

2015. java学习交流群

8918 1289

每天有免费的Java学习课堂

——学习Java就是这么简单

——为Java而燃烧——



# Java 语言编码规范(Java Code Conventions)

译者 晨光 (Morning)

搜集整理：华竹技术实验室 <http://sinoprise.com>

简介：

本文档讲述了 Java 语言的编码规范，较之陈世忠先生《c++ 编码规范》的浩繁详尽，此文当属短小精悍了。而其中所列之各项条款，从编码风格，到注意事项，不单只 Java，对于其他语言，也都很有借鉴意义。因为简短，所以易记，大家不妨将此作为 handbook，常备案头，逐一对验。

声明：

如需复制、传播，请附上本声明，谢谢。

原文出处：<http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html>,

译文出处：<http://morningspace.51.net/>，[moyingzz@etang.com](mailto:moyingzz@etang.com)

## 目录

### 1 介绍

- [1.1 为什么要有编码规范](#)
- [1.2 版权声明](#)

### 2 文件名

- [2.1 文件后缀](#)
- [2.2 常用文件名](#)

### 3 文件组织

- [3.1 Java源文件](#)
  - [3.1.1 开头注释](#)
  - [3.1.2 包和引入语句](#)
  - [3.1.3 类和接口声明](#)

### 4 缩进排版

- [4.1 行长度](#)
- [4.2 换行](#)

### 5 注释

- [5.1 实现注释的格式](#)
  - [5.1.1 块注释](#)
  - [5.1.2 单行注释](#)
  - [5.1.3 尾端注释](#)
  - [5.1.4 行末注释](#)

- [5.2 文档注释](#)

## [6 声明](#)

- [6.1 每行声明变量的数量](#)
- [6.2 初始化](#)
- [6.3 布局](#)
- [6.4 类和接口的声明](#)

## [7 语句](#)

- [7.1 简单语句](#)
- [7.2 复合语句](#)
- [7.3 返回语句](#)
- [7.4 if, if-else, if else-if else语句](#)
- [7.5 for语句](#)
- [7.6 while语句](#)
- [7.7 do-while语句](#)
- [7.8 switch语句](#)
- [7.9 try-catch语句](#)

## [8 空白](#)

- [8.1 空行](#)
- [8.2 空格](#)

## [9 命名规范](#)

## [10 编程惯例](#)

- [10.1 提供对实例以及类变量的访问控制](#)
- [10.2 引用类变量和类方法](#)
- [10.3 常量](#)
- [10.4 变量赋值](#)
- [10.5 其它惯例](#)
  - [10.5.1 圆括号](#)
  - [10.5.2 返回值](#)
  - [10.5.3 条件运算符"?"前的表达式"?"前的表达式](#)
  - [10.5.4 特殊注释](#)

## [11 代码范例](#)

- [11.1 Java源文件范例](#)

## **1 介绍(Introduction)**

### **1.1 为什么要有编码规范(Why Have Code Conventions)**

编码规范对于程序员而言尤为重要，有以下几个原因：

- 一个软件的生命周期中，80%的花费在于维护
- 几乎没有任何一个软件，在其整个生命周期中，均由最初的开发人员来维护
- 编码规范可以改善软件的可读性，可以让程序员尽快而彻底地理解新的代码
- 如果你将源码作为产品发布，就需要确任它是否被很好的打包并且清晰无误，一如你已构建的其它任何产品

为了执行规范，每个软件开发人员必须一致遵守编码规范。每个人。

## 1.2 版权声明(Acknowledgments)

本文档反映的是 Sun Microsystems 公司，Java 语言规范中的编码标准部分。主要贡献者包括：Peter King, Patrick Naughton, Mike DeMoney, Jonni Kanerva, Kathy Walrath 以及 Scott Hommel。

本文档现由 Scott Hommel 维护，有关评论意见请发至 [shommel@eng.sun.com](mailto:shommel@eng.sun.com)

## 2 文件名(File Names)

这部分列出了常用的文件名及其后缀。

### 2.1 文件后缀(File Suffixes)

Java 程序使用下列文件后缀：

文件类别	文件后缀
Java 源文件	.java
Java 字节码文件	.class

### 2.2 常用文件名(Common File Names)

常用的文件名包括：

文件名	用途
GNUMakefile	makefiles 的首选文件名。我们采用 gnumake 来创建（build）软件。
README	概述特定目录下所含内容的文件的首选文件名

## 3 文件组织(File Organization)

一个文件由被空行分割而成的段落以及标识每个段落的可选注释共同组成。超过 2000 行的程序难以阅读，应该尽量避免。"Java 源文件范例"提供了一个布局合理的 Java 程序范例。

### 3.1 Java 源文件(Java Source Files)

每个 Java 源文件都包含一个单一的公共类或接口。若私有类和接口与一个公共类相关联，可以将它们和公共类放入同一个源文件。公共类必须是这个文件中的第一个类或接口。

Java 源文件还遵循以下规则：

- 开头注释（参见["开头注释"](#)）
- 包和引入语句（参见["包和引入语句"](#)）
- 类和接口声明（参见["类和接口声明"](#)）

### 3.1.1 开头注释(Beginning Comments)

所有的源文件都应该在开头有一个 C 语言风格的注释，其中列出类名、版本信息、日期和版权声明：

```
/*
 * Classname
 *
 * Version information
 *
 * Date
 *
 * Copyright notice
 */
```

### 3.1.2 包和引入语句(Package and Import Statements)

在多数 Java 源文件中，第一个非注释行是包语句。在它之后可以跟引入语句。例如：

```
package java.awt;

import java.awt.peer.CanvasPeer;
```

### 3.1.3 类和接口声明(Class and Interface Declarations)

下表描述了类和接口声明的各个部分以及它们出现的先后次序。参见["Java源文件范例"](#)中一个包含注释的例子。

	类/接口声明的各部分	注解
1	类/接口文档注释 ( <code>/** ..... */</code> )	该注释中所需包含的信息，参见 <a href="#">"文档注释"</a>
2	类或接口的声明	
3	类/接口实现的注释	该注释应包含任何有关整个类或接口的信息，而这些信息又不适合

	(/*.....*/)如果有必要的话	作为类/接口文档注释。
4	类的(静态)变量	首先是类的公共变量，随后是保护变量，再后是包一级别的变量(没有访问修饰符， <code>access modifier</code> )，最后是私有变量。
5	实例变量	首先是公共级别的，随后是保护级别的，再后是包一级别的(没有访问修饰符)，最后是私有级别的。
6	构造器	
7	方法	这些方法应该按功能，而非作用域或访问权限，分组。例如，一个私有的类方法可以置于两个公有的实例方法之间。其目的是为了更便于阅读和理解代码。

## 4 缩进排版(Indentation)

4 个空格常被作为缩进排版的一个单位。缩进的确切解释并未详细指定(空格 vs. 制表符)。一个制表符等于 8 个空格(而非 4 个)。

### 4.1 行长度(Line Length)

尽量避免一行的长度超过 80 个字符，因为很多终端和工具不能很好处理之。

注意：用于文档中的例子应该使用更短的行长，长度一般不超过 70 个字符。

### 4.2 换行(Wrapping Lines)

当一个表达式无法容纳在一行内时，可以依据如下一般规则断开之：

- 在一个逗号后面断开
- 在一个操作符前面断开
- 宁可选择较高级别(**higher-level**)的断开，而非较低级别(**lower-level**)的断开
- 新的一行应该与上一行同一级别表达式的开头处对齐
- 如果以上规则导致你的代码混乱或者使你的代码都堆挤在右边，那就代之以缩进 8 个空格。

以下是断开方法调用的一些例子：

```
someMethod(longExpression1, longExpression2, longExpression3,
           longExpression4, longExpression5);

var = someMethod1(longExpression1,
                  someMethod2(longExpression2,
                              longExpression3));
```

以下是两个断开算术表达式的例子。前者更好，因为断开处位于括号表达式的外边，这是个较高级别的断开。

```

longName1 = longName2 * (longName3 + longName4 - longName5)
                + 4 * longname6; //PREFFER

longName1 = longName2 * (longName3 + longName4
                        - longName5) + 4 * longname6;
//AVOID

```

以下是两个缩进方法声明的例子。前者是常规情形。后者若使用常规的缩进方式将会使第二行和第三行移得很靠右，所以代之以缩进 8 个空格

```

//CONVENTIONAL INDENTATION
someMethod(int anArg, Object anotherArg, String yetAnotherArg,
           Object andStillAnother) {
    ...
}

//INDENT 8 SPACES TO AVOID VERY DEEP INDENTS
private static synchronized horkingLongMethodName(int anArg,
           Object anotherArg, String yetAnotherArg,
           Object andStillAnother) {
    ...
}

```

if 语句的换行通常使用 8 个空格的规则，因为常规缩进(4 个空格)会使语句体看起来比较费劲。比如：

```

//DON' T USE THIS INDENTATION
if ((condition1 && condition2)
    || (condition3 && condition4)
    ||!(condition5 && condition6)) { //BAD WRAPS
    doSomethingAboutIt();           //MAKE THIS LINE EASY TO MISS
}

//USE THIS INDENTATION INSTEAD
if ((condition1 && condition2)
    || (condition3 && condition4)
    ||!(condition5 && condition6)) {
    doSomethingAboutIt();
}

//OR USE THIS
if ((condition1 && condition2) || (condition3 && condition4)
    ||!(condition5 && condition6)) {

```

```
        doSomethingAboutIt();
    }
```

这里有三种可行的方法用于处理三元运算表达式：

```
alpha = (aLongBooleanExpression) ? beta : gamma;
```

```
alpha = (aLongBooleanExpression) ? beta
      : gamma;
```

```
alpha = (aLongBooleanExpression)
      ? beta
      : gamma;
```

## 5 注释(Comments)

Java 程序有两类注释：实现注释(implementation comments)和文档注释(document comments)。实现注释是那些在 C++ 中见过的，使用 `/*...*/` 和 `//` 界定的注释。文档注释(被称为 "doc comments") 是 Java 独有的，并由 `/**...*/` 界定。文档注释可以通过 `javadoc` 工具转换成 HTML 文件。

实现注释用以注释代码或者实现细节。文档注释从实现自由(implementation-free)的角度描述代码的规范。它可以被那些手头没有源码的开发人员读懂。

注释应被用来给出代码的总括，并提供代码自身没有提供的附加信息。注释应该仅包含与阅读和理解程序有关的信息。例如，相应的包如何被建立或位于哪个目录下之类的信息不应包括在注释中。

在注释里，对设计决策中重要的或者不是显而易见的地方进行说明是可以的，但应避免提供代码中已清晰表达出来的重复信息。多余的注释很容易过时。通常应避免那些代码更新就可能过时的注释。

注意：频繁的注释有时反映出代码的低质量。当你觉得被迫要加注释的时候，考虑一下重写代码使其更清晰。

注释不应写在用星号或其他字符画出来的大框里。注释不应包括诸如制表符和回退符之类的特殊字符。

### 5.1 实现注释的格式(Implementation Comment Formats)

程序可以有 4 种实现注释的风格：块(block)、单行(single-line)、尾端(trailing)和行末(end-of-line)。

#### 5.1.1 块注释(Block Comments)



块注释通常用于提供对文件，方法，数据结构和算法的描述。块注释被置于每个文件的开始处以及每个方法之前。它们也可以被用于其他地方，比如方法内部。在功能和方法内部的块注释应该和它们所描述的代码具有一样的缩进格式。

块注释之首应该有一个空行，用于把块注释和代码分割开来，比如：

```
/*
 * Here is a block comment.
 */
```

块注释可以以`/*-`开头，这样`indent(1)`就可以将之识别为一个代码块的开始，而不会重排它。

```
/*-
 * Here is a block comment with some very special
 * formatting that I want indent(1) to ignore.
 *
 *     one
 *         two
 *             three
 */
```

注意：如果你不使用`indent(1)`，就不必在代码中使用`/*-`，或为他人可能对你的代码运行`indent(1)`作让步。

参见["文档注释"](#)

