# Eclipse

## 快捷键

Eclipse快捷键，熟悉快捷键可以帮助开发事半功倍，节省更多的时间来用于做有意义的事情。Ctrl+Shift+L 查看所有快捷键说明

Ctrl+1 快速修复(最经典的快捷键,就不用多说了)

Ctrl+D: 删除当前行

Ctrl+Alt+↓ 复制当前行到下一行(复制增加)

Ctrl+Alt+↑ 复制当前行到上一行(复制增加)

Alt+↓ 当前行和下面一行交互位置(特别实用,可以省去先剪切,再粘贴了)

Alt+↑ 当前行和上面一行交互位置(同上)

Alt+← 前一个编辑的页面

Alt+→ 下一个编辑的页面(当然是针对上面那条来说了)

Alt+Enter 显示当前选择资源(工程,or 文件 or文件)的属性

Shift+Enter 在当前行的下一行插入空行(这时鼠标可以在当前行的任一位置,不一定是最后)

Shift+Ctrl+Enter 在当前行插入空行(原理同上条)

Ctrl+Q 定位到最后编辑的地方

Ctrl+L 定位在某行 (对于程序超过100的人就有福音了)

Ctrl+M 最大化当前的Edit或View (再按则反之)

Ctrl+/ 注释当前行,再按则取消注释

Ctrl+O 快速显示 OutLine

Ctrl+T 快速显示当前类的继承结构

Ctrl+W 关闭当前Editer

Ctrl+K 参照选中的Word快速定位到下一个

Ctrl+E 快速显示当前Editer的下拉列表(如果当前页面没有显示的用黑体表示)

Ctrl+/(小键盘) 折叠当前类中的所有代码

Ctrl+×(小键盘) 展开当前类中的所有代码

Ctrl+Space 代码助手完成一些代码的插入(但一般和输入法有冲突,可以修改输入法的热键,也可以暂用Alt+/来代替)

Ctrl+Shift+E 显示管理当前打开的所有的View的管理器(可以选择关闭,激活等操作)

Ctrl+J 正向增量查找(按下Ctrl+J后,你所输入的每个字母编辑器都提供快速匹配定位到某个单词,如果没

有,则在stutes line中显示没有找到了,查一个单词时,特别实用,这个功能Idea两年前就有了)

Ctrl+Shift+J 反向增量查找(和上条相同,只不过是从后往前查)

Ctrl+Shift+F4 关闭所有打开的Editer

Ctrl+Shift+X 把当前选中的文本全部变为大写

Ctrl+Shift+Y 把当前选中的文本全部变为小写

Ctrl+Shift+F 格式化当前代码

Ctrl+Shift+P 定位到对于的匹配符(譬如{}) (从前面定位后面时,光标要在匹配符里面,后面到前面,则反之)

下面的快捷键是重构里面常用的,本人就自己喜欢且常用的整理一下(注:一般重构的快捷键都是Alt+Shift开头的了)

Alt+Shift+R 重命名 (是我自己最爱用的一个了,尤其是变量和类的Rename,比手工方法能节省很多劳动力)

Alt+Shift+M 抽取方法 (这是重构里面最常用的方法之一了,尤其是对一大堆泥团代码有用)

Alt+Shift+C 修改函数结构(比较实用,有N个函数调用了这个方法,修改一次搞定)

Alt+Shift+L 抽取本地变量( 可以直接把一些魔法数字和字符串抽取成一个变量,尤其是多处调用的时候)

Alt+Shift+F 把Class中的local变量变为field变量 (比较实用的功能)

Alt+Shift+I 合并变量(可能这样说有点不妥Inline)

Alt+Shift+V 移动函数和变量(不怎么常用)

Alt+Shift+Z 重构的后悔药(Undo)

1编辑

编辑

作用域

功能

快捷键

全局

查找并替换

Ctrl+F

文本编辑器

查找上一个

Ctrl+Shift+K

文本编辑器

查找下一个

Ctrl+K

文本编辑器

删除当前行

Ctrl+D

文本编辑器

当前行的下一行插入空行

Shift+Enter

文本编辑器

当前行插入空行

Ctrl+Shift+Enter

文本编辑器

定位到最后编辑的位置

Ctrl+Q

全局

恢复上一个选择

Alt+Shift+↓

全局

快速修正

Ctrl+1

全局

内容辅助（代码提示）

Alt+/

全局

全部选中

Ctrl+A

全局

删除

Delete

全局

上下文信息

Alt+/

Alt+Shift+?

Ctrl+Shift+Space

Java编辑器

显示工具提示描述

F2

Java编辑器

选择封装元素

Alt+Shift+↑

Java编辑器

增量选择上一个同级元素

Alt+Shift+←

Java编辑器

增量选择下一个同级元素

Alt+Shift+→

文本编辑器

增量查找

Ctrl+J

文本编辑器

增量逆向查找

Ctrl+Shift+J

java编辑器

自动生成get set方法

Alt+Shift+s 再按 r

java编辑器

列出所有实现此接口的类

ctrl+T

2查看

编辑

作用域

功能

快捷键

全局

放大

Ctrl+=

全局

缩小

Ctrl+-

3窗口

编辑

作用域

功能

快捷键

全局

激活编辑器

F12

全局

关闭所有编辑器

Ctrl+Shift+W

全局

上一个编辑器

Ctrl+Shift+F6

全局

上一个视图

Ctrl+Shift+F7

全局

上一个透视图

Ctrl+Shift+F8

全局

下一个b编辑器

Ctrl+F6

全局

下一个视图

Ctrl+F7

全局

下一个透视图

Ctrl+F8

文本编辑器

关闭当前窗口

Ctrl+W

全局

显示视图菜单

Ctrl+F10

全局

显示系统菜单

Alt+-

4导航

编辑

作用域

功能

快捷键

Java编辑器

打开结构

Ctrl+F3

全局

打开类型

Ctrl+Shift+T

全局

打开类型层次结构

F4

全局

打开声明

F3

全局

打开外部javadoc

Shift+F2

全局

打开资源

Ctrl+Shift+R

全局

后退历史记录

Alt+←

全局

前进历史记录

Alt+→

全局

上一个

Ctrl+,

全局

下一个

Ctrl+.

Java编辑器

显示大纲

Ctrl+O

全局

在层次结构中打开类型

Ctrl+Shift+H

全局

转至匹配的括号

Ctrl+Shift+P

全局

转至上一个编辑位置

Ctrl+Q

Java编辑器

转至上一个成员

Ctrl+Shift+↑

Java编辑器

转至下一个成员

Ctrl+Shift+↓

文本编辑器

转至行

Ctrl+L

5搜索

编辑

作用域

功能

快捷键

全局

出现在文件中

Ctrl+Shift+U

全局

查找目标文件

ctrl+shift+R

全局

打开搜索对话框

Ctrl+H

全局

工作区中的声明

Ctrl+G

全局

工作区中的引用

Ctrl+Shift+G

工作区域的类

查看某一个类的继承类或者实现类

ctrl+T

6文本编辑

编辑

作用域

功能

快捷键

文本编辑器

改写切换

Insert

文本编辑器

上滚行

Ctrl+↑

文本编辑器

下滚行

Ctrl+↓

7文件

编辑

作用域

功能

快捷键

全局

保存

Ctrl+S

全局

打印

Ctrl+P

全局

关闭

Ctrl+F4

全局

全部保存

Ctrl+Shift+S

全局

全部关闭

Ctrl+Shift+F4

全局

属性

Alt+Enter

全局

新建

Ctrl+N

8项目

编辑

作用域

功能

快捷键

全局

全部构建

Ctrl+B

9源代码

编辑

作用域

功能

快捷键

Java编辑器

格式化

Ctrl+Shift+F

Java编辑器

添加/取消注释

Ctrl+/

Java编辑器

添加导入

Ctrl+Shift+M

Java编辑器

组织导入

Ctrl+Shift+O

Java编辑器

使用try/catch块来包围

未设置，太常用了，所以在这里列出，建议自己设置。

也可以使用Ctrl+1自动修正。

Java编辑器

将所选区域字母设置为小写

Ctrl+Shift+Y

Java编辑器

将所选区域字母设置为大写

Ctrl+Shift+X

Java编辑器

方法添加注释

Alt+Shift+J

10运行

编辑

作用域 功能 快捷键

全局 单步返回 F7

全局 单步执行 F6

全局 单步跳入 F5

全局 单步跳入选择 Ctrl+F5

全局 调试上次启动 F11

全局 继续 F8

全局 使用过滤器单步执行 Shift+F5

全局 添加/去除断点 Ctrl+Shift+B

全局 显示 Ctrl+D

全局 运行上次启动 Ctrl+F11

全局 运行至行 Ctrl+R

全局 执行 Ctrl+U

11重构 编辑

作用域 功能 快捷键

全局 撤销重构 Alt+Shift+Z

全局 抽取方法 Alt+Shift+M

全局 抽取局部变量 Alt+Shift+L

全局 内联 Alt+Shift+I

全局 移动 Alt+Shift+V

全局 重命名 Alt+Shift+R

全局 重做 Alt+Shift+Y

切换控制台 Alt+Shift+Q

12修改快捷键

编辑

依次打开window->preferrence->general->keys,在 binding一栏选择修改快捷键。

## Eclipse右键New菜单项的自定义设置

大家都知道一般在Eclipse中新建一个项目或者文件都是通过菜单项的File-New来创建，然而有些右键出来的选项可能从来都不会使用到，而有些可能会常用到但是右键中又没有，这个时候就可以自定义New中的右键选项，具体设置方式如下：

**Window --> Customize Perspective... --> Shortcuts --> 选择"New"**  进行相关设置即可(如下图)



## web项目run as 菜单中没有 run on server

eclipse中已安装好tomcat，某maven web项目原本也可以在tomcat中运行，突然发现右击该项目的 run as 菜单中，没有了 run on serven 选项，项目也无法在tomcat中跑了。

原因是该项目的Dynamic Web Module 被我不经意间删除了，如下图。MyEclipse中的Web项目导入到Eclipse后，也会出现这样的情况。主要把Dynamic Web Module勾上即可。



有时勾选Dynamic Web Module 时会出现这个错误提示：Dynamic web module 3.0 requires java 1.6 or newer



解决这个错误的办法是：在项目的pom.xml的<build></build>标签中加入：

1. **<plugins>**
2. **<plugin>**
3. **<groupId>**org.apache.maven.plugins**</groupId>**
4. **<artifactId>**maven-compiler-plugin**</artifactId>**
5. **<version>**2.3.2**</version>**
6. **<configuration>**
7. **<source>**1.6**</source>**
8. **<target>**1.6**</target>**
9. **</configuration>**
10. **</plugin>**
11. **</plugins>**

保存，项目构建完成后在项目文件夹上点右键，选择Maven->Update Project Configuration，问题解决。

## 为什么eclipse新建项目的默认编码为何是GBK,而不是UTF-8

我们用的操作系统是中文操作系统，windows操作系统的编码是GBK，所以eclipse默认的新项目的编码是GBK的，你在Window->Preferences->General->Workspace->Text file encoding 将其改为UFT-8即可，同样，如果你想修改html,css等文件的编码格式，你在Window->Preferences->General->Content Types中可以修改

## 如何删除Eclipse弃用的工作空间？

在 eclipse\configuration\.settings\org.eclipse.ui.ide.prefs 文件里找到相应的空间，删除即可。

## eclipse因引用本地的JDK而不是JRE。



## Eclipse中安装Tomcat

1、在本地安装并配置好tomcat

2、新建一个Web项目，右击该项目，选择Run as -> run on server.

3、在弹出的run on server框上选择要安装的tomcat版本







## 将MyEclipse的Web项目导入到Eclipse

将一个MyEclipse的Web项目导入到Eclipse后，项目并没有报错，奇怪的是，右击该项目，选择run as菜单，此时并没有出现run on server 选项。项目结构如下：



解决的办法如下：

1、为项目添加Dynamic Web Module



2、添加Dynamic Web Module后，run as菜单出现run on server了。执行run on server，在tomcat上运行该项目，结果切报错。



3、原来添加Dynamic Web Module后，项目结构中多了一个WebContent目录。Myeclipse中默认的Web目录是WebRoot，而Eclipse中默认的Web目录是Webcontent，因此报错。



4、在项目属性中修改项目的部署目录。添加Dynamic Web Module后项目部署目录如下：



删除WebContent目录，然后点击Add按钮添加Folder，让Folder指向WebRoot





5、在项目的目录结构中，删除WebContent目录，删除后如下：



6、重新执行run on server即可。

## maven项目导入Eclipse

maven web 项目导入Eclipse后，执行maven build 命令，编译运行报错，找不到配置文件。

原因是导入项目的部署路径没有设置好，右击项目选择属性，按下图设置好即可。



## Eclipse中Project Explorer和Package Explorer这两个有什么区别

Package Explorer不会显示修改某些文件后会自动更改的文件，如class文件，在web工程里web-inf下的classes文件夹在Package Explorer下会隐藏  
  
Project Explorer会显示所有文件

## 修改无效

有时会出现修改某个类或某个配置文件，出现修改无效的情况，而程序上有没发现其他错误。这时可以试下把WEB-INF/classes 目录下对应的文件删除，再重新编译就可以了。

# idea

## 快捷键

最近刚接触IntelliJ这个工具，用了几天，很多操作都不太熟。之前一直用Eclipse，猛的一用这个工具还真有点不习惯。但是这个工具有些方面确实比较优秀，比如javascript自动补全功能、html页面编写功能等，相比于其他IDE工具就会好很多。下面是一些关于本工具的快捷键。但有些不是太常用，仅供参考。其中红色部分表示常用的。

Alt+回车 导入包,自动修正  
Ctrl+N   查找类  
Ctrl+Shift+N 查找文件  
Ctrl+Alt+L  格式化代码

Ctrl+Alt+O 优化导入的类和包  
Alt+Insert 生成代码(如get,set方法,构造函数等)  
Ctrl+E或者Alt+Shift+C  最近更改的代码  
Ctrl+R 替换文本

Ctrl+F 查找文本

CTRL+SHIFT+F     在项目中查找文本  
Ctrl+Shift+Space 自动补全代码  
Ctrl+空格 代码提示

Ctrl+Alt+Space 类名或接口名提示

Ctrl+P 方法参数提示

Ctrl+Shift+Alt+N 查找类中的方法或变量

Alt+Shift+C 对比最近修改的代码

Shift+F6  重构-重命名  
Ctrl+Shift+先上键  
Ctrl+X 删除行  
Ctrl+D 复制行  
Ctrl+/ 或 Ctrl+Shift+/  注释（// 或者 ）  
Ctrl+J  自动代码  
Ctrl+E 最近打开的文件

Ctrl+H 显示类结构图

Ctrl+Q 显示注释文档  
Alt+F1 查找代码所在位置  
Alt+1 快速打开或隐藏工程面板

Ctrl+Alt+ left/right 返回至上次浏览的位置  
Alt+ left/right 切换代码视图

Alt+ Up/Down 在方法间快速移动定位

Ctrl+Shift+Up/Down 代码向上/下移动。

F2 或Shift+F2 高亮错误或警告快速定位

代码标签输入完成后，按Tab，生成代码。

选中文本，按Ctrl+Shift+F7 ，高亮显示所有该文本，按Esc高亮消失。

Ctrl+W 选中代码，连续按会有其他效果

选中文本，按Alt+F3 ，逐个往下查找相同文本，并高亮显示。

Ctrl+Up/Down 光标跳转到第一行或最后一行下

Ctrl+B 快速打开光标处的类或方法

下面是快捷键大全，如果在上面的常用区找不到的内容可以试试到这个区来看看：

1、写代码时用Alt-Insert（Code|Generate…）可以创建类里面任何字段的getter与setter方法。  
  
2、右键点击断点标记（在文本的左边栏里）激活速查菜单，你可以快速设置enable/disable断点或者条件它的属性。  
  
3、CodeCompletion（代码完成）属性里的一个特殊的变量是，激活Ctrl-Alt-Space可以完成在或不在当前文件里的类名。如果类没有引入则import标志会自动创建。  
  
4、使用Ctrl-Shift-V快捷键可以将最近使用的剪贴板内容选择插入到文本。使用时系统会弹出一个含有剪贴内容的对话框，从中你可以选择你要粘贴的部分。  
  
5、利用CodeCompletion（代码完成）属性可以快速地在代码中完成各种不同地语句，方法是先键入一个类名地前几个字母然后再用Ctrl-Space完成全称。如果有多个选项，它们会列在速查列表里。  
  
6、用Ctrl-/与Ctrl-Shift-/来注释/反注释代码行与代码块。  
  
-/用单行注释标记（“//…”）来注释/反注释当前行或者选择地代码块。而Ctrl-Shift-/则可以用块注释标记（“”）把所选块包围起来。要反注释一个代码块就在块中任何一个地方按Ctrl-Shift-/即可。  
  
7、按Alt-Q（View|Context Info）可以不需要移动代码就能查看当前方法地声明。连续按两次会显示当前所编辑的类名。  
  
8、使用Refactor|Copy Class…可以创建一个所选择的类的“副本”。这一点很有用，比如，在你想要创建一个大部分内容都和已存在类相同的类时。  
  
9、在编辑器里Ctrl-D可以复制选择的块或者没有所选块是的当前行。  
  
10、Ctrl-W（选择字）在编辑器里的功能是先选择脱字符处的单词，然后选择源代码的扩展区域。举例来说，先选择一个方法名，然后是调用这个方法的表达式，然后是整个语句，然后包容块，等等。  
  
11、如果你不想让指示事件细节的“亮球”图标在编辑器上显示，通过按Alt-Enter组合键打开所有事件列表然后用鼠标点击它就可以把这个事件文本附件的亮球置成非活动状态。  
  
这样以后就不会有指示特殊事件的亮球出现了，但是你仍然可以用Alt-Enter快捷键使用它。  
  
12、在使用CodeCompletion时，可以用逗点（.）字符，逗号（，）分号（；），空格和其它字符输入弹出列表里的当前高亮部分。选择的名字会随着输入的字符自动输入到编辑器里。  
  
13、在任何工具窗口里使用Escape键都可以把焦点移到编辑器上。  
  
Shift-Escape不仅可以把焦点移到编辑器上而且还可以隐藏当前（或最后活动的）工具窗口。  
  
F12键把焦点从编辑器移到最近使用的工具窗口。  
  
14、在调试程序时查看任何表达式值的一个容易的方法就是在编辑器中选择文本（可以按几次Ctrl-W组合键更有效地执行这个操作）然后按Alt-F8。  
  
15、要打开编辑器脱字符处使用的类或者方法Java文档的浏览器，就按Shift-F1（右键菜单的External JavaDoc）。  
  
要使用这个功能须要把加入浏览器的路径，在“General”选项中设置（Options | IDE Settings），另外还要把创建的Java文档加入到工程中（File | Project Properties）。  
  
16、用Ctrl-F12（View | File Structure Popup）键你可以在当前编辑的文件中快速导航。  
  
这时它会显示当前类的成员列表。选中一个要导航的元素然后按Enter键或F4键。要轻松地定位到列表中的一个条目，只需键入它的名字即可。  
  
17、在代码中把光标置于标记符或者它的检查点上再按Alt-F7（右键菜单中的Find Usages…）会很快地查找到在整个工程中使用地某一个类、方法或者变量的位置。  
  
18、按Ctrl-N（Go to | Class…）再键入类的名字可以快速地在编辑器里打开任何一个类。从显示出来的下拉列表里选择类。  
  
同样的方法你可以通过使用Ctrl-Shift-N（Go to | File…）打开工程中的非Java文件。  
  
19、要导航代码中一些地方使用到的类、方法或者变量的声明，把光标放在查看项上再按Ctrl-B即可。也可以通过按Ctrl键的同时在查看点上单击鼠标键调转到声明处。  
  
20、把光标放到查看点上再按Ctrl-Alt-B可以导航到一个抽象方法的实现代码。  
  
21、要看一个所选择的类的继承层次，按Ctrl-H（Browse Type Hierarchy）即可。也可以激活编辑器中的继承关系视图查看当前编辑类的继承关系。  
  
22、使用Ctrl-Shift-F7（Search | Highlight Usages in File）可以快速高亮显示当前文件中某一变量的使用地方。按Escape清除高亮显示。  
  
23、用Alt-F3（Search | Incremental Search）在编辑器中实现快速查查找功能。  
  
在“Search for:”提示工具里输入字符，使用箭头键朝前和朝后搜索。按Escape退出。  
  
24、按Ctrl-J组合键来执行一些你记不起来的Live Template缩写。比如，键“it”然后按Ctrl-J看看有什么发生。  
  
25、Introduce Variable整合帮助你简化代码中复杂的声明。  
  
26、Ctrl-Shift-J快捷键把两行合成一行并把不必要的空格去掉以匹配你的代码格式。  
  
27、Ctrl-Shift-Backspace（Go to | Last Edit Location）让你调转到代码中所做改变的最后一个地方。  
  
多按几次Ctrl-Shift-Backspace查看更深的修改历史。  
  
28、用Tools | Reformat Code…根据你的代码样式参考（查看Options | IDE Setting | Code Style）格式化代码。  
  
使用Tools | Optimize Imports…可以根据设置（查看Options | IDE Setting | Code Style | Imports）自动“优化”imports（清除无用的imports等）。  
  
29、使用IDEA的Live Templates | Live Templates让你在眨眼间创建许多典型代码。  
  
30、要查看一个文件中修改的本地历史，激活右键菜单里的Local VCS | Show History…。也许你可以导航不同的文件版本，看看它们的不同之处再回滚到以前的任何一个版本吧。  
  
使用同样的右键菜单条目还可以看到一个目录里修改的历史。有了这个特性你就不会丢失任何代码了。  
  
31、如果要了解主菜单里每一个条目的用途，把鼠标指针移到菜单条目上再应用程序框架的底部的状态栏里就会显示它们的一些简短描述，也许会对你有帮助。  
  
32、要在编辑器里显示方法间的分隔线，打开Options | IDE Settings | Editor，选中“Show method separators”检查盒（checkbox）。  
  
33、用Alt-Up和Alt-Down键可以在编辑器里不同的方法之间快速移动。  
  
34、用F2/Shift-F2键在高亮显示的语法错误间跳转。  
  
用Ctrl-Alt-Down/Ctrl-Alt-Up快捷键则可以在编译器错误信息或者查找操作结果间跳转。  
  
35、通过按Ctrl-O（Code | Override Methods…）可以很容易地重载基本类地方法。  
  
要完成当前类implements的（或者抽象基本类的）接口的方法，就使用Ctrl-I（Code | Implement Methods…）。  
  
36、如果光标置于一个方法调用的括号间，按Ctrl-P会显示一个可用参数的列表。  
  
37、要快速查看编辑器脱字符处使用的类或方法的Java文档，按Ctrl-Q（在弹出菜单的Show Quick JavaDoc里）即可。  
  
38、像Ctrl-Q（Show Quick JavaDoc显示简洁Java文档），Ctrl-P（Show Parameter Info显示参数信息），Ctrl-B（Go to Declaration跳转到声明），Shift-F1（External JavaDoc外部Java文档）以及其它一些快捷键不仅可以在编辑器里使用，也可以应用在代码完成右键列表里。  
  
39、Ctrl-E（View | Recent Files）弹出最近访问的文件右键列表。选中文件按Enter键打开。  
  
40、在IDEA中可以很容易地对你的类，方法以及变量进行重命名并在所有使用到它们的地方自动更正。  
  
试一下，把编辑器脱字符置于任何一个变量名字上然后按Shift-F6（Refactor | Rename…）。在对话框里键入要显示地新名字再按Enter。你会浏览到使用这个变量地所有地方然后按“Do Refactor”按钮结束重命名操作。  
  
41、要在任何视图（Project View工程视图，Structure View结构视图或者其它视图）里快速选择当前编辑地部分（类，文件，方法或者字段），按Alt-F1（View | Select in…）。  
  
42、在“new”字符后实例化一个已知类型对象时也许你会用到SmartType代码完成这个特性。  
  
43、通过使用SmartType代码完成，在IDEA中创建接口的整个匿名implementation也是非常容易的，比如，对于一些listener（监听器），可以键入  
  
Component component;  
  
component.addMouseListener(  
  
new <caret is here>  
  
);  
  
然后再按Ctrl-Shift-Space看看有什么发生了。  
  
44、在你需要设置一个已知类型的表达式的值时用SmartType代码完成也很有帮助。比如，键入  
  
String s = (<caret is here>  
  
再按Ctrl-Shift-Space看看会有什么出现。  
  
45、在所有视图里都提供了速查功能：在树里只需键入字符就可以快速定位到一个条目。  
  
46、当你想用代码片断捕捉异常时，在编辑器里选中这个片断，按Ctrl-Alt-T（Code | Surround with…）然后选择“try/catch”。它会自动产生代码片断中抛出的所有异常的捕捉块。在Options | File Templates | Code tab中你还可以自己定制产生捕捉块的模板。  
  
44、在你需要设置一个已知类型的表达式的值时用SmartType代码完成也很有帮助。比如，键入  
  
String s = (<caret is here>  
  
再按Ctrl-Shift-Space看看会有什么出现。  
  
45、在所有视图里都提供了速查功能：在树里只需键入字符就可以快速定位到一个条目。  
  
46、当你想用代码片断捕捉异常时，在编辑器里选中这个片断，按Ctrl-Alt-T（Code | Surround with…）然后选择“try/catch”。它会自动产生代码片断中抛出的所有异常的捕捉块。在Options | File Templates | Code tab中你还可以自己定制产生捕捉块的模板。  
  
44、在你需要设置一个已知类型的表达式的值时用SmartType代码完成也很有帮助。比如，键入  
  
String s = (<caret is here>  
  
再按Ctrl-Shift-Space看看会有什么出现。  
  
45、在所有视图里都提供了速查功能：在树里只需键入字符就可以快速定位到一个条目。  
  
46、当你想用代码片断捕捉异常时，在编辑器里选中这个片断，按Ctrl-Alt-T（Code | Surround with…）然后选择“try/catch”。它会自动产生代码片断中抛出的所有异常的捕捉块。在Options | File Templates | Code tab中你还可以自己定制产生捕捉块的模板。  
  
