

## 用Proxool来配置连接池

很想采用opensource的项目proxool来实现连接池。可它可真要我伤心的呀，测试了很久，也没有成功。不是说没有找到配置文件，就是说没有合适的驱动。真是晕，真是奇怪。我明明按照要求把配置文件和一直用的jdbc驱动程序放在了classpath下，可就是出现了问题，只能说明一点，那就是我还是没有明白它的原理。最后，找了很多的资料，终于测试成功了一种。

下面说明一下，在web application下用proxool来配置pool：

1) 下载proxool类包

去<http://proxool.sf.net/>下最新的proxool。

2) 把proxool类包，jdbc驱动程序放到WEB-INF/LIB

下

3) 添加连接信息的配置文件,并放入WEB-INF/LIB下，如protest.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!-- the proxool configuration can be embedded within your own application's.
Anything outside the "proxool" tag is ignored. -->
<something-else-entirely>
<proxool>
  <alias>dglabour</alias>
  <driver-url>jdbc:microsoft:sqlserver://192.168.1.35:1433;DatabaseName=dg_labour;SelectMethod=cursor;</driver-url>
  <driver-class>com.microsoft.jdbc.sqlserver.SQLServerDriver</driver-class>
  <driver-properties>
    <property name="user" value="sa"/>
    <property name="password" value="123"/>
  </driver-properties>
  <maximum-connection-count>10</maximum-connection-count>
</proxool>
</something-else-entirely>
```

```
n-count>
  <house-keeping-test-sql>select CURRENT_DATE</hou
se-keeping-test-sql>
</proxool>
</something-else-entirely>
```

4)在web.xml文件中添加如下代码

```
<servlet>
  <servlet-name>ServletConfigurator</servlet-name>
  <servlet-class>org.logicalcobwebs.proxool.configur
ation.ServletConfigurator</servlet-class>
  <init-param>
    <param-name>xmlFile</param-name>
    <param-value>WEB-INF/protest.xml</param-va
lue>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>

<servlet>
  <servlet-name>Admin</servlet-name>
  <servlet-class>
org.logicalcobwebs.proxool.admin.servlet.AdminSer
vlet
  </servlet-class>
</servlet>

<servlet-mapping>
  <servlet-name>Admin</servlet-name>
  <url-pattern>/admin</url-pattern>
</servlet-mapping>
```

5) 在程序中用如下代码来调用，

```
connection = DriverManager.getConnection("proxool.dgla
bour");
```

6)测试页面

<http://localhost:8080/shipment/admin> shipment是  
web applicate名字

7)相关参考资料请看下载包中的文档，和<http://sentom.net/list.asp?id=33>。 😁

😊 [Eclipse快速上手指南](#) henrydeng 发表于 2005-10-3 15:37:15

## Eclipse快速上手指南(1)

副标题：

作者：asklxf 文章来源：本站原创 点击数：6734 更新时间：2004-11-16

Eclipse是一款非常优秀的开源IDE，非常适合Java开发，由于支持插件技术，受到了越来越多的开发者的欢迎。最新的Eclipse 3.0不但界面作了很大的增强，而且增加了代码折叠等众多优秀功能，速度也有明显的提升。配合众多令人眼花缭乱的插件，完全可以满足从企业级Java应用到手机终端Java游戏的开发。本文将带您手把手步入Eclipse的广阔天地，详细介绍在Eclipse下如何开发普通Java程序，Web应用，J2EE应用，手机Java程序，以及如何进行单元测试，重构，配置CVS等详细内容。

我的开发环境是JDK1.4.2+Eclipse3.0+Windows XP SP2，如果你在其他平台上遇到任何问题，欢迎来信交流。

### 1. 安装JDK1.4

Eclipse是一个基于Java平台的开发环境，它本身也要运行在Java虚拟机上，还要使用JDK的编译器，因此我们必须首先安装JDK。JDK1.4是目前最稳定的版本，同时也是Eclipse运行的必须条件。先从SUN的官方站点<http://java.sun.com>下载JDK1.4 Windows版，目前最新的是1.4.2\_06，然后运行j2sdk-1\_4\_2\_06-windows-i586-p.exe安装，你可以自行设定安装目录，我把它安装到D:\software\j2sdk1.4目录下。

接下来要配置环境变量，以便Java程序能找到已安装的JDK和其他配置信息。右键点击“我的电脑”，选择“属性”，在弹出的对话框中选择“高级”，“环境变量”，就可以看到环境变量对话框：

[500\)this.width=500" align=baseline border=0>](#)

上面是用户变量，只对当前用户有效，下面是系统变量，对所有用户都有效。如果你希望所有用户都能使用，就在系统变量下点击“新建”，填入：

[500\)this.width=500" align=baseline border=0>](#)

JAVA\_HOME是JDK的安装目录，许多依赖JDK的开发环境都靠它来定位JDK，所以必须保证正确无误。

下一步，找到系统变量Path，点击“编辑”，在最后添上JDK的可执行文件的所在目录，即%JAVA\_HOME%\bin，我的对应目录便是D:\software\j2sdk1.4\bin，附加到Path中即可，注意要以分号“;”隔开：

[500\)this.width=500" align=baseline border=0>](#)

注意：如果系统安装了多个Java虚拟机（比如安装了Oracle 9i就有自带的JDK1.3），必须把JDK1.4的路径放在其他JVM的前面，否则Eclipse启动将报错。

最后一个系统变量是CLASSPATH，Java虚拟机会根据CLASSPATH的设定来搜索class文件所在目录，但这不是必需的，可以在运行Java程序时指定CLASSPATH，比如在Eclipse中运行写好的Java程序时，它会自动设定CLASSPATH，但是为了在控制台能方便地运行Java程序，我建议最好还是设置一个CLASSPATH，把它的值设为“.”，注意是一个点“.”代表当前目录。用惯了Windows的用户可能会以为Java虚拟机在搜索时会搜索当前目录，其实不会，这是UNIX中的习惯，出于安全考虑。许多初学Java的朋友兴匆匆地照着书上写好了Hello，world程序，一运行却弹出java.lang.NoClassDefFoundError，其实就是没有设置好CLASSPATH，只要添加一个当前目录“.”就可以了。

## 2. 安装Eclipse 3.0

配置好JDK后，下一步便是安装Eclipse 3.0，可以从Eclipse的官方站点<http://www.eclipse.org>上下载，你会看到如下版本：

- Eclipse SDK
- RCP Runtime Binary
  - RCP SDK
- Platform Runtime Binary
  - Platform SDK
- JDT Runtime Binary

Eclipse SDK包括了Eclipse开发环境，Java开发环境，Plug-in开发环境，所有源代码和文档，如果你需要所有的功能，可以下载这个版本。

如果你和我一样，只是用Eclipse开发Java应用，而不是开发Eclipse插件或者研究Eclipse代码，那么下载一个**Platform Runtime Binary**再加上**JDT Runtime Binary**是最好的选择。

下载eclipse-platform-3.0-win32.zip和eclipse-JDT-3.0.zip后，将它们解压到同一个目录，勿需安装，直接找到目录下的eclipse.exe运行，出现启动画面：

[500\)this.width=500" align=baseline border=0>](#)

稍等片刻，Eclipse界面就出来了。

如果遇到错误，启动失败，可以检查Eclipse目录下的log文件，我曾经遇到过XmlParser异常，仔细检查发现原来Path中还有一个Oracle的Java1.3版本的虚拟机，将它从Path中去掉后Eclipse启动正常。

## 3. 第一个Java程序

运行Eclipse，选择菜单“File”，“New”，“Project”，新建一个Java Project，我把它命名为HelloWorld，然后新建一个Java Class：

[500\)this.width=500" align=baseline border=0>](#)

我把它命名为HelloWorld，并且填上Package为example，钩上“public static void main(String[] args)”，点击“Finish”，Eclipse自动生成了代码框架，我们只需在main方法中填入：

[500\)this.width=500" align=baseline border=0>](#)

默认设置下，Eclipse会自动在后台编译，我们只需保存，然后选择“Run”，“Run As”，“Java Application”，即可在Eclipse的控制台看到输出。

要调试Java程序也非常简单，Run菜单里包含了标准的调试命令，可以非常方便地在IDE环境下调试应用程序。

1.4版本支持：

选择菜单“Window”，“Preferences”，在对话框中找到“Java”，“Compiler”，“Compliance and Classfiles”，将编译选项改成1.4，就可以使用JDK1.4版的assert（断言）语法，使得测试更加方便：

[500\)this.width=500" align=baseline border=0>](#)

Eclipse快速上手指南(2)

副标题：

作者：asklxf 文章来源：本站原创 点击数：4080 更新时间：2004-11-22

#### 4. 在Eclipse中使用JUnit

测试对于保证软件开发质量有着非常重要的作用，单元测试更是必不可少，JUnit是一个非常强大的单元测试包，可以对一个/多个类的单个/多个方法测试，还可以将不同的TestCase组合成TestSuite，使测试任务自动化。Eclipse同样集成了JUnit，可以非常方便地编写TestCase。

我们创建一个Java工程，添加一个example.Hello类，首先我们给Hello类添加一个abs()方法，作用是返回绝对值：

[500\)this.width=500" align=baseline border=0>](#)

下一步，我们准备对这个方法进行测试，确保功能正常。选中Hello.java，右键点击，选择New->JUnit Test Case：

[500\)this.width=500" align=baseline border=0>](#)

Eclipse会询问是否添加junit.jar包，确定后新建一个HelloTest类，用来测试Hello类。

[500\)this.width=500" align=baseline border=0>](#)

选中setUp()和tearDown()，然后点击“Next”：

[500\)this.width=500" align=baseline border=0>](#)

选择要测试的方法，我们选中abs(int)方法，完成后在HelloTest.java中输入：

[500\)this.width=500" align=baseline border=0>](#)

JUnit会以以下顺序执行测试：（大致的代码）

```
try {  
    HelloTest test = new HelloTest(); // 建立测试类实例  
    test.setUp(); // 初始化测试环境  
    test.testAbs(); // 测试某个方法  
    test.tearDown(); // 清理资源  
}  
catch...
```

setUp()是建立测试环境，这里创建一个Hello类的实例；tearDown()用于清理资源，如释放打开的文件等等。以test开头的方法被认为是测试方法，JUnit会依次执行testXxx()方法。在testAbs()方法中，我们对abs()的测试分别选择正数，负数和0，如果方法返回值与期待结果相同，则assertEquals不会产生异常。

如果有多个testXxx方法，JUnit会创建多个XxxTest实例，每次运行一个testXxx方法，setUp()和tearDown()会在testXxx前后被调用，因此，不要在一个testA()中依赖testB()。

直接运行Run->Run As->JUnit Test，就可以看到JUnit测试结果：

[500\)this.width=500" align=baseline border=0>](#)

绿色表示测试通过，只要有1个测试未通过，就会显示红色并列出不通过测试的方法。可以试图改变abs()的代码，故意返回错误的结果（比如return n+1;），然后再运行JUnit就会报告错误。

如果没有JUnit面板，选择Window->Show View->Other，打开JUnit的Vie

w :

[500\)this.width=500" align=baseline border=0>](#)

JUnit通过单元测试，能在开发阶段就找出许多Bug，并且，多个Test Case可以组合成Test Suite，让整个测试自动完成，尤其适合于XP方法。每增加一个小的新功能或者对代码进行了小的修改，就立刻运行一遍Test Suite，确保新增和修改的代码不会破坏原有的功能，大大增强软件的可维护性，避免代码逐渐“腐烂”。

### Eclipse快速上手指南(3)

副标题：

作者：asklxf 文章来源：本站原创 点击数：3668 更新时间：2004-11-22

## 5. 在Eclipse中使用Ant

Ant是Java平台下非常棒的批处理命令执行程序，能非常方便地自动完成编译，测试，打包，部署等一系列任务，大大提高开发效率。如果你现在还没有开始使用Ant，那就要赶快开始学习使用，使自己的开发水平上一个新台阶。

Eclipse中已经集成了Ant，我们可以直接在Eclipse中运行Ant。

以前面建立的Hello工程为例，创建以下目录结构：

[500\)this.width=500" align=baseline border=0>](#)

新建一个build.xml，放在工程根目录下。build.xml定义了Ant要执行的批处理命令。虽然Ant也可以使用其它文件名，但是遵循标准能更使开发更规范，同时易于与别人交流。

通常，src存放Java源文件，classes存放编译后的class文件，lib存放编译和运行用到的所有jar文件，web存放JSP等web文件，dist存放打包后的jar文件，doc存放API文档。

然后在根目录下创建build.xml文件，输入以下内容：

```

    <?xml version="1.0"?>
    <project name="Hello world" default="doc">

        <!-- properies -->
        <property name="src.dir" value="src" />
        <property name="report.dir" value="report" />
        <property name="classes.dir" value="classes" />
        <property name="lib.dir" value="lib" />
        <property name="dist.dir" value="dist" />
        <property name="doc.dir" value="doc"/>

        <!-- 定义classpath -->
        <path id="master-classpath">
            <fileset file="{lib.dir}/*.jar" />
            <pathelement path="{classes.dir}"/>
        </path>

        <!-- 初始化任务 -->
        <target name="init">
            </target>

        <!-- 编译 -->
        <target name="compile" depends="init" description="compile the source
            files">
            <mkdir dir="{classes.dir}"/>
            <javac srcdir="{src.dir}" destdir="{classes.dir}" target="1.4">
                <classpath refid="master-classpath"/>
            </javac>
        </target>

        <!-- 测试 -->
        <target name="test" depends="compile" description="run junit test">
            <mkdir dir="{report.dir}"/>
            <junit printsummary="on"
                haltonfailure="false"
                failureproperty="tests.failed"
                showoutput="true">
                <classpath refid="master-classpath" />
                <formatter type="plain"/>
            </junit>
        </target>
    </project>

```

```

        <batchtest todir="\${report.dir}">
            <fileset dir="\${classes.dir}">
                <include name="**/*Test.*"/>
            </fileset>
        </batchtest>
    </junit>
    <fail if="tests.failed">
        *****
        ***
        **** One or more tests failed! Check the output ... ****
        *****
        ***
    </fail>
</target>

    <!-- 打包成jar -->
<target name="pack" depends="test" description="make .jar file">
    <mkdir dir="\${dist.dir}" />
    <jar destfile="\${dist.dir}/hello.jar" basedir="\${classes.dir}">
        <exclude name="**/*Test.*" />
        <exclude name="**/Test*.*" />
    </jar>
</target>

    <!-- 输出api文档 -->
<target name="doc" depends="pack" description="create api doc">
    <mkdir dir="\${doc.dir}" />
    <javadoc destdir="\${doc.dir}"
        author="true"
        version="true"
        use="true"
        windowtitle="Test API">
        <packageset dir="\${src.dir}" defaultexcludes="yes">
            <include name="example/**" />
        </packageset>
        <doctitle><![CDATA[<h1>Hello, test</h1>]]></doctitle>
        <bottom><![CDATA[<i>All Rights Reserved.</i>]]></bottom>
        <tag name="todo" scope="all" description="To do:" />
    </javadoc>

```

```
</target>
</project>
```

以上xml依次定义了init（初始化），compile（编译），test（测试），doc（生成文档），pack（打包）任务，可以作为模板。

选中Hello工程，然后选择“Project”，“Properties”，“Builders”，“New...”  
，选择“Ant Build”：

[500\)this.width=500" align=baseline border=0>](#)

填入Name：Ant\_Builder；Buildfile：build.xml；Base Directory：\${workspace\_loc:/Hello}（按“Browse Workspace”选择工程根目录），由于用到了junit.jar包，搜索Eclipse目录，找到junit.jar，把它复制到Hello/lib目录下，并添加到Ant的Classpath中：

[500\)this.width=500" align=baseline border=0>](#)

然后在Builder面板中钩上Ant\_Build，去掉Java Builder：

[500\)this.width=500" align=baseline border=0>](#)

再次编译，即可在控制台看到Ant的输出：

```
Buildfile: F:\eclipse-projects\Hello\build.xml

init:

compile:
[mkdir] Created dir: F:\eclipse-projects\Hello\classes
[javac] Compiling 2 source files to F:\eclipse-projects\Hello\classes

test:
[mkdir] Created dir: F:\eclipse-projects\Hello\report
[junit] Running example.HelloTest
[junit] Tests run: 1, Failures: 0, Errors: 0, Time elapsed: 0.02 sec

pack:
[mkdir] Created dir: F:\eclipse-projects\Hello\dist
[jar] Building jar: F:\eclipse-projects\Hello\dist\hello.jar
```

```
doc:
[mkdir] Created dir: F:\eclipse-projects\Hello\doc
[javadoc] Generating Javadoc
[javadoc] Javadoc execution
[javadoc] Loading source files for package example...
[javadoc] Constructing Javadoc information...
[javadoc] Standard Doclet version 1.4.2_04
[javadoc] Building tree for all the packages and classes...
[javadoc] Building index for all the packages and classes...
[javadoc] Building index for all classes...
[javadoc] Generating F:\eclipse-projects\Hello\doc\stylesheet.css...
[javadoc] Note: Custom tags that could override future standard tags:
@todo. To avoid potential overrides, use at least one period character (.) in
custom tag names.
[javadoc] Note: Custom tags that were not seen: @todo
BUILD SUCCESSFUL
Total time: 11 seconds
```

Ant依次执行初始化，编译，测试，打包，生成API文档一系列任务，极大地提高了开发效率。将来开发J2EE项目时，还可加入部署等任务。并且，即使脱离了Eclipse环境，只要正确安装了Ant，配置好环境变量ANT\_HOME=<Ant解压目录>，Path=...;%ANT\_HOME%\bin，在命令行提示符下切换到Hello目录，简单地键入ant即可。

# Eclipse插件的安装与配置

[ 2005-9-12 ]

安装时有些小问题值得注意:

- 1、下载时注意Eclipse和Lomboz版本号很重要

一般来说相同版本号的才比较兼容。

即：2.0的Eclipse对应插件就装2.0的Lomboz；Eclipse3.01就对应装插件lomboz.301，这样不容易出错，下载时最好认真看清楚网站上的说明，否则有时安装总出现莫名其妙的问题，浪费时间。

- 2、安装Lomboz需要gef和emf插件的支持，所以光下载安装 Eclipse3.01,和lomboz.301.zip还不够，必须下载以下四个插件：

Eclipse3.01

lomboz.301.zip

GEF-runtime-3.0.1.zip

emf-sdo-runtime-2.0.0.zip

这一点很容易被忽视，结果安装时有时就会碰到一个问题：

按照步骤安装好后，启动Eclipse，结果【window】--->【Preference】左侧菜单上跟本找不到lomboz一项。

而在【Help】--

>【About Eclipse Platform】--->【Plugin Details】里面查看，却又明明显示lomboz插件已都安装好了。这个问题折腾了我半天，网上一些中文安装教程都没明确说清楚，其实要安装好lomboz，gef和emf这两个插件也是需要装的。

- 3、有时启动Eclipse未加载插件，解决方法很多，总结一下：

a、删除整个目录/eclipse/configuration/org.eclipse.update/，  
重启Eclipse

b、在启动Eclipse时带上 -clean参数  
如：d:\eclipse\eclipse.exe -clean

c、如果Eclipse启动找不到插件了，解决办法：在/configuration/config.ini文件中加入一行

osgi.checkConfiguration=true

这样它会寻找并安装插件，找到后可以把那行再注释掉，这样以后每次启动就不会因寻找插件而显得慢了。

4、为了便于管理众多插件，建议用links方式安装各种eclipse插件

Eclipse基础 - - 使用links方式安装Eclipse  
插件

eclipse想必大家都很熟悉了，一般来说，eclipse插件都是安装在plugins目录下。不过这样一来，当安装了许多插件之后，eclipse变的很大，最主要的是不便于更新和管理众多插件。用links方式安装eclipse插件，可以解决这个问题。

当前配置XP SP1，eclipse3.0.1

现在假设我的eclipse安装目录是D:\eclipse，待安装插件目录是D:\plug-in，我将要安装LanguagePackFeature（语言包）、emf-sdo-xsd-SDK、GEF-SDK、Lomboz这四个插件。

先把这四个插件程序全部放在D:\plug-in目录里，分别解压。如Lomboz3.0.1.zip解压成Lomboz3.0.1目录，这个目录包含一个plugins目录，要先在Lomboz3.0.1目录中新

建一个子目录eclipse，然后把plugins目录移动到刚建立的eclipse目录中，即目录结构要是这样的：D:\plugin\Lomboz3.0.1\eclipse\plugins

Eclipse 将会到指定的目录下去查找 eclipse\features 目录和eclipse\plugins 目录，看是否有合法的功能部件和（或）插件。也就是说，目标目录必须包含一个\eclipse 目录。如果找到，附加的功能部件和插件在运行期配置是将是可用的，如果链接文件是在工作区创建之后添加的，附加的功能部件和插件会作为新的配置变更来处理。

其它压缩文件解压后若已经包含 eclipse\plugins目录，则不需要建立eclipse 目录。

然后在 eclipse安装目录D:\eclipse目录中建立一个子目录links，在links目录中建立一个link文件，比如 LanguagePackFeature.link，改文件内容为 path=D:/plugin/LanguagePackFeature 即这个link文件要对应一个刚解压后的插件目录。

说明：

1. 插件可以分别安装在多个自定义的目录中。
2. 一个自定义目录可以安装多个插件。
3. link文件的文件名及扩展名可以取任意名称，比如ddd.txt，myplugin都可以。

4. link文件中path=插件目录的path路径  
分隔要用\\或是/
5. 在links目录也可以有多个link文件，每个link文件中的path参数都将生效。
6. 插件目录可以使用相对路径。
7. 可以在links目录中建立一个子目录，转移暂时不用的插件到此子目录中，加快eclipse启动。
8. 如果安装后看不到插件，把eclipse目录下的configuration目录删除，重启即可。

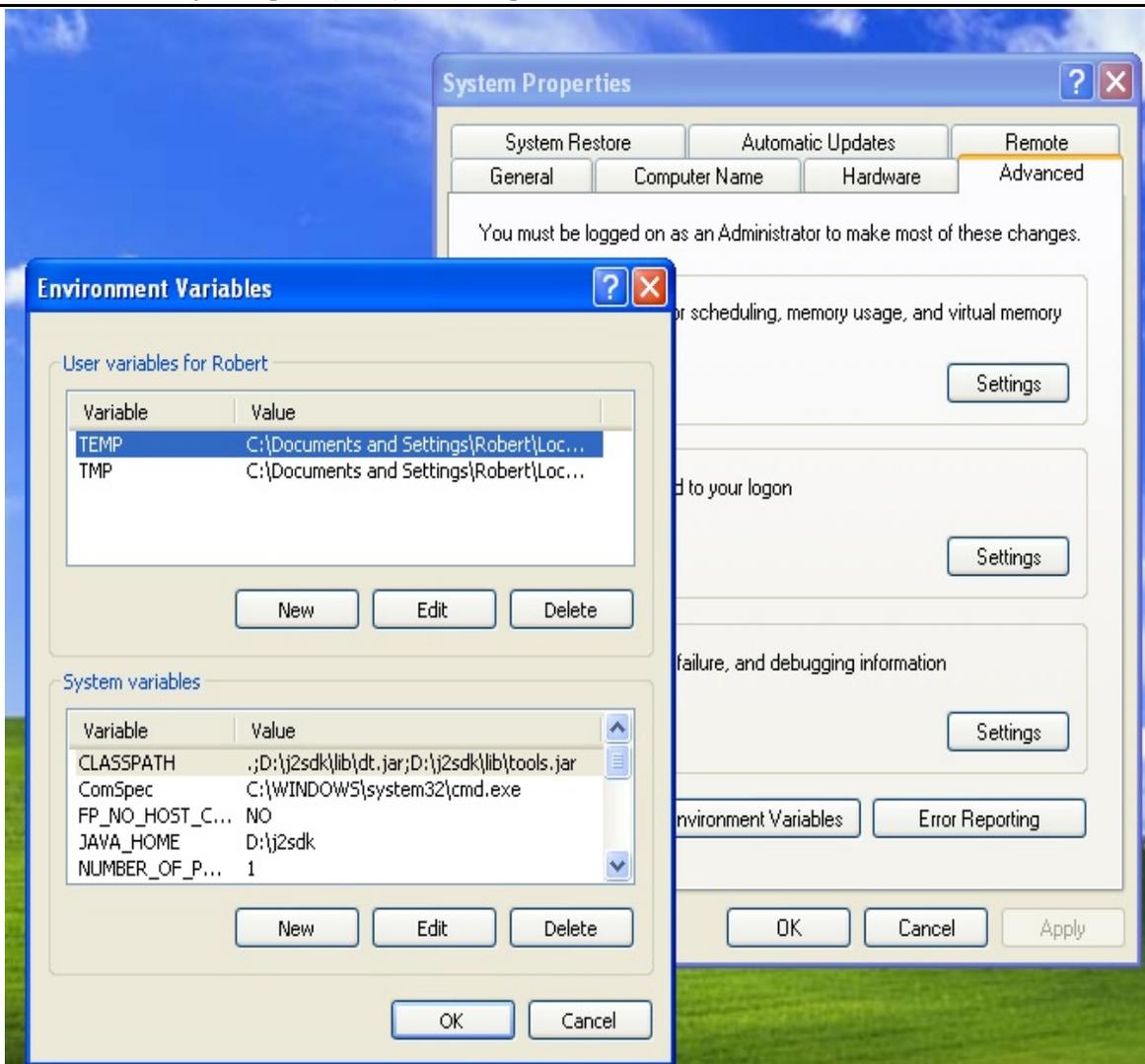
## 用Eclipse3.1 + Myeclipse4.0 + Tomcat5.0 + j2sdk1.5 搭建J2EE开发环境

(by [Robert Hu](#))

一、安装j2sdk1.5，设定环境变量。

我的安装目录是 **D:\j2sdk**。所以设定环境变量如下：

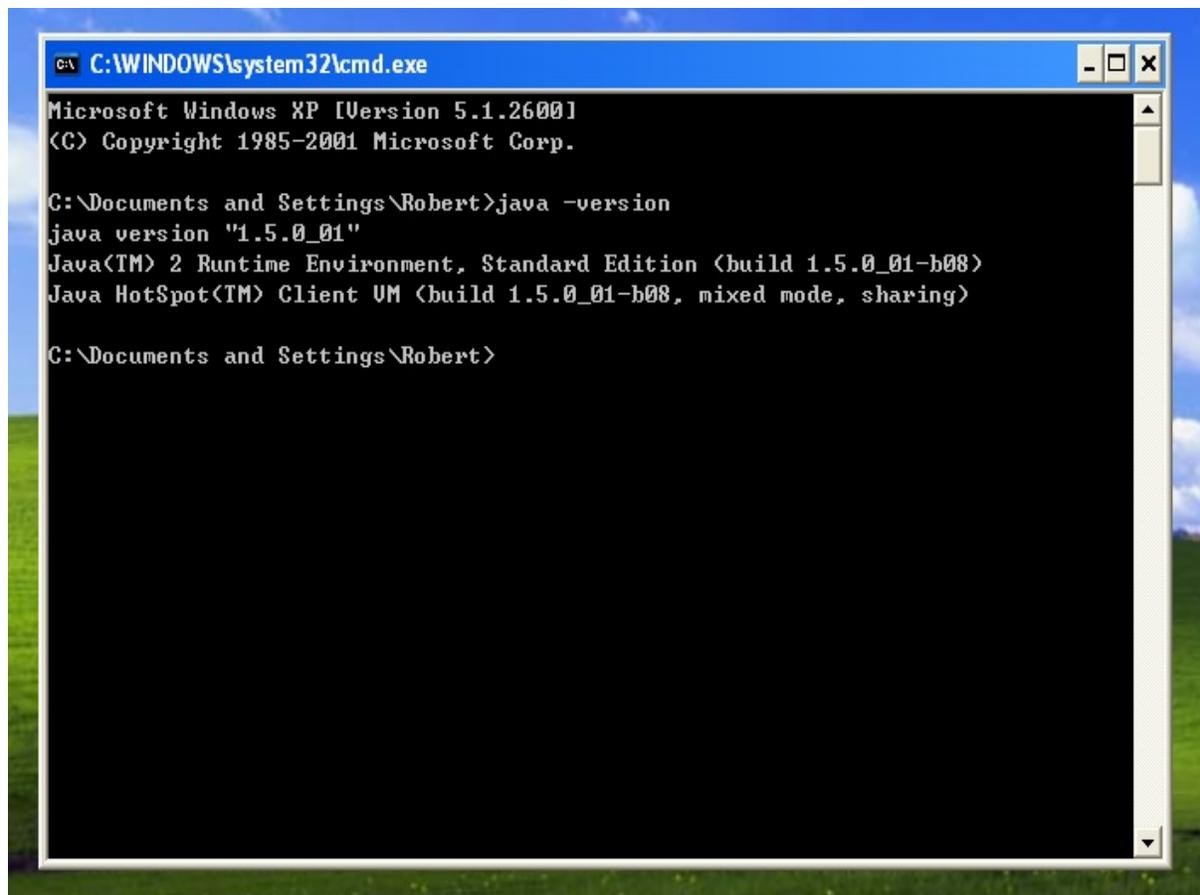
操作：My Computer(右键) ----> Properties ----> Advanced ----> Environment Variables：



设定环境变量为：

1. (新建)JAVA\_HOME: D:\j2sdk
2. (新建)CLASSPATH: .;D:\j2sdk\lib\dt.jar;D:\j2sdk\lib\tools.jar (注意：点号不能省略，表示当前目录)
3. 编辑PATH的变量值，在后面加上 ;%JAVA\_HOME%\bin (注意：要有分号隔开)

到这里，j2sdk安装完毕，用命令简单测试一下: java -version



```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\Robert>java -version
java version "1.5.0_01"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0_01-b08)
Java HotSpot(TM) Client VM (build 1.5.0_01-b08, mixed mode, sharing)