### 5.1.2 单行注释(Single-Line Comments)

短注释可以显示在一行内，并与其后的代码具有一样的缩进层级。如果一个注释不能在一行内写完，就该采用块注释(参见["块注释"](#))。单行注释之前应该有一个空行。以下是一个Java代码中单行注释的例子：

```
if (condition) {

    /* Handle the condition. */
    ...
}
```

### 5.1.3 尾端注释(Trailing Comments)

极短的注释可以与它们所要描述的代码位于同一行，但是应该有足够的空白来分开代码和注释。若有多个短注释出现于大段代码中，它们应该具有相同的缩进。

以下是一个 Java 代码中尾端注释的例子：

```
if (a == 2) {  
    return TRUE;           /* special case */  
} else {  
    return isPrime(a);     /* works only for odd a */  
}
```

#### 5.1.4 行末注释(End-Of-Line Comments)

注释界定符"/"，可以注释掉整行或者一行中的一部分。它一般不用于连续多行的注释文本；然而，它可以用来注释掉连续多行的代码段。以下是所有三种风格的例子：

```
if (foo > 1) {  
  
    // Do a double-flip.  
    ...  
}  
else {  
    return false;          // Explain why here.  
}  
  
//if (bar > 1) {  
//  
//    // Do a triple-flip.  
//    ...  
//}  
//else {  
//    return false;  
//}
```

## 5.2 文档注释(Documentation Comments)

注意：此处描述的注释格式之范例，参见"[Java源文件范例](#)"

若想了解更多，参见"How to Write Doc Comments for Javadoc"，其中包含了有关文档注释标记的信息(@return, @param, @see)：

<http://java.sun.com/javadoc/writingdoccomments/index.html>

若想了解更多有关文档注释和 javadoc 的详细资料，参见 javadoc 的主页：

<http://java.sun.com/javadoc/index.html>

文档注释描述 Java 的类、接口、构造器，方法，以及字段(field)。每个文档注释都会被置于注释定界符`/**...*/`之中，一个注释对应一个类、接口或成员。该注释应位于声明之前：

```
/**
 * The Example class provides ...
 */
public class Example { ...
```

注意顶层(top-level)的类和接口是不缩进的，而其成员是缩进的。描述类和接口的文档注释的第一行(`/**`)不需缩进；随后的文档注释每行都缩进 1 格(使星号纵向对齐)。成员，包括构造函数在内，其文档注释的第一行缩进 4 格，随后每行都缩进 5 格。

若你想给出有关类、接口、变量或方法的信息，而这些信息又不适合写在文档中，则可使用实现块注释(见 5.1.1)或紧跟在声明后面的单行注释(见 5.1.2)。例如，有关一个类实现的细节，应放入紧跟在类声明后面的实现块注释中，而不是放在文档注释中。

文档注释不能放在一个方法或构造器的定义块中，因为 Java 会将位于文档注释之后的第一个声明与其相关联。

## 6 声明(Declarations)

### 6.1 每行声明变量的数量(Number Per Line)

推荐一行一个声明，因为这样以利于写注释。亦即，

```
int level; // indentation level
int size;  // size of table
```

要优于，

```
int level, size;
```

不要将不同类型变量的声明放在同一行，例如：

```
int foo,  fooarray[];  //WRONG!
```

注意：上面的例子中，在类型和标识符之间放了一个空格，另一种被允许的替代方式是使用制表符：

```
int      level;           // indentation level
int      size;            // size of table
Object   currentEntry;    // currently selected table entry
```

## 6.2 初始化(Initialization)

尽量在声明局部变量的同时初始化。唯一不这么做的理由是变量的初始值依赖于某些先前发生的计算。

## 6.3 布局(Placement)

只在代码块的开始处声明变量。（一个块是指任何被包含在大括号 "{" 和 "}" 中间的代码。）不要在首次用到该变量时才声明之。这会把注意力不集中的程序员搞糊涂，同时会妨碍代码在该作用域内的可移植性。

```
void myMethod() {
    int int1 = 0;           // beginning of method block

    if (condition) {
        int int2 = 0;      // beginning of "if" block
        ...
    }
}
```

该规则的一个例外是 for 循环的索引变量

```
for (int i = 0; i < maxLoops; i++) { ... }
```

避免声明的局部变量覆盖上一级声明的变量。例如，不要在内部代码块中声明相同的变量名：

```
int count;
...
myMethod() {
    if (condition) {
        int count = 0;    // AVOID!
        ...
    }
    ...
}
```

## 6.4 类和接口的声明(Class and Interface Declarations)

当编写类和接口是，应该遵守以下格式规则：

- 在方法名与其参数列表之前的左括号 "(" 间不要有空格
- 左大括号 "{" 位于声明语句同行的末尾
- 右大括号 "}" 另起一行，与相应的声明语句对齐，除非是一个空语句，"}" 应紧跟在 "{" 之后

```
class Sample extends Object {
    int ivar1;
    int ivar2;

    Sample(int i, int j) {
        ivar1 = i;
        ivar2 = j;
    }

    int emptyMethod() {}

    ...
}
```

- 方法与方法之间以空行分隔

## 7 语句(Statements)

### 7.1 简单语句(Simple Statements)

每行至多包含一条语句，例如：

```
argv++;          // Correct
argc--;          // Correct
argv++; argc--;  // AVOID!
```

### 7.2 复合语句(Compound Statements)

复合语句是包含在大括号中的语句序列，形如 "{ 语句 }"。例如下面各段。

- 被括其中的语句应该较之复合语句缩进一个层次
- 左大括号 "{" 应位于复合语句起始行的行尾；右大括号 "}" 应另起一行并与复合语句首行对齐。
- 大括号可以被用于所有语句，包括单个语句，只要这些语句是诸如 if-else 或 for 控制结构的一部分。这样便于添加语句而无需担心由于忘了加括号而引入 bug。

### 7.3 返回语句(return Statements)

一个带返回值的 return 语句不使用小括号 "()"，除非它们以某种方式使返回值更为显见。例如：

```
return;
```

```
return myDisk.size();

return (size ? size : defaultSize);
```

#### 7.4 if, if-else, if else-if else 语句(if, if-else, if else-if else Statements)

if-else 语句应该具有如下格式:

```
if (condition) {
    statements;
}

if (condition) {
    statements;
} else {
    statements;
}

if (condition) {
    statements;
} else if (condition) {
    statements;
} else {
    statements;
}
```

注意: if 语句总是用"{"和"}"括起来, 避免使用如下容易引起错误的格式:

```
if (condition) //AVOID! THIS OMITTS THE BRACES {}!
    statement;
```

#### 7.5 for 语句(for Statements)

一个 for 语句应该具有如下格式:

```
for (initialization; condition; update) {
    statements;
}
```

一个空的 for 语句(所有工作都在初始化, 条件判断, 更新子句中完成) 应该具有如下格式:

```
for (initialization; condition; update);
```

当在 **for** 语句的初始化或更新子句中使用逗号时,避免因使用三个以上变量,而导致复杂度提高。若需要,可以在 **for** 循环之前(为初始化子句)或 **for** 循环末尾(为更新子句)使用单独的语句。

## 7.6 while 语句(while Statements)

一个 **while** 语句应该具有如下格式

```
while (condition) {  
    statements;  
}
```

一个空的 **while** 语句应该具有如下格式:

```
while (condition);
```

## 7.7 do-while 语句(do-while Statements)

一个 **do-while** 语句应该具有如下格式:

```
do {  
    statements;  
} while (condition);
```

## 7.8 switch 语句(switch Statements)

一个 **switch** 语句应该具有如下格式:

```
switch (condition) {  
    case ABC:  
        statements;  
        /* falls through */  
    case DEF:  
        statements;  
        break;  
  
    case XYZ:  
        statements;  
        break;
```

```
default:
    statements;
    break;
}
```

每当一个 `case` 顺着往下执行时(因为没有 `break` 语句), 通常应在 `break` 语句的位置添加注释。上面的示例代码中就包含注释 `/* falls through */`。

## 7.9 try-catch 语句(try-catch Statements)

一个 `try-catch` 语句应该具有如下格式:

```
try {
    statements;
} catch (ExceptionClass e) {
    statements;
}
```

一个 `try-catch` 语句后面也可能跟着一个 `finally` 语句, 不论 `try` 代码块是否顺利执行完, 它都会被执行。

```
try {
    statements;
} catch (ExceptionClass e) {
    statements;
} finally {
    statements;
}
```

## 8 空白(White Space)

### 8.1 空行(Blank Lines)

空行将逻辑相关的代码段分隔开, 以提高可读性。

下列情况应该总是使用两个空行:

- 一个源文件的两个片段(section)之间
- 类声明和接口声明之间

下列情况应该总是使用一个空行:



- 两个方法之间
- 方法内的局部变量和方法的第一条语句之间
- 块注释（参见[5.1.1](#)）或单行注释（参见[5.1.2](#)）之前
- 一个方法内的两个逻辑段之间，用以提高可读性

## 8.2 空格(Blank Spaces)

下列情况应该使用空格：

- 一个紧跟着括号的关键字应该被空格分开，例如：

```
while (true) {
    ...
}
```

注意：空格不应该置于方法名与其左括号之间。这将有助于区分关键字和方法调用。

- 空白应该位于参数列表中逗号的后面
- 所有的二元运算符，除了"."，应该使用空格将之与操作数分开。一元操作符和操作数之间不因该加空格，比如：负号("-")、自增("++")和自减("--")。例如：

```
a += c + d;
a = (a + b) / (c * d);

while (d++ = s++) {
    n++;
}
printSize("size is " + foo + "\n");
```

- for 语句中的表达式应该被空格分开，例如：

```
for (expr1; expr2; expr3)
```

- 强制转型后应该跟一个空格，例如：

```
myMethod((byte) aNum, (Object) x);
myMethod((int) (cp + 5), ((int) (i + 3)) + 1);
```

## 9 命名规范(Naming Conventions)

命名规范使程序更易读，从而更易于理解。它们也可以提供一些有关标识符功能的信息，以助于理解代码，例如，不论它是一个常量，包，还是类。

标识符类型	命名规则	例子
包(Packages)	一个唯一包名的前缀总是全部小写的 ASCII 字母并且是一个顶级域名，通常是 com, edu, gov, mil, net, org, 或 1981 年 ISO 3166	com.sun.eng com.apple.quicktime.v2 edu.cmu.cs.bovik.cheese

	标准所指定的标识国家的英文双字符代码。包名的后续部分根据不同机构各自内部的命名规范而不尽相同。这类命名规范可能以特定目录名的组成来区分部门(department)，项目(project)，机器(machine)，或注册名(login names)。	
类(Classess)	命名规则：类名是个一名词，采用大小写混合的方式，每个单词的首字母大写。尽量使你的类名简洁而富于描述。使用完整单词，避免缩写词(除非该缩写词被更广泛使用，像 URL，HTML)	<pre>class Raster; class ImageSprite;</pre>
接口 (Interfaces)	命名规则：大小写规则与类名相似	<pre>interface RasterDelegate; interface Storing;</pre>
方法 (Methods)	方法名是一个动词，采用大小写混合的方式，第一个单词的首字母小写，其后单词的首字母大写。	<pre>run(); runFast(); getBackground();</pre>
变量 (Variables)	除了变量名外，所有实例，包括类，类常量，均采用大小写混合的方式，第一个单词的首字母小写，其后单词的首字母大写。变量名不应以下划线或美元符号开头，尽管这在语法上是允许的。变量名应简短且富于描述。变量名的选用应该易于记忆，即，能够指出其用途。尽量避免单个字符的变量名，除非是一次性的临时变量。临时变量通常被取名为 i, j, k, m 和 n，它们一般用于整型；c, d, e，它们一般用于字符型。	<pre>char c; int i; float myWidth;</pre>
实例变量 (Instance Variables)	大小写规则和变量名相似，除了前面需要一个下划线	<pre>int _employeeId; String _name; Customer _customer;</pre>
常量 (Constants)	类常量和 ANSI 常量的声明，应该全部大写，单词间用下划线隔开。(尽量避免 ANSI 常量，容易引起错误)	<pre>static final int MIN_WIDTH = 4; static final int MAX_WIDTH = 999; static final int GET_THE_CPU = 1;</pre>