47、在使用代码完成时，用Tab键可以输入弹出列表里的高亮显示部分。  
  
不像用Enter键接受输入，这个选中的名字会覆盖掉脱字符右边名字的其它部分。这一点在用一个方法或者变量名替换另一个时特别有用。

## 显示Maven Projects窗口

在Help的Find Action中搜索Maven，应该是第一个

在View里第一个Tool Windows里找Maven

## 将普通文件夹设置为package目录

发现所建的maven web 项目的java目录，只是普通的文件夹，而不是java的包目录，需要将其设置成包目录。右击该文件夹 -> Mark Directory as -> Source Root。

## 编辑界面失焦

如果打开了两个idea，则切换到另一个idea，用鼠标点击另一个idea的编辑界面使聚焦，再切回原来的idea即可。

## 在idea中更新git代码

代码提交：VCS -> Commit Changes

代码更新：VCS -> Update Project

## 设置idea的tomcat热加载



## tomcat控制面板消失



# tomcat

## tomcat的安装与配置

**以tomcat7.0为例，Tomcat 7.0 （请到官网下载）的免安装版的配置(假如将Tomcat 解压到C:\Program Files目录，目录结构为：C:\Program Files\apache-tomcat-7.0.39。**

**tomcat的安装目录最好在某个盘的根目录下，路径不能带中文、空格或符号，例如：D:\tomcat7。以免在运行过程中产生其他问题。**

**1、添加环境变量：在 我的电脑->属性->高级->环境变量**

**2、新建系统变量，变量名：CATALINA\_HOME 变量值：C:\Program Files\apache-tomcat-7.0.39 (Tomcat 解压到的目录)。**

**3、在系统变量 Path的最后面添加%CATALINA\_HOME%\lib;%CATALINA\_HOME%\lib\servlet-api.jar;%CATALINA\_HOME%\lib\jsp-api.jar 注意它们之间的分号，一定是英文的分号。**

**4、Tomcat 7.0的管理员的配置，进入C:\Program Files\apache-tomcat-7.0.11(Tomcat目录)下的conf目录，编辑tomcat-users.xml，找到最后的：**

**<!--**

**<role rolename="tomcat"/>**

**<role rolename="role1"/>**

**<user username="tomcat" password="tomcat" roles="tomcat"/>**

**<user username="both" password="tomcat" roles="tomcat,role1"/>**

**<user username="role1" password="tomcat" roles="role1"/>**

**-->**

**在上面这段后面 添加上：**

1. **<role** rolename="admin-gui"**/>**
2. **<role** rolename="manage-gui"**/>**
3. **<user** username="admin" password="admin" roles="admin-gui,manage-gui"**/>**

**到这应该知道我们添加的管理员的用户名和密码是多少了吧!**

**5、进入Tomcat 目录下的bin目录，双击startup.bat 启动Tomcat 在命令行窗口会出来英文提示。**

**6、浏览器输入：http://localhost:8080 可以看到Tomcat的欢迎页面就说明配置成功了，点击右上角上manager连接，输入上面配置的用户名和密码，就可以进入管理页面。**

****

# java 基础

## 环境变量配置

需要设置以下三个环境变量，注意三个变量都设置到系统变量里。

JAVA\_HOME：C:\Program Files\Java\jdk1.7.0\_21

CLASSPATH：.;%JAVA\_HOME%\lib\dt.jar;%JAVA\_HOME%\lib\tools.jar

Path：JAVA\_HOME%\bin;%JAVA\_HOME%\jre\bin;

设置完成后，在cmd命令行执行java和javac验证是否成功。

## [使用命令wsimport构建WebService客户端](http://www.cnblogs.com/simle/archive/2011/11/03/2233417.html)

建立 a 文件夹，执行命令：

E:\阳光\医院自助终端\华侨医院>wsimport -s a -p com.sun309.frontend.remote.settle http://www.jd120.com/Settle/SettleService.asmx?wsdl

## 乱码问题

1、在http方式进行交互时，可通过制定StringEntity的编码来处理乱码问题，如：

StringEntity StringEntity = new StringEntity(input.toString(), "UTF-8");

2、在接收网络数据时，可通过指定InputStreamReader的编码来处理乱码问题，如：

BufferedReader rd = new BufferedReader(new InputStreamReader(response.getEntity().getContent(), "UTF-8"));

3、jsp页面的get方法乱码

GET方式提交数据，IE乱码火狐、chrome正常

这是因为使用GET方式通过浏览器的地址栏提交数据时，IE默认是GBK编码，chrome默认是UTF-8编码，所以会出现IE乱码chrome正常 或 chrome乱码IE正常 的情况。

一个自创的办法是，在提交数据时增加一个隐藏字段：chinese=中文。

在接收端通过chinese字段的长度来判断是什么编码，是长度是4，则是UTF-8编码，如果长度是6，则是GBK编码。

if(request.getParameter("chinese").length()==4)

String remark = new String(

request.getParameter("remark").getBytes("ISO-8859-1"),"GB2312");

if(request.getParameter("chinese").length()==6)

String remark = new String(

request.getParameter("remark").getBytes("ISO-8859-1"),"UTF-8");

4、BufferedOutputStream 输出数据乱码

可以在输出字节流时指定编码UTF-8

BufferedOutputStream bufferedOutputStream = new BufferedOutputStream(outputStream);

bufferedOutputStream.write(sb.toString().getBytes("UTF-8"));

5、response 对象返回数据乱码

HttpServletResponse 的对象 response，返回数据乱码，可以在返回数据之前设置好编码：

response.setContentType("text/html;charset=UTF-8");

## 封装异常

异常封装提高Java代码质量

Java语言的异常处理机制可以确保程序的健壮性，提高系统的开发效率，但是Java API提供的异常都是比较低级（这里的低级指的是"低级别的异常"），只有开发人员才能看的懂，才明白发生了什么问题。对于终端用户来说，这些异常基本上是天书，与业务无关，是纯计算机语言的描述。

这就需要我们对异常进行封装了。

异常封装有三方面的优点：

 1.提高系统的友好性

  例如，打开一个文件，如果文件不存在，则会报FileNotFoundException异常，如果该方法的编写不做任何处理，直接上抛上层，则会降低系统的友好性，代码如下所示：

[java]

public static void doStuff（）throws Exception{

  InputStream is = new FileInputStream（"无效文件。txt"）；   /\*文件操作\*/

}

此时doStuff方法的友好性极差：出现异常时（比如文件不存在），该方法直接把FileNotFoundException异常抛出到上层应用中（或者是终端用户），而上层应用（或用户）要么自己处理，要么接着抛出，最终的结果就是让用户对着"天书"式的文字发呆，用户不知道这是什么问题，只是系统告诉他"哦，我出错了，什么错误？你自己看着办吧".

解决办法就是封装异常，可以把系统的阅读者分为两类：开发人员和用户。开发人员查找问题，需要打印出堆栈信息，而用户则需要了解具体的业务原因，比如文件太大，不能同时编写文件等，代码如下：

[java]

public static void doStuff2（）throws MyBussinessException{

try{

   InputStream is = new FileInputStream（"无效文件。txt"）；

}catch（FileNotFoundException e）{

//为了方便开发和维护人员而设置的异常信息

e.printStackTree（）；

  //抛出业务异常

    throw new MyBussinessException（e）；

 }

}

2.提高系统的可维护性

[java]

public void doStuff（）{

try{

   //do something

}catch（Exception e）{

e.printStackTrace（）；

}

}

这是很多程序员容易犯的错误，抛出异常是吧？分类处理多麻烦，就写一个catch块来处理所有异常吧。而且还信誓旦旦的说"JVM会打印出栈中的错误信息",虽然这没有错，但是该信息只有开发人员自己才看的懂，维护人员看见这段异常基本上无法处理，因为需要深入到代码逻辑中去分析问题。   正确的做法是对异常进行分类处理，并进行封装输出，代码如下：

[java]

public void doStuff（）{

  try{

   //do something

   }catch（FileNotFoundException e）{

   log.info（"文件夹未找到，使用默认配置文件…"）；

  }catch（SecurityException 3）{

log.info（"无权访问，可能原因是…"）；

 e.printStackTrace（）；

 }

}

   如此包装后，维护人员看到这样子的异常就有了初步的判断，或者检查配置，或者初始化环境，不需要直接到代码层级去分析了。

3.解决Java异常机制自身的缺陷

   Java中的异常一次只能抛出一个，比如，doStuff方法有两个逻辑代码片段，如果在第一个逻辑片段中抛出异常，则第二个逻辑片段就不执行了，也就无法抛出第二个异常了。那么如何才能一次抛出两个异常呢？

其实，使用自行封装的异常可以解决问题，代码如下：

  [java]

  class MyException extends Exception{

//容纳所有异常

private List<Throwable> causes = new ArrayList<Throwable>（）；

  //构造函数，传递一个异常列表

   public MyException（List<? extends Throwable> \_causes）{

cause.addAll（\_causes）；

}

   //读取所有的异常

   public List<Throwable> getException（）{

return causes;

 }

 }

MyException异常只是一个异常容器，可以容纳多个异常，但它本身并不代表任何异常含义，它所解决的是一次抛出多个异常的问题，具体调用如下：

[java]

  public static void doStuff（）throws MyException{

List<Throwable> list = new ArrayList<Throwable>（）；

//第一个逻辑片段

try{

   //Do something

}catch（Exception e）{

list.add（e）；

}

   //第二个逻辑片段

  try{

   //Do something

}catch（Exception e）{

   list.add（e）；

}

   //检查是否有必要抛出异常

if（list.size（） > 0）{

throw new MyException（list）；

 }

}

   这样一来，doStuff方法的调用者就可以一次获得多个异常，也能够为用户提供完整的例外情况说明。

   那么在什么情况下，需要一个方法抛出多个异常呢？比如Web界面注册时，展示层依次把User对象传递给逻辑层，Register方法需要对各个Field进行校验并注册，例如，用户名不能重复，密码必须符合密码策略等，不要出现第一次提交系统提示"用户名重复",在修改在提交后，系统提示"密码长度少6位"的情况，这种操作模式用户体验非常糟糕，最好的解决办法就是封装异常，建立异常容器，一次性地对User对象进行校验，然后返回所有异常。

建议：

  在开发的过程中，根据具体的情况和需要，对异常进行封装。要区分出代码异常和业务异常，把代码异常转换成业务异常。例如一个Java Web 项目中，包含Controller层、Service层、Dao层，通常在Service层抛出业务异常给Controller，Controller再把业务异常封装成数据返回给客户端。

业务异常类代码例子：

package com.sun309.health.exception;

/\*\*

\* author: Leo Zhang

\*/

public class BussException extends Exception {

private static final long serialVersionUID = 1684692390174507202L;

public BussException() {

}

public BussException(String message) {

super(message);

}

}

封装异常代码和异常信息的类：

1. **import** java.util.Map;
2. **import** java.util.concurrent.ConcurrentHashMap;
3. /\*\*常量类型定义\*/
4. **public** **class** ExceptionConstants {
6. **public** **static** **final** String Code\_0000 = "0000";
7. **public** **static** **final** String Msg\_0000 = "输入参数不合法!";
9. **public** **static** **final** String Code\_0100 = "0100";
10. **public** **static** **final** String Msg\_0100 = "";
12. **public** **static** **final** String Code\_9999 = "9999";
13. **public** **static** **final** String Msg\_9999 = "其它错误!";
15. **private** **static** Map<String, String> returnCodeMap = **new** ConcurrentHashMap<String, String>();
17. **public** **static** Map<String, String> getReturnCodeMap() {
18. **if** (returnCodeMap.isEmpty()) {
19. returnCodeMap.put(Code\_0000, Msg\_0000);
20. returnCodeMap.put(Code\_0100, Msg\_0100);
21. returnCodeMap.put(Code\_9999, Msg\_9999);
22. }
23. **return** returnCodeMap;
24. }
25. }

## [EXCEL导出经典实现](http://blog.csdn.net/evangel_z/article/details/7332535)

1、 [**Java POI 导出EXCEL经典实现 Java导出Excel弹出下载框**](http://blog.csdn.net/evangel_z/article/details/7332535)

在web开发中，有一个经典的功能，就是数据的导入导出。特别是数据的导出，在生产管理或者财务系统中用的非常普遍，因为这些系统经常要做一些报表打印的工作。而数据导出的格式一般是EXCEL或者PDF，我这里就用两篇文章分别给大家介绍下。（注意，我们这里说的数据导出可不是数据库中的数据导出！么误会啦^\_^）

         呵呵，首先我们来导出EXCEL格式的文件吧。现在主流的操作Excel文件的开源工具有很多,用得比较多的就是Apache的POI及JExcelAPI。这里我们用Apache POI！我们先去Apache的大本营下载POI的jar包：http://poi.apache.org/ ，我这里使用的是3.0.2版本。

        将3个jar包导入到classpath下，什么？忘了怎么导包？不会吧！好，我们来写一个导出Excel的实用类（所谓实用，是指基本不用怎么修改就可以在实际项目中直接使用的！）。我一直强调做类也好，做方法也好，一定要通用性和灵活性强。下面这个类就算基本贯彻了我的这种思想。那么，熟悉许老师风格的人应该知道，这时候该要甩出一长串代码了。没错，大伙请看：

**[java]** [view plaincopy](http://blog.csdn.net/evangel_z/article/details/7332535)[在CODE上查看代码片](https://code.csdn.net/snippets/235064)

1. **import** java.util.Date;
3. **public** **class** Student
4. {
5. **private** **long** id;
6. **private** String name;
7. **private** **int** age;
8. **private** **boolean** sex;
9. **private** Date birthday;
11. **public** Student()
12. {
13. }
15. **public** Student(**long** id, String name, **int** age, **boolean** sex, Date birthday)
16. {
17. **this**.id = id;
18. **this**.name = name;
19. **this**.age = age;
20. **this**.sex = sex;
21. **this**.birthday = birthday;
22. }
24. **public** **long** getId()
25. {
26. **return** id;
27. }
29. **public** **void** setId(**long** id)
30. {
31. **this**.id = id;
32. }
34. **public** String getName()
35. {
36. **return** name;
37. }
39. **public** **void** setName(String name)
40. {
41. **this**.name = name;
42. }
44. **public** **int** getAge()
45. {
46. **return** age;
47. }
49. **public** **void** setAge(**int** age)
50. {
51. **this**.age = age;
52. }
54. **public** **boolean** getSex()
55. {
56. **return** sex;
57. }
59. **public** **void** setSex(**boolean** sex)
60. {
61. **this**.sex = sex;
62. }
64. **public** Date getBirthday()
65. {
66. **return** birthday;
67. }
69. **public** **void** setBirthday(Date birthday)
70. {
71. **this**.birthday = birthday;
72. }
74. }

**[java]** [view plaincopy](http://blog.csdn.net/evangel_z/article/details/7332535)[在CODE上查看代码片](https://code.csdn.net/snippets/235064)

1. **public** **class** Book
2. {
3. **private** **int** bookId;
4. **private** String name;
5. **private** String author;
6. **private** **float** price;
7. **private** String isbn;
8. **private** String pubName;
9. **private** **byte**[] preface;
11. **public** Book()
12. {
13. }
15. **public** Book(**int** bookId, String name, String author, **float** price,
16. String isbn, String pubName, **byte**[] preface)
17. {
18. **this**.bookId = bookId;
19. **this**.name = name;
20. **this**.author = author;
21. **this**.price = price;
22. **this**.isbn = isbn;
23. **this**.pubName = pubName;
24. **this**.preface = preface;
25. }
27. **public** **int** getBookId()
28. {
29. **return** bookId;
30. }
32. **public** **void** setBookId(**int** bookId)
33. {
34. **this**.bookId = bookId;
35. }
37. **public** String getName()
38. {
39. **return** name;
40. }
42. **public** **void** setName(String name)
43. {
44. **this**.name = name;
45. }
47. **public** String getAuthor()
48. {
49. **return** author;
50. }
52. **public** **void** setAuthor(String author)
53. {
54. **this**.author = author;
55. }
57. **public** **float** getPrice()
58. {
59. **return** price;
60. }
62. **public** **void** setPrice(**float** price)
63. {
64. **this**.price = price;
65. }
67. **public** String getIsbn()
68. {
69. **return** isbn;
70. }
72. **public** **void** setIsbn(String isbn)
73. {
74. **this**.isbn = isbn;
75. }
77. **public** String getPubName()
78. {
79. **return** pubName;
80. }
82. **public** **void** setPubName(String pubName)
83. {
84. **this**.pubName = pubName;
85. }
87. **public** **byte**[] getPreface()
88. {
89. **return** preface;
90. }
92. **public** **void** setPreface(**byte**[] preface)
93. {
94. **this**.preface = preface;
95. }
96. }

上面这两个类一目了然，就是两个简单的javabean风格的类。再看下面真正的重点类：

**[java]** [view plaincopy](http://blog.csdn.net/evangel_z/article/details/7332535)[在CODE上查看代码片](https://code.csdn.net/snippets/235064)

1. **import** java.io.BufferedInputStream;
2. **import** java.io.FileInputStream;
3. **import** java.io.FileNotFoundException;
4. **import** java.io.FileOutputStream;
5. **import** java.io.IOException;
6. **import** java.io.OutputStream;
7. **import** java.lang.reflect.Field;
8. **import** java.lang.reflect.InvocationTargetException;
9. **import** java.lang.reflect.Method;
10. **import** java.text.SimpleDateFormat;
11. **import** java.util.ArrayList;
12. **import** java.util.Collection;
13. **import** java.util.Date;
14. **import** java.util.Iterator;
15. **import** java.util.List;
16. **import** java.util.regex.Matcher;
17. **import** java.util.regex.Pattern;
19. **import** javax.swing.JOptionPane;
21. **import** org.apache.poi.hssf.usermodel.HSSFCell;
22. **import** org.apache.poi.hssf.usermodel.HSSFCellStyle;
23. **import** org.apache.poi.hssf.usermodel.HSSFClientAnchor;
24. **import** org.apache.poi.hssf.usermodel.HSSFComment;
25. **import** org.apache.poi.hssf.usermodel.HSSFFont;
26. **import** org.apache.poi.hssf.usermodel.HSSFPatriarch;
27. **import** org.apache.poi.hssf.usermodel.HSSFRichTextString;
28. **import** org.apache.poi.hssf.usermodel.HSSFRow;
29. **import** org.apache.poi.hssf.usermodel.HSSFSheet;
30. **import** org.apache.poi.hssf.usermodel.HSSFWorkbook;
31. **import** org.apache.poi.hssf.util.HSSFColor;
33. /\*\*
34. \* 利用开源组件POI3.0.2动态导出EXCEL文档 转载时请保留以下信息，注明出处！
35. \*
36. \* @author leno
37. \* @version v1.0
38. \* @param <T>
39. \*            应用泛型，代表任意一个符合javabean风格的类
40. \*            注意这里为了简单起见，boolean型的属性xxx的get器方式为getXxx(),而不是isXxx()
41. \*            byte[]表jpg格式的图片数据
42. \*/
43. **public** **class** ExportExcel<T>
44. {
45. **public** **void** exportExcel(Collection<T> dataset, OutputStream out)
46. {
47. exportExcel("测试POI导出EXCEL文档", **null**, dataset, out, "yyyy-MM-dd");
48. }
50. **public** **void** exportExcel(String[] headers, Collection<T> dataset,
51. OutputStream out)
52. {
53. exportExcel("测试POI导出EXCEL文档", headers, dataset, out, "yyyy-MM-dd");
54. }
56. **public** **void** exportExcel(String[] headers, Collection<T> dataset,
57. OutputStream out, String pattern)
58. {
59. exportExcel("测试POI导出EXCEL文档", headers, dataset, out, pattern);
60. }
62. /\*\*
63. \* 这是一个通用的方法，利用了JAVA的反射机制，可以将放置在JAVA集合中并且符号一定条件的数据以EXCEL 的形式输出到指定IO设备上
64. \*
65. \* @param title
66. \*            表格标题名
67. \* @param headers
68. \*            表格属性列名数组
69. \* @param dataset
70. \*            需要显示的数据集合,集合中一定要放置符合javabean风格的类的对象。此方法支持的
71. \*            javabean属性的数据类型有基本数据类型及String,Date,byte[](图片数据)
72. \* @param out
73. \*            与输出设备关联的流对象，可以将EXCEL文档导出到本地文件或者网络中
74. \* @param pattern
75. \*            如果有时间数据，设定输出格式。默认为"yyy-MM-dd"
76. \*/
77. @SuppressWarnings("unchecked")
78. **public** **void** exportExcel(String title, String[] headers,
79. Collection<T> dataset, OutputStream out, String pattern)
80. {
81. // 声明一个工作薄
82. HSSFWorkbook workbook = **new** HSSFWorkbook();
83. // 生成一个表格
84. HSSFSheet sheet = workbook.createSheet(title);
85. // 设置表格默认列宽度为15个字节
86. sheet.setDefaultColumnWidth((**short**) 15);
87. // 生成一个样式
88. HSSFCellStyle style = workbook.createCellStyle();
89. // 设置这些样式
90. style.setFillForegroundColor(HSSFColor.SKY\_BLUE.index);
91. style.setFillPattern(HSSFCellStyle.SOLID\_FOREGROUND);
92. style.setBorderBottom(HSSFCellStyle.BORDER\_THIN);
93. style.setBorderLeft(HSSFCellStyle.BORDER\_THIN);
94. style.setBorderRight(HSSFCellStyle.BORDER\_THIN);
95. style.setBorderTop(HSSFCellStyle.BORDER\_THIN);
96. style.setAlignment(HSSFCellStyle.ALIGN\_CENTER);
97. // 生成一个字体
98. HSSFFont font = workbook.createFont();
99. font.setColor(HSSFColor.VIOLET.index);
100. font.setFontHeightInPoints((**short**) 12);
101. font.setBoldweight(HSSFFont.BOLDWEIGHT\_BOLD);
102. // 把字体应用到当前的样式
103. style.setFont(font);
104. // 生成并设置另一个样式
105. HSSFCellStyle style2 = workbook.createCellStyle();
106. style2.setFillForegroundColor(HSSFColor.LIGHT\_YELLOW.index);
107. style2.setFillPattern(HSSFCellStyle.SOLID\_FOREGROUND);
108. style2.setBorderBottom(HSSFCellStyle.BORDER\_THIN);
109. style2.setBorderLeft(HSSFCellStyle.BORDER\_THIN);
110. style2.setBorderRight(HSSFCellStyle.BORDER\_THIN);
111. style2.setBorderTop(HSSFCellStyle.BORDER\_THIN);
112. style2.setAlignment(HSSFCellStyle.ALIGN\_CENTER);
113. style2.setVerticalAlignment(HSSFCellStyle.VERTICAL\_CENTER);
114. // 生成另一个字体
115. HSSFFont font2 = workbook.createFont();
116. font2.setBoldweight(HSSFFont.BOLDWEIGHT\_NORMAL);
117. // 把字体应用到当前的样式
118. style2.setFont(font2);
120. // 声明一个画图的顶级管理器
121. HSSFPatriarch patriarch = sheet.createDrawingPatriarch();
122. // 定义注释的大小和位置,详见文档
123. HSSFComment comment = patriarch.createComment(**new** HSSFClientAnchor(0,
124. 0, 0, 0, (**short**) 4, 2, (**short**) 6, 5));
125. // 设置注释内容
126. comment.setString(**new** HSSFRichTextString("可以在POI中添加注释！"));
127. // 设置注释作者，当鼠标移动到单元格上是可以在状态栏中看到该内容.
128. comment.setAuthor("leno");
130. // 产生表格标题行
131. HSSFRow row = sheet.createRow(0);
132. **for** (**short** i = 0; i < headers.length; i++)
133. {
134. HSSFCell cell = row.createCell(i);
135. cell.setCellStyle(style);
136. HSSFRichTextString text = **new** HSSFRichTextString(headers[i]);
137. cell.setCellValue(text);
138. }
140. // 遍历集合数据，产生数据行
141. Iterator<T> it = dataset.iterator();
142. **int** index = 0;
143. **while** (it.hasNext())
144. {
145. index++;
146. row = sheet.createRow(index);
147. T t = (T) it.next();
148. // 利用反射，根据javabean属性的先后顺序，动态调用getXxx()方法得到属性值
149. Field[] fields = t.getClass().getDeclaredFields();
150. **for** (**short** i = 0; i < fields.length; i++)
151. {
152. HSSFCell cell = row.createCell(i);
153. cell.setCellStyle(style2);
154. Field field = fields[i];
155. String fieldName = field.getName();
156. String getMethodName = "get"
157. + fieldName.substring(0, 1).toUpperCase()
158. + fieldName.substring(1);
159. **try**
160. {
161. Class tCls = t.getClass();
162. Method getMethod = tCls.getMethod(getMethodName,
163. **new** Class[]
164. {});
165. Object value = getMethod.invoke(t, **new** Object[]
166. {});
167. // 判断值的类型后进行强制类型转换
168. String textValue = **null**;
169. // if (value instanceof Integer) {
170. // int intValue = (Integer) value;
171. // cell.setCellValue(intValue);
172. // } else if (value instanceof Float) {
173. // float fValue = (Float) value;
174. // textValue = new HSSFRichTextString(
175. // String.valueOf(fValue));
176. // cell.setCellValue(textValue);
177. // } else if (value instanceof Double) {
178. // double dValue = (Double) value;
179. // textValue = new HSSFRichTextString(
180. // String.valueOf(dValue));
181. // cell.setCellValue(textValue);
182. // } else if (value instanceof Long) {
183. // long longValue = (Long) value;
184. // cell.setCellValue(longValue);
185. // }
186. **if** (value **instanceof** Boolean)
187. {
188. **boolean** bValue = (Boolean) value;
189. textValue = "男";
190. **if** (!bValue)
191. {
192. textValue = "女";
193. }
194. }
195. **else** **if** (value **instanceof** Date)
196. {
197. Date date = (Date) value;
198. SimpleDateFormat sdf = **new** SimpleDateFormat(pattern);
199. textValue = sdf.format(date);
200. }
201. **else** **if** (value **instanceof** **byte**[])
202. {
203. // 有图片时，设置行高为60px;
204. row.setHeightInPoints(60);
205. // 设置图片所在列宽度为80px,注意这里单位的一个换算
206. sheet.setColumnWidth(i, (**short**) (35.7 \* 80));
207. // sheet.autoSizeColumn(i);
208. **byte**[] bsValue = (**byte**[]) value;
209. HSSFClientAnchor anchor = **new** HSSFClientAnchor(0, 0,
210. 1023, 255, (**short**) 6, index, (**short**) 6, index);
211. anchor.setAnchorType(2);
212. patriarch.createPicture(anchor, workbook.addPicture(
213. bsValue, HSSFWorkbook.PICTURE\_TYPE\_JPEG));
214. }
215. **else**
216. {
217. // 其它数据类型都当作字符串简单处理
218. textValue = value.toString();
219. }
220. // 如果不是图片数据，就利用正则表达式判断textValue是否全部由数字组成
221. **if** (textValue != **null**)
222. {
223. Pattern p = Pattern.compile("^//d+(//.//d+)?$");
224. Matcher matcher = p.matcher(textValue);
225. **if** (matcher.matches())
226. {
227. // 是数字当作double处理
228. cell.setCellValue(Double.parseDouble(textValue));
229. }
230. **else**
231. {
232. HSSFRichTextString richString = **new** HSSFRichTextString(
233. textValue);
234. HSSFFont font3 = workbook.createFont();
235. font3.setColor(HSSFColor.BLUE.index);
236. richString.applyFont(font3);
237. cell.setCellValue(richString);
238. }
239. }
240. }
241. **catch** (SecurityException e)
242. {
243. e.printStackTrace();
244. }
245. **catch** (NoSuchMethodException e)
246. {
247. e.printStackTrace();
248. }
249. **catch** (IllegalArgumentException e)
250. {
251. e.printStackTrace();
252. }
253. **catch** (IllegalAccessException e)
254. {
255. e.printStackTrace();
256. }
257. **catch** (InvocationTargetException e)
258. {
259. e.printStackTrace();
260. }
261. **finally**
262. {
263. // 清理资源
264. }
265. }
266. }
267. **try**
268. {
269. workbook.write(out);
270. }
271. **catch** (IOException e)
272. {
273. e.printStackTrace();
274. }
275. }
277. **public** **static** **void** main(String[] args)
278. {
279. // 测试学生
280. ExportExcel<Student> ex = **new** ExportExcel<Student>();
281. String[] headers =
282. { "学号", "姓名", "年龄", "性别", "出生日期" };
283. List<Student> dataset = **new** ArrayList<Student>();
284. dataset.add(**new** Student(10000001, "张三", 20, **true**, **new** Date()));
285. dataset.add(**new** Student(20000002, "李四", 24, **false**, **new** Date()));
286. dataset.add(**new** Student(30000003, "王五", 22, **true**, **new** Date()));
287. // 测试图书
288. ExportExcel<Book> ex2 = **new** ExportExcel<Book>();
289. String[] headers2 =
290. { "图书编号", "图书名称", "图书作者", "图书价格", "图书ISBN", "图书出版社", "封面图片" };
291. List<Book> dataset2 = **new** ArrayList<Book>();
292. **try**
293. {
294. BufferedInputStream bis = **new** BufferedInputStream(
295. **new** FileInputStream("V://book.bmp"));
296. **byte**[] buf = **new** **byte**[bis.available()];
297. **while** ((bis.read(buf)) != -1)
298. {
299. //
300. }
301. dataset2.add(**new** Book(1, "jsp", "leno", 300.33f, "1234567",
302. "清华出版社", buf));
303. dataset2.add(**new** Book(2, "java编程思想", "brucl", 300.33f, "1234567",
304. "阳光出版社", buf));
305. dataset2.add(**new** Book(3, "DOM艺术", "lenotang", 300.33f, "1234567",
306. "清华出版社", buf));
307. dataset2.add(**new** Book(4, "c++经典", "leno", 400.33f, "1234567",
308. "清华出版社", buf));
309. dataset2.add(**new** Book(5, "c#入门", "leno", 300.33f, "1234567",
310. "汤春秀出版社", buf));
312. OutputStream out = **new** FileOutputStream("E://a.xls");
313. OutputStream out2 = **new** FileOutputStream("E://b.xls");
314. ex.exportExcel(headers, dataset, out);
315. ex2.exportExcel(headers2, dataset2, out2);
316. out.close();
317. JOptionPane.showMessageDialog(**null**, "导出成功!");
318. System.out.println("excel导出成功！");
319. }
320. **catch** (FileNotFoundException e)
321. {
322. e.printStackTrace();
323. }
324. **catch** (IOException e)
325. {
326. e.printStackTrace();
327. }
328. }
329. }