C:\Documents and Settings\Robert>
```

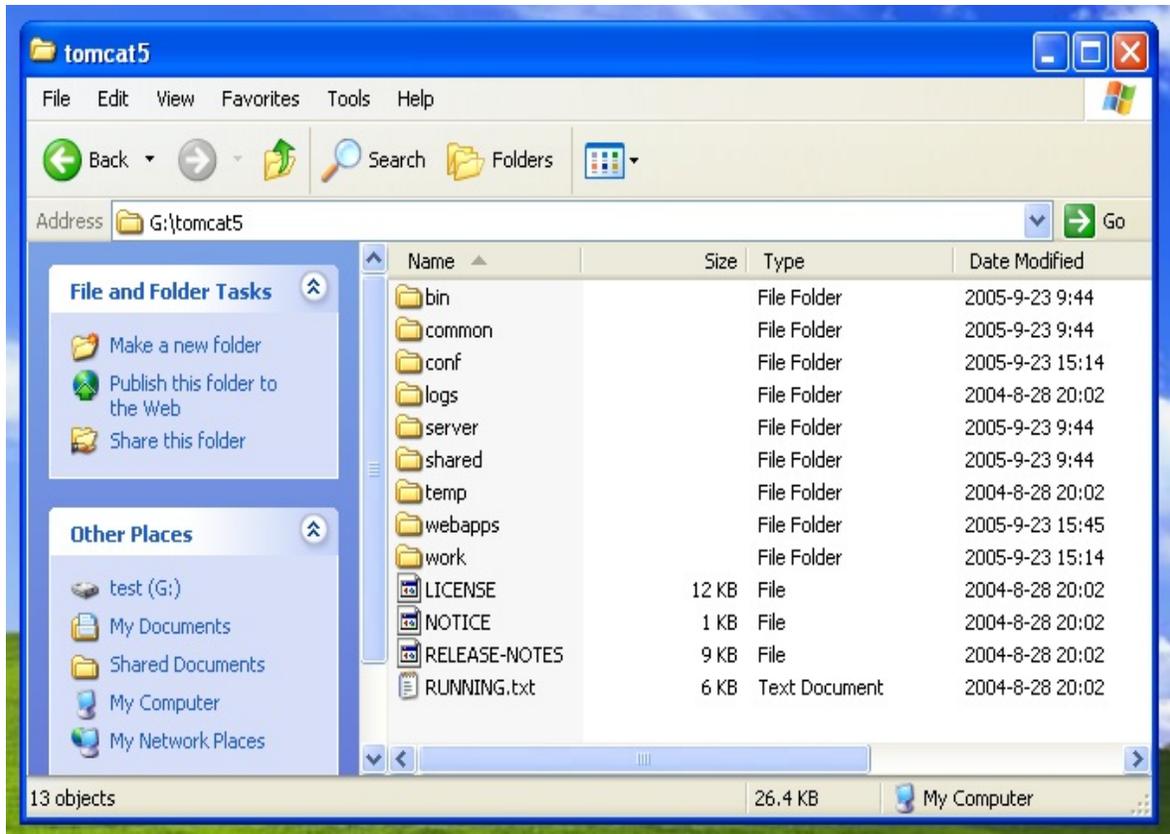
---

## 二、安装Tomcat5.0

去下载一个Tomcat的zip版，我用的是 jakarta-tomcat-5.0.28.zip ，不下安装版的目的是可以使用多个Tomcat。

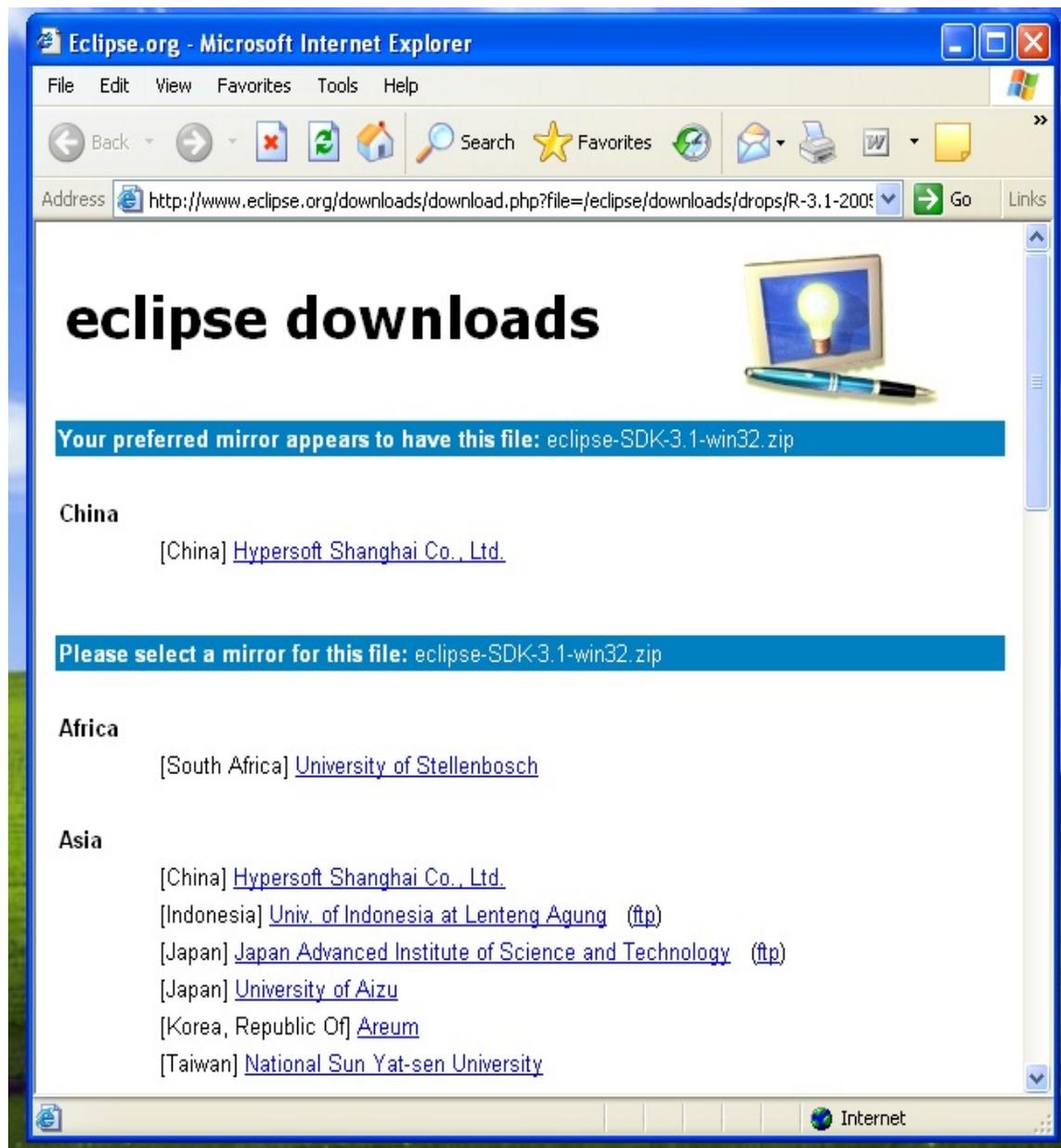
直接解压缩到 G 盘：

---



### 三、安装Eclipse3.1

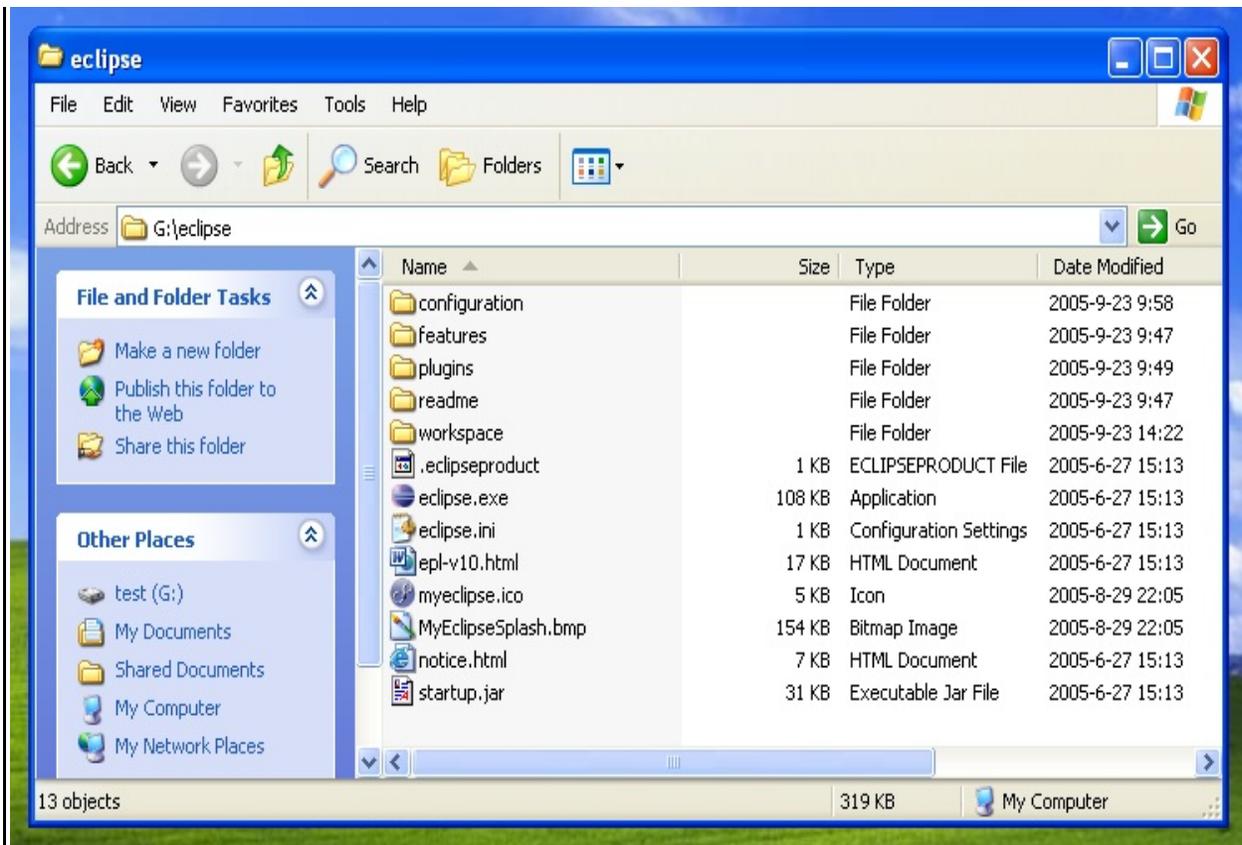
下载地址：<http://www.eclipse.org/downloads/index.php> 找到下面的页面：



找个亚洲的服务器下载,速度会快一点。在flashget中如下：



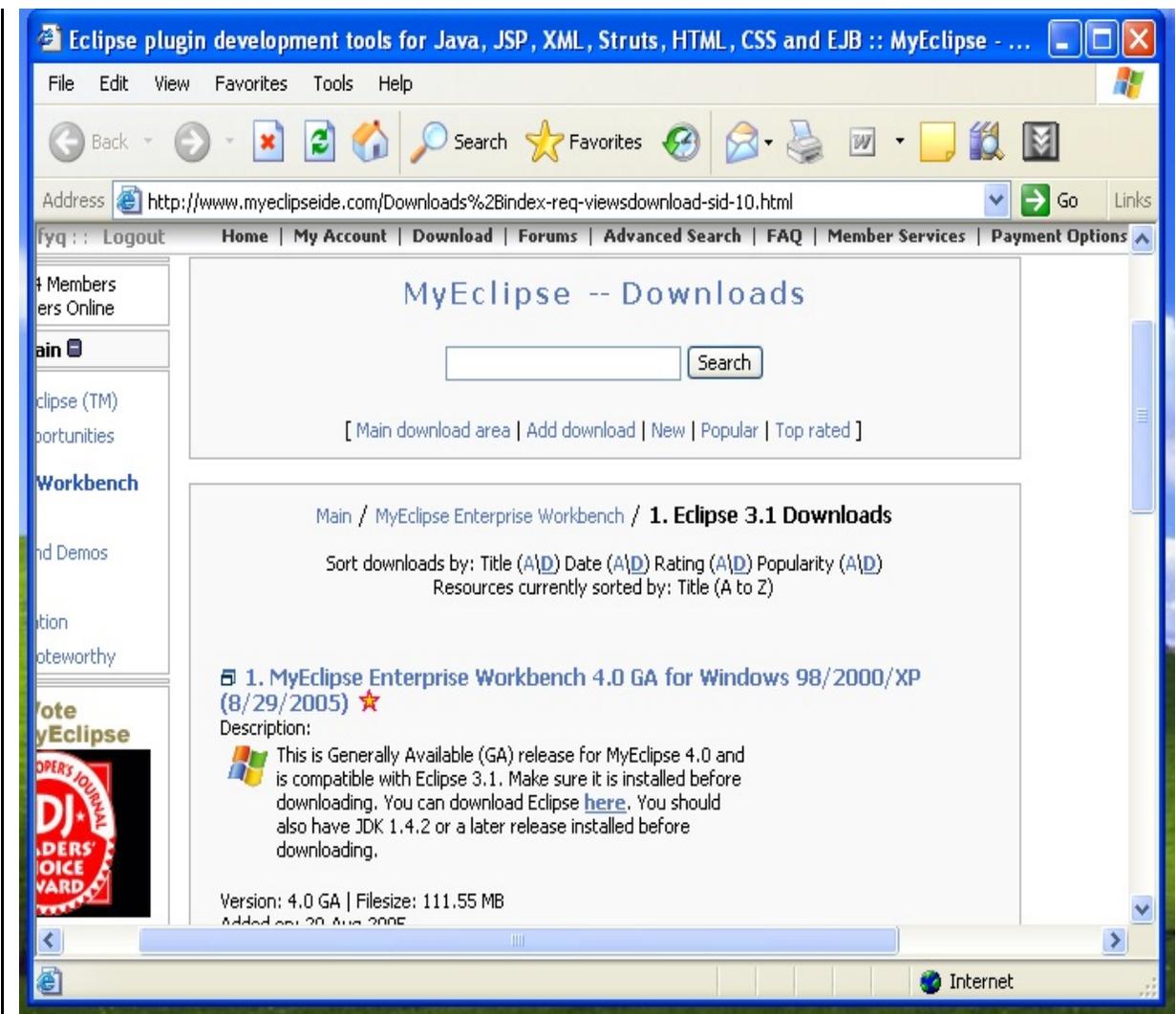
直接解压缩就可以用了，我是解压缩在 G 盘：



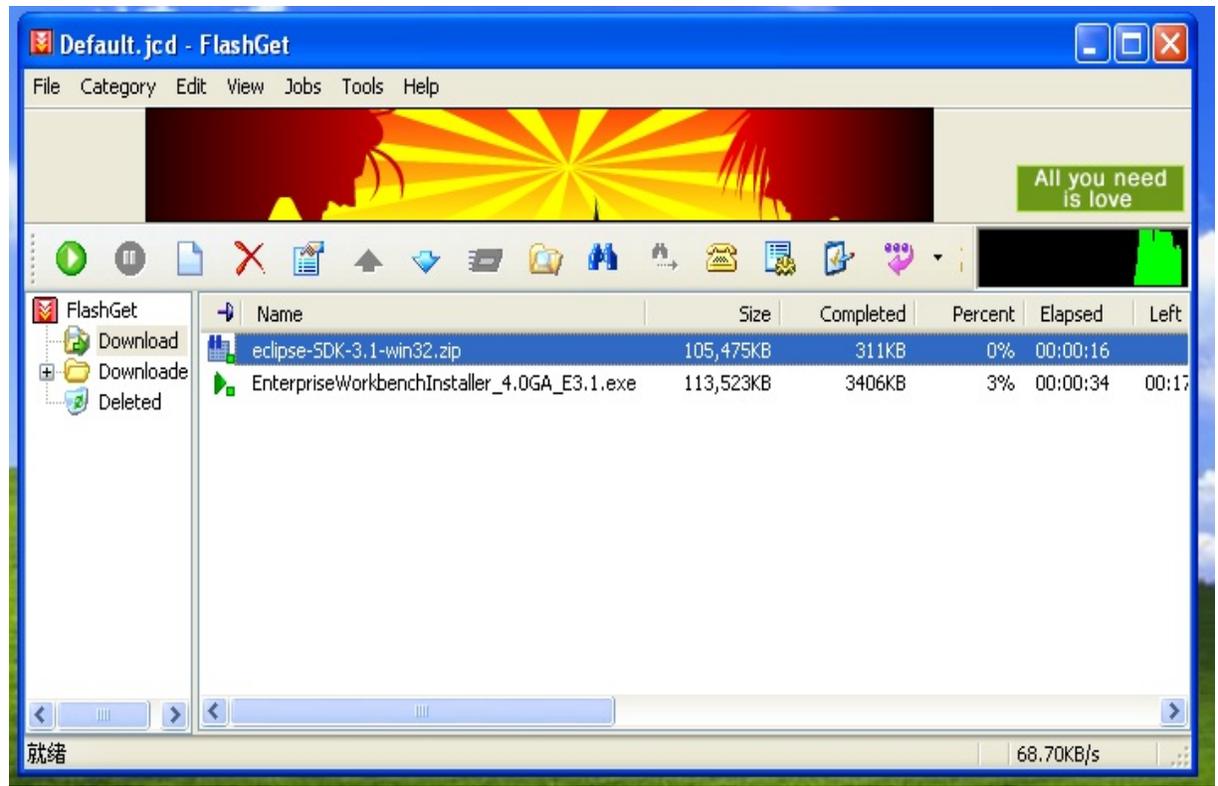
到这里，Eclipse3.1 安装完毕。

四、安装 Myeclipse4.0 (myeclipse不是免费的，所以我们先下载一个试用版，再去网上下载一个注册机)

下载版本：MyEclipse Enterprise Workbench 4.0 GA for Windows 98/2000/XP (8/29/2005) 如下图：

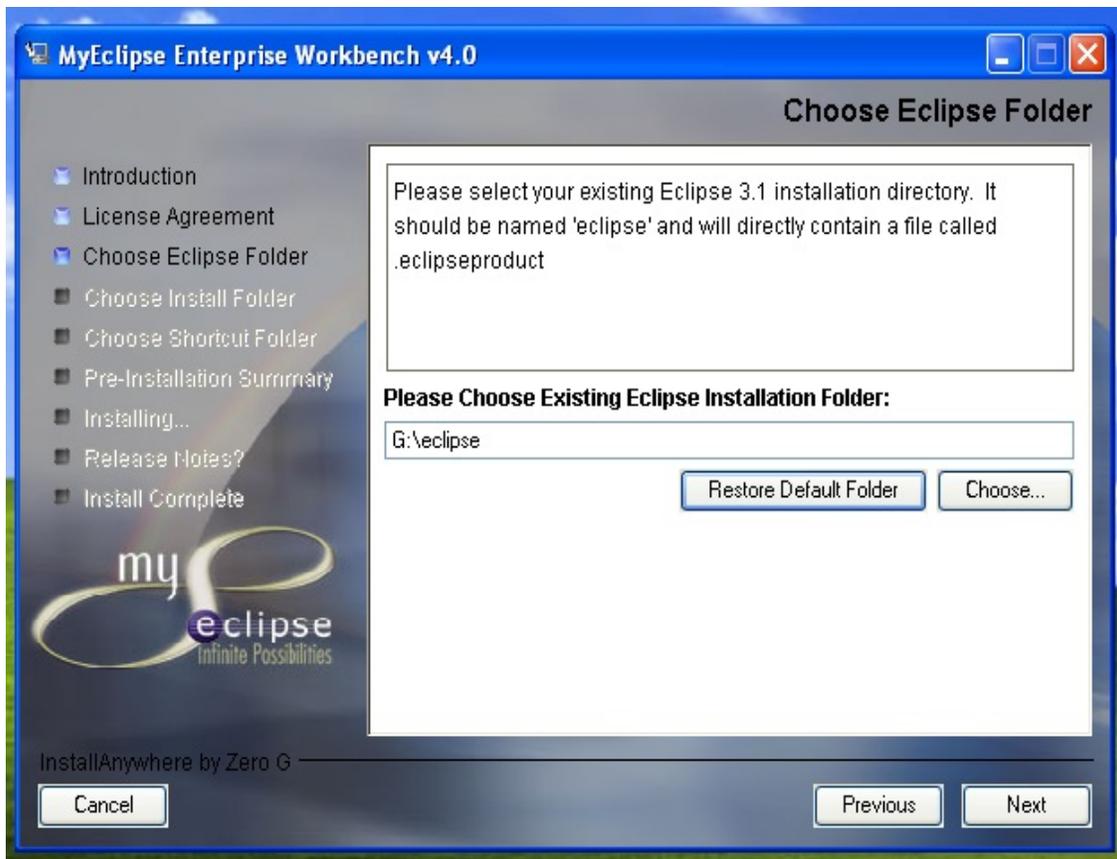


在flashget下载中显示如下：

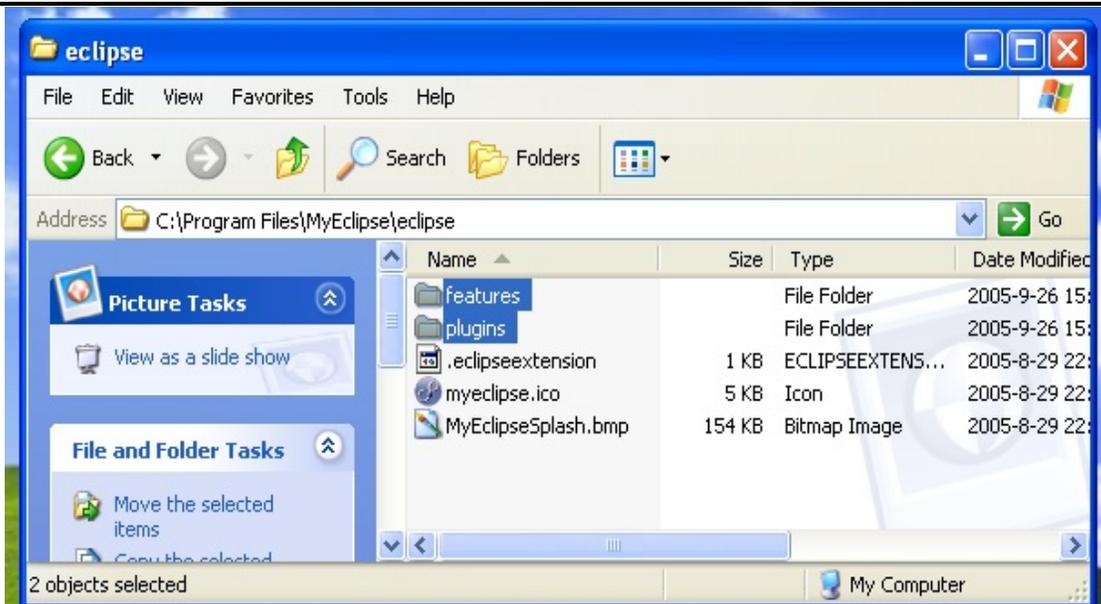


下载完成后，开始安装：

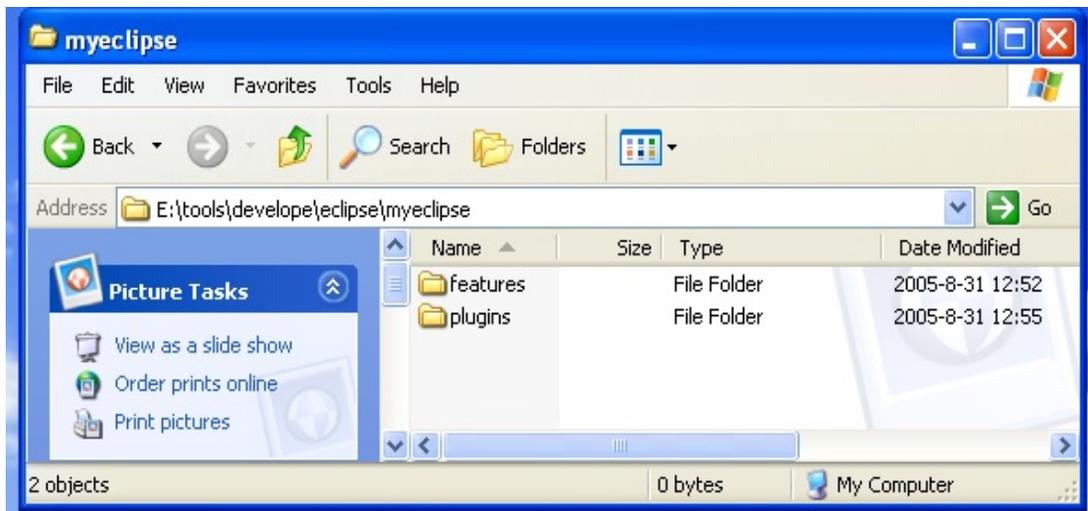
1. 选择你的eclipse所在目录，其它都点下一步，默认安装。



2.安装完毕之后，找到myeclipse的安装目录，如下图所示：

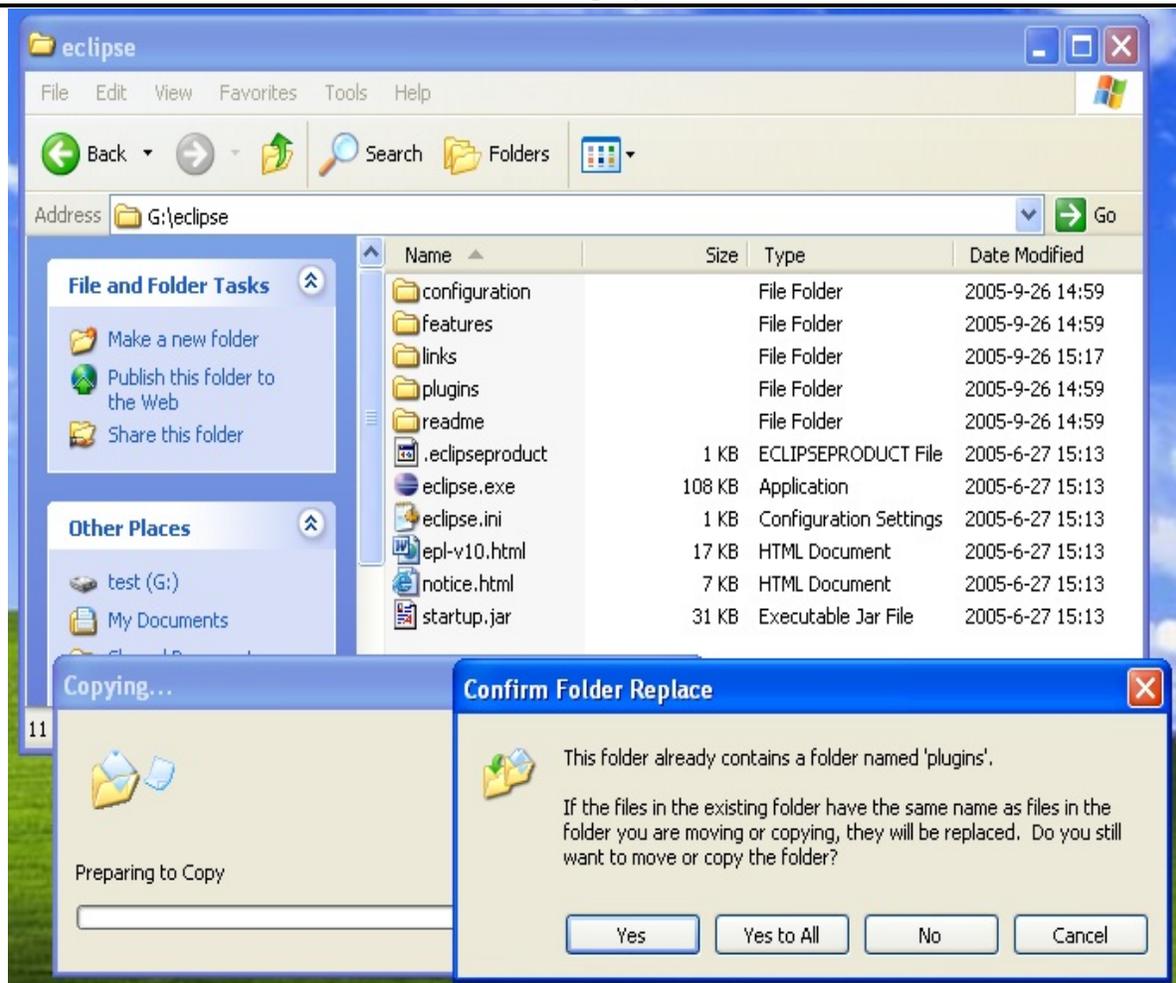


3. 将features和plugins这两个目录复制出来。这样做的目的是为了在一台电脑上可以使用多个独立的eclipse开发平台。如下图所示：



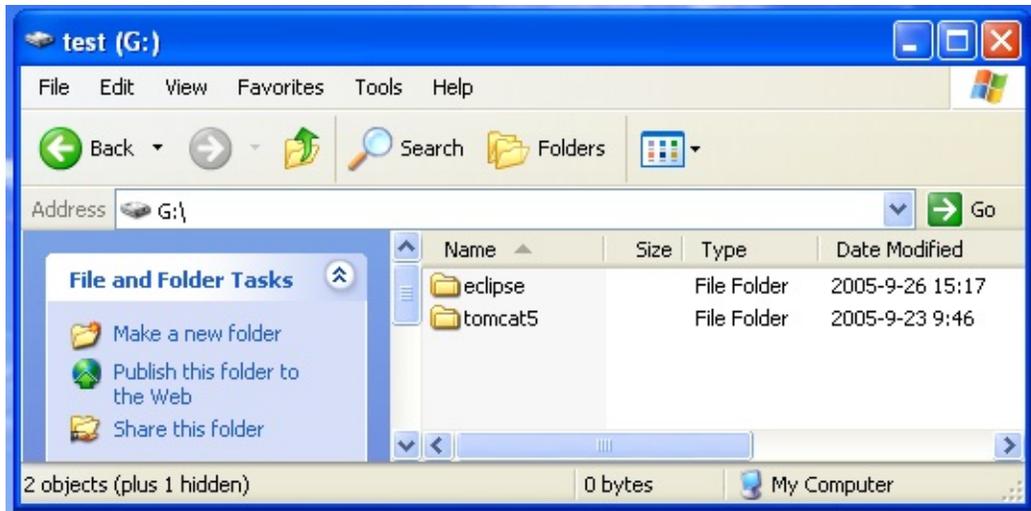
4. 将刚才安装的 myeclipse 删除（反安装）。

5. 将第3步复制的features和plugins这2个目录, 覆盖到 G



解释一下上面做的原因：因为myeclipse是eclipse的插件，所以只要把features和plugins这2个目录覆盖在eclipse的相应目录，eclipse会自动认出这些插件。

到这一步，myeclipse算是安装完毕。G 盘目录如下：

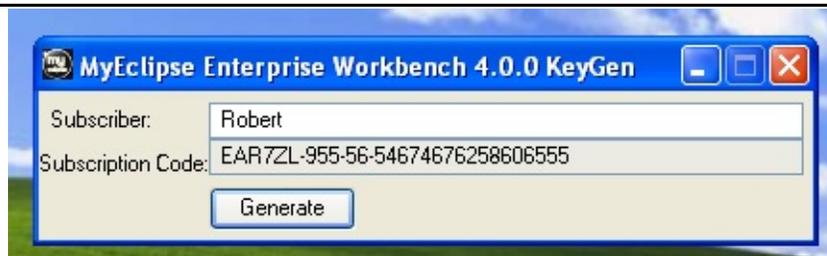


## 五、破解myeclipse4.0

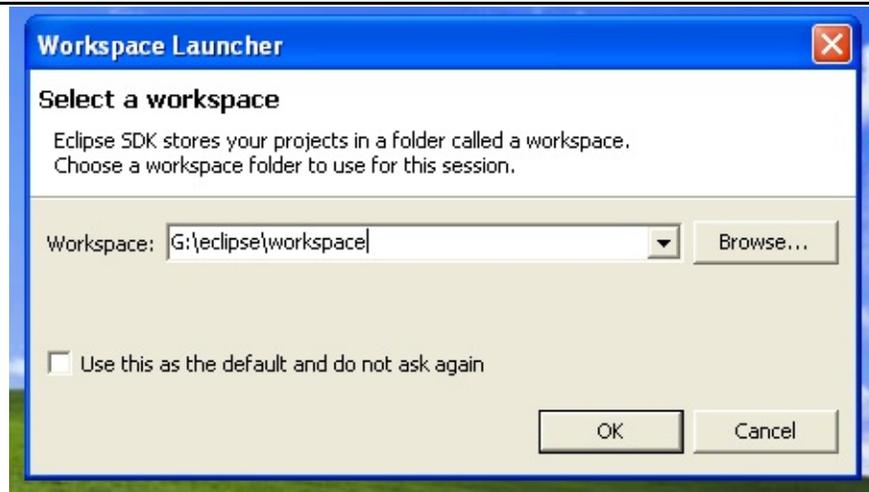
1.去google搜索下载一个注册机。如下图：



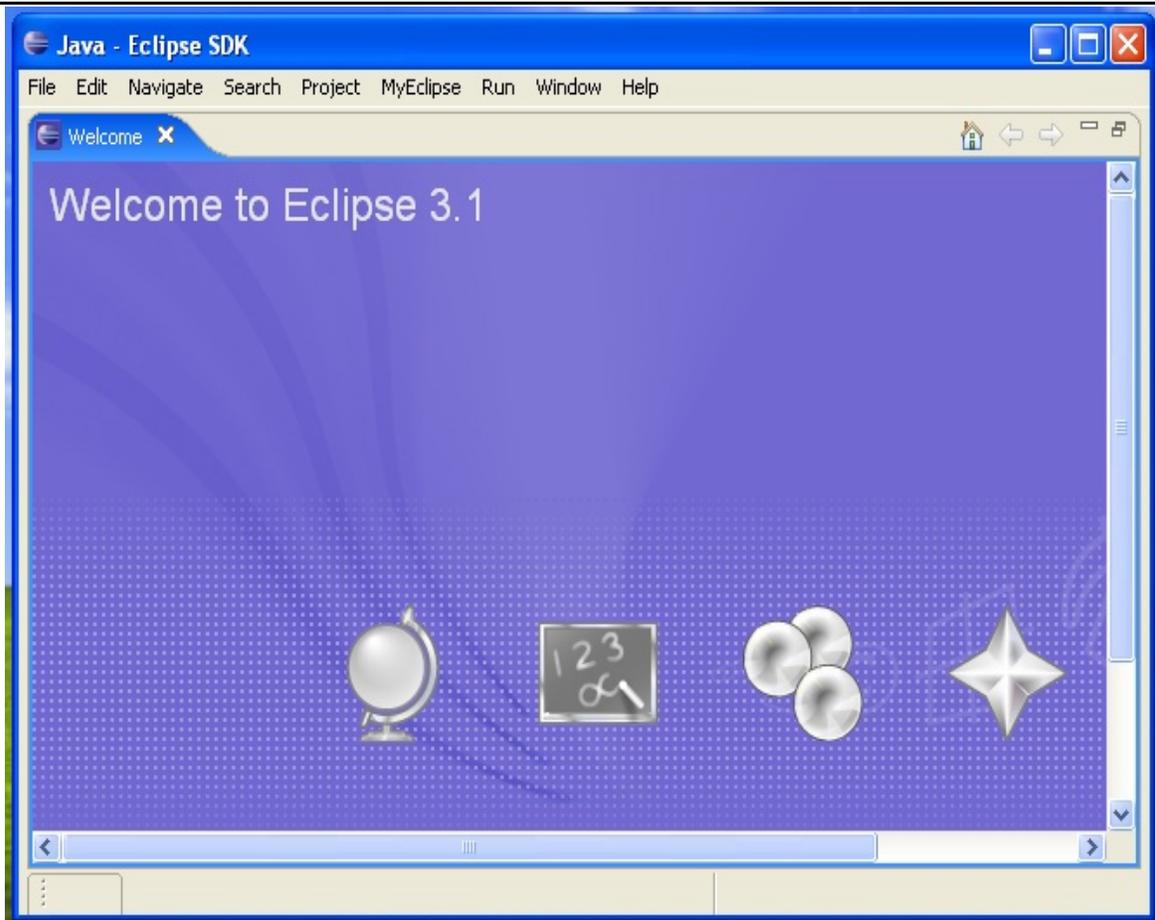
2. 输入名字, 算号, 如下图:



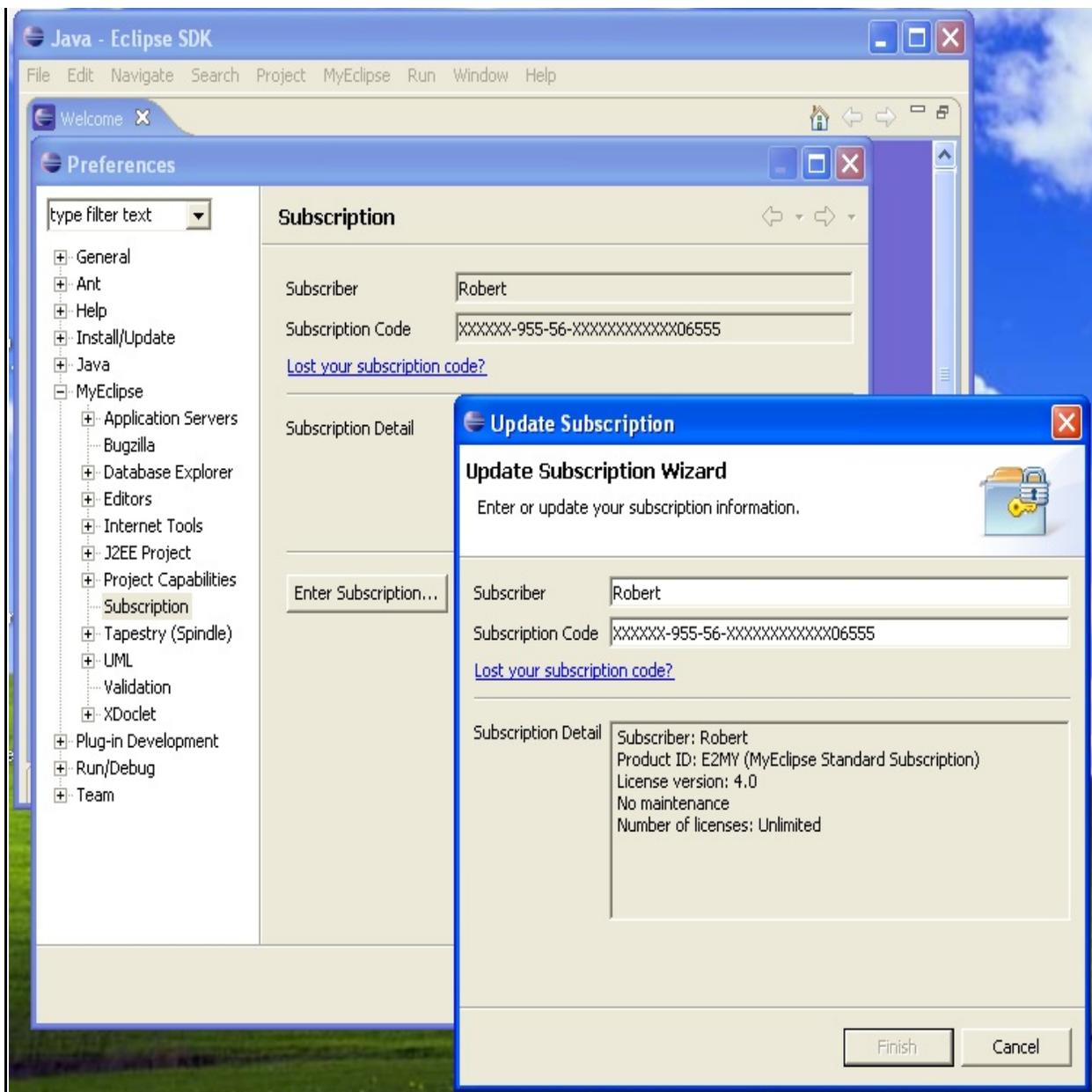
3.运行eclipse.exe, 设定同目录下的workspace , 如下图 :



4.myeclipse插件已经被自动认出, 如下图 :



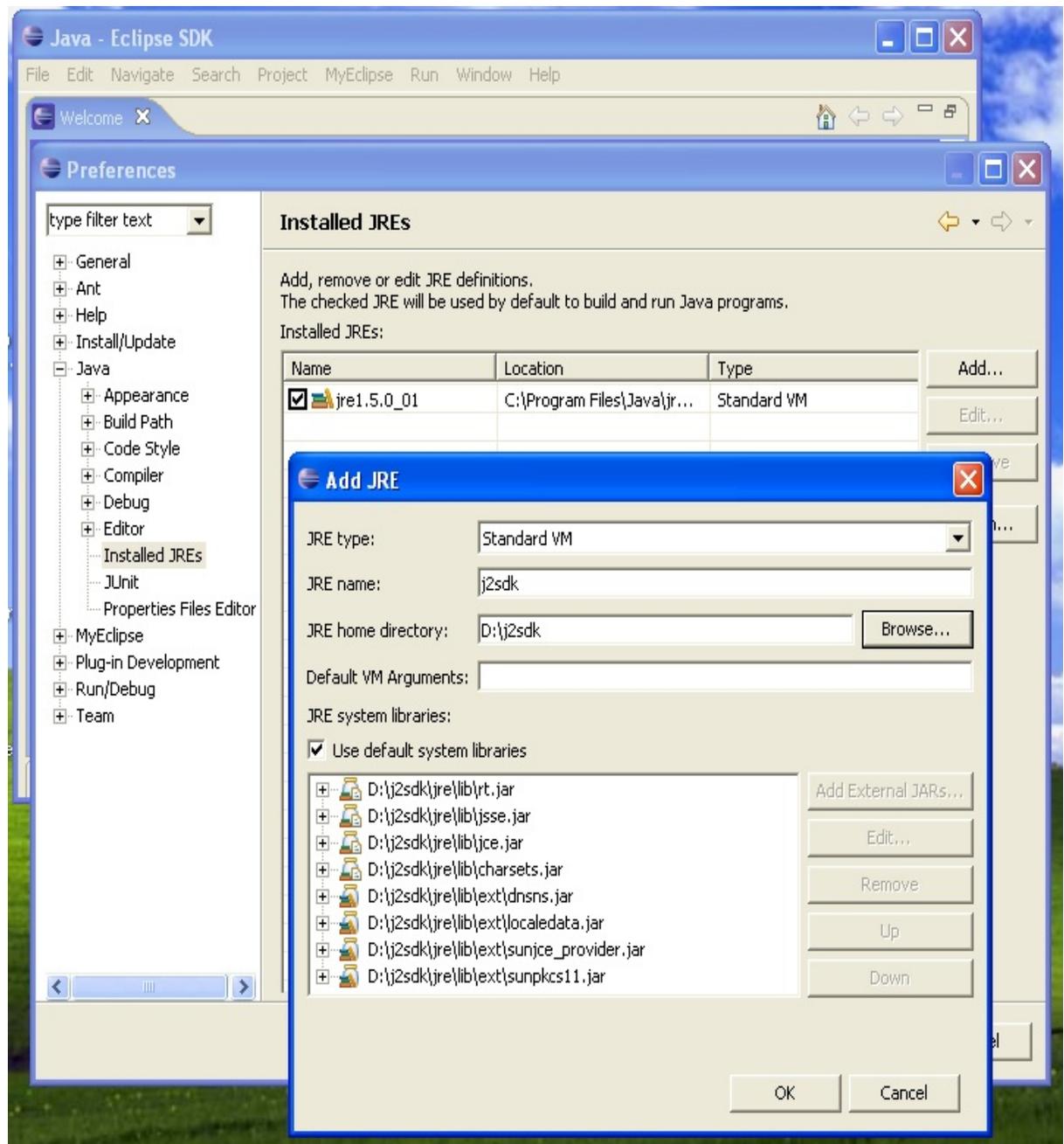
5. 点击 Window ----> Preferences ----> Subscription ----> Enter Subscription, 输入注册码, 破解 myeclipse , 下图 :



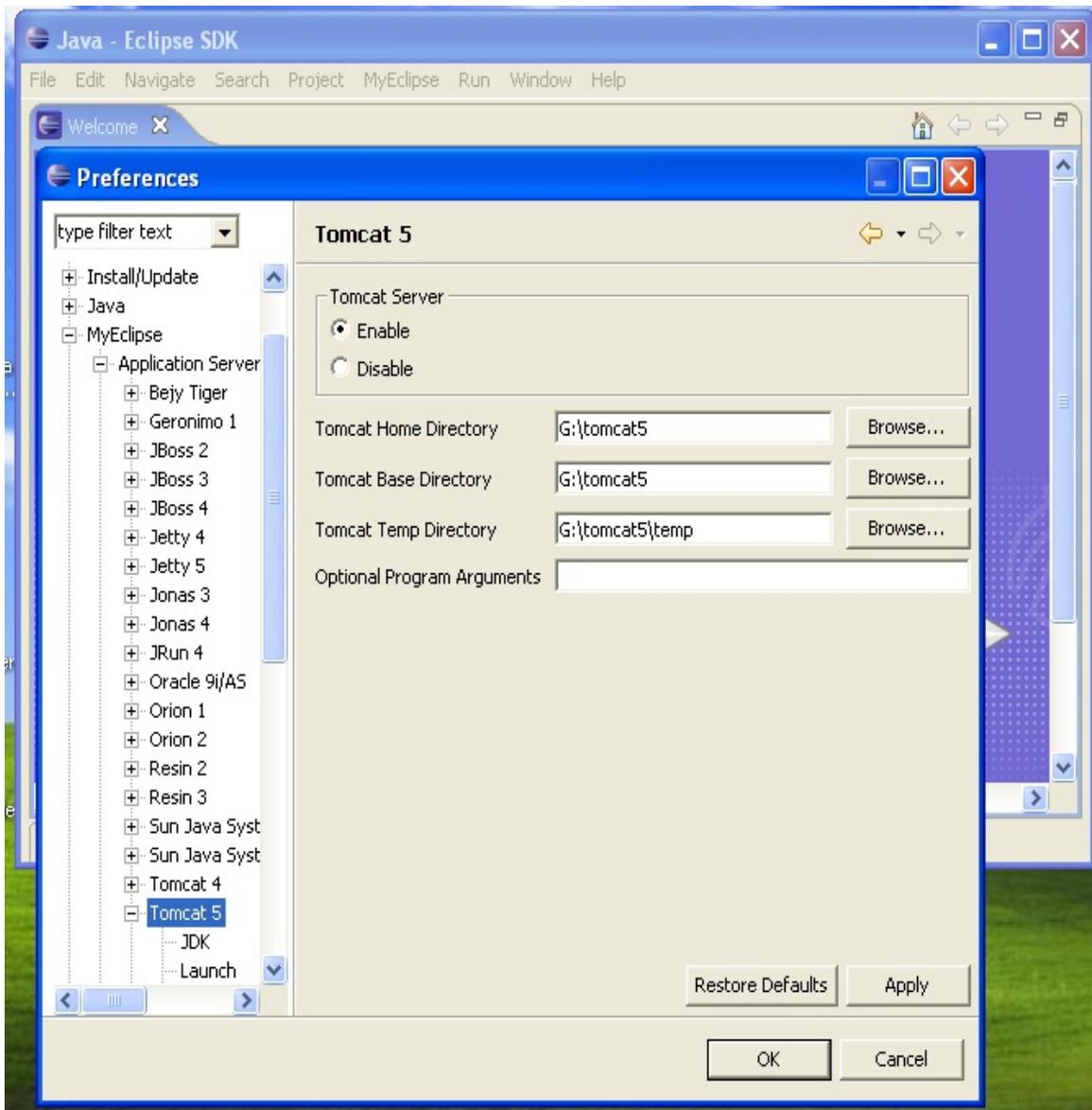
到这里，myeclipse破解完毕。

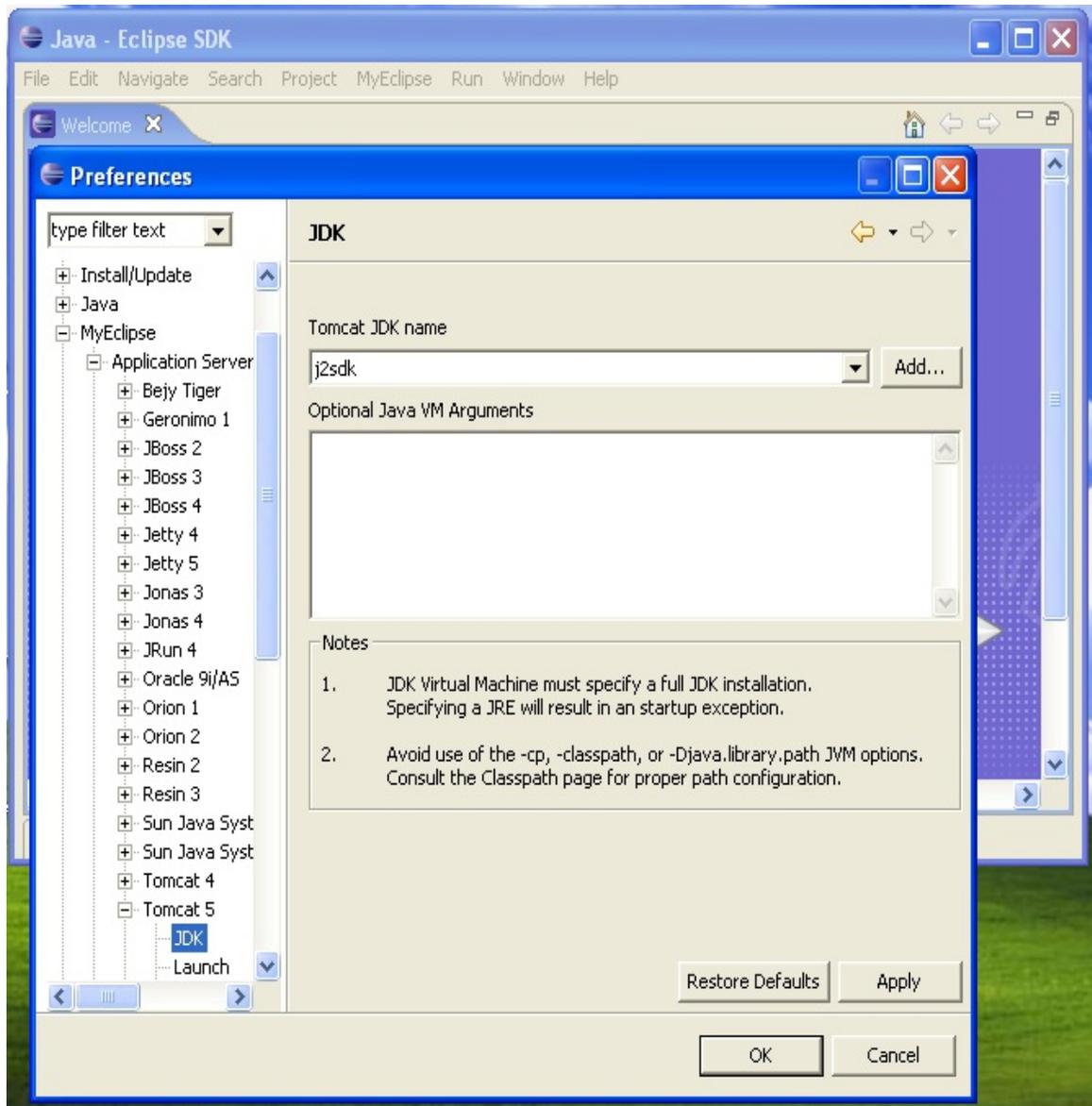
#### 六、设定myeclipse的jdk和tomcat服务器。

1. 增加一个jre，点击 Window ----> Preferences ----> Installed JREs ----> Add 如下图：（找到自己的j2sdk安装目录）



2. 指定Tomcat服务器，Window ----> Preferences ----> Tomcat5，并设定该tomcat所用的jre为我们上一步新增的。如下面2个图：

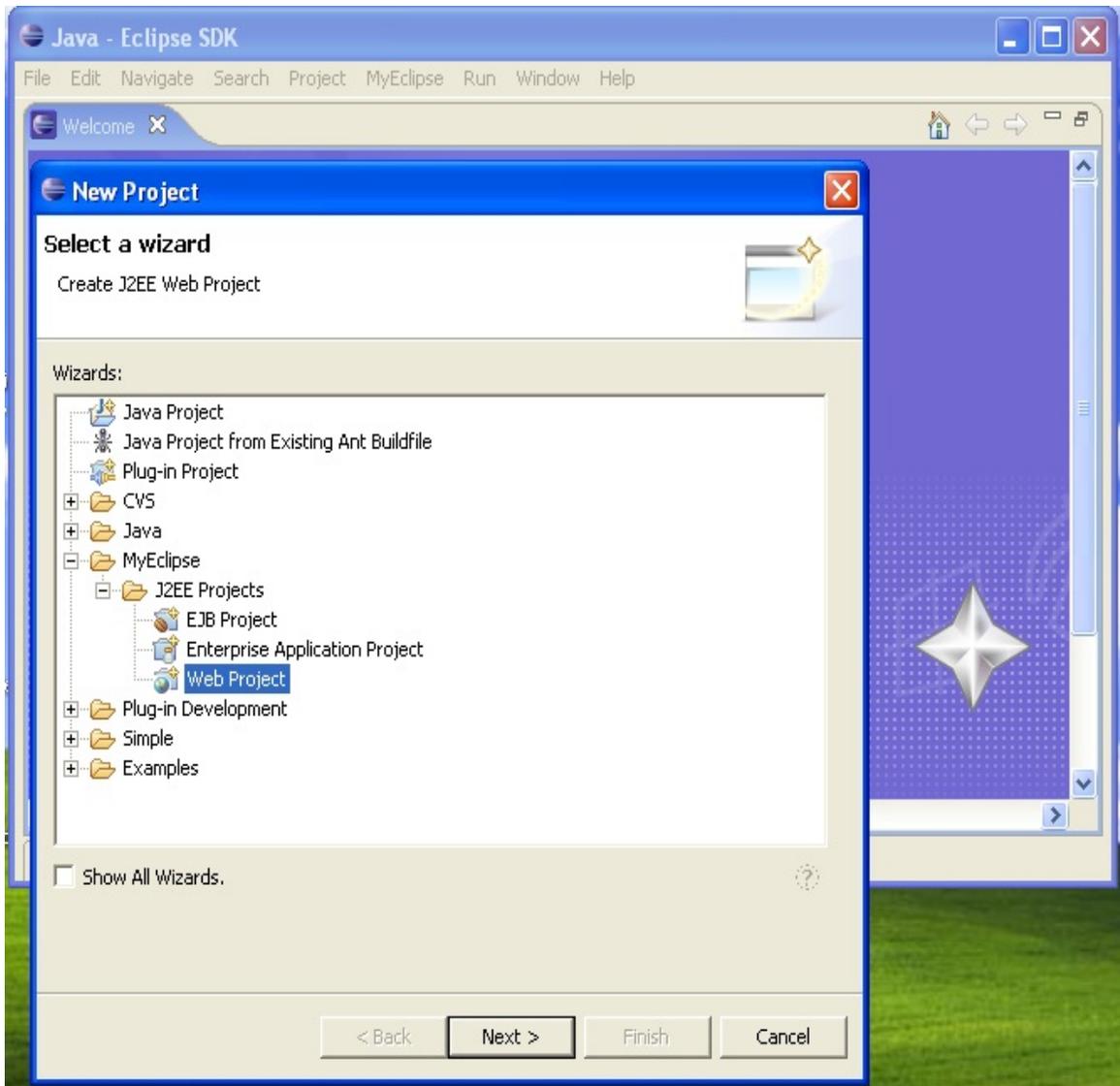




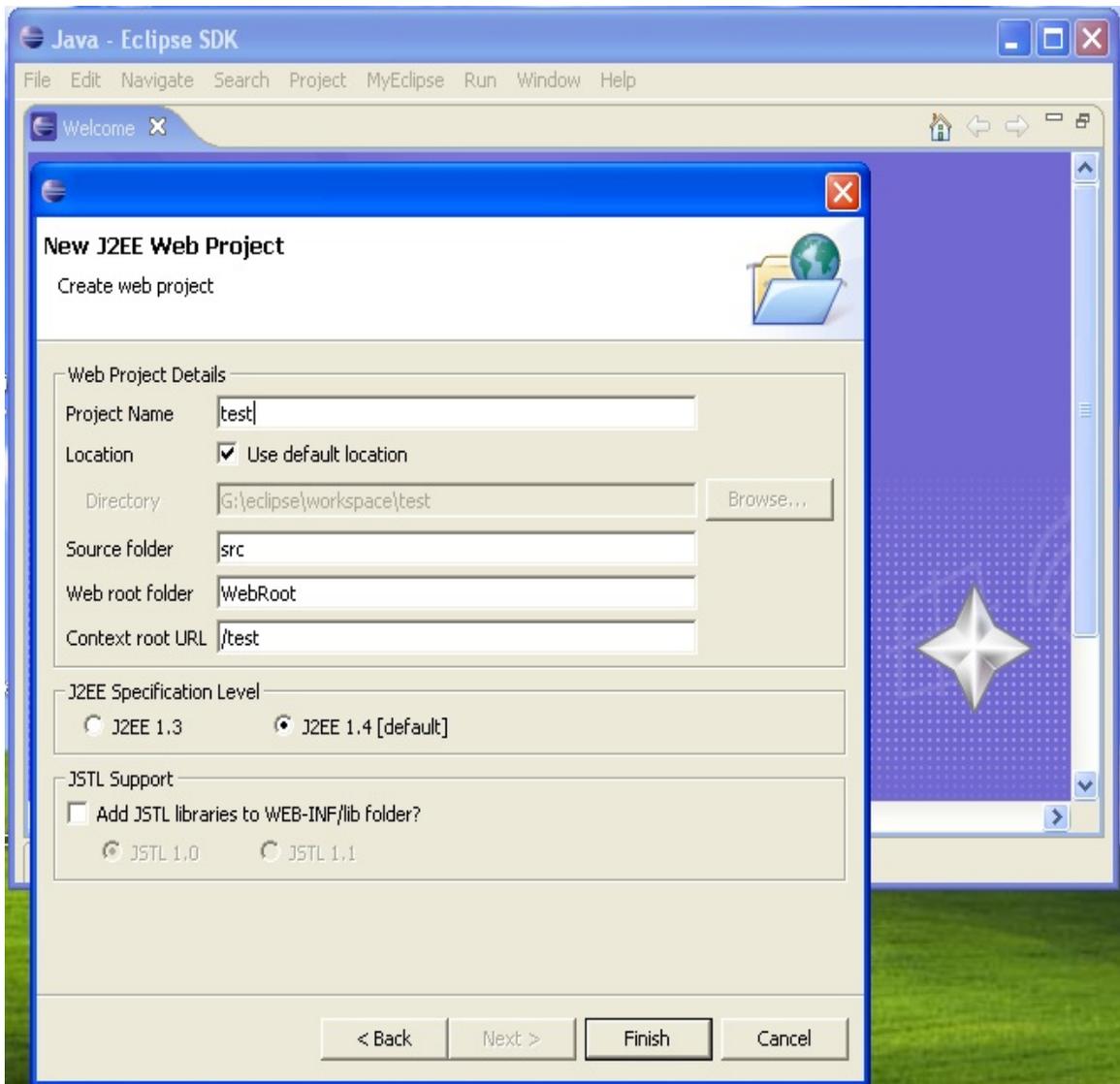
到这一步，我们的开发环境就搭建完毕了。

七、新建一个项目来试一试？

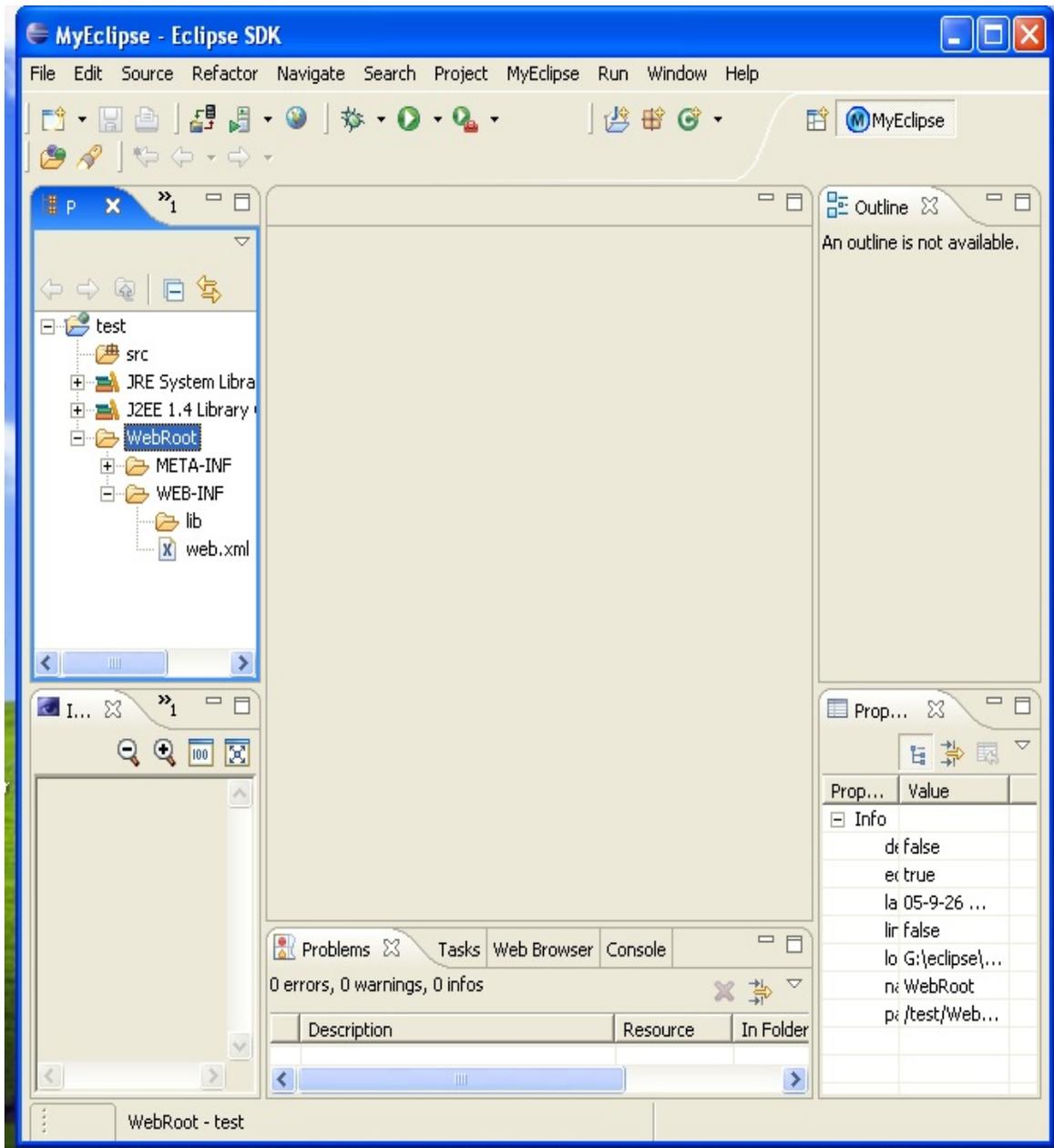
1. 点击 File ----> New ----> Project，选择Web Project，再点下一步，如下图：



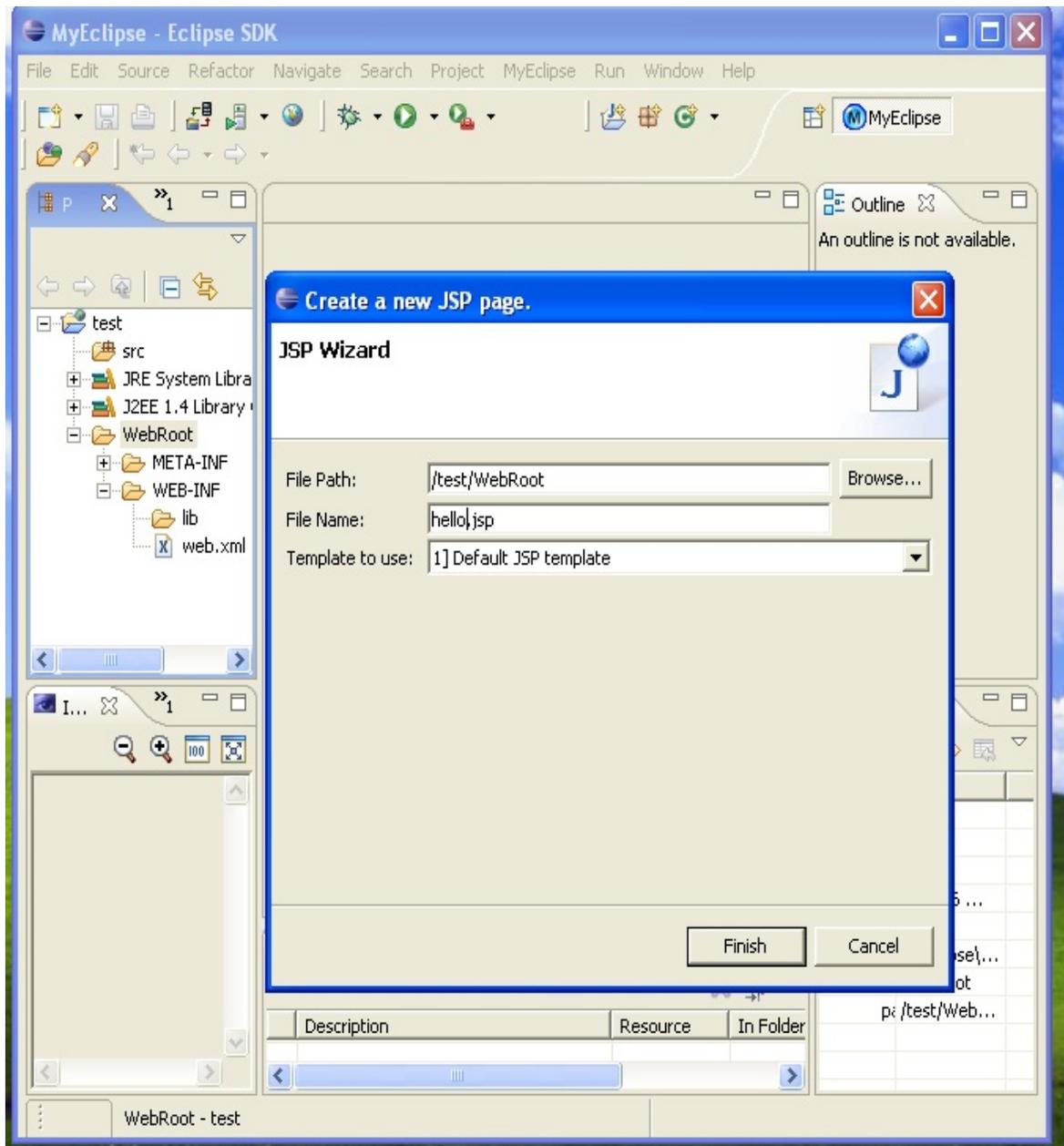
2. 在 Project Name 中输入test，点 finish。如下图：



3. 开发工具自动生成了test项目。 如下图：



4. 点击 WebRoot ----> (右键)New ----> JSP, 新建一个 JSP 页面 (hello.jsp), 如下图:



5. 编辑器生成的是jsp文件是按模板生成的，简单修改一下，代码如下图：

```
hello.jsp x
<%@ page language="java" import="java.util.*" pageEncoding="GBK"%>
<%
String path = request.getContextPath();
String basePath = request.getScheme()+"://"+request.getServerName()+":"+request.get
%>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <head>
    <base href="<%=basePath%>">

    <title>First Project</title>

    <meta http-equiv="pragma" content="no-cache">
    <meta http-equiv="cache-control" content="no-cache">
    <meta http-equiv="expires" content="0">
    <meta http-equiv="keywords" content="keyword1,keyword2,keyword3">
    <meta http-equiv="description" content="This is my page">

    <!--
    <link rel="stylesheet" type="text/css" href="styles.css">
    -->
  </head>

  <body>
    <FORM name="form1" action="hello_result.jsp" method="post">
      <INPUT type="text" name="hello">
      <input type="submit" name="Submit" value="提交">
    </FORM>
  </body>
</html>
```

6. 再用同样的方法，新建一个叫 hello\_result.jsp 的JSP文件，代码如下图：