## 10 编程惯例(Programming Practices)

### 10.1 提供对实例以及类变量的访问控制(Providing Access to Instance and Class Variables)

若没有足够理由，不要把实例或类变量声明为公有。通常，实例变量无需显式的设置(set)和获取(gotten)，通常这作为方法调用的边缘效应 (side effect)而产生。

一个具有公有实例变量的恰当例子，是类仅作为数据结构，没有行为。亦即，若你要使用一个结构(struct)而非一个类(如果 java 支持结构的话)，那么把类的实例变量声明为公有合适的。

## 10.2 引用类变量和类方法(Referring to Class Variables and Methods)

避免用一个对象访问一个类的静态变量和方法。应该用类名替代。例如：

```
classMethod();           //OK
AClass.classMethod();    //OK
anObject.classMethod();  //AVOID!
```

## 10.3 常量(Constants)

位于 for 循环中作为计数器值的数字常量，除了 -1, 0 和 1 之外，不应被直接写入代码。

## 10.4 变量赋值(Variable Assignments)

避免在一个语句中给多个变量赋相同的值。它很难读懂。例如：

```
fooBar.fChar = barFoo.lchar = 'c'; // AVOID!
```

不要将赋值运算符用在容易与相等关系运算符混淆的地方。例如：

```
if (c++ = d++) {           // AVOID! (Java disallows)
    ...
}
```

应该写成

```
if ((c++ = d++) != 0) {
    ...
}
```

不要使用内嵌(embedded)赋值运算符试图提高运行时的效率，这是编译器的工作。例如：

```
d = (a = b + c) + r;        // AVOID!
```

应该写成

```
a = b + c;
d = a + r;
```

## 10.5 其它惯例(Miscellaneous Practices)

### 10.5.1 圆括号(Parentheses)

一般而言，在含有多种运算符的表达式中使用圆括号来避免运算符优先级问题，是个好方法。即使运算符的优先级对你而言可能很清楚，但对其他人未必如此。你不能假设别的程序员和你一样清楚运算符的优先级。

```
if (a == b && c == d)    // AVOID!
if ((a == b) && (c == d)) // RIGHT
```

### 10.5.2 返回值(Returning Values)

设法让你的程序结构符合目的。例如：

```
if (booleanExpression) {
    return true;
} else {
    return false;
}
```

应该代之以如下方法：

```
return booleanExpression;
```

类似地：

```
if (condition) {
    return x;
}
return y;
```

应该写做：

```
return (condition ? x : y);
```

### 10.5.3 条件运算符"?"前的表达式(Expressions before '?' in the Conditional Operator)

如果一个包含二元运算符的表达式出现在三元运算符"?:"的"?"之前，那么应该给表达式添上一对圆括号。例如：

```
(x >= 0) ? x : -x;
```

#### 10.5.4 特殊注释(Special Comments)

在注释中使用 XXX 来标识某些未实现(bogus)的但可以工作(works)的内容。用 FIXME 来标识某些假的和错误的内容。

### 11 代码范例(Code Examples)

#### 11.1 Java 源文件范例(Java Source File Example)

下面的例子，展示了如何合理布局一个包含单一公共类的Java源程序。接口的布局与其相似。更多信息参见["类和接口声明"](#)以及["文档注释"](#)。

```
/*
 * @(#)Blah. java          1.82 99/03/18
 *
 * Copyright (c) 1994-1999 Sun Microsystems, Inc.
 * 901 San Antonio Road, Palo Alto, California, 94303, U.S.A.
 * All rights reserved.
 *
 * This software is the confidential and proprietary information of Sun
 * Microsystems, Inc. ("Confidential Information"). You shall not
 * disclose such Confidential Information and shall use it only in
 * accordance with the terms of the license agreement you entered into
 * with Sun.
 */
```

```
package java.blah;
```

```
import java.blah.blahdy.BlahBlah;
```

```
/**
 * Class description goes here.
 *
 * @version    1.82 18 Mar 1999
 * @author     Firstname Lastname
 */
public class Blah extends SomeClass {
    /* A class implementation comment can go here. */
}
```

```

/** classVar1 documentation comment */
public static int classVar1;

/**
 * classVar2 documentation comment that happens to be
 * more than one line long
 */
private static Object classVar2;

/** instanceVar1 documentation comment */
public Object instanceVar1;

/** instanceVar2 documentation comment */
protected int instanceVar2;

/** instanceVar3 documentation comment */
private Object[] instanceVar3;

/**
 * ...constructor Blah documentation comment...
 */
public Blah() {
    // ...implementation goes here...
}

/**
 * ...method doSomething documentation comment...
 */
public void doSomething() {
    // ...implementation goes here...
}

/**
 * ...method doSomethingElse documentation comment...
 * @param someParam description
 */
public void doSomethingElse(Object someParam) {
    // ...implementation goes here...
}
}

```

# Spring 快速入门教程

——开发你的第一个 Spring 程序

翻译整理：Hantsy Bai<hantsy@tom.com>

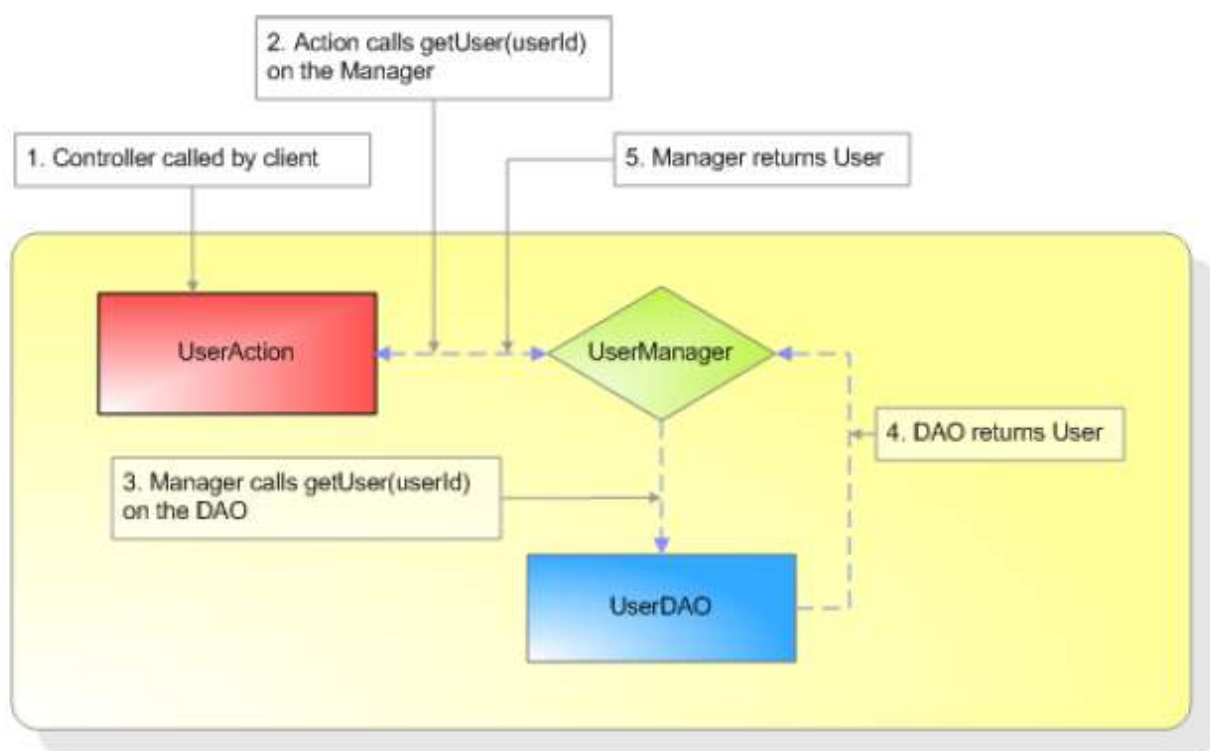
本章学习用 struts MVC 框架作前端，Spring 做中间层，Hibernate 作后端来开发一个简单的 Spring 应用程序。在第 4 章将使用 Spring MVC 框架对它进行重构。

本章包含以下内容：

- ◆ 编写功能性测试。
- ◆ 配置 Hibernate 和 Transaction。
- ◆ 载入 Spring 的 applicationContext.xml 文件。
- ◆ 设置业务代理(business delegates)和 DAO 的依赖性。
- ◆ 把 spring 写入 Struts 程序。

## 概述

你将会创建一个简单的程序完成最基本的 CRUD(Create, Retrieve, Update 和 Delete) 操作。这个程序叫 MyUsers，作为本书的样例。这是一个 3 层架构的 web 程序，通过一个 Action 来调用业务代理，再通过它来回调 DAO 类。下面的流程图表示了 MyUsers 是如何工作的。数字表明了流程的先后顺序，从 web 层 (UserAction) 到中间层 (UserManager)，再到数据层 (UserDAO)，然后返回。



鉴于大多数读者都比较熟悉 struts，本程序采用它作为 MVC 框架。Spring 的魅力在于它宣称式的事务处理，依赖性的绑定和持久性的支持。第 4 章中将用 Spring 框架对它进行重构。

接下来你会进行以下几个步骤：

1. 下载 Struts 和 Spring。
2. 创建项目目录和 ant Build 文件。
3. 为持久层创建一个单元测试(unit tests)。
4. 配置 Hibernate 和 Spring。
5. 编写 HIbernate DAO 的实现。
6. 进行单元测试，通过 DAO 验证 CRUD。



7. 创建一个 Manager 来声明事务处理。
8. 为 struts Action 编写测试程序。
9. 为 web 层创建一个 Action 和 model (DynaActionForm)。
10. 进行单元测试，通过 Action 验证 CRUD。
11. 创建 JSP 页面，以通过浏览器来进行 CRUD 操作。
12. 通过浏览器来验证 JSP 页面的功能。
13. 用 velocity 模板替换 JSP 页面。
14. 使用 Commons Validator 添加验证。

## 下载 Struts 和 Spring

1. 下载安装以下组件:

- JDK1.4.2(或以上)
- Tomcat5.0+
- Ant 1.6.1+

2. 设置以下环境变量:

- JAVA\_HOME
- ANT\_HOME
- CATALINA\_HOME

3. 把以下路径添加到 PATH 中:

- JAVA\_HOME/bin
- ANT\_HOME/bin
- CATALINA\_HOME/bin

为了开发基于 java 的 web 项目, 开发人员必须事先下载必需的 jars, 建好开发目录结构和 ant build 文件。对于单一的 struts 项目, 可以利用 struts 中现成的 struts-blank.war。对于基于 Spring MVC 框架的项目, 可以用 Spring 中的 webapp-minimal.war。这只为开发作准备, 两者都没有进行 struts-spring 集成, 也没有考虑单元测试。为此, 我们为读者准备了 Equinox。

Equinox 为开发 Struts-spring 式的程序提供一个基本框架。它已经定义好了目录结构, 和 ant build 文件(针对 compiling, deploying, testing), 并且提供了 struts, spring, Hibernate 开发要用到的 jars 文件。Equinox 中大部分目录结构和 ant build 文件来自我的开源项目——AppFuse。可以说, Equinox 是一个简化的 AppFuse, 它在最小配置情况下, 为快速 web 开发提供了便利。由于 Equinox 源于 AppFuse, 所以在包名, 数据库名, 及其它地方都找到它的影子。这是为让你从基于 Equinox 的程序过渡到更为复杂的 AppFuse。