        写完之后，如果您不是用eclipse工具生成的Servlet,千万别忘了在web.xml上注册这个Servelt。而且同样的，**拷贝一张小巧的图书图片命名为book.jpg放置到当前WEB根目录的/WEB-INF/下**。部署好web工程，用浏览器访问Servlet看下效果吧！是不是下载成功了。呵呵，您可以将下载到本地的excel报表用打印机打印出来，这样您就大功告成了。完事了我们就思考：我们发现，我们做的方法，不管是本地调用，还是在WEB服务器端用Servlet调用；不管是输出学生列表，还是图书列表信息，代码都几乎一样，而且这些数据我们很容器结合后台的DAO操作数据库动态获取。恩，类和方法的通用性和灵活性开始有点感觉了。好啦，祝您学习愉快！

2、Java导出Excel弹出下载框

        将ExportExcel类的main方法改成public void test()，OutputStream out = new FileOutputStream("E://a.xls");这边可以对应Servlet适当改下路径，Servlet代码如下：

**[java]** [view plaincopy](http://blog.csdn.net/evangel_z/article/details/7332535)[在CODE上查看代码片](https://code.csdn.net/snippets/235064)

1. **public** **class** ExcelServlet **extends** HttpServlet {
2. **public** **void** doGet(HttpServletRequest request, HttpServletResponse response)
3. **throws** ServletException, IOException {
4. (**new** ExportExcel()).test();
5. String str = "a.xls";
6. //String path = request.getSession().getServletContext().getRealPath(str);
7. download("E://a.xls", response);
8. }
9. **private** **void** download(String path, HttpServletResponse response) {
10. **try** {
11. // path是指欲下载的文件的路径。
12. File file = **new** File(path);
13. // 取得文件名。
14. String filename = file.getName();
15. // 以流的形式下载文件。
16. InputStream fis = **new** BufferedInputStream(**new** FileInputStream(path));
17. **byte**[] buffer = **new** **byte**[fis.available()];
18. fis.read(buffer);
19. fis.close();
20. // 清空response
21. response.reset();
22. // 设置response的Header
23. response.addHeader("Content-Disposition", "attachment;filename="
24. + **new** String(filename.getBytes()));
25. response.addHeader("Content-Length", "" + file.length());
26. OutputStream toClient = **new** BufferedOutputStream(
27. response.getOutputStream());
28. response.setContentType("application/vnd.ms-excel;charset=gb2312");
29. toClient.write(buffer);
30. toClient.flush();
31. toClient.close();
32. } **catch** (IOException ex) {
33. ex.printStackTrace();
34. }
35. }
36. }

# 内存管理

## 内存溢出实例分析

里水医院自助终端的前置机程序，出现内存溢出的情况。程序启动后不到一周，内存就从200M 飙升至3000M。

1、在Java程序中实时监测内存

Runtime.getRuntime().totalMemory(); //当前程序总内存

Runtime.getRuntime().freeMemory(); //当前程序空闲内存

Runtime.getRuntime().maxMemory()); //操作系统可分配给当前程序的最大内存

System.gc(); //回收

手动调用System.gc()或把对象设置为null,如 list=null，可以加速内存释放。

通过日志观察发现，无论白天系统多人使用时，还是夜晚没有人使用时，内存都在持续上升。因此可能是订单数据同步线程导致。该线程每个10秒同步一次，从数据库中查出未同步的新订单，发送给服务端。

2、使用 jmap工具查看java进程的内存使用情况。对比系统刚启动时和内存飙升后的对象列表情况，发现数据库相关类的对象多了很多。可能与c3p0或Hibernate有关。

3、将订单同步线程由10秒同步一次改为10分钟同步一次，则内存使用正常，没有出现内存持续上升的情况。

## 内存工具介绍

### jmap的用法

打印出某个java进程（使用pid）内存内的，所有‘对象’的情况（如：产生那些对象，及其数量）。

可以输出所有内存中对象的工具，甚至可以将VM 中的heap，以二进制输出成文本。使用方法 jmap -histo pid。如果连用SHELL jmap -histo pid>a.log可以将其保存到文本中去，在一段时间后，使用文本对比工具，可以对比出GC回收了哪些对象。jmap -dump:format=b,file=String 3024可以将3024进程的内存heap输出出来到String文件里。

[root@localhost bin]# jmap -histo 25917

Attaching to process ID 26221, please wait...

Debugger attached successfully.

Client compiler detected.

JVM version is 1.5.0\_08-b03

Iterating over heap. This may take a while...

Unknown oop at 0xaa6e42d0

Oop's klass is null

Object Histogram:

Size Count Class description

-------------------------------------------------------

3722768 30467 \* ConstMethodKlass

1976480 25334 char[]

1907880 46994 \* SymbolKlass

1762088 2947 byte[]

1709536 30467 \* MethodKlass

1487816 2600 \* ConstantPoolKlass

1009576 2600 \* InstanceKlassKlass

904880 2199 \* ConstantPoolCacheKlass

741432 30893 java.lang.String

653576 4785 int[]

351760 4397 java.lang.reflect.Method

277824 2894 java.lang.Class

248704 3401 short[]

200888 4411 java.lang.Object[]

193656 4045 java.lang.Object[]

179744 5617 java.util.TreeMap$Entry

175688 1800 java.util.HashMap$Entry[]

165288 6887 java.util.HashMap$Entry

104736 3273 java.lang.ref.SoftReference

104136 4339 java.lang.ref.WeakReference

96096 3521 java.lang.String[]

86160 3590 java.util.Hashtable$Entry

85584 3566 java.util.ArrayList

83472 1206 java.util.Hashtable$Entry[]

82944 1728 java.beans.MethodDescriptor

80560 265 \* ObjArrayKlassKlass

69120 1728 java.util.HashMap

52464 3055 java.lang.Class[]

43040 1076 java.util.Hashtable

42496 664 org.apache.commons.modeler.AttributeInfo

37880 947 java.util.TreeMap

33896 557 javax.management.modelmbean.ModelMBeanAttributeInfo[]

33152 518 java.beans.PropertyDescriptor

616 11 org.springframework.aop.framework.ProxyFactory

608 19 java.util.PropertyPermission

608 38 org.springframework.beans.MutablePropertyValues

608 38 org.springframework.beans.factory.support.MethodOverrides

608 2 \* ArrayKlassKlass

608 38 org.springframework.beans.factory.config.ConstructorArgumentValues

608 4 org.apache.xerces.impl.XMLDTDScannerImpl

576 24 java.util.Stack

576 36 java.util.regex.Pattern$Category

576 24 org.apache.naming.NamingEntry

560 7 java.net.URL[]

552 23 sun.management.MappedMXBeanType$BasicMXBeanType

552 1 java.util.Locale[]

552 22 java.io.ObjectStreamField[]

544 17 java.util.Collections$SynchronizedMap

176 11 java.util.regex.Pattern$Ctype

8 1 sun.reflect.GeneratedMethodAccessor49

8 1 sun.reflect.GeneratedMethodAccessor6

8 1 sun.reflect.GeneratedConstructorAccessor10

Heap traversal took 12.003 seconds.

# Java Web

## 各种实体类对比

### Entity

entity 实体类。常用在数据库访问的时候。

Spring可在实体类中使用注解与数据表和字段进行映射。

例如科室表的Entity类：

import org.hibernate.annotations.Cache;

import org.hibernate.annotations.CacheConcurrencyStrategy;

import javax.persistence.\*;

import javax.xml.bind.annotation.XmlRootElement;

import java.io.Serializable;

@XmlRootElement

@Entity

@Cache(usage=CacheConcurrencyStrategy.READ\_WRITE)

@Table(name = "T\_OPC\_DEPTS")

public class DeptEntity implements Serializable {

private static final long serialVersionUID = -1697061357355856964L;

@Id

@Column(name = "DEPTID")

@GeneratedValue(strategy=GenerationType.SEQUENCE,generator="S\_OPC\_DEPTID")

@SequenceGenerator(name="S\_OPC\_DEPTID",sequenceName="S\_OPC\_DEPTID",allocationSize=1)

private long deptid; //科室ID

@ManyToOne(cascade=CascadeType.REFRESH,optional=false)

@JoinColumn(name="HOSPITALID")

private HospitalEntity hospital; //医院

@Column(name = "DEPTNAME")

private String deptname; //科室名称

@Column(name = "HISDEPTID")

private String hisdeptid;//HIS对应的科室ID

@ManyToOne(cascade=CascadeType.REFRESH,optional=true)

@JoinColumn(name="PARENTID")

private DeptEntity parentdept;//上级科室

@Enumerated(EnumType.ORDINAL)

@Column(name = "status")

private StatusEnum status;

public long getDeptid() {

return deptid;

}

……

}

### Model

Model 模型类。常在数据库访问中使用。

Hibernate通过模型层与数据库进行映射。

例如某使用了Hibernate的预约挂号项目中的科室模型：

DeptModel.java 类：

**public** **class** DeptModel {

**private** String deptId;

**private** String deptName;

**public** String getDeptId() {

**return** deptId;

}

**public** **void** setDeptId(String deptId) {

**this**.deptId = deptId;

}

**public** String getDeptName() {

**return** deptName;

}

**public** **void** setDeptName(String deptName) {

**this**.deptName = deptName;

}

}

DeptModel.hbm.xml配置文件：

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<!DOCTYPE hibernate-mapping PUBLIC

'-//Hibernate/Hibernate Mapping DTD 3.0//EN'

'http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd'>

<hibernate-mapping>

<class name=*"com.sun309.frontend.db.model.DeptModel"* table=*"dept"*>

<id name=*"deptId"* column=*"dept\_id"*>

<generator class=*"assigned"*/>

</id>

<property name=*"deptName"* column=*"dept\_name"*></property>

</class>

</hibernate-mapping>

DeptDao.java：

**import** java.util.List;

**import** org.apache.commons.logging.Log;

**import** org.hibernate.Session;

**import** org.hibernate.SessionFactory;

**import** org.hibernate.Transaction;

**import** org.hibernate.cfg.Configuration;

**import** org.hibernate.criterion.Restrictions;

**import** com.sun309.frontend.db.model.FrontendDeptModel;

**public** **class** FrontendDeptDao {

**public** **static** **void** main(String[] args) {

FrontendDeptModel model = **new** FrontendDeptModel();

model.setHospitalId(8000001);

FrontendDeptDao dao = **new** FrontendDeptDao();

dao.queryByHospitalId(8000001);

}

**public** **void** insert(FrontendDeptModel model, Log logger){

SessionFactory sessionFactory = **new** Configuration().configure().~~buildSessionFactory~~();

Session session = **null**;

Transaction transaction = **null**;

**try**{

session = sessionFactory.openSession();

transaction = session.beginTransaction();

session.save(model);

transaction.commit();

}**catch**(Exception e){

logger.debug("写入数据库异常", e);

transaction.rollback();

}**finally**{

session.close();

sessionFactory.close();

}

}

**public** **void** query(){

SessionFactory sessionFactory = **new** Configuration().configure().~~buildSessionFactory~~();

Session session = sessionFactory.openSession();

**try**{

List<FrontendDeptModel> list = session.createCriteria(FrontendDeptModel.**class**)

.list();

System.*out*.println(list.size());

**if**(list != **null**){

**for**(FrontendDeptModel model : list){

}

}

}**catch**(Exception e){

e.printStackTrace();

}**finally**{

session.close();

}

}

**public** List<FrontendDeptModel> queryByHospitalId(Integer hospitalId){

SessionFactory sessionFactory = **new** Configuration().configure().~~buildSessionFactory~~();

Session session = sessionFactory.openSession();

**try**{

List<FrontendDeptModel> list = session.createCriteria(FrontendDeptModel.**class**)

.add(Restrictions.*eq*("hospitalId", hospitalId))

.list();

**if**(list != **null**){

**return** list;

}

}**catch**(Exception e){

e.printStackTrace();

}**finally**{

session.close();

sessionFactory.close();

}

**return** **null**;

}

}

### Pojo

POJO（Plain Ordinary Java Object）简单的Java对象，使用POJO名称是为了避免和[EJB](http://baike.baidu.com/view/3542.htm)混淆起来, 而且简称比较直接. 其中有一些属性及其getter setter方法的类,没有业务逻辑，有时可以作为[VO](http://baike.baidu.com/view/1097386.htm)(value -object)或[dto](http://baike.baidu.com/view/160599.htm)(Data Transform Object)来使用.当然,如果你有一个简单的运算属性也是可以的,但不允许有业务方法,也不能携带有connection之类的方法。

POJO实质上可以理解为简单的实体类，顾名思义POJO类的作用是方便程序员使用数据库中的数据表。如果项目中使用了Hibernate[框架](http://baike.baidu.com/view/66971.htm)，有一个关联的xml文件，使对象与数据库中的表对应，对象的属性与表中的字段相对应。一般在web应用程序中建立一个数据库的映射对象时，我们称它为POJO。



### po

### dto

DTO(Data Transfer Object)数据传输对象。

在不同的系统中用于远程调用的数据传输。在远程系统调用中使用dto，可以很好的隐藏内部情况。例如调用医院HIS系统获取科室信息，将HIS接口返回的XML或Json转换成DTO。如果HIS返回的数据要入库，则把DTO转成Entity再入库。

做HIS对接时的科室DTO代码如下：

import com.sun309.health.model.enums.DataSourceEnum;

import java.io.Serializable;

public class DeptDto implements Serializable {

private static final long serialVersionUID = 5992620632409876956L;

private String hisDeptId; //科室ID

private String hisDeptName; //科室名称

private String hisParentDeptId; //父科室ID

private String hisDeptDesc; //科室描述

private DataSourceEnum dataSource; //来源

public DeptDto() {

}

public DeptDto(String hisDeptId, String hisDeptName, String hisParentDeptId, String hisDeptDesc, DataSourceEnum dataSource) {

this.hisDeptDesc = hisDeptDesc;

this.hisDeptName = hisDeptName;

this.hisDeptId = hisDeptId;

this.hisParentDeptId = hisParentDeptId;

this.dataSource = dataSource;

}

public String getHisDeptId() {

return hisDeptId;

}

public void setHisDeptId(String hisDeptId) {

this.hisDeptId = hisDeptId;

}

public String getHisDeptName() {

return hisDeptName;

}

public void setHisDeptName(String hisDeptName) {

this.hisDeptName = hisDeptName;

}

public String getHisParentDeptId() {

return hisParentDeptId;

}

public void setHisParentDeptId(String hisParentDeptId) {

this.hisParentDeptId = hisParentDeptId;

}

public String getHisDeptDesc() {

return hisDeptDesc;

}

public void setHisDeptDesc(String hisDeptDesc) {

this.hisDeptDesc = hisDeptDesc;

}

public DataSourceEnum getDataSource() {

return dataSource;

}

public void setDataSource(DataSourceEnum dataSource) {

this.dataSource = dataSource;

}

}

### vo

值对象（Value Object）

### Result

为外部系统提供的接口返回结果。

例如预约系统为网站、微信、手机APP提供的科室查询接口，返回的科室树结构列表信息定义如下：

@XmlRootElement

public class DeptTreeResult implements Serializable {

private static final long serialVersionUID = -6475480881588309698L;

private long deptId;

private String deptName;

private String hisDeptId;

private long sortNo;

private List<DeptTreeResult> childs;

public long getDeptId() {

return deptId;

}

public void setDeptId(long deptId) {

this.deptId = deptId;

}

public String getDeptName() {

return deptName;

}

public void setDeptName(String deptName) {

this.deptName = deptName;

}

public String getHisDeptId() {

return hisDeptId;

}

public void setHisDeptId(String hisDeptId) {

this.hisDeptId = hisDeptId;

}

public long getSortNo() {

return sortNo;

}

public void setSortNo(long sortNo) {

this.sortNo = sortNo;

}

public List<DeptTreeResult> getChilds() {

return childs;

}

public void setChilds(List<DeptTreeResult> childs) {

this.childs = childs;

}

}

### Bean

Form Bean： 很多 Web 框架 都实现了 Form 表单域与 Java 对象属性的自动装配功能，该功能确实非常有用，试想如果没这功能则势必到处冲积着 request.getParameter() 系列方法与类型转换方法的调用。重复代码量大，容易出错，同时又不美观。这种封装表单域的对象就是Form Bean。

Java Bean比FormBean复杂很多，JavaBean 是可复用的组件，对 Java Bean 并没有严格的规范，理论上讲，任何一个 Java 类都可以是一个 Bean 。但通常情况下，由于 Java Bean 是被容器所创建（如 Tomcat) 的，所以 Java Bean 应具有一个无参的构造器，另外，通常 Java Bean 还要实现 Serializable 接口用于实现 Bean 的持久性。 Java Bean 是不能被跨进程访问的。JavaBean是一种组件技术，就好像你做了一个扳子，而这个扳子会在很多地方被拿去用，这个扳子也提供多种功能(你可以拿这个扳子扳、锤、撬等等)，而这个扳子就是一个组件。

http://www.cnblogs.com/ldcsaa/archive/2012/02/16/2353030.html

### Domain Object

### Enum

Enum枚举类型。

例如订单状态的枚举类型代码如下：

OrderStatusEnum.java：

package com.sun309.health.model.enums;

import javax.faces.model.SelectItem;

public enum OrderStatusEnum {

预约成功(0),

预约失败(1),

等待HIS预约结果(2),

等待人工预约结果(3),

等待诊疗卡资料(4),

等待撤销中(5),

成功撤销(6),

预约成功不可撤销(7),

等待支付(8),

支付超时被撤销(9);

private int status;

public int getValue() {

return status;

}

private OrderStatusEnum(int status) {

this.status = status;

}

public static OrderStatusEnum getOpcOrderStatusEnum(int status) {

switch (status) {

case 0:

return 预约成功;

case 1:

return 预约失败;

case 2:

return 等待HIS预约结果;

case 3:

return 等待人工预约结果;

case 4:

return 等待诊疗卡资料;

case 5:

return 等待撤销中;

case 6:

return 成功撤销;

case 7:

return 预约成功不可撤销;

case 8:

return 等待支付;

case 9:

return 支付超时被撤销;

default:

return null;

}

}

public static SelectItem[] getFilterOptions() {

SelectItem[] options = new SelectItem[11];

options[0] = new SelectItem("SELECT", "所有状态");

options[1] = new SelectItem("预约成功", "预约成功");

options[2] = new SelectItem("预约失败", "预约失败");

options[3] = new SelectItem("等待HIS预约结果", "等待HIS预约结果");

options[4] = new SelectItem("等待人工预约结果", "等待人工预约结果");

options[5] = new SelectItem("等待诊疗卡资料", "等待诊疗卡资料");

options[6] = new SelectItem("等待撤销中", "等待撤销中");

options[7] = new SelectItem("成功撤销", "成功撤销");

options[8] = new SelectItem("预约成功不可撤销", "预约成功不可撤销");

options[9] = new SelectItem("等待支付", "等待支付");

options[10] = new SelectItem("支付超时被撤销", "支付超时被撤销");

return options;

}

}

在数据库订单表中，status字段是整数类型，在与数据库映射的OrderEntity类中，status字段是Integer 类型，而在Controller返回给客户端的订单结果类里，OrderResult的status是String类型。

服务层在查询数据库获取OrderEntity后，是这样转换的：

orderResult.setStatus(orderEntity.getStatus().name());

在这行代码中，orderEntity.getStatus()返回OrderStatusEnum类型，再调用它的name()方法，这样就取得了状态的描述字符串传给orderResult。

### Json或XML

## Session

# MVC

## 数据访问层（Model）

### Hibernate

当数据表不多，数据库访问量不大时，可以把所有涉及数据库访问的操作集中到一个数据服务类里（DataServiceImpl），这个服务类可供其他服务类或控制器访问。

public class DataServiceImpl implements IDataService {

@Autowired

private DeptDao deptDao;

@Autowired

private DoctorDao doctorDao;

……

}

### Repository

@Repository

spring data jpa

当数据表不多，数据库访问量不大时，可以把所有涉及数据库访问的操作集中到一个数据服务类里（DataServiceImpl），这个服务类可供其他服务类或控制器访问。

public class DataServiceImpl implements IDataService {

@Autowired

private DeptRepository deptRepository;

@Autowired

private DoctorRepository doctorRepository;

……

}

## 第一章：Spring Data JPA入门

Spring Data是什么

Spring Data是一个用于简化数据库访问，并支持云服务的开源框架。其主要目标是使得对数据的访问变得方便快捷，并支持map-reduce框架和云计算数据服务。 Spring Data 包含多个子项目：

