="/view/"/>
  <property name="suffix" value=".html"/>
  <property name="order" value="1"/>
  <property name="contentType" value="text/html; charset=utf-8"/>
</bean>
```

更多、更详细的配置项及其说明，可以通过查看 JFinalViewResolver 头部的注释来了解，在绝大部分情况下，上面的配置项可以满足需求。

3、Spring Boot整合

Spring boot 下整合配置如下：

```
@Configuration
public class SpringBootConfig {
  @Bean(name = "jfinalViewResolver")
  public JFinalViewResolver getJFinalViewResolver() {
    JFinalViewResolver jfr = new JFinalViewResolver();
    // setDevMode 配置放在最前面
    jfr.setDevMode(true);

    // 使用 ClassPathSourceFactory 从 class path 与 jar 包中加载模板文件
    jfr.setSourceFactory(new ClassPathSourceFactory());

    // 在使用 ClassPathSourceFactory 时要使用 setBaseTemplatePath
    // 代替 jfr.setPrefix("/view/")
    JFinalViewResolver.engine.setBaseTemplatePath("/view/");

    jfr.setSuffix(".html");
    jfr.setContentType("text/html;charset=UTF-8");
    jfr.setOrder(0);
    jfr.addSharedFunction("/view/common/_layout.html");
    jfr.addSharedFunction("/view/common/_paginate.html");
```

```
    return jfr;
  }
}
```

如上所示，jfr.setSourceFactory(...) 配置的 ClassPathSourceFactory 将从class path和jar包中加载模板文件。jfr.addSharedFunction(...) 配置共享模板函数。

如果从项目的web路径下加载模板文件则无需配置为 ClassPathSourceFactory。

< 6.9 Extension Method扩展
6.11 任意环境下使用Engine >

6.11 任意环境下使用Engine

Enjoy Template Engine 的使用不限于 web，可以使用在任何 java 开发环境中。Enjoy 常被用于代码生成、email 生成、模板消息生成等具有模板特征数据的应用场景，使用方式极为简单。

1、基本用法

直接举例：

```
Engine.use().getTemplate("demo.html").renderToString(Kv.by("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。

2、进阶用法

直接举例：

```
Engine engine = Engine.create("myEngine");
engine.setDevMode(true);
engine.setToClassPathSourceFactory();
Template template = engine.getTemplate("wxAppMsg.txt");
String wxAppMsg = template.renderToString(Kv.by("toUser", "james"));

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

上例第一行代码创建了名为 "myEngine" 的Engine对象，第二行代码设置了热加载模板文件，第三行代码设置引擎从 class path 以及 jar 包中加载模板文件，第四行代码利用wxAppMsg.txt这个模板创建一个Template对象，第五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 wirter)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(Kv.by("x", 456));

template = engine.getTemplate(new MySource());
result = template.renderToString(Kv.by("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 MySource()，MySource类是ISource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，ISource接口用法极其简单，在此不再赘述。

3、Engine对象管理

Engine对象的创建方式有两种，一种是通过 Engine.create(name) 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(name) 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(name) 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(name) 获取到，前者可以通过**RenderManager.me().getEngine()** 获取到，后者可以通过 **activeRecordPlugin.getEngine()** 获取到。

Engine对象管理的设计，允许在同一个应用程序中使用多个 Engine 实例用于不同的用途，JFinal 自身的 render、activerecord 模块对 Engine 的使用就是典型的例子。

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**

< 6.10 Spring整合
7.1 概述 >

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。

< 6.11 任意环境下使用Engine
7.2 EhCachePlugin >

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());
  }
}
```

< 7.1 概述
7.3 CacheInterceptor >

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cacheName，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"…>。

< 7.2 EhCachePlugin
7.4 EvictInterceptor >

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。

jfinal 3.6 版本支持 @CacheName 参数使用逗号分隔多个 cacheName，方便针对多个 cacheName 进行清除，例如：

```
@Before(EvictInterceptor.class)
@CacheName("blogList, hotBlogList")   // 逗号分隔多个 cacheName
public void update() {
   ...
}
```