```
<%@ page language="java" import="java.util.*" pageEncoding="GBK"%>
<%
String path = request.getContextPath();
String basePath = request.getScheme()+"://"+request.getServerName()+":"+request.get
%>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
  <base href="<%=basePath%>">

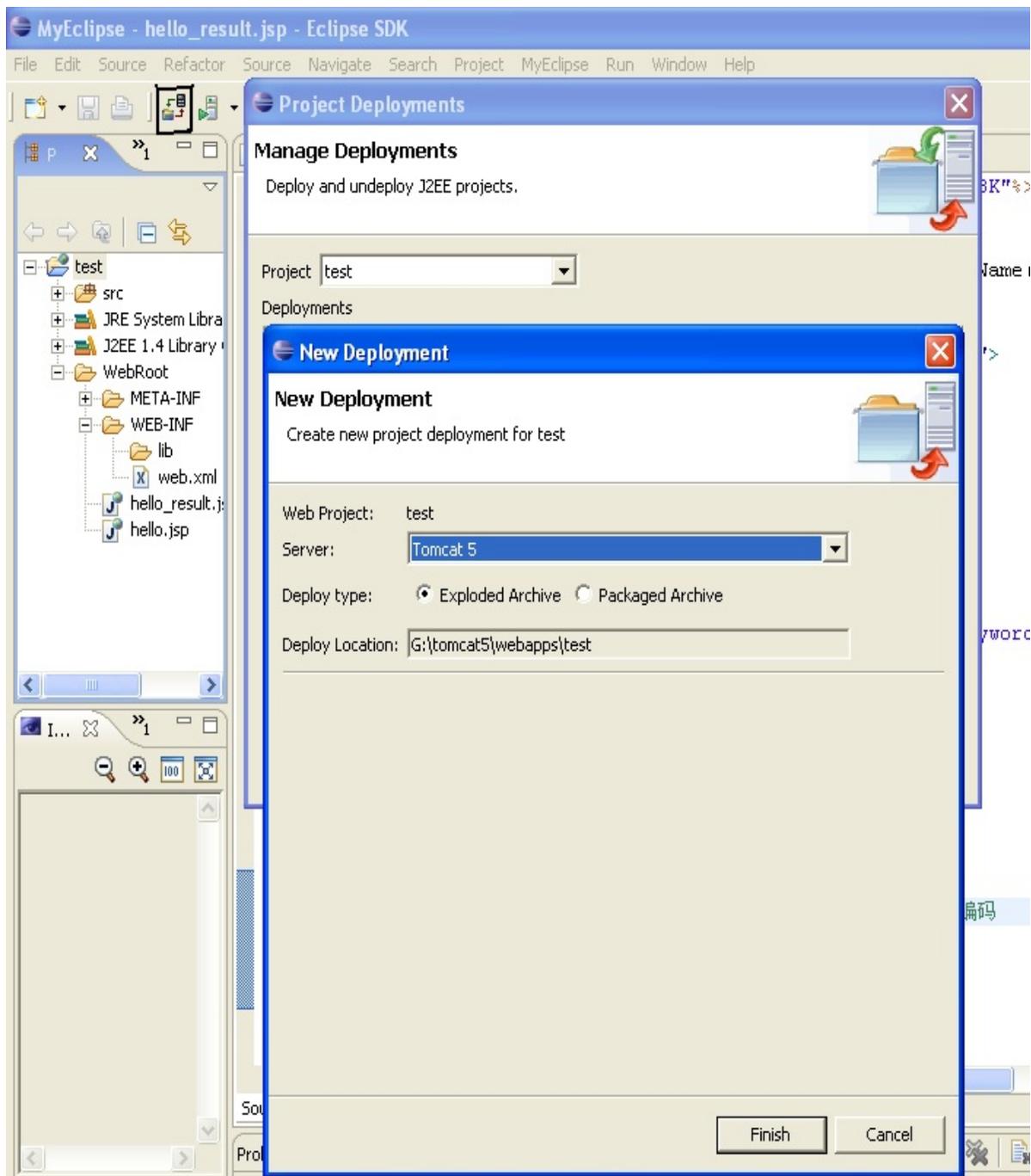
  <title>Faint, First Project Again!</title>

  <meta http-equiv="pragma" content="no-cache">
  <meta http-equiv="cache-control" content="no-cache">
  <meta http-equiv="expires" content="0">
  <meta http-equiv="keywords" content="keyword1,keyword2,keyword3">
  <meta http-equiv="description" content="This is my page">

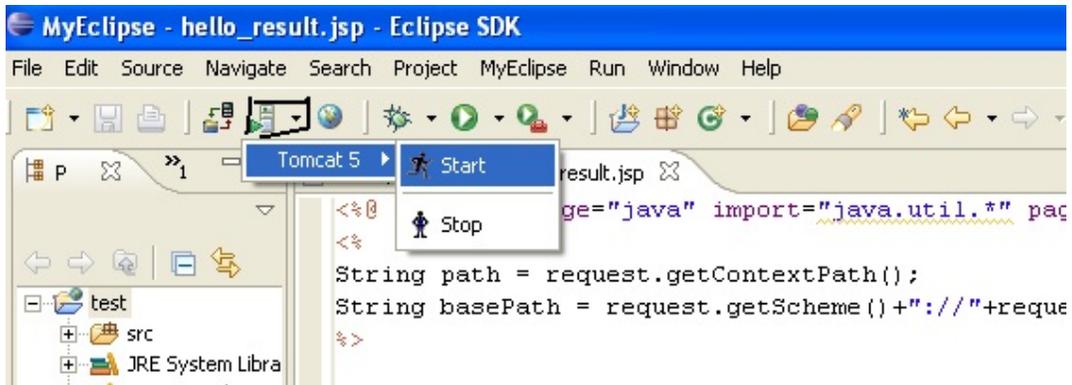
  <!--
  <link rel="stylesheet" type="text/css" href="styles.css">
  -->
</head>

<body>
  <%
    request.setCharacterEncoding("GBK"); //设定中文GBK编码
    String temp = request.getParameter("hello");
    out.println(temp);
  %>
</body>
</html>
```

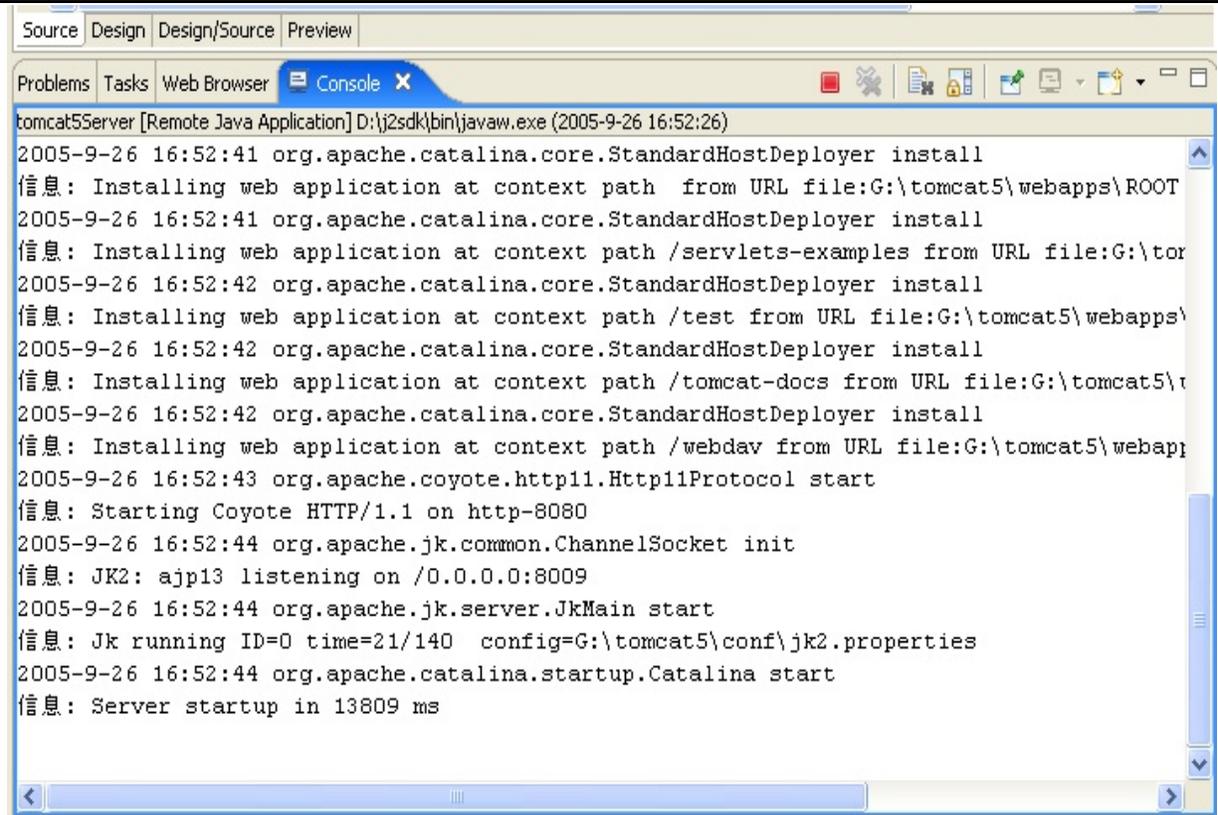
7. deploy (部署) 这个test项目。如下图：(用黑线画了的按钮就是 deploy 的按钮)，在弹出窗口点 Add，部署test项目。



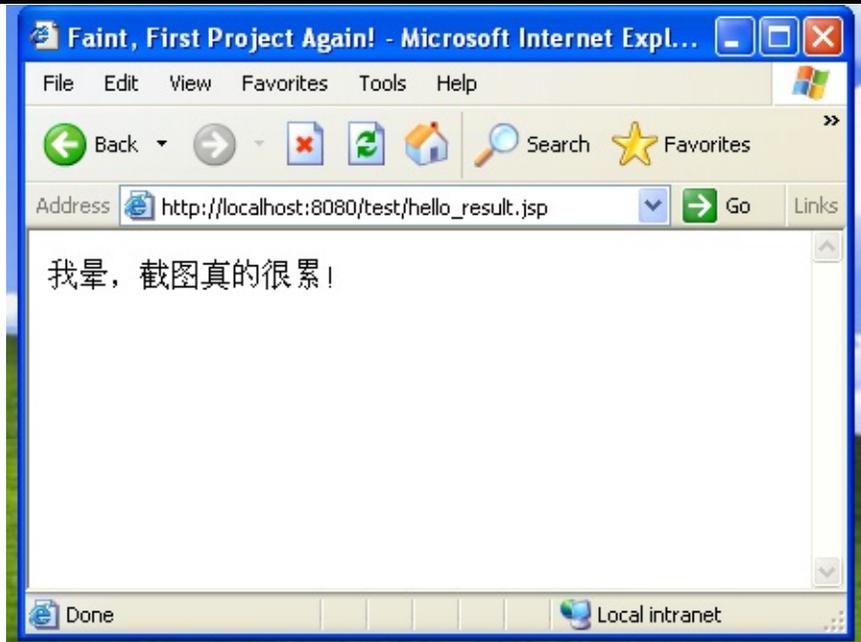
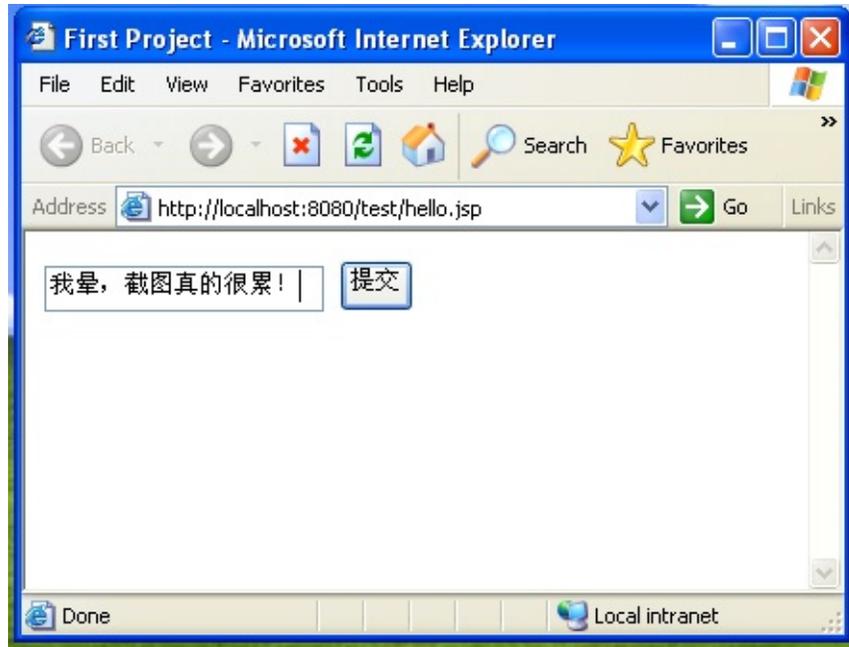
8. 启动Tomcat服务器。如下图：（用黑线画了的按钮就是 启动Tomcat 的按钮）



9. Tomcat 启动输出信息如下图：



10. 网页访问和结果图.



到这里，终于搞完了!!! 大家可以去eclipse的workspace目录下看看test项目的代码，也可以去Tomcat的webapps目录下，看看部署之后的test项目的结构。

# Struts入门指引

[hepeng421](#) 发表于2005-05-20 作者:hepeng421 评价:18/11 评论数:3  
点击数:2587 [[收藏](#)]



摘要：

或许有人觉得struts不容易学，似乎里面的一些概念让未接触过的人迷惑，MVC1、MVC2、模式.....我写这篇文章是想让从来没有接触过struts的人，能有个简单的入门指引.该案例包括首页，用户登陆、网站向导页面。就这么简单，没有深奥的struts概念，主要靠动手，然后用心体会

本文Matrix永久镜

像：<http://www.matrix.org.cn/resource/article/1/1556.html>

说明：本文可能由Matrix原创，也可能由Matrix的会员整理，或者由

Matrix的Crawler在全球知名Java或者其他技术相关站点抓取并永久

保留镜像，Matrix会保留所有原来的出处URL，并在显著地方作出说明，

如果你发觉出处URL有误，请联系Matrix改正.

或许有人觉得struts不容易学，似乎里面的一些概念让未接触过的人迷惑，MVC1、MVC2、模式.....我写这篇文章是想让从来没有接触过struts的人，能有个简单的入门指引，当然，系统地学习struts是必要的，里面有很多让人心醉的东东，那是后话了。

该案例包括首页，用户登陆、网站向导页面。就这么简单，没有深奥的struts概念，主要靠动手，然后用心体会。

WEB Server用tomcat4。到<http://jakarta.apache.org>下载struts1.1，把zip文件释放到c:\struts，拷贝C:\struts\webapps\struts-example.war到c:\tomcat4\webapps中，启动tomcat，war包被释放为struts-example文件夹，删除war包，把struts-example文件夹更名为test。

一、把WEB-INF\web.xml改成：

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web
```

```
<web-app>
```

```
<!--这是struts中的Controller（控制器），系统的指令中转由其，
既ActionServlet类负责，它从struts-config.xml中读取配置信息，
并在服务器后台自动启动一个线程。如果没有特别的要求（如添加
语言编转功能），程序员可以不管这部分，照用就可以了。-->
```

```
<servlet>
<servlet-name>action</servlet-name>
<servlet-class>org.apache.struts.action.ActionServlet</servlet-
<init-param>
<param-name>config</param-name>
<param-value>/WEB-INF/struts-config.xml</param-value>
</init-param>
<load-on-startup>1</load-on-startup>
</servlet>
```

```
<!--该系统的servlet可以映射成cool为后缀的文件，而不是常见的.jspdo等，后缀名可以改成任何名称，当然名字要健康#◎¥%!-->
```

```
<servlet-mapping>  
<servlet-name>action</servlet-name>  
<url-pattern>*.cool</url-pattern>  
</servlet-mapping>  
<!--该系统的默认首页是index.jsp，可以有多个，系统按次序找，类似IIS-->  
<welcome-file-list>  
<welcome-file>index.jsp</welcome-file>  
</welcome-file-list>  
</web-app>
```

## 二、把test\WEB-INF\struts-config.xml改成：

```
<?xml version="1.0" encoding="ISO-8859-1" ?>  
<!DOCTYPE struts-config PUBLIC "-//Apache Software Foundation  
"http://jakarta.apache.org/struts/dtds/struts-config_1_1.dtd
```

```
<struts-config>  
<!--FormBean是struts的一个概念，本质是JavaBean，用来自动存储页面表单  
<form-beans>  
<!--稍后我们会新增一个UserForm类，用来存储用户信息。-->  
<form-bean name="userForm" type="test.UserForm"/>  
</form-beans>  
<!--这里存放整个系统都可以使用的全局转向中转(Forward)地址，类似于java  
<global-forwards>  
<!--failed.cool将被当成servlet请求，到action-mappings中寻找对应的  
<forward name="failed" path="/failed.cool"/>  
<forward name="regist" path="/regist.jsp"/>  
</global-forwards>  
<!--还记得web.xml中后缀为cool的请求吗？它们是转到这里处理的。这里相当
```

```
<action-mappings>
<!--处理regist.cools的请求，使用的FormBean是userForm，既test.UserForm
<action path="/regist" type="test.RegistAction" name="userForm" />
<action path="/overview" forward="/hello.jsp"/>
<action path="/failed" forward="/wuwu.jsp" />
</action-mappings>
</struts-config>
```

三、增加一个FormBean，类路径为test.UserForm，以下是这个类的内容：

```
package test;
import org.apache.struts.action.ActionForm;
public class UserForm extends ActionForm
{
    private String name="lpw";//用户名
    private String ps="1111";//密码
    public UserForm(){}
    public void setName(String s) {name=s;}
    public String getName() {return name;}
    public void setPs(String s) {ps=s;}
    public String getPs() {return ps;}
}
```

四、增加一个Action的子类，类路径为test.RegistAction，以下是这个类的内容：

```
package test;
import java.lang.reflect.InvocationTargetException;
import java.util.Locale;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpSession;
```

```

import javax.servlet.http.HttpServletResponse;
import org.apache.struts.action.Action;
import org.apache.struts.action.ActionError;
import org.apache.struts.action.ActionErrors;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;
import org.apache.struts.util.MessageResources;
import test.UserForm;
public final class RegistAction extends Action
{
    public ActionForward execute(ActionMapping mapping,ActionFo
throws Exception
    {
        Locale locale = getLocale(request);
        MessageResources messages = getResources(request);
        HttpSession session = request.getSession();
        UserForm userform = (UserForm) form;
        //此处可以调用其他类来执行数据库写入或其他逻辑判断
        // 如果UserForm传来的参数name的值为默认的lpw, 将forward到faile
        // 该名称将到struts-config.xml的<global-forwards>中寻找映射的
        // (可以是绝对路径, 也可以是相对路径), 对于本例, 是转到failed.co
        // 还记得吗? 后缀为cool的请求全部到action-mappings中寻找
        // 对应的action处理, 最终目录是wuwu.jsp*/
        if( "lpw".equals(userform.getName()) )
            return (mapping.findForward("failed"));
        else
            return (mapping.findForward("regist"));
    }
}

```

五、以下所有新增或修改的页面相当于struts的View部分，把首页index.jsp改成：

```

<%@ page contentType="text/html;charset=GBK" language="java"
<%@ page import = "test.*" %>
<a href="overview.cool">站点导航</a><br>
<form action="regist.cool" method="post">

```

```
<!--表单中的域的名称要和UserForm中的参数一样，就可以实现数据自动获取功能  
用户:<input type="text" name="name"><br>  
密码:<input type="password" name="ps"><br>  
<input type="submit" value="新增用户">  
</form>
```

## 六、增加hello.jsp，用于站点导航：

```
<h1>site map</h1>The following is content filling by reader
```

## 七、增加wuwu.jsp，当没有新用户登陆时，将转到这个页面：

```
<%@ page contentType="text/html;charset=GBK" language="java"  
<jsp:useBean id="beanlpw" class="test.UserForm" scope="session"  
现有用户：<%=beanlpw.getName()%><br>  
密码：<%=beanlpw.getPs()%><br>
```

没有得到新的用户！

## 八、增加regist.jsp，当有新用户登陆时，将转到这个页面：

```
<%@ page contentType="text/html;charset=GBK" language="java"  
<jsp:useBean id="beanlpw" class="test.UserForm" scope="session"  
新用户帐号：<%=beanlpw.getName()%><br>  
密码：<%=beanlpw.getPs()%>
```

## 九、启动tomcat4，浏览器中键入

<http://localhost:8080/test/index.jsp>，操作一下，就可以看到结果，并初步理解struts的M、V、C各部分的协同工作原理，当然这是作者的良好意愿，如果读者看得一头雾水，欢迎指出错误在哪里：  
：)

## 实例学习 Struts

作者：胡海生

Email : hanson at silentme  
net

网

站：<http://vip.6to23.com/hanson/>

申明：未经作者同意，谢绝转

选用纯 JSP 还是纯 Servlet 设计站点都有它的局限性，Struts 就是把它系在一起的一种有力工具。采用 Struts 能开发出基于 MVC 模式的应用于 MVC 的概念可以参见 GoF 的《设计模式——可复用面向对象软件的基础》。

你现在要做的是，下载、安装、配置好以下的工具，版本不同的话操作有些差异，具体的看它们的文档吧：

- Tomcat 4.1.24
- Apache 2.0.43, w/ mod\_jk2 2.0.43
- Java 2 SDK Standard Edition 1.4.0
- Struts 1.1
- Eclipse 2.1.0

Struts 是用 Java 写的，应此它需要 JDK 1.2 或者更高版本。如果你用 JDK 1.4，就像我，XML parser 和 JDBC 2.0 Optional Package Bin 已经被默认的了。

## 新项目

在这个例程中我们要开发一个简单的 web 应用，允许用户登录和注销。起见，数据被设定为常数，而不是保存在数据库中，毕竟这里要讲的是 Struts，而不是 Java。

首先在你的 Tomcat 配置的应用主目录中创建一个目录，比方说 logonApp。在 logonApp 中创建目录 src 和 WEB-INF，在 WEB-INF 建目录 classes 和 lib，从 Struts 的分发中拷贝 struts.jar 到 lib 目录且也把拷贝 \$CATALINA\_HOME/common/lib/servlets.jar 到 lib 目录

Struts 的分发中拷贝所有的 struts\*.tld 到 WEB-INF 目录。

现在打开 Eclipse，你会看到四个 view。现在我们要建立一个新的项目。点击 File -> New Project，打开了一个窗口，在第一个窗格中选择 Java，第二个窗格中选择 Java Project，点击 Next。输入项目名称（为了好记也叫 logonApp 吧），去掉 use default 复选框的对勾，浏览到 logonApp 目录，点击 Next。出现一个新的窗口，在 Source tab 上点击 Add Folder，添加 \$APP\_BASE/src，在 Default output folder 中填入 \$APP\_BASE/WEB-INF/classes，点击 Finish。点击 Window -> Open Perspective -> Resource，看看 .project 文件是否已经自动包含了 I 目录中所有的 jar 文件。

你的 logonApp/WEB-INF/web.xml 应该如下所示：

```
<?xml version="1.0"?>
<!DOCTYPE web-app
PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/j2ee/dtds/web-app_2_3.dtd">

<web-app>
  <!-- Action Servlet Configuration -->
  <servlet>
    <servlet-name>action</servlet-name>
    <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
    <init-param>
      <param-name>config</param-name>
      <param-value>/WEB-INF/struts-config.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <!-- Action Servlet Mapping -->
  <servlet-mapping>
    <servlet-name>action</servlet-name>
    <url-pattern>*.do</url-pattern>
  </servlet-mapping>

  <!-- The Welcome File List -->
```

```

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

<!-- Struts Tag Library Descriptors -->
<taglib>
  <taglib-uri>/WEB-INF/struts-bean.tld</taglib-uri>
  <taglib-location>/WEB-INF/struts-bean.tld</taglib-location>
</taglib>

<taglib>
  <taglib-uri>/WEB-INF/struts-html.tld</taglib-uri>
  <taglib-location>/WEB-INF/struts-html.tld</taglib-location>
</taglib>

<taglib>
  <taglib-uri>/WEB-INF/struts-logic.tld</taglib-uri>
  <taglib-location>/WEB-INF/struts-logic.tld</taglib-location>
</taglib>
</web-app>

```

Struts 的配置文件 logonApp/WEB-INF/struts-config.xml 如下 :

```

<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE struts-config PUBLIC
  "-//Apache Software Foundation//DTD Struts Configuration 1.1//EN"
  "http://jakarta.apache.org/struts/dtds/struts-config_1_1.dtd">

<struts-config>
  <form-beans>
    <form-bean name="logonForm"
      type="org.apache.struts.validator.DynaValidatorForm">
      <form-property name="username" type="java.lang.String"/>
      <form-property name="password" type="java.lang.String"/>
    </form-bean>
  </form-beans>

  <global-forwards>

```

```

    <forward name="success"          path="/main.jsp"/>
    <forward name="logoff"           path="/logoff.do"/>
</global-forwards>

<action-mappings>
  <action path="/logon"
        type="org.monotonous.struts.LogonAction"
        name="logonForm"
        scope="session"
        input="logon">
  </action>

  <action path="/logoff"
        type="org.monotonous.struts.LogoffAction">
    <forward name="success"          path="/index.jsp"/>
  </action>
</action-mappings>

<controller>
  <!-- The "input" parameter on "action" elements is the name of a
        local or global "forward" rather than a module-relative path -->
  <set-property property="inputForward" value="true"/>
</controller>

<message-resources parameter="org.monotonous.struts.ApplicationResource" />
</struts-config>

```

## 创建 View

现在回到 Eclipse , 建立一个新页面 index.jsp :

```

<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean" %>
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html" %>

<html:html locale="true">
<head>
  <title><bean:message key="index.title" /></title>

```

```
<html:base/>
</head>

<body>
<html:errors/>
<html:form action="/logon">
  <bean:message key="prompt.username"/>
  <html:text property="username"/>
  <br/>
  <bean:message key="prompt.password"/>
  <html:password property="password"/>
  <br/>
  <html:submit>
    <bean:message key="index.logon"/>
  </html:submit>
</html:form>
</body>
</html:html>
```

成功登录后的页面 main.jsp :

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean" %>
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html" %>

<html:html>
<head>
  <title><bean:message key="main.title"/></title>
  <html:base/>
</head>
<body>
<html:link forward="logoff">
<bean:message key="main.logoff"/>
</html:link>
</body>
</html:html>
```

你可能注意到这两个页面中都使用了方便国际化的特性，这至少需要一

的属性文件 ApplicationResources.properties :

```
index.title=Struts Homepage
prompt.username=Username
prompt.password=Password
index.logon=Log on
main.title=Struts Main page
main.logoff=Log off
error.password.mismatch=Invalid username and/or password.
```

## 创建 **Controller**

LogonAction.java :

```
package org.monotonous.struts;

import java.util.Locale;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpSession;
import javax.servlet.http.HttpServletResponse;
import org.apache.struts.action.Action;
import org.apache.struts.action.ActionError;
import org.apache.struts.action.ActionErrors;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;
import org.apache.struts.util.MessageResources;
import org.apache.commons.beanutils.PropertyUtils;

public final class LogonAction extends Action {
    public ActionForward execute(
        ActionMapping mapping,
        ActionForm form,
        HttpServletRequest request,
        HttpServletResponse response)
        throws Exception {
        Locale locale = getLocale(request);
```

```

MessageResources messages = getResources(request);

// Validate the request parameters specified by the user
ActionErrors errors = new ActionErrors();
String username =
    (String) PropertyUtils.getSimpleProperty(form, "username");
String password =
    (String) PropertyUtils.getSimpleProperty(form, "password");

if ((username != "foo") || (password != "bar"))
    errors.add(ActionErrors.GLOBAL_ERROR,
        new ActionError("error.password.mismatch"));

// Report any errors we have discovered back to the original form
if (!errors.isEmpty()) {
    saveErrors(request, errors);
    return (mapping.getInputForward());
}

// Save our logged-in user in the session
HttpSession session = request.getSession();
// Do something with session...

// Remove the obsolete form bean
if (mapping.getAttribute() != null) {
    if ("request".equals(mapping.getScope()))
        request.removeAttribute(mapping.getAttribute());
    else
        session.removeAttribute(mapping.getAttribute());
}

// Forward control to the specified success URI
return (mapping.findForward("success"));
}
}

```

LogoffAction.java :

```
package org.monotonous.struts;

import java.util.Locale;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpSession;
import javax.servlet.http.HttpServletResponse;
import org.apache.struts.action.Action;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;
import org.apache.struts.util.MessageResources;

public final class LogoffAction extends Action {
    public ActionForward execute(
        ActionMapping mapping,
        ActionForm form,
        HttpServletRequest request,
        HttpServletResponse response)
        throws Exception {

        Locale locale = getLocale(request);
        MessageResources messages = getResources(request);
        HttpSession session = request.getSession();

        session.removeAttribute("userattrib");
        session.invalidate();

        // Forward control to the specified success URI
        return (mapping.findForward("success"));
    }
}
```

到浏览器里面欣赏一下吧，不过还不到开香槟的时候，也许你应该为这考虑一些安全措施，下一次我再讲咯。

## Struts起步之Helloworld

从今天开始，将进入神秘的Struts之旅，因为以前没有Jsp开发经验，猛一看Jsp、Struts、taglib真是乱七八糟的。还好今天几天的Struts研究，总算是有些新的，希望写出来与大家共享，也算是抛砖引玉吧，废话少说，下面开始吧：

在没有Struts经验之前，最好的办法是先建立一个Struts开发平台，先做出一个Hello world的小程序，然后再来研究它实现的原理。

### 功能说明

在一个jsp页面(HelloWorld.jsp)中，输入你的名字，通过struts 将你的名字加上Helloworld字样，然后在另外一个jsp页面(ShowHelloWorld.jsp)显示出来。

### 试验过程

整个过程大致可以分为下面七个步骤：

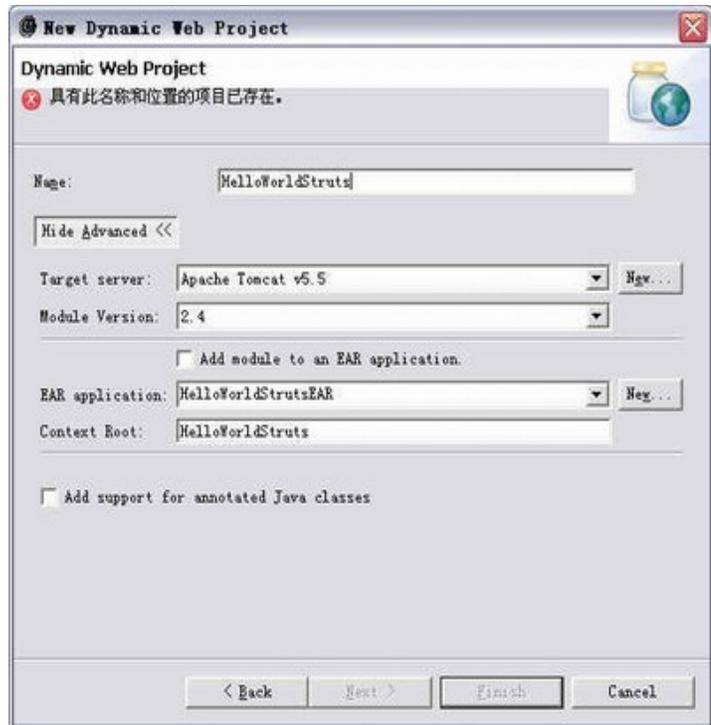
- 1、建立struts开发环境。
- 2、写Helloworld jsp页面
- 3、写struts 中hello word Action实现
- 4、写struts 中hello word ActionForm实现
- 5、写显示Helloworld jsp页面
- 6、配置web.xml文件
- 7、配置struts\_config.xml文件

### 建立struts开发环境

我使用的开发环境是Lomboz 现在已经支持Eclipse3.1,现在lomboz已经将Eclipse3.1集成在一起提供下载，可以直接下载[lomboz-eclipse-emf-gef-jem-3.1RC1.zip](#)，进行Jsp的web开发，当然我这里还使用了tomcat，版本是5.5.9，在安装Lomboz之前，请先安装tomcat.如何使用lomboz进行配置和web开发，或许你可以看看这篇文章，写的很详细:[Building and Running a Web Application](#)

配置好了Lomboz之后，下一步添加Lomboz对Struts的支持，先从Apache下载[Struts](#) 我下载的版本是1.2.7。打开下载所得到的zip文件，可以看到里面有三个目录：contrib，lib和webapps。对我们重要的是lib和webapps。将webapps目录下的war文件拷贝到tomcat的webapps目录下，假设你的tomcat安装在c:/tomcat下，你将war文件拷贝到c:/tomcat/webapps/目录下即可。重启tomcat。打开浏览器，在地址栏输入：<http://localhost:8080/Struts-example/index.jsp>，若能看到“powered by Struts”的深蓝图标，即说明安装成功。

下面用Lomboz创建一个HelloWorldStruts工程。在New project ->选择J2EE web project,在New Dynamic Web Project对话框中，输入工程名HelloWorldStruts，注意Target server的名称一定要选:Apache Tomcat v5.5



在lomboz Package Explore中，右击 HelloWorldStruts 项目 ->Properties, 在弹出的对话框中，选择右边的Java Build Path ->Libraries->Add External Jars 将\struts-1.2.7\lib目录下的所有Jar文件添加进来。然后将\struts-1.2.7\lib目录下的除了Jar文件的\*.dtd/\*.tld/\*.xml文件全都拷贝到 HelloWorldStruts项目的 WebContent/WEB-INF目录下。这样这个项目已经可以支持struts开发了。

配置了半天，终于可以编码了：

## 写Helloworld jsp页面

右击Helloworldstruts工程中的 WebContent目录上->new ->other->jsp添加一个HelloWorld.jsp文件，在这个文件中，用户输入自己的用户

名，然后提交，源代码清单如下：

```
<%@ taglib uri="/WEB-INF/struts-  
html.tld" prefix=SPAN>"html"%>  
⊞ <%@ page contentType="text/html  
; charset=gb2312"%>  
  <!--  
      * This page shows Struts  
Hello world Demo jsp page  
      * Title: Struts Hello world D  
emo  
      * Description: Struts Hello  
world Demo Page  
      * Copyright: Copyright (c)  
2005  
      * Company: http://www.cn  
weblog.com/sundy  
      * @author Sundy(一缕阳  
光) <sundy26@126.com>  
      * @version 1.0  
  -->  
  <html>  
  <head>  
  <title>Struts Hello world Demo</titl  
e>  
  </head>  
  <body>  
    输入你的用户名，提交显示结果：  
    <html:form action="/HelloWorld" fo  
cus="username">  
      用户名称:<html:text property="user  
name" size="25" />  
      <html:submit property="submit"  
/>  
    </html:form>  
  </body>  
</html>
```

注意到第一行的:

```
<%@ taglib uri="/WEB-INF/struts-  
html.tld" prefix="html"%>
```

这里到我们添加了struts的taglib的引用，struts标签库的使用，在后续的文章中将陆续介绍。

## 写struts 中hello word Action实现

在HelloWorldStruts/JavaSource目录下，添加HelloWorldAction类，它继承自

org.apache.struts.action.Action，并实现其execute方法，其代码清单如下:

```
package com.sundy.struts;  
  
import org.apache.struts.action.Action;  
import org.apache.struts.action.ActionForward;  
import org.apache.struts.action.ActionMapping;  
import org.apache.struts.action.ActionForm;  
import javax.servlet.http.*;  
import org.apache.struts.action.*;  
  
/**  
 * This page shows HelloWorldAction  
 * Copyright: Copyright (c) 2005  
 * Company: http://www.cnweblog.com/sundy  
 * @author Sundy(一缕阳光) <sundy26@126.com>
```

```

| * @version 1.0
| */
| public class HelloWorldAction extends Action {
|     public ActionForward execute(ActionMapping actionMapping,
|         ActionForm actionForm, HttpServletRequest request,
|         HttpServletResponse response) throws Exception {
|         HelloWorldForm form = (HelloWorldForm) actionForm;
|         ActionErrors errors = new ActionErrors();
|         String username = form.getUsername();
|
|         username += ",Hello world!";
|         request.setAttribute("hello", username);
|
|         return actionMapping.findForward("success");
|     }
| }
}

```

## 写struts 中hello word ActionForm 实现

在HelloWorldStruts/JavaSource目录下，添加HelloWorldForm类，它继承自  
org.apache.struts.action.ActionForm

, 在里面只有一个username属性, 其代码清单如下:

```
package com.sundy.struts;

import javax.servlet.http.*;
import org.apache.struts.action.*;

/**
 * This page shows Hello world De
 * mo Helloworld Form page
 * Copyright: Copyright (c) 2005
 * Company: http://www.cnweblog.com/sundy
 * @author Sundy(一缕阳光) <sundy26@126.com>
 * @version 1.0
 */
public class HelloWorldForm extends ActionForm {
    private static final long serialVersionUID = 3256445798169261619L;
    private String username;
    public HelloWorldForm() {
        username = null;
    }
    public void setUsername(String username) {
        this.username = username;
    }
    public String getUsername() {
        return this.username;
    }
    public void reset(ActionMapping mapping, HttpServletRequest request) {
        username = null;
    }
}
```

```
}  
}
```

## 写显示Helloworld jsp页面

右击Helloworldstruts工程中的WebContent目录上->new ->other->jsp添加一个ShowHelloWorld.jsp文件，在这个文件中，显示输出结果，代码清单如下：

```
<html>  
<head>  
<title>Show Hellow world</title>  
</head>  
<body>  
    <h2><%= request.getAttribute("hello")%></h2>  
</body>  
</html>
```

## 配置web.xml文件

需要在WebContent/WEB-INF目录下修改web.xml文件，添加对struts的支持，代码清单如下：

```
<?xml version="1.0" encoding="ISO-8859-1"?>  
<!DOCTYPE web-app  
    PUBLIC "-//Sun Microsystems, Inc.  
    //DTD Web Application 2.3//EN"  
    "http://java.sun.com/j2ee/dtds/web-app_2_3.dtd">  
<web-app>  
    <display-name>Struts Logon Ap
```

```
plication</display-name>
  <!-- Standard Action Servlet Con
figuration (with debugging) -->
  <servlet>
    <servlet-name>action</servle
t-name>
    <servlet-class>
      org.apache.struts.action.Actio
nServlet
    </servlet-class>
    <init-param>
      <param-name>config</par
am-name>
      <param-value>/WEB-INF/s
truts-config.xml</param-value>
    </init-param>
    <init-param>
      <param-name>debug</par
am-name>
      <param-value>2</param-v
alue>
    </init-param>
    <init-param>
      <param-name>detail</para
m-name>
      <param-value>2</param-v
alue>
    </init-param>
    <load-on-startup>2</load-on-
startup>
  </servlet>
  <!-- Standard Action Servlet Ma
pping -->
  <servlet-mapping>
    <servlet-name>action</servle
t-name>
    <url-pattern>*.do</url-pattern
```

```
>
</servlet-mapping>
<!-- The Welcome File List -->
<welcome-file-list>
  <welcome-file>HelloWorld.jsp
</welcome-file>
</welcome-file-list>
<!-- Struts Tag Library Descriptors -->
<taglib>
  <taglib-uri>/tags/struts-bean</taglib-uri>
  <taglib-location>/WEB-INF/struts-bean.tld</taglib-location>
</taglib>
<taglib>
  <taglib-uri>/tags/struts-html</taglib-uri>
  <taglib-location>/WEB-INF/struts-html.tld</taglib-location>
</taglib>
<taglib>
  <taglib-uri>/tags/struts-logic</taglib-uri>
  <taglib-location>/WEB-INF/struts-logic.tld</taglib-location>
</taglib>
<taglib>
  <taglib-uri>/tags/struts-nested</taglib-uri>
  <taglib-location>/WEB-INF/struts-nested.tld</taglib-location>
</taglib>
<taglib>
  <taglib-uri>/tags/struts-tiles</taglib-uri>
  <taglib-location>/WEB-INF/str
```

```
uts-tiles.tld</taglib-location>
</taglib>
</web-app>
```

## 配置struts\_config.xml文件

在WebContent/WEB-INF目录下修改struts-config.xml文件，添加HelloWorldAction和HelloWorldForm的映射，代码清单如下：

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE struts-config PUBLIC

"-//Apache Software Foundation//DTD Struts Configuration 1.1//EN"
"http://jakarta.apache.org/struts/dtds/struts-config_1_1.dtd">
<struts-config>
  <form-beans>
    <form-bean name="HelloWorldForm" type="com.sundy.struts.HelloWorldForm"/>
  </form-beans>
  <action-mappings>
    <action path="/HelloWorld" type="com.sundy.struts.HelloWorldAction"
      name="HelloWorldForm" scope="request" input="">
      <forward name="success" path="/ShowHelloWorld.jsp"/>
    </action>
  </action-mappings>
  <message-resources paramet
```

```
er="resources.application"/>  
</struts-config>
```

运行:

好了，所有的代码都已经编好了，虽然脑子里还有无数的疑问，它又怎么工作的呢，Action和ActionForm又有什么联系呢?不要紧，先来看看运行结果吧，姑且让这些疑问，放到后面来解答。

在HelloWorld.jsp文件上 ->右键->run  
as -> run on server

ApplicationContext是Spring的核心，Context我们通常解释为上下文环境，我想用“容器”来表述它更容易理解一些，ApplicationContext则是“应用的容器”了:P，Spring把Bean放在这个容器中，在需要的时候，用getBean方法取出，虽然我没有看过这一部分的源代码，但我想它应该是一个类似Map的结构。

在Web应用中，我们会用到WebApplicationContext，WebApplicationContext继承自ApplicationContext，先让我们看看在Web应用中，怎么初始化WebApplicationContext，在web.xml中定义：

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

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

<!-- OR USE THE CONTEXTLOADERSERVLET INSTEAD OF THE LI
  STENER
  <servlet>
    <servlet-name>context</servlet-name>
<servlet-class>org.springframework.web.context.ContextLoaderServlet</se
  rvlet-class>
  <load-on-startup>1</load-on-startup>
  </servlet>
  -->
```

可以看出，有两种方法，一个是用ContextLoaderListener这个Listener，另一个是ContextLoaderServlet这个Servlet，这两个方法都是在web应用启动的时候来初始化WebApplicationContext，我个人认为Listener要比Servlet更好一些，因为Listener监听应用的启动和结束，而Servlet得启动要稍微延迟一些，如果在这时要做一些业务的操作，启动的前后顺序是有影响的。

那么在ContextLoaderListener和ContextLoaderServlet中到底做了什么呢？

```
以ContextLoaderListener为例，我们可以看到
public void contextInitialized(ServletContextEvent event) {
    this.contextLoader = createContextLoader();
    this.contextLoader.initWebApplicationContext(event.getServletContext());
```

```

    }
    protected ContextLoader createContextLoader() {
        return new ContextLoader();
    }
}

```

ContextLoader是一个工具类，用来初始化WebApplicationContext，其主要方法就是initWebApplicationContext，我们继续追踪initWebApplicationContext这个方法（具体代码我不贴出，大家可以看Spring中的源码），我们发现，原来ContextLoader是把WebApplicationContext（XmlWebApplicationContext是默认实现类）放在了ServletContext中，ServletContext也是一个“容器”，也是一个类似Map的结构，而WebApplicationContext在ServletContext中的KEY就是WebApplicationContext.ROOT\_WEB\_APPLICATION\_CONTEXT\_ATTRIBUTE，我们如果要使用WebApplicationContext则需要从ServletContext取出，Spring提供了一个WebApplicationContextUtils类，可以方便的取出WebApplicationContext，只要把ServletContext传入就可以了。

上面我们介绍了WebApplicationContext在Servlet容器中初始化的原理，一般的Web应用就可以轻松的使用了，但是，随着Struts的广泛应用，把Struts和Spring整个起来，是一个需要面对的问题，Spring本身也提供了Struts的相关类，主要使用的有org.springframework.web.struts.ActionSupport，我们只要把自己的Action继承自ActionSupport，就是可以调用ActionSupport中getWebApplicationContext()的方法取出WebApplicationContext，但这样一来在Action中，需要取得业务逻辑的地方都要getBean，看上去不够简洁，所以Spring又提供了另一个方法，用org.springframework.web.struts.ContextLoaderPlugIn，这是一个Struts的Plug，在Struts启动时加载，对于Action，可以像管理Bean一样来管理，在struts-config.xml中Action的配置变成类似下面的样子

```

<action attribute="aForm" name="aForm" path="/aAction" scope="request"
    type="org.springframework.web.struts.DelegatingActionProxy">
    <forward name="forward" path="forward.jsp" />
</action>

```

注意type变成了org.springframework.web.struts.DelegatingActionProxy，之后我们需要建立action-servlet.xml这样的文件，action-servlet.xml符合Spring的spring-beans.dtd标准，在里面定义类似下面的

```

<bean name="/aAction" class="com.web.action.Aaction" singleton="false"
    >
    <property name="businessService">
        <ref bean="businessService"/>
    </property>
</bean>

```

com.web.action.Aaction是Action的实现类，businessService是需要的业

务逻辑，Spring会把businessService注入到Action中，在Action中只要写businessService的get和set方法就可以了，还有一点，action的bean是singleton="false"，即每次新建一个实例，这也解决了Struts中Action的线程同步问题，具体过程是当用户做"/aAction"的HTTP请求（当然应该是"/aAction.do"），Struts会找到这个Action的对应类org.springframework.web.struts.DelegatingActionProxy，DelegatingActionProxy是个代理类，它会去找action-servlet.xml文件中"/aAction"对应的真正实现类，然后把它实例化，同时把需要的业务对象注入，然后执行Action的execute方法。

使用了ContextLoaderPlugIn，在struts-config.xml中变成类似这样配置

```
<plug-in className="org.springframework.web.struts.ContextLoaderPlugIn">
  <set-property property="contextConfigLocation" value="/WEB-INF/applicationContext.xml,/WEB-INF/action-servlet.xml" />
</plug-in>
```

而在web.xml中不再需要ContextLoaderListener或是ContextLoaderServlet

说到这里不知道大家会不会有这样的疑问，如果使用ContextLoaderPlugIn，如果我们有些程序是脱离Struts的Action环境，我们怎么处理，比如我们要自定义标记库，在标记库中，我们需要调用Spring管理的业务层逻辑对象，这时候我们就很麻烦，因为只有在action中动态注入业务逻辑，其他我们似乎不能取得Spring的WebApplicationContext。

别急，我们还是来看一下ContextLoaderPlugIn的源码（源码不再贴出），我们可以发现，原来ContextLoaderPlugIn仍然是把WebApplicationContext放在ServletContext中，只是这个KEY不太一样了，这个KEY值为ContextLoaderPlugIn.SERVLET\_CONTEXT\_PREFIX+ModuleConfig.getPrefix()（具体请查看源代码），这下好了，我们知道了WebApplicationContext放在哪里，只要我们在Web应用中能够取到ServletContext也就能取到WebApplicationContext了:)

Spring是一个很强大的框架，希望大家在使用过程中不断的深入，了解其更多的特性，我在这里抛砖引玉，有什么不对的地方，请大家指出。

## 使用Spring Quartz执行定时任务

风间残月

[Quartz](#)是OpenSymphony下的一个开源项目，提供了比JDK的TimeTask更功能。[Spring](#)在Quartz的基础上包装了一层，使得在不使用数据库配置(再用Quartz的JavaBean设置参数，代码更优雅，可配置性

下面我就举个简单的例子。首先，配置Spring的配置文件，起名叫ap

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

        <!-- 配置 -->
        <bean name="randomPriceJob" class="org.springframework.scheduling.qu

            <property name="jobClass">
                <value>test.RandomPriceJob</value>
            </property>

            <property name="jobDataAsMap">
                <map>
                    <entry key="timeout"><value>5</value></entry>
                </map>
            </property>

        </bean>
        <!-- 配置触发器 -->
        <bean id="cronTrigger" class="org.springframework.scheduling.quartz.

            <property name="jobDetail">
                <ref bean="randomPriceJob"/>
            </property>
        <!-- 每天的11点到11点59分中，每分钟触发RandomPriceJob，具1
            <property name="cronExpression">
                <value>0 * 11 * * ?</value>
            </property>
```

```

        </bean>

<bean class="org.springframework.scheduling.quartz.SchedulerF

        <!-- 添加触发器 -->
        <property name="triggers">
            <list>
                <ref local="cronTrigger"/>
            </list>
        </property>
    </bean>

</beans>

```

然后编写具体操作代码

```

        package test;

        import org.apache.log4j.Category;

        import org.quartz.JobExecutionContext;
        import org.quartz.JobExecutionException;
        import org.springframework.scheduling.quartz.QuartzJob

                /**
                 * @author shenshan
                 * @version 1.0
                 */
        public class RandomPriceJob extends QuartzJobBean
        {
            private static final Category cat = Category
                .getInstance( RandomPriceJob.class );

            private int    timeout;

                /**

```

```

        * @param timeout
        */
        public void setTimeout( int timeout )
        {
            this.timeout = timeout;
        }

        /*
        * (non-Javadoc)
        *
        * @see org.springframework.scheduling.quartz.QuartzJobBean#executeInternal(
        *      JobExecutionContext)
        */
        protected void executeInternal( JobExecutionContext context )
            throws JobExecutionException
        {
            cat.debug( "Job start" );

            //执行具体操作

        }
    }

```

### 最后编写运行程序

```

package test;

import org.quartz.Scheduler;
import org.quartz.impl.StdSchedulerFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.ClassPathResource;
import org.springframework.scheduling.quartz.CronTrigger;
import org.springframework.scheduling.quartz.JobDetail;

/**
 * @author shenshan
 * @version 1.0

```

```

        */
        public class RandomPrice
        {
            public static void main( String[ ] args ) throws Excepti
            {
                ClassPathResource res = new ClassPathResource( "applicationCo
                XmlBeanFactory factory = new XmlBeanFactory( res
                JobDetailBean job = ( JobDetailBean ) factory
                .getBean( "randomPriceJob" );
                CronTriggerBean trigger = ( CronTriggerBean ) facto
                .getBean( "cronTrigger" );
                Scheduler scheduler = StdSchedulerFactory.getDefaultSche
                scheduler.start( );
                scheduler.scheduleJob( job, trigger );
            }
        }
    }

```

编译后运行RandomPrice就OK了。需要注意的是，必须使用main函数才  
it。

#### 附：cronExpression配置说明

字段	允许值	允许的特殊字符
秒	0-59	, - * /
分	0-59	, - * /
小时	0-23	, - * /
日期	1-31	, - * ? / L W C
月份	1-12 或者 JAN-DEC	, - * /
星期	1-7 或者 SUN-SAT	, - * ? / L C #
年（可选）	留空, 1970-2099	, - * /

表达式	意义
"0 0 12 * * ?"	每天中午12点触发
"0 15 10 ? * *"	每天上午10:15触发
"0 15 10 * * ?"	每天上午10:15触发

"0 15 10 * * ? *"	每天上午10:15触发
"0 15 10 * * ? 2005"	2005年的每天上午10:15触发
"0 * 14 * * ?"	在每天下午2点到下午2:59期间的每1分钟触发
"0 0/5 14 * * ?"	在每天下午2点到下午2:55期间的每5分钟触发
"0 0/5 14,18 * * ?"	在每天下午2点到2:55期间和下午6点到6:55期间
"0 0-5 14 * * ?"	在每天下午2点到下午2:05期间的每1分钟触发
"0 10,44 14 ? 3 WED"	每年三月的星期三的下午2:10和2:44触发
"0 15 10 ? * MON-FRI"	周一至周五的上午10:15触发
"0 15 10 15 * ?"	每月15日上午10:15触发
"0 15 10 L * ?"	每月最后一日的上午10:15触发
"0 15 10 ? * 6L"	每月的最后一个星期五上午10:15触发
"0 15 10 ? * 6L 2002-2005"	2002年至2005年的每月的最后一个星期五上午10
"0 15 10 ? * 6#3"	每月的第三个星期五上午10:15触发

## 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="mySessionFactory" class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">

<property name="mappingResources">
<list>
<value>GameCity.hbm.xml</value>
</list>
</property>

<property name="entityCacheStrategies">
<props>
<prop key="test.GameCity">read-only</prop>
</props>
</property>

<property name="hibernateProperties">
<props>
<prop key="hibernate.dialect">org.hibernate.dialect.MySQLDialect</prop>

<prop key="hibernate.connection.provider_class">org.hibernate.connection.ProxoolConnectionProvider</prop>
<prop key="hibernate.proxool.pool_alias">xjgame</prop>
<prop key="hibernate.proxool.xml">proxool.xml</prop>

<prop key="hibernate.cache.provider_class">org.hibernate.cache.EhCacheProvider</prop>

<prop key="hibernate.cache.use_query_cache">true</prop>
</props>
</property>
</bean>
</beans>
```

## proxool.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<something-else-entirely>
<proxool>
<alias>xjgame</alias>
<driver-url>jdbc:mysql://localhost/xjgame</driver-url>
<driver-class>com.mysql.jdbc.Driver</driver-class>
<driver-properties>
<property name="user" value="root"/>
<property name="password" value="1234"/>
</driver-properties>
```

```
<maximum-connection-count>10</maximum-connection-count>  
<house-keeping-test-sql>select CURRENT_DATE</house-keeping-test-sql>  
<statistics>30s</statistics>  
<statistics-log-level>FATAL</statistics-log-level>  
</proxool>  
</something-else-entirely>
```

## Test.java

```
/**  
 *  
 */  
package test;  
  
import java.util.List;  
  
import org.hibernate.Session;  
import org.hibernate.SessionFactory;  
import org.springframework.beans.factory.xml.XmlBeanFactory;  
import org.springframework.core.io.ClassPathResource;  
  
/**  
 * @author shenshan  
 * @version 1.0  
 */  
public class Test  
{  
  
    /**  
     * @param args  
     */  
    public static void main( String[ ] args )  
    {  
        ClassPathResource res = new ClassPathResource( "applicationContext.xml" );  
        XmlBeanFactory factory = new XmlBeanFactory( res );  
        SessionFactory sf = ( SessionFactory ) factory  
            .getBean( "mySessionFactory" );  
        Session sess = sf.openSession( );  
  
        List rs = sess.createQuery( "FROM test.GameCity" ).list( );  
  
        for ( int i = 0; i < rs.size( ); i++ )  
        {  
            System.out.println( rs.get( i ) );  
        }  
  
        sess.close( );  
    }  
}
```



# Hibernate3与spring的整合应用

作者：TOM 来源：技术学习者 更新日期：2005-09-15

Spring为应用程序提供一个容器, 为应用程序的管理带来了方便. 它与hibernate的结合, 形成一个完整的后台体系, 也是当今应用开发流行的做法. 奋斗了一个晚上, 终于把hibernate3与spring整合了起来, hibernate2.x和hibernate3与spring的结合稍有不同, 关键是引入的spring的包的不同, 下面我会标识出来.