**Commons - 提供共享的基础框架，适合各个子项目使用，支持跨数据库持久化**

**JPA - 简化创建 JPA 数据访问层和跨存储的持久层功能**

Hadoop - 基于 Spring 的 Hadoop 作业配置和一个 POJO 编程模型的 MapReduce 作业

Key-Value  - 集成了 Redis 和 Riak ，提供多个常用场景下的简单封装

Document - 集成文档数据库：CouchDB 和 MongoDB 并提供基本的配置映射和资料库支持

Graph - 集成 Neo4j 提供强大的基于 POJO 的编程模型

Graph Roo AddOn - Roo support for Neo4j

JDBC Extensions - 支持 Oracle RAD、高级队列和高级数据类型

Mapping - 基于 Grails 的提供对象映射框架，支持不同的数据库

Examples - 示例程序、文档和图数据库

Guidance - 高级文档

Spring Data JPA是什么

由Spring提供的一个用于简化JPA开发的框架

nSpring Data JPA能干什么

可以极大的简化JPA的写法，可以在几乎不用写实现的情况下，实现对数据的访问和操作。除了CRUD外，还包括如分页、排序等一些常用的功能。

Spring Data JPA有什么

主要来看看Spring Data JPA提供的接口，也是Spring Data JPA的核心概念：

1：Repository：最顶层的接口，是一个空的接口，目的是为了统一所有Repository的类型，且能让组件扫描的时候自动识别。

2：CrudRepository ：是Repository的子接口，提供CRUD的功能

3：PagingAndSortingRepository：是CrudRepository的子接口，添加分页和排序的功能

4：JpaRepository：是PagingAndSortingRepository的子接口，增加了一些实用的功能，比如：批量操作等。

5：JpaSpecificationExecutor：用来做负责查询的接口

6：Specification：是Spring Data JPA提供的一个查询规范，要做复杂的查询，只需围绕这个规范来设置查询条件即可

**HelloWorld**

n环境构建

在Eclipse里面构建一个普通的Java工程，主要就是要加入一堆的jar包。

1：首先去官网下载Spring Data Common 和 Spring Data JPA的包，把里面dist的jar包加入到工程中，这里是spring-data-commons-1.5.0.RELEASE.jar和spring-data-jpa-1.3.2.RELEASE.jar

2：把Spring3.2.3的jar包添加到工程中

3：JPA的实现选用的是Hibernate4.2.0，总共还需要额外加入如下的jar：

antlr-2.7.7.jar、aopalliance-1.0.jar、asm-3.2.jar、aspectjrt-1.7.1.jar、aspectjweaver-1.7.1.jar、commons-beanutils-1.8.3.jar、commons-codec-1.7.jar、commons-collections-3.2.1.jar、commons-dbcp-1.4.jar、commons-fileupload-1.2.2.jar、commons-io-2.4.jar、commons-lang3-3.1.jar、commons-logging-1.1.1.jar、commons-pool-1.6.jar、dom4j-1.6.1.jar、hibernate-commons-annotations-4.0.1.Final.jar、hibernate-core-4.2.0.Final.jar、hibernate-entitymanager-4.2.0.Final.jar、hibernate-jpa-2.0-api-1.0.1.Final.jar、javassist-3.15.0-GA.jar、jboss-logging-3.1.0.GA.jar、jboss-transaction-api\_1.1\_spec-1.0.0.Final.jar、mysql-connector-java-5.1.9.jar、slf4j-api-1.7.3.jar

n实体对象，就是以前的实现方式

@Entity

@Table(name="tbl\_user")

public class UserModel {

@Id

private Integer uuid;

private String name;

private Integer age;

//省略getter/setter

}

nDAO的接口

public interface UserRepository extends JpaRepository<UserModel, Integer>{

//空的，可以什么都不用写

}

无需提供实现，Spring Data JPA会为我们搞定一切

n写个逻辑层的Service，其实就相当于DAO的客户端，用来测试

@Service

@Transactional

public class Client {

@Autowired

private UserRepository ur;

public void testAdd(UserModel um){ ur.save(um); }

public static void main(String[] args) {

ApplicationContext ctx = new ClassPathXmlApplicationContext("applicationContext.xml");

Client c = (Client)ctx.getBean("client");

UserModel um = new UserModel();

um.setAge(1);

um.setName("张三");

um.setUuid(1);

c.testAdd(um);

} }

n同样需要在Spring的配置文件中配置，基本跟使用注解的配置类似：

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:context="http://www.springframework.org/schema/context"

xmlns:aop="http://www.springframework.org/schema/aop"

xmlns:tx="http://www.springframework.org/schema/tx"

xmlns:jpa="http://www.springframework.org/schema/data/jpa"

xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-3.0.xsd

http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-3.0.xsd

http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-3.0.xsd

http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-3.0.xsd

http://www.springframework.org/schema/data/jpa http://www.springframework.org/schema/data/jpa/spring-jpa.xsd

">

<context:component-scan base-package="cn.javass">

<context:exclude-filter type="annotation“ expression="org.springframework.stereotype.Controller"/>

</context:component-scan>

<aop:aspectj-autoproxy proxy-target-class="true"/>

<!-- 开启注解事务 只对当前配置文件有效 -->

<tx:annotation-driven transaction-manager="transactionManager" proxy-target-class="true"/>

**<jpa:repositories**

**base-package="cn.javass"**

**repository-impl-postfix="Impl"**

**entity-manager-factory-ref="entityManagerFactory"**

**transaction-manager-ref="transactionManager">**

**</jpa:repositories>**

**<bean id="entityManagerFactory"**

**class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">**

        <property name="dataSource" ref="dataSource"/>

        <property name="packagesToScan" value="cn.javass"/>

        <property name="persistenceProvider">

            <bean class="org.hibernate.ejb.HibernatePersistence"/>

        </property>

        <property name="jpaVendorAdapter">

            <bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">

                <property name="generateDdl" value="false"/>

                <property name="database" value="MYSQL"/>

                <property name="databasePlatform" value="org.hibernate.dialect.MySQL5InnoDBDialect"/>

                <property name="showSql" value="true"/>

            </bean>

        </property>

        <property name="jpaDialect">

            <bean class="org.springframework.orm.jpa.vendor.HibernateJpaDialect"/>

        </property>

        <property name="jpaPropertyMap">

            <map>

                <entry key="hibernate.query.substitutions" value="true 1, false 0"/>

                <entry key="hibernate.default\_batch\_fetch\_size" value="16"/>

                <entry key="hibernate.max\_fetch\_depth" value="2"/>

                <entry key="hibernate.generate\_statistics" value="true"/>

                <entry key="hibernate.bytecode.use\_reflection\_optimizer" value="true"/>

                <entry key="hibernate.cache.use\_second\_level\_cache" value="false"/>

                <entry key="hibernate.cache.use\_query\_cache" value="false"/>

            </map>

        </property>

    </bean>

<!--事务管理器配置-->

    <bean id="transactionManager" class="org.springframework.orm.jpa.JpaTransactionManager">

        <property name="entityManagerFactory" ref="entityManagerFactory"/>

    </bean>

<bean name="dataSource" class="org.apache.commons.dbcp.BasicDataSource">

<property name="driverClassName"><value>org.gjt.mm.mysql.Driver</value></property>

<property name="url"><value>jdbc:mysql://localhost:3306/cc?useUnicode=true&amp;characterEncoding=UTF-8</value></property>

<property name="username"> <value>root</value> </property>

<property name="password" value="cc"/>

</bean>

</beans>

配置完成后，可以去运行Client测试一下了，当然数据库和表需要先准备好

也可以在<jpa:repositories>下面添加filter，形如：

<repositories base-package="com.acme.repositories">

<context:exclude-filter type="regex" expression=".\*SomeRepository" />

</repositories>

## 第二章：JpaRepository基本功能

JpaRepository的基本功能示范

具体的看代码演示

其中：Pageable接口的实现类是PageRequest，Page接口的实现类是PageImpl。

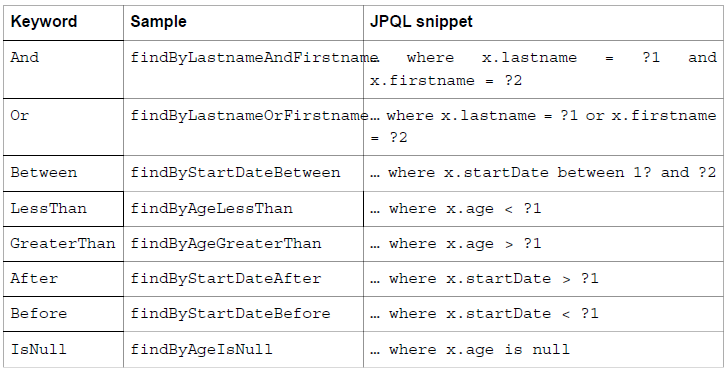
示例如下：

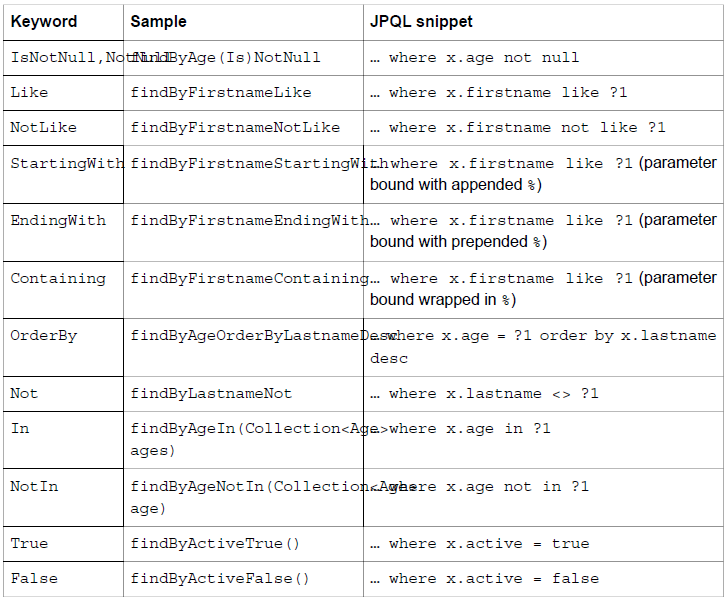
Page<UserModel> p =  ur.findAll(new PageRequest(0,2,new Sort(new Order(Direction. DESC,"uuid"))));

System. out.println("list="+p.getContent());

## 第三章：JpaRepository的查询

直接在接口中定义查询方法，如果是符合规范的，可以不用写实现，目前支持的关键字写法如下：





Spring Data JPA框架在进行方法名解析时，会先把方法名多余的前缀截取掉，比如 find、findBy、read、readBy、get、getBy，然后对剩下部分进行解析。

假如创建如下的查询：findByUserDepUuid()，框架在解析该方法时，首先剔除 findBy，然后对剩下的属性进行解析，假设查询实体为Doc

1：先判断 userDepUuid （根据 POJO 规范，首字母变为小写）是否为查询实体的一个属性，如果是，则表示根据该属性进行查询；如果没有该属性，继续第二步；

2：从右往左截取第一个大写字母开头的字符串此处为Uuid），然后检查剩下的字符串是否为查询实体的一个属性，如果是，则表示根据该属性进行查询；如果没有该属性，则重复第二步，继续从右往左截取；最后假设user为查询实体的一个属性；

3：接着处理剩下部分（DepUuid），先判断 user 所对应的类型是否有depUuid属性，如果有，则表示该方法最终是根据 “ Doc.user.depUuid” 的取值进行查询；否则继续按照步骤 2 的规则从右往左截取，最终表示根据 “Doc.user.dep.uuid” 的值进行查询。

4:可能会存在一种特殊情况，比如 Doc包含一个 user 的属性，也有一个 userDep 属性，此时会存在混淆。可以明确在属性之间加上 "\_" 以显式表达意图，比如 "findByUser\_DepUuid()" 或者 "findByUserDep\_uuid()"

特殊的参数： 还可以直接在方法的参数上加入分页或排序的参数，比如：

Page<UserModel> findByName(String name, Pageable pageable);

List<UserModel> findByName(String name, Sort sort);

也可以使用JPA的NamedQueries，方法如下：

1：在实体类上使用@NamedQuery，示例如下：

@NamedQuery(name = "UserModel.findByAge",query = "select o from UserModel o where o.age >= ?1")

2：在自己实现的DAO的Repository接口里面定义一个同名的方法，示例如下：

public List<UserModel> findByAge(int age);

3：然后就可以使用了，Spring会先找是否有同名的NamedQuery，如果有，那么就不会按照接口定义的方法来解析。

**使用@Query**

可以在自定义的查询方法上使用@Query来指定该方法要执行的查询语句，比如：

@Query("select o from UserModel o where o.uuid=?1")

public List<UserModel> findByUuidOrAge(int uuid);

**注意：**

1：方法的参数个数必须和@Query里面需要的参数个数一致

2：如果是like，后面的参数需要前面或者后面加“%”，比如下面都对：

@Query("select o from UserModel o where o.name like ?1%")

public List<UserModel> findByUuidOrAge(String name);

@Query("select o from UserModel o where o.name like %?1")

public List<UserModel> findByUuidOrAge(String name);

@Query("select o from UserModel o where o.name like %?1%")

public List<UserModel> findByUuidOrAge(String name);

当然，这样在传递参数值的时候就可以不加‘%’了，当然加了也不会错

n还可以使用@Query来指定本地查询，只要设置nativeQuery为true，比如：

@Query(value="select \* from tbl\_user where name like %?1" ,nativeQuery=true)

public List<UserModel> findByUuidOrAge(String name);

**注意：**当前版本的本地查询不支持翻页和动态的排序

使用命名化参数，使用@Param即可，比如：

@Query(value="select o from UserModel o where o.name like %:nn")

public List<UserModel> findByUuidOrAge(@Param("nn") String name);

同样支持更新类的Query语句，添加@Modifying即可，比如：

@Modifying

@Query(value="update UserModel o set o.name=:newName where o.name like %:nn")

public int findByUuidOrAge(@Param("nn") String name,@Param("newName") String newName);

**注意：**

1：方法的返回值应该是int，表示更新语句所影响的行数

2：在调用的地方必须加事务，没有事务不能正常执行

**JpaRepository的查询功能**

创建查询的顺序

Spring Data JPA 在为接口创建代理对象时，如果发现同时存在多种上述情况可用，它该优先采用哪种策略呢？

<jpa:repositories> 提供了 query-lookup-strategy 属性，用以指定查找的顺序。它有如下三个取值：

1：create-if-not-found：如果方法通过@Query指定了查询语句，则使用该语句实现查询；如果没有，则查找是否定义了符合条件的命名查询，如果找到，则使用该命名查询；如果两者都没有找到，则通过解析方法名字来创建查询。这是 query-lookup-strategy 属性的默认值

2：create：通过解析方法名字来创建查询。即使有符合的命名查询，或者方法通过 @Query指定的查询语句，都将会被忽略

3：use-declared-query：如果方法通过@Query指定了查询语句，则使用该语句实现查询；如果没有，则查找是否定义了符合条件的命名查询，如果找到，则使用该命名查询；如果两者都没有找到，则抛出异常

## 第四章：客户化扩展JpaRepository

如果你不想暴露那么多的方法，可以自己订制自己的Repository，还可以在自己的Repository里面添加自己使用的公共方法

当然更灵活的是自己写一个实现类，来实现自己需要的方法

1：写一个与接口同名的类，加上后缀为Impl，这个在前面xml里面配置过，可以自动被扫描到。这个类不需要实现任何接口。

2：在接口中加入自己需要的方法，比如：

public Page<Object[]> getByCondition(UserQueryModel u);

3：在实现类中，去实现这个方法就好了，会被自动找到

**java代码：**

[查看复制到剪贴板打印](http://sishuok.com/forum/posts/list/7000.html)

1. **public** **class** UserRepositoryImpl {
2. @PersistenceContext
3. **private** EntityManager em;
4. **public** Page<Object[]> getByCondition(UserQueryModel u){
5. String hql = "select o.uuid,o.name from UserModel o where 1=1 and o.uuid=:uuid";
6. Query q = em.createQuery(hql);
7. q.setParameter("uuid", u.getUuid());
8. q.setFirstResult(0);
9. q.setMaxResults(1);
10. Page<Object[]> page = **new** PageImpl<Object[]>(q.getResultList(),**new** PageRequest(0,1),3);
11. **return** page;
12. }}

## 

## 第五章：Specifications查询

Spring Data JPA支持JPA2.0的Criteria查询，相应的接口是JpaSpecificationExecutor。

Criteria 查询：是一种类型安全和更面向对象的查询

这个接口基本是围绕着Specification接口来定义的， Specification接口中只定义了如下一个方法：

Predicate toPredicate(Root<T> root, CriteriaQuery<?> query, CriteriaBuilder cb);

要理解这个方法，以及正确的使用它，就需要对JPA2.0的Criteria查询有一个足够的熟悉和理解，因为这个方法的参数和返回值都是JPA标准里面定义的对象。

**Criteria查询基本概念**

Criteria 查询是以元模型的概念为基础的，元模型是为具体持久化单元的受管实体定义的，这些实体可以是实体类，嵌入类或者映射的父类。

CriteriaQuery接口：代表一个specific的顶层查询对象，它包含着查询的各个部分，比如：select 、from、where、group by、order by等

**注意：**CriteriaQuery对象只对实体类型或嵌入式类型的Criteria查询起作用

Root接口：代表Criteria查询的根对象，Criteria查询的查询根定义了实体类型，能为将来导航获得想要的结果，它与SQL查询中的FROM子句类似

1：Root实例是类型化的，且定义了查询的FROM子句中能够出现的类型。

2：查询根实例能通过传入一个实体类型给 AbstractQuery.from方法获得。

3：Criteria查询，可以有多个查询根。

4：AbstractQuery是CriteriaQuery 接口的父类，它提供得到查询根的方法。

CriteriaBuilder接口：用来构建CritiaQuery的构建器对象

Predicate：一个简单或复杂的谓词类型，其实就相当于条件或者是条件组合。

**Criteria查询**

基本对象的构建

1：通过EntityManager的getCriteriaBuilder或EntityManagerFactory的getCriteriaBuilder方法可以得到CriteriaBuilder对象

2：通过调用CriteriaBuilder的createQuery或createTupleQuery方法可以获得CriteriaQuery的实例

3：通过调用CriteriaQuery的from方法可以获得Root实例

过滤条件

1：过滤条件会被应用到SQL语句的FROM子句中。在criteria 查询中，查询条件通过Predicate或Expression实例应用到CriteriaQuery对象上。

2：这些条件使用 CriteriaQuery .where 方法应用到CriteriaQuery 对象上

3：CriteriaBuilder也作为Predicate实例的工厂，通过调用CriteriaBuilder 的条件方法（ equal，notEqual， gt， ge，lt， le，between，like等）创建Predicate对象。

4：复合的Predicate 语句可以使用CriteriaBuilder的and, or andnot 方法构建。

构建简单的Predicate示例：

Predicate p1=cb.like(root.get(“name”).as(String.class), “%”+uqm.getName()+“%”);

Predicate p2=cb.equal(root.get("uuid").as(Integer.class), uqm.getUuid());

Predicate p3=cb.gt(root.get("age").as(Integer.class), uqm.getAge());

构建组合的Predicate示例：

Predicate p = cb.and(p3,cb.or(p1,p2));

当然也可以形如前面动态拼接查询语句的方式，比如：

**java代码：**

[查看复制到剪贴板打印](http://sishuok.com/forum/posts/list/7000.html)

1. Specification<UserModel> spec = **new** Specification<UserModel>() {
2. **public** Predicate toPredicate(Root<UserModel> root,
3. CriteriaQuery<?> query, CriteriaBuilder cb) {
4. List<Predicate> list = **new** ArrayList<Predicate>();
6. **if**(um.getName()!=**null** && um.getName().trim().length()>0){
7. list.add(cb.like(root.get("name").as(String.**class**), "%"+um.getName()+"%"));
8. }
9. **if**(um.getUuid()>0){
10. list.add(cb.equal(root.get("uuid").as(Integer.**class**), um.getUuid()));
11. }
12. Predicate[] p = **new** Predicate[list.size()];
13. **return** cb.and(list.toArray(p));
14. }
15. };

也可以使用CriteriaQuery来得到最后的Predicate，示例如下：

**java代码：**

[查看复制到剪贴板打印](http://sishuok.com/forum/posts/list/7000.html)

1. Specification<UserModel> spec = **new** Specification<UserModel>() {
2. **public** Predicate toPredicate(Root<UserModel> root,
3. CriteriaQuery<?> query, CriteriaBuilder cb) {
4. Predicate p1 = cb.like(root.get("name").as(String.**class**), "%"+um.getName()+"%");
5. Predicate p2 = cb.equal(root.get("uuid").as(Integer.**class**), um.getUuid());
6. Predicate p3 = cb.gt(root.get("age").as(Integer.**class**), um.getAge());
7. //把Predicate应用到CriteriaQuery中去,因为还可以给CriteriaQuery添加其他的功能，比如排序、分组啥的
8. query.where(cb.and(p3,cb.or(p1,p2)));
9. //添加排序的功能
10. query.orderBy(cb.desc(root.get("uuid").as(Integer.**class**)));
12. **return** query.getRestriction();
13. }
14. };

**多表联接**

n多表连接查询稍微麻烦一些，下面演示一下常见的1:M，顺带演示一下1:1

n使用Criteria查询实现1对多的查询

1：首先要添加一个实体对象DepModel，并设置好UserModel和它的1对多关系，如下：

@Entity

@Table(name="tbl\_user")

public class UserModel {

@Id

private Integer uuid;

private String name;

private Integer age;

@OneToMany(mappedBy = "um", fetch = FetchType. LAZY, cascade = {CascadeType. ALL})

private Set<DepModel> setDep;

//省略getter/setter

}

@Entity

@Table(name="tbl\_dep")

public class DepModel {

@Id

private Integer uuid;

private String name;

@ManyToOne()

  @JoinColumn(name = "user\_id", nullable = false)

//表示在tbl\_dep里面有user\_id的字段

private UserModel um = new UserModel();

//省略getter/setter

}

2：配置好Model及其关系后，就可以在构建Specification的时候使用了，示例如下：

Specification<UserModel> spec = new Specification<UserModel>() {

public Predicate toPredicate(Root<UserModel> root, CriteriaQuery<?> query, CriteriaBuilder cb) {

Predicate p1 = cb.like(root.get("name").as(String.class), "%"+um.getName()+"%");

Predicate p2 = cb.equal(root.get("uuid").as(Integer.class), um.getUuid());

Predicate p3 = cb.gt(root.get("age").as(Integer.class), um.getAge());

**SetJoin<UserModel,DepModel> depJoin =** **root.join(root.getModel().getSet("setDep",DepModel.class) , JoinType.LEFT);**

**Predicate p4 = cb.equal(depJoin.get("name").as(String.class), "ddd");**

**//把Predicate应用到CriteriaQuery去,因为还可以给CriteriaQuery添加其他的功能，比如排序、分组啥** **的**

**query.where(cb.and(cb.and(p3,cb.or(p1,p2)),p4));**

//添加分组的功能

query.orderBy(cb.desc(root.get("uuid").as(Integer.class)));

return query.getRestriction();

}};

n接下来看看使用Criteria查询实现1:1的查询

1：在UserModel中去掉setDep的属性及其配置，然后添加如下的属性和配置：

@OneToOne()

@JoinColumn(name = "depUuid")

private DepModel dep;

public DepModel getDep() { return dep;}

public void setDep(DepModel dep) {this.dep = dep;  }

2：在DepModel中um属性上的注解配置去掉，换成如下的配置：

@OneToOne(mappedBy = "dep", fetch = FetchType. EAGER, cascade = {CascadeType. ALL})

3：在Specification实现中，把SetJoin的那句换成如下的语句：

Join<UserModel,DepModel> depJoin =

root.join(root.getModel().getSingularAttribute("dep",DepModel.class),JoinType.LEFT);

//root.join(“dep”,JoinType.LEFT); //这句话和上面一句的功能一样，更简单

### JDBC

当数据表不多，数据库访问量不大时，可以把所有涉及数据库访问的操作集中到一个数据服务类里（DataServiceImpl），这个服务类可供其他服务类或控制器访问。

public class DataServiceImpl implements IDataService {

……

}

### c3p0

### dubbo

## 服务层（Service）

服务层位于数据访问层和控制层之间，负责业务逻辑处理。通常一个大的功能模块需要一个Service类，这些Service类继承同一个父类ServiceBase。在父类中包含了一些各个Service类通用的方法，例如渠道权限检查等。

在服务层中会用到哪些实体类？他们之间怎样相互转换？服务层上有数据访问层返回的数据库实体，如Model、Entity，有远程调用返回的数据传输对象dto，也有页面表单提交的Bean。服务层用这些实体进行业务逻辑处理后，返回Result对象给控制层接口，控制层接口再把Result对象转成XML或Json返回给接口调用者。在WEB项目中，服务层也会返回数据库映射对象POJO给Controller，Controller再传给页面展示。

例如预约系统中部分Service类示例如下：

信息查询服务类InfoServiceImpl：

@Component("infoService")

public class InfoServiceImpl extends ServiceBase implements IInfoService{

private IDeptDao deptDao;

@Autowired

private IDoctorDao doctorDao;

……

}

订单服务类OrderServiceImpl：

@Component("orderService")

public class OrderServiceImpl extends ServiceBase implements IOrderService{

private IOpcOrderDao opcOrderDao;

@Autowired

private OpcOrderService opcOrderService;

……

}

系统管理功能服务类SysServiceImpl：

@Component("sysService")

public class SysServiceImpl extends ServiceBase implements ISysService{

private final String PASSWORD = "69089462873";

@Autowired

private SystemConfig systemConfig;

@Autowired

private ITraderAccountDao traderAccountDao;

……

}

在业务复杂的系统中，一层Service类是不够的。为了使业务逻辑清晰，需要另外提炼一些特殊服务类。这些服务类不继承父类ServiceBase，他们是专门负责某一个复杂的业务功能。

例如预约挂号是预约系统的核心业务，涉及诸多复杂业务处理，因此需要提供一个订单处理服务类OpcOrderService，他和OrderServiceImpl并不同，不继承父类ServiceBase：

public class OpcOrderService implements OpcOrderService {

@Autowired

private JmsTemplate jmsTemplate;

@Autowired

private IHisApp hisApp;

……

}

又如短信通知功能可以抽出来单独提供一个服务类NotifyService：

public class OpcNotifyService implements NotifyService {

@Autowired

private JmsTemplate jmsTemplate;

@Autowired

private ITraderNotifyDao traderNotifyDao;

……

}

又如，当数据表不多数据库访问量不大时，可以把所有涉及数据库访问的操作集中到一个数据服务类里（DataServiceImpl），这个服务类可供其他服务类或控制器访问。

public class DataServiceImpl implements IDataService {

@Autowired

private DeptDao deptDao;

@Autowired

private DoctorDao doctorDao;