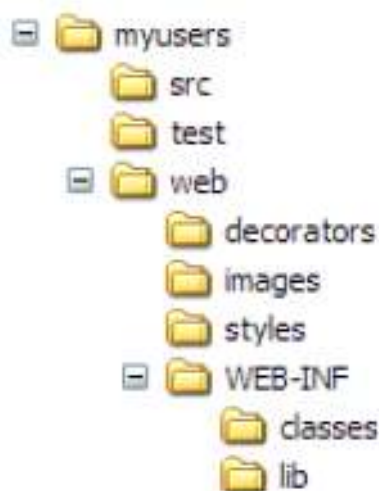
从 <http://sourcebeat.com/downloads> 上下载 Equinox, 解压到一个合适的位置, 开始准备 MyUsers 的开发。

## 创建项目目录和 ant build 文件

为了设置初始的目录结构，把下载的 Equinox 解压到硬盘。建议 windows 用户把项目放在 C:\Source，Unix/Linux 用户放在 ~/dev (译注：在当前用户目录建一个 dev 目录) 中。windows 用户可以设置一个 HOME 环境变量，值为 C:\Source。最简单的方法是把 Equinox 解压到你的喜欢的地方，进入 equinox 目录，运行 `ant new -Dapp.name=myusers`。

tips: 在 windows 使用 cygwin ([www.cygwin.org](http://www.cygwin.org)) 就可以像 Unix/Linux 系统一样使用正斜杠，本书所有路径均采用正斜杠。其它使用反斜杠系统 (如 windows 中命令行窗口) 的用户请作相应的调整。

现在 MyUsers 程序已经有如下的目录结构：



Equinox 包含一个简单而功能强大的 build.xml, 它可以用 ant 来进行编译，布署，和测试。ant 中已经定义好 targets, 在 equinox 运行 ant, 将看到如下内容：

```
[echo] Available targets are:
[echo] compile --> Compile all Java files
[echo] war --> Package as WAR file
[echo] deploy --> Deploy application as directory
[echo] deploywar --> Deploy application as a WAR file
[echo] install --> Install application in Tomcat
[echo] remove --> Remove application from Tomcat
[echo] reload --> Reload application in Tomcat
[echo] start --> Start Tomcat application
[echo] stop --> Stop Tomcat application
[echo] list --> List Tomcat applications
```

[echo] clean --> Deletes compiled classes and WAR

[echo] new --> Creates a new project

Equinox 支持 tomcat 的 ant tasks(任务)。这些已经集成在 Equinox 中，解讲一下如何进行集成的有助于理解它们的工作原理。

## **Tomcat 和 ant**

tomcat 中定义了一组任务，可以通过 Manager 来安装(install), 删除(remove), 重载(reload)webapps。要使用这些任务，可以把所有的定义写在一个属性文件中。在 Equinox 的根目录下，有一个名为 tomcatTasks.properties 包含如下内容。

```
deploy=org.apache.catalina.ant.DeployTask
undeploy=org.apache.catalina.ant.UndeployTask
remove=org.apache.catalina.ant.RemoveTask
reload=org.apache.catalina.ant.ReloadTask
start=org.apache.catalina.ant.StartTask
stop=org.apache.catalina.ant.StopTask
list=org.apache.catalina.ant.ListTask
```

在 build.xml 定义一些任务来安装，删除，重新载入应用程序。

```
<!-- Tomcat Ant Tasks -->
<taskdef file="tomcatTasks.properties">
  <classpath>
    <pathelement path="${tomcat.home}/server/lib/catalina-ant.jar"/>
  </classpath>
</taskdef>
<target name="install" description="Install application in Tomcat"
depends="war">
  <deploy url="${tomcat.manager.url}" username="${tomcat.manager.username}"
password="${tomcat.manager.password}" path="/${webapp.name}" war="file:${
{dist.dir}/${webapp.name}.war"/>
</target>
<target name="remove" description="Remove application from Tomcat">
  <undeploy url="${tomcat.manager.url}" username="${tomcat.manager.username}"
password="${tomcat.manager.password}" path="/${webapp.name}"/>
```

```

</target>
<target name="reload" description="Reload application in Tomcat">
  <reload url="${tomcat.manager.url}" username="${tomcat.manager.username}"
password="${tomcat.manager.password}" path="/${webapp.name}"/>
</target>
<target name="start" description="Start Tomcat application">
  <start url="${tomcat.manager.url}" username="${tomcat.manager.username}"
password="${tomcat.manager.password}" path="/${webapp.name}"/>
</target> <target name="stop" description="Stop Tomcat application">
  <stop url="${tomcat.manager.url}" username="${tomcat.manager.username}"
password="${tomcat.manager.password}" path="/${webapp.name}"/>
</target>
<target name="list" description="List Tomcat applications">
  <list url="${tomcat.manager.url}"
    username="${tomcat.manager.username}"
    password="${tomcat.manager.password}"/>
</target>

```

在上面列出的 target 中，必须定义一些\${tomcat.\*}变量。在根目录下有一个 build.properties 默认定义如下：

```

# Properties for Tomcat Server
tomcat.manager.url=http://localhost:8080/manager
tomcat.manager.username=admin
tomcat.manager.password=admin

```

确保 admin 用户可以访问 Manager 应用，打开\$CATALINA\_HOME/conf/tomcat-users.xml 中是否存在下面一行。如果不存在，请自己添加。注意，roles 属性可能是一个以逗号(“,”)隔开的系列。

```

<user username="admin" password="admin" roles="manager"/>

```

为了测试所有修改，保存所有文件，启动 tomcat。从命令行中进行 myusers 目录，运行 ant list，可以看到 tomcat server 上运行的应用程序。

```
Cygwin
$cd myusers
$ant list
Buildfile: build.xml

list:
[[list] OK - Listed applications for virtual host localhost
[[list] /admin:running:0:../server/webapps/admin
[[list] /balancer:running:0:balancer
[[list] /:running:0:C:\Tools\jakarta-tomcat-5.0.19\webapps\ROOT
[[list] /manager:running:0:../server/webapps/manager

BUILD SUCCESSFUL
Total time: 1 second
$
```

好了，现在运行 `ant deploy` 来安装 MyUsers。打开浏览器，在地址栏中输入 `http://localhost:8080/myusers`，出现如图 2.4 的“Equinox Welcome”画面。



下一节，将写一个 User 对象和一个维护其持久性的 Hibernate DAO 对象。用 Spring 来管理 DAO 类及其依赖性。最后会写一个业务代理，用到 AOP 和声明式事务处理。

## 为持久层编写单元测试

在 myUsers 程序，使用 Hibernate 作为持久层。Hibernate 是一个 O/R 映像框架，用来关联 java 对象和数据库中的表(tables)。它使得对象的 CRUD 操作变得非常简单，Spring 结合了 Hibernate 变得更加容易。从 Hibernate 转向 Spring+Hibernate 会减少 75%的代码。这主要是因为，ServiceLocator 和一些 DAOFactory 类的废弃，spring 的实时异常代替了 Hibernate 的检测式的异常。

写一个单元测试有助于规范 UserDao 接口。为 UserDao 写一个 JUnit 测试程序，要完成以下几步：

1.在 test/org/appfuse/dao 下新建一个 UserDaoTest.java 类。它继承了同一个包中的 BaseDAOTestCase，其父类初始化了 Spring 的 ApplicationContext(来自 web/WEB-INF/applicationContext.xml)，以下是 JUnit 测试的代码。

```
package org.appfuse.dao;

// use your IDE to handle imports

public class UserDaoTest extends BaseDAOTestCase {
    private User user = null;
    private UserDao dao = null;
    protected void setUp() throws Exception {
        log = LogFactory.getLog(UserDaoTest.class);
        dao = (UserDao) ctx.getBean("userDAO");
    }
    protected void tearDown() throws Exception {
        dao = null;
    }
    public static void main(String[] args) {
        junit.textui.TestRunner.run(UserDaoTest.class);
    }
}
```

这个类还不能编译，因为还没有 UserDao 接口。在这之前，来写一些来验证 User 的 CRUD 操作。

2.为 UserDaoTest 类添加 testSave 和 testAddAndRemove 方法，如下：

```
public void testSaveUser() throws Exception {
```

```

        user = new User();
        user.setFirstName("Rod");
        user.setLastName("Johnson");
        dao.saveUser(user);
        assertTrue("primary key assigned", user.getId() != null);
        log.info(user);
        assertTrue(user.getFirstName() != null);
    }

    public void testAddAndRemoveUser() throws Exception {
        user = new User();
        user.setFirstName("Bill");
        user.setLastName("Joy");
        dao.saveUser(user);
        assertTrue(user.getId() != null);
        assertTrue(user.getFirstName().equals("Bill"));
        if (log.isDebugEnabled()) {
            log.debug("removing user...");
        }
        dao.removeUser(user.getId());
        assertNull(dao.getUser(user.getId()));
    }
}

```

从这些方法中可以看到，你需要在 UserDao 创建以下方法：

```

saveUser(User)
removeUser(Long)
getUser(Long)
getUsers() (返回数据库的所有用户)

```

3.在 src/org/appfuse/dao 目录下建一个名为 UserDao.java 类的，输入以下代码：

tips:如果你使用 eclipse,idea 之类的 IDE，左边会出现在一个灯泡，提示类不存在，可以即时创建。

```

package org.appfuse.dao;

// use your IDE to handle imports

public interface UserDao extends DAO {
    public List getUsers();
    public User getUser(Long userId);
}

```



```
public void saveUser(User user);  
public void removeUser(Long userId);  
}
```

为了 UserDao.java, UserDaoTest.java 编译通过，还要建一个 User.java 类。

4.在 src/org/appfuse/model 下建一个 User.java 文件，添加几个成员变量：

id, firstName, lastName，如下。

```
package org.appfuse.model;  
  
public class User extends BaseObject {  
private Long id;  
private String firstName;  
private String lastName;
```

/\* 用你熟悉的 IDE 来生成 getters 和 setters，Eclipse 中右击> Source -> Generate Getters and Setters \*/

```
}
```

注意，你继承了 BaseObject 类，它包含几个有用的方法：toString(), equals(), hashCode(), 后两个是 Hibernate 必须的。建好 User 后，用 IDE 打开 UserDao 和 UserDaoTest 两个类，优化导入。

## 配置 Hibernate 和 Spring

现在已经有了 POJO(Plain Old Java Object),写一个映像文件 Hibernate 就可能维护它。

1.在 org/appfuse/model 中新建一个名为 User.hbm.xml 文件，内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 2.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">
<hibernate-mapping>
<class name="org.appfuse.model.User" table="app_user">
<id name="id" column="id" unsaved-value="0">
    <generator class="increment" />
</id>
<property name="firstName" column="first_name" not-null="true"/>
<property name="lastName" column="last_name" not-null="true"/>
</class>
</hibernate-mapping>
```

2.在 web/WEB-INF/下的 applicationContext.xml 中添加映像。打开文件，找到<property name= mappingResources >，改成如下：