Spring 的配置文件applicationContext.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "spring" "../..../lib/spring-beans.dtd" >
<beans default-autowire="no" default-dependency-check="none" default-lazy-init="false">

  <!--
    配置数据源
    注意: 用org.apache.commons.dbcp.BasicDataSource, 要引入 apache commons
    的commons-collections-3.1.jar, commons-dbcp-1.2.1.jar, commons-pool-
    1.2.jar三个包
  -->
  <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
  destroy-method="close">
    <property name="driverClassName">
      <value>org.gjt.mm.mysql.Driver</value>
    </property>
    <property name="url">
      <value>jdbc:mysql://localhost/sparkcrm</value>
    </property>
    <property name="username">
      <value>root</value>
    </property>
    <property name="password">
      <value>1111</value>
    </property>
  </bean>

  <!-- 配置sessionFactory, 注意这里引入的包的不同 -->
```

```
<bean id="sessionFactory"
class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
<property name="dataSource">
<ref local="dataSource" />
</property>
<property name="mappingResources">
<list>
<value>com/sparkcrm/schema/entities/Lead.hbm.xml</value>
</list>
</property>
<property name="hibernateProperties">
<props>
<prop key="hibernate.dialect">
org.hibernate.dialect.MySQLDialect
</prop>
<prop key="hibernate.show_sql">true</prop>
</props>
</property>
</bean>
```

<!-- 配置transactionManager, 注意这里引入的包的不同 -->

```
<bean id="transactionManager"
class="org.springframework.orm.hibernate3.HibernateTransactionManager">
<property name="sessionFactory">
<ref local="sessionFactory" />
</property>
</bean>
```

<!--事务代理在这里配置, 这里省略了 -->

```
<bean id="leadDAO" class="com.sparkcrm.schema.dao.LeadDao">
<property name="sessionFactory">
<ref local="sessionFactory" />
</property>
</bean>
</beans>
```

## 一个示例的hibernate的映射文件

```
<?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 package="com.sparkcrm.schema.entities">

<!-- 我在这里用了hibernate的动态模型(dynamic models), 没用pojo-->
    <class entity-name="Lead" table="Lead">
        <id name="id" column="id" type="string">
            <generator class="uuid.hex"/>
        </id>
        <property name="companyName" type="string"/>
        <property name="topic" type="string"/>
        <property name="contactName" type="string"/>
    </class>
</hibernate-mapping>
```

## DAO代码:

```
import java.util.Map;
/**
 * DAO接口
 */
public interface IDAO {
    String create(Map<String, Object> map);
    void update(Map<String, Object> map);
    Map<String, Object> delete(String id);
}
```

```

    boolean share(String id, String userId, int rights);

    boolean assign(String id, String userId);
}

import java.util.Map;

import org.springframework.orm.hibernate3.support.HibernateDaoSupport;

import com.sparkcrm.schema.IDAO;
import com.sparkcrm.schema.Schema;
import com.sparkcrm.schema.metadata.Lead;
/**
 *一个示例的DAO实现, 继承HibernateDaoSupport, 用spring带来的管理
 session等的便利
 */
public class LeadDao extends HibernateDaoSupport implements IDAO ... {
    public String create(Map<String, Object> map) ... {
        getHibernateTemplate().saveOrUpdate(Schema.LEAD, map);
        return (String) map.get(Lead.ID);
    }

    public void update(Map<String, Object> map) ... {

    }

    public Map<String, Object> delete(String id) ... {
        return null;
    }

    public boolean share(String id, String userId, int rights) ... {
        return false;
    }

    public boolean assign(String id, String userId) ... {
        return false;
    }
}

```

```
}  
}
```

示意性的测试代码:

```
import java.sql.Timestamp;  
import java.util.Date;  
import java.util.HashMap;  
import java.util.Map;  
  
import junit.framework.TestCase;  
  
import org.springframework.context.support.ClassPathXmlApplicationCont  
ext;  
  
import com.sparkcrm.schema.IDAO;  
  
public class testLeadDAO extends TestCase {  
    ClassPathXmlApplicationContext ctx = null;  
  
    public void setUp(){  
        ctx = new ClassPathXmlApplicationContext("applicationContext.xml"  
);  
    }  
  
    public void testCreateLead(){  
        IDAO leadDao = (IDAO) ctx.getBean("leadDAO");  
  
        Map<String, Object> map = new HashMap<String, Object>();  
        map.put("companyName", "Spark Ltd.");  
        map.put("topic", "This is a Good Lead!");  
        map.put("contactName", "abcd");  
  
        String id = leadDao.create(map);  
        System.out.println(id);  
    }  
}
```

Hibernate3支持DetachedCriteria，这是一个非常有意义的特性！我们知道，在常规的Web编程中，有大量的动态条件查询，即用户在网页上面自由选择某些条件，程序根据用户的选择条件，动态生成SQL语句，进行查询。

针对这种需求，对于分层应用程序来说，Web层要传递一个查询的条件列表给业务层对象，业务层对象获得这个条件列表之后，然后依次取出条件，构造SQL语句。这里的一个难点是条件列表用什么来构造？如果使用Map，但是这种方式缺陷很大，Map可以传递的信息非常有限，只能传递name和value，无法传递条件做怎样的条件运算，究竟是大于，小于，like，还是其他的什么，业务层对象必须确切掌握每条entry的条件。因此一旦隐含条件改变，业务层对象的查询方法必须相应修改，但是这种查询条件的改变是隐式的，而不是程序代码约束的，因此非常容易出错。

DetachedCriteria可以解决这个问题，即在web程序员使用DetachedCriteria来构造查询条件，然后将这个DetachedCriteria作为方法调用参数传递给业务层对象。而业务层对象获得DetachedCriteria之后，可以在session范围内直接构造Criteria，进行查询。就此，SQL语句的构造完全被搬离到web层实现，而业务层则

责完成持久化和查询的封装即可，与查询条件构造解耦，非常完美！这恐怕也是以前很多企图在web码中构造HQL语句的人想实现的梦想吧！

示例代码片段如下：

web层程序构造查询条件：

java代码:

```
DetachedCriteria detachedCriteria =  
    DetachedCriteria.forClass(Department.class);  
    detachedCriteria.add(Restrictions.eq("name",  
    "department")).createAlias("employees", "e").add(Restrictions.gt(("e.age",  
    new Integer(20)));
```

Department和Employee是一对多关联，查询条件为：

名称是“department”开发部门；  
部门里面的雇员年龄大于20岁；

业务层对象使用该条件执行查询：

java代码:

---

```
detachedCriteria.getExecutableCriteria(session).list();
```

最大的意义在于，业务层代码是固定不变的，查询条件的构造都在web层完成，业务层只负责在session内执行之。这样代码就可放之四海而皆准，须修改了。

然而Spring和Hibernate3的DetachedCriteria有不足的问题，因此在Spring环境下面使用Hibernate3需要注意：

Spring的HibernateTemplate提供了Hibernate的封装，即通过匿名类实现回调，来保证Session的自动管理和事务的管理。其中核心方法是：

java代码:

```
HibernateTemplate.execute(new HibernateCallback() {  
    public Object doInHibernate(Session session) throws HibernateExceptio  
        ....  
    }  
}
```

回调方法提供了session作为参数，有了session可以自由的使用Hibernate API编程了。使用了spring后，代码修改如下：

web层代码：

java代码:

```
DetachedCriteria detachedCriteria =
    DetachedCriteria.forClass(Department.class);
detachedCriteria.createAlias("employees", "e").add(Restrictions.eq("name",
    "department")).add(Restrictions.gt("e.age", new Integer(20)));
departmentManager.findByCriteria(detachedCriteria);
```

构造detachedCriteria，作为参数传递给  
departmentManager

业务层代码使用spring，DepartmentManager的  
findByCriteria如下：

java代码:

```
public List findByCriteria(final DetachedCriteria detachedCriteria) {
    return (List) getHibernateTemplate().execute(new HibernateCallback() {
        public Object doInHibernate(Session session) throws HibernateException {
            Criteria criteria = detachedCriteria.getExecutableCriteria(session);
            return criteria.list();
        }
    });
}
```

实际上也就是：

java代码:

```
Criteria criteria = detachedCriteria.getExecutableCriteria(session);  
return criteria.list();
```

而已

但是该程序代码执行，会抛出强制类型转换异常

我跟踪了一下spring和Hibernate源代码，原因如下：

spring的HibernateTemplate的execute方法提供的接口具有Session作为参数，但是实际上，默认情况下HibernateTemplate传递给回调接口的session并不是org.hibernate.impl.SessionImpl类，而是SessionImpl的一个Proxy类。之所以替换成为一个Proxy类，HibernateTemplate的注释说明，Proxy提供了一些高级功能，包括自动设置Cachable，Transaction的超时间，Session资源的更积极的关闭等等。

java代码:

---

```
private boolean exposeNativeSession = false;
    ...
```

execute方法内部：

```
Session sessionToExpose = (exposeNativeSession ? session :
    createSessionProxy(session));
```

但是遗憾的是，Hibernate的DetachedCriteria的setExecutableCriteria方法却要求将session参数强制SessionImpl，但是spring传过来的却是一个Proxy类，这就报错了。

java代码:

```
public Criteria getExecutableCriteria(Session session) {
    impl.setSession( (SessionImpl) session ); // 要求SessionImpl，Spring传
    的是Proxy
    return impl;
}
```

解决方法，禁止Spring的HibernateTemplate传递Proxy类，强制要求它传递真实的SessionImpl类，在execute方法增加一个参数，提供参数为true，如下

java代码:

---

```
public List findByCriteria(final DetachedCriteria detachedCriteria) {
return (List) getHibernateTemplate().execute(new HibernateCallback() {
public Object doInHibernate(Session session) throws HibernateException {
Criteria criteria = detachedCriteria.getExecutableCriteria(session);
return criteria.list();
}
}, true);
}
```

## Testing JChord

- 1 , Added ConstantIniter && FreemarkerIniter To JChord
- 2 , RAD : JChord : Explaining;
- 3 , Bought 1.5w Stocks From Pubinfo;
- 4 , Refactoring Eclipse Plugins;
- 5, Added JdbcDAO && Paged By SQL To JChord;
- 6, 设置Tomcat5的server.xml中<connector URIEncoding;
- 7, Review Webwork Doc;
- 8, The Party Meeting In Pubinfo;企业转型;
- 9, Kristy's Training: 网络基础知识;
- 10, Added RunData=>BaseAction && ParamInterceptor
- 11, I have got the <http://www.chineseren.cn> 😊
- 12,(1)webwork中为Action的变量设置初值:

Action代码如下:

```
public class ListCity implements Action {  
    private Page page = new Page();  
  
    private int number;  
}
```

Action配置如下:

```
<action name="listCity" class="com.ebt.action.city.ListCity">  
    <param name="number">10</param>  
    <param name="page.everyPage">10</param>  
    <result name="success">/city/city_list.jsp</result>  
    <result name="exception">/no_object.jsp</result>  
</action>
```

(2)自己系统的配置文件如果是properties文件，可以与其它文件结合，参考Spring参考手册中PropertyPlaceholderConfigurer

如果是其它格式的配置文件，可以参考PropertyPlaceholderConfigurer处理过程。

(3)C3P0 dataSource可提供给JdbcTemplate使用直接执行  
<bean id="dataSource" class="com.mchange.v2.c3p0.C3P0DataSource">

```
method="close">
  <property name="driverClass">
    <value>${driver}</value>
  </property>
  <property name="jdbcUrl">
    <value>${url}</value>
  </property>
  <property name="user">
    <value>${username}</value>
  </property>
  <property name="password">
    <value>${password}</value>
  </property>
  <!-- C3P0属性配置 -->
  <property name="initialPoolSize">
    <value>5</value>
  </property>
  <property name="minPoolSize">
    <value>3</value>
  </property>
  <property name="maxPoolSize">
    <value>15</value>
  </property>
  <property name="checkoutTimeout">
    <value>5000</value>
  </property>
  <property name="maxIdleTime">
    <value>1800</value>
  </property>
  <property name="idleConnectionTestPeriod">
    <value>3000</value>
  </property>
  <property name="acquireIncrement">
    <value>2</value>
  </property>
</bean>
```

```

<bean id="sessionFactory" class="org.springframework
<property name="dataSource"><ref local="dataSource"
<property name="hibernateProperties">
  <props>
    <!-- Miscellaneous Settings -->
    <prop key="hibernate.dialect">org.hibernate.dialect.M
    <prop key="hibernate.order_updates">true</prop>
    <prop key="hibernate.max_fetch_depth">1</prop>
    <prop key="hibernate.default_batch_fetch_size">8</p
    <prop key="hibernate.show_sql">true</prop>
    <prop key="hibernate.use_outer_join">true</prop>
    <prop key="hibernate.jdbc.fetch_size">50</prop>
    <prop key="hibernate.jdbc.batch_size">30</prop>
    <prop key="hibernate.jdbc.use_streams_for_binary">
    <!-- hibernate.statement_cache.size=maximum number
for Interbase ) -->
    <!--<prop key="hibernate.statement_cache.size">0</p
    <!-- 事务管理类型，这里默认使用JDBC Transaction
    <!--<prop
key="hibernate.transaction.factory_class">org.hibernate
->
    <!-- 缓存设置默认是EhCache -->
    <prop key="hibernate.cache.provider_class">org.hibe
    <!-- enable the query cache -->
    <prop key="hibernate.cache.use_query_cache">true<
    <!-- store the second-level cache entries in a more hun
    <prop key="hibernate.cache.use_structured_entries">1
  </props>
</property>

```

(4)用#default#savehistory防止后退清空text文本框; 1

```

<HTML>
<HEAD>
<META NAME="save" CONTENT="history">
<STYLE>
  .saveHistory {behavior:url(#default#savehistory);}
</STYLE>

```

```
</HEAD>  
<BODY>  
<INPUT class=saveHistory type=text id=oPersistInput>  
</BODY>  
</HTML>
```

前一段时间用c3p0作hibernate3的连接池，发现连接数总是很多，并不能用100多个数据库连接，尝试修改了几次配置文件问题依旧，无奈换成解决了。

### c3p0配置

```
<property name="hibernate.connection.provider_class">org.hibernate.conr  
nProvider</property>  
<property name="hibernate.c3p0.max_size">20</property>  
<property name="hibernate.c3p0.min_size">5</property>  
<property name="hibernate.c3p0.timeout">120</property>  
<property name="hibernate.c3p0.max_statements">100</property>  
<property name="hibernate.c3p0.idle_test_period">120</property>  
<property name="hibernate.c3p0.acquire_increment">2</property>
```

### proxool配置

```
<property name="hibernate.connection.provider_class">org.hibernate.conr  
ionProvider</property>  
<property name="hibernate.proxool.pool_alias">xjgame</property>  
<property name="hibernate.proxool.xml">proxool.xml</property>
```

```
<?xml version="1.0" encoding="ISO-8859-1"?>  
<something-else-entirely>  
<proxool>  
  <alias>xjgame</alias>  
  <driver-url>jdbc:oracle:thin:@123.123.123.12:1521:game</driver-url>  
  <driver-class>oracle.jdbc.driver.OracleDriver</driver-class>  
  <driver-properties>  
    <property name="user" value="a"/>  
    <property name="password" value="a"/>  
  </driver-properties>  
</proxool>  
</something-else-entirely>  
</maximum-connection-count>20</maximum-connection-count>
```

```
<house-keeping-test-sql>select CURRENT_DATE from dual</house-keep  
</proxool>  
</something-else-entirely>
```

# **DBCP/Hibernate**

## About Hibernate & DBCP

• [Hibernate](#) 2 includes a limited [DBCP ConnectionProvider](#). Not all features of DBCP were supported and with Hibernate3 this limited implementation was deprecated. The implementation below supports all DBCP features and is a drop in replacement for the default version. This provider can be used on Hibernate 2 & 3 (with some import statements adjustments).

## DBCPConnectionProvider

```
/*
 * Copyright 2004 The Apache Software Foundation.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *     http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */
package org.hibernate.connection;

import java.io.PrintWriter;
import java.io.StringWriter;
```

```
import java.sql.Connection;
import java.sql.SQLException;
import java.util.Iterator;
import java.util.Properties;

import org.apache.commons.dbcp.BasicDataS
ource;
import org.apache.commons.dbcp.BasicDataS
ourceFactory;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFact
ory;
import org.hibernate.HibernateException;
import org.hibernate.cfg.Environment;

/**
 * <p>A connection provider that uses an
Apache commons DBCP connection pool.</p>
 *
 * <p>To use this connection provider set
:<br>
 * <code>hibernate.connection.provider_cl
ass   org.hibernate.connection.DBCPCon
nectionProvider</code></p>
 *
 * <pre>Supported Hibernate properties:
 *   hibernate.connection.driver_class
 *   hibernate.connection.url
 *   hibernate.connection.username
 *   hibernate.connection.password
 *   hibernate.connection.isolation
 *   hibernate.connection.autocommit
 *   hibernate.connection.pool_size
```

\* hibernate.connection (JDBC driver properties)</pre>

\* <br>

\* All DBCP properties are also supported by using the hibernate.dbcp prefix.

\* A complete list can be found on the DBCP configuration page:

\* <a href="configuration.html">http://jakarta.apache.org/commons/dbcp/configuration.html</a>.

\* <br>

\* <pre>Example:

\* hibernate.connection.provider\_class  
org.hibernate.connection.DBCPConnectionPr  
ovider

\* hibernate.connection.driver\_class or  
g.hsqldb.jdbcDriver

\* hibernate.connection.username sa

\* hibernate.connection.password

\* hibernate.connection.url jdbc:hsqldb  
:test

\* hibernate.connection.pool\_size 20

\* hibernate.dbcp.initialSize 10

\* hibernate.dbcp.maxWait 3000

\* hibernate.dbcp.validationQuery selec  
t 1 from dual</pre>

\*

\* <p>More information about configuring/  
using DBCP can be found on the

\* <a href="http://jakarta.apache.org/com  
mons/dbcp/">DBCP website</a>.

\* There you will also find the DBCP wiki  
, mailing lists, issue tracking

```

* and other support facilities</p>
*
* @see org.hibernate.connection.ConnectionProvider
* @author Dirk Verbeeck
*/
public class DBCPConnectionProvider implements ConnectionProvider {

    private static final Log log = LogFactory.getLog(DBCPConnectionProvider.class)
;
    private static final String PREFIX = "hibernate.dbcp.";
    private BasicDataSource ds;

    // Old Environment property for backward-compatibility (property removed in Hibernate3)
    private static final String DBCP_PS_MAXACTIVE = "hibernate.dbcp.ps.maxActive";

    // Property doesn't exist in Hibernate2
    private static final String AUTOCOMMIT = "hibernate.connection.autocommit";

    public void configure(Properties props) throws HibernateException {
        try {
            log.debug("Configure DBCPConnectionProvider");

```

```
        // DBCP properties used to create the BasicDataSource
        Properties dbcpProperties = new Properties();

        // DriverClass & url
        String jdbcDriverClass = props.getProperty(Environment.DRIVER);
        String jdbcUrl = props.getProperty(Environment.URL);
        dbcpProperties.put("driverClassName", jdbcDriverClass);
        dbcpProperties.put("url", jdbcUrl);

        // Username / password
        String username = props.getProperty(Environment.USER);
        String password = props.getProperty(Environment.PASS);
        dbcpProperties.put("username", username);
        dbcpProperties.put("password", password);

        // Isolation level
        String isolationLevel = props.getProperty(Environment.ISOLATION);
        if ((isolationLevel != null) && (isolationLevel.trim().length() > 0))
        {
            dbcpProperties.put("defaultTransactionIsolation", isolationLevel);
        }
    }
}
```

```

    }

    // Turn off autocommit (unless
autocommit property is set)
    String autocommit = props.getProperty(AUTOCOMMIT);
    if ((autocommit != null) && (
autocommit.trim().length() > 0)) {
        dbcpProperties.put("defaultAutoCommit", autocommit);
    } else {
        dbcpProperties.put("defaultAutoCommit", String.valueOf(
Boolean.FALSE));
    }

    // Pool size
    String poolSize = props.getProperty(Environment.POOL_SIZE);
    if ((poolSize != null) && (poolSize.trim().length() > 0)
&& (Integer.parseInt(poolSize) > 0)) {
        dbcpProperties.put("maxActive", poolSize);
    }

    // Copy all "driver" properties into "connectionProperties"
    Properties driverProps = ConnectionProviderFactory.getConnectionProperties(props);
    if (driverProps.size() > 0) {

```

```

        StringBuffer connectionPr
properties = new StringBuffer();
        for (Iterator iter = driv
erProps.keySet().iterator(); iter.hasNext
());) {
            String key = (String)
iter.next();
            String value = driver
Props.getProperty(key);
            connectionProperties.
append(key).append('=').append(value);
            if (iter.hasNext()) {
                connectionPropert
ies.append(';');
            }
        }
        dbcpProperties.put("conne
ctionProperties", connectionProperties.to
String());
    }

    // Copy all DBCP properties r
emoving the prefix
    for (Iterator iter = props.ke
ySet().iterator() ; iter.hasNext() ;) {
        String key = String.value
Of(iter.next());
        if (key.startsWith(PREFIX
)) {
            String property = key
.substring(PREFIX.length());
            String value = props.
getProperty(key);

```

```

        dbcpProperties.put(prop
erty, value);
    }
}

// Backward-compatibility
if (props.getProperty(DBCP_PS
_MAXACTIVE) != null) {
    dbcpProperties.put("poolP
reparedStatements", String.valueOf(Boolea
n.TRUE));
    dbcpProperties.put("maxOp
enPreparedStatements", props.getProperty(
DBCP_PS_MAXACTIVE));
}

// Some debug info
if (log.isDebugEnabled()) {
    log.debug("Creating a DBC
P BasicDataSource with the following DBCP
factory properties:");
    StringWriter sw = new Str
ingWriter();
    dbcpProperties.list(new P
rintWriter(sw, true));
    log.debug(sw.toString());
}

// Let the factory create the
pool
    ds = (BasicDataSource) BasicD
ataSourceFactory.createDataSource(dbcpPro
perties);

```

```

        // The BasicDataSource has lazy initialization
        // borrowing a connection will start the DataSource
        // and make sure it is configured correctly.
        Connection conn = ds.getConnection();
        conn.close();

        // Log pool statistics before continuing.
        logStatistics();
    }
    catch (Exception e) {
        String message = "Could not create a DBCP pool";
        log.fatal(message, e);
        if (ds != null) {
            try {
                ds.close();
            }
            catch (Exception e2) {
                // ignore
            }
            ds = null;
        }
        throw new HibernateException(message, e);
    }
    log.debug("Configure DBCPConnectionProvider complete");

```

```

    }

    public Connection getConnection() throws SQLException {
        Connection conn = null;
        try {
            conn = ds.getConnection();
;
        }
        finally {
            logStatistics();
        }
        return conn;
    }

    public void closeConnection(Connection conn) throws SQLException {
        try {
            conn.close();
        }
        finally {
            logStatistics();
        }
    }

    public void close() throws HibernateException {
        log.debug("Close DBCPConnectionProvider");
        logStatistics();
        try {
            if (ds != null) {
                ds.close();
            }
        }
    }
}

```

```

        ds = null;
    }
    else {
        log.warn("Cannot close DB
CP pool (not initialized)");
    }
}
catch (Exception e) {
    throw new HibernateException(
"Could not close DBCP pool", e);
}
log.debug("Close DBCPConnectionPr
ovider complete");
}

protected void logStatistics() {
    if (log.isInfoEnabled()) {
        log.info("active: " + ds.getN
umActive() + " (max: " + ds.getMaxActive(
) + ") "
                + "idle: " + ds.getNu
mIdle() + "(max: " + ds.getMaxIdle() + ")
");
    }
}
}
}

```

## Hibernate 事务天生适合 Spring AOP

级别: 中级

[Naveen Balani](#), 技术架构师, Webify Solutions

2005 年 9 月 26 日

Naveen Balani 继续他的 Spring 系列，介绍把 Hibernate 事务与 Spring 方面编程（AOP）集成的知识。结果是一个可以依靠的持久性框架。

在这个系列的 [前一期](#)中，我介绍了 Spring 框架的 7 个模块，包括 Spring 和控制反转（IOC）容器。然后我用一个简单的示例演示了 IOC 模式（在 Spring IOC 容器实现）如何用松散耦合的方式集成分散的系统。

现在，我从我上次结束的地方开始，采用与上次类似的示例，演示 Spring 和 Spring Hibernate 持久性支持的声明性事务处理，所以我首先从对这方面的深入研究开始。

[下载这篇文章的源代码](#)。请参阅 [参考资料](#) 访问 Spring 框架和 Apache A 用程序需要它们。

## Spring AOP

软件系统通常由多个组件构成，每个组件负责一个特定的功能领域。但是他们的核心功能之外的额外责任。系统服务（例如日志、事务管理和安全性的组件的领域里，而这些组件的核心职责是其他事情。结果就是所谓的“说“一团糟”。面向方面编程是一种试图解决这个问题的编程技术，它把关程概念。

使用 AOP 时，仍然是在一个地方定义系统的公共功能，但是可以声明性用这个功能。如果对横切关注点（例如日志和事务管理）进行了模块化，类，就可以向代码中添加新特性。这类模块化的关注点称作 方面。

以一个企业应用程序为例。这类应用程序通常要求类似于安全性和事务支持的服务。显然，可以把这些服务的支持直接编写到要求服务的每个类当中，但是更希望能够不必为大量事务性上下文编写同样的事务处理代码。如果使用 Spring AOP 进行事务处理，那么可以声明性地安排适当的方法调用，而不必逐个安排。

您知道么  
可以在任  
服务器中  
而且，还  
能，使其

Spring AOP 提供了几个方面，可以为 JavaBean 声明事务。例如，TransactionProxyFactoryBean 是个方便的代理类，能够拦截对现有类的方法调用，并把事务上下文应用到事务 bean。在下面的示例中会看到这个类的实际应用。

的中心焦  
定 J2EEE  
据访问对  
( Web 或  
JavaBear  
序、测试  
而不会有

---

## Hibernate

Spring 框架提供了对 Hibernate、JDO 和 iBATIS SQL Maps 的集成支持是第一级的，整合了许多 IOC 的方便特性，解决了许多典型的 Hibernate 的支持符合 Spring 通用的事务和数据访问对象 (DAO) 异常。

Spring 为使用选择的 OR 映射层来创建数据访问应用程序提供了支持。[可重用 JavaBean，所以不管选择什么技术，都能以库的格式访问大多数 ApplicationContext 或 BeanFactory 内部的 OR 映射的好处是简化了配置。

Hibernate 是 Java 平台上一个功能全面的、开源的 OR 映射框架。Hibernate 秉承了 Java 理念的持久性类——包括关联、继承、多态、复合以及 Java 集合 (HQL) 被设计成 SQL 的一个微型面向对象扩展，它是对象和关系世界支持用原始 SQL 或基于 Java 的标准和示例查询表达查询。Hibernate 使物件把 Java 类映射到表，把 JavaBean 属性映射到数据库表。

通过 JDBC 技术，支持所有的 SQL 数据库管理系统。Hibernate 与所有应用服务器和 Web 容器都很好地集成。

---

## 实际示例

一个银行应用程序示例可以让您自己看到 Spring AOP 和 Hibernate 一起工作。该示例允许用户 (Customer) 在一个事务中打开一个或多个银行帐户。用户可以指定是支票帐户类型或者是储蓄帐户类型。

应用程序数据库 (Cloudscape?) 容纳所有客户和帐户信息。在这个例子

Account 类之间存在 1:N 的关联。在实际生活场景中，关联可能需要按  $n$  户。

由于用户必须可以在一个事务中申请多个帐户，所以首先要为数据库交互设置 Spring AOP 的 TransactionProxyFactoryBean，让它拦截方法调用应用到 DOA。

---

## Hibernate 实践

在 Spring 框架中，像 JDBC DataSource 或 Hibernate SessionFactory 这文中可以用 bean 实现。需要访问资源的应用程序对象只需通过 bean 引例的引用即可（这方面的更多内容在 [下一节中](#)）。在清单 1 中，可以看到摘录：XML 应用程序上下文定义显示了如何设置 JDBC DataSource，并 SessionFactory。

### 清单 1. JDBC DataSource 和 HibernateSessionFactory 连接

```
<!-- DataSource Property -->
<bean id="exampleDataSource"
      class="org.apache.commons.dbcp.BasicDataSource">
  <property name="driverClassName">
    <value>org.apache.derby.jdbc.EmbeddedDriver</value>
  </property>
  <property name="url">
    <value>jdbc:derby:springexample;create=true</value>
  </property>
</bean>

<!-- Database Property -->
<bean id="exampleHibernateProperties"
      class="org.springframework.beans.factory.config.Pro
  <property name="properties">
    <props>
```

```
<prop key="hibernate.hbm2ddl.auto">update</prop>
<prop
  key="hibernate.dialect">net.sf.hibernate.dialect
<prop
  key="hibernate.query.substitutions">>true 'T', fa
<prop key="hibernate.show_sql">>false</prop>
<prop key="hibernate.c3p0.minPoolSize">5</prop>
<prop key="hibernate.c3p0.maxPoolSize">20</prop>
<prop key="hibernate.c3p0.timeout">600</prop>
<prop key="hibernate.c3p0.max_statement">50</prop>
<prop
  key="hibernate.c3p0.testConnectionOnCheckout">f
</props>
</property>
</bean>
```

```
<!-- Hibernate SessionFactory -->
<bean id="exampleSessionFactory"
  class="org.springframework.orm.hibernate.LocalSes
<property name="dataSource">
  <ref local="exampleDataSource"/>
</property>
<property name="hibernateProperties">
  <ref bean="exampleHibernateProperties" />
</property>
<!-- OR mapping files. -->
<property name="mappingResources">
  <list>
    <value>Customer.hbm.xml</value>
    <value>Account.hbm.xml</value>
  </list>
</property>
</bean>
```

---

[清单 1](#) 显示了如何为示例应用程序数据库（是 Cloudscape）配置数据源（exampleDataSource）。exampleDataSource 被连接到 Spring Hibernate \*.hbm.xml 指定了示例应用程序的 OR 映射文件。

数据源和会话工厂设置好之后，下一步就是在 DAO 中连接，在 CustomerSessionFactory。接下来，插入 Spring 的 TransactionProxyFactoryBean CustomerDAOImpl 对象的方法调用，并声明性地在它上面应用事务。

在 [清单 2](#) 的这个示例中，CustomerDAOImpl 类的 addCustomer 方法是作为一个事务属性 PROPAGATION\_REQUIRED。这个属性等价于 EJB 容器的 TX\_REQ 一直在事务中运行，可以使用 PROPAGATION\_REQUIRED。如果事务已经在这事务，否则 Spring 的轻量级事务管理器会启动一个事务。如果想在调用事务，可以使用 PROPAGATION\_REQUIRES\_NEW 属性。

应用程序的连接完成之后，现在来进一步查看源代码。

---

## 分析这个!

如果以前没这么做过，那么请 [下载这篇文章的源代码](#)。把源 zip 文件释放上，例如 c:\。会创建一个叫作 SpringProjectPart2 的文件夹。src\spring 的 Hibernate 映射文件和 Spring 配置文件。src\springexample\hibernate 代码。

在这里会发现两个类，即 Customer 和 Account，它们用 Hibernate 映射文表。Customer 类代表客户信息，Account 代表客户的帐户信息。正如前面照 1: N 关系进行建模，即一个 Customer 可以拥有多个 Account。清单 3 Hibernate 映射文件。

### 清单 3. Customer 对象的 Hibernate 映射文件

---

```
<?xml version="1.0"?>
```

```
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 2.0//EN
    "http://hibernate.sourceforge.net/hibernate
<hibernate-mapping>
  <class
    name="springexample.hibernate.Customer"
    table="TBL_CUSTOMER"
    dynamic-update="false"
    dynamic-insert="false">

    <id
      name="id"
      column="CUSTOMER_ID"
      type="java.lang.Long"
      unsaved-value="-1"
    >
      <generator class="native">
      </generator>
    </id>

    <set name = "accounts"
      inverse = "true"
      cascade="all-delete-orphan">
      <key column = "CUSTOMER_ID"/>
      <one-to-many class="springexample.hibernate

    </set>

    <property
      name="email"
      type="string"
      update="false"
      insert="true"
```

```
        column="CUSTOMER_EMAIL"  
        length="82"  
        not-null="true"  
  
    />  
  
    <property  
        name="password"  
        type="string"  
        update="false"  
        insert="true"  
        column="CUSTOMER_PASSWORD"  
        length="10"  
        not-null="true"  
  
    />  
  
    <property  
        name="userId"  
        type="string"  
        update="false"  
        insert="true"  
        column="CUSTOMER_USERID"  
        length="12"  
        not-null="true"  
        unique="true"  
  
    />  
  
    <property  
        name="firstName"  
        type="string"  
        update="false"  
        insert="true"  
        column="CUSTOMER_FIRSTNAME"  
        length="25"  
        not-null="true"
```

```

    />
    <property
        name="lastName"
        type="string"
        update="false"
        insert="true"
        column="CUSTOMER_LASTTTNAME"
        length="25"
        not-null="true"

    />

</class>

</hibernate-mapping>

```

set name="accounts" 和一对多类标签指定了 Customer 和 Account 之间的映射。*Account.hbm.xml* 文件中定义了 Account 对象的映射。

CustomerDAOImpl.java 代表应用程序的 DAO，它在应用程序数据库中插入信息。CustomerDAOImpl 扩展了 Spring 的 HibernateDaoSupport，它用 Spring 的会话管理。这样，可以通过 getHibernateTemplate() 方法保存或检索数据。getCustomerAccountInfo() 对 Customer 进行查找，通过 getHibernateTemplate() 得到客户的帐户信息，如清单 4 所示。

#### 清单 4. DAO 实现

```

public class CustomerDAOImpl extends HibernateDaoSupport
    implements CustomerDAO{

```

```
public void addCustomer(Customer customer) {
    getHibernateTemplate().save(customer);
    // TODO Auto-generated method stub

}

public Customer getCustomerAccountInfo(Customer cu
Customer cust = null;
List list = getHibernateTemplate().find("from Cu
    "where customer.userId = ?" ,
    customer.getUserId(), Hibernate.STRING);

if(list.size() > 0){
    cust = (Customer) list.get(0);
}

return cust;

}
```

所有这些都应当很容易掌握。现在来看代码的实际应用!

---

## 运行应用程序

要运行示例应用程序，必须首先 [下载 Spring 框架](#) 和它的全部依赖文件。位置（比如 c:\），这会创建文件夹 C:\spring-framework-1.2-rc2（针对：还必须下载和释放 [Apache Ant](#) 和 [Cloudscape](#)。下载 Cloudscape 之后建文件夹 C:\Cloudscape\_10.0。

接下来，释放源代码到 c:\，这会创建 SpringProject2 文件夹。接下来修用实际安装 Spring 的位置代替 C:\spring-framework-1.2-rc2，用实际安

C:\Program Files\IBM\Cloudscape\_10.0。

打开命令行提示符，进入 *SpringProject* 目录，在命令行提示符下输入以这会构建并运行 *CreateBankCustomerClient* 类，它会创建 *Customer* 类对 *Account* 对象，填充它，并把它添加到 *Customer* 对象。

然后 *CreateBankCustomerClient* 会调用 *CustomerDAOImpl.addCustomer* 类且插入完成，*CreateBankCustomerClient* 会调用 *CustomerDAOImpl.getCus* 据 *userid* 得到客户和帐户信息。如果 *CreateBankCustomerClient* 执行成出 *userid*。也可以查询 *Cloudscape* 数据库检索客户和帐户信息。

---

## 结束语

在三部分的 *Spring* 系列的第 2 部分中，我介绍了如何集成 *Spring Hiber* 是一个强健的持久性框架，支持声明性的实现事务。

在这个系列的下一篇，也是最后一篇文章中，我将介绍 *Spring* 的 *MVC* 和基于 *Web* 的应用程序的创建。

---

## 下载

描述	名字
Example source code, spring files, build scripts	wa-spring2-SpringProje

→ [关于下载方法的信息](#)

→ [获取 Adobe? Reader?](#)

---

## 参考资料

### 学习

- 您可以参阅本文在 developerWorks 全球站点上的 [英文原文](#)。
- 虽然 Spring AOP 提供了独特的优势，但它并不是惟一的 AOP 实现。请参见 [AOP 工具比较，第 1 部分](#) 了解 Spring AOP 的构成（*developerWorks*）。
- 通过“[无需容器的对象关系映射](#)”（*developerWorks*，2004 年 4 月）AOP 开发事务性持久性层。
- [Web 架构专区](#) 的文章专门涉及各种基于 Web 的解决方案。

### 获得产品和技术

- 从 [Spring 主页](#) 下载 Spring 框架。
- 从 [Ant 主页](#) 下载 Apache Ant。

### 讨论

- 通过参与 [developerWorks blogs](#) 加入 developerWorks 社区。

## 【spring+hibernate学习文档】---配置篇

```
-----web.xml-----
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app
xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-
app_2_4.xsd" version="2.4">
  <display-name>framework</display-name>

  <description>framework sample
application</description>

  <!--
    - Key of the system property that should
specify the root directory of this
    - web app. Applied by WebAppRootListener or
Log4jConfigListener.
  -->
  <context-param>
    <param-name>webAppRootKey</param-
name>
    <param-value>framework.root</param-value>
  </context-param>

  <!--
    - Location of the Log4J config file, for
initialization and refresh checks.
    - Applied by Log4jConfigListener.
  -->
  <context-param>
    <param-name>log4jConfigLocation</param-
name>
    <param-value>/WEB-
INF/log4j.properties</param-value>
```

```

</context-param>

    <filter>
        <filter-name>Set Character
Encoding</filter-name>
        <filter-
class>cn.rhui.framework.common.filter.SetCharacte
class>
        <init-param>
            <param-name>encoding</param-
name>
            <param-value>GBK</param-value>
        </init-param>
    </filter>

    <filter-mapping>
        <filter-name>Set Character
Encoding</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>

<context-param>
    <param-name>contextConfigLocation</param-
name>
    <param-value>/WEB-
INF/applicationContext.xml
    /WEB-INF/dataAccessContext-local.xml
    </param-value>
</context-param>

<!--
- Configures Log4J for this web app.
- As this context specifies a context-param
"log4jConfigLocation", its file path
- is used to load the Log4J configuration,
including periodic refresh checks.
-

```

- Would fall back to default Log4J initialization (non-refreshing) if no special context-params are given.
- 
- Exports a "web app root key", i.e. a system property that specifies the root directory of this web app, for usage in log file paths.
- This web app specifies "petclinic.root" (see log4j.properties file).

-->

<!-- Leave the listener commented-out if using JBoss -->

```
<listener>
<listener-
class>org.springframework.web.util.Log4jConfigList
class>
</listener>
```

```
<servlet>
<servlet-name>context</servlet-name>
<servlet-
class>org.springframework.web.context.ContextLoa
class>
<load-on-startup>1</load-on-startup>
</servlet>
```

```
<servlet>
<servlet-name>action</servlet-name>
<servlet-
class>org.apache.struts.action.ActionServlet</servl
class>
```

```
<init-param>
<param-name>config</param-name>
<param-value>/WEB-INF/struts-
```

```

config.xml</param-value>
</init-param>

    <init-param>
        <param-
name>config/workflow</param-name>
        <param-value>/WEB-
INF/workflow/struts-config.xml</param-value>
    </init-param>

    <init-param>
        <param-
name>config/testflow</param-name>
        <param-value>/WEB-
INF/testflow/struts-config.xml</param-value>
    </init-param>
    <load-on-startup>2</load-on-startup>
</servlet>

<!--
- Maps the petclinic dispatcher to *.htm. All
handler mappings in
- petclinic-servlet.xml will by default be applied
to this subpath.
- If a mapping isn't a /* subpath, the handler
mappings are considered
- relative to the web app root.
-
- NOTE: A single dispatcher can be mapped to
multiple paths, like any servlet.
-->
<servlet-mapping>
    <servlet-name>action</servlet-name>
    <url-pattern>*.do</url-pattern>
</servlet-mapping>

<session-config>
    <session-timeout>30</session-timeout>

```

```
</session-config>
<welcome-file-list>
  <!-- Redirects to "welcome.htm" for dispatcher
handling -->
  <!--welcome-file>index.jsp</welcome-file-->
  <welcome-file>default.jsp</welcome-file>
</welcome-file-list>

<error-page>
  <exception-
type>java.lang.Exception</exception-type>
  <!-- Displays a stack trace -->
  <location>/WEB-
INF/jsp/uncaughtException.jsp</location>
</error-page>
<jsp-config>
  <taglib>
    <taglib-uri>/WEB-INF/page-
tag.tld</taglib-uri>
    <taglib-location>/WEB-INF/tld/page-
tag.tld</taglib-location>
  </taglib>

  <taglib>
    <taglib-uri>core</taglib-uri>
    <taglib-location>/WEB-
INF/tld/c.tld</taglib-location>
  </taglib>

  <taglib>
    <taglib-uri>/WEB-INF/tld/struts-
bean.tld</taglib-uri>
    <taglib-location>/WEB-INF/tld/struts-
bean.tld</taglib-location>
  </taglib>

  <taglib>
```

```

    <taglib-uri>/WEB-INF/tld/struts-html.tld</taglib-
uri>
    <taglib-location>/WEB-INF/tld/struts-
html.tld</taglib-location>
    </taglib>

    <taglib>
    <taglib-uri>/WEB-INF/tld/struts-logic.tld</taglib-
uri>
    <taglib-location>/WEB-INF/tld/struts-
logic.tld</taglib-location>
    </taglib>
    <taglib>
    <taglib-uri>t</taglib-uri>
    <taglib-location>/WEB-INF/tld/t.tld</taglib-
location>
    </taglib>
</jsp-config>

<!--
- Reference to Petclinic database.
- Only needed if not using a local DataSource
but a JNDI one instead.
-->
<!--
<resource-ref>
    <res-ref-name>jdbc/framework</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
    <res-auth>Container</res-auth>
</resource-ref>
-->
</web-app>
-----applicationContext.xml-----
--
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD

```

```

BEAN//EN"
"http://www.springframework.org/dtd/spring-
beans.dtd">
<!--
- Application context definition for framework.
-->
<beans>

<!-- =====
RESOURCE DEFINITIONS
===== -->

<!-- Configurer that replaces ${...} placeholders
with values from a properties file -->
<!-- (in this case, JDBC-related settings for the
dataSource definition below) -->
<bean id="propertyConfigurer"
class="org.springframework.beans.factory.config.Pr
<property name="location"><value>/WEB-
INF/jdbc.properties</value></property>
</bean>

<!-- Transaction manager for a single Hibernate
SessionFactory (alternative to JTA) -->
<bean id="transactionManager"
class="org.springframework.orm.hibernate3.Hiberna
<property name="sessionFactory"><ref
bean="sessionFactory"/></property>
</bean>

<!-- Transaction manager that delegates to JTA
(for a transactional JNDI DataSource) -->
<!--
<bean id="transactionManager"
class="org.springframework.transaction.jta.JtaTrans
-->
<!-- id generator -->

```

```

<bean id="idGenerateService"
parent="baseTransactionProxyFactoryBean">
  <property name="target">
    <bean
class="cn.rhui.framework.idgenerator.service.IdGer
autowire="byName"/>
  </property>
</bean>
<bean id="generate"
class="cn.rhui.framework.idgenerator.Generator">
  <property name="idService">
    <ref local="idGenerateService"></ref>
  </property>
  <property name="path">
<value>${generator.path}</value></property>
</bean>
<!-- end id generator -->
<!--
- Transactional proxy for Petclinic's central data
access object.
-
- Defines specific transaction attributes with
"readOnly" markers,
- which is an optimization that is particularly
valuable with Hibernate
- (to suppress unnecessary flush attempts for
read-only operations).
-
- Note that in a real-life app with multiple
transaction proxies,
- you will probably want to use parent and child
bean definitions
- as described in the manual, to reduce
duplication.
-->
<bean id="baseTransactionProxyFactoryBean"
class="org.springframework.transaction.interceptor.

```