……

}

## 控制层（Controller）

在控制层中会用到哪些实体类？他们怎样相互转换？

@ResponseBody

这个Spring注解用于标识一个Controller方法是返回页面还是返回数据。如果Controller方法添加了这个注解，将不会根据返回结果来匹配页面，而是直接把结果发送给远程调用者。

**SpringMVC Controller 介绍**

### 简介

         在SpringMVC 中，控制器Controller 负责处理由DispatcherServlet 分发的请求，它把用户请求的数据经过业务处理层处理之后封装成一个Model ，然后再把该Model 返回给对应的View 进行展示。在SpringMVC 中提供了一个非常简便的定义Controller 的方法，你无需继承特定的类或实现特定的接口，只需使用@Controller 标记一个类是Controller ，然后使用@RequestMapping 和@RequestParam 等一些注解用以定义URL 请求和Controller 方法之间的映射，这样的Controller 就能被外界访问到。此外Controller 不会直接依赖于HttpServletRequest 和HttpServletResponse 等HttpServlet 对象，它们可以通过Controller 的方法参数灵活的获取到。为了先对Controller 有一个初步的印象，以下先定义一个简单的Controller ：

**Java代码  [收藏代码](javascript:void())**

1. @Controller
2. **public** **class** MyController {
4. @RequestMapping ( "/showView" )
5. **public** ModelAndView showView() {
6. ModelAndView modelAndView = **new** ModelAndView();
7. modelAndView.setViewName( "viewName" );
8. modelAndView.addObject( " 需要放到 model 中的属性名称 " , " 对应的属性值，它是一个对象 " );
9. **return** modelAndView;
10. }
12. }

在上面的示例中，@Controller 是标记在类MyController 上面的，所以类MyController 就是一个SpringMVC Controller 对象了，然后使用@RequestMapping(“/showView”) 标记在Controller 方法上，表示当请求/showView.do 的时候访问的是MyController 的showView 方法，该方法返回了一个包括Model 和View 的ModelAndView 对象。这些在后续都将会详细介绍。

### 使用 @Controller 定义一个 Controller 控制器

         @Controller 用于标记在一个类上，使用它标记的类就是一个SpringMVC Controller 对象。分发处理器将会扫描使用了该注解的类的方法，并检测该方法是否使用了@RequestMapping 注解。@Controller 只是定义了一个控制器类，而使用@RequestMapping 注解的方法才是真正处理请求的处理器，这个接下来就会讲到。

   单单使用@Controller 标记在一个类上还不能真正意义上的说它就是SpringMVC 的一个控制器类，因为这个时候Spring 还不认识它。那么要如何做Spring 才能认识它呢？这个时候就需要我们把这个控制器类交给Spring 来管理。拿MyController 来举一个例子

**Java代码  [收藏代码](javascript:void())**

1. @Controller
2. **public** **class** MyController {
3. @RequestMapping ( "/showView" )
4. **public** ModelAndView showView() {
5. ModelAndView modelAndView = **new** ModelAndView();
6. modelAndView.setViewName( "viewName" );
7. modelAndView.addObject( " 需要放到 model 中的属性名称 " , " 对应的属性值，它是一个对象 " );
8. **return** modelAndView;
9. }
11. }

这个时候有两种方式可以把MyController 交给Spring 管理，好让它能够识别我们标记的@Controller 。

   第一种方式是在SpringMVC 的配置文件中定义MyController 的bean 对象。

<bean class="com.host.app.web.controller.MyController"/>

   第二种方式是在SpringMVC 的配置文件中告诉Spring 该到哪里去找标记为@Controller 的Controller 控制器。

**Xml代码  [收藏代码](javascript:void())**

1. **<** **context:component-scan** base-package = "com.host.app.web.controller" **>**
2. **<** **context:exclude-filter** type = "annotation"
3. expression = "org.springframework.stereotype.Service" **/>**
4. **</** **context:component-scan** **>**

    注：

       上面 context:exclude-filter 标注的是不扫描 @Service 标注的类

### 使用 @RequestMapping 来映射 Request 请求与处理器

         可以使用@RequestMapping 来映射URL 到控制器类，或者是到Controller 控制器的处理方法上。当@RequestMapping 标记在Controller 类上的时候，里面使用@RequestMapping 标记的方法的请求地址都是相对于类上的@RequestMapping 而言的；当Controller 类上没有标记@RequestMapping 注解时，方法上的@RequestMapping 都是绝对路径。这种绝对路径和相对路径所组合成的最终路径都是相对于根路径“/ ”而言的。

**Java代码  [收藏代码](javascript:void())**

1. @Controller
2. **public** **class** MyController {
3. @RequestMapping ( "/showView" )
4. **public** ModelAndView showView() {
5. ModelAndView modelAndView = **new** ModelAndView();
6. modelAndView.setViewName( "viewName" );
7. modelAndView.addObject( " 需要放到 model 中的属性名称 " , " 对应的属性值，它是一个对象 " );
8. **return** modelAndView;
9. }
11. }

在这个控制器中，因为MyController 没有被@RequestMapping 标记，所以当需要访问到里面使用了@RequestMapping 标记的showView 方法时，就是使用的绝对路径/showView.do 请求就可以了。

**Java代码  [收藏代码](javascript:void())**

1. @Controller
2. @RequestMapping ( "/test" )
3. **public** **class** MyController {
4. @RequestMapping ( "/showView" )
5. **public** ModelAndView showView() {
6. ModelAndView modelAndView = **new** ModelAndView();
7. modelAndView.setViewName( "viewName" );
8. modelAndView.addObject( " 需要放到 model 中的属性名称 " , " 对应的属性值，它是一个对象 " );
9. **return** modelAndView;
10. }
12. }

   这种情况是在控制器上加了@RequestMapping 注解，所以当需要访问到里面使用了@RequestMapping 标记的方法showView() 的时候就需要使用showView 方法上@RequestMapping 相对于控制器MyController上@RequestMapping 的地址，即/test/showView.do 。

#### 使用 URI 模板

   URI 模板就是在URI 中给定一个变量，然后在映射的时候动态的给该变量赋值。如URI 模板http://localhost/app/{variable1}/index.html ，这个模板里面包含一个变量variable1 ，那么当我们请求http://localhost/app/hello/index.html 的时候，该URL 就跟模板相匹配，只是把模板中的variable1 用hello 来取代。在SpringMVC 中，这种取代模板中定义的变量的值也可以给处理器方法使用，这样我们就可以非常方便的实现URL 的RestFul 风格。这个变量在SpringMVC 中是使用@PathVariable 来标记的。

   在SpringMVC 中，我们可以使用@PathVariable 来标记一个Controller 的处理方法参数，表示该参数的值将使用URI 模板中对应的变量的值来赋值。

**Java代码  [收藏代码](javascript:void())**

1. @Controller
2. @RequestMapping ( "/test/{variable1}" )
3. **public** **class** MyController {
5. @RequestMapping ( "/showView/{variable2}" )
6. **public** ModelAndView showView( @PathVariable String variable1, @PathVariable ( "variable2" ) **int** variable2) {
7. ModelAndView modelAndView = **new** ModelAndView();
8. modelAndView.setViewName( "viewName" );
9. modelAndView.addObject( " 需要放到 model 中的属性名称 " , " 对应的属性值，它是一个对象 " );
10. **return** modelAndView;
11. }
12. }

   在上面的代码中我们定义了两个URI 变量，一个是控制器类上的variable1 ，一个是showView 方法上的variable2 ，然后在showView 方法的参数里面使用@PathVariable 标记使用了这两个变量。所以当我们使用/test/hello/showView/2.do 来请求的时候就可以访问到MyController 的showView 方法，这个时候variable1 就被赋予值hello ，variable2 就被赋予值2 ，然后我们在showView 方法参数里面标注了参数variable1 和variable2 是来自访问路径的path 变量，这样方法参数variable1 和variable2 就被分别赋予hello 和2 。方法参数variable1 是定义为String 类型，variable2 是定义为int 类型，像这种简单类型在进行赋值的时候Spring 是会帮我们自动转换的，关于复杂类型该如何来转换在后续内容中将会讲到。

   在上面的代码中我们可以看到在标记variable1 为path 变量的时候我们使用的是@PathVariable ，而在标记variable2 的时候使用的是@PathVariable(“variable2”) 。这两者有什么区别呢？第一种情况就默认去URI 模板中找跟参数名相同的变量，但是这种情况只有在使用debug 模式进行编译的时候才可以，而第二种情况是明确规定使用的就是URI 模板中的variable2 变量。当不是使用debug 模式进行编译，或者是所需要使用的变量名跟参数名不相同的时候，就要使用第二种方式明确指出使用的是URI 模板中的哪个变量。

   除了在请求路径中使用URI 模板，定义变量之外，@RequestMapping 中还支持通配符“\* ”。如下面的代码我就可以使用/myTest/whatever/wildcard.do 访问到Controller 的testWildcard 方法。

**Java代码  [收藏代码](javascript:void())**

1. @Controller
2. @RequestMapping ( "/myTest" )
3. **public** **class** MyController {
4. @RequestMapping ( "\*/wildcard" )
5. **public** String testWildcard() {
6. System. out .println( "wildcard------------" );
7. **return** "wildcard" ;
8. }
9. }

#### 使用 @RequestParam 绑定 HttpServletRequest 请求参数到控制器方法参数

**Java代码  [收藏代码](javascript:void())**

1. @RequestMapping ( "requestParam" )
2. ublic String testRequestParam( @RequestParam(required=**false**) String name, @RequestParam ( "age" ) **int** age) {
3. **return** "requestParam" ;
4. }

在上面代码中利用@RequestParam 从HttpServletRequest 中绑定了参数name 到控制器方法参数name ，绑定了参数age 到控制器方法参数age 。值得注意的是和@PathVariable 一样，当你没有明确指定从request 中取哪个参数时，Spring 在代码是debug 编译的情况下会默认取更方法参数同名的参数，如果不是debug 编译的就会报错。此外，当需要从request 中绑定的参数和方法的参数名不相同的时候，也需要在@RequestParam 中明确指出是要绑定哪个参数。在上面的代码中如果我访问/requestParam.do?name=hello&age=1 则Spring 将会把request请求参数name 的值hello 赋给对应的处理方法参数name ，把参数age 的值1 赋给对应的处理方法参数age 。

在@RequestParam 中除了指定绑定哪个参数的属性value 之外，还有一个属性required ，它表示所指定的参数是否必须在request 属性中存在，默认是true ，表示必须存在，当不存在时就会报错。在上面代码中我们指定了参数name 的required 的属性为false ，而没有指定age 的required 属性，这时候如果我们访问/requestParam.do而没有传递参数的时候，系统就会抛出异常，因为age 参数是必须存在的，而我们没有指定。而如果我们访问/requestParam.do?age=1 的时候就可以正常访问，因为我们传递了必须的参数age ，而参数name 是非必须的，不传递也可以。

#### 使用 @CookieValue 绑定 cookie 的值到 Controller 方法参数

**Java代码  [收藏代码](javascript:void())**

1. @RequestMapping ( "cookieValue" )
2. **public** String testCookieValue( @CookieValue ( "hello" ) String cookieValue, @CookieValue String hello) {
3. System. out .println(cookieValue + "-----------" + hello);
4. **return** "cookieValue" ;
5. }

    在上面的代码中我们使用@CookieValue 绑定了cookie 的值到方法参数上。上面一共绑定了两个参数，一个是明确指定要绑定的是名称为hello 的cookie 的值，一个是没有指定。使用没有指定的形式的规则和@PathVariable、@RequestParam 的规则是一样的，即在debug 编译模式下将自动获取跟方法参数名同名的cookie 值。

#### 使用 @RequestHeader 注解绑定 HttpServletRequest 头信息到Controller 方法参数

**Java代码  [收藏代码](javascript:void())**

1. @RequestMapping ( "testRequestHeader" )
2. **public** String testRequestHeader( @RequestHeader ( "Host" ) String hostAddr, @RequestHeader String Host, @RequestHeader String host ) {
3. System. out .println(hostAddr + "-----" + Host + "-----" + host );
4. **return** "requestHeader" ;
5. }

         在上面的代码中我们使用了 @RequestHeader 绑定了 HttpServletRequest 请求头 host 到Controller 的方法参数。上面方法的三个参数都将会赋予同一个值，由此我们可以知道在绑定请求头参数到方法参数的时候规则和 @PathVariable 、 @RequestParam 以及 @CookieValue 是一样的，即没有指定绑定哪个参数到方法参数的时候，在 debug 编译模式下将使用方法参数名作为需要绑定的参数。但是有一点 @RequestHeader 跟另外三种绑定方式是不一样的，那就是在使用 @RequestHeader 的时候是大小写不敏感的，即 @RequestHeader(“Host”) 和 @RequestHeader(“host”) 绑定的都是 Host 头信息。记住在 @PathVariable 、 @RequestParam 和 @CookieValue 中都是大小写敏感的。

#### @RequestMapping 的一些高级应用

         在RequestMapping 中除了指定请求路径value 属性外，还有其他的属性可以指定，如params 、method 和headers 。这样属性都可以用于缩小请求的映射范围。

##### params属性

   params 属性用于指定请求参数的，先看以下代码。

**Java代码  [收藏代码](javascript:void())**

1. @RequestMapping (value= "testParams" , params={ "param1=value1" , "param2" , "!param3" })
2. **public** String testParams() {
3. System. out .println( "test Params..........." );
4. **return** "testParams" ;
5. }

   在上面的代码中我们用@RequestMapping 的params 属性指定了三个参数，这些参数都是针对请求参数而言的，它们分别表示参数param1 的值必须等于value1 ，参数param2 必须存在，值无所谓，参数param3 必须不存在，只有当请求/testParams.do 并且满足指定的三个参数条件的时候才能访问到该方法。所以当请求/testParams.do?param1=value1&param2=value2 的时候能够正确访问到该testParams 方法，当请求/testParams.do?param1=value1&param2=value2&param3=value3 的时候就不能够正常的访问到该方法，因为在@RequestMapping 的params 参数里面指定了参数param3 是不能存在的。

##### method属性

   method 属性主要是用于限制能够访问的方法类型的。

**Java代码  [收藏代码](javascript:void())**

1. @RequestMapping (value= "testMethod" , method={RequestMethod. GET , RequestMethod. DELETE })
2. **public** String testMethod() {
3. **return** "method" ;
4. }

在上面的代码中就使用method 参数限制了以GET 或DELETE 方法请求/testMethod.do 的时候才能访问到该Controller 的testMethod 方法。

##### headers属性

         使用headers 属性可以通过请求头信息来缩小@RequestMapping 的映射范围。

**Java代码  [收藏代码](javascript:void())**

1. @RequestMapping (value= "testHeaders" , headers={ "host=localhost" , "Accept" })
2. **public** String testHeaders() {
3. **return** "headers" ;
4. }

   headers 属性的用法和功能与params 属性相似。在上面的代码中当请求/testHeaders.do 的时候只有当请求头包含Accept 信息，且请求的host 为localhost 的时候才能正确的访问到testHeaders 方法。

#### 以 @RequestMapping 标记的处理器方法支持的方法参数和返回类型

##### 支持的方法参数类型

**（1 ）HttpServlet 对象，主要包括HttpServletRequest 、HttpServletResponse 和HttpSession 对象。**这些参数Spring 在调用处理器方法的时候会自动给它们赋值，所以当在处理器方法中需要使用到这些对象的时候，可以直接在方法上给定一个方法参数的申明，然后在方法体里面直接用就可以了。但是有一点需要注意的是在使用HttpSession 对象的时候，如果此时HttpSession 对象还没有建立起来的话就会有问题。

**（2 ）Spring 自己的WebRequest 对象。**使用该对象可以访问到存放在HttpServletRequest 和HttpSession中的属性值。

**（3 ）InputStream 、OutputStream 、Reader 和Writer 。**InputStream 和Reader 是针对HttpServletRequest 而言的，可以从里面取数据；OutputStream 和Writer 是针对HttpServletResponse 而言的，可以往里面写数据。

**（4 ）使用@PathVariable 、@RequestParam 、@CookieValue 和@RequestHeader 标记的参数。**

**（5 ）使用@ModelAttribute 标记的参数。**

**（6 ）java.util.Map 、Spring 封装的Model 和ModelMap 。**这些都可以用来封装模型数据，用来给视图做展示。

**（7 ）实体类。**可以用来接收上传的参数。

**（8 ）Spring 封装的MultipartFile 。**用来接收上传文件的。

**（9 ）Spring 封装的Errors 和BindingResult 对象。**这两个对象参数必须紧接在需要验证的实体对象参数之后，它里面包含了实体对象的验证结果。

##### 支持的返回类型

         （1 ）一个包含模型和视图的ModelAndView 对象。

   （2 ）一个模型对象，这主要包括Spring 封装好的Model 和ModelMap ，以及java.util.Map ，当没有视图返回的时候视图名称将由RequestToViewNameTranslator 来决定。

   （3 ）一个View 对象。这个时候如果在渲染视图的过程中模型的话就可以给处理器方法定义一个模型参数，然后在方法体里面往模型中添加值。

   （4 ）一个String 字符串。这往往代表的是一个视图名称。这个时候如果需要在渲染视图的过程中需要模型的话就可以给处理器方法一个模型参数，然后在方法体里面往模型中添加值就可以了。

   （5 ）返回值是void 。这种情况一般是我们直接把返回结果写到HttpServletResponse 中了，如果没有写的话，那么Spring 将会利用RequestToViewNameTranslator 来返回一个对应的视图名称。如果视图中需要模型的话，处理方法与返回字符串的情况相同。

   （6 ）如果处理器方法被注解@ResponseBody 标记的话，那么处理器方法的任何返回类型都会通过HttpMessageConverters 转换之后写到HttpServletResponse 中，而不会像上面的那些情况一样当做视图或者模型来处理。

   （7 ）除以上几种情况之外的其他任何返回类型都会被当做模型中的一个属性来处理，而返回的视图还是由RequestToViewNameTranslator 来决定，添加到模型中的属性名称可以在该方法上用@ModelAttribute(“attributeName”) 来定义，否则将使用返回类型的类名称的首字母小写形式来表示。使用@ModelAttribute 标记的方法会在@RequestMapping 标记的方法执行之前执行。

#### 使用 @ModelAttribute 和 @SessionAttributes 传递和保存数据

       SpringMVC 支持使用 @ModelAttribute 和 @SessionAttributes 在不同的模型和控制器之间共享数据。 **@ModelAttribute**主要有两种使用方式，一种是标注在方法上，一种是标注在 Controller 方法参数上。

当 @ModelAttribute 标记在方法上的时候，该方法将在处理器方法执行之前执行，然后把返回的对象存放在 session 或模型属性中，属性名称可以使用 @ModelAttribute(“attributeName”) 在标记方法的时候指定，若未指定，则使用返回类型的类名称（首字母小写）作为属性名称。关于 @ModelAttribute 标记在方法上时对应的属性是存放在 session 中还是存放在模型中，我们来做一个实验，看下面一段代码。

**Java代码  [收藏代码](javascript:void())**

1. @Controller
2. @RequestMapping ( "/myTest" )
3. **public** **class** MyController {
5. @ModelAttribute ( "hello" )
6. **public** String getModel() {
7. System. out .println( "-------------Hello---------" );
8. **return** "world" ;
9. }
11. @ModelAttribute ( "intValue" )
12. **public** **int** getInteger() {
13. System. out .println( "-------------intValue---------------" );
14. **return** 10;
15. }
17. @RequestMapping ( "sayHello" )
18. **public** **void** sayHello( @ModelAttribute ( "hello" ) String hello, @ModelAttribute ( "intValue" ) **int** num, @ModelAttribute ( "user2" ) User user, Writer writer, HttpSession session) **throws** IOException {
19. writer.write( "Hello " + hello + " , Hello " + user.getUsername() + num);
20. writer.write( "\r" );
21. Enumeration enume = session.getAttributeNames();
22. **while** (enume.hasMoreElements())
23. writer.write(enume.nextElement() + "\r" );
24. }
26. @ModelAttribute ( "user2" )
27. **public** User getUser() {
28. System. out .println( "---------getUser-------------" );
29. **return** **new** User(3, "user2" );
30. }
31. }

当我们请求 /myTest/sayHello.do 的时候使用 @ModelAttribute 标记的方法会先执行，然后把它们返回的对象存放到模型中。最终访问到 sayHello 方法的时候，使用 @ModelAttribute 标记的方法参数都能被正确的注入值。执行结果如下图所示：

http://dl.iteye.com/upload/attachment/0078/3377/06750228-d8ab-336b-bb00-06455d11c895.png

       由执行结果我们可以看出来，此时 session 中没有包含任何属性，也就是说上面的那些对象都是存放在模型属性中，而不是存放在 session 属性中。那要如何才能存放在 session 属性中呢？这个时候我们先引入一个新的概念 @SessionAttributes ，它的用法会在讲完 @ModelAttribute 之后介绍，这里我们就先拿来用一下。我们在 MyController 类上加上 @SessionAttributes 属性标记哪些是需要存放到 session 中的。看下面的代码：

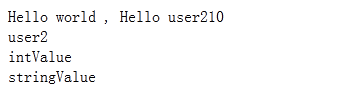
**Java代码  [收藏代码](javascript:void())**

1. @Controller
2. @RequestMapping ( "/myTest" )
3. @SessionAttributes (value={ "intValue" , "stringValue" }, types={User. **class** })
4. **public** **class** MyController {
6. @ModelAttribute ( "hello" )
7. **public** String getModel() {
8. System. out .println( "-------------Hello---------" );
9. **return** "world" ;
10. }
12. @ModelAttribute ( "intValue" )
13. **public** **int** getInteger() {
14. System. out .println( "-------------intValue---------------" );
15. **return** 10;
16. }
18. @RequestMapping ( "sayHello" )
19. **public** **void** sayHello(Map<String, Object> map, @ModelAttribute ( "hello" ) String hello, @ModelAttribute ( "intValue" ) **int** num, @ModelAttribute ( "user2" ) User user, Writer writer, HttpServletRequest request) **throws** IOException {
20. map.put( "stringValue" , "String" );
21. writer.write( "Hello " + hello + " , Hello " + user.getUsername() + num);
22. writer.write( "\r" );
23. HttpSession session = request.getSession();
24. Enumeration enume = session.getAttributeNames();
25. **while** (enume.hasMoreElements())
26. writer.write(enume.nextElement() + "\r" );
27. System. out .println(session);
28. }
30. @ModelAttribute ( "user2" )
31. **public** User getUser() {
32. System. out .println( "---------getUser-------------" );
33. **return** **new** User(3, "user2" );
34. }
35. }

       在上面代码中我们指定了属性为 intValue 或 stringValue 或者类型为 User 的都会放到 Session中，利用上面的代码当我们访问 /myTest/sayHello.do 的时候，结果如下：

http://dl.iteye.com/upload/attachment/0078/3377/06750228-d8ab-336b-bb00-06455d11c895.png

       仍然没有打印出任何 session 属性，这是怎么回事呢？怎么定义了把模型中属性名为 intValue 的对象和类型为 User 的对象存到 session 中，而实际上没有加进去呢？难道我们错啦？我们当然没有错，只是在第一次访问 /myTest/sayHello.do 的时候 @SessionAttributes 定义了需要存放到 session 中的属性，而且这个模型中也有对应的属性，但是这个时候还没有加到 session 中，所以 session 中不会有任何属性，等处理器方法执行完成后 Spring 才会把模型中对应的属性添加到 session 中。所以当请求第二次的时候就会出现如下结果：



当 @ModelAttribute 标记在处理器方法参数上的时候，表示该参数的值将从模型或者 Session 中取对应名称的属性值，该名称可以通过 @ModelAttribute(“attributeName”) 来指定，若未指定，则使用参数类型的类名称（首字母小写）作为属性名称。

**Java代码  [收藏代码](javascript:void())**

1. @Controller
2. @RequestMapping ( "/myTest" )
3. **public** **class** MyController {
5. @ModelAttribute ( "hello" )
6. **public** String getModel() {
7. **return** "world" ;
8. }
10. @RequestMapping ( "sayHello" )
11. **public** **void** sayHello( @ModelAttribute ( "hello" ) String hello, Writer writer) **throws** IOException {
12. writer.write( "Hello " + hello);
13. }
14. }

在上面代码中，当我们请求/myTest/sayHello.do 的时候，由于MyController 中的方法getModel 使用了注解@ModelAttribute 进行标记，所以在执行请求方法sayHello 之前会先执行getModel 方法，这个时候getModel 方法返回一个字符串world 并把它以属性名hello 保存在模型中，接下来访问请求方法sayHello 的时候，该方法的hello参数使用@ModelAttribute(“hello”) 进行标记，这意味着将从session 或者模型中取属性名称为hello 的属性值赋给hello 参数，所以这里hello 参数将被赋予值world ，所以请求完成后将会在页面上看到Hello world 字符串。

**@SessionAttributes**用于标记需要在Session 中使用到的数据，包括从Session 中取数据和存数据。@SessionAttributes 一般是标记在Controller 类上的，可以通过名称、类型或者名称加类型的形式来指定哪些属性是需要存放在session 中的。

**Java代码  [收藏代码](javascript:void())**

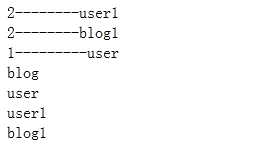
1. @Controller
2. @RequestMapping ( "/myTest" )
3. @SessionAttributes (value={ "user1" , "blog1" }, types={User. **class** , Blog. **class** })
4. **public** **class** MyController {
6. @RequestMapping ( "setSessionAttribute" )
7. **public** **void** setSessionAttribute(Map<String, Object> map, Writer writer) **throws** IOException {
8. User user = **new** User(1, "user" );
9. User user1 = **new** User(2, "user1" );
10. Blog blog = **new** Blog(1, "blog" );
11. Blog blog1 = **new** Blog(2, "blog1" );
12. map.put( "user" , user);
13. map.put( "user1" , user1);
14. map.put( "blog" , blog);
15. map.put( "blog1" , blog1);
16. writer.write( "over." );
17. }