< 7.3 CacheInterceptor
7.5 CacheKit >

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。

< 7.4 EvictInterceptor
7.6 ehcache.xml简介 >

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。

< 7.5 CacheKit
8.1 概述 >

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。

< 7.6 ehcache.xml简介
8.2 RedisPlugin >

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")

< 8.1 概述
8.3 Redis与Cache >

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

< 8.2 RedisPlugin
8.4 非web环境使用RedisPlugin >

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");
  }
}
```

< 8.3 Redis与Cache
9.1 概述 >

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。

< 8.4 非web环境使用RedisPlugin
9.2 Cron4jPlugin >

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 表达式最多只允许五部分，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 */5表示每隔5分钟执行一次（序列：0:00, 0:05, 0:10, 0:15 等等）。再例如3-18/5 * * * * 是指在从3到18分钟值这个范围之中每隔5分钟执行一次（序列：0:03, 0:08, 0:13, 0:18, 1:03, 1:08 等等）。

常见错误：cron4j在表达式中使用除号指定增加幅度时与linux稍有不同。例如在linux中表达式 10/3 * * * * 的含义是从第10分钟开始，每隔三分钟调度一次，而在cron4j中需要使用 10-59/3 * * * * 来表达。避免这个常见错误的技巧是：当需要使用除号指定增加幅度时，始终指定其范围。

基于上面的技巧，每隔2分钟调度一次的表达式为：0-59/2 * * * * 或者 */2 * * * *， 而不能是0/2 * * * *

以上规则不是JFinal创造，是linux通用的cron表达式规则**（注意不是quartz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。原始文档链接：http://www.sauronsoftware.it/projects/cron4j/manual.php

两大疑问：第一个疑问是当某个任务调度抛出了异常，那么这个任务在下次被调度的时间点上还会不会被调度，答案是肯定的，不管什么时候出现异常，时间一到调度仍然会被执行。

第二个疑问是假如某个任务执行时间很长，如果这个任务上次调度后直到本次调度到来的时候还没执行完，那么本次调度是否还会进行，答案也是肯定的。

总结一句话就是：每次调度都是独立的，上次调度是否抛出异常、是否执行完，都与本次调度无关。

特别提醒：Cron4jPlugin的cron表达式与linux一样只有5个部分，与quartz这个项目的7个部分不一样，但凡在网上搜索到的7部分cron表达式都不要试图应用在Cron4jPlugin之中。

< 9.1 概述
9.3 使用外部配置文件 >

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
```

上例中的cront4j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"。

上例中的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。

上例中的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、deamon、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。

< 9.2 Cron4jPlugin
9.4 高级用法 >

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:\\jdks\\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。

< 9.3 使用外部配置文件
10.1 概述 >

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。

< 9.4 高级用法
10.2 Validator >

10.2 Validator

1、基本用法

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 validator(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。同理，如果传递过来的是传统 java bean 对象，可以使用 **keepBean(...)** 方法来保持住用户输入过的数据。

keepPara(…) 方法默认将所有数据keep成String类型传给客户端，如果希望keep成为特定的类型，使用keepPara(Class, …) 即可，例如：keepPara(Integer.class, “age”)。

注意：如果keepPara() 造成模板中出现类型相关异常，解决方法参见Template Engine这章的Extension Method小节。

2、高级用法

虽然 Validator 中提供了丰富的 validateXxx(...) 系列方法，但毕竟方法个数是有限的，当 validateXxx(...) 系列的方法不能满足需求时，除了可以用 **validateRegex(...) 定制正则表达式**来满足需求以外，还可以通过普通 java 代码来实现，例如：

```
protected void validate(Controller c) {
    String nickName = c.getPara("nickName");
    if (userService.isExists(nickName)) {
        addError("msg", "昵称已被注册，请使用别的昵称！");
    }
}
```