```

abstract="true">
  <property name="transactionManager"><ref
local="transactionManager"/></property>
  <property name="transactionAttributes">
    <props>
      <prop
key="find*">PROPAGATION_REQUIRED,readOnly
      <prop
key="select*">PROPAGATION_REQUIRED,readOnly
      <prop
key="get*">PROPAGATION_REQUIRED,readOnly
      <prop
key="remove*">PROPAGATION_REQUIRED</prop>
      <prop
key="save*">PROPAGATION_REQUIRED</prop>
      <prop
key="create*">PROPAGATION_REQUIRED</prop>
    </props>
  </property>
</bean>
</beans>

```

-----dataAccessContext-local.xml-----

```

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

  <!-- ===== owners
bean ===== -->
  <!-- Local DataSource that works in any
environment -->
  <!-- Note that DriverManagerDataSource does
not pool; it is not intended for production -->
  <!-- See JPetStore for an example of using

```

Commons DBCP BasicDataSource as  
alternative -->

<!-- See Image Database for an example of  
using C3P0 ComboPooledDataSource as  
alternative -->

```
<bean id="dataSource"  
class="org.springframework.jdbc.datasource.Driverl  
  <property name="driverClassName">  
  <value>${jdbc.driverClassName}</value>  
  </property>  
  <property name="url"><value>${jdbc.url}</value></property>  
  <property name="username">  
  <value>${jdbc.username}</value></property>  
  <property name="password">  
  <value>${jdbc.password}</value></property>  
</bean>
```

<!-- JNDI DataSource for J2EE environments --  
>

```
<!--  
<bean id="dataSource"  
class="org.springframework.jndi.JndiObjectFactoryl  
  <property name="jndiName">  
  <value>java:comp/env/jdbc/petclinic</value>  
  </property>  
</bean>  
-->
```

<!-- Hibernate SessionFactory -->

```
<bean id="sessionFactory"  
class="org.springframework.orm.hibernate3.LocalS  
  <property name="dataSource"><ref  
local="dataSource"/></property>  
  <property name="schemaUpdate">  
  <value>>true</value>  
  </property>  
  <property name="mappingResources">
```

```
<list>
  <value>cn/rhui/framework/test/domain/Owners.hi
  <value>cn/rhui/framework/idgenerator/vo/SysG
</list>
</property>
<property name="hibernateProperties">
  <props>
    <prop
key="hibernate.query.factory_class">org.hibernate.l
    <prop
key="hibernate.dbcp.maxActive">100</prop>
    <prop
key="hibernate.dbcp.whenExhaustedAction">1</pr
    <prop
key="hibernate.dbcp.maxWait">120000</prop>
    <prop
key="hibernate.dbcp.maxIdle">10</prop>
    <prop
key="hibernate.dbcp.ps.maxActive">100</prop>
    <prop
key="hibernate.dbcp.ps.whenExhaustedAction">1<
    <prop
key="hibernate.dbcp.ps.maxWait">120000</prop>
    <prop
key="hibernate.dbcp.ps.maxIdle">10</prop>
    <!--prop
key="hibernate.dbcp.validationQuery">select 1
from AclUser</prop-->
    <prop
key="hibernate.dbcp.testOnBorrow">true</prop>
    <prop
key="hibernate.dbcp.testOnReturn">false</prop>
    <prop
key="hibernate.dialect">${hibernate.dialect}
</prop>
    <prop
key="hibernate.connection.pool_size">10</prop>
```

```
<prop
key="hibernate.cache.use_query_cache">true</prop>
<!--prop
key="hibernate.cache.provider_class">net.sf.ehcache
-->
<prop
key="hibernate.use_outer_join">true</prop>
<prop
key="hibernate.max_fetch_depth">3</prop>
<prop
key="hibernate.jdbc.fetch_size">100</prop>
<prop
key="hibernate.jdbc.batch_size">30</prop>
<prop
key="hibernate.default_batch_fetch_size">50</prop>
<prop key="hibernate.show_sql">true</prop>
<prop
key="hibernate.query.substitutions">true 1, false
0, yes 'Y', no 'N'</prop>
</props>
</property>
</bean>

<bean id="ownersDao"
class="cn.rhui.framework.test.dao.impl.OwnersImpl"
autowire="byName"/>
</beans>
```

本文引用通告地址：

<http://blog.csdn.net/lovechineseboy/services/trackback>

## 如何把Hibernate2.1升级到Hibernate3.0 ?

选自<<精通Hibernate：Java对象持久化技术详解>> 作者：孙卫琴 来源：  
:www.javathinker.org  
如果转载，请标明出处，谢谢

- 1.1 Hibernate API 变化
  - 1.1.1 包名
  - 1.1.2 org.hibernate.classic包
  - 1.1.3 Hibernate所依赖的第三方软件包
  - 1.1.4 异常模型
  - 1.1.5 Session接口
  - 1.1.6 createSQLQuery()
  - 1.1.7 Lifecycle 和 Validatable 接口
  - 1.1.8 Interceptor接口
  - 1.1.9 UserType和CompositeUserType接口
  - 1.1.10 FetchMode类
  - 1.1.11 PersistentEnum类
  - 1.1.12 对Blob 和Clob的支持
  - 1.1.13 Hibernate中供扩展的API的变化
- 1.2 元数据的变化
  - 1.2.1 检索策略
  - 1.2.2 对象标识符的映射
  - 1.2.3 集合映射
  - 1.2.4 DTD
- 1.3 查询语句的变化
  - 1.3.1 indices()和elements()函数

尽管Hibernate 3.0 与Hibernate2.1的源代码是不兼容的，但是当Hibernate开发小组在设计Hibernate3.0时，为简化升级Hibernate版本作了周到的考虑。对于现有的基于Hibernate2.1的Java项目，可以很方便的把它升级到Hibernate3.0。

本文描述了Hibernate3.0版本的新变化，Hibernate3.0版本的变化包括三个方面：

- (1) API的变化，它将影响到Java程序代码。
- (2) 元数据，它将影响到对象-关系映射文件。
- (3) HQL查询语句。

值得注意的是，Hibernate3.0并不会完全取代Hibernate2.1。在同一个应

用程序中，允许Hibernate3.0和Hibernate2.1并存。

## 1.1 Hibernate API 变化

### 1.1.1 包名

Hibernate3.0的包的根路径为：“org.hibernate”，而在Hibernate2.1中为“net.sf.hibernate”。这一命名变化使得Hibernate2.1和Hibernate3.0能够同时运行在同一个应用程序中。

如果希望把已有的应用升级到Hibernate3.0，那么升级的第一步是把Java源程序中的所有“net.sf.hibernate”替换为“org.hibernate”。

Hibernate2.1中的“net.sf.hibernate.expression”包被改名为“org.hibernate.criterion”。假如应用程序使用了Criteria API，那么在升级的过程中，必须把Java源程序中的所有“net.sf.hibernate.expression”替换为“org.hibernate.criterion”。

如果应用使用了除Hibernate以外的其他外部软件，而这个外部软件又引用了Hibernate的接口，那么在升级时必须十分小心。例如EHCache拥有自己的CacheProvider：`net.sf.ehcache.hibernate.Provider`，在这个类中引用了Hibernate2.1中的接口，在升级应用时，可以采用以下办法之一来升级EHCache：

- (1) 手工修改`net.sf.ehcache.hibernate.Provider`类，使它引用Hibernate3.0中的接口。
- (2) 等到EHCache软件本身升级为使用Hibernate3.0后，使用新的EHCache软件。
- (3) 使用Hibernate3.0中内置的CacheProvider：`org.hibernate.cache.EhCacheProvider`。

### 1.1.2 org.hibernate.classic包

Hibernate3.0把一些被废弃的接口都转移到org.hibernate.classic中。

### 1.1.3 Hibernate所依赖的第三方软件包

在Hibernate3.0的软件包的lib目录下的README.txt文件中，描述了Hibernate3.0所依赖的第三方软件包的变化。

### 1.1.4 异常模型

在Hibernate3.0中，HibernateException异常以及它的所有子类都继承了java.lang.RuntimeException。因此在编译时，编译器不会再检查HibernateException。

### 1.1.5 Session接口

在Hibernate3.0中，原来Hibernate2.1的Session接口中的有些基本方法也被废弃，但为了简化升级，这些方法依然是可用的，可以通过org.hibernate.classic.Session子接口来访问它们，例如：

```
org.hibernate.classic.Session session=sessionFactory.openSession();
    session.delete("delete from Customer ");
```

在Hibernate3.0中，org.hibernate.classic.Session接口继承了org.hibernate.Session接口，在org.hibernate.classic.Session接口中包含了一系列被废弃的方法，如find()、iterate()等。SessionFactory接口的openSession()方法返回org.hibernate.classic.Session类型的实例。如果希望在程序中完全使用Hibernate3.0，可以采用以下方式创建Session实例：

```
org.hibernate.Session session=sessionFactory.openSession();
```

如果是对已有的程序进行简单的升级，并且希望仍然调用Hibernate2.1中Session的一些接口，可以采用以下方式创建Session实例：

```
org.hibernate.classic.Session session=sessionFactory.openSession();
```

在Hibernate3.0中，Session接口中被废弃的方法包括：

- \* 执行查询的方法：find()、iterate()、filter()和delete(String hqlSelectQuery)
- \* saveOrUpdateCopy()

Hibernate3.0一律采用createQuery()方法来执行所有的查询语句，采用DELETE 查询语句来执行批量删除，采用merge()方法来替代 saveOrUpdateCopy()方法。

提示：在Hibernate2.1中，Session的delete()方法有几种重载形式，其中参数为HQL查询语句的delete()方法在Hibernate3.0中被废弃，而参数为Object类型的delete()方法依然被支持。delete(Object o)方法用于删除参数指定的对象，该方法支持级联删除。

Hibernate2.1没有对批量更新和批量删除提供很好的支持，参见<<精通Hibernate>>一书的第13章的13.1.1节（批量更新和批量删除），而Hibe

Hibernate3.0对批量更新和批量删除提供了支持，能够直接执行批量更新或批量删除语句，无需把被更新或删除的对象先加载到内存中。以下是通过Hibernate3.0执行批量更新的程序代码：

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();
```

```
String hqlUpdate = "update Customer set name = :newName where name =
                    :oldName";
int updatedEntities = s.createQuery( hqlUpdate )
    .setString( "newName", newName )
    .setString( "oldName", oldName )
    .executeUpdate();
tx.commit();
session.close();
```

以下是通过Hibernate3.0执行批量删除的程序代码：

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

String hqlDelete = "delete Customer where name = :oldName";
int deletedEntities = s.createQuery( hqlDelete )
    .setString( "oldName", oldName )
    .executeUpdate();
tx.commit();
session.close();
```

### 1.1.6 createSQLQuery()

在Hibernate3.0中，Session接口的createSQLQuery()方法被废弃，被移到org.hibernate.classic.Session接口中。Hibernate3.0采用新的SQLQuery接口来完成相同的功能。

### 1.1.7 Lifecycle 和 Validatable 接口

Lifecycle和Validatable 接口被废弃，并且被移到org.hibernate.classic包中。

### 1.1.8 Interceptor接口

在Interceptor 接口中加入了两个新的方法。用户创建的Interceptor实现

类在升级的过程中，需要为这两个新方法提供方法体为空的实现。此外，`instantiate()`方法的参数作了修改，`isUnsaved()`方法被改名为`isTransient()`。

### 1.1.9 UserType和CompositeUserType接口

在`UserType`和`CompositeUserType`接口中都加入了一些新的方法，这两个接口被移到`org.hibernate.usertype`包中，用户定义的`UserType`和`CompositeUserType`实现类必须实现这些新方法。

Hibernate3.0提供了`ParameterizedType`接口，用于更好的重用用户自定义的类型。

### 1.1.10 FetchMode类

`FetchMode.LAZY` 和 `FetchMode.EAGER`被废弃。取而代之的分别为`FetchMode.SELECT` 和`FetchMode.JOIN`。

### 1.1.11 PersistentEnum类

`PersistentEnum`被废弃并删除。已经存在的应用应该采用`UserType`来处理枚举类型。

### 1.1.12 对Blob 和Clob的支持

Hibernate对`Blob`和`Clob`实例进行了包装，使得那些拥有`Blob`或`Clob`类型的属性的类的实例可以被游离、序列化或反序列化，以及传递到`merge()`方法中。

### 1.1.13 Hibernate中供扩展的API的变化

`org.hibernate.criterion`、`org.hibernate.mapping`、`org.hibernate.persister`和`org.hibernate.collection` 包的结构和实现发生了重大的变化。多数基于Hibernate

2.1 的应用不依赖于这些包，因此不会被影响。如果你的应用扩展了这些包中的类，那么必须非常小心的对受影响的程序代码进行升级。

## 1.2 元数据的变化

### 1.2.1 检索策略

在Hibernate2.1中，`lazy`属性的默认值为“false”，而在Hibernate3.0中，`la`

zy属性的默认值为“true”。在升级映射文件时，如果原来的映射文件中的有关元素，如<set>、<class>等没有显式设置lazy属性，那么必须把它们都显式的设置为lazy=“true”。如果觉得这种升级方式很麻烦，可以采取另一简单的升级方式：在<hibernate-mapping>元素中设置: default-lazy=“false”。

### 1.2.2 对象标识符的映射

unsaved-value属性是可选的，在多数情况下，Hibernate3.0将把unsaved-value="0" 作为默认值。

在Hibernate3.0中，当使用自然主键和游离对象时，不再强迫实现Interceptor.isUnsaved()方法。如果没有设置这个方法，当Hibernate3.0无法区分对象的状态时，会查询数据库，来判断这个对象到底是临时对象，还是游离对象。不过，显式的使用Interceptor.isUnsaved()方法会获得更好的性能，因为这可以减少Hibernate直接访问数据库的次数。

### 1.2.3 集合映射

<index>元素在某些情况下被<list-index>和<map-key>元素替代。此外，Hibernate3.0用<map-key-many-to-many> 元素来替代原来的<key-many-to-many>.元素，用<composite-map-key>元素来替代原来的<composite-index>元素。

### 1.2.4 DTD

对象-关系映射文件中的DTD文档，由原来的：

<http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd>

改为：

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

## 1.3 查询语句的变化

Hibernate3.0 采用新的基于ANTLR的HQL/SQL查询翻译器，不过，Hibernate2.1的查询翻译器也依然存在。在Hibernate的配置文件中，hibernate.query.factory\_class属性用来选择查询翻译器。例如：

(1) 选择Hibernate3.0的查询翻译器：

```
hibernate.query.factory_class= org.hibernate.hql.ast.ASTQueryTranslatorFactory
```

(2) 选择Hibernate2.1的查询翻译器

```
hibernate.query.factory_class= org.hibernate.hql.classic.ClassicQueryTranslatorFactory
```

提示：ANTLR是用纯Java语言编写出来的一个编译工具，它可生成Java语言或者是C++的词法和语法分析器，并可产生语法分析树并对该树进行遍历。ANTLR由于是纯Java的，因此可以安装在任意平台上，但是需要JDK的支持。

Hibernate开发小组尽力保证Hibernate3.0的查询翻译器能够支持Hibernate2.1的所有查询语句。不过，对于许多已经存在的应用，在升级过程中，也不妨仍然使用Hibernate2.1的查询翻译器。

值得注意的是，Hibernate3.0的查询翻译器存在一个Bug：不支持某些theta-style连结查询方言：如Oracle8i的OracleDialect方言、Sybase11Dialect。解决这一问题的办法有两种：（1）改为使用支持ANSI-style连结查询的方言，如Oracle9Dialect，（2）如果升级的时候遇到这一问题，那么还是改为使用Hibernate2.1的查询翻译器。

### 1.3.1 indices()和elements()函数

在HQL的select子句中废弃了indices()和elements()函数，因为这两个函数的语法很让用户费解，可以用显式的连接查询语句来替代select elements(...)。而在HQL的where子句中，仍然可以使用elements()函数。

## Hibernate3之Hello, Word程序

这个板块太冷清了，记得范智曾经转贴过一个网上的“史上最简单的Hibernate入门”  
现在Falcon要开始使用Hibernate了，我就勉为其难，出个cauchy版本的“Hibernate入门”

Hibernate运行时依赖于以下软件包：

```
cglib-full-2.0.2.jar
commons-collections-2.1.1.jar
commons-logging-1.0.4.jar
dom4j-1.5.jar
ehcache-0.9.jar
hibernate3.jar
jta.jar
```

使用Hibernate首先要定义主配置文件hibernate.cfg.xml，再定义数据库映射文件

其中数据库映射文件和DAO类在实践中有3种使用方法：

1. 全部手工编写
2. 编写数据库映射文件，使用Hibernate工具，产生DAO类
3. 编写DAO类，使用类似XDoclet的标签技术，产生数据库映射文件

为了简化流程，下面以第一种方法为例说明：

### 1. Hibernate 配置文件

[code]

```
< ? xml version='1.0' encoding='utf-8'?>
< !DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration.dtd"
</hibernate-configuration>

<session-factory>
  <!--
  <property name="connection.datasource">java:comp/env/jdbc/hibernate
  -->

  <property name="generate_statistics">true</property>

  <property name="dialect">org.hibernate.dialect.SQLServer dialect</property>
```

```

    <property name="connection.driver_class">com.inet.tds.Td
    <property name="connection.pool_size">150</property>
    <property name="connection.url">jdbc:inetdae7:127.0.0.1::
    <property name="connection.username">Falcon </property>
    <property name="connection.password">Falcon</property>

    <property name="show_sql">>false</property>

    <mapping resource="cauchy/Region.hbm.xml"/>
</session-factory>

```

```

</hibernate-configuration>
[/code]

```

## 2. 数据库映射定义

```

[code]
< ? xml version="1.0"?>
< !DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.
<hibernate-mapping>

    <!-- table created by:
        CREATE TABLE REGION ( RECORD_ID VARCHAR(64) PRIMARY I
    -->
    <class name="cauchy.Region" table="REGION">
        <id type="string" name="ID" column="RECORD_ID" len
            <generator class="uuid.hex"/>
        </id>

        <property type="string" name="name" column="NAME" length:
        <property type="string" name="code" column="CODE" length:
    </class>

</hibernate-mapping>
[/code]

```

## 3. 自动产生或者手工编写DAO对象

```

[code]
package cauchy;

```

```
import java.io.Serializable;

public class Region implements Serializable {

    private String id;
    private String code;
    private String name;

    /**
     * @param id The id to set.
     */
    public void setID(String id) {
        this.id = id;
    }

    /**
     * @return Returns the id.
     */
    public String getID() {
        return id;
    }

    /**
     * @param code The code to set.
     */
    public void setCode(String code) {
        this.code = code;
    }

    /**
     * @return Returns the code.
     */
    public String getCode() {
        return code;
    }

    /**
     * @param name The name to set.
     */
    public void setName(String name) {
        this.name = name;
    }
}
```

```

    }

    /**
     * @return Returns the name.
     */
    public String getName() {
        return name;
    }
}
[/code]

```

#### 4. 运行测试

```

[code]
package cauchy;

import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;

public class TestRegion {

    public static void main(String[] args) {
        SessionFactory sf = new Configuration().conf:
        Session session = sf.openSession();

        long t = System.currentTimeMillis();

        Transaction tx = session.beginTransaction();
        for(int i = 1000; i <= 9999; i++) {
            Region region = new Region();
            region.setCode("CODE_" + i);
            region.setName("NAME_" + i);
            session.save(region);
        }
        tx.commit();

        t = System.currentTimeMillis() - t;

        System.out.println("\nTotal Use " + t + " ms'

```

```
        session.close();
        sf.close();
    }
}
[/code]
```

## 5. 程序代码

它是用Hibernate3和Eclipse3开发的，打包为[Hibernate3Test.jar](#)

## 在WebLogic中使用Hibernate

关于这个话题，javaeye其实有一篇文章专门介绍了

( <http://www.javaeye.com/viewtopic.php?t=245> )，但是可能不是很详细，最近也有一些人我这方面的问题，所以在这里重新介绍一下。不过我还是推荐你在看本文之前首先看一下上面提到的那篇文章。

首先说明一下我们这里使用的程序，为了更容易理解，我们使用hibernate文档（英文版：

[http://www.hibernate.org/hib\\_docs/v3/reference/en中文](http://www.hibernate.org/hib_docs/v3/reference/en中文)

版：[http://www.hibernate.org/hib\\_docs/v3/reference/html/](http://www.hibernate.org/hib_docs/v3/reference/html/) ) 中刚开始介绍与Tomcat进行整合时候的那个程序。

为了更加清晰，我仍然把代码贴在下面：

```
1 package example;
2
3 public class Cat {
4
5     private String id;
6     private String name;
7     private char sex;
8     private float weight;
9
10    public Cat() {
11    }
12
13    public String getId() {
14        return id;
15    }
```

```
16
17     private void setId(String id) {
18         this.id = id;
19     }
20
21     public String getName() {
22         return name;
23     }
24
25     public void setName(String name) {
26         this.name = name;
27     }
28
29     public char getSex() {
30         return sex;
31     }
32
33     public void setSex(char sex) {
34         this.sex = sex;
35     }
36
37     public float getWeight() {
38         return weight;
39     }
40
41     public void setWeight(float weight) {
42         this.weight = weight;
43     }
44
45 }
46
```

还有就是Cat.hbm.xml:

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping
    PUBLIC "-//Hibernate/Hibernate Mapping DT
    D 3.0//EN"
```

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

<hibernate-mapping>

  <class name="org.hibernate.examples.quickstart.Cat" table="CAT">

    <!-- A 32 hex character is our surrogate key. It's automatically
         generated by Hibernate with the UUID pattern. -->
    <id name="id" type="string" unsaved-value="null" >
      <column name="CAT_ID" sql-type="char(32)" not-null="true"/>
      <generator class="uuid.hex"/>
    </id>

    <!-- A cat has to have a name, but it shouldn't be too long. -->
    <property name="name">
      <column name="NAME" length="16" not-null="true"/>
    </property>

    <property name="sex"/>

    <property name="weight"/>

  </class>

</hibernate-mapping>
```

关于数据库表的建立，在这里就不再赘述了。下面我们分成几步来介绍，为了介绍方便，我们假设你现在有一个weblogic配置

在D:\bea\user\_projects\domains\mydomain下。

1. 设置classpath ,

- A. hibernate本身需要一些jar（到底需要哪些jar可以参照hibernate文档），你需要在classpath里面引入这些jar。另外你还会写这个方法也仍然是修改startWeblogic.cmd。举例来讲，假如你把这些jar拷贝到了D:\bea\user\_projects\domains\mydomain那么可以在startWeblogic.cmd中添加这样两句话：

```
set HIBERNATE_LIB=D:\bea\user_projects\domains\mydomain\lib
```

```
set CLASSPATH=%HIBERNATE_LIB%\antlr-2.7.5H3.jar;%HIBERNATE_LIB%\asm-attrs.jar;%HIBERNATE_LIB%\cglib-2.1.jar;%HIBERNATE_LIB%\commons-collections-2.1.1.jar;%HIBERNATE_LIB%\commons-logging-1.0.4.jar;%HIBERNATE_LIB%\concurrent-1.3.2.jar;%HIBERNATE_LIB%\dom4j-1.6.jar;%HIBERNATE_LIB%\jaas.jar;%HIBERNATE_LIB%\jacc-1_0-fr.jar;%HIBERNATE_LIB%\jaxen-1.1-beta-4.jar;%HIBERNATE_LIB%\log4j-1.2.9.jar;%HIBERNATE_LIB%\xml-apis.jar;%HIBERNATE_LIB%\asm.jar;%HIBERNATE_LIB%\hsqldb.jar;%HIBERNATE_LIB%\hibernate3.jar;lib\classes;%HIBERNATE_LIB%\ehcache-1.1.jar;%CLASSPATH%
```

- B. 设置你编译后的程序目录，我们这里假设假如你编译后的代码在D:\bea\user\_projects\domains\mydomain下，那么仍然是参照上面的方法，在startWeblogic.cmd中添加这样两句话：

```
set MY_CLASSES=D:\bea\user_projects\domains\mydomain\classes
```

```
set CLASSPATH=%MY_CLASSES%;%CLASSPATH%
```

这样classpath导入的工作就完成了。

2. 打开Weblogic Administration Console ，然后配置好你的连接池和datasource ，这里我使用datasource的JNDI Name用了mydatasource
3. 书写hibernate配置文件，大家都知道hibernate配置文件可以写成xml也可以写成properties的形式，这里我使用的是xml的方式。

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD
    3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
  <session-factory>
    <!-- SQL dialect -->
    <property name="dialect">org.hibernate.dialect.Oracle9Dialect</property>
    <!-- Echo all executed SQL to stdout -->
    <property name="show_sql">true</property>
  >
    <property name="connection.datasource">mydatasource</property>
    <property name="session_factory_name">hibernate.quickstart</property>
```

```
<property name="transaction.manager_lo  
okup_class">org.hibernate.transaction.Weblogi  
cTransactionManagerLookup  
</property>  
    <mapping resource="example/Cat.hbm.x  
ml" />  
</session-factory>  
</hibernate-configuration>
```

配置时需要注意的就是  
session\_factory\_name中使用了一个点来代  
替/，也就是hibernate.quickstart，实际程序  
lookup时候仍然使用hibernate/quickstart

至于transaction.manager\_lookup\_class如  
果你不打算用JTA可以不配。

#### 4. 写WebLogic的启动类，WebLogic的启 动类需要实现

weblogic.common.T3StartupDef接口，编  
程时候你要引入这个接口，可以通过引  
入weblogic.jar实现。假如你weblogic安  
装在D:\bea下面，你可以在相应的  
weblogic81\server\lib下找到这个jar。其  
实只是获得SessionFactory，hibernate会  
自动绑定到相应的JNDI name上的。

```
package example;  
  
import java.util.Hashtable;  
  
import org.apache.log4j.Logger;  
import org.hibernate.SessionFactory;  
import org.hibernate.cfg.Configuration;  
  
import weblogic.common.T3ServicesDef;  
import weblogic.common.T3StartupDef;
```

```

public class StartHibernateConfig
    implements T3StartupDef
{
    private Logger log = Logger.getLogger(Start
HibernateConfig.class);
    public String startup(String arg0, Hashtable a
rg1)
        throws Exception
    {
        Configuration config = new Configuration()
.configure();
        SessionFactory sf = config.buildSessionFa
ctory();

        log.info("Initial hibernate SessionFactory s
uccessfully,sf:" + sf);

        return "Initial hibernate SessionFactory su
ccessfully";
    }

    public void setServices(T3ServicesDef t3serv
icesdef)
    {
    }
}

```

5. 仍然是在Weblogic Administration Console中，从左边的applet树中找到 StartUp & Shutdown，然后选择Configure a new Startup Class...，按照提示一步一步配置就可以了。然后重启一下 Weblogic
6. 到这里为止，所有的配置工作就完成

了，你可以在程序里面使用Hibernate了。

下面是一些关于编程的简单介绍。

如果不使用JTA，比如在一个servlet中可以这样写

```
Context ctx=new InitialContext();
SessionFactory sessions=(SessionFactory)ctx.lookup("hibernate/quickstart");
Session sess = factory.openSession();
Transaction tx = null;
try {
    tx = sess.beginTransaction();

    // do some work
    □

    tx.commit();
}
catch (RuntimeException e) {
    if (tx != null) tx.rollback();
    throw e; // or display error message
}
finally {
    sess.close();
}
```

使用BMT的话其实写法和上面是一样的。

如果使用CMT，那么你的程序里面就不需要自己管理事务，容器会替你完成的。

另外在获得Session的时候可以使用SessionFactory的getCurrentSession()方法。

下面我们通过一个完整的SLSB的例子来看一下。这里仍然使用了我们在前面提到过的Cat。

首先是需要的java程序：

Remote Interface:

```
package example;

import java.rmi.RemoteException;
import javax.ejb.EJBObject;
import javax.naming.NamingException;

public interface Sample extends EJBObject {

    public String countCats() throws RemoteException, NamingException;

}
```

Home Interface:

```
package example;

import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.EJBHome;

public interface SampleHome extends EJBHome {

    public Sample create() throws RemoteException, CreateException;

}
```

Bean Class:

```
package example;

import java.rmi.RemoteException;
import java.util.List;
import javax.ejb.EJBException;
import javax.ejb.SessionBean;
import javax.ejb.SessionContext;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;

import org.apache.log4j.Logger;
import org.hibernate.Session;
import org.hibernate.SessionFactory;

public class SampleBean implements SessionBean {

    private Logger log = Logger.getLogger(Sa
```

```
mpleBean.class);

    private SessionContext sctx;

    public void setSessionContext(SessionContext ctx) throws EJBException,
        RemoteException {

    }

    public void ejbCreate() throws EJBException, RemoteException {
    }

    public void ejbRemove() throws EJBException, RemoteException {
    }

    public void ejbActivate() throws EJBException, RemoteException {
    }

    public void ejbPassivate() throws EJBException, RemoteException {
    }

    public String countCats() throws RemoteException, NamingException {
        Context ctx = new InitialContext();
        SessionFactory sf = (SessionFactory) ctx
            .lookup("hibernate/quickstart");
        Session s = sf.getCurrentSession();
        try {
```

```
        List ls = s.createQuery("from example
.Cat").list();
        String x = String.valueOf(ls.size());
        log.info("length:" + x);
        return x;
    } catch (RuntimeException e) {

    } finally {
        s.close();
    }
    return null;
}
}
```

这段程序一个特殊的地方就是我们使用 `sf.getCurrentSession ()` 来得到一个 `Session` 对象，另外一个就是没有在里面手动地处理事务。当然这只是一个查询而已，不过其他的程序写法都是类似的。

Trackback:

<http://tb.donews.net/TrackBack.aspx?>

PostId=511731

## Weblogic 8.1与Hibernate的结合的解决方案

---

日期：2005-7-11 15:57:49 人气：0 [大中小]

版权声明：方便学习使用，本文可以任意转载

基于Hibernate在O/R Mapping方面的优势，目前项目Hibernate实体替代EJB EntityBean,本人在WebLogic 8.1做了一个测试，用EJB SessionBean调用Hibernate的数据。为Weblogic和Hibernate都提供了数据库连接池,JNDI,等功能。主导思想还是想利用Weblogic Server的在这些服务能管理。

设计思想：

使用WebLogic的数据库连接池，而不是Hibernate自带的池。

将Hibernate的SessionFactory配置到Weblogic JNDI目录；在SessionBean中直接调用Hibernate的实体访问数

准备条件：

1、安装以下软件(都可以免费下载使用)

1.1 Mysql 4.0.21 c:\mysql

创建数据库study,创建数据表cat

1.2 mysql-connector-java-3.0.15-ga.zip mysql驱动程序

### 1.3 Weblogic platform 8.1 c:\bea

Weblogic配置完成，域mydomain和服务器myserver  
studyjndi,数据源名称mysqldatasource

### 1.4 Hibernate 2.1.2

参考其它文档编写一个hibernate的实例cat,编写Cat.h  
hibernate.cfg.xml文件，了解hibernate的基本配置  
注意数据库的差异。

## 2.创建目录结构

C:\Test\lib 将hibernate解压后lib目录下的全部文件拷.

C:\Test\src\com\chenm 源代码存放地(\*.java)

C:\Test\classes 将hibernate的配置文件

(hibernate.properties,log4j.properties,cache.ccf)

C:\Test\classes\com\chenm 编译好的代码(\*.class) + Cat.h

hibernate.cfg.xml

步骤1:配置hibernate的环境目录到Weblogic的CLASSP/

修改Weblogic启动脚本

C:\bea\user\_projects\domains\mydomain\startweblogic.c

@REM Call WebLogic Server前加入

@rem set hibernate classpath

set HIBERNATE\_LIB=C:\Test\lib

set HIBERNATE\_CLASSES=C:\Test\classes

```
SET CLASSPATH=%HIBERNATE_LIB%\cglib-rc2.jar;%HIBERNATE_LIB%\commons-collection-2.1.jar;%HIBERNATE_LIB%\commons-lang-1.0.1.jar;%HIBERNATE_LIB%\commons-logging-1.0.3.jar;%HIBERNATE_LIB%\dom4j-1.4.jar;%HIBERNATE_LIB%\hibernate2.jar;%HIBERNATE_LIB%\hibernate2-1.0-dev.jar;%HIBERNATE_LIB%\log4j-1.2.8.jar;%HIBERNATE_LIB%\odmg-3.0.jar;%HIBERNATE_CLASSES%;%CLASSPATH;
```

## 步骤2:修改hibernat.properties文件

### 2.1 修改以下内容

注释掉mysql缺省数据库连接

```
## HypersonicSQL
```

```
#hibernate.dialect net.sf.hibernate.dialect.HSQLDialect
#hibernate.connection.driver_class org.hsqldb.jdbcDriver
#hibernate.connection.username sa
#hibernate.connection.password
#hibernate.connection.url jdbc:hsqldb:hsqldb://localhost:9001
#hibernate.connection.url jdbc:hsqldb:test
#hibernate.connection.url jdbc:hsqldb:..
```

## 使用mysql数据库

## MySQL

```
hibernate.dialect net.sf.hibernate.dialect.MySQLDi
#hibernate.connection.driver_class org.gjt.mm.mysql.
hibernate.connection.driver_class com.mysql.jdbc.D
hibernate.connection.url jdbc:mysql://localhost:3306
hibernate.connection.username test
hibernate.connection.password weblogic
```

### 调整数据库查询和插入的性能参数

修改hibernate.jdbc.fetch\_size 50

修改hibernate.jdbc.batch\_size 25

### 调整Transaction API

```
#hibernate.transaction.factory_class
net.sf.hibernate.transaction.JTATransactionFactory
#hibernate.transaction.factory_class
net.sf.hibernate.transaction.JDBCTransactionFacto
```

为

```
hibernate.transaction.factory_class
net.sf.hibernate.transaction.JTATransactionFactory
hibernate.transaction.factory_class
```

```
net.sf.hibernate.transaction.JDBCTransactionFacto
```

## 使用JCS缓存

```
hibernate.transaction.manager_lookup_class  
net.sf.hibernate.transaction.WeblogicTransactionManager
```

### 2.2 在文件尾增加以下内容

```
hibernate.dialect net.sf.hibernate.dialect.MySQLDi  
hibernate.connection.datasource studyjndi // 此处为web  
据连接池JNDI名称
```

```
hibernate.connection.provider_class  
net.sf.hibernate.connection.DatasourceConnectionPro  
hibernate.session_factory_name hibernate.session_facto  
到weblogic JNDI目录树中的名称
```

步骤3. 实现SessionFactory的预创建，使用Weblog  
T3StartUpDef接口创建一个StartUp类，配置成Web  
启动时自动运行。

3.1 创建文件HibernateStartUp.java,并编译成  
C:\Test\classes\com\chenm\HibernateStartUp.class文  
package com.chenm;

```

import java.util.Hashtable;
import weblogic.common.T3StartupDef;
import weblogic.common.T3ServicesDef;
import net.sf.hibernate.cfg.Configuration;
import net.sf.hibernate.SessionFactory;

public class HibernateStartUp implements T3StartupI
{
    public void setServices(T3ServicesDef services)
    {
    }

    public String startup(String name,Hashtable args) throws
        Exception {
        Configuration conf = new
            Configuration().addClass(Cat.class);
        SessionFactory sf = conf.buildSessionFactory();
        return "Hibernate Startup completed successfully";
    }
}

```

### 3.2 配置StartUp类

启动Weblogic控制台，打开左边mydomain\部署\启动点,选择右边"配置新的 Startup Class..."  
 填写名称HibernateStartup, 类名com.chenm.HibernateS  
 后点击"创建", 如果没有出错信息就算成功。

确认成功：关闭Weblogic并重启，观察DOS窗口的信  
看到在Weblogic启动后显示很多行INFO，如果没  
错误，证明配置成功。再打开weblogic控制台，  
mydomain\服务器\myserver,点右键，选择察看JNDI树  
看到Hibernate的JNDI对象，在右边可以看见以下1

绑定名称: session\_factory

对象类: net.sf.hibernate.impl.SessionFactoryImpl

对象散列代码: 45706641

对象转换成字符串:

net.sf.hibernate.impl.SessionFactoryImpl@2b96d9

Config OK!

#### 4. 编写SessionBean操作Hibernate实体

在SessionBean中定义Remote方法

```
public void InsertCat(String cat_id,String name, char s  
weight) {  
/**@todo Complete this method*/  
try {
```

```
Context ctx = getInitialContext();
    SessionFactory sf =
(SessionFactory)ctx.lookup("hibernate/session_factory");
    Session s = sf.openSession() ;
Transaction t = s.beginTransaction() ;
```

```
    Cat myCat = new Cat();
        myCat.setId(cat_id);
        myCat.setName(name);
        myCat.setSex(sex);
myCat.setWeight(weight);s.save(myCat);
        s.save(myCat);
        t.commit() ;
        s.close();
    }
catch( Exception ex ) {
    }

}
```

```
private Context getInitialContext() throws Exception {
String url = "t3://chenming:7001"; // chenming服务器
    String user = null;
    String password = null;
```

```

        Properties properties = null;
        try {
            properties = new Properties();
            properties.put(Context.INITIAL_CONTEXT_FACTORY,
                "weblogic.jndi.WLInitialContextFactory");
            properties.put(Context.PROVIDER_URL, url);
            if (user != null) {
                properties.put(Context.SECURITY_PRINCIPAL,
                    user);
                properties.put(Context.SECURITY_CREDENTIALS,
                    password);
            }
            return new InitialContext(properties);
        }
        catch(Exception e) {
            throw e;
        }
    }
}

```

编写测试并运行，在cat表中插入一条纪录

```

Context context = getInitialContext();

//look up jndi name
Object ref = context.lookup("CatSession");
//look up jndi name and cast to Home interface

```

```
catSessionHome = (CatSessionHome)
PortableRemoteObject.narrow(ref, CatSessionHome.c
catSession = catSessionHome.create();
catSession.InsertCat("007","Chenm.cat",'1',100)
```

## 使用Hibernate的一个完整例子

陈亚强 (cyqcims@mail.tsinghua.edu.cn)

北京华园天一科技有限公司高级软件工程师  
2003 年 10 月

对象、关系的映射 (ORM) 是一种耗时的工作, 在Java环境下, 有几种框架来表示持久数据, 如实体Bean、OJB、JDO、Hibernate等。Hibernate是一种新的ORM映射工具, 它不仅提供了从Java类到数据表的映射, 也提供了数据查询和恢复等机制。本文介绍怎么在Web应用开发中配置Hibernate的环境, 并且使用Hibernate来开发一个具体的实例。

阅读本文前您需要以下的知识和工具:

- Tomcat 5.09, 可以从 <http://www.ipx.cn/html/java/20040301/index/v> 下载;
- Hibernate2.0 相关运行环境, 可以从 <http://www.hibernate.org/> 下载;
- 至少一个数据库服务器并且有相关的 JDBC驱动程序。

本文的参考资料见 [参考资料](#)。

## 介绍

面向对象的开发方法是当今的主流，但是同时我们不得不使用关系型数据库，所以在企业级应用开发的环境中，对象、关系的映射（ORM）是一种耗时的工作。围绕对象关系的映射和持久数据的访问，在Java领域中发展起来了一些API和框架，下面分别简单介绍。

JDBC可以说是访问持久数据层最原始、最直接的方法。在企业级应用开发中，我们可能使用DAO（Data Access Object）模式来把数据访问封装起来，然后在其它的层中同一调用。这种方式的优点是运行效率最高，缺点是把DAO对象和SQL语言紧密耦合在一起使得在大项目中难以维护。但是不管怎么说，使用JDBC来直接访问持久数据层是当今企业级应用开发中使用最广泛

的。

实体Bean是J2EE平台中用来表示和访问持久数据的方式。虽然实体Bean是一种方便快捷的方法，但是在运行时我们需要额外购买EJB容器（当然，如今也有免费的EJB容器，如JBOSS），并且使用不同的应用服务器，需要重新书写不同的部署描述，使得在不同应用服务器下移植企业级应用会带来一些困难。

另外，在Java领域中，还有一些表示持久数据的框架，比如JDO和OJB，在这里就不详细介绍了。

Hibernate是一种新的ORM映射工具，它不仅提供了从Java类到数据表之间的映射，也提供了数据查询和恢复机制。相对于使用JDBC和SQL来手工操作数据库，使用Hibernate，可以大大减少操作数据库的工作量。

Hibernate可以和多种Web服务器或者应用服务器良好集成，如今已经支持几乎所有的流行的数据库服务器（达16种）。

下面我们来介绍怎么结合Hibernate2.0和Apache Tomcat5.0在Web应用中使用Hibernate。

## 配置

1、 下载安装Tomcat，并且下载Hibernate的运行环境（主要包含一些JAR包）。

2、 把要使用的数据库的JDBC驱动程序拷贝到%TOMCAT\_HOME%\common\lib目录下。笔者使用的是MYSQL，对应的驱动程序的JAR包为mm.mysql-2.0.4-bin.jar。

3、 在Tomcat的Webapps目录下新建一个Web应用，名字为hibernate。

4、把Hibernate提供的hibernate2.jar和一些第三方的运行库拷贝到hibernate\WEB\INF\lib目录下。（这些第三方的运行库包含在下载到的Hibernate lib目录下）

5、在%TOMCAT\_HOME%\conf\server.xml中配置Web应用和数据源。在server.xml中加入以下的配置描述。

#### 例程1 配置web应用

```
<Context path="/hibernate" docBase="hibernate"
<Resource name="jdbc/hibernate" auth="Container"
<ResourceParams name="jdbc/hibernate">
<parameter>
<name>factory</name>
<value>org.apache.commons.dbcp.BasicDataSource
</parameter>
<parameter>
<name>driverClassName</name>
<value>org.gjt.mm.mysql.Driver</value>
</parameter>
<parameter>
```

```
<name>url</name>
<value>jdbc:mysql:///test</value>
</parameter>
<parameter>
<name>username</name>
<value>root</value>
</parameter>
<parameter>
<name>password</name>
<value></value>
</parameter>
<parameter>
<name>maxActive</name>
<value>20</value>
</parameter>
<parameter>
<name>maxIdle</name>
<value>10</value>
</parameter>
<parameter>
<name>maxWait</name>
<value>-1</value>
</parameter>
</ResourceParams>
</Context>
```

在这里，配置了一个名为hibernate的Web应用，并且配置了一个数据源，数据源的JNDI名称为jdbc/hibernate。您需要根据情况修改数据源的链接属性。

6、 下一步就是书写Hibernate的配置描述符。可以使用XML的配置描述，也可以使用基于属性的配置描述。在这里使用基于XML的配置描述。在

hibernate\WEB-INF\classes目录下新建一个hibernate.cfg.xml文件。然后加入例程2所示的内容。

```
<!DOCTYPE hibernate-configuration
PUBLIC "-//Hibernate/Hibernate Configuration D
"http://hibernate.sourceforge.net/hibernate-co

<hibernate-configuration>
<session-factory>
<property name="connection.datasource">java:co
<property name="show_sql">>false</property>
<property name="dialect">net.sf.hibernate.dial

<!-- Mapping files -->

</session-factory>

</hibernate-configuration>
```

注意connection.datasource属性必须和server.xml中配置的数据源的属性一样。如果不是使用MYSQL，那么需要更改dialect属性。

到现在，配置基本完成，下面我们来开发一个最

简单的应用。

开发持久对象、编写映射描述

我们使用hibernate来封装一个简单的数据表。这个表的名字为Courses，它有两个字段，一个是ID，它是Courses表的主键；另一个是name，表示Courses的名字。在数据库中使用以下的脚本来创建这个表：

```
create table Courses(CourseId varchar(32) not null, name varchar(32), constraint pk_Courses primary key (CourseId) );
```

接下来的任务就是为Courses表书写持久对象，如例程3所示。

例程3 Courses的持久对象（Courses.java）

```
package com.hellking.study.hibernate;

import java.util.Set;

/**
 *在hibernate中代表了Course表的类。
 */
public class Course
{
    /**每个属性和表的一个字段对应**/
    private String id;
    private String name;

    /**students表示course中的学生，在后面才会用到，暂时不
    private Set students;

    /**属性的访问方法**/
    public void setId(String string) {
        id = string;
    }

    public String getId() {
        return id;
    }

    public void setName(String name)
    {
        this.name=name;
    }
    public String getName()
    {
        return this.name;
    }
    public void setStudents(Set stud)
    {
        this.students=stud;
    }
    public Set getStudents()
    {
        return this.students;
    }
}
```

```
}  
}
```

可以看出，在Course类中也包含了两个属性，id和name，它的属性和表Courses的字段是一一对应的，并且类型一致。

书写好了持久对象，接下来的任务就是书写对象、关系映射描述。在hibernate\WEB-INF\classes目录下新建一个Course.hbm.xml描述文件，内容如例程4所示。

#### 例程4 Course.hbm.xml

```
<?xml version="1.0"?>  
<!DOCTYPE hibernate-mapping PUBLIC  
"-//Hibernate/Hibernate Mapping DTD 2.0//EN"  
"http://hibernate.sourceforge.net/hibernate-ma  
  
<hibernate-mapping>  
<class  
name="com.hellking.study.hibernate.Course"  
table="Courses"  
dynamic-update="false"  
>  
  
<id  
name="id"  
column="CourseId"  
type="string"
```

```
unsaved-value="any"  
>  
<generator class="assigned"/>  
</id>  
  
<property  
name="name"  
type="string"  
update="true"  
insert="true"  
column="Name"  
>  
</class>  
</hibernate-mapping>
```

在Course.hbm.xml映射文件中，指定了要映射的类和映射的表，并且指定了表的各个字段和Java对象中各个字段的映射关系，比如Course对象中的id属性对应了Courses表的courseId字段。

接下来的任务就是在hibernate.cfg.xml中指定这个映射关系。如下所示：

```
<session-factory>  
...
```

```
<!-- Mapping files -->  
<mapping resource="Course.hbm.xml"/>  
</session-factory>
```

## 编写业务逻辑

到此，我们已经封装了一个名为Courses的表，并且配置完成。接下来的任务就是在Web应用开发中使用它们，为了演示在Hibernate中对数据库的不同类型的操作，我们开发的Web应用有以下功能：

- 增加一个Course；
- 删除一个Course；
- 按照Course的名字进行模糊搜索；
- 查看系统中所有的Course。

虽然我们可以直接在JSP中使用hibernate，但是往往我们不这样，而是把这些业务逻辑封装在JavaBean中，然后在JSP中通过调用JavaBean以访问Hibernate封装的对象。

由于访问通过使用hibernate有一些共性的操作，在这里我们把这些共性的操作封装在一个专门的类中，这样其它的类可以继承它，如例程5所示。

#### 例程5 HibernateBase.java

```
package com.hellking.study.hibernate;

import net.sf.hibernate.*;
import net.sf.hibernate.cfg.*;
import java.util.*;
import java.io.IOException;
import java.io.PrintWriter;

public abstract class HibernateBase
```

```

{
protected SessionFactory sessionFactory;//会话工厂
protected Session session;//hibernate会话
protected Transaction transaction; //hibernate事务

public HibernateBase()throws HibernateException
{
this.initHibernate();
}
// 帮助方法
protected void initHibernate()
throws HibernateException {

// 装载配置，构造SessionFactory对象
sessionFactory = new Configuration().configure()
}

/**
*开始一个hibernate事务
*/
protected void beginTransaction()
throws HibernateException {

session = sessionFactory.openSession();
transaction = session.beginTransaction();
}

/**
*结束一个hibernate事务。
*/
protected void endTransaction(boolean commit)
throws HibernateException {

if (commit) {
transaction.commit();
} else {
//如果是只读的操作，不需要commit这个事务。
transaction.rollback();
}
session.close();
}
}
}

```

下面编写业务逻辑类，新建一个名为CourseBean的JavaBean，并且CourseBean继承HibernateBase类，代码如例程6所示。

#### 例程6 CourseBean.java

```
package com.hellking.study.hibernate;

import net.sf.hibernate.*;
import net.sf.hibernate.cfg.*;
import java.util.*;

/**
 *和course相关的业务逻辑
 */
public class CourseBean extends HibernateBase
{
    public CourseBean()throws HibernateException
    {
        super();
    }
    /**
    *增加一个Course
    */
    public void addCourse(Course st)throws Hibernate
    {
        beginTransaction();
        session.save(st);
        endTransaction(true);
    }

    /**
    *查询系统中所有的Course，返回的是包含有Course持久对象
```

```

*/
public Iterator getAllCourses()throws Hibernat
{
String queryString = "select courses from Cour
beginTransaction();
Query query = session.createQuery(queryString)
Iterator it= query.iterate();
return it;
}

/**
 *删除给定ID的course
 */
public void deleteCourse(String id)throws Hibe
{
beginTransaction();
Course course=(Course)session.load(Course.clas
session.delete(course);
endTransaction(true);
}

/**
 *按course的名字进行模糊查找，返回的是包含有Course持久
 */
public Iterator getSomeCourse(String name)thro
{
String queryString = "select c from Course as
beginTransaction();
Query query = session.createQuery(queryString)
query.setString("name", "%"+name+"%");
Iterator it= query.iterate();
return it;
}
}
}