21. @RequestMapping ( "useSessionAttribute" )
22. **public** **void** useSessionAttribute(Writer writer, @ModelAttribute ( "user1" ) User user1, @ModelAttribute ( "blog1" ) Blog blog1) **throws** IOException {
23. writer.write(user1.getId() + "--------" + user1.getUsername());
24. writer.write( "\r" );
25. writer.write(blog1.getId() + "--------" + blog1.getTitle());
26. }
28. @RequestMapping ( "useSessionAttribute2" )
29. **public** **void** useSessionAttribute(Writer writer, @ModelAttribute ( "user1" ) User user1, @ModelAttribute ( "blog1" ) Blog blog1, @ModelAttribute User user, HttpSession session) **throws** IOException {
30. writer.write(user1.getId() + "--------" + user1.getUsername());
31. writer.write( "\r" );
32. writer.write(blog1.getId() + "--------" + blog1.getTitle());
33. writer.write( "\r" );
34. writer.write(user.getId() + "---------" + user.getUsername());
35. writer.write( "\r" );
36. Enumeration enume = session.getAttributeNames();
37. **while** (enume.hasMoreElements())
38. writer.write(enume.nextElement() + " \r" );
39. }
41. @RequestMapping ( "useSessionAttribute3" )
42. **public** **void** useSessionAttribute( @ModelAttribute ( "user2" ) User user) {
44. }
45. }

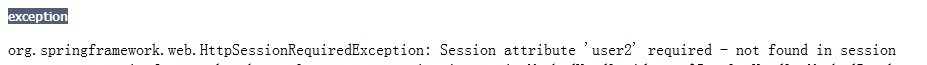
   在上面代码中我们可以看到在MyController 上面使用了@SessionAttributes 标记了需要使用到的Session 属性。可以通过名称和类型指定需要存放到Session 中的属性，对应@SessionAttributes 注解的value 和types 属性。当使用的是types 属性的时候，那么使用的Session 属性名称将会是对应类型的名称（首字母小写）。当value 和types 两个属性都使用到了的时候，这时候取的是它们的并集，而不是交集，所以上面代码中指定要存放在Session 中的属性有名称为user1 或blog1 的对象，或类型为User 或Blog 的对象。在上面代码中我们首先访问/myTest/setSessionAttribute.do ，该请求将会请求到MyController 的setSessionAttribute 方法，在该方法中，我们往模型里面添加了user 、user1 、blog 和blog1 四个属性，因为它们或跟类上的@SessionAttributes 定义的需要存到session 中的属性名称相同或类型相同，所以在请求完成后这四个属性都将添加到session 属性中。接下来访问/myTest/useSessionAttribute.do ，该请求将会请求MyController 的useSessionAttribute(Writer writer, @ModelAttribute(“user1”) User user1, @ModelAttribute(“blog1”) Blog blog) 方法，该方法参数中用@ModelAttribute 指定了参数user1 和参数blog1 是需要从session 或模型中绑定的，恰好这个时候session 中已经有了这两个属性，所以这个时候在方法执行之前会先绑定这两个参数。执行结果如下图所示：

http://dl.iteye.com/upload/attachment/0078/3381/9c9232dc-5444-3d2c-a16a-e69b8ee560a2.png

   接下来访问/myTest/useSessionAttribute2.do ，这个时候请求的是上面代码中对应的第二个useSessionAttribute 方法，方法参数user 、user1 和blog1 用@ModelAttribute 声明了需要session 或模型属性注入，我们知道在请求/myTest/setSessionAttribute.do 的时候这些属性都已经添加到了session 中，所以该请求的结果会如下图所示：



   接下来访问/myTest/useSessionAttribute3.do ，这个时候请求的是上面代码中对应的第三个useSessionAttribute 方法，我们可以看到该方法的方法参数user 使用了@ModelAttribute(“user2”) 进行标记，表示user 需要session 中的user2 属性来注入，但是这个时候我们知道session 中是不存在user2 属性的，所以这个时候就会报错了。执行结果如图所示：



#### 定制自己的类型转换器

         在通过处理器方法参数接收 request 请求参数绑定数据的时候，对于一些简单的数据类型 Spring 会帮我们自动进行类型转换，而对于一些复杂的类型由于 Spring 没法识别，所以也就不能帮助我们进行自动转换了，这个时候如果我们需要 Spring 来帮我们自动转换的话就需要我们给 Spring 注册一个对特定类型的识别转换器。 Spring 允许我们提供两种类型的识别转换器，一种是注册在 Controller 中的，一种是注册在SpringMVC 的配置文件中。聪明的读者看到这里应该可以想到它们的区别了，定义在 Controller 中的是局部的，只在当前 Controller 中有效，而放在 SpringMVC 配置文件中的是全局的，所有 Controller 都可以拿来使用。

#### 在 @InitBinder 标记的方法中定义局部的类型转换器

         我们可以使用 @InitBinder 注解标注在 Controller 方法上，然后在方法体里面注册数据绑定的转换器，这主要是通过 WebDataBinder 进行的。我们可以给需要注册数据绑定的转换器的方法一个WebDataBinder 参数，然后给该方法加上 @InitBinder 注解，这样当该 Controller 中在处理请求方法时如果发现有不能解析的对象的时候，就会看该类中是否有使用 @InitBinder 标记的方法，如果有就会执行该方法，然后看里面定义的类型转换器是否与当前需要的类型匹配。

**Java代码  [收藏代码](javascript:void())**

1. @Controller
2. @RequestMapping ( "/myTest" )
3. **public** **class** MyController {
5. @InitBinder
6. **public** **void** dataBinder(WebDataBinder binder) {
7. DateFormat dateFormat = **new** SimpleDateFormat( "yyyyMMdd" );
8. PropertyEditor propertyEditor = **new** CustomDateEditor(dateFormat, **true** ); // 第二个参数表示是否允许为空
9. binder.registerCustomEditor(Date. **class** , propertyEditor);
10. }
12. @RequestMapping ( "dataBinder/{date}" )
13. **public** **void** testDate( @PathVariable Date date, Writer writer) **throws** IOException {
14. writer.write(String.valueOf (date.getTime()));
15. }
17. }

       在上面的代码中当我们请求 /myTest/dataBinder/20121212.do 的时候， Spring 就会利用@InitBinder 标记的方法里面定义的类型转换器把字符串 20121212 转换为一个 Date 对象。这样定义的类型转换器是局部的类型转换器，一旦出了这个 Controller 就不会再起作用。类型转换器是通过WebDataBinder 对象的 registerCustomEditor 方法来注册的，要实现自己的类型转换器就要实现自己的 PropertyEditor 对象。 Spring 已经给我们提供了一些常用的属性编辑器，如 CustomDateEditor 、CustomBooleanEditor 等。

       PropertyEditor 是一个接口，要实现自己的 PropertyEditor 类我们可以实现这个接口，然后实现里面的方法。但是 PropertyEditor 里面定义的方法太多了，这样做比较麻烦。在 java 中有一个封装类是实现了 PropertyEditor 接口的，它是 PropertyEditorSupport 类。所以如果需要实现自己的PropertyEditor 的时候只需要继承 PropertyEditorSupport 类，然后重写其中的一些方法。一般就是重写 setAsText 和 getAsText 方法就可以了， setAsText 方法是用于把字符串类型的值转换为对应的对象的，而 getAsText 方法是用于把对象当做字符串来返回的。在 setAsText 中我们一般先把字符串类型的对象转为特定的对象，然后利用 PropertyEditor 的 setValue 方法设定转换后的值。在 getAsText 方法中一般先使用 getValue 方法取代当前的对象，然后把它转换为字符串后再返回给 getAsText 方法。下面是一个示例：

**Java代码  [收藏代码](javascript:void())**

1. @InitBinder
2. **public** **void** dataBinder(WebDataBinder binder) {
3. // 定义一个 User 属性编辑器
4. PropertyEditor userEditor = **new** PropertyEditorSupport() {
6. @Override
7. **public** String getAsText() {
8. // TODO Auto-generated method stub
9. User user = (User) getValue();
10. **return** user.getUsername();
11. }
13. @Override
14. **public** **void** setAsText(String userStr) **throws** IllegalArgumentException {
15. // TODO Auto-generated method stub
16. User user = **new** User(1, userStr);
17. setValue(user);
18. }
19. };
20. // 使用 WebDataBinder 注册 User 类型的属性编辑器
21. binder.registerCustomEditor(User. **class** , userEditor);
22. }

#### 实现 WebBindingInitializer 接口定义全局的类型转换器

       如果需要定义全局的类型转换器就需要实现自己的 WebBindingInitializer 对象，然后把该对象注入到AnnotationMethodHandlerAdapter 中，这样 Spring 在遇到自己不能解析的对象的时候就会到全局的WebBindingInitializer 的 initBinder 方法中去找，每次遇到不认识的对象时， initBinder 方法都会被执行一遍。

**Java代码  [收藏代码](javascript:void())**

1. **public** **class** MyWebBindingInitializer **implements** WebBindingInitializer {
3. @Override
4. **public** **void** initBinder(WebDataBinder binder, WebRequest request) {
5. // TODO Auto-generated method stub
6. DateFormat dateFormat = **new** SimpleDateFormat( "yyyyMMdd" );
7. PropertyEditor propertyEditor = **new** CustomDateEditor(dateFormat, **true** );
8. binder.registerCustomEditor(Date. **class** , propertyEditor);
9. }
11. }

定义了这么一个 WebBindingInitializer 对象之后 Spring 还是不能识别其中指定的对象，这是因为我们只是定义了 WebBindingInitializer 对象，还没有把它交给 Spring ， Spring 不知道该去哪里找解析器。要让 Spring 能够识别还需要我们在 SpringMVC 的配置文件中定义一个AnnotationMethodHandlerAdapter 类型的 bean 对象，然后利用自己定义的WebBindingInitializer 覆盖它的默认属性 webBindingInitializer 。

**Xml代码  [收藏代码](javascript:void())**

1. **<** **bean** class = "org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter" **>**
2. **<** **property** name = "webBindingInitializer" **>**
3. **<** **bean** class = "com.host.app.web.util.MyWebBindingInitializer" **/>**
4. **</** **property** **>**
5. **</** **bean** **>**

#### 触发数据绑定方法的时间

当Controller处理器方法参数使用@RequestParam、@PathVariable、@RequestHeader、@CookieValue和@ModelAttribute标记的时候都会触发initBinder方法的执行，这包括使用WebBindingInitializer定义的全局方法和在Controller中使用@InitBinder标记的局部方法。而且每个使用了这几个注解标记的参数都会触发一次initBinder方法的执行，这也意味着有几个参数使用了上述注解就会触发几次initBinder方法的执行。

## 视图层（View）

在实体层中会用到哪些实体类，他们怎样相互转换？

单个的HTML页面，如何转成系统的JSP文件

1、改文件后缀名，将 .html 改为 .jsp

2、在每个文件的开头地方，指定页面的编码类型，引入标签库，引入类库，定义一些公共变量。

指定页面编码类型：

<%@ page contentType="text/html;charset=UTF-8" language="java" %>

引入JSTL标签库：

<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<%@ taglib uri="http://java.sun.com/jsp/jstl/functions" prefix="fn" %>

<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>

引入Spring标签库：

<%@ taglib uri="http://www.springframework.org/tags" prefix="spring" %>

定义一些变量：

<% request.setAttribute("urlPath",request.getContextPath()); %>

<spring:url var="resources" value="${urlPath}/resources" />

注意：JSP页面中同时包含了Java代码块和标签语言，在Java代码块中不能引用标签语言定义的变量，但在标签中可以引用Java的request对象的变量。

例如:

<% request.setAttribute("resources",request.getContextPath()+"/resources"); %>

<img src="${resources}/images/hs1.jpg" class="g-left">

在标签中可以引用其他标签定义的变量，例如JSTL标签中引用Spring标签的变量：

<spring:url var="resources" value="${urlPath}/resources" />

<img src="${resources}/images/hs1.jpg" class="g-left">

标签中不能直接引用Java代码块的普通变量，必须要引用request对象中的变量，例如这样的引用是无效的：

<% String resources = request.getContextPath()+"/resources"; %>

<img src="${resources}/images/hs1.jpg" class="g-left">

在JSP页面的java代码中：

System.out.println("") 输出信息到服务端的控制台

out.println("") 打印信息到页面

修改图片、样式、脚本、链接的路径地址。

编写与服务器交互部分

## 日志

### log4j

### slf4j

### SLF4J+Log4J

SLF4J(Simple Logging Facade for Java) 是一个通用的日志框架，不能何以谓之 Facade(门面)，所扮眼的角色相当于 Jakarta Commons Logging。就像 JCL 需要底层的日志实现，如 Log4J、java.util.logging、Simple Logger 等来完成具体的信息输出，事实上基本总是 JCL+Log4J 那么一个绝配。SLF4J 的原旨也是能支持多种下层日志框架实现，但最好的日志实现仍然是 Log4J，所以本篇讲述 SLF4J 的第一种用法 SLF4J+Log4J。

 需要的配置文件和组件包，下面三个 jar 文件和一个 properties 文件都是要放在项目的 ClassPath 上。   
  
1. slf4j-api-1.5.11.jar   
2. slf4j-log4j12-1.5.11.jar   
3. log4j-1.2.15.jar   
4. log4j.properties(也可以是 log4j.xml，本例中用 log4j.propertes)

log4j.properties

**[java]** [view plaincopyprint?](http://blog.csdn.net/huangbiao86/article/details/6575894)

1. log4j.appender.stdout=org.apache.log4j.ConsoleAppender
2. log4j.appender.stdout.Target=System.out
3. log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
4. log4j.appender.stdout.layout.ConversionPattern=%d{ABSOLUTE} %5p %c{1}:%L - %m%n
5. log4j.appender.db=org.apache.log4j.FileAppender
6. log4j.appender.db.File=${s2sh.root}/WEB-INF/logs/ssh.log
7. log4j.appender.db.Append=**true**
8. log4j.appender.db.layout=org.apache.log4j.PatternLayout
9. log4j.appender.db.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss} [%-5p][%c{1}] - %m%n

使用 SLF4J  的代码：

**[java]** [view plaincopyprint?](http://blog.csdn.net/huangbiao86/article/details/6575894)

1. **import** org.slf4j.Logger;
2. **import** org.slf4j.LoggerFactory;
4. **public** **class** TestSlf4J {
5. **private** **static** **final** Logger logger = LoggerFactory.getLogger(TestSlf4j.**class**);
6. **public** **static** **void** main(String[] args) {
7. logger.info("Hello {}","SLF4J");
8. }
9. }

执行它，控制台输出： 

**[java]** [view plaincopyprint?](http://blog.csdn.net/huangbiao86/article/details/6575894)

1. 2010-04-07 17:14:51,390 [com.unmi.TestSlf4j]-[INFO] Hello SLF4J

把这种 SLF4J+Log4J 的使用模式与曾为霸主地位的 JCL+Log4J 的用法进行一下对比(请忽略掉包文件中的版本号)：

把这种 SLF4J+Log4J 的使用模式与曾为霸主地位的 JCL+Log4J 的用法进行一下对比(请忽略掉包文件中的版本号)：   
  
SLF4J+Log4j 组合 对比 JCL+Log4J 组合   
slf4j-api-1.5.11.jar 相当，定义高层 API commons-logging-1.1.jar   
slf4j-log4j12-1.5.11.jar 相当，左边是用绑定包，右边   
是用配置文件来指定日志实现 commons-logging.properties，内容为：   
org.apache.commons.logging.LogFactory=   
org.apache.commons.logging.impl.LogFactoryImpl   
或者   
org.apache.commons.logging.Log=   
org.apache.commons.logging.impl.Log4JLogger   
log4j-1.2.15.jar 一样 log4j-1.2.15.jar   
log4j.properties 一样，原来怎么配置现在也是 log4j.properties   
程序代码中：   
import org.slf4j.Logger;   
import org.slf4j.LoggerFactory;   
  
Logger logger = LoggerFactory.getLogger(TestSlf4j.class);   
logger.info("Hello {}","SLF4J"); 左边侵入的是 SLF4J API，右边是被 JCL 的 API 污染了   
  
SLF4J 支持参数化，而 JCL 不能 程序代码中：   
import org.apache.commons.logging.Log;   
import org.apache.commons.logging.LogFactory;   
  
Log log = LogFactory.getLog(TestJCL.class);   
  
log.info("Hello JCL");   
  
从上面的对比来看，SLF4j+Log4j 与 JCL+Log4J 的使用方式差不多，主要差异就在 SLF4J 用 jar 来告知用哪种日志实现，而 JCL 是通过配置文件来获得该选择哪个日志实现。

为什么会兴起 SLF4J，看看我们原来哪一个框架中，大的如 SSH 三雄(Spring、Struts、Hibernate)，还有 WAS 应用服务器，小的就不计其数以前用的通用日志框架都清一色的 Jakarta Commons Logging(JCL)，日志实现会选用 Log4j，为何现在 Hibernate、Tapesty、DbUnit、Jetty V6 等纷纷变节，都采用了 SLF4J 了呢？SLF4J 与 JCL 相比，定然是有其可表之处。而其中 SLF4J 受类加载器的影响较小，不易产生内存溢出的问题，性能得到了改善，更主要是顺应了潮流的发展--可方便部署到 OSGI 环境中.

### SLF4J+LogBack

前面讲的 SLF4J 的用法之一是 SLF4J+Log4J，而这里要推出的组合是 SLF4J+LogBack。不用 Log4J？难道还有比 Log4J 更好的日志实现吗？是的，答案就是 LogBack。假如你知道 LogBack 和 Log4J 是同出一位大师之手，你就不会觉得惊讶了。LogBack 对 Log4J 进行了增强，并投入了 SLF4J 的怀抱，它还被作者认定为 Log4J 的继任者。

Logback 分为三个模块：logback-core，logback-classic，logback-access。logback-core 是核心；logback-classic 改善了 log4j，且自身实现了 SLF4J API，所以即使用 Logback 你仍然可以使用其他的日志实现，如原始的 Log4J，java.util.logging 等；logback-access 让你方便的访问日志信息，如通过 http 的方式。   
  
还得提一点，Logback 能更好的放到 OSGI 环境中。好了，简单绍完了 Logback，就来看看具体怎么用它。   
  
需要的配置文件和组件包，下面三个 jar 文件和一个 xml文件都是要放在项目的 ClassPath 上。

slf4j-api-1.5.11.jar   
logback-core-0.9.20.jar   
logback-classic-0.9.20.jar   
logback.xml 或 logback-test.xml (类似于 log4j.properties/log4j.xml)

Logback 先找 logback-test.xml，没有则找 logback.xml 文件，都找不到就使用 BasicConfigurator 基本配置，BasicConfigurator 就是相当于等会贴出的 logback.xml 文件内容的配置。这里我们用 logback.xml 配置文件，以前的 log4j.properties  文件可以用 PropertiesTranslator 转换成 logback.xml 文件内容。

下面是一个最简单的 logback.xml 文件内容

**[xhtml]** [view plaincopyprint?](http://blog.csdn.net/huangbiao86/article/details/6575894)

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. **<configuration>**
3. **<appender** name="stdout" class="ch.qos.logback.core.ConsoleAppender"**>**
4. **<encoder** charset="GBK"**>**
5. **<pattern>**%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n**</pattern>**
6. **</encoder>**
7. **</appender>**
8. **<root** level="DEBUG"**>**
9. **<appender-ref** ref="stdout" **/>**
10. **</root>**
11. **</configuration>**

在 log4j.xml 能做的配置，logback.xml 中也能做到，而且还增强了诸如 <if><then><else>、<filter>、<sift> 等更强的控制，请参考 logback 的手册 <http://logback.qos.ch/manual/index.html>。

使用 Logback  的代码

**[xhtml]** [view plaincopyprint?](http://blog.csdn.net/huangbiao86/article/details/6575894)

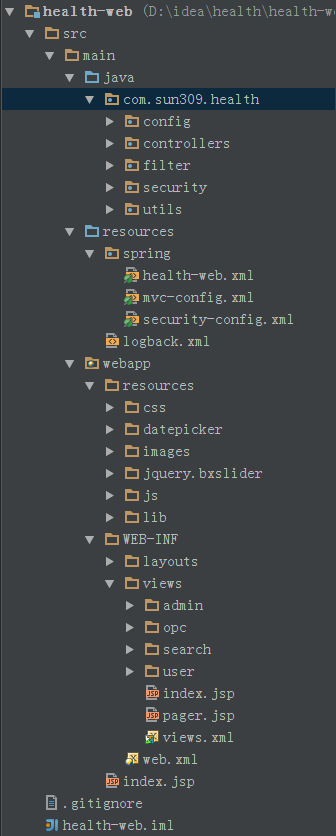
1. package com.unmi;
3. import org.slf4j.Logger;
4. import org.slf4j.LoggerFactory;
6. public class TestLogback {
7. private static final Logger logger = LoggerFactory.getLogger(TestLogback.class);
9. public static void main(String[] args) {
10. logger.info("Hello {}","TestLogback");
11. }
12. }

可以看到与使用 SLF4J 的代码没有任何区别，因为本来用的通用日志框架就是 SLF4J，说到底，这里的 Logback 就相当于 SLF4J+Log4J 使用方式中的 Log4J 绑定和 Log4J 的功能。

执行上面的代码，输出：   
  
21:38:43.031 [main] INFO  com.unmi.TestLogback - Hello TestLogback

## 外围系统远程调度

## 目录结构



# maven

## Eclipse创建Maven项目部署war包到外部tomcat

Eclipse+Maven创建webapp项目<二>

1、开启eclipse，右键new——》other，如下图找到maven project



2、选择maven project，显示创建maven项目的窗口，此次不勾选Create a simple project，直接next，如图所示：



3、在Filter中搜索webapp，选择如图所示，并next：



4、输入maven项目的基本信息，如下图所示：



5、完成maven项目的创建，生成相应的webapp项目结构，如下所示，此处有部分结构是项目不需要的，我们需要去掉：



6、选择项目，右键选择Properties，进入属性页面，选择到Project Facets菜单下，如下图所示：



7、选择java版本为1.7，并去掉其他两项，如下图：



8、点击ok之后，再次回到项目结构，此时项目结构比较清晰，符合我们想要创建的maven项目：



9、创建没有显示的源代码文件夹，如下图所示：





10、最后得到完成的maven webapp项目结构如下：



11、右击项目，执行 Run s -> Maven install . 在项目的target目录下生成对应的war包maven.archetype.webapp.war 。

12、将生成的war包拷到tomcat的webapps目录下



启动tomcat，在浏览器输入地址：http://localhost:8080/maven.archetype.webapp/ 即可访问。



正常情况tomcat启动时会把war把解压，生成一个同名的文件夹在webapps目录下。如果有的tomcat不能解压war包导致war包不能正常运行，则把正常解压后的项目文件拷到这个tomcat下就OK了。

例如SunPlatform.war 在本机的tomcat上能正常运行，在服务器上就不能。则把本机tomcat启动后解压出来的SunPlatform目录复制到服务器，就可以了。

13、也可以不用外部tomcat，而在项目中使用tomcat-maven-plugin 运行项目。

在pom.xml中加入以下代码：

<build>

<finalName>maven.archetype.webapp</finalName>

<pluginManagement>

<plugins>

<plugin>

<groupId>org.apache.tomcat.maven</groupId>

<artifactId>tomcat7-maven-plugin</artifactId>

<version>2.0</version>

</plugin>

</plugins>

</pluginManagement>

</build>

保存后右击项目，执行 Run As -> Run Configurations



即可在tomcat-maven插件中运行该项目。打开浏览器输入地址http://localhost:8080/maven.archetype.webapp/ 访问项目首页。

注意：此时在项目属性的Project Facets 中，并没有引入Dynamic Web Module，因此不能执行Run As -> Run on Server 来引入本地tomcat运行。

## Eclipse+Maven构建springmvc项目

一.背景介绍

　　对于初学者，用maven构建项目并不是一件容易的事，springmvc并不是MVC中的主流，但是本人觉得springmvc比struts要好用，真正做到了零配置。一次使用，就放不下了。

二.准备工作

　1.Eclipse 3.7

　　2.maven

　　3.Eclipse 需要安装maven插件。url:maven - <http://download.eclipse.org/technology/m2e/releases> 。安装maven-3.0.4。并且选择本地的maven，如下图：



三.构建工程

　　1.用maven插件构建项目框架

　　maven具有强大构建功能，使用maven可以构建多种不同类型的工程。这里我们构建maven-archetype-webapp类型的项目。在Eclipse->New中选择other,找到maven Project型。如下图：



在选完路径之后，我们选择构建类型，如下图：



接下来，填写工程的Group Id，Artifact Id，如下图：



这里的Group Id就是大项目的id，Arifact Id就是该项目的Id。就像一个大项目中有许多小项目组成一样。此时，我们的项目已经成型了，样子如下图：



接下来，我们要完善项目的目录，配置。

　　2.完善项目

　　首先，完善目录，增加重要的source Folder，这个不是简单的Floder,这些文件夹是会参与编译的。增加src/main/java，src/test/resources，src/test/java目录。让目录变成标准的maven结构。如下图：



接下来，改变一些配置：

让工程的JDK用本地的jdk;

让工程的字符集为UTF-8;

改变工程的目录顺序；

　　这些都完成之后，工程目录应该是如下的样子：



　　3.将工程变成web工程

　　此时，我们的工程还不是标准的web工程，可以在eclipse中增加web工程的特性，选择工程的Properties,选Project Facets,如下图：



这里，我们选择Dynamic Web Module,版本选择2.4，这个版本比较通用。如下图：



此时，我们看到目录中多了一个WebContent目录，由于使用maven构建，web目录是src/main/webapp,所以我们删除WebContent目录。接下来，要配置web项目的发布目录，就是Deployment Assembly，如图：



test目录不要发布，WebContent目录没有了，所以这三项都删掉。并且增加src/main/webapp目录，和Maven Dependenices,完成之后如下图：



于是，我们的工程就完全是一个web工程了。

　　4.赋予工程的springmvc特性

　　配置web.xml，使其具有springmvc特性，主要配置两处，一个是ContextLoaderListener，一个是DispatcherServlet。代码如下：

http://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

1 <?xml version="1.0" encoding="UTF-8"?>

2 <web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"