如上代码所示，只需要利用普通的 java 代码配合一个 **addError(...)** 方法就可以无限制、灵活定制验证功能。

这里特别强调：**addError(...)** 方法是自由定制验证的关键。

此外，Validator 在碰到验证失败项时，默认会一直往下验证所有剩下的验证项，如果希望程序在碰到验证失败项时略过后续验证项立即返回，可以通过如下代码来实现：

```
protected void validate(Controller c) {
    this.setShortCircuit(true);
    ....
}
```

setShortCircuit(boolean) 用于设置验证方式是否为 “短路型验证”。

< 10.1 概述
10.3 Validator配置 >

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() {
    }
}
```

< 10.2 Validator
11.1 概述 >

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。

< 10.3 Validator配置
11.2 I18n与Res >

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");
```

以下是在 jfinal template engine 中的使用示例：

```
#(_res.get("msg"))
```

注意，上面的用法需要添加 I18nInterceptor，将在后面一小节进行介绍。

< 11.1 概述
11.3 I18nInterceptor >

11.3 I18nInterceptor

I18nInterceptor拦截器是针对于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 jfinal 模板引擎中即可通过 _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。

< 11.2 I18n与Res
12.1 概述 >

12.1 概述

jfinal 的 json 模块以抽象类 Json 为核心，方便扩展第三方实现，jfinal 官方给出了三个 Json 实现，分别是 JFinalJson、FastJson、Jackson，这三个实现继承自抽象类 Json。

抽象类 Json 的核心抽象如下：

```
public abstract class Json {
    public abstract String toJson(Object object);
    public abstract <T> T parse(String jsonString, Class <T> type);
}
```

如上代码可以看出 Json 抽象就是 Object 与 json string 互转的两个方法，toJson(...)将任意 java 类型转成 json string，而 parse 将 json string 再反向转成范型指定的对象。

< 11.3 I18nInterceptor
12.2 Json 配置 >

12.2 Json 配置

jfinal 官方提供了 Json 抽象类的三个实现：JFinalJson、FastJson、Jackson，如果不进行配置，那么默认使用 JFinalJson 实现，指定为其它实现需要在 configConstant 进行如下配置：

```
public void configConstant(Constants me) {
    me.setJsonFactory(new FastJsonFactory());
}
```

上面配置将系统默认使用的 JFinalJson 切换到了 FastJson。还可以通过扩展 Json 抽象类以及 JsonFactory 来实现定制的 Json 实现。

假定用户扩展出了一个 MyJson 与 MyJsonFactory，那么可以通过如下的方式切换到自己的实现上去：

```
public void configConstant(Constants me) {
    me.setJsonFactory(new MyJsonFactory());
}
```

此外，jfinal 官方还提供了 MixedJson、MixedJsonFactory 实现，这个实现让转 json string 时使用 JFinalJson，反向转成对象则使用 FastJson。

如果希望在非 web 下进行配置，需要使用 JsonManager，例如：

```
JsonManager.me().setDefaultJsonFactory(new MixedJsonFactory());
```

还可以配置 Date 类型转 json 后的格式：

```
public void configConstant(Constants me) {
    me.setJsonDatePattern("yyyy-MM-dd");
}
```

< 12.1 概述
12.3 Json 的四个实现 >

12.3 Json 的四个实现

jfinal 官方默认给出了三种 json 实现，可以满足绝大多数需求。

1、JFinalJson

JFinalJson 是 jfinal 官方最早的一个实现，这个实现最重要一点就是在转换 jfinal 的 Model 时是先获取 Model 中的 Map attrs 属性，然后再去转换这个 Map 对象。即便你的 Model 生成了 getter 方法，也不会被转换时调用。