```
<property name="mappingResources">
<list>
    <value>org/appfuse/model/User.hbm.xml</value>
</list>
</property>
```

在 applicationContext.xml 中，你可以看到数据库是怎么工作的，Hibernate 和 Spring 是如何协作的。Equinox 会使用名为 db/appfuse 的 HSQL 数据库。它将在你的 ant “db”目录下创建，详细配置在 “How Spring Is Configured in Equinox ”一节中描述。

3.运行 ant deploy reload(Tomcat 正在运行)，在 Tomcat 控制台的日志中可以看到，数据表正在创建。

INFO - SchemaExport.execute(98) | Running hbm2ddl schema export

INFO - SchemaExport.execute(117) | exporting generated schema to database

INFO - ConnectionProviderFactory.newConnectionProvider(53) | Initializing connection

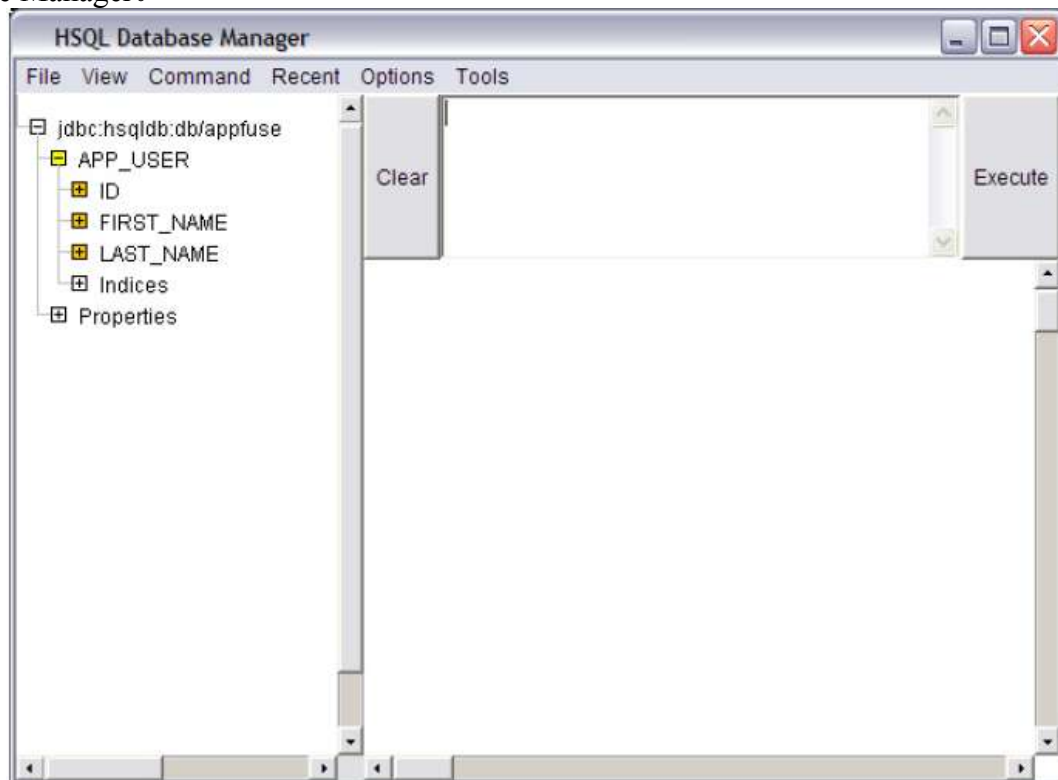
provider: org.springframework.orm.hibernate.LocalDataSourceConnectionProvider

INFO - DriverManagerDataSource.getConnectionFromDriverManager(140) | Creating new JDBC connection to [jdbc:hsqldb:db/appfuse]

INFO - SchemaExport.execute(160) | schema export complete

Tip: 如果你想看到更多或更少的日志, 修改 web/WEB-INF/ classes/log4j.xml 的设置。

4.为了验证数据库已经建好, 运行 ant browser 启动 hsql console 。你会看到如的 HSQL Database Manager。



## **Equinox** 中 **spring** 是怎么配置的

使用 Spring 配置任何基于 j2ee 的 web 程序都很简单。至少, 你简单的添加 Spring 的 ContextLoaderListener 到你的 web.xml 中。

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

这是一个 ServletContextListener, 它会在启动 web 应用进行初始化。默认情况下, 它会

查找 web/WEB-INF/applicationContext.xml 文件，你可以指定名为 contextConfigLocation 的 <context-param>元素来进行修改，例如：

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/sampleContext.xml</param-value>
</context-param>
```

<param-value>元素可以是以空格或是逗号隔开的一系列路径。在 Equinox 中，Spring 的配置使用了这个 Listener 和默认的 contextConfigLocation。

那么，Spring 怎么知道 Hibernate 的存在？这就 Spring 的魅力所在，它让依赖性的绑定变得非常简单。请参阅 applicationContext.xml 的全部内容：

```
<?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="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
            <property name="driverClassName">
                <value>org.hsqldb.jdbcDriver</value>
            </property>
            <property name="url">
                <value>jdbc:hsqldb:db/appfuse</value>
            </property>
            <property name="username">
                <value>sa</value>
            </property>
            <property name="password">
                <value></value>
            </property> </bean>
        <!-- Hibernate SessionFactory -->
        <bean id="sessionFactory"
class="org.springframework.orm.hibernate.LocalSessionFactoryBean">
            <property name="dataSource">
                <ref local="dataSource"/>
```

```

</property>
<property name="mappingResources">
<list>
    <value>org/appfuse/model/User.hbm.xml</value>
</list>
</property>
<property name="hibernateProperties">
<props>
    <prop key="hibernate.dialect"> net.sf.hibernate.dialect.HSQLDialect </prop>
    <prop key="hibernate.hbm2ddl.auto">create</prop>
</props>
</property>
</bean>
<!-- Transaction manager for a single Hibernate SessionFactory (alternative to JTA) -->
<bean id="transactionManager"
class="org.springframework.orm.hibernate.HibernateTransactionManager">
<property name="sessionFactory">
    <ref local="sessionFactory"/>
</property>
</bean>
</beans>

```

第一 bean 代表 HSQL 数据库，Spring 仅仅是调用 LocalSessionFactoryBeanr 的 setDataSource(DataSource)使之工作。如果你想用 JNDI DataSource 替换，可以 bean 的定义改成类似下面的几行：

```

<bean id="dataSource" class="org.springframework.jndi.JndiObjectFactoryBean">
<property name="jndiName">
    <value>java:comp/env/jdbc/appfuse</value>
</property>
</bean>

```

hibernate.hbm2ddl.auto 属性在 sessionFactory 定义中，这个属性是为了在应用启动时自动创建表，也可能是 update 或 create-drop。

最后一个 bean 是 transactionManager(你也可以使用 JTA transaction)，在处理跨越两个

数据库的分布式的事务处理中尤其重要。如果你想使用 jta transaction manager，将此 bean 的 class 属性改成

```
org.springframework.transaction.jta.JtaTransactionManager
```

下面实现 UserDao 类。

## 用 hibernate 实现 UserDAO

为了实现 Hibernate UserDAO，需要完成以下几步：

1. 在 src/org/appfuse/dao/hibernate(你要新建这个目录/包) 新建一个文件

UserDAOHibernate.java。这个类继承了 HibernateDaoSupport 类，并实现了 UserDAO 接口。

```
package org.appfuse.dao.hibernate;

// use your IDE to handle imports

public class UserDAOHibernate extends HibernateDaoSupport implements UserDAO {
    private Log log = LogFactory.getLog(UserDAOHibernate.class);

    public List getUsers() {
        return getHibernateTemplate().find("from User");
    }

    public User getUser(Long id) {
        return (User) getHibernateTemplate().get(User.class, id);
    }

    public void saveUser(User user) {
        getHibernateTemplate().saveOrUpdate(user);
        if (log.isDebugEnabled()) {
            log.debug( "userId set to: " + user.getID());
        }
    }

    public void removeUser(Long id) {
        Object user = getHibernateTemplate().load(User.class, id);
        getHibernateTemplate().delete(user);
    }
}
```

Spring 的 HibernateDaoSupport 类是一个方便实现 Hibernate DAO 的超类，你可以它的

一些有用的方法，来获得 Hibernate DAO 或是 SessionFactory。最方便的方法是 `getHibernateTemplate()`，它返回一个 `HibernateTemplate` 对象。这个模板把检测式异常 (checked exception) 包装成实时式异常 (runtime exception)，这使得你的 DAO 接口无需抛出 Hibernate 异常。

程序还没有把 UserDAO 绑定到 UserDAOHibernate 上，必须创建它们之间的关联。

2. 在 Spring 配置文件 (web/WEB-INF/applicationContext.xml) 中添加以下内容：

```
<bean id="userDAO" class="org.appfuse.dao.hibernate.UserDAOHibernate">
    <property name="sessionFactory">
        <ref local="sessionFactory"/>
    </property>
</bean>
```

这样就在你的 UserDAOHibernate (从 HibernateDaoSupport 的 `setSessionFactory` 继承) 中构建了一个 Hibernate Session Factory。Spring 会检测一个 Session (也就是，它在 web 层是开放的) 是否已经存在，并且直接使用它，而不是新建一个。这样你可以使用 Spring 流行的 “Open Session in View” 模式来载入 collections。

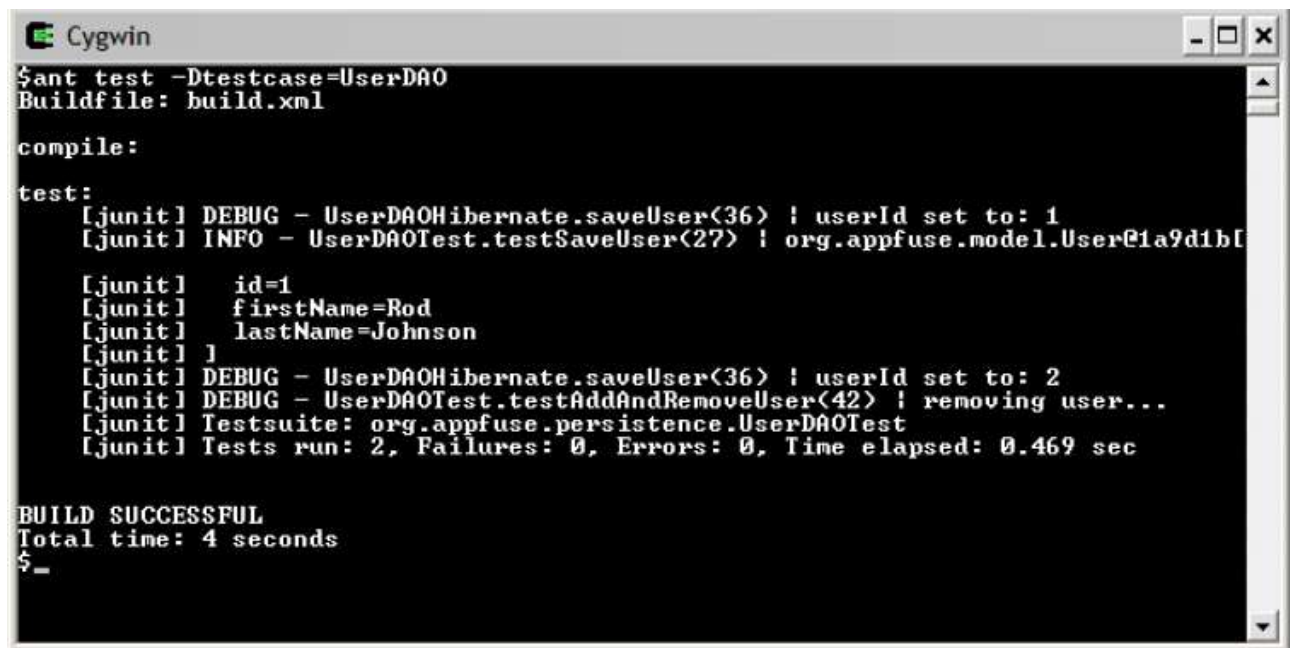


## 进行单元测试，来验证 DAO 的 CRUD 操作

在进行第一个测试之前，把你的日志级别从 “INFO”调到 “WARN”。

1.把 log4j.xml(在 web/WEB-INF/classes 目录下)中<level value="INFO"/>改为<level value="WARN"/>。

2.键入 ant test 来运行 UserDAOTest。如果你有多个测试，你必须用 ant test -Dtestcase=UserDAOTest 来指定要运行的测试。运行之后，你会如下类似的一些日志信息。



```
Cygwin
$ant test -Dtestcase=UserDAO
Buildfile: build.xml

compile:

test:
[junit] DEBUG - UserDAOHibernate.saveUser(36) : userId set to: 1
[junit] INFO - UserDAOTest.testSaveUser(27) : org.appfuse.model.User@1a9d1b[
    id=1
    firstName=Rod
    lastName=Johnson
[junit] ]
[junit] DEBUG - UserDAOHibernate.saveUser(36) : userId set to: 2
[junit] DEBUG - UserDAOTest.testAddAndRemoveUser(42) : removing user...
[junit] Testsuite: org.appfuse.persistence.UserDAOTest
[junit] Tests run: 2, Failures: 0, Errors: 0, Time elapsed: 0.469 sec

BUILD SUCCESSFUL
Total time: 4 seconds
$_
```