```

在CourseBean封装了4个业务方法，你可以根据情况增加其它的业务方法。在CourseBean中，通过Hibernate来操作潜在的数据库资源。

要保存Course数据到数据库，可以通过：

```
session.save(course);
```

方法来保存，它相当于使用在JDBC中执行以下语句：

```
Connection con=...  
Statement stmt=con.createStatement();  
stmt.executeUpdate("insert into courses values ...");  
con.close();
```

可以看出，通过使用Hibernate，可以大大减少数

据访问的复杂度。

在JSP中调用业务逻辑

添加数据

CourseBean这个业务对象封装了和Hibernate的交互关系，从而使JSP和Hibernate关系的解藕。我们来看测试主页面的部分代码，如例程7所示。

例程7 测试Hibernate开发的应用（course.jsp）

```
<%@ page import="java.sql.*,java.util.*" error
<jsp:useBean id="course" class="com.hellking.s
<jsp:setProperty name="course" property="*" />
</jsp:useBean>
<jsp:useBean id="courseBusiness" class="com.he
<html><body><center>
<%
try
{
if(course.getId().equals(null)||course.getId()
```

```

else courseBusiness.addCourse(course);

%>
成功添加了Course : <br>
name : <%=course.getName() %>
Id : <%=course.getId() %>
<%
}
}
catch(Exception e)
{
}
%>

<hr>
<br>::增加一个course::<br>
<form action="course.jsp" method="get" name="a
id:<input type="text" name="id"><br>
name:<input type="text" name="name"><br>
<input type="submit" value="submit"><br>
</form>
<hr>
::按名字模糊查找::<br>
<form action="queryCourse.jsp" method="get" na
name:<input type="text" name="name"><br>
<input type="submit" value="query"><br>
</form>
<hr>
::删除一个Course::<br>
<form action="deleteCourse.jsp" method="get" n
id:<input type="text" name="id"><br>
<input type="submit" value="delete"><br>
</form>
<hr>
<a href=viewAll.jsp>__查看所有Course__<a>_BR> <
</html>

```

首先通过一个值对象Course（这个类正好是Hibernate使用的持久对象，这里作为值对象来传递数据）接收获得的参数，然后CourseBean的addCourse(Course)方法把数据保存到数据库。

可以看出，通过使用Hibernate，把数据从表单中添加到数据库非常简单。

## 查询

下面来看模糊查找的JSP代码，如例程8所示。

### 例程8 按名字模糊查找Course

```
<%@ page import="java.sql.*,java.util.*,com.he
<jsp:useBean id="courseBusiness" class="com.he
...
<% try
{
Iterator it=courseBusiness.getSomeCourse((Stri
while(it.hasNext())
{
Course temp=(Course)it.next();
out.println("<tr><td>"+temp.getId()+"</td>");
out.println("<td>"+temp.getName()+"</td></tr>"
}
}
catch(Exception e)
{
out.println(e.getMessage());
}
%>
...
```

它实际上调用的是CourseBean的Iterator  
getSomeCourse(String name)方法。我们来回顾  
一下这个方法中的代码：

```
/**
 *按course的名字进行模糊查找
 */
public Iterator getSomeCourse(String name)thro
{
String queryString = "select c from Course as
beginTransaction();
Query query = session.createQuery(queryString)
query.setString("name", "%"+name+"%");
Iterator it= query.iterate();
return it;
}
```

在查询前，首先调用beginTransaction方法启动  
新的Hibernate事务，然后创建一个Query对象，  
在创建这个对象时，同时指定查询的语句。

注意，在查询语句：

```
select c from Course as c where c.name like :n
```

中，它虽然和普通的SQL语句相似，但是不同，在数据库中，使用的表的名称是Courses，而在这个查询语句中使用的是Course，它和持久对象的名字一致，也就是说，这个查询的概念是查询持久对象，而不是数据库的记录。

创建了查询对象Query后，需要设置查询的参数，它和在JDBC中PreparedStatement对象中设置参数的方法相似。通过"Iterator it=query.iterate()"语句来执行查询，并且返回一个Iterator对象。在这里使用了Hibernate提供的查询机制，一般的JDBC查询返回的是ResultSet对象，而这里返回的是包含了CourseBean对象的Iterator。

要查询系统中所有的Course，也同样非常简单，可以通过例程9所示的代码实现。

### 例程9 查询数据库中所有的Course

```
...
<jsp:useBean id="courseBusiness" class="com.he
...
<% try
{
Iterator it=courseBusiness.getAllCourses();
while(it.hasNext())
{
Course temp=(Course)it.next();
out.println("<tr><td>" + temp.getId() + "</td>");
out.println("<td>" + temp.getName() + "</td></tr>"
}
}
catch(Exception e)
{
out.println(e.getMessage());
}
%>
...
```

实际上调用的是CourseBean的getAllCourses方法，它和getSomeCourse方法机制一样，就不再介绍了。

## 删除数据

在JSP中，使用以下的代码来执行删除操作。

### 例程10 删除数据库中Courses表的记录

```
<jsp:useBean id="courseBusiness" class="com.he
...
删除id为 :<%=request.getParameter("id")%>的course

<% try
{
courseBusiness.deleteCourse(request.getParameter
out.println("删除成功");
}
catch(Exception e)
{
out.println("不存在这个记录");
}
%>
```

我们来看CourseBean中执行删除操作的具体代码：

```
/**
 *删除给定ID的course
 */
public void deleteCourse(String id)throws Hibe
{
    beginTransaction();
    Course course=(Course)session.load(Course.class, id);
    session.delete(course);
    endTransaction(true);
}
```

在这个方法中，首先开始一个事务，然后通过session.load(Course.class,id)方法来装载指定ID的持久对象，接下来通过"session.delete(course)"来删除已经装载的course，并且结束Hibernate事务。

总结

下面总结一下使用Hibernate的开发过程：

- 1、 配置Hibernate（一次即可）；
- 2、 确定数据表；
- 3、 创建持久对象；
- 4、 编写对象和数据表的映射描述；
- 5、 编写和业务逻辑。

实际上，上面的过程和使用EJB没有什么区别：在使用EJB时，首先当然也是配置环境，初始化数据表；然后创建实体Bean（对象于Hibernate的持久对象）；接下来编写部署描述符（ejb-

jar.xml，厂商专有的部署描述），在这些部署描述符里，指定了EJB和数据表的映射关系，如果多个实体Bean存在关联关系，需要描述它们之间的关系，这些描述对应于Hibernate中持久对象的描述，如Course.hbm.xml；往往我们并不在应用程序中直接操作实体Bean，而是通过业务对象（如会话Bean）来操作，这里的会话Bean可以简单的和Hibernate中执行业务逻辑的JavaBean对应。这里只是简单的类比，不是绝对的，比如我们同样可以在会话Bean中访问Hibernate持久对象，也就是说使用Hibernate，同样可以把业务逻辑放在会话Bean中。

通过本文的学习，相信读者对Hibernate已经有了初步的认识，并且能够使用Hibernate开发简单的应用。在下一篇中，我们将学习怎么使用Hibernate来为复杂的数据表进行映射，并且维护它们之间的关系。

#### 参考资料

- <http://www.apache.org/> 下载Tomcat。
- Hibernate的官方网站，<http://hibernate.bluemars.net/>，包含了Hibernate最新资料。
- Hibernate中文论坛，[hibernate.fankai.com](http://hibernate.fankai.com)包含了Hibernate较多的参考资料。
- 包含了Hibernate技术讨论网站，<http://www.ipx.cn/html/java/20040301/index.html>
- 于Hibernate、JDO、CMP等技术的热烈讨论1：

<http://www.jdon.com/jive/thread.jsp?forum=16&thread=6062&start=0&msRange=15>

- 于Hibernate、JDO、CMP等技术的热烈讨论 2 :

[http://www.theserverside.com/discussion/thread/thread\\_id=19732](http://www.theserverside.com/discussion/thread/thread_id=19732)

- Hibernate2 Reference Documentation , 可以从Hibernate官方网站获得 , 非常好的参考资料。
- Hibernate In Action , 一本非常专业的Hibernate参考书 , 由Hibernate项目主要开发人员Gavin King 等著 , Manning出版社出版。您可以从 <http://www.theserverside.com/> 获得本书的部分章节。

Trackback:

[http://tb.donews.net/TrackBack.aspx?](http://tb.donews.net/TrackBack.aspx?PostId=122374)

[PostId=122374](http://tb.donews.net/TrackBack.aspx?PostId=122374)

[\[点击此处收藏本文\]](#) 发表于 2004年10月07日 5:45 PM

## Hibernate下数据批量处理解决方案

作者：weiwenking 出处：csdn 责任编辑：方舟 [2004-11-27 15:53]

很多人都对Java在批量数据的处理方面是否是其合适的场所持有怀疑的念头，由此延伸，那么就会认为ORM可能也不是特别适合数据的批量处理。其实，我想如果我们应用得当的话，完全可以消除ORM批量处理性能问题这方面的顾虑。下面以Hibernate为例来做为说明，假如我们真的不得不在Java中使用Hibernate来对数据进行批量处理的话。向数据库插入100 000条数据，用Hibernate可能像这样：

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();
for ( int i=0; i<100000; i++ ) {
    Customer customer = new Customer(.....);
    session.save(customer); }
tx.commit();
session.close();
```

大概在运行到第50 000条的时候，就会出现内存溢出而失败。这是Hibernate把最近插入的Customer都以session-level cache在内存做缓存，我们不要忘记Hibernate并没有限制first-level cache 的缓存大小：

# 持久对象实例被管理在事务结束时，此时Hibernate与数据库同步任何已经发生变化的被管理的对象。

# Session实现了异步write-behind，它允许Hibernate显式地写操作的批处理。这里，我给出Hibernate如何实现批量插入的方法：

首先，我们设置一个合理的JDBC批处理大小，hibernate.jdbc.batch\_size 20。然后在一定间隔对Session进行flush()和clear()。

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();
for ( int i=0; i<100000; i++ ) {
    Customer customer = new Customer(.....);
    session.save(customer);
```

```
        if ( i % 20 == 0 ) {
            //flush 插入数据和释放内存:
            session.flush(); session.clear(); }
        }
        tx.commit();
        session.close();
```

那么，关于怎样删除和更新数据呢？那好，在 Hibernate2.1.6或者更后版本，scroll() 这个方法将是最好的途径：

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();
ScrollableResults customers =
    session.getNamedQuery("GetCustomers")
        .scroll(ScrollMode.FORWARD_ONLY);
int count=0;
while ( customers.next() ) {
    Customer customer = (Customer) customers.get(0);
    customer.updateStuff(...);
    if ( ++count % 20 == 0 ) {
        //flush 更新数据和释放内存:
        session.flush(); session.clear(); } }
tx.commit(); session.close();
```

这种做法并不困难，也不算不优雅。请注意，如果 Customer启用了second-level caching，我们仍然会有一些内存管理的问题。原因就是对于用户的每一次插入和更新，Hibernate在事务处理结束后不得不通告second-level cache。因此，我们在批处理情况下将要禁用用户使用缓存。

# 利用weblogic的数据源作为hibernate的例子

Jagie 原创 (参与分：90219，专家分：2595) 发表：2003-11-03 16:42 更新  
04 08:03 版本：1.0 阅读：7617次

在网上，我们可以看到许多关于hibernate入门的例子，多是让hibernate自己管理连接池的。我在这里给出一个用weblogic发布的数据源的例子。步骤如下

1.写一个准备用于持久化的类

1. **package** com.jagie.business.organizat
- 2.
3. **import** java.io.[Serializable](#);
- 4.
5. **/\*\***
6. **\* <p>Title: </p>**
7. **\* <p>Description: 权限</p>**
8. **\* <p>Copyright: Copyright (c) 2003**  
>

```

9.  * <p>Company: www.jagie.com</p>
10. * @author Jagie
11. * @version 1.0
12. */
13.
14. public class Permission implements s
    {
15.     private String ID;//pk
16.     private String name;//名称
17.     private String description;//描述
18.     private String module;//模块id
19.     private String power;//权值,$分隔的操作
    例如:browse$add$delete$change
20.     private int scope;//范围 , 0:本人 ,
    本单位,2:所有单位
21.     public static void main(String[] args) {
22.     }
23.     public String getID() {
24.         return ID;
25.     }

```

```
26. public void setID(String ID) {
27.     this.ID = ID;
28. }
29. public String getName() {
30.     return name;
31. }
32. public void setName(String name) {
33.     this.name = name;
34. }
35. public String getDescription() {
36.     return description;
37. }
38. public void setDescription(String description) {
39.     this.description = description;
40. }
41. public String getModule() {
42.     return module;
43. }
44. public void setModule(String module) {
```

```
45.     this.module = module;
46.     }
47.     public String getPower() {
48.         return power;
49.     }
50.     public void setPower(String power) {
51.         this.power = power;
52.     }
53.     public int getScope() {
54.         return scope;
55.     }
56.     public void setScope(int scope) {
57.         this.scope = scope;
58.     }
59. }
```

2.编写一个xml文件，名称为Permission.hbm.xml，一定要确保在运行时该和Permission.class在一起

```
[pre]<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-mapping SYSTEM "http://hibernate.org/hibernate-mapping-2.0.dtd">
<hibernate-mapping>
  <class name="com.jagie.business.organization.Permission"
    name="SYS_Permission">
    <id name="ID">
      <generator class="uuid.hex"/>
    </id>
    <property name="name"/>
    <property name="module"/>
    <property name="description"/>
    <property name="power"/>
    <property name="scope"/>
  </class>
</hibernate-mapping>[/pre]
```

3.在weblogic 上配置连接池和数据源，我的数据源的jndi名字为OilDS

4.修改classpath下的hibernate.properties文件，并保存

- a.添加一行:hibernate.dialect net.sf.hibernate.dialect.Oracle
- b.找到JNDI Datasource这一段，在下面设置hibernate.connection.datasource OilDS
- c.找到Plugin ConnectionProvider部分，去掉hibernate.connection.provider\_class  
net.sf.hibernate.connection.DatasourceConnectionProvider
- d.找到 Transaction API部分,去掉hibernate.transaction.marshal\_class  
net.sf.hibernate.transaction.WeblogicTransactionManagerL

一句的注释  
e.保存修改

5.在类路径中编写一个jndi.properties文件，为了考虑灵活性，防止硬编码非常重要，内容如下

```
java.naming.factory.initial=weblogic.jndi.WLInitialContext  
y  
java.naming.provider.url=t3://localhost:7001（我的weblog  
器就在本机上，也许你的需要适当修改）
```

6.好啦，万事俱备，让我们写一个Test类来测试一下hibernate的威力好了，

1. **package** com.jagie.business.organizat
  - 2.
  3. **import** net.sf.hibernate.Session;
  4. **import** net.sf.hibernate.Transaction;
  5. **import** net.sf.hibernate.SessionFactor
  6. **import** net.sf.hibernate.cfg.Configura
- ;

```
7. import net.sf.hibernate.tool.hbm2ddl.  
   emaExport;  
8.  
9. import javax.naming.InitialContext;  
10. import javax.naming.Context;  
11. import javax.sql.*;  
12. import java.sql.*;  
13. import java.util.*;  
14. import com.jagie.utils.j2ee.*;  
15.  
16. public class Test {  
17.     private static SessionFactory session  
18.  
19.     public static void main(String [] args) thi  
20.     Exception {  
21.         Configuration conf = new Configur  
22.         ().addClass(Permission.class);  
23.         sessions = conf.buildSessionFactory
```

```
24.    //生成并输出sql到文件（当前目录  
和数据库  
25.    SchemaExport dbExport = new Sch  
aExport(conf);  
26.    dbExport.setOutputFile("sql.txt");  
27.    dbExport.create(true, true);  
28.  
29.  
30.    //start.....  
31.    Session s = sessions.openSession();  
32.    Transaction t = s.beginTransaction(  
33.  
34.    //1.用普通使用方式建立对象，填  
数据  
35.    Permission p1 = new Permission();  
36.    p1.setName("1111");  
37.  
38.    //2.持久化  
39.    s.save(p1);  
40.    //此时p1已经可以在数据库中找到
```

```
41.     t.commit();
42.     s.close();
43.
44. }
45. }
```

7.运行该类，即可看到数据库已经建立了一个sys\_permission的表，并且插条数据。很简单吧!

本文是开发基于spring的web应用的入门文章，前端采用Struts MVC框架，中间层采用spring，后台采用Hibernate。

本文包含以下内容：

- 配置Hibernate和事务
- 装载Spring的applicationContext.xml文件
- 建立业务层和DAO之间的依赖关系
- 将Spring应用到Struts中

## 简介

这个例子是建立一个简单的web应用，叫MyUsers,完成用户管理操作，包含简单的数据库增，删，查，该即CRUD（新建，访问，更新，删除）操作。这是一个三层的web应用，通过Action（Struts）访问业务层，业务层访问DAO。图一简要说明了该应用的总体结构。图上的数字说明了流程顺序 - 从web（UserAction）到中间层（UserManager），再到数据访问层（UserDAO），然后将结果返回。

Spring层的真正强大在于它的声明型事务处理，帮定和对持久层支持（例如Hibernate和iBATIS）

以下下是完成这个例子的步骤：

1. 安装Eclipse插件
2. 数据库建表
3. 配置Hibernate和Spring

- 4 . 建立Hibernate DAO接口的实现类
- 5 . 运行测试类 , 测试DAO的CRUD操作
- 6 . 创建一个处理类 , 声明事务
- 7 . 创建web层的Action和model
- 8 . 运行Action的测试类测试CRUD操作
- 9 . 创建jsp文件通过浏览器进行CRUD操作
- 10 . 通过浏览器校验jsp

### 安装eclipse插件

- 1 . Hibernate插件<http://www.binamics.com/hibernatesync>
- 2 . Spring插件<http://springframework.sourceforge.net/spring-ide/eclipse/updatesite/>
- 3 . MyEclipse插件(破解版)
- 4 . Tomcat插件. tanghan
- 5 . 其他插件包括xml , jsp ,

### 数据库建表

```
create table app_user(id number not null primary,firstname varchar(32),lastname varchar(32));
```

### 新建项目

新建一个web project，新建后的目录结构同时包含了新建文件夹page用于放jsp文件，和源文件夹test用于放junit测试文件。同时将用到的包，包括struts，hibernate，spring都导入到lib目录下。

## 创建持久层O/R mapping

1. 在src/com.jandar.model下用hibernate插件从数据库导出app\_user的.hbm.xml文件改名为User.hbm.xml

```
<?xml version="1.0"? >
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd" >
< hibernate-mapping package="com.jandar.model" >
  < class name="User" table="APP_USER" >
    < id
      column="ID"
      name="id"
      type="integer"
    >
      < generator class="assigned" / >
    < /id >
    < property
      column="LASTNAME"
      length="10"
      name="lastname"
      not-null="false"
      type="string"
    / >
    < property
      column="FIRSTNAME"
      length="10"
      name="firstname"
      not-null="true"
      type="string"
    / >
  < /class >
```

## 2 . 通过hibernate synchronizer- > synchronizer file生成User.java文件,User对象对应于数据库中的app\_user表

注：在eclipse下自动生成的对象文件不完全相同，相同的是每个对象文件必须实现Serializable接口，必需又toString和hashCode方法；

```
import java.io.Serializable;
import org.apache.commons.lang.builder.EqualsBuilder;
import org.apache.commons.lang.builder.HashCodeBuilder;
import org.apache.commons.lang.builder.ToStringBuilder;
import org.apache.commons.lang.builder.ToStringStyle;

public class BaseObject implements Serializable {
    public String toString() {
        return ToStringBuilder.reflectionToString(this,
            ToStringStyle.MULTI_LINE_STYLE);
    }

    public boolean equals(Object o) {
        return EqualsBuilder.reflectionEquals(this, o);
    }

    public int hashCode() {
        return HashCodeBuilder.reflectionHashCode(this);
    }
}

public class User extends BaseObject {
    private Long id;
    private String firstName;
    private String lastName;

    /**
     * @return Returns the id.
     */

    public Long getId() {
        return id;
    }

    /**
     * @param id The id to set.
```

```

        */

        public void setId(Long id) {
            this.id = id;
        }

        /**
         * @return Returns the firstName.
         */

        public String getFirstName() {
            return firstName;
        }

        /**
         * @param firstName The firstName to set.
         */

        public void setFirstName(String firstName) {
            this.firstName = firstName;
        }

        /**
         * @return Returns the lastName.
         */

        public String getLastName() {
            return lastName;
        }

        /**
         * @param lastName The lastName to set.
         */

        public void setLastName(String lastName) {
            this.lastName = lastName;
        }
    }
}

```

## 创建DAO访问对象

1. 在src/com.jandar.service.dao新建IDAO.java接口，所有的DAO都继承该接口

```
package com.jandar.services.dao;
```

```
public interface IDAO {  
  
    }  
}
```

## 2 . 在src/com.jandar.service.dao下新建IUserDAO.java接口

```
public interface IUserDAO extends DAO {  
    List getUsers();  
    User getUser(Integer userid);  
    void saveUser(User user);  
    void removeUser(Integer id);  
}
```

该接口提供了访问对象的方法，

## 3 . 在src/com.jandar.service.dao.hibernate下新建UserDAOHibernate.java

```
import java.util.List;  
import org.apache.commons.logging.Log;  
import org.apache.commons.logging.LogFactory;  
import org.springframework.orm.hibernate.support.HibernateDaoSupport;  
import com.jandar.model.User;  
import com.jandar.service.dao.IUserDAO;  
  
public class UserDaoHibernate extends HibernateDaoSupport implements IUserDAO {  
  
    private Log log=LogFactory.getLog(UserDaoHibernate.class);  
        /* (非 Javadoc )  
        * @see com.jandar.dao.IUserDAO#getUsers()  
        */  
  
        public List getUsers() {  
            return getHibernateTemplate().find("from User");  
        }  
  
        /* (非 Javadoc )  
        * @see com.jandar.dao.IUserDAO#getUser(java.lang.Long)  
        */  
  
        public User getUser(Integer id) {  
            // TODO 自动生成方法存根  
            return (User) getHibernateTemplate().get(User.class,id);  
        }  
}
```

```

        /* (非 Javadoc)
        * @see com.jandar.dao.IUserDAO#saveUser(com.jandar.model.User)
        */

        public void saveUser(User user) {
            log.debug("xxxxxxx");
            System.out.println("yyyy");
            getHibernateTemplate().saveOrUpdate(user);
            if(log.isDebugEnabled())
            {
                log.debug("userId set to "+user.getId());
            }
        }

        /* (非 Javadoc)
        * @see com.jandar.dao.IUserDAO#removeUser(java.lang.Long)
        */

        public void removeUser(Integer id) {
            Object user=getHibernateTemplate().load(User.class,id);
            getHibernateTemplate().delete(user);
            if(log.isDebugEnabled()){
                log.debug("del user "+id);
            }
        }
    }

```

在这个类中实现了IUserDAO接口的方法，并且继承了HibernateDAOSupport类。这个类的作用是通过hibernate访问、操作对象，进而实现对数据库的操作。

# HIBERNATE - 符合Java习惯的关系数据库持久化

# Hibernate2 参考文档

## 2.1.2

---

### Table of Contents

#### [前言](#)

#### 1. [在Tomcat中快速上手](#)

- 1.1. [开始Hibernate之旅](#)
- 1.2. [第一个可持久化类](#)
- 1.3. [映射cat](#)
- 1.4. [与猫同乐](#)
- 1.5. [结语](#)

#### 2. [体系结构](#)

- 2.1. [总览](#)
- 2.2. [持久化对象标识 \( Persistent Object Identity \)](#)
- 2.3. [JMX集成](#)
- 2.4. [JCA支持](#)

#### 3. [SessionFactory配置](#)

- 3.1. [可编程配置方式](#)
- 3.2. [获取SessionFactory](#)
- 3.3. [用户自行提供JDBC连接](#)
- 3.4. [Hibernate提供的JDBC连接](#)
- 3.5. [其它配置属性](#)
  - 3.5.1. [SQL Dialects SQL 方言](#)
  - 3.5.2. [外连接抓取 \( Outer Join Fetching \)](#)
  - 3.5.3. [二进制流](#)
  - 3.5.4. [在控制台记录SQL](#)
  - 3.5.5. [自定义 ConnectionProvider](#)
  - 3.5.6. [常用数据库属性](#)
  - 3.5.7. [自定义CacheProvider](#)
  - 3.5.8. [事务策略](#)
  - 3.5.9. [绑定SessionFactory到JNDI](#)
  - 3.5.10. [查询语言替换](#)
- 3.6. [XML配置文件](#)
- 3.7. [Logging](#)

- 3.8. [NamingStrategy \(命名策略\)](#)
- 4. [持久化类\(Persistent Classes\)](#)
  - 4.1. [简单示例](#)
    - 4.1.1. [为持久化字段声明访问器\(accessors\)和是否可变的标志\(mutators\)](#)
    - 4.1.2. [实现一个默认的构造方法 \( constructor \)](#)
    - 4.1.3. [提供一个标识属性 \( identifier property \) \( 可选 \)](#)
    - 4.1.4. [建议使用不是final的类 \(可选\)](#)
  - 4.2. [继承 \( Inheritance \)](#)
  - 4.3. [持久化生命周期 \( Lifecycle \) 中的回调 \( Callbacks \)](#)
  - 4.4. [合法性检查 \( Validatable \)](#)
  - 4.5. [XDoclet示例](#)
- 5. [O/R Mapping基础](#)
  - 5.1. [映射声明\(Mapping declaration\)](#)
    - 5.1.1. [Doctype](#)
    - 5.1.2. [hibernate-mapping](#)
    - 5.1.3. [class](#)
    - 5.1.4. [id](#)
      - 5.1.4.1. [generator](#)
      - 5.1.4.2. [高/低位算法 \( Hi/Lo Algorithm \)](#)
      - 5.1.4.3. [UUID算法 \( UUID Algorithm \)](#)
      - 5.1.4.4. [标识字段和序列 \( Identity Columns and Sequences \)](#)
      - 5.1.4.5. [程序分配的标识符 \( Assigned Identifiers \)](#)
    - 5.1.5. [composite-id 联合ID](#)
    - 5.1.6. [识别器 \( discriminator \)](#)
    - 5.1.7. [版本 \( version \) \(可选\)](#)
    - 5.1.8. [时间戳 \( timestamp \) \(可选\)](#)
    - 5.1.9. [property](#)
    - 5.1.10. [多对一 \( many-to-one \)](#)
    - 5.1.11. [一对一](#)
    - 5.1.12. [组件 \( component \) , 动态组件 \( dynamic-component \)](#)
    - 5.1.13. [子类\(subclass\)](#)
    - 5.1.14. [连接子类 \( joined-subclass \)](#)
    - 5.1.15. [map, set, list, bag](#)
    - 5.1.16. [引用 \( import \)](#)
  - 5.2. [Hibernate 的类型](#)
    - 5.2.1. [实体 \( Entities \) 和值 \( values \)](#)
    - 5.2.2. [基本值类型](#)
    - 5.2.3. [持久化枚举 \( Persistent enum \) 类型](#)



- (reassociate)
- 8.7. [删除持久化对象](#)
- 8.8. [对象图 \( Graphs of objects \)](#)
  - 8.8.1. [自动管理生命周期的对象 \( lifecycle object \)](#)
  - 8.8.2. [通过可触及性决定持久化 \( Persistence by Reachability \)](#)
- 8.9. [清洗\(Flushing\) -- 这个词很难翻译，不能使用“刷新”，因为刷新一词已经被"refresh"使用了。有什么好的建议？](#)
- 8.10. [结束一个Session](#)
  - 8.10.1. [清洗\(Flush\)session](#)
  - 8.10.2. [提交事务](#)
  - 8.10.3. [关闭session](#)
  - 8.10.4. [处理异常](#)
- 8.11. [拦截器\(Interceptors\)](#)
- 8.12. [元数据\(Metadata\) API](#)
- 9. [父子关系\(Parent Child Relationships\)](#)
  - 9.1. [关于collections](#)
  - 9.2. [双向的一对多关系\(Bidirectional one to many\)](#)
  - 9.3. [级联 \( Cascades \)](#)
  - 9.4. [级联更新 \( Using cascading update\(\) \)](#)
  - 9.5. [结论](#)
- 10. [Hibernate查询语言\(Query Language\), 即HQL](#)
  - 10.1. [大小写敏感性\(Case Sensitivity\)](#)
  - 10.2. [from 子句](#)
  - 10.3. [联合 \( Associations \) 和连接 \( joins \)](#)
  - 10.4. [select子句](#)
  - 10.5. [统计函数\(Aggregate functions\)](#)
  - 10.6. [多态\(polymorphism\)](#)
  - 10.7. [where子句](#)
  - 10.8. [表达式\(Expressions\)](#)
  - 10.9. [order by 子句](#)
  - 10.10. [group by 子句](#)
  - 10.11. [子查询](#)
  - 10.12. [示例](#)
  - 10.13. [提示和技巧 \( Tips & Tricks \)](#)
- 11. [实例\(A Worked Example\)](#)
  - 11.1. [持久化类](#)
  - 11.2. [Hibernate 映射](#)
  - 11.3. [Hibernate 代码](#)
- 12. [性能提升 \( Improving Performance \)](#)

- 12.1. [用于延迟装载的代理](#)
- 12.2. [第二层缓存\(The Second Level Cache\)s](#)
  - 12.2.1. [映射\(Mapping\)](#)
  - 12.2.2. [只读缓存](#)
  - 12.2.3. [读/写缓存](#)
  - 12.2.4. [不严格的读/写缓存](#)
  - 12.2.5. [事务缓存 \( transactional \)](#)
- 12.3. [管理Session缓存](#)
- 12.4. [查询缓存\(Query Cache\)](#)
- 13. [理解集合类的性能 \( Understanding Collection Performance \)](#)
  - 13.1. [分类 \( Taxonomy \)](#)
  - 13.2. [Lists, maps 和sets用于更新效率最高](#)
  - 13.3. [Bag和list是反向集合类中效率最高的](#)
  - 13.4. [一次性删除\(One shot delete\)](#)
- 14. [条件查询\(Criteria Query\)](#)
  - 14.1. [创建一个Criteria实例](#)
  - 14.2. [缩小结果集范围](#)
  - 14.3. [对结果排序](#)
  - 14.4. [关联 \( Associations \)](#)
  - 14.5. [动态关联对象获取 \( Dynamic association fetching \)](#)
  - 14.6. [根据示例查询 \( Example queries \)](#)
- 15. [SQL查询](#)
  - 15.1. [创建一个基于SQL的Query](#)
  - 15.2. [别名和属性引用](#)
  - 15.3. [为SQL查询命名](#)
- 16. [继承映射\(Inheritance Mappings\)](#)
  - 16.1. [三种策略](#)
  - 16.2. [限制](#)
- 17. [事务和并行 \( Transactions And Concurrency \)](#)
  - 17.1. [配置, 会话和工厂 \( Configurations, Sessions and Factories \)](#)
  - 17.2. [线程和连接 \( Threads and connections \)](#)
  - 17.3. [乐观锁定 / 版本化 \( Optimistic Locking / Versioning \)](#)
    - 17.3.1. [使用长生命周期带有自动版本化的会话](#)
    - 17.3.2. [使用带有自动版本化的多个会话](#)
    - 17.3.3. [应用程序自己进行版本检查](#)
  - 17.4. [会话断开连接 \( Session disconnection \)](#)
  - 17.5. [悲观锁定 \( Pessimistic Locking \)](#)
- 18. [映射实例\(Mapping Examples\)](#)
  - 18.1. [雇员 / 雇主 \( Employer/Employee \)](#)

- 18.2. [作者 / 著作\(Author/Work\)](#)
- 18.3. [客户 / 订单 / 产品\(Customer/Order/Product\)](#)
- 19. [工具箱指南](#)
  - 19.1. [Schema 生成器 \( Schema Generation \)](#)
    - 19.1.1. [对schema定制化\(Customizing the schema\)](#)
    - 19.1.2. [运行该工具](#)
    - 19.1.3. [属性\(Properties\)](#)
    - 19.1.4. [使用Ant\(Using Ant\)](#)
    - 19.1.5. [对schema的增量更新\(Incremental schema updates\)](#)
    - 19.1.6. [用Ant来增量更新schema\(Using Ant for incremental schema updates\)](#)
  - 19.2. [代码生成 \( Code Generation \)](#)
    - 19.2.1. [配置文件\(可选\)](#)
    - 19.2.2. [meta属性](#)
    - 19.2.3. [基本的finder生成器 \( Basic finder generator \)](#)
    - 19.2.4. [基于Velocity的渲染器/生成器\(Velocity based renderer/generator\)](#)
  - 19.3. [映射文件生成器 \( Mapping File Generation \)](#)
    - 19.3.1. [运行此工具](#)
- 20. [最佳实践\(Best Practices\)](#)

# 前言

在今日的企业环境中，把面向对象的软件和关系数据库一起使用可能是相当麻烦、浪费时间的。Hibernate是一个面向Java环境的对象/关系数据库映射工具。对象/关系数据库映射(object/relational mapping (ORM))这个术语表示一种技术，用来把对象模型表示的对象映射到基于SQL的关系模型结构中去。

Hibernate不仅仅管理Java类到数据库表的映射，还提供数据查询和获取数据的方法，可以大幅度减少开发时人工使用SQL和JDBC处理数据的时间。Hibernate的目标是对于开发者通常的数据持久化相关的编程任务，解放其中的95%。

如果你对Hibernate和对象/关系数据库映射还是个新手，或者甚至对Java也不熟悉，请按照下面的步骤来学习。

- 阅读这个30分钟就可以结束的[Chapter 1, 在Tomcat中快速上手](#)，它使用Tomcat。
- 阅读[Chapter 2, 体系结构](#)来理解Hibernate可以使用的环境。
- 查看Hibernate发行包中的eg/目录，里面有一个简单的独立运行的程序。把你的JDBC驱动拷贝到lib/目录下，修改一下src/hibernate.properties,指定其中你的数据库的信息。进入命令行，切换到你的发行包的目录，输入ant eg(使用了Ant)，或者在Windows操作系统中使用build eg。
- 把这份参考文档作为你学习的主要信息来源。
- 在Hibernate 的网站上可以找到经常提问的问题与解答(FAQ)。
- 在Hibernate网站上还有第三方的演示、示例和教程的链接。
- Hibernate网站的“社区(Community Area)”是讨论关于设计模式以及很多整合方案(Tomcat, JBoss, Spring, Struts, EJB,等等)的好地方。
- 离线版本的Hibernate网站随着Hibernate发行包一起发布，位于doc/目录下。

如果你有问题，请使用Hibernate网站上链接的用户论坛。我们也提供一个JIRA问题追踪系统，来搜集bug报告和新功能请求。如果你对开发Hibernate有兴趣，请加入开发者的邮件列表。（译者注：目前Hibernate已经有一个中文的用户论坛，URL是<http://forum.hibernate.org.cn> 我们随时欢迎您的访问。）

## 翻译说明

=====

本文档的翻译是在网络上协作进行的，也会不断根据Hibernate的升级进行更新。提供此文档的目的是为了减缓学习Hibernate的坡度，而非代替原文档。我们建议所有有能力的读者都直接阅读英文原文。

若您对翻译有异议，或发现翻译错误，敬请不吝赐教，请到Hibernate中文论坛(<http://forum.hibernate.org.cn>)提出,或报告到如下email地址：caoxg at redsaga.com

第6章(集合类)、第7章(组件)是由jlinux翻译，第10章（父子关系）是由muziq翻译，第16章(事务和并行)、第17章（映射实例）是由liangchen翻译，其他各章节是由曹晓钢翻译的，第18、19、20章，bruce、robbin也有贡献。曹晓钢也进行了全书从2.0.4更新到2.1.1版本、2.1.2版本的工作。

更详细的翻译者与翻译更新情况，请查阅CVS目录下的TRANSLATE-LOG.TXT文件。

## 版权声明

=====

Hibernate英文文档属于Hibernate发行包的一部分，遵循LGPL协议。本翻译版本同样遵循LGPL协议。参与翻译的译者一致同意放弃除署名权外对本翻译版本的其它权利要求。

您可以自由链接、下载、传播此文档，或者放置在您的网站上，甚至作为产品的一部分发行。但前提是必须保证全文完整转载，包括完整的版权信息和作译者声明。这里“完整”的含义是，不能进行任何删除/增添/注解。若有删除/增添/注解，必须明确声明那些部分并非本文档的一部分。

## Chapter 1. 在Tomcat中快速上手

## 1.1. 开始Hibernate之旅

这份教程讨论如何在Apache Tomcat servlet容器中为web程序安装Hibernate 2.1。Hibernate在大多数主流J2EE应用服务器的受管理环境中都可以良好运作，也可以作为独立应用程序运行。在本例中的示例数据库系统是PostgreSQL 7.3,当然也可以很容易的换成Hibernate支持的其它16种数据库之一。

第一步是拷贝所有需要的运行库到Tomcat去。在这篇教程中，我们使用一个单独的web程序（webapps/quickstart）。我们要考虑全局库文件搜索路径（TOMCAT/common/lib）和本web应用程序上下文的类装载机搜索路径（对于jar来说是webapps/quickstart/WEB-INF/lib，对于class文件来说是webapps/quickstart/WEB-INF/classes）。我们把这两个类装载机级别分别称为全局类路径(global classpath)和上下文类路径(context classpath)。

- 首先，把数据库需要的JDBC驱动拷贝到全局类路径。这是tomcat附带的DBCP连接池软件所要求的。对于本教程来说，把pg73jdbc3.jar库文件（对应PostgreSQL 7.3和JDK 1.4)到全局类装载机路径去。如果你使用一个不同的数据库，拷贝相应的JDBC 驱动）。
- 不要拷贝任何其他东西到全局类装载机去。否则你可能在一些工具上遇到麻烦，比如log4j, commons-logging等。记得要使用每个web应用程序自己的上下文类路径，就是说把你自己的类库拷贝到WEB-INF/lib下去，把配置文件configuration/property拷贝到WEB-INF/classes下面去。这两个目录默认都是上下文类路径级别的。
- Hibernate本身打包成一个JAR库。hibernate2.jar文件要和你应用程序的其他库文件一起放在上下文类路径中。在运行时，Hibernate还需要一些第三方库，它们在Hibernate发行包的lib/目录下。参见Table 1.1。把你需要的第三方库文件也拷贝到上下文类路径去。
- 要为Tomcat和Hibernate都配置数据库连接。也就是说Tomcat要负责提供JDBC连接池，Hibernate通过JNDI来请求这些连接。Tomcat把连接池绑定到JNDI。

Table 1.1. Hibernate 第三方库

库	描述
---	----

dom4j (必需)	Hibernate在解析XML配置和XML映射元文件时需要使用dom4j。
CGLIB (必需)	Hibernate在运行时使用这个代码生成库强化类（与Java反射机制联合使用）。
Commons Collections, Commons Logging (必需)	Hibernat使用Apache Jakarta Commons项目提供的多个工具类库。
ODMG4 (必需)	Hibernate提供了一个可选的ODMG兼容持久化管理界面。如果你需要映射集合，你就需要这个类库，就算你不是为了使用ODMG API。我们在这个教程中没有使用集合映射，但不管怎样把这个JAR拷贝过去总是不错的。
Log4j (可选)	Hibernate使用Commons Logging API,后者可以使用Log4j作为实施log的机制。如果把Log4j库放到上下文类目录中，Commons Logging就会使用Log4j和它在上下文类路径中找到的log4j.properties文件。在Hibernate发行包中包含有一个示例的properties文件。所以，也把log4j.jar拷贝到你的上下文类路径去吧。
其他文件是不是必需的？	请察看Hibernate发行包中的/lib/README.txt文件。这是一个Hibernate发行包中附带的第三方类库的列表，总是保持更新。你可以在那里找到所有必需或者可选的类库的列表。

好了，现在所有的类库已经被拷贝过去了，让我们在Tomcat的主配置文件,TOMCAT/conf/server.xml中增加一个数据库JDBC连接池的资源声明，

```
<Context path="/quickstart" docBase="quickstart">
  <Resource name="jdbc/quickstart" scope="Shareable"
  <ResourceParams name="jdbc/quickstart">
    <parameter>
      <name>factory</name>
      <value>org.apache.commons.dbcp.BasicDataSc
    </parameter>

    <!-- DBCP database connection settings -->
    <parameter>
```

```

        <name>url</name>
        <value>jdbc:postgresql://localhost/quickst
</parameter>
<parameter>
    <name>driverClassName</name><value>org.pos
</parameter>
<parameter>
    <name>username</name>
    <value>quickstart</value>
</parameter>
<parameter>
    <name>password</name>
    <value>secret</value>
</parameter>

<!-- DBCP connection pooling options -->
<parameter>
    <name>maxWait</name>
    <value>3000</value>
</parameter>
<parameter>
    <name>maxIdle</name>
    <value>100</value>
</parameter>
<parameter>
    <name>maxActive</name>
    <value>10</value>
</parameter>
</ResourceParams>
</Context>

```

这个例子中我们要配置的上下文叫做quickstart，它位于TOMCAT/webapp/quickstart目录。要访问任何Servlet,在你的浏览器中访问<http://localhost:8080/quickstart>就可以了。

Tomcat在这个配置下，使用DBCP连接池，通过JNDI位置：`java:comp/env/jdbc/quickstart`提供带有缓冲池的JDBCConnections。如果你在让连接池工作的时候遇到困难，请查阅Tomcat文档。如果你得到了JDBC驱动的exception信息，请先不要用Hibernate,测试JDBC连接池本身是

否正确。Tomcat和JDBC的教程可以在Web上查到。

下一步是配置hibernate，来使用绑定到JNDI的连接池中提供的连接。我们使用XML格式的Hibernate配置。当然，使用properties文件的方式在功能上也是一样的，也不提供什么特别好处。我们用XML配置的原因，是因为一般会更方便。XML配置文件放在上下文类路径(WEB-INF/classes)下面，称为hibernate.cfg.xml:

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration
    PUBLIC "-//Hibernate/Hibernate Configuration DTD//
    "http://hibernate.sourceforge.net/hibernate-config

<hibernate-configuration>

    <session-factory>

        <property name="connection.datasource">java:co
        <property name="show_sql">>false</property>
        <property name="dialect">net.sf.hibernate.dial

        <!-- Mapping files -->
        <mapping resource="Cat.hbm.xml"/>

    </session-factory>

</hibernate-configuration>
```

我们关闭了SQL命令的log,告诉Hibernate使用哪种SQL数据库方言(dialect),还有如何得到JDBC连接(通过声明数据源池绑定的JNDI地址)。方言是必需的,因为不同的数据库都和SQL"标准"有一些出入。Hibernate会替你照管这些差异之处,发行包包含了所有主流的商业和开放源代码数据库的方言。

SessionFactory是Hibernate的概念,对应一个数据存储源,如果有多个数据库,可以创建多个XML配置文件,也在你的程序中创建多个Configuration和SessionFactory对象。

在hibernate.cfg.xml中的最后一个元素声明了Cat.hbm.xml是一个Hibernate

XML映射文件，对应持久化类cat。这个文件包含了把POJO类映射到数据库表（或多个数据库表）的元数据。我们稍后就回来看这个文件。让我们先编写这个POJO类，再在声明它的映射元数据。

## 1.2. 第一个可持久化类

通过 Hibernate 让普通的 Java 对象 (Plain Old Java Objects, 就是 POJOs, 有时候也称为 Plain Ordinary Java Objects) 变成持久化类。一个 POJO 很像 JavaBean, 属性通过 getter 和 setter 方法访问, 对外隐藏了内部实现的细节。

```
package net.sf.hibernate.examples.quickstart;

public class Cat {

    private String id;
    private String name;
    private char sex;
    private float weight;

    public Cat() {
    }

    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public char getSex() {
        return sex;
    }
}
```

```
public void setSex(char sex) {
    this.sex = sex;
}

public float getWeight() {
    return weight;
}

public void setWeight(float weight) {
    this.weight = weight;
}
}
```

Hibernate对属性使用的类型不加限制。所有的Java JDK类型和原始类型（比如String,char和float）都可以被映射，也包括Java 集合框架（Java collections framework）中的类。你可以把它们映射成为值，值集合，或者其他实体相关联。id是一个特殊的属性，代表了这个类的数据库标识符(主键)，它对于类似于cat这样的实体是必需的。

持久化类不需要实现什么特别的接口，也不需要从一个特别的持久化根类继承下来。Hibernate也不需要使用任何编译期处理，比如字节码增强操作，它独立的使用Java反射机制和运行时类增强（通过CGLIB）。所以，在Hibernate中，POJO的类不需要任何前提条件，我们就可以把它映射成为数据库表。

## 1.3. 映射cat

Cat.hbm.xml映射文件包含了对象/关系映射所需的元数据。

元数据包含了持久化类的声明和把它与其属性映射到数据库表的信息（属性作为值或者是指向其他实体的关联）。

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping
  PUBLIC "-//Hibernate/Hibernate Mapping DTD//EN"
  "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" [
<hibernate-mapping>

  <class name="net.sf.hibernate.examples.quickstart.Cat">

    <!-- A 32 hex character is our surrogate key.
      generated by Hibernate with the UUID pattern. -->
    <id name="id" type="string" unsaved-value="null">
      <column name="CAT_ID" sql-type="char(32)">
        <generator class="uuid.hex"/>
      </column>
    </id>

    <!-- A cat has to have a name, but it shouldn't be null -->
    <property name="name">
      <column name="NAME" sql-type="varchar(16)">
        <not-null/>
      </column>
    </property>

    <property name="sex"/>

    <property name="weight"/>

  </class>

</hibernate-mapping>
```

每个持久化类都需要一个标识属性（实际上，只是哪些代表一手对象的

类，而不是代表值对象的类，后者会被映射称为一手对象中的一个组件)。这个属性用来区分持久化对象：如果 `catA.getId().equals(catB.getId())` 结果是 `true` 的话，两只猫就是相同的。这个概念称为数据库标识。Hibernate 附带了几种不同的标识符生成器，用于不同的场合（包括数据库本地的顺序(sequence)生成器和hi/lo高低位标识模式）。我们在这里使用UUID生成器，并指定CAT表的CAT\_ID字段（作为表的主键）存放生成的标识值。

cat的其他属性都映射到同一个表。对name属性来说，我们把它显式地声明映射到一个数据库字段。如果数据库schema是由映射声明使用Hibernate的SchemaExport工具自动生成的（作为SQL DDL指令），这特别有用。所有其它的属性都用Hibernate的默认值映射，大多数情况你都会这样做。数据库中的CAT表看起来是这样的：

Column	Type	Modifiers
cat_id	character(32)	not null
name	character varying(16)	not null
sex	character(1)	
weight	real	

Indexes: cat\_pkey primary key btree (cat\_id)

你现在可以在你的数据库中首先创建这个表了，如果你需要使用SchemaExport工具把这个步骤自动化，请参阅[Chapter 19, 工具箱指南](#)。这个工具能够创建完整的SQL DDL，包括表定义，自定义的字段类型约束，惟一约束和索引。

## 1.4. 与猫同乐

我们现在可以开始Hibernate的Session了。我们用它来从数据库中存取Cat。首先，我们要从SessionFactory中获取一个Session(Hibernate的工作单元)。

```
SessionFactory sessionFactory =  
    new Configuration().configure().buildSessioni
```

SessionFactory负责一个数据库，也只对应一个XML配置文件（hibernate.cfg.xml）。

这篇教程的关注点在于配置Tomcat的JDBC连接，绑定到JNDI上，以及Hibernate的基础配置。你可以用喜欢的任何方式编写一个Servlet,包含下面的代码，只要确保SessionFactory只创建一次。也就是说你不能把它作为你的Servlet的实例变量。一个好办法是用在辅助类中用一个静态的SessionFactory，例如这样：

```
import net.sf.hibernate.*;  
import net.sf.hibernate.cfg.*;  
  
public class HibernateUtil {  
  
    private static final SessionFactory sessionFactory  
  
    static {  
        try {  
            sessionFactory = new Configuration().confi  
        } catch (HibernateException ex) {  
            throw new RuntimeException("Exception buil  
        }  
    }  
  
    public static final ThreadLocal session = new Thre  
  
    public static Session currentSession() throws Hibe  
        Session s = (Session) session.get();
```

```

        // Open a new Session, if this Thread has none
        if (s == null) {
            s = sessionFactory.openSession();
            session.set(s);
        }
        return s;
    }

    public static void closeSession() throws HibernateException {
        Session s = (Session) session.get();
        session.set(null);
        if (s != null)
            s.close();
    }
}

```

这个类不但在它的静态属性中使用了SessionFactory，还使用了ThreadLocal来为当前工作线程保存Session。

Session不是线程安全的，代表与数据库之间的一次操作。Session通过SessionFactory打开，在所有的工作完成后，需要关闭：

```

Session session = HibernateUtil.currentSession();

Transaction tx= session.beginTransaction();

Cat princess = new Cat();
princess.setName("Princess");
princess.setSex('F');
princess.setWeight(7.4f);

session.save(princess);
tx.commit();