针对 Model.attrs 属性进行转换而不是利用 getter 方法进行转换有如下几个原因：

A：支持多表关联查询结果的转换

无论是 Model 还是传统 Java Bean，其 getter 方法都是固定的，而多表关联查询的 sql 语句中的 select 中的字段是动态的，通常还包含关联表中的字段，而这些字段值没有相关的 getter 方法，这些字段就无法被转换

B：早期的 jfinal 用户没有为 Model 生成 getter 方法

注意：JFinalJson 只支持对象转 json string，不支持反向的 json string 转对象，可以通过使用 MixedJson 来支持反向转换：me.setJsonFactory(new MixedJsonFactory());

2、FastJson

FastJson 是对第三方的 fastjson 进行的二次封装，该实现最重要的一点就是转换依赖于 Model、java bean 的 getter 方法。使用 fastjson 可以按照其官方文档去配置 fastjson 的各种转换参数。

3、Jackson

该实现与 FastJson 类似，是对第三方的 jackson 的二次封装

4、MixedJson

MixedJson 是对 JFinalJson、FastJson 的再一次封装，Object 转 json string 时使用 JFinalJson 的实现，而反向 json string 转 Object 使用 FastJson。

这个实现结合了 JFinalJson 与 FastJson 两者的优势。 前者不支持 json string 到 Object 的转换，后者不支持关联表 sql 查询动态字段的转换。

< 12.2 Json 配置
12.4 Json 转换用法 >

12.4 Json 转换用法

json 转换在 jfinal 中的使用分为两类用法，第一类是使用配置的 json 转换，第二类是指定某个实现进行 json 转换。

1、使用配置的 json 实现转换

如下代码将使用前面章节中介绍的配置的 json 实现进行转的换：

```
// 在 Controller 中使用 renderJson 进行 json 转换，并渲染给客户端
renderJson();
renderJson(key, object);
renderJson(new String[]{...});

// 使用 JsonKit 工具类进行 json 转换
JsonKit.toJson(...);
JsonKit.parse(...);
```

2、使用指定的 json 实现转换

如果下代码将使用指定的 json 实现去转换：

```
// 临时指定使用 FastJson 实现
FastJson.getJson().toJson(...);
FastJson.getJson().parse(...);

// 为 Controller.renderJson(..) 方法直接传入转换好的 json string
renderJson(FastJson.getJson().toJson(...));
```