3 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

4 xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee

5 http://java.sun.com/xml/ns/j2ee/web-app\_2\_4.xsd">

6

7 <listener>

8 <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>

9 </listener>

10

11 <servlet>

12 <servlet-name>exam</servlet-name>

13 <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

14 </servlet>

15

16 <servlet-mapping>

17 <servlet-name>exam</servlet-name>

18 <url-pattern>/</url-pattern>

19 </servlet-mapping>

20

21 <welcome-file-list>

22 <welcome-file>index.jsp</welcome-file>

23 </welcome-file-list>

24 </web-app>

[复制代码](javascript:void(0);)

　　配置ContextLoaderListener表示，该工程要以spring的方式启动。启动时会默认在/WEB-INF目录下查找applicationContext.xml作为spring容器的配置文件，这里可以初始化一些bean，如DataSource。我们这里什么也不做。代码如下：

http://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

1 <?xml version="1.0" encoding="UTF-8"?>

2 <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-beans.dtd">

3 <beans>

4 </beans>

　　配置DispatcherServlet表示，该工程将采用springmvc的方式。启动时也会默认在/WEB-INF目录下查找XXX-servlet.xml作为配置文件，XXX就是DispatcherServlet的名字，该文件中将配置两项重要的mvc特性：

HandlerMapping,负责为DispatcherServlet这个前端控制器的请求查找Controller；

ViewResolver,负责为DispatcherServlet查找ModelAndView的视图解析器。

代码如下：

http://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

1 <?xml version="1.0" encoding="UTF-8"?>

2 <!-- Bean头部 -->

3 <beans xmlns="http://www.springframework.org/schema/beans"

4 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:p="http://www.springframework.org/schema/p"

5 xmlns:mvc="http://www.springframework.org/schema/mvc" xmlns:context="http://www.springframework.org/schema/context"

6 xmlns:util="http://www.springframework.org/schema/util"

7 xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-3.0.xsd

8 http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-3.0.xsd

9 http://www.springframework.org/schema/mvc http://www.springframework.org/schema/mvc/spring-mvc-3.0.xsd

10 http://www.springframework.org/schema/util http://www.springframework.org/schema/util/spring-util-3.0.xsd">

11

12 <!-- 激活@Controller模式 -->

13 <mvc:annotation-driven />

14 <!-- 对包中的所有类进行扫描，以完成Bean创建和自动依赖注入的功能 需要更改 -->

15 <context:component-scan base-package="cc.monggo.web.controller" />

16

17 <bean class="org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter" />

18

19 <bean id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver">

20 <property name="prefix">

21 <value>/WEB-INF/jsp/</value>

22 </property>

23 <property name="suffix">

24 <value>.jsp</value>

25 </property>

26 </bean>

27 </beans>

[复制代码](javascript:void(0);)

　　5.让maven自动配置jar包

　　在用maven生成框架时，就生成了pop.xml，这就是maven的配置文件。我们要引入spring-web,servlet等特性的包。代码如下：

http://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

1 <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

2 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4\_0\_0.xsd">

3 <modelVersion>4.0.0</modelVersion>

4 <groupId>exam</groupId>

5 <artifactId>exam\_3</artifactId>

6 <packaging>war</packaging>

7 <version>0.0.1-SNAPSHOT</version>

8 <name>exam\_3 Maven Webapp</name>

9 <url>http://maven.apache.org</url>

10 <dependencies>

11 <dependency>

12 <groupId>junit</groupId>

13 <artifactId>junit</artifactId>

14 <version>3.8.1</version>

15 <scope>test</scope>

16 </dependency>

17

18 <dependency>

19 <groupId>org.springframework</groupId>

20 <artifactId>spring-web</artifactId>

21 <version>3.0.5.RELEASE</version>

22 </dependency>

23

24 <dependency>

25 <groupId>org.springframework</groupId>

26 <artifactId>spring-webmvc</artifactId>

27 <version>3.0.5.RELEASE</version>

28 </dependency>

29

30 <dependency>

31 <groupId>org.apache.geronimo.specs</groupId>

32 <artifactId>geronimo-servlet\_2.5\_spec</artifactId>

33 <version>1.2</version>

34 </dependency>

35

36 </dependencies>

37 <build>

38 <finalName>exam\_3</finalName>

39 </build>

40 </project>

[复制代码](javascript:void(0);)

　　maven就是这么简单，一旦保存，maven就会自动下载pop.xml的jar包。此时可以看到目录中Maven Dependencies下生成了jar包。

　　更多的jar包可以在maven中心库下载：[http://mvnrepository.com](http://mvnrepository.com/)。

　　6.做个测试

　　说了一大堆，只有运行起来才有意思，下面写个简单的测试。先写Controller。编写两个类，LoginControler.java，LoginForm.java。代码如下：

http://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

1 package cc.monggo.web.controller;

2

3

4 import javax.servlet.http.HttpServletRequest;

5 import javax.servlet.http.HttpServletResponse;

6

7 import org.springframework.stereotype.Controller;

8 import org.springframework.web.bind.annotation.RequestMapping;

9 import org.springframework.web.servlet.ModelAndView;

10

11 import cc.monggo.domain.LoginForm;

12

13 @Controller

14 public class LoginController {

15 @RequestMapping(value="login")

16 public ModelAndView login(HttpServletRequest request,HttpServletResponse response,LoginForm command ){

17 String username = command.getUsername();

18 ModelAndView mv = new ModelAndView("/index/index","command","LOGIN SUCCESS, " + username);

19 return mv;

20 }

21 }

[复制代码](javascript:void(0);)

http://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

1 package cc.monggo.domain;

2

3

4 public class LoginForm {

5 private String username;

6 private String password;

7 public String getUsername() {

8 return username;

9 }

10 public void setUsername(String username) {

11 this.username = username;

12 }

13 public String getPassword() {

14 return password;

15 }

16 public void setPassword(String password) {

17 this.password = password;

18 }

19 }

[复制代码](javascript:void(0);)

　　再增加一些jsp,首页的index.jsp,主要是做跳转，代码如下：

http://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

1 <%

2 request.getRequestDispatcher("/WEB-INF/jsp/login/login.jsp").forward(request,response);

3 %>

　  还有两个jsp，做些简单的功能，一个表单login.jsp,一个表单提交的返回index.jsp，代码如下：

http://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

1 <%@ page language="java" contentType="text/html; charset=utf-8"

2 pageEncoding="utf-8"%>

3 <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">

4 <html>

5 <head>

6 <meta http-equiv="Content-Type" content="text/html; charset=utf-8">

7 <title>Insert title here</title>

8 </head>

9 <body>

10 <div>

11 <form action="login" methed="get">

12 <input type="text" name="username">

13 <input type="submit" value="SUBMIT">

14 </form>

15 </div>

16 </body>

17 </html>

[复制代码](javascript:void(0);)

http://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

1 <%@ page language="java" contentType="text/html; charset=utf-8"

2 pageEncoding="utf-8"%>

3 <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">

4 <html>

5 <head>

6 <meta http-equiv="Content-Type" content="text/html; charset=utf-8">

7 <title>Insert title here</title>

8 </head>

9 <body>

10 ${command}

11 </body>

12 </html>

[复制代码](javascript:void(0);)

　　整个项目的目录结构如下：



　　在tomcat中运行，即可。不要使用Tomcat插件，可能有问题。就用普通的方式。运行效果如下图：



## Maven项目打包缺少XML文件

解决办法，在pom.xml中把xml文件包含进去。

<build>

<resources>

<resource>

<directory>src/main/java</directory>

<includes>

<include>com/sun/mapping/ChargeMapper.xml</include>

<include>com/sun/mapping/ConsumeMapper.xml</include>

<include>com/sun/mapping/RefundMapper.xml</include>

<include>com/sun/mapping/UserMapper.xml</include>

</includes>

<filtering>true</filtering>

</resource>

</resources>

</build>

## pom.xml更新下载jar包失败





1.问题描述

在通过pom.xml中的依赖坐标,从私服下载依赖包时,往往会遇到如下的类似问题:

The container 'Maven Dependencies' references non existing library '${groupid}/${artifactid}-${version}.jar'

e.g.

|  |  |
| --- | --- |
| 1 | The container 'Maven Dependencies' references non existing library 'E:\Maven\repo\org\apache\hadoop\hadoop-yarn-api\2.0.0-cdh4.5.0\hadoop-yarn-api-2.0.0-cdh4.5.0.jar' |

2.解决方案

上面问题往往是在下载依赖过程中网络出现问题导致的.

此时我们本机已经开始下载依赖代码,但是下载失败,本机仓库中会在${MAVEN\_repo}/${groupid}/${artifactid}/${version} 路径下面生成 \*.lastUpdated 的文件.

\*.lastUpdated实在可恶,下次再下载时由于它的存在Maven不会再去私服中下载

2.1 解决方法:(针对PC与Maven私服之间网络传输问题)

1. 打开本地仓库所在目录, 通过win文件夹的搜索功能,查找 \*.lastUpdated ,然后将找到的文件全部删除
2. 重新 Maven Update Project,
   1. 若在eclipse中操作时勾选上 Force Update of Snapshots / Releases
   2. 若在命令行操作时增加 -U 参数

一般情况下上面方法即可解决问题. 若问题依然存在,请确认下主机与Maven私服之间网络是否畅通.可以通过在浏览器中打开私服地址确认.

若本机与私服之前网络畅通,那这个问题可能是由于私服与中央仓库(或者其他第三方仓库)之间下载依赖时网络故障导致的.

2.2 解决方法:(针对私服与中央仓库或第三方仓库之间网络传输问题)

1. 使用管理员登陆私服页面(Sonatype Nexus)
2. 在仓库的浏览页面找到下载失败的包, 右键菜单 "Update Index", 右键菜单 "Expire Cache"

[](http://static.oschina.net/uploads/space/2013/1206/162311_NoW2_1189023.png)

1. 按照PC与私服之间网络问题的解决步骤处理一遍

## 配置Maven Web 项目启动时跳过单元测试

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<version>2.5.1</version>

<configuration>

<source>1.7</source>

<target>1.7</target>

</configuration>

</plugin>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-surefire-plugin</artifactId>

<version>2.12.4</version>

<configuration>

<skip>true</skip>

</configuration>

</plugin>

</plugins>

</build>

# repository

# Hibernate

## Eclipse 使用Hibernate Tools插件生成POJO

很多时候我们已经设计好了数据库，需要使用hibernate来做数据持久化，因此需要根据数据库中的表结构生成相应的POJO.

本例使用hibernatetools来自动创建pojo。

测试环境：eclipse3.7（indigo）+hibernate4.1+hibernatetools3.3+oracle10g

 生成POJO只需要导入数据库驱动包，不需要导入其他Hibernate包。

**1.下载hibernatetools插件**

hibernatetools可从官网下载，但是需要注意下载和自己使用的eclipse版本对应的版本包，然后安装到eclipse中取，link安装方式参见：<http://maimode.iteye.com/blog/1164524>

下载地址:

1、首先，要去Hibernate的网站下载这个工具包。一个大约14M的ZIP压缩文件。  
<http://jaist.dl.sourceforge.net/sourceforge/jboss/HibernateTools-3.2.4.Beta1-R200810311334.zip>  
2、解压缩下载好的文件，释放出features和plugins这两个文件夹和其中的内容。完成后将这两个文件夹复制到Eclipse的目录下（这两个文件夹在Eclipse中是默认存在的，直接复制就行了）。  
3、到这步为止HibernateTools的安装就算结束了

**2.创建hibernate配置文件及console配置文件**

新建java项目，建立包test,test.hibernate,test.model。

选中项目，右键-》new-》other-》hibernate-》hibernate configuration file：



然后 next，配置console文件：



**3.创建reverse文件**

选中项目，右键-》new-》other-》hibernate-》hibernate reverse engineering file



下一步，配置要生成POJO对应的表：



**4.反向工程创建pojo**

在hibernate code generation configuration菜单中配置







执行run后便生成了相应的pojo文件：



一般自动生成的mapping文件（XXX.hbm.xml）都需要修改后才能使用。

## 日期类型

某字段数据库中数据类型为date时，Hibernate的实体类中，该字段可以是Date类型，也可以是String类型。String类型时，默认是yyyy-MM-dd HH:mm:ss格式。

## hibernate注解

### [@Enumerated](http://www.cnblogs.com/minideas/archive/2011/11/04/2235262.html)

@Enumerated(value=EnumType.ORDINAL)采用枚举类型的序号值与数据库进行交互，   
此时数据库的数据类型需要是数值类型，例如在实际操作中   
CatTest ct = new CatTest();   
ct.setColor(Color.BLUE);   
当我们将对象gt保存到数据库中的时候，数据库中存储的数值是BLUE在Color枚举   
定义中的序号1(序号从零开始)；

@Enumerated(value=EnumType.STRING)采用枚举类型与数据库进行交互，   
此时数据库的数据类型需要是NVACHAR2等字符串类型，例如在实际操作中   
CatTest ct = new CatTest();   
ct.setColor(Color.BLUE);   
数据库中存储的数值是BLUE字符串。

## Hibernate异常收集

org.hibernate.LazyInitializationException: could not initialize proxy - no Session

deptResult.setHospitalId(dept.getHospital().getHospitalId());

病症：这是一个lazy使用后的Exception，使用迟时加载，在session（hibernate里的session），关闭后使用该对象的未加载变量，也就是说session已经关闭，没有保存到内存中，然后你使用了，导致该异常。

在ManyToOne注解中，把fetch= FetchType.LAZY 改为fetch=FetchType.EAGER

@ManyToOne(cascade = CascadeType.REFRESH, optional = false,fetch=FetchType. EAGER)

@JoinColumn(name = "hospital\_id",insertable = false, updatable = false)

private HospitalEntity hospital; //医院

## Entity类配置示例

@Entity

@Table(name = "health\_depts")

public class DeptEntity implements Serializable {

private static final long serialVersionUID = -1697061357355856964L;

@Id

@Column(name = "dept\_id")

private long deptId; //科室ID

@Column(name = "hospital\_id")

private long hospitalId; //医院ID

@ManyToOne(cascade = CascadeType.REFRESH, optional = false,fetch=FetchType.LAZY)

@JoinColumn(name = "hospital\_id",insertable = false, updatable = false)

private HospitalEntity hospital; //医院

@ManyToOne(cascade = CascadeType.REFRESH, optional = false)

@JoinColumn(name = "hospital\_id",insertable = false, updatable = false)

private HospitalConfigEntity hospitalConfig; //医院配置类

……

}

一个数据表字段hospital\_id，科室设置多个变量hospitalId、hospital、hospitalConfig。

定义了一个hospitalId，同时还定义一个Entity类hospital，并将这个Entity类设置为懒加载，这样是为了提高系统性能。当只需要用到hospitalId时，不用加载HospitalEntity。

# spring

## 通过ClassPathXmlApplicationContext读取配置文件

ApplicationContext context =

**new** ClassPathXmlApplicationContext(**new** String[]

{ "file:D:/eclipse-jee-juno/mavenspace/maven.springmvc/src/main/webapp/WEB-INF/applicationContext.xml" });

SpringPublisher publisher = (SpringPublisher) context

.getBean("springPublisher");

## Windows下ActiveMQ下载、安装部署

1、下载：http://activemq.apache.org/download.html 最新Windows版本

2、安装

(1) 首先配置JAVA环境变量

JAVA\_HOME=D:\Program Files\Java\jdk1.5.0

CLASSPAHT=.;D:\Program Files\Java\jdk1.5.0\lib\tools.jar;D:\Program Files\Java\jdk1.5.0\lib\td.jar;D:\Program Files\Java\jdk1.5.0\lib

PATH=D:\Program Files\Java\jdk1.5.0\bin;

(2)直接解压至任意目录(例如：D:\apache-activemq-5.3.0)

3、启动ActiveMQ服务器：直接运行\bin\win32\activemq.bat

当运行成功后

4、打开ActiveMQ消息管理后台系统 http://localhost:8161/admin/

## 消息队列：Spring JMS

### [SPRING与JMS结合的实例](http://www.cnblogs.com/phoebus0501/archive/2011/02/28/1966896.html)

|  |
| --- |
| package jms.activemq.myexample.spring;  import java.util.Date;  import javax.jms.JMSException;  import javax.jms.Message;  import javax.jms.Session;  import javax.jms.TextMessage;  import org.springframework.jms.core.MessageCreator;    public class MyMessageCreator implements MessageCreator {        /\*\*       \* 消息序号       \*/      private int msgNo;        public MyMessageCreator(int no) {          this.msgNo = no;      }        @Override      public Message createMessage(Session session) throws JMSException {          TextMessage textMsg = session.createTextMessage();          textMsg.setText(new Date() + "第" + this.msgNo + "条消息发出");            return textMsg;      }    } |
| package jms.activemq.myexample.spring;    import javax.jms.\*;    /\*\*   \* Text消息监听   \*   \* @author xiaochuanyu   \*   \*/  public class TextListener implements MessageListener {        /\*\*       \* Casts the message to a TextMessage and displays its text.       \*       \* @param message       \*            the incoming message       \*/      public void onMessage(Message message) {          TextMessage msg = null;            try {              if (message instanceof TextMessage) {                  msg = (TextMessage) message;                  System.out.println("Reading message: " + msg.getText());              } else {                  System.out.println("Message of wrong type: "                          + message.getClass().getName());              }          } catch (JMSException e) {              System.out.println("JMSException in onMessage(): " + e.toString());          } catch (Throwable t) {              System.out.println("Exception in onMessage():" + t.getMessage());          }      }  } |

|  |
| --- |
| package jms.activemq.myexample.spring;    import org.springframework.context.ApplicationContext;  import org.springframework.context.support.ClassPathXmlApplicationContext;    public class SpringJmsTestMain {        public static void main(String[] args) throws InterruptedException {            ApplicationContext context = new ClassPathXmlApplicationContext(                  new String[] { "SpringJms/SpringJms.xml" });            SpringPublisher publisher = (SpringPublisher) context                  .getBean("springPublisher");          publisher.start();      }  } |
| package jms.activemq.myexample.spring;    import javax.jms.Destination;    import org.springframework.jms.core.JmsTemplate;    public class SpringPublisher {      /\*\*       \* Jms模板       \*/      private JmsTemplate template;        /\*\*       \* Topic       \*/      private Destination topic;        public JmsTemplate getTemplate() {          return template;      }        public void setTemplate(JmsTemplate template) {          this.template = template;      }        public Destination getTopic() {          return topic;      }        public void setTopic(Destination topic) {          this.topic = topic;      }        /\*\*       \* Start       \*       \* @throws InterruptedException       \*/      public void start() throws InterruptedException {            int messageCount = 10;            while ((--messageCount) > 0) {              sendMessage(messageCount);              Thread.sleep(1000);          }      }        /\*\*       \* 消息发送       \*/      protected void sendMessage(int msgNO) {            this.template.send(this.topic, new MyMessageCreator(msgNO));      }  } |

[复制代码](javascript:void(0);)

<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">  
  
  
  
 <!-- jms 连接工厂 -->  
 <bean id="jmsFactory" class="org.apache.activemq.ActiveMQConnectionFactory">  
 <property name="brokerURL">  
 <value>tcp://localhost:61616</value>  
 </property>  
 </bean>  
  
  
 <!-- jms 连接池 -->  
   
 <!--   
 <bean id="pooledJmsFactory" class="org.apache.activemq.pool.PooledConnectionFactory">  
 <property name="connectionFactory">  
 <ref local="jmsFactory" />  
 </property>  
 </bean>  
 -->  
  
 <!-- jms Topic -->  
 <bean id="myTopic" class="org.apache.activemq.command.ActiveMQTopic"  
 autowire="constructor">  
 <constructor-arg value="STOCKS.JAVA" />  
 </bean>  
  
  
 <!-- 消息监听器 -->  
 <bean id="myTextListener" class="jms.activemq.myexample.spring.TextListener">  
 </bean>  
  
  
 <!-- jms Consumer -->  
 <bean id="javaConsumer"  
 class="org.springframework.jms.listener.DefaultMessageListenerContainer">  
 <property name="connectionFactory" ref="jmsFactory" />  
 <property name="destination" ref="myTopic" />  
 <property name="messageListener" ref="myTextListener" />  
 </bean>  
  
 <!-- jms 模板 -->  
 <bean id="jmsTemplate" class="org.springframework.jms.core.JmsTemplate">  
 <property name="connectionFactory">  
 <ref local="jmsFactory" />  
 </property>  
 </bean>  
  
  
 <!-- 消息发布器 -->  
 <bean id="springPublisher" class="jms.activemq.myexample.spring.SpringPublisher">  
 <property name="template">  
 <ref local="jmsTemplate" />  
 </property>  
 <property name="topic">  
 <ref local="myTopic" />  
 </property>  
 </bean>  
</beans>

### [Spring+JMS例子](http://www.blogjava.net/wangbing/archive/2010/03/31/317093.html)

第一: 在applicationContext.xml

<?xml version="1.0" encoding="UTF-8"?>  
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-beans.dtd">

<beans>    
 <bean id="connectionFactory" class="org.apache.activemq.ActiveMQConnectionFactory">  
   <property name="brokerURL">  
     <value>tcp://192.168.3.48:61616</value>  
   </property>  
    
 </bean>  
   
 <bean id="jmsTemplate" class="org.springframework.jms.core.JmsTemplate">  
    <property name="connectionFactory">  
      <ref bean="connectionFactory"/>  
    </property>  
 </bean>  
   
 <bean id="destination" class="org.apache.activemq.command.ActiveMQQueue">  
   <constructor-arg index="0">  
     <value>HelloWorldQueue</value>  
   </constructor-arg>  
 </bean>  
</beans>

2.写发送方

package ch13.JMS;

import javax.jms.Destination;  
import javax.jms.JMSException;  
import javax.jms.Message;  
import javax.jms.Session;

import org.springframework.context.ApplicationContext;  
import org.springframework.context.support.ClassPathXmlApplicationContext;  
import org.springframework.jms.core.JmsTemplate;  
import org.springframework.jms.core.MessageCreator;

public class HelloWorldSender  
{  
 public static void main(String args[]) throws Exception  
 {  
  ApplicationContext context = new ClassPathXmlApplicationContext(new String[] { "ch13/JMS/applicationContext.xml" });  
  JmsTemplate jmsTemplate = (JmsTemplate) context.getBean("jmsTemplate");  
  Destination destination = (Destination) context.getBean("destination");  
  jmsTemplate.send  
  (  
   destination, new MessageCreator()  
   {  
    public Message createMessage(Session session) throws JMSException  
    {  
     return session.createTextMessage("大家好这个是测试！");  
    }  
   }  
  );  
 }

}  
3.写接收方

package ch13.JMS;

import javax.jms.Destination;  
import javax.jms.TextMessage;

import org.springframework.context.ApplicationContext;  
import org.springframework.context.support.ClassPathXmlApplicationContext;  
import org.springframework.jms.core.JmsTemplate;

public class HelloWorldReciver  
{

 public static void main(String args[]) throws Exception  
 {  
  ApplicationContext context = new ClassPathXmlApplicationContext(new String[] { "ch13/JMS/applicationContext.xml" });  
  JmsTemplate jmsTemplate = (JmsTemplate) context.getBean("jmsTemplate");  
  Destination destination = (Destination) context.getBean("destination");  
  System.out.println("will wait:" + jmsTemplate.getReceiveTimeout()+ " seconds for message");  
  TextMessage msg = (TextMessage) jmsTemplate.receive(destination);  
  System.out.println("reviced msg is:" + msg.getText());  
 }

}

4.启动activemq中bin 下的activemq.bat

5.先运行 HelloWorldSender

6.再运行 HelloWorld

7.结果:

will wait:-1 seconds for message  
reviced msg is:大家好 这个是测试！

## Spring IOC

1、获取Bean的方法

ApplicationContext ctx = WebApplicationContextUtils.*getWebApplicationContext*(

request.getSession().getServletContext() );

TestService testService = (TestService)ctx.getBean("testService");

testService.sayHello();

## 区分Spring配置文件和Servlet配置文件

在web.xml中，经常需要配置Spring和Servlet，注意区分这两者。

<!-- 指定spring配置文件 -->

<listener>

<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>

</listener>

<!-- <context-param>是可选项，如果没有的话就加载applicationContext.xml，也可以用它指定其他配置文件 -->

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>/WEB-INF/conf/\*.xml</param-value>

</context-param>

<!-- 我们需要配置Servlet的配置文件，默认DispatcherServlet会加载

WEB-INF/[DispatcherServlet的Servlet名字]-servlet.xml配置文件 -->

<servlet>

<servlet-name>chapter2</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

<init-param>

<param-name>contextConfigLocation</param-name>

<param-value>/WEB-INF/conf/springmvc.xml</param-value> <!-- 指定servlet配置文件 -->

</init-param>

<load-on-startup>1</load-on-startup> <!-- 表示启动容器时初始化该Servlet -->

</servlet>

<servlet-mapping>

<servlet-name>chapter2</servlet-name>

<url-pattern>/</url-pattern> <!-- 表示哪些请求交给Spring Web MVC处理 -->

</servlet-mapping>

## [Spring MVC集成Tiles使用方法](http://teamojiao.iteye.com/blog/439796)

通过tiles配置文件来寻找页面。

首先，我们定义一个总体的tiles视图 /tiles/mainTemplate.jsp  
  
首先使用:<tiles:getAsString name="title"/>打印tiles配置文件中定义的字符串属性title  
然后是分别加载header,content,footer三个jsp页面，共同组成mainTemplate.jsp

 mainTemplate.jsp页面如下：

**Html代码  [收藏代码](javascript:void())**

1. **<**%@taglib uri="http://jakarta.apache.org/struts/tags-tiles" prefix="tiles"%**>**
2. **<html>**
3. **<head>**
5. **<title><tiles:getAsString** name="title"**/></title>**
6. **</head>**
7. **<body>**
8. the title is:**<tiles:getAsString** name="title"**/><br>**
9. **<tiles:insert** name="header"**/><br>**
10. **<tiles:insert** name="content"**/><br>**
11. **<tiles:insert** name="footer"**/><br>**
12. **</body>**
13. **</html>**

header.content.footer页面如下：

**Html代码  [收藏代码](javascript:void())**

1. **<html>**
2. **<head>**
3. **</head>**
4. **<body>**
5. this is from header
6. **</body>**
7. **</html>**