HibernateUtil.closeSession();

```

在Session中，每个数据库操作都是在一个事务(transaction)中进行的，这样就可以隔离不同的操作（甚至包括只读操作）。我们使用Hibernate的

Transaction API来从底层的事务策略中（本例中是JDBC事务）脱身。这样，如果需要把我们的程序部署到一个由容器管理事务的环境中去（使用JTA），我们就不需要更改源代码。请注意，我们上面的例子没有处理任何异常。

也请注意，你可以随心所欲的多次调用`HibernateUtil.currentSession()`，你每次都会得到同一个当前线程的`Session`。你必须确保`Session`在你的数据库事务完成后关闭，不管是在你的`Servlet`代码中，或者在`ServletFilter`中，`HTTP`结果返回之前。

Hibernate有不同的方法来从数据库中取回对象。最灵活的方式是使用Hibernate查询语言(HQL),这是一种容易学习的语言，是对SQL的面向对象的强大扩展。

```
Transaction tx= session.beginTransaction();

Query query = session.createQuery("select cat from Cat");
query.setCharacter("sex", 'F');
for (Iterator it = query.iterate(); it.hasNext();) {
    Cat cat = (Cat) it.next();
    out.println("Female Cat: " + cat.getName() );
}

tx.commit();
```

Hibernate也提供一种面向对象的按条件查询API，可以执行公式化的类型安全的查询。当然，Hibernate在所有与数据库的交互中都使用`PreparedStatement`和参数绑定。

## 1.5. 结语

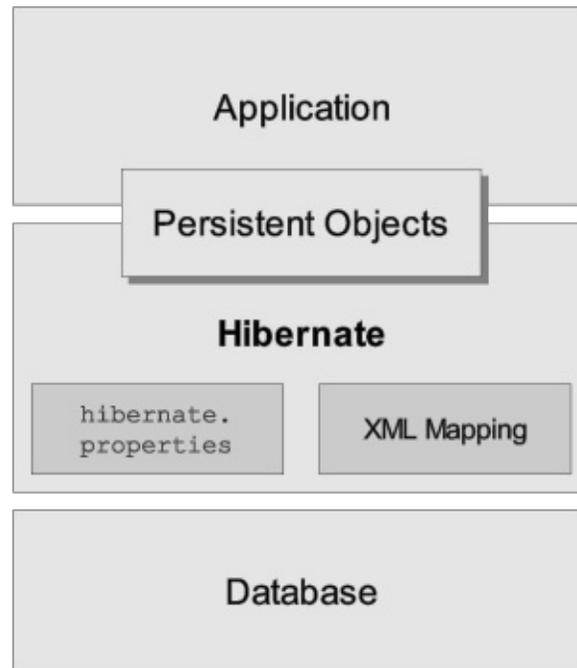
在这个短小的教程中，我们只描绘了Hibernate的基本面目。请注意我们没有在例子中包含Servlet相关代码。你必须自行编写Servlet，然后把你认为合适的Hibernate代码插入。

请记住Hibernate作为数据库访问层，是与你的程序紧密相关的。一般，所有其他层次都依赖持久机制。请确信你理解了这种设计的含义。

## Chapter 2. 体系结构

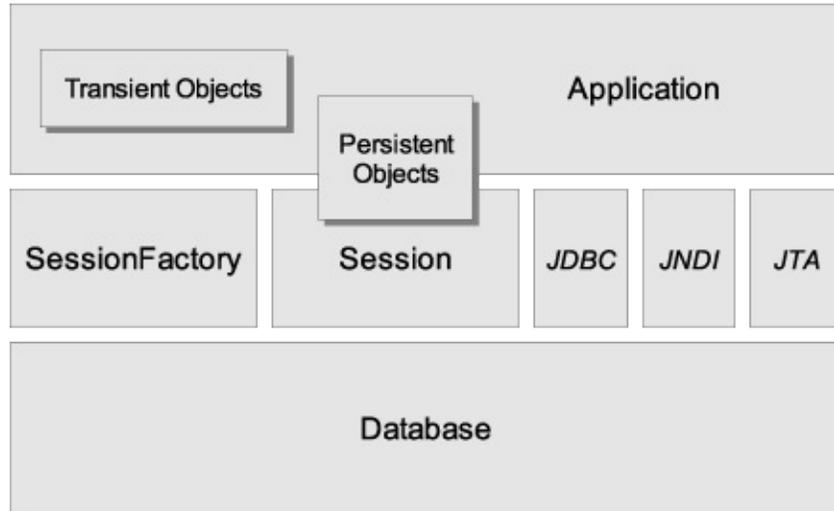
## 2.1. 总览

对Hibernate非常高层的概览：

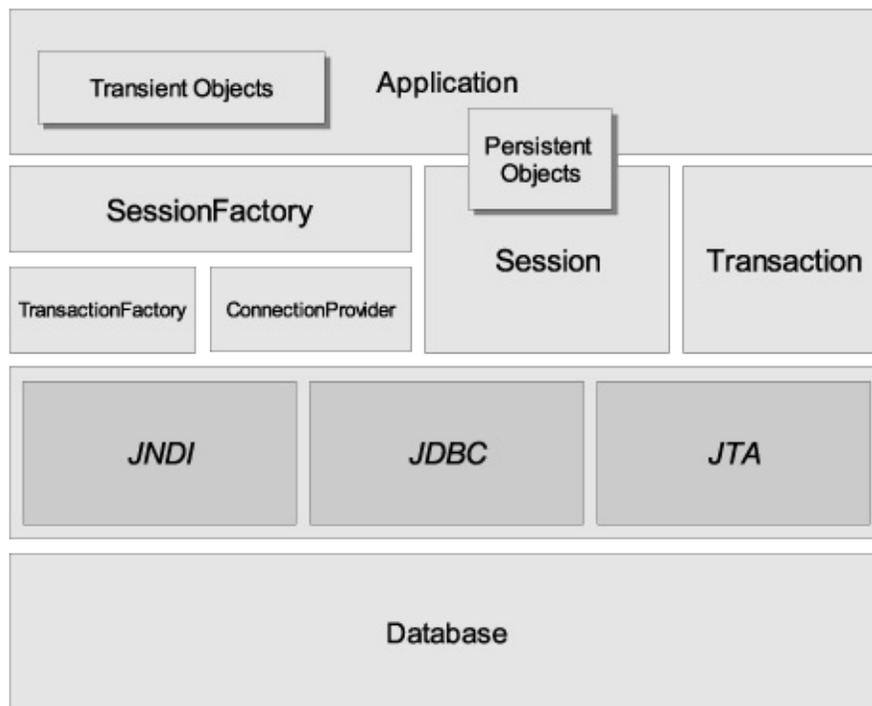


这幅图展示了Hibernate使用数据库和配置文件数据来为应用程序提供持久化服务（和持久化的对象）。

让我们更细致地观察一下运行时的体系结构。挺不幸的，Hibernate是比较复杂的，提供了好几种不同的运行方式。我们展示一下两种极端情况。轻型体系中，应用程序自己提供JDBC连接，并且自行管理事务。这种方式使用了Hibernate API的一个最小子集。



全面解决体系中，对于应用程序来说，所有的底层JDBC/JTA API都被抽象了，Hibernate会替你照管所有的细节。



下面是图中一些对象的定义：

**SessionFactory** (`net.sf.hibernate.SessionFactory`)

对编译过的映射文件的一个线程安全的，不可变的缓存快照。它是Session的工厂。是ConnectionProvider的客户。

可能持有事务之间重用的数据的缓存。

### 会话，**Session** (`net.sf.hibernate.Session`)

单线程，生命期短促的对象，代表应用程序和持久化层之间的一次对话。封装了一个JDBC连接。也是Transaction的工厂。

持有持久化对象的缓存。

### 持久化对象（**Persistent Object**）及其集合（**Collection**）

生命期短促的单线程的对象，包含了持久化状态和商业功能。它们可能是普通的JavaBeans，唯一特别的是他们现在从属于且仅从属于一个Session。

### 临时对象（**Transient Object**）及其集合（**Collection**）

目前没有从属于一个Session的持久化类的实例。他们可能是刚刚被程序实例化，还没有来得及被持久化，或者是被一个已经关闭的Session所实例化的。

### 事务，**Transaction** (`net.sf.hibernate.Transaction`)

（可选）单线程，生命期短促的对象，应用程序用它来表示一批工作的原子操作。是底层的JDBC,JTA或者CORBA事务的抽象。一个Session可能跨越多个Transaction 事务。

### **ConnectionProvider** (`net.sf.hibernate.connection.ConnectionProvider`)

（可选）JDBC连接的工厂和池。从底层的DataSource或者DriverManager抽象而来。对应用程序不可见。

### **TransactionFactory** (`net.sf.hibernate.TransactionFactory`)

（可选）事务实例的工厂。对应用程序不可见。

在上面的轻型结构中，程序没有使用Transaction / TransactionFactory 或者ConnectionProvider API,直接和JTA/JDBC对话了。

## 2.2. 持久化对象标识 ( Persistent Object Identity )

应用程序可能同时在两个不同的session中存取同一个持久化对象。然而，两个Session实例是不可能共享一个持久化类的实例的。有两种不同的用来辨别对象是否相同的方法。

### Persistent Identity , 持久化辨别

```
foo.getId().equals( bar.getId() )
```

### JVM Identity , JVM辨别

```
foo==bar
```

对于同一个特定的Session返回的对象来说，这二者是等价的。然而，当程序并行在两个不同的session中访问含义上“相同”（持久化辨别）的商业对象时，两个对象实例从JVM的角度上来看却是“不同”的（JVM辨别）

这种方式把并行访问（应用程序不需要对任何商业对象进行同步，只要求遵循每个Session一个线程的原则）和对象辨别（在应用程序的一个session之中，可以安全的用==来比较对象）的难题留给了Hibernate和数据库。

## 2.3. JMX集成

JMX是用来管理Java组件的J2EE标准。Hibernate可以被标准的JMX Mbean管理，但是因为大多数程序还没有支持JMX,Hibernate也支持一些非标准的配置方式。

请查阅Hibernate网站，可以得到关于如何在JBOSS中把Hibernate配置成为一个JMX组件的更多信息。

## 2.4. JCA支持

Hibernate也可以被配置成为一个JCA连接器。更多细节，请参阅网站。

## Chapter 3. SessionFactory配置

因为Hibernate被设计为可以在许多不同环境下工作，所以它有很多配置参数。幸运的是，大部分都已经有了默认值了，Hibernate发行包中还附带有示例的hibernate.properties文件，它演示了一些可变的参数。

## 3.1. 可编程配置方式

`net.sf.hibernate.cfg.Configuration`的一个实例代表了应用程序中所有的Java类到关系数据库的映射的集合。这些映射是从一些XML映射文件中编译得来的。你可以得到一个`Configuration`的实例，直接实例化它即可。下面有一个例子，用来从两个XML配置文件中的映射中初始化：

```
Configuration cfg = new Configuration()
    .addFile("Vertex.hbm.xml")
    .addFile("Edge.hbm.xml");
```

另外一个（或许是更好的）方法是让Hibernate自行用`getResourceAsStream()`来装载映射文件。

```
Configuration cfg = new Configuration()
    .addClass(eg.Vertex.class)
    .addClass(eg.Edge.class);
```

Hibernate 就会在classpath中寻找叫做`/eg/Vertex.hbm.xml`、`/eg/Edge.hbm.xml`的映射文件。这种方法取消了所有对文件名的硬编码。

`Configuration`也可以指定一些可选的配置项。

```
Properties props = new Properties();
...
Configuration cfg = new Configuration()
    .addClass(eg.Vertex.class)
    .addClass(eg.Edge.class)
    .setProperties(props);
```

`Configuration`是仅在配置期使用的对象，从第一个`SessionFactory`开始建立的时候，它就失效了。

## 3.2. 获取SessionFactory

当所有的映射都被`Configuration`解析之后，应用程序为了得到`Session`实例，必须先得到它的工厂。这个工厂应该是被应用程序的所有线程共享的。当然，Hibernate并不禁止你的程序实例化多个`SessionFactory`。在你使用不止一个数据库的时候，这就有用了。

```
SessionFactory sessions = cfg.buildSessionFactory();
```

### 3.3. 用户自行提供JDBC连接

SessionFactory可以使用一个用户自行提供的JDBC连接来打开一个Session。这种设计可以让应用程序来自己管理JDBC连接。应用程序必须小心，不能在同一个连接上打开多个并行的session。

```
java.sql.Connection conn = datasource.getConnection();
Session sess = sessions.openSession(conn);

// start a new transaction (optional)
Transaction tx = sess.beginTransaction();
```

上面的最后一行是可选的——应用程序也可能选择自行管理JTA或者JDBC事务。当然，假若你使用Hibernate Transaction，你的客户代码就可以从底层的实现中抽象出来了。（比如说，你可以将来在需要的时候切换到CORBA连接，而不需要更改程序代码。）

## 3.4. Hibernate提供的JDBC连接

另一种方法就是，你可以让SessionFactory替你打开连接。SessionFactory必须事先知道连接的参数，有几种不同的方法设置参数：

- 传递一个java.util.Properties到Configuration.setProperties()方法。
- 在classpath的根目录中提供hibernate.properties文件。
- 通过java -Dproperty=value指定使用系统属性。
- 在hibernate.cfg.xml文件中包含<property>元素。详情见后。

如果你使用这种方法，打开一个Session是非常简单的：

```
Session sess = sessions.openSession(); // obtain a JDE
// instantiate
// start a new transaction (optional)
Transaction tx = sess.beginTransaction();
```

所有的Hibernate属性名和约束都在net.sf.hibernate.cfg.Environment类中定义。我们讨论一下最重要的几项设置：

假若你设置了如下的属性，Hibernate会使用java.sql.DriverManager来得到连接，并建立连接池：

**Table 3.1. Hibernate JDBC属性**

属性名	用途
hibernate.connection.driver_class	<i>jdbc</i> 驱动类
hibernate.connection.url	<i>jdbc</i> URL
hibernate.connection.username	数据库用户名
hibernate.connection.password	数据库用户密码
hibernate.connection.pool_size	连接池容量最大数

Hibernate的连接池算法是非常可配置的。它的用途是让你上手，但是并非让你在生产系统中使用的,甚至不是用来做性能测试的。

C3P0是随Hibernate发行包一起发布的一个开放源代码JDBC连接池，你可以在lib 目录中找到。假若你设置了hibernate.c3p0.\* 属性，Hibernate会使用内置的C3P0ConnectionProvider作为连接池。对Apache DBCP和Proxool的支持也是内置的。你必须设置hibernate.dbcp.\*属性（DBCP连接池属性）和hibernate.dbcp.ps.\* (DBCP 语句缓存属性)才能使用DBCPCConnectionProvider。要知道它们的含义，请查阅Apache commons-pool的文档。如果你想要用Proxool,你需要设置hibernate.proxool.\*系列属性。

在Application Server内使用时，Hibernate可以从JNDI中注册的javax.sql.DataSource取得连接。需要设置如下属性：

**Table 3.2. Hibernate 数据源（DataSource）属性**

属性名	用途
hibernate.connection.datasource	<i>datasource</i> JNDI 名字
hibernate.jndi.url	JNDI 提供者的URL (可选)
hibernate.jndi.class	JNDI <i>InitialContextFactory</i> 的类名 (可选)
hibernate.connection.username	数据库用户名 (可选)
hibernate.connection.password	数据库密码 (可选)

## 3.5. 其它配置属性

下面是一些在运行时可以改变Hibernate行为的其他配置。所有这些都是可选的，也有合理的默认值。

系统级别的配置只能通过`java -Dproperty=value`或者在`hibernate.properties`文件中配置，而不能通过传递给`Configuration`的`Properties`实例来配置。

**Table 3.3. Hibernate配置属性**

属性名	用途
<code>hibernate.dialect</code>	<i>Hibernate</i> 方言 ( <i>Dialect</i> ) 的类名 让 <i>Hibernate</i> 使用某些特定的数据库的特性 取值. <code>full.classname.of.Dialect</code>
<code>hibernate.default_schema</code>	在生成的SQL中， <i>schema/table</i> 全限定名 取值. <code>SCHEMA_NAME</code>
<code>hibernate.session_factory_name</code>	把 <i>SessionFactory</i> 绑定到JNDI 取值. <code>jndi/composite/name</code>
<code>hibernate.use_outer_join</code>	允许使用外连接抓取。 取值. <code>true   false</code>
<code>hibernate.max_fetch_depth</code>	设置外连接抓取树的最大深度 取值. 建议设置为0到3之间
<code>hibernate.jdbc.fetch_size</code>	一个非零值，用来决定JDBC的 大小。(会调用 <i>calls</i> )

hibernate.jdbc.batch\_size

*Statement.setFetchSize()*).

一个非零值，会开启Hibernate JDBC2的批量更新功能

取值. 建议在 5 和 30之间。

hibernate.jdbc.use\_scrollable\_resultset

允许Hibernate使用JDBC2提供滚动结果集。只有在使用用户自己的连接时，这个参数才是必需。否则Hibernate会使用连接的元数据(*metadata*)。

取值. true | false

hibernate.jdbc.use\_streams\_for\_binary

在从JDBC读写*binary* (二进制) *serializable* (可序列化) 类型时用*stream*(流)。这是一个系统级属性。

取值. true | false

hibernate.cglib.use\_reflection\_optimizer

是否使用CGLIB来代替运行时作。(系统级属性，默认为时都使用CGLIB)。在调试的时候使用反射会有用。

取值. true | false

hibernate.jndi.<propertyName>

把*propertyName*这个属性传递到*InitialContextFactory*去 (可选) 事务隔离级别 (可选)

hibernate.connection.isolation

取值. 1, 2, 4, 8

hibernate.connection.<propertyName>

把 *propertyName*这个JDBC 属性到*DriverManager.getConnection*

hibernate.connection.provider_class	<p>指定一个自定义的<code>ConnectionFactory</code>类名</p> <p>取值. classname.of.Connection</p>
hibernate.cache.provider_class	<p>指定一个自定义的<code>CacheProvider</code>提供者的类名</p> <p>取值. classname.of.CacheProvi</p>
hibernate.cache.use_minimal_puts	<p>优化第二层缓存操作，减少写代价是读操作更频繁（对于集很有用）</p> <p>取值. true false</p>
hibernate.cache.use_query_cache	<p>打开查询缓存</p> <p>取值. true false</p>
hibernate.cache.region_prefix	<p>用于第二层缓存区域名字的前</p> <p>取值. prefix</p>
hibernate.transaction.factory_class	<p>指定一个自定义的<code>Transaction</code>类名，<code>Hibernate Transaction A</code>使用</p> <p>取值. classname.of.Transactio</p>
jta.UserTransaction	<p><code>JTATransactionFactory</code> 用来获<code>UserTransaction</code>的JNDI名</p> <p>取值. jndi/composite/name</p>
	<p><code>TransactionManagerLookup</code>的类 JTA环境中，JVM级别的缓存</p>

hibernate.transaction.manager\_lookup\_class 时候使用

取值.  
classname.of.TransactionManag

把Hibernate查询中的一些短语  
SQL短语（比如说短语可能是  
者字符）。

hibernate.query.substitutions

取值. hqlLiteral=SQL\_LITERAL,  
hqlFunction=SQLFUNC

把所有的SQL语句都输出到控  
以作为log功能的一个替代)

hibernate.show\_sql

取值. true | false

自动输出schema创建DDL语句

hibernate.hbm2ddl.auto

取值. update | create | create-d

### 3.5.1. SQL Dialects SQL 方言

你总是可以为你的数据库设置一个hibernate.dialect方言，它是net.sf.hibernate.dialect.Dialect的一个子类。如果你不需要使用基于native或者sequence的主键自动生成算法，或者悲观锁定（使用Session.lock()或Query.setLockMode())的话，方言就可以不必指定。然而，假若你指定了一个方言，Hibernate会为上面列出的一些属性使用特殊默认值，省得你手工指定它们。

Table 3.4. Hibernate SQL 方言 (hibernate.dialect)

RDBMS	方言
DB2	net.sf.hibernate.dialect.DB2Dialect
MySQL	net.sf.hibernate.dialect.MySQLDialect
SAP DB	net.sf.hibernate.dialect.SAPDBDialect

Oracle (所有版本)	<code>net.sf.hibernate.dialect.OracleDialect</code>
Oracle 9	<code>net.sf.hibernate.dialect.Oracle9Dialect</code>
Sybase	<code>net.sf.hibernate.dialect.SybaseDialect</code>
Sybase Anywhere	<code>net.sf.hibernate.dialect.SybaseAnywhereDialect</code>
Progress	<code>net.sf.hibernate.dialect.ProgressDialect</code>
Mckoi SQL	<code>net.sf.hibernate.dialect.MckoiDialect</code>
Interbase	<code>net.sf.hibernate.dialect.InterbaseDialect</code>
Pointbase	<code>net.sf.hibernate.dialect.PointbaseDialect</code>
PostgreSQL	<code>net.sf.hibernate.dialect.PostgreSQLDialect</code>
HypersonicSQL	<code>net.sf.hibernate.dialect.HSQLDialect</code>
Microsoft SQL Server	<code>net.sf.hibernate.dialect.SybaseDialect</code>
Ingres	<code>net.sf.hibernate.dialect.IngresDialect</code>
Informix	<code>net.sf.hibernate.dialect.InformixDialect</code>
FrontBase	<code>net.sf.hibernate.dialect.FrontbaseDialect</code>

### 3.5.2. 外连接抓取 ( Outer Join Fetching )

如果你的数据库支持ANSI或者Oracle风格的外连接，外连接抓取可能提高性能，因为可以限制和数据库交互的数量（代价是数据库自身进行了更多的工作）。外连接抓取允许你在一个select语句中就可以得到一个由多对一或者一对一连接构成的对象图。

默认情况下，抓取在叶对象，拥有代理的对象或者产生对自身的引用时终止。对一个特定关联来说，通过在XML映射文件中设置`outer-join`属性可以控制是否开启抓取功能。也可以设置`hibernate.use_outer_join`为`false`来全局关闭此功能。你也可以通过`hibernate.max_fetch_depth`来设置抓取得对象图的最大深度。

### 3.5.3. 二进制流

Oracle限制通过它的JDBC驱动传递的byte数组的大小。如果你希望使用很

大数量的binary或者serializable 类型的话，你需要打开hibernate.jdbc.use\_streams\_for\_binary。这只能通过JVM级别设定

### 3.5.4. 在控制台记录SQL

hibernate.show\_sql强制Hibernate把每一句SQL语句都写到控制台。这是作为打开log的一个简易替代。

### 3.5.5. 自定义 ConnectionProvider

你可以自定义你的获取JDBC连接的策略，只需要实现net.sf.hibernate.connection.ConnectionProvider接口。在hibernate.connection.provider\_class设置你自己的实现的类名。

### 3.5.6. 常用数据库属性

几个配置属性影响除了DataSourceConnectionProvider之外的所有内置连接提供者。它们是：hibernate.connection.driver\_class, hibernate.connection.url, hibernate.connection.username and hibernate.connection.password.

hibernate.connection.isolation应该指定为一个整数值。(查阅java.sql.Connection可以得到值的含义，但注意大多数数据库不会支持所有的隔离级别。)

专用的连接属性可以通过在"hibernate.connection"后面加上属性名来指定。比如，你可以通过hibernate.connection.charset指定一个charSet。

### 3.5.7. 自定义CacheProvider

通过实现net.sf.hibernate.cache.CacheProvider接口，你可以整合一个JVM级别（或者集群的）缓存进来。你可以通过hibernate.cache.provider\_class选择某个子定义的实现。

### 3.5.8. 事务策略

如果你希望使用Hibernate的Transaction API,你必须通过hibernate.transaction.factory\_class属性指定一个Transaction实例的工厂类。 内置的两个标准选择是：

`net.sf.hibernate.transaction.JDBCTransactionFactory`

使用数据库(JDBC)事务

`net.sf.hibernate.transaction.JTATransactionFactory`

使用JTA(假若已经存在一个事务，Session会在这个上下文中工作，否则会启动一个新的事务。)

你也可以自行定义你的事务策略（比如说，一个CORBA事务服务）。

如果你希望在JTA环境中为可变数据使用JVM级别的缓存，你必须指定一个获取JTA TransactionManager的策略。

**Table 3.5. JTA TransactionManagers**

事务工厂类	Applicati Server
<code>net.sf.hibernate.transaction.JBossTransactionManagerLookup</code>	JBoss
<code>net.sf.hibernate.transaction.WeblogicTransactionManagerLookup</code>	Weblogic
<code>net.sf.hibernate.transaction.WebSphereTransactionManagerLookup</code>	WebSphe
<code>net.sf.hibernate.transaction.OrionTransactionManagerLookup</code>	Orion
<code>net.sf.hibernate.transaction.ResinTransactionManagerLookup</code>	Resin
<code>net.sf.hibernate.transaction.JOTMTransactionManagerLookup</code>	JOTM
<code>net.sf.hibernate.transaction.JOnASTransactionManagerLookup</code>	JOnAS
<code>net.sf.hibernate.transaction.JRun4TransactionManagerLookup</code>	JRun4

### 3.5.9. 绑定SessionFactory到JNDI

假若你希望把SessionFactory绑定到一个JNDI命名空间，用hibernate.session\_factory\_name这个属性指定一个名字（比如，java:comp/env/hibernate/SessionFactory）。如果这个属性省略了，SessionFactory不会被绑定到JNDI。（在一个只读的JNDI默认值实现的环境中，这特别有用。比如，Tomcat。）

当把SessionFactory绑定到JNDI,Hibernate会使用hibernate.jndi.url,hibernate.jndi.class的值来获得一个初始化上下文的实例。如果他们没指定，就会使用默认的InitialContext。

如果你选择使用JNDI,EJB或者其他工具类就可以通过JNDI查询得到SessionFactory。

### 3.5.10. 查询语言替换

你可以使用hibernate.query.substitutions定义新的Hibernate查询短语。比如说：

```
hibernate.query.substitutions true=1, false=0
```

会在生成的SQL中把短语true和false替换成整数值。

```
hibernate.query.substitutions toLowercase=LOWER
```

这可以让你重新命名SQL的LOWER函数。

## 3.6. XML配置文件

另一种配置属性的方法是把所有的配置都放在一个名为hibernate.cfg.xml的文件中。这个文件应该放在你的CLASSPATH的根目录中。

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD 2.0/"
    "http://hibernate.sourceforge.net/hibernate-configuration-2.0.dtd" [ ]>
<hibernate-configuration>

    <!-- a SessionFactory instance listed as /jndi/name -->
    <session-factory
        name="java:comp/env/hibernate/SessionFactory">

        <!-- properties -->
        <property name="connection.datasource">my/first</property>
        <property name="dialect">net.sf.hibernate.dialect</property>
        <property name="show_sql">>false</property>
        <property name="use_outer_join">>true</property>
        <property name="transaction.factory_class">net.sf.hibernate.transaction</property>
        <property name="jta.UserTransaction">java:comp/env/jta/TransactionManager</property>

        <!-- mapping files -->
        <mapping resource="eg/Edge.hbm.xml"/>
        <mapping resource="eg/Vertex.hbm.xml"/>

    </session-factory>
</hibernate-configuration>
```

配置Hibernate只需如此简单：

```
SessionFactory sf = new Configuration().configure().buildSessionFactory();
```

你可以使用另外一个名字的配置文件：

```
SessionFactory sf = new Configuration()  
    .configure("catdb.cfg.xml")  
    .buildSessionFactory();
```

## 3.7. Logging

通过Apache commons-logging , Hibernate记录很多事件。commons-logging 服务会直接输出到Apache log4j(如果你把log4j.jar放在你的classpath里),或者JDK1.4 logging (如果你运行JDK 1.4或以上版本)。你可以从<http://jakarta.apache.org>下载log4j。要使用log4j,你需要在你的classpath中放置一个log4j.properties文件。Hibernate发行包中包含一个示例的properties配置文件。

我们强烈建议你熟悉Hibernate的log信息。Hibernate的很多工作都会尽量详细的留下log,也没有让它变的难以阅读。这是用来解决问题的最基本的设施。

## 3.8. NamingStrategy (命名策略)

`net.sf.hibernate.cfg.NamingStrategy` 接口允许你对数据库对象指定“命名标准”。你可以定义从Java标识符自动生成数据库标识符的规则，或者是映射文件中给出的“逻辑”字段名和表名处理为“物理”表名和字段名的规则。这个功能可以让映射文件变得简洁，消除无用的噪音（比如TBL\_前缀等）。Hibernate使用的默认策略是几乎什么都不错。你可以在增加映射(`add mappings`)之前调用`Configuration.setNamingStrategy()`来指定不同的策略。

```
SessionFactory sf = new Configuration()
    .setNamingStrategy(ImprovedNamingStrategy.INSTANCE)
    .addFile("Vertex.hbm.xml")
    .addFile("Edge.hbm.xml")
    .buildSessionFactory();
```

`net.sf.hibernate.cfg.ImprovedNamingStrategy` 是一个内置的策略，对某些程序，你可以把它作为改造的起点。

## Chapter 4. 持久化类(Persistent Classes)

## 4.1. 简单示例

大多数java程序需要一个持久化类的表示方法。

```
package eg;
import java.util.Set;
import java.util.Date;

public class Cat {
    private Long id; // identifier
    private Date birthdate;
    private Cat mate;
    private Set kittens
    private Color color;
    private char sex;
    private float weight;

    private void setId(Long id) {
        this.id=id;
    }
    public Long getId() {
        return id;
    }

    void setMate(Cat mate) {
        this.mate = mate;
    }
    public Cat getMate() {
        return mate;
    }

    void setBirthdate(Date date) {
        birthdate = date;
    }
    public Date getBirthdate() {
        return birthdate;
    }
}
```

```

void setWeight(float weight) {
    this.weight = weight;
}
public float getWeight() {
    return weight;
}

public Color getColor() {
    return color;
}
void setColor(Color color) {
    this.color = color;
}
void setKittens(Set kittens) {
    this.kittens = kittens;
}
public Set getKittens() {
    return kittens;
}
// addKitten not needed by Hibernate
public void addKitten(Cat kitten) {
    kittens.add(kitten);
}
void setSex(char sex) {
    this.sex=sex;
}
public char getSex() {
    return sex;
}
}

```

有三条主要的规则：

#### 4.1.1. 为持久化字段声明访问器(accessors)和是否可变的标志(mutators)

cat为它的所有可持久化字段声明了访问方法。很多其他ORM工具直接对实例变量进行持久化。我们相信在持久化机制中不限定这种实现细节，感

觉要好得多。Hibernate对JavaBeans风格的属性实行持久化，采用如下格式来辨认方法：`getFoo`, `isFoo` 和 `setFoo`。

属性不一定需要声明为`public`的。Hibernate可以对`default`,`protected`或者`private`的`get/set`方法对的属性一视同仁地执行持久化。

#### 4.1.2. 实现一个默认的构造方法（**constructor**）

`Cat`有一个显式的无参数默认构造方法。所有的持久化类都必须具有一个默认的构造方法（可以不是`public`的），这样的话Hibernate就可以使用`Constructor.newInstance()`来实例化它们。

#### 4.1.3. 提供一个标识属性（**identifier property**）（可选）

`Cat`有一个属性叫做`id`。这个属性包含了数据库表中的主关键字字段。这个属性可以叫任何名字，其类型可以是任何的原始类型、原始类型的包装类型、`java.lang.String` 或者是 `java.util.Date`。（如果你的老式数据库表有联合主键，你甚至可以用一个用户自定义的类，其中每个属性都是这些类型之一。参见后面的关于联合标识符的章节。）

用于标识的属性是可选的。你可以不管它，让Hibernate内部来追踪对象的识别。当然，对于大多数应用程序来说，这是一个好的（也是很流行的）设计决定。

更进一步，一些功能只能对声明了标识属性的类起作用：

- 级联更新（`Cascaded updates`）（参阅“自我管理生命周期的对象（`Lifecycle Objects`）”）
- `Session.saveOrUpdate()`

我们建议你对所有的持久化类采取同样的名字作为标识属性。更进一步，我们建议你使用一个可以为空（也就是说，不是原始类型）的类型。

#### 4.1.4. 建议使用不是**final**的类 (可选)

Hibernate的关键功能之一，代理（`proxies`），要求持久化类不是`final`的，

或者是一个全部方法都是public的接口的具体实现。

你可以对一个final的，也没有实现接口的类执行持久化，但是不能对它们使用代理——多多少少会影响你进行性能优化的选择。

## 4.2. 继承 ( Inheritance )

子类也必须遵守第一条和第二条规则。它从Cat继承了标识属性。

```
package eg;

public class DomesticCat extends Cat {
    private String name;

    public String getName() {
        return name;
    }
    protected void setName(String name) {
        this.name=name;
    }
}
```

## 4.3. 持久化生命周期 ( Lifecycle ) 中的回调 ( Callbacks )

作为一个可选的步骤，可持久化类可以实现Lifecycle接口，它可以提供一些用于回调的方法，可以让持久化对象在save或load之后，或者在delete或update之前进行必要的初始化与清除步骤。

```
public interface Lifecycle {
    public boolean onSave(Session s) throws Callba
    public boolean onUpdate(Session s) throws Call
    public boolean onDelete(Session s) throws Call
    public void onLoad(Session s, Serializable id)
}
```

❶

onSave - 在对象即将被save或者insert的时候回调

❷

onUpdate - 在对象即将被update的时候回调 ( 也就是对象被传递给Session.update()的时候)

❸

onDelete - 在对象即将被delete(删除)的时候回调

❹

onLoad - 在对象刚刚被load(装载)后的时候回调

onSave(), onDelete() 和 onUpdate() 可以被用来级联保存或者删除依赖的对象。这种做法是在映射文件中声明级联操作外的另外一种选择。onLoad() 可以用来让对象从其持久化 ( 当前 ) 状态中初始化某些暂时的属性。不能用这种方式来装载依赖的对象，因为可能无法在此方法内部调用Session接口。onLoad(), onSave()和onUpdate()另一种用法是用来在当前Session中保存一个引用，以备后用。

请注意onUpdate()并不是在每次对象的持久化状态被更新的时候就被调用

的。它只在处于尚未被持久化的对象被传递给`Session.update()`的时候才会被调用。

如果`onSave()`, `onUpdate()` 或者 `onDelete()`返回`true`，那么操作就被悄悄地取消了。如果其中抛出了`CallbackException`异常，操作被取消，这个异常会被继续传递给应用程序。

请注意`onSave()`是在标识符已经被赋予对象后调用的，除非是使用本地(`native`)方式生成关键字的。

## 4.4. 合法性检查 ( Validatable )

如果持久化类需要在保存其持久化状态前进行合法性检查，它可以实现下面的接口：

```
public interface Validatable {  
    public void validate() throws ValidationFailure;  
}
```

如果发现对象违反了某条规则，应该抛出一个`ValidationFailure`异常。在`Validatable`实例的`validate()`方法内部不应该改变它的状态。

和`Lifecycle`接口的回调方法不同，`validate()`可能在任何时间被调用。应用程序不应该把`validate()`调用和商业功能联系起来。

## 4.5. XDoclet示例

下一节中我们将会展示Hibernate映射是如何用简单的，可阅读的XML格式表达的。很多Hibernate用户喜欢使用XDoclet的@hibernate.tags标签直接在源代码中嵌入映射信息。我们不会在这份文档中讨论这个话题，因为严格的来说这属于XDoclet的一部分。但我们仍然在这里给出一份带有XDoclet映射的cat类的示例。

```
package eg;
import java.util.Set;
import java.util.Date;

/**
 * @hibernate.class
 *   table="CATS"
 */
public class Cat {
    private Long id; // identifier
    private Date birthdate;
    private Cat mate;
    private Set kittens
    private Color color;
    private char sex;
    private float weight;

    /**
     * @hibernate.id
     *   generator-class="native"
     *   column="CAT_ID"
     */
    public Long getId() {
        return id;
    }
    private void setId(Long id) {
        this.id=id;
    }
}
```

```
/**
 * @hibernate.many-to-one
 * column="MATE_ID"
 */
public Cat getMate() {
    return mate;
}
void setMate(Cat mate) {
    this.mate = mate;
}

/**
 * @hibernate.property
 * column="BIRTH_DATE"
 */
public Date getBirthdate() {
    return birthdate;
}
void setBirthdate(Date date) {
    birthdate = date;
}

/**
 * @hibernate.property
 * column="WEIGHT"
 */
public float getWeight() {
    return weight;
}
void setWeight(float weight) {
    this.weight = weight;
}

/**
 * @hibernate.property
 * column="COLOR"
 * not-null="true"
 */
public Color getColor() {
    return color;
}
```

```

}
void setColor(Color color) {
    this.color = color;
}
/**
 * @hibernate.set
 * lazy="true"
 * order-by="BIRTH_DATE"
 * @hibernate.collection-key
 * column="PARENT_ID"
 * @hibernate.collection-one-to-many
 */
public Set getKittens() {
    return kittens;
}
void setKittens(Set kittens) {
    this.kittens = kittens;
}
// addKitten not needed by Hibernate
public void addKitten(Cat kitten) {
    kittens.add(kitten);
}

/**
 * @hibernate.property
 * column="SEX"
 * not-null="true"
 * update="false"
 */
public char getSex() {
    return sex;
}
void setSex(char sex) {
    this.sex=sex;
}
}

```

## Chapter 5. O/R Mapping基础

## 5.1. 映射声明(Mapping declaration)

对象和关系数据库之间的映射是用一个XML文档(XML document)来定义的。这个映射文档被设计为易读的，并且可以手工修改。映射语言是以Java为中心的，意味着映射是按照持久化类的定义来创建的，而非表的定义。

请注意，虽然很多Hibernate用户选择手工定义XML映射文档，也有一些工具来生成映射文档，包括XDoclet, Middlegen和AndroMDA。

让我们从一个映射的例子开始：

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 2.0//EN"
    "http://hibernate.sourceforge.net/hibernate-ma
<hibernate-mapping package="eg">

    <class name="Cat" table="CATS" discriminator-v
        <id name="id" column="uid" type="long"
            <generator class="hilo"/>
        </id>
        <discriminator column="subclass" type=
        <property name="birthdate" type="date"
        <property name="color" not-null="true"
        <property name="sex" not-null="true" u
        <property name="weight"/>
        <many-to-one name="mate" column="mate_
        <set name="kittens">
            <key column="mother_id"/>
            <one-to-many class="Cat"/>
        </set>
        <subclass name="DomesticCat" discrimir
            <property name="name" type="st
        </subclass>
    </class>
```

```
        <class name="Dog">
            <!-- mapping for Dog could go here -->
        </class>
</hibernate-mapping>
```

我们现在开始讨论映射文档的内容。我们只描述Hibernate在运行时用到的文档元素和属性。映射文档还包括一些额外的可选属性和元素，它们在使用schema导出工具的时候会影响到导出的数据库schema结果。（比如，not-null 属性。）

### 5.1.1. Doctype

所有的XML映射都需要定义如上所示的doctype。DTD可以从上述URL中获取，或者在hibernate-x.x.x/src/net/sf/hibernate目录中，或hibernate.jar文件中找到。Hibernate总是会在它的classpath中首先搜索DTD文件。

### 5.1.2. hibernate-mapping

这个元素包括三个可选的属性。schema属性，指明了这个映射所引用的表所在的schema名称。假若指定了这个属性，表名会加上所指定的schema的名字扩展为全限定名。假若没有指定，表名就不会使用全限定名。default-cascade指定了未明确注明cascade属性的Java属性和集合类Java会采取什么样的默认级联风格。auto-import属性默认让我们在查询语言中可以使用非全限定名的类名。

```
<hibernate-mapping
    schema="schemaName"
    default-cascade="none|save-update"
    auto-import="true|false"
    package="package.name"
/>
```



schema (可选): 数据库schema名称。

②

`default-cascade` (可选 - 默认为 `none`): 默认的级联风格。

③

`auto-import` (可选 - 默认为 `true`): 指定是否我们可以在查询语言中使用非全限定的类名 (仅限于本映射文件中的类)。

④

`package` (可选): 指定一个包前缀, 如果在映射文档中没有指定全限定名, 就使用这个包名。

假若你有两个持久化类, 它们的非全限定名是一样的 (就是在不同的包里面--译者注), 你应该设置 `auto-import="false"`。假若说你把一个“import 过”的名字同时对应两个类, Hibernate 会抛出一个异常。

### 5.1.3. class

你可以使用 `class` 元素来定义一个持久化类:

```
<class
  name="ClassName"
  table="tableName"
  discriminator-value="discriminator_value"
  mutable="true|false"
  schema="owner"
  proxy="ProxyInterface"
  dynamic-update="true|false"
  dynamic-insert="true|false"
  select-before-update="true|false"
  polymorphism="implicit|explicit"
  where="arbitrary sql where condition"
  persister="PersisterClass"
  batch-size="N"
  optimistic-lock="none|version|dirty|all"
  lazy="true|false"
/>
```

❶

name: 持久化类（或者接口）的Java全限定名。

❷

table: 对应的数据库表名。

❸

discriminator-value (辨别值) (可选 - 默认和类名一样) : 一个用于区分不同的子类的值，在多态行为时使用。

❹

mutable (可变) (可选, 默认值为 true): 表明该类的实例可变（不可变）。

❺

schema (可选): 覆盖在根<hibernate-mapping>元素中指定的schema名字。

❻

proxy (可选): 指定一个接口，在延迟装载时作为代理使用。你可以在这里使用该类自己的名字。

❼

dynamic-update (动态更新) (可选，默认为false): 指定用于UPDATE 的SQL将会在运行时动态生成，并且只更新那些改变过的字段。

❽

dynamic-insert (动态插入) (可选, 默认为false): 指定用于INSERT的 SQL将会在运行时动态生成，并且只包含那些非空值字段。

❾

select-before-update (可选，默认值为false): 指定Hibernate除非确定对象的确被修改了，不会执行SQL UPDATE操作。在特定场合（实际上，只会发生在一个临时对象关联到一个新的session中去，执行update()的时候），这说明Hibernate会在UPDATE之前执行一次额外的SQL SELECT操作，来决定是否应该进行UPDATE。

- ⑩ polymorphism (多形, 多态) (可选, 默认值为 implicit (隐式)): 界定是隐式还是显式的使用查询多态。
  
- (11) where (可选) 指定一个附加的SQLWHERE 条件, 在抓取这个类的对象时会一直增加这个条件。
  
- (12) persister (可选): 指定一个定制ClassPersister。
  
- (13) batch-size (可选, 默认是1) 指定一个用于根据标识符抓取实例时使用的"batch size" (批次抓取数量)。
  
- (14) optimistic-lock (乐观锁定) (可选, 默认是version): 决定乐观锁定的策略。
  
- (15) lazy (延迟) (可选): 假若设置 lazy="true", 就是设置这个类自己的名字作为proxy接口的一种等价快捷形式。

若指明的持久化类实际上是一个接口, 也可以被完美地接受。其后你可以用<subclass>来指定该接口的实际实现类名。你可以持久化任何static (静态的) 内部类。记得应该使用标准的类名格式, 就是说比如: Foo\$Bar。

不可变类, mutable="false"不可以被应用程序更新或者删除。这可以让Hibernate做一些小小的性能优化。

可选的proxy属性可以允许延迟加载类的持久化实例。Hibernate开始会返回实现了这个命名接口的CGLIB代理。当代理的某个方法被实际调用的时候, 真实的持久化对象才会被装载。参见下面的“用于延迟装载的代理”。

Implicit (隐式)的多态是指, 如果查询中给出的是任何超类、该类实现的接口或者该类的名字, 都会返回这个类的实例; 如果查询中给出的是子类的名字, 则会返回子类的实例。Explicit (显式)的多态是指, 只有在查询

中给出的明确是该类的名字时才会返回这个类的实例；同时只有当在这个<class>的定义中作为<subclass>或者<joined-subclass>出现的子类，才会可能返回。大多数情况下，默认的polymorphism="implicit"都是合适的。显式的多态在有两个不同的类映射到同一个表的时候很有用。（允许一个“轻型”的类，只包含部分表字段）。

persist属性可以让你定制这个类使用的持久化策略。你可以指定你自己实现的net.sf.hibernate.persister.EntityPersister的子类，你甚至可以完全从头开始编写一个net.sf.hibernate.persister.ClassPersister接口的实现，可能是用储存过程调用、序列化到文件或者LDAP数据库来实现的。参阅net.sf.hibernate.test.CustomPersister，这是一个简单的例子（“持久化”到一个Hashtable）。

请注意dynamic-update和dynamic-insert的设置并不会继承到子类，所以在<subclass>或者<joined-subclass>元素中可能需要再次设置。这些设置是否能够提高效率要视情形而定。请用你的智慧决定是否使用。

使用select-before-update通常会降低性能.当是在防止数据库不必要的触发update触发器，这就很有用了。

如果你打开了dynamic-update，你可以选择几种乐观锁定的策略：

- version（版本检查）检查version/timestamp字段
- all（全部）检查全部字段
- dirty（脏检查）只检查修改过的字段
- none（不检查）不使用乐观锁定

我们非常强烈建议你在Hibernate中使用version/timestamp字段来进行乐观锁定。对性能来说，这是最好的选择，并且这也是唯一能够处理在session外进行操作的策略（就是说，当使用Session.update()的时候）。

#### 5.1.4. id

被映射的类必须声明对应数据库表主键字段。大多数类有一个JavaBeans风格的属性，为每一个实例包含唯一的标识。<id>元素定义了该属性到数据库表主键字段的映射。

```

<id
    name="propertyName"           ❶
    type="typename"               ❷
    column="column_name"          ❸
    unsaved-value="any|none|null|id_value" ❹
    access="field|property|ClassName"> ❺

    <generator class="generatorClass"/>
</id>

```

- ❶ name (可选): 标识属性的名字。
- ❷ type (可选): 标识Hibernate类型的名字。
- ❸ column (可选 - 默认为属性名): 主键字段的名称。
- ❹ unsaved-value (可选 - 默认为null): 一个特定的标识属性值，用来标志该实例是刚刚创建的，尚未保存。这可以把这种实例和从以前的session中装载过（可能又做过修改--译者注）但未再次持久化的实例区分开来。
- ❺ access (可选 - 默认为property): Hibernate用来访问属性值的策略。

如果 name 属性不存在，会认为这个类没有标识属性。

unsaved-value 属性很重要！如果你的类的标识属性不是默认为null的，你应该指定正确的默认值。

还有一个另外的<composite-id>声明可以访问旧式的多主键数据。我们强烈不鼓励使用这种方式。

### 5.1.4.1. generator

必须声明的<generator>子元素是一个Java类的名字，用来为该持久化类的实例生成唯一的标识。如果这个生成器实例需要某些配置值或者初始化参数，用<param>元素来传递。

```
<id name="id" type="long" column="uid" unsaved-value="
    <generator class="net.sf.hibernate.id.TableHiL
        <param name="table">uid_table</param>
        <param name="column">next_hi_value_col
    </generator>
</id>
```

所有的生成器都实现net.sf.hibernate.id.IdentifierGenerator接口。这是一个非常简单的接口；某些应用程序可以选择提供他们自己特定的实现。当然，Hibernate提供了很多内置的实现。下面是一些内置生成器的快捷名字：

#### increment（递增）

用于为long, short或者int类型生成唯一标识。只有在没有其他进程往同一张表中插入数据时才能使用。在集群下不要使用。

#### identity

对DB2,MySQL, MS SQL Server, Sybase和HypersonicSQL的内置标识字段提供支持。返回的标识符是long, short 或者int类型的。

#### sequence（序列）

在DB2,PostgreSQL, Oracle, SAP DB, McKoi中使用序列（sequence），而在Interbase中使用生成器(generator)。返回的标识符是long, short 或者int类型的。

#### hilo（高低位）

使用一个高/低位算法来高效的生成long, short 或者 int类型的标识符。给定一个表和字段（默认分别是hibernate\_unique\_key 和next）作为

高位值得来源。高/低位算法生成的标识符只在一个特定的数据库中是唯一的。在使用JTA获得的连接或者用户自行提供的连接中，不要使用这种生成器。

**seqhilo** (使用序列的高低位)

使用一个高/低位算法来高效的生成long, short或者int类型的标识符，给定一个数据库序列(sequence)的名字。

**uuid.hex**

用一个128-bit的UUID算法生成字符串类型的标识符。在一个网络中唯一(使用了IP地址)。UUID被编码为一个32位16进制数字的字符串。

**uuid.string**

使用同样的UUID算法。UUID被编码为一个16个字符长的任意ASCII字符组成的字符串。不能使用在PostgreSQL数据库中

**native** (本地)

根据底层数据库的能力选择identity, sequence 或者hilo中的一个。

**assigned** (程序设置)

让应用程序在save()之前为对象分配一个标示符。

**foreign** (外部引用)

使用另外一个相关联的对象的标识符。和<one-to-one>联合一起使用。

#### 5.1.4.2. 高/低位算法 (Hi/Lo Algorithm)

hilo 和 seqhilo生成器给出了两种hi/lo算法的实现，这是一种很令人满意的标识符生成算法。第一种实现需要一个“特殊”的数据库表来保存下一个可用的“hi”值。第二种实现使用一个Oracle风格的序列(在被支持的情况下)。

```
<id name="id" type="long" column="cat_id">
```

```
<generator class="hilo">
    <param name="table">hi_value</param>
    <param name="column">next_value</param>
    <param name="max_lo">100</param>
</generator>
</id>
```

```
<id name="id" type="long" column="cat_id">
    <generator class="seqhilo">
        <param name="sequence">hi_value</param>
        <param name="max_lo">100</param>
    </generator>
</id>
```

很不幸，你在为Hibernate自行提供connection，或者Hibernate使用JTA获取应用服务器的数据源连接的时候无法使用hilo。Hibernate必须能够在一个新的事务中得到一个"hi"值。在EJB环境中实现hi/lo算法的标准方法是使用一个无状态的session bean。

### 5.1.4.3. UUID算法 ( UUID Algorithm )

UUID包含：IP地址，JVM的启动时间（精确到1/4秒），系统时间和一个计数器值（在JVM中唯一）。在Java代码中不可能获得MAC地址或者内存地址，所以这已经是我们在不使用JNI的前提下的能做的最好实现了。

不要试图在PostgreSQL中使用uuid.string。

### 5.1.4.4. 标识字段和序列 ( Identity Columns and Sequences )

对于内部支持标识字段的数据库(DB2,MySQL,Sybase,MS SQL)，你可以使用identity关键字生成。对于内部支持序列的数据库 (DB2,Oracle, PostgreSQL, Interbase, McKoi,SAP DB),你可以使用sequence风格的关键字生成。这两种方式对于插入一个新的对象都需要两次SQL查询。

```
<id name="id" type="long" column="uid">
    <generator class="sequence">
```

```
        <param name="sequence">uid_sequence</param>
    </generator>
</id>
```

```
<id name="id" type="long" column="uid" unsaved-value="0">
    <generator class="identity"/>
</id>
```

对于跨平台开发，`native`策略会从`identity`、`sequence`和`hilo`中进行选择，取决于底层数据库的支持能力。

#### 5.1.4.5. 程序分配的标识符 (Assigned Identifiers)

如果你需要应用程序分配一个标识符（而非Hibernate来生成它们），你可以使用`assigned`生成器。这种特殊的生成器会使用已经分配给对象的标识符属性的标识符值。用这种特性来分配商业行为的关键字要特别小心（基本上总是一种可怕的设计决定）。

#### 5.1.5. composite-id 联合ID

```
<composite-id
    name="propertyName"
    class="ClassName"
    unsaved-value="any|none">

    <key-property name="propertyName" type="typeName">
    <key-many-to-one name="propertyName" class="ClassName">
    . . . . .
</composite-id>
```

如果表使用联合主键，你可以把类的多个属性组合成为标识符属性。`<composite-id>`元素接受`<key-property>`属性映射和`<key-many-to-one>`属性映射作为子元素。

```
<composite-id>
    <key-property name="medicareNumber"/>
```

```
<key-property name="dependent"/>
</composite-id>
```

你的持久化类必须重载`equals()`和`hashCode()`方法，来实现组合的标识符判断等价.也必须实现`Serializable`接口。

不幸的是，这种组合关键字的方法意味着一个持久化类是它自己的标识。除了对象自己之外，没有什么方便的“把手”可用。你必须自己初始化持久化类的实例，在使用组合关键字`load()`持久化状态之前，必须填充他的联合属性。我们会在[Section 7.4, “作为联合标识符\(As Composite Identifiers\)”](#)章中说明一种更加方便的方法，把联合标识实现为一个独立的类，下面描述的属性只对这种备用方法有效:

- `name` (可选) : 一个组件类型，持有联合标识 (参见下一节)。
- `class` (可选 - 默认为通过反射(reflection)得到的属性类型) : 作为联合标识的组件类名(参见下一节)。
- `unsaved-value` (可选 - 默认为 `none`): 假如被设置为非`none`的值，就表示新创建，尚未被持久化的实例将持有的值。

### 5.1.6. 识别器 ( discriminator )

在“一棵对象继承树对应一个表”的策略中,<discriminator>元素是必需的,它声明了表的识别器字段。识别器字段包含标志值，用于告知持久化层应该为某个特定的行创建哪一个子类的实例。只能使用如下受到限制的一些类型： `string`, `character`, `integer`, `byte`, `short`, `boolean`, `yes_no`, `true_false`.