上面这种用法可以临时摆脱配置的 json 实现，从而使用指定的 json 实现。

< 12.3 Json 的四个实现
13.1 概述 >

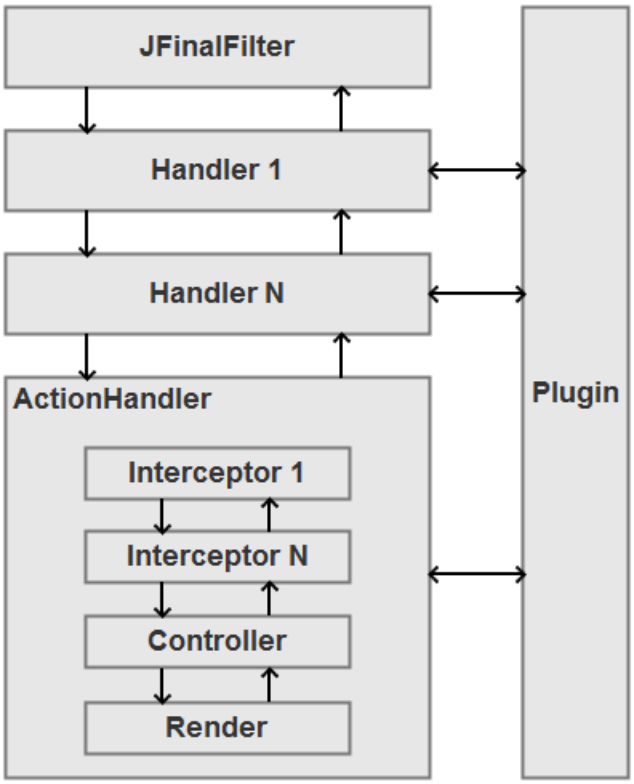
13.1 概述

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。

< 12.4 Json 转换用法
13.2 架构 >

13.2 架构

JFinal 顶层架构图如下：



< 13.1 概述
14.1 极速升级 >

14.1 极速升级

一、jfinal 3.0 之前版本的升级

jfinal 3.0 是大版本升级，此前版本升到 jfinal 3.0 请移步 14.2、14.3、14.4、14.5 小节，这几个小节中的内容极少，升级很方便。

二、jfinal 3.0 之后版本的升级

1、升级到 3.1

- 无需修改，平滑升级

2、升级到 3.2

- IStringSource 更名为 ISource
- 按照 14.2 小节 升级 Ret

3、升级到 3.3

- 指令扩展中的 java.io.Writer 改为 com.jfinal.template.io.Writer，eclipse/IDEA 开发工具会主动给出提示

4、升级到 3.4

- 由于 Json 中的 defaultDatePattern 初始值由 null 改为 "yyyy-MM-dd HH:mm:ss"，JFinalJson 中删掉 datePattern 属性，所以要在 configConstant(Constants me) 中配置：me.setJsonDatePattern(null) 或者具体值

5、升级到 3.5

- ISource.getKey() 更名为 ISource.getCacheKey()

6、升级到 3.6

- Db、Model 针对多主键(联合主键)的 findById、deleteById 方法添加一个 's' 后缀，改成 findByIds、deleteByIds
- 用到 jfinal weixin 项目的 MsgController 时，需要在 configRoutes 中配置 me.setMappingSuperClass(true)
- 由于 jfinal 3.6 用于 sql 管理的 Engine 对象，默认配置了 engine.setToClassPathSourceFactory()，jfinal 将从 class path 和 jar 包中加载 sql 文件，所以如果 ActiveRecordPlugin arp 对象配置过 arp.setBaseSqlTemplatePath(...)，那么需要做出相应变化，例如：arp.setBaseSqlTemplatePath(null)。通过 arp.getEngine().setBaseTeplatePath(...) 配置过的也照此办理。

< 13.2 架构
14.2 Ret >

14.2 Ret

如果待升级项目中未使用过 Ret，那么可以忽略本小节。如果是 jfinal 3.2、3.3、3.4 或者更高版本的 jfinal 也可以忽略。

JFinal 3.2 对Ret工具类进行了改进，使其更加适用于json数据格式交互的API类型项目。新版本状态名只有一个：state，取值为：ok/fail，而老版本状态名有两个：isOk与isFail，取值为：true/false。

所以，新旧版本Ret对象生成的json数据会有差异，对于大多数web项目来说，升级方法如下：

- 利用查找替换功能将 html 与js文件中的 ret.isOk 替换为 ret.state == "ok"
- 利用查找替换功能将 html 与js文件中的 ret.isFail 替换为 ret.state == "fail"

如果希望尽可能小的改动代码进行升级，可以调用一次Ret.setToOldWorkMode()方法沿用老版本模式即可。

< 14.1 极速升级
14.3 configEngine >

14.3 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.setDevMode(true);

  me.addSharedFunction("/view/common/layout.html");
  me.addSharedFunction("/view/common/paginate.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。

< 14.2 Ret
14.4 baseViewPath >

14.4 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：

```
public class FrontRoutes extends Routes {

  public void config() {
    setBaseViewPath("/_view");

    add("/", IndexController.class, "/index");
    add("/project", ProjectController.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);
}
```

< 14.3 configEngine
14.5 RenderFactory >

14.5 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类中添加上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。

< 14.4 baseViewPath
14.6 其它 >

14.6 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。

JFinal 官方QQ群: 用”jfinal”关键字搜索QQ群

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**

< 14.5 RenderFactory