## 创建 Manager，声明事务处理(create manager and declare transactions)

J2EE 开发中建议将各层进行分离。换言之，不要把数据层(DAOs)和 web 层(servlets)混在一起。使用 Spring, 很容易做到这一点，但使用“业务代理”(business delegate)模式，可以对这些层进一步分离。

使用业务代理模式的主要原因是：

- 大多数持久层组件执行一个业务逻辑单元，把逻辑放在一非 web 类中的最大好处是，web service 或是丰富平台客户端(rich platform client)可以像使用 servlet 一样来用同一 API。
- 大多数业务逻辑都在同一方法中完成，当然可能多个 DAO。使用业务代理，使得你可以高级业务代理层(level)使用 Spring 的声明式业务代理特性。

MyUsers 应用中 UserManager 和 UserDAO 拥有相同的一个方法。主要不同的是 Manager 对于 web 更为友好(web-friendly)，它可以接受 Strings, 而 UserDAO 只能接受 Longs, 并且它可以在 saveUser 方法中返回一个 User 对象。这在插入一个新用户比如，要获得主键，是非常方便的。Manager(或称为业务代理)中也可以添加一些应用中所需要的其它业务逻辑。

1. 启动“service”层，在 test/org/appfuse/service(你必须先建好这个目录)中新建一个 UserManagerTest 类，这个类继承了 JUnit 的 TestCase 类，代码如下：

```
package org.appfuse.service;

// use your IDE to handle imports

public class UserManagerTest extends TestCase {

    private static Log log = LogFactory.getLog(UserManagerTest.class);

    private ApplicationContext ctx;

    private User user;

    private UserManager mgr;

    protected void setUp() throws Exception {

        String[] paths = {"/WEB-INF/applicationContext.xml"};

        ctx = new ClassPathXmlApplicationContext(paths);

        mgr = (UserManager) ctx.getBean("userManager");

    }
```

```

protected void tearDown() throws Exception {
    user = null;
    mgr = null;
}

// add testXXX methods here
public static void main(String[] args) {
    junit.textui.TestRunner.run(UserDAOTest.class);
}

```

在 setup 方法中，使用 `ClassPathXmlApplicationContext` 把 `applicationContext.xml` 载入变量 `ApplicationContext` 中。载入 `ApplicationContext` 有几种途径，从 `classpath` 中，文件系统，或 web 应用内。这些方法将在第三章（The BeanFactory and How It Works.）中描述。

2. 输入第一个测试方法的代码，来验证 Manager 成功完成添加或是删除 User 对象。

```

public void testAddAndRemoveUser() throws Exception {
    user = new User();
    user.setFirstName("Easter");
    user.setLastName("Bunny");
    user = mgr.saveUser(user);
    assertTrue(user.getId() != null);
    if (log.isDebugEnabled()) {
        log.debug("removing user...");
    }

    String userId = user.getId().toString();
    mgr.removeUser(userId);
    user = mgr.getUser(userId);
    if (user != null) {
        fail("User object found in database!");
    }
}

```

这个测试实际上是一个集成测试(integration test)，而不是单元测试(unit test)。

为了更接近单元测试，可以使用 EasyMock 或是类似工具来“伪装”（fake）DAO。这样，就不必关心 ApplicationContext 和任何依赖 Spring API 的东西。建议在测试项目依赖（Hibernate, Spring, 自己的类）的内部构件，包括数据库。第 9 章，讨论重构 UserManagerTest，使用 mock 解决 DAO 的依赖性。

3. 为了编译 UserManagerTest，在 src/org/appfuse/service 中新建一个接口——UserManager。在 org.appfuse.service 包中创建这个类，代码如下：

```
package org.appfuse.service;

// use your IDE to handle imports

public interface UserManager {

    public List getUsers();

    public User getUser(String userId);

    public User saveUser(User user);

    public void removeUser(String userId);

}
```

4. 建一个名为 org.appfuse.service.impl 的子包，新建一个类实现 UserManager 接口的。

```
package org.appfuse.service.impl;

// use your IDE to handle imports

public class UserManagerImpl implements UserManager {

    private static Log log = LogFactory.getLog(UserManagerImpl.class);

    private UserDAO dao;

    public void setUserDAO(UserDAO dao) {
        this.dao = dao;
    }

    public List getUsers() {
        return dao.getUsers();
    }

    public User getUser(String userId) {
        User user = dao.getUser(Long.valueOf(userId));
        if (user == null) {
```

```

        log.warn("UserId '" + userId + "' not found in database.");
    }
    return user;
}

```

```

public User saveUser(User user) {
    dao.saveUser(user);
    return user;
}

```

```

public void removeUser(String userId) {
    dao.removeUser(Long.valueOf(userId));
}
}

```

这个类看不出你在使用 Hibernate。当你打算把持久层转向一种不同的技术时，这样做很重要。

这个类提供一个私有 dao 成员变量，和 setUserDAO 方法一样。这样能够让 Spring 能够表演“依赖性绑定”魔术(perform “dependency binding” magic)，把这些对象扎在一起。在使用 mock 重构这个类时，你必须在 UserManager 接口中添加 setUserDAO 方法。

5. 在进行测试之前，配置 Spring, 让 getBeans 返回一个 UserManagerImpl 类。在 web/WEB-INF/applicationContext.xml 文件中，添加以下几行：

```

<bean id="userManager" class="org.appfuse.service.UserManagerImpl">
    <property name="userDAO">
        <ref local="userDAO"/>
    </property>
</bean>

```

唯一的问题，你还没有使 Spring 的 AOP, 特别是声明式的事务处理发挥作用。

6. 为了达到目的，使用 ProxyFactoryBean 代替 userManager。ProxyFactoryBean 是一个类的不同的实现，这样 AOP 能够解释和覆盖调用的方法。在事务处理中，使用 TransactionProxyFactoryBean 代替 UserManagerImpl 类。在 context 文件中添加下面 bean 的定义：

```

<bean id="userManager"
class="org.springframework.transaction.interceptor.TransactionProxy
FactoryBean">

```

```

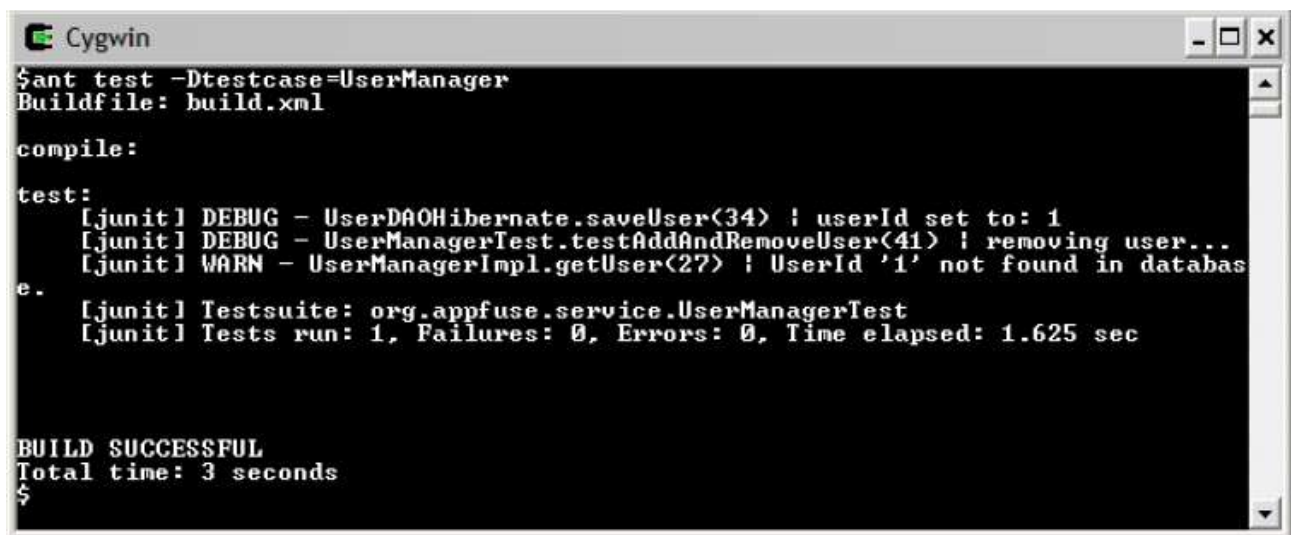
<property name="transactionManager">
    <ref local="transactionManager"/>
</property>
<property name="target">
    <ref local="userManagerTarget"/>
</property>
<property name="transactionAttributes">
<props>
    <prop key="save*">PROPAGATION_REQUIRED</prop>
    <prop key="remove*">PROPAGATION_REQUIRED</prop>
    <prop key="*">PROPAGATION_REQUIRED,readOnly</prop>
</props>
</property>
</bean>

```

从这个 xml 代码片断中可以看出，TransactionProxyFactoryBean 必须有一个 transactionManager 属性设置和 transactionAttributes 定义。

7. 让事务处理代理服务器(Transaction Proxy)知道你要模仿的对象：userManagerTarget。作为新 bean 的一部分，把原来的 userManager bean 改成拥有一个 userManagerTarget 的 id 属性。

编辑 applicationContext.xml 添加 userManager 和 userManagerTarget 的定义后，运行 ant test -Dtestcase=UserManager，看看终端输出：



```

Cygwin
$ant test -Dtestcase=UserManager
Buildfile: build.xml

compile:

test:
[junit] DEBUG - UserDAOHibernate.saveUser(34) : userId set to: 1
[junit] DEBUG - UserManagerTest.testAddAndRemoveUser(41) : removing user...
[junit] WARN - UserManagerImpl.getUser(27) : UserId '1' not found in database
e.
[junit] Testsuite: org.appfuse.service.UserManagerTest
[junit] Tests run: 1, Failures: 0, Errors: 0, Time elapsed: 1.625 sec

BUILD SUCCESSFUL
Total time: 3 seconds
$

```

8. 如果你想看看事务处理的执行和提交情况，在 log4j.xml 中添加：

```

<logger name="org.springframework.transaction">
<level value="DEBUG"/>

```

```
<!-- INFO does nothing -->  
</logger>
```

重新运行测试，将看到大量日志信息，如它相关的对象，事务的创建和提交等。测试完毕，最好删除上面的日志定义(logger)。

祝贺你！你已经实现了一个web应用的Spring/Hibernate后端解决方案。并且你已经用AOP和声明式业务处理配置好了业务代理。了不起，自我鼓励一下！(This is no small feat; give yourself a pat on the back!)