10. **<html>**
11. **<head>**
12. **</head>**
13. **<body>**
14. this is from content
15. **</body>**
16. **</html>**

19. **<html>**
20. **<head>**
21. **</head>**
22. **<body>**
23. this is from footer**</body>**
24. **</html>**

Spring MVC 控制类：

**Java代码  [收藏代码](javascript:void())**

1. **package** Action;
3. **import** java.util.ArrayList;
4. **import** java.util.List;
6. **import** javax.servlet.http.HttpServletRequest;
7. **import** javax.servlet.http.HttpServletResponse;
9. **import** model.Student;
11. **import** org.springframework.validation.BindException;
12. **import** org.springframework.web.servlet.ModelAndView;
13. **import** org.springframework.web.servlet.mvc.AbstractCommandController;
15. **public** **class** TilesHomeController **extends** AbstractCommandController {

18. **protected** ModelAndView handle(HttpServletRequest arg0, HttpServletResponse arg1, Object arg2, BindException arg3) **throws** Exception {
19. **return** **null**;
20. }
22. **protected** ModelAndView handleRequestInternal(HttpServletRequest arg0, HttpServletResponse arg1) **throws** Exception {

25. **return** **new** ModelAndView("courseDetail","","");
26. }


30. }

 当我们运行上面这个控制器时候，系统会自动去寻找courseDetail.jsp这个文件，但我们还没有定义这个文件，所以，我们要这样写tiles-def.xml

**Xml代码  [收藏代码](javascript:void())**

1. <!DOCTYPE tiles-definitions PUBLIC
2. "-//Apache Software Foundation//DTD Tiles Configuration//EN"
3. "http://jakarta.apache.org/struts/dtds/tiles-config\_1\_1.dtd"**>**

6. **<tiles-definitions>**
7. **<definition** name="template" page="/tiles/mainTemplate.jsp"**>**
8. **<put** name="title" value="default title"**/>**
9. **<put** name="header" value="="/tiles/header.jsp"**/>**
10. **<put** name="content" value="/tiles/content.jsp"**/>**
11. **<put** name="footer" value="/tiles/footer.jsp"**/>**
12. **</definition>**
14. <!--以courseDetail命名的配置，覆盖了模板中的title和content对应的页面-->
15. **<definition** name="courseDetail" extends="template"**>**
16. **<put** name="title" value="courseDetail title"**/>**
17. **<put** name="content" value="/tiles/courseContent.jsp"**/>**
18. **</definition>**
20. **</tiles-definitions>**

/title/courseContent.jsp

**Html代码  [收藏代码](javascript:void())**

1. **<**%@taglib uri="http://jakarta.apache.org/struts/tags-tiles" prefix="tiles" %**>**
2. **<html>**
3. **<head></head>**
4. **<body>**
5. this is courseContent
6. **</body>**
7. **</html>**

spring配置文件：

**Xml代码  [收藏代码](javascript:void())**

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-beans.dtd" **>**
3. **<beans>**
5. **<bean** id="simpleUrlMapping" class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping"**>**
6. **<property** name="mappings"**>**
7. **<props>**
8. **<prop** key="/tiles.mvc"**>**TilesController**</prop>**
9. **</props>**
10. **</property>**
11. **</bean>**

14. **<bean** id="TilesController" class="Action.TilesHomeController"**>**
15. **</bean>**
17. <!-- tiles页面解析器-->
18. **<bean** id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver"**>**
19. **<property** name="viewClass"**>**
20. **<value>**org.springframework.web.servlet.view.tiles.TilesView**</value>**
21. **</property>**
22. **</bean>**

25. <!-- tiles配置器-->
26. **<bean** id="tilesConfigurer" class="org.springframework.web.servlet.view.tiles.TilesConfigurer"**>**
27. **<property** name="definitions"**>**
28. **<list>**
29. **<value>**/WEB-INF/train-def.xml**</value>**
30. **</list>**
31. **</property>**
32. **</bean>**
33. **</beans>**

运行/title.mvc （mvc是我自定义的扩展名）我们可以看到如下结果：

courseDetail title  
this is from header  
this is courseDetail  
this is from footer

如果我们需要在每一个页面的header部分显示一些我们从数据库里读取的数值，我们可以在不同的控制器中加入访问数据库的代码，但这样显得很啰唆，因为header部门在每个页面都是公用的，也就是说不同的页面只有content部分不同，这样，我们就会有很多冗余代码，为了解决这个问题，我们可以使用spring为我们提供的ComponentControllerSupport

**Java代码  [收藏代码](javascript:void())**

1. **package** Action;
3. **import** javax.servlet.http.HttpServletRequest;
4. **import** javax.servlet.http.HttpServletResponse;
6. **import** model.Student;
8. **import** org.apache.struts.tiles.ComponentContext;
9. **import** org.springframework.web.servlet.view.tiles.ComponentControllerSupport;
11. **public** **class** HeaderTilesController **extends** ComponentControllerSupport {

14. **protected** **void** doPerform(ComponentContext context, HttpServletRequest request,
15. HttpServletResponse response) **throws** Exception {
16. Student stu=**new** Student();
17. stu.setName("gaoxiang");
18. stu.setSex("male");
19. //tiles context中存放简单数据
20. context.putAttribute("name", stu.getName());
21. context.putAttribute("sex", stu.getSex());
22. //tiles context中存放对象
23. context.putAttribute("student", stu);
24. //request中存放数据
25. request.setAttribute("requestname", stu.getName()+"-request");
26. request.setAttribute("requestsex", stu.getSex()+"-request");
27. }
29. }

改写train-def.xml

**Xml代码  [收藏代码](javascript:void())**

1. <!DOCTYPE tiles-definitions PUBLIC
2. "-//Apache Software Foundation//DTD Tiles Configuration//EN"
3. "http://jakarta.apache.org/struts/dtds/tiles-config\_1\_1.dtd"**>**

6. **<tiles-definitions>**
7. **<definition** name="template" page="/tiles/mainTemplate.jsp"**>**
8. **<put** name="title" value="default title"**/>**
9. **<put** name="header" value=".header"**/>**
10. **<put** name="content" value="/tiles/content.jsp"**/>**
11. **<put** name="footer" value="/tiles/footer.jsp"**/>**
12. **</definition>**
14. **<definition** name="courseDetail" extends="template"**>**
15. **<put** name="title" value="courseDetail title"**/>**
16. **<put** name="content" value="/tiles/courseContent.jsp"**/>**
17. **</definition>**
19. **<definition** name=".header" page="/tiles/header.jsp" controllerClass="Action.HeaderTilesController"**>**
20. **</definition>**
21. **</tiles-definitions>**

改header.jsp

**Html代码  [收藏代码](javascript:void())**

1. **<**%@ taglib uri="http://jakarta.apache.org/struts/tags-tiles" prefix="tiles" %**>**

## [spring 3.0.5的<mvc:resources> 标签的使用](http://ericxiong.iteye.com/blog/1009030)

spring mvc 的<mvc;resources mapping="\*\*\*" location="\*\*\*">标签是在spring3.0.4出现的，主要是用来进行静态资源的访问。在spring3.0.4出来的时候spring还没有更新其schema所以在配置文件中有可能找不到<mvc:resources >标签，这个问题在spring3.0.5中已经解决，而且网上也有很多其他的解决方案，我在这里就不记录了。   
  
首先使用spring mvc需要配置其使用的servlet.在web.xml中:

**Java代码  [收藏代码](javascript:void())**

1. <servlet>
2. <servlet-name>springMVC</servlet-name>
3. <servlet-**class**>org.springframework.web.servlet.DispatcherServlet</servlet-**class**>
4. <load-on-startup>1</load-on-startup>
5. </servlet>
7. <servlet-mapping>
8. <servlet-name>springMVC</servlet-name>
9. <url-pattern>/</url-pattern>
10. </servlet-mapping>

这里给 servlet-name定义的名称是springMVC,这样的话会在web-inf下spring会自动扫描一个XML文件名叫springMVC-servlet.xml文件，这里都是spring自动扫描的，如果你没有提供，将会报一个文件查找不到的异常。看了下org.springframework.web.servlet.DispatcherServlet加载这个文件的过程，貌似这个文件存放的地址也是可以进行设置的，具体怎么搞我还没有研究。   
  
由于spring mvc拦截了所有请求，所以当你设置

**引用**

<servlet-mapping>   
<servlet-name>springMVC</servlet-name>   
<url-pattern>/</url-pattern>   
</servlet-mapping>

的时候会影响到静态资源文件的获取，这样就需要有这个标签来帮你分类完成获取静态资源的责任。   
  
springMVC-servlet.xml文件

**Java代码  [收藏代码](javascript:void())**

1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xmlns:mvc="http://www.springframework.org/schema/mvc"
5. xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
6. http://www.springframework.org/schema/mvc http://www.springframework.org/schema/mvc/spring-mvc-3.0.xsd">
8. <mvc:resources mapping="/javascript/\*\*" location="/static\_resources/javascript/"/>
9. <mvc:resources mapping="/styles/\*\*" location="/static\_resources/css/"/>
10. <mvc:resources mapping="/images/\*\*" location="/static\_resources/images/"/>
11. <mvc:**default**-servlet-handler />

14. <bean **class**="org.springframework.web.servlet.view.InternalResourceViewResolver">
15. <property name="prefix" value="/WEB-INF/views/"/>
16. <property name="suffix" value=".jsp"/>
17. </bean>
19. </beans>

这里可以看到我所有的页面引用到/styles/\*\*的资源都从/static\_resources/css里面进行查找。   
  
页面的一段静态资源访问的代码。

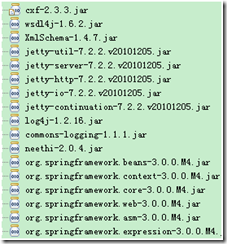
**Java代码  [收藏代码](javascript:void())**

1. <%@ taglib prefix="c" uri="http://java.sun.com/jstl/core"%>
2. <HTML>
3. <HEAD>
4. <TITLE> ABCDEFG </TITLE>
5. <link type="text/css" rel="stylesheet" href="<c:url value='/styles/siteboard.css'/>">
6. ...
7. ...
8. ...

可能这个标签的真谛就是为了引用资源的访问不会类似CONTROLLER一样被拦截，区分出关注的资源的访问，一般我们在springMVC里面的拦截都会配置为"/"，拦截所有的。

## [CXF WebService整合Spring](http://www.cnblogs.com/hoojo/archive/2011/03/30/1999563.html)

首先，CXF和spring整合需要准备如下jar包文件：

[](http://images.cnblogs.com/cnblogs_com/hoojo/201103/201103301049065256.png)

这边我是用Spring的jar包是Spring官方提供的，并没有使用CXF中的Spring的jar文件。

添加这么多文件后，首先在web.xml中添加如下配置：

<!-- 加载Spring容器配置 -->

<listener>

<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>

</listener>

<!-- 设置Spring容器加载配置文件路径 -->

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>classpath\*:applicationContext-server.xml</param-value>

</context-param>

<listener>

<listener-class>org.springframework.web.util.IntrospectorCleanupListener</listener-class>

</listener>

<servlet>

<servlet-name>CXFService</servlet-name>

<servlet-class>org.apache.cxf.transport.servlet.CXFServlet</servlet-class>

</servlet>

<servlet-mapping>

<servlet-name>CXFService</servlet-name>

<url-pattern>/\*</url-pattern>

</servlet-mapping>

然后在src目录中，新建一个applicationContext-server.xml文件，文件内容如下：

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:context="http://www.springframework.org/schema/context"

**xmlns:jaxws="http://cxf.apache.org/jaxws"**

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans >

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context-3.0.xsd

**http://cxf.apache.org/jaxws**

**http://cxf.apache.org/schemas/jaxws.xsd**"

注意上面的带下划线加粗部分，这个很重要的哦！不能写错或是遗漏了。

添加完这个文件后，还需要在这个文件中导入这么几个文件。文件内容如下：

<import resource="classpath:META-INF/cxf/cxf.xml"/>

<import resource="classpath:META-INF/cxf/cxf-extension-soap.xml"/>

<import resource="classpath:META-INF/cxf/cxf-servlet.xml"/>

下面开始写服务器端代码，首先定制服务器端的接口，代码如下：

package com.hoo.service;

import javax.jws.WebParam;

import javax.jws.WebService;

import javax.jws.soap.SOAPBinding;

import javax.jws.soap.SOAPBinding.Style;

import com.hoo.entity.User;

import com.hoo.entity.Users;

/\*\*

\* <b>function:</b>定制客户端请求WebService所需要的接口

\* @author hoojo

\* @createDate 2011-3-18 上午08:22:55

\* @file ComplexUserService.java

\* @package com.hoo.service

\* @project CXFWebService

\* @blog http://blog.csdn.net/IBM\_hoojo

\* @email hoojo\_@126.com

\* @version 1.0

\*/

@WebService

@SOAPBinding(style = Style.RPC)

public interface IComplexUserService {

public User getUserByName(@WebParam(name = "name") String name);

public void setUser(User user);

}

下面编写WebService的实现类，服务器端实现代码如下：

package com.hoo.service;

import java.util.ArrayList;

import java.util.Date;

import java.util.HashMap;

import java.util.List;

import javax.jws.WebParam;

import javax.jws.WebService;

import javax.jws.soap.SOAPBinding;

import javax.jws.soap.SOAPBinding.Style;

import com.hoo.entity.User;

import com.hoo.entity.Users;

/\*\*

\* <b>function:</b> WebService传递复杂对象，如JavaBean、Array、List、Map等

\* @author hoojo

\* @createDate 2011-3-18 上午08:22:55

\* @file ComplexUserService.java

\* @package com.hoo.service

\* @project CXFWebService

\* @blog http://blog.csdn.net/IBM\_hoojo

\* @email hoojo\_@126.com

\* @version 1.0

\*/

@WebService

@SOAPBinding(style = Style.RPC)

@SuppressWarnings("deprecation")

public class ComplexUserService implements IComplexUserService {

public User getUserByName(@WebParam(name = "name") String name) {

User user = new User();

user.setId(new Date().getSeconds());

user.setName(name);

user.setAddress("china");

user.setEmail(name + "@hoo.com");

return user;

}

public void setUser(User user) {

System.out.println("############Server setUser###########");

System.out.println("setUser:" + user);

}

}

注意的是和Spring集成，这里一定要完成接口实现，如果没有接口的话会有错误的。

下面要在applicationContext-server.xml文件中添加如下配置：

<bean id="userServiceBean" class="com.hoo.service.ComplexUserService"/>

<bean id="inMessageInterceptor" class="com.hoo.interceptor.MessageInterceptor">

<constructor-arg value="receive"/>

</bean>

<bean id="outLoggingInterceptor" class="org.apache.cxf.interceptor.LoggingOutInterceptor"/>

<!-- 注意下面的address，这里的address的名称就是访问的WebService的name -->

<jaxws:server id="userService" serviceClass="com.hoo.service.IComplexUserService" address="/Users">

<jaxws:serviceBean>

<!-- 要暴露的 bean 的引用 -->

<ref bean="userServiceBean"/>

</jaxws:serviceBean>

<jaxws:inInterceptors>

<ref bean="inMessageInterceptor"/>

</jaxws:inInterceptors>

<jaxws:outInterceptors>

<ref bean="outLoggingInterceptor"/>

</jaxws:outInterceptors>

</jaxws:server>

下面启动tomcat服务器后，在WebBrowser中请求：

<http://localhost:8080/CXFWebService/Users?wsdl>

如果你能看到wsdl的xml文件的内容，就说明你成功了，注意的是上面地址的Users就是上面xml配置中的address的名称，是一一对应的。

下面编写客户端请求的代码，代码如下：

package com.hoo.client;

import org.apache.cxf.jaxws.JaxWsProxyFactoryBean;

import com.hoo.entity.User;

import com.hoo.service.IComplexUserService;

/\*\*

\* <b>function:</b>请求Spring整合CXF的WebService客户端

\* @author hoojo

\* @createDate 2011-3-28 下午03:20:35

\* @file SpringUsersWsClient.java

\* @package com.hoo.client

\* @project CXFWebService

\* @blog http://blog.csdn.net/IBM\_hoojo

\* @email hoojo\_@126.com

\* @version 1.0

\*/

public class SpringUsersWsClient {

public static void main(String[] args) {

//调用WebService

JaxWsProxyFactoryBean factory = new JaxWsProxyFactoryBean();

factory.setServiceClass(IComplexUserService.class);

factory.setAddress("http://localhost:8080/CXFWebService/Users");

IComplexUserService service = (IComplexUserService) factory.create();

System.out.println("#############Client getUserByName##############");

User user = service.getUserByName("hoojo");

System.out.println(user);

user.setAddress("China-Guangzhou");

service.setUser(user);

}

}

运行后，可以在控制台中看到

log4j:WARN No appenders could be found for logger (org.apache.cxf.bus.spring.BusApplicationContext).

log4j:WARN Please initialize the log4j system properly.

log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.

2011-3-28 18:12:26 org.apache.cxf.service.factory.ReflectionServiceFactoryBean buildServiceFromClass

信息: Creating Service {http://service.hoo.com/}IComplexUserServiceService from class com.hoo.service.IComplexUserService

#############Client getUserByName##############

27#hoojo#hoojo@hoo.com#china

Tomcat控制台

[](http://images.cnblogs.com/cnblogs_com/hoojo/201103/201103301049197217.png)

这个server端是通过Spring整合配置的，下面我们将Client端也通过Spring配置完成整合。

首先增加applicationContext-client.xml配置文件，文件内容如下：

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:context="http://www.springframework.org/schema/context"

xmlns:jaxws="http://cxf.apache.org/jaxws"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans >

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context-3.0.xsd

http://cxf.apache.org/jaxws

http://cxf.apache.org/schemas/jaxws.xsd"

<import resource="classpath:META-INF/cxf/cxf.xml"/>

<import resource="classpath:META-INF/cxf/cxf-extension-soap.xml"/>

<import resource="classpath:META-INF/cxf/cxf-servlet.xml"/>

<jaxws:client id="userWsClient" serviceClass="com.hoo.service.IComplexUserService"

address="http://localhost:8080/CXFWebService/Users"/>

</beans>

客户端请求代码如下：

package com.hoo.client;

import org.apache.cxf.jaxws.JaxWsProxyFactoryBean;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.hoo.entity.User;

import com.hoo.service.IComplexUserService;

/\*\*

\* <b>function:</b>请求Spring整合CXF的WebService客户端

\* @author hoojo

\* @createDate 2011-3-28 下午03:20:35

\* @file SpringUsersWsClient.java

\* @package com.hoo.client

\* @project CXFWebService

\* @blog http://blog.csdn.net/IBM\_hoojo

\* @email hoojo\_@126.com

\* @version 1.0

\*/

public class SpringUsersWsClient {

public static void main(String[] args) {

ApplicationContext ctx = new ClassPathXmlApplicationContext("applicationContext-client.xml");

IComplexUserService service = ctx.getBean("userWsClient", IComplexUserService.class);

System.out.println("#############Client getUserByName##############");

User user = service.getUserByName("hoojo");

System.out.println(user);

user.setAddress("China-Guangzhou");

service.setUser(user);

}

}

运行后结果如下：

#############Client getUserByName##############

45#hoojo#hoojo@hoo.com#china

############Server setUser###########

setUser:45#hoojo#hoojo@hoo.com#China-Guangzhou

# JSP

JSP中动作与指令的区别

指令：  
 <%@include tile=""%>  
 该指令只是静态包含文件，不能为一个动态的地址，而且不带任何参数，服务器将对包含近来的文件进行编译。  
  
动作：  
 <JSP:include page="">  
 可以包含静态或动态文件，包含静态文件时服务器不对包含的文件编译，而在包含动态文件时服务器会编译包含动态文件，会自动判断包含文件是动态还是静态。  
 与include指令不同，该动作元素在JSP页面被用户请求时，才将指定文件插入服务器的响应输出中。

中文乱码

GET方式提交数据，IE乱码火狐、chrome正常

这是因为使用GET方式通过浏览器的地址栏提交数据时，IE默认是GBK编码，chrome默认是UTF-8编码，所以会出现IE乱码chrome正常 或 chrome乱码IE正常 的情况。

一个自创的办法是，在提交数据时增加一个隐藏字段：chinese=中文。

在接收端通过chinese字段的长度来判断是什么编码，是长度是4，则是UTF-8编码，如果长度是6，则是GBK编码。

**if**(request.getParameter("chinese").length()==4)

String remark = **new** String(

request.getParameter("remark").getBytes("ISO-8859-1"),"GB2312");

**if**(request.getParameter("chinese").length()==6)

String remark = **new** String(

request.getParameter("remark").getBytes("ISO-8859-1"),"UTF-8");

另一种方法，在web.xml中添加过滤器：

<filter>

<filter-name>encodingFilter</filter-name>

<filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>

<init-param>

<param-name>encoding</param-name>

<param-value>UTF-8</param-value>

</init-param>

<init-param>

<param-name>forceEncoding</param-name>

<param-value>true</param-value>

</init-param>

</filter>

<filter-mapping>

<filter-name>encodingFilter</filter-name>

<url-pattern>/\*</url-pattern>

</filter-mapping>

# JSLT

## 选择语句

1. **<c:choose>**
2. **<c:when** test="${username=='xiaoming'}"**>**
3. welcome, xiaoming
4. **</c:when>**
5. **<c:when** test="${username=='xiaohong'}"**>**
6. welcome, xiaohong
7. **</c:when>**
8. **<c:otherwise>**
9. sorry, you are not xiaoming or xiaohong
10. **</c:otherwise>**
11. **</c:choose>**

when和otherwise必须嵌套再choose中

<c:choose>、<c:when>和<c:otherwise>标签的使用必须符合以下语法规则：

<c:when>和<c:otherwise>不能单独使用，它们必须位于<c:choose>父标签中。

在<c:choose>标签中可以包含一个或多个<c:when>标签。

在<c:choose>标签中可以不包含<c:otherwise>标签。

在<c:choose>标签中如果同时包含<c:when>和<c:otherwise>标签，那么<c:otherwise>必须位于<c:when>标签之后。

## JSTL运算

加法：

<input type="hidden" name="fee" value="${timeRegDto.regFee + timeRegDto.treatFee}" >

## JSTL判断list的size()大小

<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>

<%@ taglib uri="http://java.sun.com/jsp/jstl/functions" prefix="fn"%>--------这里就需要有standard.jar这个包

必须要导的两个包。

jsp页面判断获得action传的list的是否为空或者list.size的长度，就可以用fn这个标签：

<c:if test="${list== null || fn:length(list) == 0}"></c:if>这样就可以判断了。

## jstl处理字符串

JSTL使用表达式来简化页面的代码，这对一 些标准的方法，例如bean的getter/setter方法，请求参数或者context以及session中的数据的访问非常方便，但是我们在实际应 用中经常需要在页面调用对象的某些方法，例如我需要调用字符串的length方法来获取字符串的长度时，在以往的开发过程中我们必须把对象先转为 String类，然后在调用其length方法，这样的代码繁琐而且容易出错。

因此JSTL内置了几个用于字符串操作的方法，可以直接在表达式中使用，大大的简化了代码，提供代码的可读性。在JSTL的表达是中要使用一个函 数，其格式如下

**${ns:methodName(args....)}**

在使用这些函数之前必须在JSP中引入标准函数的声明

**<%@ taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions" %>**

下面是JSTL中自带的方法列表以及其描述

|  |  |  |
| --- | --- | --- |
| **函数名** | **函数说明** | **使用举例** |
| contains | 判断字符串是否包含另外一个字符串 | <c:if test="${fn:contains(name, searchString)}"> |
| containsIgnoreCase | 判断字符串是否包含另外一个字符串(大小写无关) | <c:if test="${fn:containsIgnoreCase(name, searchString)}"> |
| endsWith | 判断字符串是否以另外字符串结束 | <c:if test="${fn:endsWith(filename, ".txt")}"> |
| escapeXml | 把一些字符转成XML表示，例如<字符应该转为&lt; | ${fn:escapeXml(param:info)} |
| indexOf | 子字符串在母字符串中出现的位置 | ${fn:indexOf(name, "-")} |
| join | 将数组中的数据联合成一个新字符串，并使用指定字符格开 | ${fn:join(array, ";")} |
| length | 获取字符串的长度，或者数组的大小 | ${fn:length(shoppingCart.products)} |
| replace | 替换字符串中指定的字符 | ${fn:replace(text, "-", "&#149;")} |
| split | 把字符串按照指定字符切分 | ${fn:split(customerNames, ";")} |
| startsWith | 判断字符串是否以某个子串开始 | <c:if test="${fn:startsWith(product.id, "100-")}"> |
| substring | 获取子串 | ${fn:substring(zip, 6, -1)} |
| substringAfter | 获取从某个字符所在位置开始的子串 | ${fn:substringAfter(zip, "-")} |
| substringBefore | 获取从开始到某个字符所在位置的子串 | ${fn:substringBefore(zip, "-")} |
| toLowerCase | 转为小写 | ${fn.toLowerCase(product.name)} |
| toUpperCase | 转为大写字符 | ${fn.UpperCase(product.name)} |
| trim | 去除字符串前后的空格 | ${fn.trim(name)} |

## JSTL判断字符串是否相等

<c:if test="${title==0}">主任医师</c:if>

或 <c:if test="${title=='0'}">主任医师</c:if>

## JavaScript与JSTL

不但在HMTL中可以嵌入JSTL，在JavaScript中也可以直接嵌入JSTL。例如：

<script type="text/javascript">

$(document).ready(function(){

alert("${hospitalId}");

<c:forEach var="regDateDto" items="${planDto.regDateDtoList}" >

alert("${regDateDto.regDate}");

</c:forEach>

});

</script>

# 定时任务

# 消息中心

# 系统安全

MD5

# CMS内容管理系统