## 对象 **struts Action** 进行单元测试

业务代理和 DAO 都起作用，我们看看 MVC 框架吸盘(sucker)的上部。停！不止这些吧，你可以进行 C(Controller), 不是 V(View)。为了管理用户建一个 Struts Action, 继续进行驱动测试(Test-Driven)开发。

Equinox 是为 Struts 配置的。配置 Struts 需要在 web.xml 中进行一些设置，并在 web/WEB-INF 下定义一个 struts-config.xml 文件。由于 Struts 开发人员比较多，这里先使用 Struts。第 4 章用 Spring 进行处理。你想跳过这一节，直接学习 Spring MVC 方法，请参考第 4 章：Spring s MVC Framework。

在 test/org/appfuse/web 目录下新建一个文件 UserActionTest.java，开发你的第一个 Struts Action 单元测试。文件内容如下：

```
package org.appfuse.web;

// use your IDE to handle imports

public class UserActionTest extends MockStrutsTestCase {

    public UserActionTest(String testName) {
        super(testName);
    }

    public void testExecute() {
        setRequestPathInfo("/user");
        addRequestParameter("id", "1");
        actionPerform();
        verifyForward("success");
        verifyNoActionErrors();
    }
}
```



## 为 web 层创建 Action 和 Model(DynaActionForm)

1. 在 src/org/appfuse/web 下新建一个文件 UserAction.java。这个扩展了 DispatchAction，你可以花几分钟，在这个类中，创建 CRUD 方法。

```
package org.appfuse.web;

// use your IDE to handle imports

public class UserAction extends DispatchAction {
    private static Log log = LogFactory.getLog(UserAction.class);

    public ActionForward execute(ActionMapping mapping,
                                ActionForm form,
                                HttpServletRequest request,
                                HttpServletResponse response) throws Exception{
        request.getSession().setAttribute("test", "succeeded!");
        log.debug("looking up userId: " + request.getParameter("id"));
        return mapping.findForward("success");
    }
}
```

2. 为了配置 Struts, 使” /user” 这个请求路径代表其它。在 web/WEB-INF/struts-config.xml 中加入一个 action-mapping。打开文件加入：

```
<action path="/user" type="org.appfuse.web.UserAction">
    <forward name="success" path="/index.jsp"/>
</action>
```

3. 执行命令 ant test -Dtestcase=UserAction，你会看到友好的“BUILD SUCCESSFUL” 信息。

4. 在 struts-config.xml 中添加 form-bean 定义。对于 Struts ActionForm, 使用 DynaActionForm, 这是一个 javabean, 可以从 XML 定义中动态的创建。

```
<form-bean name="userForm" type="org.apache.struts.action.DynaActionForm">
    <form-property name="user" type="org.appfuse.model.User"/>
</form-bean>
```

这里没有使用具体的 ActionForm, 因为只使用一个 User 对象的包装器。理想情况下, 你可以 User 对象, 但会失去 Struts 环境下的一些特性: 验证属性(validate properties), checkbox 复位(reset checkboxs)。后面, 将演示用 Spring 怎么会更加简单,

它可以让你在 web 层使用 User 对象。

5. 修改 action 定义, 在 request 中使用这个 form。

```
<action path="/user" type="org.appfuse.web.UserAction" name="userForm"
scope="request">
    <forward name="success" path="/index.jsp"/>
</action>
```

6. 修改 UserActionTest, 测试不同的 CRUD 方法。

```
public class UserActionTest extends MockStrutsTestCase {
    public UserActionTest(String testName) {
        super(testName);
    }
    // Adding a new user is required between tests because HSQL creates
    // an in-memory database that goes away during tests.
```

```
    public void addUser() {
        setRequestPathInfo("/user");
        addRequestParameter("method", "save");
        addRequestParameter("user.firstName", "Juergen");
        addRequestParameter("user.lastName", "Hoeller");
        actionPerform();
        verifyForward("list");
        verifyNoActionErrors();
    }
```

```
    public void testAddAndEdit() {
        addUser();
        // edit newly added user
        addRequestParameter("method", "edit");
        addRequestParameter("id", "1");
        actionPerform();
        verifyForward("edit");
        verifyNoActionErrors();
    }
```

```
    public void testAddAndDelete() {
        addUser();
```

```

// delete new user
setRequestPathInfo("/user");
addRequestParameter("method", "delete");
addRequestParameter("user.id", "1");
actionPerform();
verifyForward("list");
verifyNoActionErrors();
}

public void testList() {
addUser();
setRequestPathInfo("/user");
addRequestParameter("method", "list");
actionPerform();
verifyForward("list");
verifyNoActionErrors();
List users = (List) getRequest().getAttribute("users");
assertNotNull(users);
assertTrue(users.size() == 1);
}
}

```

7. 修改 UserAction, 这样测试程序才能通过, 并能处理(客户端)请求。最简单的方法是添加 edit, save 和 delete 方法, 请确保你事先已经删除了 execute 方法。下面是修改过的 UserAction.java 文件。

```

public class UserAction extends DispatchAction {
private static Log log = LogFactory.getLog(UserAction.class);
private UserManager mgr = null;

public void setUserManager(UserManager userManager) {
    this.mgr = userManager;
}

public ActionForward delete(ActionMapping mapping,
                           ActionForm form,

```

```

        HttpServletRequest request,
        HttpServletResponse response) throws Exception{
    if (log.isDebugEnabled()) {
        log.debug("entering 'delete' method...");
    }
    mgr.removeUser(request.getParameter("user.id"));
    ActionMessages messages = new ActionMessages();
    messages.add(ActionMessages.GLOBAL_MESSAGE, new ActionMessage
("user.deleted"));
    saveMessages(request, messages);
    return list(mapping, form, request, response);
}

public ActionForward edit(ActionMapping mapping,
        ActionForm form,
        HttpServletRequest request,
        HttpServletResponse response) throws Exception {
    if (log.isDebugEnabled()) {
        log.debug("entering 'edit' method...");
    }
    DynaActionForm userForm = (DynaActionForm) form;
    String userId = request.getParameter("id");
    // null userId indicates an add

    if (userId != null) {
        User user = mgr.getUser(userId);
        if (user == null) {
            ActionMessages errors = new ActionMessages();
            errors.add(ActionMessages.GLOBAL_MESSAGE, new ActionMessage
("user.missing"));
            saveErrors(request, errors);
            return mapping.findForward("list");
        }
        userForm.set("user", user);
    }
}

```

```
return mapping.findForward("edit");
}
```

```
public ActionForward list(ActionMapping mapping,
                        ActionForm form,
                        HttpServletRequest request,
                        HttpServletResponse response) throws Exception {
    if (log.isDebugEnabled()) {
        log.debug("entering 'list' method...");
    }
    request.setAttribute("users", mgr.getUsers());
    return mapping.findForward("list");
}
```

```
public ActionForward save(ActionMapping mapping,
                        ActionForm form,
                        HttpServletRequest request,
                        HttpServletResponse response) throws Exception {
    if (log.isDebugEnabled()) {
        log.debug("entering 'save' method...");
    }
    DynaActionForm userForm = (DynaActionForm) form;
    mgr.saveUser((User)userForm.get("user"));
    ActionMessages messages = new ActionMessages();
    messages.add(ActionMessages.GLOBAL_MESSAGE, new ActionMessage
("user.saved"));
    saveMessages(request, messages);
    return list(mapping, form, request, response);
}
}
```

接下来，你可以修改这个类的 CRUD 方法。

8. 修改 struts-config.xml，使用 ContextLoaderPlugin 来配置 Spring 的 UserManager 设置。要配置 ContextLoaderPlugin，把下面内容添加到你的 struts-config.xml 中。

```
<plug-in className= org.springframework.web.struts.ContextLoaderPlugIn >
    <set-property property= contextConfigLocation value= /WEB-
```

```
INF/applicationContext.xml, /WEB-INF/action-servlet.xml />
```

```
</plug-in>
```

默认情况下这个插件会载入 action-servlet.xml 文件。要让 Test Action 知道你的 Manager，必须配置这个插件，如同载入 applicationContext。

9. 对每个使用 Spring 的 action, 定义一个 type=org.springframework.web.struts.DelegatingActionProxy 的 action-mapping, 为每个 Spring action 声明一个配对的 Spring bean。这样修改一下你的 action mapping 就能使用这个新类。

10. 为 DispatchAction 修改 action mapping。

为了让 DispatchAction 运行，在 mapping 中添加参数 parameter= “method” ，它表示(在一个 URL 或是隐藏字段 hidden field)要调用的方法，同时转向(forwards)edit 和 list forwards(参考能进行 CRUD 操作的 UserAction 类)。

```
<action path="/user"
type="org.springframework.web.struts.DelegatingActionProxy" name="userForm"
scope="request" parameter="method">
    <forward name="list" path="/userList.jsp"/>
    <forward name="edit" path="/userForm.jsp"/>
</action>
```

确保 web 目录下已经建好 userList.jsp 和 userForm.jsp 两个文件。暂时不必在文件中写入内容。

11. 作为插件的一部分，配置 Spring, 将/user bean 设置成 “UserManager” 。在 we/WEB-INF/action-servlet.xml 中添加以下定义。

```
<bean name="/user" class="org.appfuse.web.UserAction" singleton="false">
    <property name="userManager">
        <ref bean="userManager"/>
    </property>
</bean>
```

定义中，使用 singleton= false 。这样就为每个请求，新建一个 Action，减少线程安全的 Action 需求。不管是 Manager 还是 DAO 都有成员变量，可能不需要这些属性(默认 singleton= true )。

12. 在 message.properties 上配置资源绑定。

在 userAction 类中，在完成一些操作后，会显示 “成功” 或是 “错误” 页面，这些信息的键可以存放在这个应用的 ResourceBundle(或 messages.properties 文件中)。特别是：

- user.saved
- user.missing
- user.deleted

把这些键存入 web/WEB-INF/下的 messages.properties 文件中。例如：

user.saved=User has been saved successfully.

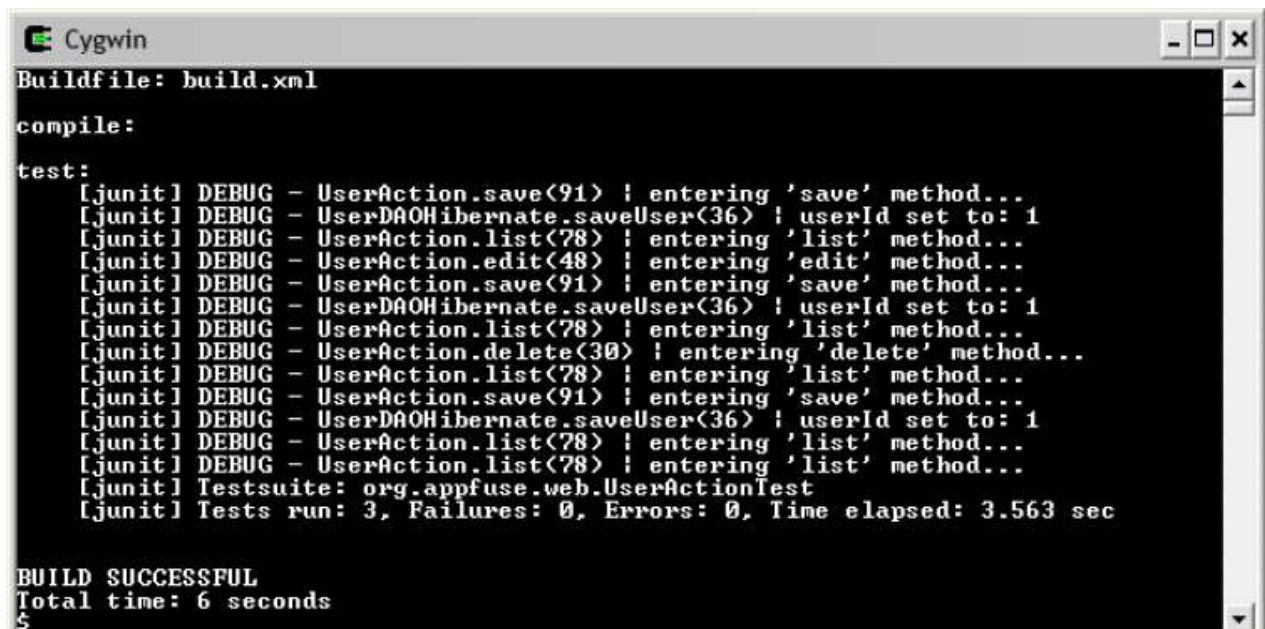
user.missing=No user found with this id.

user.deleted=User successfully deleted.

这个文件通过 struts-config.xml 中的 <message-resources> 元素进行加载。

<message-resources parameter="messages"/>

运行 ant test -Dtestcase=UserAction. 输出结果如下：



```
Buildfile: build.xml
compile:
test:
[junit] DEBUG - UserAction.save(91) ! entering 'save' method...
[junit] DEBUG - UserDaoHibernate.saveUser(36) ! userId set to: 1
[junit] DEBUG - UserAction.list(78) ! entering 'list' method...
[junit] DEBUG - UserAction.edit(48) ! entering 'edit' method...
[junit] DEBUG - UserAction.save(91) ! entering 'save' method...
[junit] DEBUG - UserDaoHibernate.saveUser(36) ! userId set to: 1
[junit] DEBUG - UserAction.list(78) ! entering 'list' method...
[junit] DEBUG - UserAction.delete(30) ! entering 'delete' method...
[junit] DEBUG - UserAction.list(78) ! entering 'list' method...
[junit] DEBUG - UserAction.save(91) ! entering 'save' method...
[junit] DEBUG - UserDaoHibernate.saveUser(36) ! userId set to: 1
[junit] DEBUG - UserAction.list(78) ! entering 'list' method...
[junit] DEBUG - UserAction.list(78) ! entering 'list' method...
[junit] Testsuite: org.appfuse.web.UserActionTest
[junit] Tests run: 3, Failures: 0, Errors: 0, Time elapsed: 3.563 sec

BUILD SUCCESSFUL
Total time: 6 seconds
$
```

## 填充 JSP 文件，这样可以通过浏览器来进行 CRUD 操作

1. 在你的 jsp 文件 (userFrom.jsp 和 userList.jsp) 中添加代码，这样它们可以表示 actions 的结果。如果还事先准备，在 web 目录下建一个文件 userList.jsp。添加一些代码你就可以看到数据库中所有的用户资料。在下面代码中，第一行包含(include)了一个文件 taglibs.jsp。这个文件包含了应用所有 JSP Tag Library 的声明。大部分是 Struts Tag, JSTL 和 SiteMesh(用来美化 JSP 页面)。

```
<%@ include file="/taglibs.jsp"%>
<title>MyUsers ~ User List</title>
<button onclick="location.href='user.do?method=edit'">Add User</button>
<table class="list">
<thead>
<tr>
<th>User Id</th>
<th>First Name</th>
<th>Last Name</th>
</tr>
</thead>
<tbody>
<c:forEach var="user" items="${users}" varStatus="status">
<c:choose>
<c:when test="${status.count % 2 == 0}">
<tr class="even">
</c:when>
<c:otherwise>
<tr class="odd">
</c:otherwise>
</c:choose>
<td><a href="user.do?method=edit&id=${user.id}">${user.id}</a></td>
<td>${user.firstName}</td>
<td>${user.lastName}</td>
</tr>
</c:forEach>
</tbody>
</table>
```



你可以有一行“标题头”(headings)(在<thead>中)。JSTL 的 <c:forEach>进行结果迭代, 显示所有的用户。

2. 向数据库添加一些数据, 你就会看到一些真实(actual)的用户(users)。你可以选择一种方法, 手工添加, 使用 ant browse, 或是在 build.xml 中添加如下的 target:

```
<target name="populate">
    <echo message="Loading sample data..."/>
    <sql driver="org.hsqldb.jdbcDriver" url="jdbc:hsqldb:db/appfuse"
userid="sa" password="">
        <classpath refid="classpath"/>
        INSERT INTO app_user (id, first_name, last_name) values (5, 'Julie',
'Raible');
        INSERT INTO app_user (id, first_name, last_name) values (6, 'Abbie',
'Raible');
    </sql>
</target>
```

警告:为了使内置的 HSQLDB 正常工作, 从能运行 ant 的目录下启动 tomcat。在 Unix/Linux 键入 \$CATALINA\_HOME/bin/startup.sh , 在 win 上 %CATALINA\_HOME%\bin\startup.bat 。

## 通过浏览器验证 JSP 的功能

1. 有了这个 JSP 文件和里面的样例数据，就可以通过浏览器来查看这个页面。运行 ant deploy reload, 转到地址 `http://localhost:8080/myusers/user.do?method=list`。出现以下画面：



2. 这个样例中，缺少国际化的页面标题头，和列标题头(column headings)。在 web/WEB-INF/classes 中 messages.properties 中加入一些键：

```
user.id=User Id
user.firstName=First Name
user.lastName=Last Name
```

修改过的国际化的标题头如下：

```
<thead>
<tr>
  <th><bean:message key= user.id /></th>
  <th><bean:message key= user.firstName /></th>
  <th><bean:message key= user.lastName /></th>
</tr>
</thead>
```

注意同样可以使用 JSTL 的 `<fmt:message key= ... >` 标签。如果想为表添加排序和分

布功能，可以使用 Display Tag (<http://displaytag.sf.net>)。下面是使用这个标签的一个样例：

```
<display:table name="users" pagesize="10" styleClass="list"
requestURI="user.do?method=list">
    <display:column property="id" paramId="id" paramProperty="id"
href="user.do?method=edit" sort="true"/>
    <display:column property="firstName" sort="true"/>
    <display:column property="lastName" sort="true"/>
</display:table>
```

请参考 display tag 文档中有关的列标题头国际化的部分。

3. 你已经建好了显示(list), 创建 form 就可以添加/编辑(add/edit)数据。如果事先没有准备，可以在 web 目录下新建一个 userForm.jsp 文件。向文件中添加以下代码：

```
<%@ include file="/taglibs.jsp"%>
<title>MyUsers ~ User Details</title>
<p>Please fill in user's information below:</p>
<html:form action="/user" focus="user.firstName">
    <input type="hidden" name="method" value="save"/>
    <html:hidden property="user.id"/>
    <table>
    <tr>
    <th><bean:message key="user.firstName"/>: </th>
    <td><html:text property="user.firstName"/></td>
    </tr>
    <tr>
    <th><bean:message key="user.lastName"/>: </th>
    <td><html:text property="user.lastName"/></td>
    </tr>
    <tr>
    <td></td>
    <td><html:submit styleClass="button">Save</html:submit>
    <c:if test="$ {not empty param.id}">
    <html:submit styleClass="button" onclick="this.form.method.value='delete'">
Delete</html:submit>
    </c:if>
    </td>
```

</table>

</html:form>

注意：如果你正在开发一个国际化的应用，把上面的信息和按钮标签替换成 `<bean:message>` 或是 `<fmt:message>` 标签。这是一个很好的练习。对于信息 `message`，建议把 `key` 名称写成 “`pageName.message`”（例如：`userForm.message`）的形式，按钮名字写成 “`button.name`”（例如 `button.save`）。

4. 运行 `ant deploy`，通过浏览器页面的 `user form` 来进行 `CRUD` 操作。

最后大部分 web 应用都需要验证。下一节中，配置 `struts validator`，要求用户的 `last name` 是必填的。

## 用 **commons Validator** 添加验证

为了在 Struts 中使用验证, 执行以下几步:

1. 在 struts-config.xml 中添加 ValidatorPlugin。
2. 创建 validation.xml, 指定 lastName 为必填字段。
3. 仅为 save() 方法设置验证(validation)。
4. 在 message.properties 中添加 validation errors。

### 在 **struts-config.xml** 中添加 **ValidatorPlugin**

配置 Validatorp plugins, 添加以下片断到 struts-config.xml (紧接着 Spring plugin):

```
<plug-in className="org.apache.struts.validator.ValidatorPlugIn">
    <set-property property="pathnames" value="/WEB-INF/validator-
rules.xml, /WEB-INF/validation.xml"/>
</plug-in>
```

从这里你可以看出, Validator 会查找 WEB-INF 下的两个文件 validator-ruls.xml 和 validation.xml。第一个文件, validator-rules.xml, 是一个标准文件, 作为 Struts 的一部分发布, 它定义了所有可用的验证器(validators), 功能和客户端的 javascript 类似。第二个文件, 包含针对每个 form 的验证规则。

### 创建 **validation.xml**, 指定 **lastName** 为必填字段

validation.xml 文件中包含很多 DTD 定义的标准元素。但你只需要如下所示的<form>和<field>, 更多信息请参阅 Validator 的文档。在 web/WEB-INF/validation.xml 中的 form-validation 标签之间添加 form-set 元素。

```
<formset>
    <form name="userForm">
        <field property="user.lastName" depends="required">
    </form>
</formset>
```

### 把 **DynaActionForm** 改为 **DynaValidatorForm**

把 struts-config.xml 中的 DynaActionForm 改为 DynaValidatorForm。

```
<form-bean name="userForm"
type="org.apache.struts.validator.DynaValidatorForm">
```

## 为 **save()** 方法设置验证 (**validation**)

使用 Struts DispatchAction 弊端是，验证会在映射层(mapping level)激活。为了在 list 和 edit 页面关闭验证。你必须单独建一个" validate=false" 的映射。例如，AppFuse 的 UserAction 有两个映射："/editUser" 和 "/listUser"。然而有一个更简单的方法，可以减少 xml，只是多了一些 java 代码。

1.在/user 映射中，添加 validate=false 。

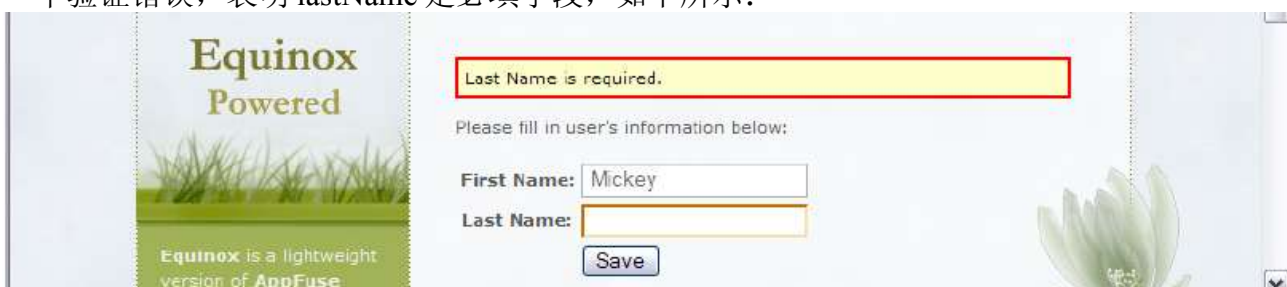
2.修改 UserAction 中的 save()方法，调用 form.validate()方法，如果发现错误，返回编辑页面。

```
if (log.isDebugEnabled()) {
log.debug("entering 'save' method...");
}
// run validation rules on this form
ActionMessages errors = form.validate(mapping, request);
if (!errors.isEmpty()) {
saveErrors(request, errors);
return mapping.findForward("edit");
}
DynaActionForm userForm = (DynaActionForm) form;
```

当 dispatchAction 运行时，与附带一个属性的两个映射相比，这样更加简洁。但用两个映射也有一些优点：

- 验证失败时，可以指定转向" input"属性。
- 在映射中可以添加"role"属性，可以指定谁有访问权限。例如，任何人都可以看到编辑(edit)页面，但只有管理员可以保存(save)。

运行 ant deploy 重新载入(reload)，尝试添加一个新用户，不要填写 lastName。你会看到一个验证错误，表明 lastName 是必填字段，如下所示：



Struts Validator 的另一种比较好的特性是客户端验证(client-side validation)。

4. 在 form 标签(web/userForm.jsp)中添加”onsubmit”属性, 在 form 末尾添加<html:javascript>。

```
<html:form action="/user" focus="user.firstName" onsubmit="return
validateUserForm(this)">
```

```
...
```

```
</html:form>
```

```
<html:javascript formName="userForm"/>
```

现在如果运行 ant deploy, 试图保存一个 lastname 为空的用户, 会弹出一个 JavaScript 提示: “Last Name is required”。这里有一个问题, 这个带 JavaScript 的 form 把 validator 的 JavaScript 功能都载入了页面。再好的方法是, 从外部文件导入 Javascript。参见第 5 章。

恭喜你! 你已经开发一个 web 应用, 它包含数据库交互, 验证实现, 成功信息和错误信息的显示。第 4 章, 将会把这个转向 Spring 框架。第 5 章中, 会添加异常处理, 文件上传, 邮件发送等特性。第 6 章会看一下 JSP 的替代品, 在第 7 章, 会添加 DAO 的不同实现, 包括 iBATIS, JDO 和 Spring 的 JDBC。