```
<discriminator
    column="discriminator_column" ❶
    type="discriminator_type"      ❷
    force="true|false"             ❸
/>
```

❶

`column` (可选 - 默认为 `class`) 识别器字段的名称

❷

type (可选 - 默认为 string) 一个Hibernate字段类型的名字

③

force(强制) (可选 - 默认为 false) "强制"Hibernate指定允许的认可器值, 就算取得的所有实例都是根类的。

标识器字段的实际值是根据<class> 和<subclass>元素的discriminator-value 得来的。

force属性仅仅是在表包含一些未指定应该映射到哪个持久化类的时候才是有用的。这种情况不是经常会遇到。

### 5.1.7. 版本 ( version ) (可选)

<version>元素是可选的, 表明表中包含附带版本信息的数据。这在你准备使用长事务 ( long transactions ) 的时候特别有用。(见后)

```
<version
  column="version_column"      ❶
  name="propertyName"        ❷
  type="typename"             ❸
  access="field|property|ClassName"
  unsaved-value="null|negative|undefined"
/>
```

❶

column (可选 - 默认为属性名): 指定持有版本号的字段名。

❷

name: 持久化类的属性名。

❸

type (可选 - 默认是 integer): 版本号类型。

❹

access (可选 - 默认是 property): Hibernate用于访问属性值的策略。

⑤

unsaved-value (可选 - 默认是undefined): 用于标明某个实例时刚刚被实例化的 (尚未保存) 版本属性值, 依靠这个值就可以把这种情况和已经在先前的session中保存或装载的实例区分开来。 (undefined指明使用标识属性值进行这种判断。)

版本号必须是以下类型: long, integer, short, timestamp或者calendar。

### 5.1.8. 时间戳 ( timestamp ) (可选)

可选的<timestamp>元素指明了表中包含时间戳数据。这用来作为版本的替代。时间戳本质上是一种对乐观锁定的一种不是特别安全的实现。当然, 有时候应用程序可能在其他方面使用时间戳。

```
<timestamp
    column="timestamp_column"           ❶
    name="propertyName"                 ❷
    access="field|property|ClassName"    ❸
    unsaved-value="null|undefined"      ❹
/>
```

❶

column (可选 - 默认为属性名): 持有时间戳的字段名。

❷

name: 在持久化类中的JavaBeans风格的属性名, 其Java类型是 Date 或者 Timestamp的。

❸

access (可选 - 默认是 property): Hibernate用于访问属性值的策略。

❹

unsaved-value (可选 - 默认是null): 用于标明某个实例时刚刚被实例化的 (尚未保存) 版本属性值, 依靠这个值就可以把这种情况和已经在先前的session中保存或装载的实例区分开来。 (undefined指明使用标识属性值进行这种判断。)

注意, `<timestamp>` 和 `<version type="timestamp">` 是等价的。

### 5.1.9. property

`<property>` 元素为类声明了一个持久化的,JavaBean风格的属性。

```
<property
    name="propertyName"           ❶
    column="column_name"         ❷
    type="typename"              ❸
    update="true|false"          ❹
    insert="true|false"          ❺
    formula="arbitrary SQL expression" ❻
    access="field|property|ClassName"
/>
```

❶

name: 属性的名字,以小写字母开头。

❷

column (可选 - 默认为属性名字): 对应的数据库字段名。

❸

type (可选): 一个Hibernate类型的名字。

❹

update, insert (可选 - 默认为 true): 表明在用于UPDATE 和/或 INSERT的SQL语句中是否包含这个字段。这二者如果都设置为false则表明这是一个“外源性 (derived)”的属性,它的值来源于映射到同一个 (或多个)字段的某些其他属性,或者通过一个trigger(触发器),或者其他程序。

❺

formula (可选): 一个SQL表达式,定义了这个计算 (computed) 属性的值。计算属性没有和它对应的数据库字段。

⑥

`access` (可选 - 默认值为 `property`): Hibernate用来访问属性值的策略。

`typename`可以是如下几种：

- Hibernate基础类型之一（比如：`integer`, `string`, `character`, `date`, `timestamp`, `float`, `binary`, `serializable`, `object`, `blob`）。
- 一个Java类的名字，这个类属于一种默认基础类型（比如：`int`, `float`, `char`, `java.lang.String`, `java.util.Date`, `java.lang.Integer`, `java.sql.Clob`）。
- 一个`PersistentEnum`的子类的名字。（比如：`.eg.Color`）。
- 一个可以序列化的Java类的名字。
- 一个自定义类型的类的名字。（比如：`com.illflow.type.MyCustomType`）。

如果你没有指定类型，Hibernate会使用反射来得到这个名字的属性，以此来猜测正确的Hibernate类型。Hibernate会对属性读取器(getter方法)的返回类进行解释，按照规则2,3,4的顺序。然而，这并不足够。在某些情况下你仍然需要`type`属性。（比如，为了区别`Hibernate.DATE`和`Hibernate.TIMESTAMP`,或者为了指定一个自定义类型。）

`access`属性用来让你控制Hibernate如何在运行时访问属性。在默认情况下，Hibernate会使用属性的`get/set`方法对。如果你指明`access="field"`,Hibernate会忽略`get/set`方法对，直接使用反射来访问成员变量。你也可以指定你自己的策略，这就需要你自己实现`net.sf.hibernate.property.PropertyAccessor`接口，再在`access`中设置你自定义策略类的名字。

### 5.1.10. 多对一 (many-to-one)

通过`many-to-one`元素,可以定义一种常见的与另一个持久化类的关联。这种关系模型是多对一关联。（实际上是一个对象引用。）

```
<many-to-one
    name="propertyName"           ❶
    column="column_name"         ❷
    class="ClassName"            ❸
    cascade="all|none|save-update|delete" ❹
    outer-join="true|false|auto" ❺
    update="true|false"          ❻
    insert="true|false"          ❻
    property-ref="propertyNameFromAssociatedClass"
    access="field|property|ClassName"
/>
```

❶

name: 属性名。

❷

column (可选): 字段名。

❸

class (可选 - 默认是通过反射得到属性类型): 关联的类的名字。

❹

cascade (级联) (可选): 指明哪些操作会从父对象级联到关联的对象。

❺

outer-join (外连接) (可选 - 默认为自动): 当设置hibernate.use\_outer\_join的时候，对这个关联允许外连接抓取。

❻

update, insert (可选 - defaults to true) 指定对应的字段是否在用于UPDATE 和/或 INSERT的SQL语句中包含。如果二者都是false,则这是一个纯粹的“外源性 (derived)”关联，它的值是通过映射到同一个 (或多个) 字段的某些其他属性得到的，或者通过trigger(除法器)，或者是其他程序。

❼

property-ref: (可选) 指定关联类的一个属性，这个属性将会和本外键

相对应。如果没有指定，会使用对方关联类的主键。

⑧

access (可选 - 默认是 property): Hibernate用来访问属性的策略。

cascade 属性允许下列值：all, save-update, delete, none。设置除了none以外的其它值会传播特定的操作到关联的（子）对象中。参见后面的“Lifecycle Objects(自动管理生命周期的对象)”。

outer-join参数允许下列三个不同值：

- auto (默认) 使用外连接抓取关联（对象），如果被关联的对象没有代理(proxy)
- true 一直使用外连接来抓取关联
- false 永远不使用外连接来抓取关联

一个典型的简单many-to-one声明例子：

```
<many-to-one name="product" class="Product" column="PR
```

property-ref属性只应该用来对付老旧的数据库系统，可能出现外键指向对方关联表的是个非主键字段（但是应该是一个惟一关键字）的情况。这是一种十分丑陋的关系模型。比如说，假设Product类有一个惟一的序列号，它并不是主键。（unique属性控制Hibernate通过SchemaExport工具生成DDL的过程。）

```
<property name="serialNumber" unique="true" type="stri
```

那么关于OrderItem 的映射可能是：

```
<many-to-one name="product" property-ref="serialNumber
```

当然，我们决不鼓励这种用法。

### 5.1.11. 一对一

持久化对象之间一对一的关联关系是通过`one-to-one`元素定义的。

```
<one-to-one
    name="propertyName"           ❶
    class="ClassName"             ❷
    cascade="all|none|save-update|delete"  ❸
    constrained="true|false"      ❹
    outer-join="true|false|auto"  ❺
    property-ref="propertyNameFromAssociatedClass"
    access="field|property|ClassName"
/>
```

❶

name: 属性的名字。

❷

class (可选 - 默认是通过反射得到的属性类型) : 被关联的类的名字。

❸

cascade(级联) (可选) 表明操作是否从父对象级联到被关联的对象。

❹

constrained(约束) (可选) 表明该类对应的表对应的数据库表, 和被关联的对象所对应的数据库表之间, 通过一个外键引用对主键进行约束。这个选项影响`save()`和`delete()`在级联执行时的先后顺序(也在`schema export tool`中被使用)。

❺

outer-join(外连接) (可选 - 默认为自动): 当设置`hibernate.use_outer_join`的时候, 对这个关联允许外连接抓取。

❻

property-ref: (可选) 指定关联类的一个属性, 这个属性将会和本外键相对应。如果没有指定, 会使用对方关联类的主键。

❼

access (可选 - 默认是 property): Hibernate用来访问属性的策略。

有两种不同的一对一关联：

- 主键关联
- 惟一外键关联

主键关联不需要额外的表字段;两行是通过这种一对一关系相关联的，那么这两行就共享同样的主关键字值。所以如果你希望两个对象通过主键一对一关联，你必须确认它们被赋予同样的标识值！

比如说，对下面的Employee和Person进行主键一对一关联:

```
<one-to-one name="person" class="Person"/>
```

```
<one-to-one name="employee" class="Employee" constrained="true"/>
```

Now we must ensure that the primary keys of related rows in the PERSON and EMPLOYEE tables are equal. We use a special Hibernate identifier generation strategy called foreign: 现在我们必须确保PERSON和EMPLOYEE中相关的字段是相等的。我们使用一个特别的称为foreign的Hibernate标识符生成器策略：

```
<class name="person" table="PERSON">
  <id name="id" column="PERSON_ID">
    <generator class="foreign">
      <param name="property">employee</param>
    </generator>
  </id>
  ...
  <one-to-one name="employee"
    class="Employee"
    constrained="true"/>
</class>
```

一个刚刚保存的Person实例被赋予和该Person的employee属性所指向的

Employee实例同样的关键字值。

另一种方式是一个外键和一个惟一关键字对应，上面的Employee和Person的例子，如果使这种关联方式，应该表达成：

```
<many-to-one name="person" class="Person" column="PERS
```

如果在Person的映射加入下面几句，这种关联就是双向的：

```
<one-to-one name="employee" class="Employee" property-r
```

### 5.1.12. 组件 ( component ) , 动态组件 ( dynamic-component )

<component>元素把子对象的一些元素与父类对应的表的一些字段映射起来。然后组件可以声明它们自己的属性、组件或者集合。参见后面的“Components”一章。

```
<component
  name="propertyName"           ❶
  class="className"             ❷
  insert="true|false"          ❸
  upate="true|false"           ❹
  access="field|property|ClassName"> ❺

  <property ...../>
  <many-to-one .... />
  .....
</component>
```

- ❶ name: 属性名
- ❷ class (可选 - 默认为通过反射得到的属性类型):组件 (子) 类的名字。
- ❸ insert: 被映射的字段是否出现在SQL的INSERT语句中?

④

update: 被映射的字段是否出现在SQL的UPDATE语句中?

⑤

access (可选 - 默认是 property): Hibernate用来访问属性的策略。

其<property>子标签为子类的一些属性和表字段建立映射。

<component>元素允许加入一个<parent>子元素，在组件类内部就可以有一个指向其容器的实体的反向引用。

<dynamic-component>元素允许把一个Map映射为组件，其属性名对应map的键值。

### 5.1.13. 子类(subclass)

最后，多态持久化需要为父类的每个子类都进行声明。对于我们建议的“每一棵类继承树对应一个表”的策略来说，就需要使用<subclass>声明。

```
<subclass
    name="ClassName"
    discriminator-value="discriminator_value"
    proxy="ProxyInterface"
    lazy="true|false"
    dynamic-update="true|false"
    dynamic-insert="true|false">

    <property .... />
    .....
</subclass>
```

①

name: 子类的全限定名。

②

discriminator-value(辨别标志) (可选 - 默认为类名): 一个用于区分每个独立的子类的值。

③

proxy(代理) (可选): 指定一个类或者接口, 在延迟装载时作为代理使用。

④

lazy(延迟装载) (可选): 设置lazy="true"是把自己的名字作为proxy接口的一种等价快捷方式。

每个子类都应该声明它自己的持久化属性和子类。 <version> 和<id> 属性可以从根父类继承下来。在一棵继承树上的每个子类都必须声明一个唯一的discriminator-value。如果没有指定, 就会使用Java类的全限定名。

#### 5.1.14. 连接的子类 (joined-subclass)

另外一种情况, 如果子类是持久化到一个属于它自己的表 (每一个子类对应一个表的映射策略), 那么就需要使用<joined-subclass>元素。

```
<joined-subclass
  name="ClassName"                ❶
  proxy="ProxyInterface"          ❷
  lazy="true|false"               ❸
  dynamic-update="true|false"
  dynamic-insert="true|false">

  <key .... >

  <property .... />
  .....
</subclass>
```

❶

name: 子类的全限定名。

② proxy (可选): 指定一个类或者接口, 在延迟装载时作为代理使用。

③

lazy (延迟装载) (可选): 设置lazy="true"是把自己的名字作为proxy接口的一种等价快捷方式。

这种映射策略不需要指定辨别标志 ( discriminator ) 字段。但是, 每一个都必须使用<key>元素指定一个表字段包含对象的标识符。本章开始的映射可以被用如下方式重写:

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD//EN"
    "http://hibernate.sourceforge.net/hibernate-ma
<hibernate-mapping package="eg">

    <class name="Cat" table="CATS">
        <id name="id" column="uid" type="long"
            <generator class="hilo"/>
        </id>
        <property name="birthdate" type="date"
        <property name="color" not-null="true"
        <property name="sex" not-null="true"/>
        <property name="weight"/>
        <many-to-one name="mate"/>
        <set name="kittens">
            <key column="MOTHER"/>
            <one-to-many class="Cat"/>
        </set>
        <joined-subclass name="DomesticCat" ta
            <key column="CAT"/>
            <property name="name" type="st
        </joined-subclass>
    </class>

    <class name="eg.Dog">
```

```
        <!-- mapping for Dog could go here -->
    </class>
</hibernate-mapping>
```

### 5.1.15. map, set, list, bag

集合类在后面讨论。

### 5.1.16. 引用 ( import )

假设你的应用程序有两个同样名字的持久化类，但是你不想在Hibernate查询中使用他们的全限定名。除了依赖`auto-import="true"`以外，类也可以被显式地“import(引用)”。你甚至可以引用没有明确被映射的类和接口。

```
<import class="java.lang.Object" rename="Universe"/>
```

```
<import
    class="ClassName"
    rename="ShortName"
/>
```

- ❶ `class`: 任何Java类的全限定名。
- ❷ `rename` (可选 - 默认为类的全限定名): 在查询语句中可以使用的名字。

## 5.2. Hibernate 的类型

### 5.2.1. 实体 (Entities) 和值 (values)

为了理解很多与持久化服务相关的Java语言级对象的行为，我们需要把它们分为两类：

实体`entity` 独立于任何持有实体引用的对象。与通常的Java模型相比，不再被引用的对象会被当作垃圾收集掉。实体必须被显式的保存和删除（除非保存和删除是从父实体向子实体引发的级联）。这和ODMG模型中关于对象通过可触及保持持久性有一些不同——比较起来更加接近应用程序对象通常在一个大系统中的使用方法。实体支持循环引用和交叉引用，它们也可以加上版本信息。

实体的持久化状态包含有指向其他实体的连接和一些值类型的实例。值是原始类型、集合、组件或者特定的不可变对象。与实体不同，值（特别是集合和组件）是通过可触及性来进行持久化和删除的。因为值对象（和原始类型数据）是随着包含它们的实体而被持久化和删除的，它们不能够被独立的加上版本信息。值没有独立的标识，所以它们不能被两个实体或者集合共享。

所有的Hibernate对象，除了集合，都支持null语义。

直到现在，我们都一直使用"持久化对象"来代表实体。我们仍然会这么做。然而严格的来说，并不是所有用户定义的，带有持久化状态的类都是实体。组件(`component`)就是一个用户定义的类，仅仅由值语义构成。

### 5.2.2. 基本值类型

基本类型可以大致的分为：

`integer, long, short, float, double, character, byte, boolean, yes_no, true_false`

这些类型都对应Java的原始类型或者其包装类，来适合（特定厂商的）SQL 字段类型。`boolean, yes_no` 和 `true_false`都是Java中`boolean`

或者`java.lang.Boolean`的另外说法。

#### **string**

从`java.lang.String` 到 `VARCHAR` (或者 Oracle的 `VARCHAR2`)的映射。

#### **date, time, timestamp**

从`java.util.Date`和其子类到SQL类型`DATE`, `TIME` 和`TIMESTAMP` (或等价类型)的映射。

#### **calendar, calendar\_date**

从`java.util.Calendar` 到SQL 类型`TIMESTAMP`和 `DATE` ( 或等价类型 ) 的映射。

#### **big\_decimal**

从`java.math.BigDecimal` 到 `NUMERIC` (或者 Oracle 的`NUMBER`类型)的映射。

#### **locale, timezone, currency**

从`java.util.Locale`, `java.util.TimeZone` 和`java.util.Currency` 到`VARCHAR` (或者 Oracle 的`VARCHAR2`类型)的映射. `Locale`和 `Currency` 的实例被映射为它们的ISO代码。 `TimeZone`的实例被映射为它的ID。

#### **class**

从`java.lang.Class` 到 `VARCHAR` (或者 Oracle 的`VARCHAR2`类型)的映射。 `Class`被映射为它的全限定名。

#### **binary**

把字节数组 ( `byte arrays`)映射为对应的 SQL二进制类型。

#### **text**

把长Java字符串映射为SQL的`CLOB`或者`TEXT`类型。

#### **serializable**

把可序列化的Java类型映射到对应的SQL二进制类型。你也可以为一个并非默认为基本类型或者实现`PersistentEnum`接口的可序列化Java类

或者接口指定Hibernate类型serializable。

### **clob, blob**

JDBC 类 `java.sql.Clob` 和 `java.sql.Blob`的映射。某些程序可能不适合使用这个类型，因为blob和clob对象可能在一个事务之外是无法重用的。（而且，驱动程序对这种类型的支持充满着补丁和前后矛盾。）

实体及其集合的唯一标识可以是任何基础类型，除了binary、 blob 和 clob 之外。（联合标识也是允许的，后面会说到。）

在`net.sf.hibernate.Hibernate`中，定义了基础类型对应的Type常量。比如，`Hibernate.STRING`代表string 类型。

### **5.2.3. 持久化枚举（ Persistent enum ）类型**

枚举（*enumerated*）类型是一种常见的Java习惯用语，它是一个类，拥有一些（不多）的不可变实例。你可以为枚举类型实现`net.sf.hibernate.PersistentEnum`接口，定义`toInt()`和`fromInt()`方法：

```
package eg;
import net.sf.hibernate.PersistentEnum;

public class Color implements PersistentEnum {
    private final int code;
    private Color(int code) {
        this.code = code;
    }
    public static final Color TABBY = new Color(0);
    public static final Color GINGER = new Color(1);
    public static final Color BLACK = new Color(2);

    public int toInt() { return code; }

    public static Color fromInt(int code) {
        switch (code) {
            case 0: return TABBY;

```

```

        case 1: return GINGER;
        case 2: return BLACK;
        default: throw new RuntimeException("Unknc
    }
}
}

```

Hibernate可以使用枚举类的名字作为类型名，这个例子中就是eg.Color。

## 5.2.4. 自定义值类型

开发者创建属于他们自己的值类型也是很容易的。比如说，你可能希望持久化java.lang.BigInteger类型的属性，持久化成为VARCHAR字段。Hibernate没有内置这样一种类型。自定义类型能够映射一个属性（或集合元素）到不止一个数据库表字段。比如说，你可能有这样的Java属性：getName()/setName()，这是java.lang.String类型的，对应的持久化到三个字段：FIRST\_NAME, INITIAL, SURNAME。

要实现一个自定义类型，可以实现net.sf.hibernate.UserType或net.sf.hibernate.CompositeUserType中的任一个，并且使用类型的Java全限定类名来声明属性。请查看net.sf.hibernate.test.DoubleStringType这个例子，看看它是怎么做。

```

<property name="twoStrings" type="net.sf.hibernate.test
    <column name="first_string"/>
    <column name="second_string"/>
</property>

```

注意使用<column>标签来把一个属性映射到多个字段的作法。

虽然Hibernate内置的丰富类型和对component的支持意味着你可能很少需要使用自定义类型，至少对于你程序中经常出现的自定义类（并非实体）来说，这是一种好方法。比如说，MonetaryAmount(价格总额)对比使用CompositeUserType来说更好,虽然它可以很容易的使用一个component实现。这样做的动机之一是抽象。通过自定义类型，以后假若你改变表示金额值的方法时，你的映射文件不需要更改，这就得到了保护。

## 5.2.5. 映射到"任意"(any)类型

这是属性映射的又一种类型。<any>映射元素定义了一种从多个表到类的多形联合。这种类型的映射总是需要多于一个字段。第一个字段持有被从属的实体的类型。其他的字段持有标识符。对于这种类型的联合来说，不可能指定一个外键约束，所以当然这不是（多形）联合映射的通常方式。你只应该在非常特殊的情况下使用它（比如，审计log,用户会话数据等等）。

```
<any name="anyEntity" id-type="long" meta-type="eg.cus"
  <column name="table_name"/>
  <column name="id"/>
</any>
```

meta-type属性让应用程序指定一个自定义类型，把数据库字段值映射到一个持久化类，该类的标识属性是用id-type定义的。如果meta-type返回java.lang.Class的实例，不需要其他处理。另一方面，如果是类似string或者character这样的基本类型，你必须指定从值到类的映射。

```
<any name="anyEntity" id-type="long" meta-type="string"
  <meta-value value="TBL_ANIMAL" class="Animal"/>
  <meta-value value="TBL_HUMAN" class="Human"/>
  <meta-value value="TBL_ALIEN" class="Alien"/>
  <column name="table_name"/>
  <column name="id"/>
</any>
```

```
<any
  name="propertyName"           ❶
  id-type="idtypename"         ❷
  meta-type="metatypename"     ❸
  cascade="none|all|save-update" ❹
  access="field|property|ClassName" ❺
>
  <meta-value ... />
  <meta-value ... />
  .....
```

```
        <column .... />
        <column .... />
        .....
</any>
```

- ❶ name: 属性名。
- ❷ id-type: 标识符类型。
- ❸ meta-type (可选 - 默认为class): 一个用于把java.lang.Class映射到一个数据库字段的类或者允许分辨映射的类型。
- ❹ cascade(级联) (可选- 默认为 none): 级联风格。
- ❺ access (可选 - 默认是 property): Hibernate用来访问属性的策略。

老式的object 类型是用来在Hibernate 1.2中起到类似作用的，他仍然被支持，但是已经基本废弃了。

## 5.3. SQL中引号包围的标识符

你可强制Hibernate在生成的SQL中把标识符用引号前后包围起来，这需要在映射文档中使用反向引号(`)把表名或者字段名包围（可能比较拗口，请看下面的例子）。Hibernate会使用相应的SQLDialect（方言）来使用正确的引号风格（通常是双引号，但是在SQL Server中是括号，MySQL中是反向引号）。

```
<class name="LineItem" table="`Line Item`">
  <id name="id" column="`Item Id`"/><generator class=
  <property name="itemNumber" column="`Item #`"/>
  ...
</class>
```

## 5.4. 自定义DDL

Hibernate映射文档也包含一些只为了SchemaExport命令行工具生成DDL使用的信息。比如，你可以使用<column>元素的sql-type属性覆盖字段类型。

```
<property
  name="amount"
  type="big_decimal">
  <column
    name="AMOUNT"
    sql-type="NUMERIC(11, 2)"/>
</property>
```

或者,你可以指定字段长度和约束。下面是等价的：

```
<property
  name="socialSecurityNumber"
  type="string"
  length="9"
  column="SSN"
  not-null="true"
  unique="true"/>

<property
  name="socialSecurityNumber"
  type="string">
  <column
    name="SSN"
    length="9"
    not-null="true"
    unique="true"/>
</property>
```

最后,也可以通过SchemaExport命令行工具自动生成所需的索引。如果要这么做,请在<column>元素上使用index属性。

```
<property
  name="lastname">
  <column
    name="LASTNAME"
    index="by_last_first"/>
</property>

<property
  name="firstname">
  <column
    name="FIRSTNAME"
    index="by_last_first"/>
</property>
```

关于SchemaExport的更多内容，请参见本文档的“工具箱”那一章。

## 5.5. 映射文件的模块化 ( Modular mapping files )

允许在独立的映射文档中定义 `subclass` 和 `joined-subclass`，直接位于 `hibernate-mapping` 下。这就可以让你每次扩展你的类层次的时候，加入新的映射文件就行了。在子类的映射中你必须指定一个 `extends` 属性，指明先前已经映射过的超类。使用这个功能的时候，一定要注意映射文件的排序是非常重要的！

```
<hibernate-mapping>
  <subclass name="eg.subclass.DomesticCat" extends="eg.subclass.DomesticCat" >
    <property name="name" type="string"/>
  </subclass>
</hibernate-mapping>
```

## Chapter 6. 集合类(Collections)

## 6.1. 持久化集合类(Persistent Collections)

(译者注：在阅读本章的时候，以后整个手册的阅读过程中，我们都会面临一个名词方面的问题，那就是“集合”。“Collections”和“Set”在中文里对应都被翻译为“集合”，但是他们的含义很不一样。Collections是一个超集，Set是其中的一种。大部分情况下，本译稿中泛指而未加英文注明的“集合”，都应当理解为“Collections”。在有些二者同时出现，可能造成混淆的地方，我们用“集合类”来特指“Collections”，“集合(Set)”来指“Set”，一般都会在后面的括号中给出英文。希望大家在阅读时联系上下文理解，不要造成误解。与此同时，“元素”一词对应的英文“element”，也有两个不同的含义。其一为集合的元素，是内存中的一个变量；另一含义则是XML文档中的一个标签所代表的元素。也请注意区别。本章中，特别是后半部分是需要反复阅读才能理解清楚的。如果遇到任何疑问，请记住，英文版本的reference是惟一标准的参考资料。)

这部分不包含大量的Java代码例子。我们假定你已经了解如何使用Java自身的集合类框架(Java's collections framework)。其实如果是这样，这里就真的没有什么东西需要学习了... 用一句话来做个总结，你就用你已经掌握的知识来使用它们吧，不用为了适应Hibernate而作出改变。

Hibernate可以持久化以下java集合的实例，包括java.util.Map, java.util.Set, java.util.SortedMap, java.util.SortedSet, java.util.List, 和任何持久实体或值的数组。类型为java.util.Collection或者java.util.List的属性还可以使用"bag"语义来持久。

警告：用于持久化的集合，除了集合接口外，不能保留任何实现这些接口的类所附加的语义(例如:LinkedHashSet带来的迭代顺序)。所有的持久化集合，实际上都各自按照HashMap, HashSet, TreeMap, TreeSet 和 ArrayList 的语义直接工作。更深入地说，对于一个包含集合的属性来说，必须把Java类型定义为接口（也就是Map, Set 或者List等），而绝不能是HashMap, TreeSet 或者 ArrayList。存在这个限制的原因是，在你不知道的时候，Hibernate暗中把你的Map, Set 和 List 的实例替换成了它自己的关于Map, Set 或者 List 的实现。（所以在你的程序中，谨慎使用==操作符。）(译者说明：为了提高性能等方面的原因，在Hibernate中实现了几乎所有的Java集合的接口。)

```
Cat cat = new DomesticCat();
Cat kitten = new DomesticCat();
```

```
.....
Set kittens = new HashSet();
kittens.add(kitten);
cat.setKittens(kittens);
session.save(cat);
kittens = cat.getKittens(); //Okay, kittens collection
(HashSet) cat.getKittens(); //Error!
```

集合遵从对值类型的通常规则：不能共享引用, 与其包含的实体共存亡。由于存在底层的关联模型，集合不支持空值语义；并且hibernate不会区分一个null的集合引用和一个不存在元素的空集合。

集合实例在数据库中根据指向对应实体的外键而得到区别。这个外键被称为集合的关键字。在Hibernate配置文件中使用<key> 元素来映射这个集合的关键字。

集合可以包含几乎所有的Hibernate类型, 包括所有的基本类型, 自定义类型, 实体类型和组件。集合不能包含其他集合。这些被包含的元素的类型被称为集合元素类型。集合的元素在Hibernate中被映射为<element>, <composite-element>, <one-to-many>, <many-to-many> 或者 <many-to-any>。

除了Set和Bag之外的所有集合类型都有一个索引(*index*)字段, 这个字段映射到一个数组或者List的索引或者Map的key。Map的索引的类型可以是任何基本类型, 实体类型或者甚至是一个组合类型(但不能是一个集合类型)。数组和list的索引肯定是整型, integer。在Hibernate配置文件中使用 <index>, <index-many-to-many>, <composite-index> 或者 <index-many-to-any>等元素来映射索引。

集合类可以产生相当多种类的映射，涵盖了很多通常的关系模型。我们建议你练习使用schema生成工具, 以便对如何把不同的映射定义转换为数据库表有一个感性认识。

## 6.2. 映射集合 ( Mapping a Collection )

在Hibernate配置文件中使用<set>, <list>, <map>, <bag>, <array> 和 <primitive-array>等元素来定义集合,而<map>是最典型的一个。

```
<map
  name="propertyName"
  table="table_name"
  schema="schema_name"
  lazy="true|false"
  inverse="true|false"
  cascade="all|none|save-update|delete|all-delete-or
  sort="unsorted|natural|comparatorClass"
  order-by="column_name asc|desc"
  where="arbitrary sql where condition"
  outer-join="true|false|auto"
  batch-size="N"
  access="field|property|ClassName"
>

  <key .... />
  <index .... />
  <element .... />
</map>
```

❶

name 集合属性的名称

❷

table ( 可选——默认为属性的名称 ) 这个集合表的名称(不能在一对多的关联关系中使用)

❸

schema (可选) 表的schema的名称, 他将覆盖在根元素中定义的schema

❹

lazy (可选——默认为false) lazy(可选--默认为false) 允许延迟加载  
( lazy initialization ) (不能在数组中使用)

⑤

inverse (可选——默认为false) 标记这个集合作为双向关联关系中的  
方向一端。

⑥

cascade (可选——默认为none) 让操作级联到子实体

⑦

sort(可选)指定集合的排序顺序, 其可以为自然的(natural)或者给定一个  
用来比较的类。

⑧

order-by (可选, 仅用于jdk1.4) 指定表的字段(一个或几个)再加上asc或  
者desc(可选), 定义Map,Set和Bag的迭代顺序

⑨

where (可选) 指定任意的SQL where条件, 该条件将在重新载入或者删  
除这个集合时使用(当集合中的数据仅仅是所有可用数据的一个子集  
时这个条件非常有用)

⑩

outer-join(可选)指定这个集合,只要可能,应该通过外连接(outer join)  
取得。在每一个SQL语句中,只能有一个集合可以被通过外连接抓取  
(译者注: 这里提到的SQL语句是取得集合所属类的数据的Select语句)

(11)

batch-size (可选, 默认为1) 指定通过延迟加载取得集合实例的批处理  
块大小 ( "batch size" ) 。

(12)

access(可选-默认为属性property):Hibernate取得属性值时使用的策略

建立列表(List)和数组(Array)需要一个单独表字段用来保存列表(List)或数组

(Array)的索引(`foo[i]`中的*i*)。如果你的关系模型中没有索引字段,例如:如果你处理的是老式的遗留数据,你可以用无序的Set来替代。这会让那些以为List应该是访问无序集合的更方便的人感到气馁。

Hibernate集合严格遵守Set,List和Map接口中包涵的自然语义。List元素不能正确的自发对他们自己进行排序!

在另一方面,那些准备使用List来模拟*bag*的语义的人有一个合法的委屈(a legitimate grievance)。bag是一个无序,没有索引的集合并且可能包含多个相同的元素。在Java集合框架中没有Bag接口(虽然你可以用List模拟它)。

Hibernate允许你映射类型为List或者Collection的属性到<bag>元素。注意:Bag语义事实上并不是Collection规范(contract)的一部分并且事实上它和List规范中的语义是相矛盾的。

具有`inverse="false"`标记的大型Hibernate bag效率是相当低的,应该尽量避免。Hibernate无法创建,删除和更新它的单个记录,因为他们没有关键字来识别单个记录。

## 6.3. 值集合和多对多关联(Collections of Values and Many To Many Associations)

任何值集合和实体集合如果被映射为多对多关联(Java集合中的语义)就需要一个集合表。这个表中包含外键字段,元素字段还可能有索引字段。

使用<key>元素来申明从集合表到其拥有者类的表(from the collection table to the table of the owning class)的外键关键字。

```
<key column="column_name"/>
```

❶

column(必需):外键字段的名称

对于类似与map和list的带索引的集合,我们需要一个<index>元素。对于list来说,这个字段包含从零开始的连续整数。对于map来说,这个字段可以包含任意Hibernate类型的值。

```
<index  
    column="column_name"           ❶  
    type="typename"               ❷  
>
```

❶

column(必需):保存集合索引值的字段名。

❷

type (可选,默认为整型integer):集合索引的类型。

还有另外一个选择,map可以是实体类型的对象。在这里我们使用<index-many-to-many>元素。

```
<index-many-to-many  
    column="column_name"           ❶
```

```
class="ClassName" ❷  
/>
```

❶

column(必需):集合索引值中外键字段的名称

❷

class (required):(必需):集合的索引使用的实体类。

对于一个值集合, 我们使用<element>标签。

```
<element  
    column="column_name" ❶  
    type="typename" ❷  
/>
```

❶

column(必需):保存集合元素值的字段名。

❷

type (必需):集合元素的类型

一个拥有自己表的实体集合对应于多对多(*many-to-many*)关联关系概念。多对多关联是针对Java集合的最自然映射关联关系, 但通常并不是最好的关系模型。

```
<many-to-many  
    column="column_name"  
    class="ClassName"  
    outer-join="true|false|auto"  
/>
```

❶

column(必需): 这个元素的外键关键字段名

❷

class (必需): 关联类的名称

③

outer-join (可选 - 默认为auto): 在Hibernate系统参数中hibernate.use\_outer\_join被打开的情况下,该参数用来允许使用outer join来载入此集合的数据。

例子 :

首先, 一组字符串 :

```
<set name="names" table="NAMES">
  <key column="GROUPID"/>
  <element column="NAME" type="string"/>
</set>
```

包含一组整数的bag(还设置了order-by参数指定了迭代的顺序) :

```
<bag name="sizes" table="SIZES" order-by="SIZE ASC">
  <key column="OWNER"/>
  <element column="SIZE" type="integer"/>
</bag>
```

一个实体数组,在这个案例中是一个多对多的关联(注意这里的实体是自动管理生命周期的对象 ( lifecycle objects ) ,cascade="all"):

```
<array name="foos" table="BAR_FOOS" cascade="all">
  <key column="BAR_ID"/>
  <index column="I"/>
  <many-to-many column="FOO_ID" class="com.illflow.F
</array>
```

一个map,通过字符串的索引来指明日期 :

```
<map name="holidays" table="holidays" schema="dbo" or
  <key column="id"/>
  <index column="hol_name" type="string"/>
```

```
<element column="hol_date" type="date"/>
</map>
```

一个组件的列表：

```
<list name="carComponents" table="car_components">
  <key column="car_id"/>
  <index column="posn"/>
  <composite-element class="com.illflow.CarComponent"
    <property name="price" type="float"/>
    <property name="type" type="com.illflow.Cc
    <property name="serialNumber" column="seri
  </composite-element>
</list>
```

## 6.4. 一对多关联 ( One To Many Associations )

一对多关联直接连接两个类对应的表,而没有中间集合表。(这实现了一个一对多的关系模型)(译者注:这有别与多对多的关联需要一张中间表)。这个关系模型失去了一些Java集合的语义:

- map,set或list中不能包含null值
- 一个被包含的实体的实例只能被包含在一个集合的实例中
- 一个被包含的实体的实例只能对应于集合索引的一个值中

一个从Foo到Bar的关联需要额外的关键字字段,可能还有一个索引字段指向这个被包含的实体类,Bar所对应的表。这些字段在映射时使用前面提到的<key>和<index>元素。

<one-to-many>标记指明了一个一对多的关联。

```
<one-to-many class="ClassName"/>
```

### ❶

class(必须):被关联类的名称。

### 例子

```
<set name="bars">
  <key column="foo_id"/>
  <one-to-many class="com.illflow.Bar"/>
</set>
```

注意:<one-to-many>元素不需要定义任何字段。也不需要指定表名。

重要提示:如果一对多关联中的<key>字段定义成NOT NULL,那么当创建和更新关联关系时Hibernate可能引起约束违例。为了预防这个问题,你必须使用双向关联,并且在“多”这一端( Set或者是bag)指明inverse="true"。

## 6.5. 延迟初始化(延迟加载) ( Lazy Initialization )

(译者注: 本翻译稿中, 对Lazy Initiazation和Eager fetch中的lazy,eager采取意译的方式, 分别翻译为延迟初始化和预先抓取。lazt initiazation就是指直到第一次调用时才加载。)

集合(不包括数组)是可以延迟初始化的,意思是仅仅当应用程序需要访问时,才载入他们的值。对于使用者来说,初始化是透明的,因此应用程序通常不需要关心这个(事实上,透明的延迟加载也就是为什么Hibernate需要自己的集合实现的主要原因)。但是,如何应用程序试图执行以下程序:

```
s = sessions.openSession();
User u = (User) s.find("from User u where u.name=?", u);
Map permissions = u.getPermissions();
s.connection().commit();
s.close();

Integer accessLevel = (Integer) permissions.get("accou
```

这个错误可能令你感到意外。因为在这个Session被提交(commit)之前,permissions没有被初始化,那么这个集合将永远不能载入他的数据了。解决方法是把读取集合数据的语句提到Session被提交之前。

另外一种选择是不使用延迟初始化集合。既然延迟初始化可能引起上面这样错误,默认是不使用延迟初始化的。但是,为了效率的原因,我们对绝大多数集合(特别是实体集合)使用延迟初始化。

延迟初始化集合时发生的例外被封装在LazyInitializationException中。

使用可选的 lazy 属性来定义延迟初始化集合:

```
<set name="names" table="NAMES" lazy="true">
  <key column="group_id"/>
  <element column="NAME" type="string"/>
</set>
```

在一些应用程序的体系结构中,特别是使用hibernate访问数据的结构,代码可

能会用在不用的应用层中,可能没有办法保证当一个集合在初始化的时候,session仍然打开着。这里有两个基本方法来解决这个问题:

- 在基于Web的应用程序中,一个servlet过滤器可以用来在用户请求的完成之前来关闭session。当然,这个地方(关闭session)严重依赖于你的应用程序结构中例外处理的正确性。在请求返回给用户之前关闭session和结束事务是非常重要的,即使是在构建视图(译者注:返回给用户的HTML页面)的时候发生了例外,也必须确保这一点。考虑到这一点,servlet过滤器可以保证能够操作这个session。我们推荐使用一个ThreadLocal变量来保存当前的session。
- 在一个有单独的商业层的应用程序中,商业逻辑必须在返回之前“准备好”Web层所需要的所有集合。通常,应用程序为每个Web层需要的集合调用hibernate.initialize()(必须在session被关闭之前调用)或者通过使用FETCH子句来明确获取到整个集合。

你可以使用Hibernate Session API中的filter()方法来在初始化之前得到集合的大小:

```
( (Integer) s.filter( collection, "select count(*)" ).
```

filter() 或者 createFilter()同样被用于有效的重新载入一个集合的子集而不需要载入整个集合。

## 6.6. 集合排序 ( Sorted Collections )

Hibernate支持实现`java.util.SortedMap`和`java.util.SortedSet`的集合。你必须在映射文件中指定一个比较器：

```
<set name="aliases" table="person_aliases" sort="natural"
  <key column="person"/>
  <element column="name" type="string"/>
</set>

<map name="holidays" sort="my.custom.HolidayComparator"
  <key column="year_id"/>
  <index column="hol_name" type="string"/>
  <element column="hol_date" type="date"/>
</map>
```

`sort`属性中允许的值包括`unsorted`,`natural`和某个实现了`java.util.Comparator`的类的名称。

分类集合的行为事实上象`java.util.TreeSet`或者`java.util.TreeMap`。

## 6.7. 对collection排序的其他方法 ( Other Ways To Sort a Collection )

如果你希望数据库自己对集合元素排序，可以利用set,bag或者map映射中的order-by属性。这个解决方案只能在jdk1.4或者更高的jdk版本中才可以实现(通过LinkedHashSet或者 LinkedHashMap实现)。它是在SQL查询中完成排序，而不是在内存中。

```
<set name="aliases" table="person_aliases" order-by="r
  <key column="person"/>
  <element column="name" type="string"/>
</set>

<map name="holidays" order-by="hol_date, hol_name" laz
  <key column="year_id"/>
  <index column="hol_name" type="string"/>
  <element column="hol_date type="date"/>
</map>
```

注意: 这个order-by属性的值是一个SQL排序子句而不是HQL的！

关联还可以在运行时使用filter()根据任意的条件来排序。

```
sortedUsers = s.filter( group.getUsers(), "order by th
```

## 6.8. 垃圾收集 ( Garbage Collection )

集合是在被持久对象引用时自动持久化，并且不再引用时自动删除的。如果集合被从一个持久化对象转移到另外一个，他的数据可能会被从一个表移到另外一个。你不需要担心这些。就跟你通常使用java集合一样使用Hibernate集合即可。

注意:上面的论断在inverse="true"的情况下不适用。我们将在接下来的章节中解释这一点。

## 6.9. 双向关联 ( Bidirectional Associations )

双向关联允许通过关联的任一端访问另外一端。在Hibernate中, 支持两种类型的双向关联:

### 一对多 ( **one-to-many** )

Set或者bag值在一端, 单独值(非集合)在另外一端

### 多对多 ( **many-to-many** )

两端都是set或bag值

请注意Hibernate不支持带有索引的集合(list,map或者array)作为"多"的那一端的双向one-to-many关联。

要建立一个双向的多对多关联, 只需要映射两个many-to-many关联到同一个数据库表中, 并再定义其中的一端为*inverse*。这里有一个从一个类关联到他自身的many-to-many的双向关联的例子: (原文: You may specify a bidirectional many-to-many association simply by mapping two many-to-many associations to the same database table and declaring one end as *inverse*. Heres an example of a bidirectional many-to-many association from a class back to itself: )

```
<class name="eg.Node">
  <id name="id" column="id"/>
  ....
  <bag name="accessibleTo" table="node_access" lazy=
    <key column="to_node_id"/>
    <many-to-many class="eg.Node" column="from_node_id"/>
  </bag>
  <!-- inverse end -->
  <bag name="accessibleFrom" table="node_access" inverse=
    <key column="from_node_id"/>
    <many-to-many class="eg.Node" column="to_node_id"/>
  </bag>
</class>
```

```
</bag>
</class>
```

如果只对关联的反向端进行了改变，这个改变不会被持久化。（原文：Changes made only to the inverse end of the association are *not* persisted.）

要建立一个一对多的双向关联，你可以通过把一个一对多关联，作为一个多对一关联映射到同一张表的字段上，并且在“多”的那一端定义 `inverse="true"`。（原文：You may map a bidirectional one-to-many association by mapping a one-to-many association to the same table column(s) as a many-to-one association and declaring the many-valued end `inverse="true"`。）

```
<class name="eg.Parent">
  <id name="id" column="id"/>
  ....
  <set name="children" inverse="true" lazy="true">
    <key column="parent_id"/>
    <one-to-many class="eg.Child"/>
  </set>
</class>

<class name="eg.Child">
  <id name="id" column="id"/>
  ....
  <many-to-one name="parent" class="eg.Parent" column="parent_id"/>
</class>
```

在“一”这一端定义 `inverse="true"` 不会影响级联操作。（原文：Mapping one end of an association with `inverse="true"` doesn't affect the operation of cascades.）

## 6.10. 三重关联 ( Ternary Associations )

这里有两种可能的途径来映射一个三重关联。其中一个是使用组合元素(下面将讨论).另外一个是一个使用一个map, 并且带有关联作为其索引。

```
<map name="contracts" lazy="true">
  <key column="employer_id"/>
  <index-many-to-many column="employee_id" class="Emp
  <one-to-many column="contract_id" class="Contract"
</map>
```

```
<map name="connections" lazy="true">
  <key column="node1_id"/>
  <index-many-to-many column="node2_id" class="Node"
  <many-to-many column="connection_id" class="Connec
</map>
```

## 6.11. 异类关联(Heterogeneous Associations)

<many-to-any>和<index-many-to-any>元素提供真正的异类关联。这些元素和<any>元素工作方式是同样的,他们都应该很少用到。

## 6.12. 集合例子 ( Collection Example )

在前面的几个章节的确非常令人迷惑。因此让我们来看一个例子。这个类：

```
package eg;
import java.util.Set;

public class Parent {
    private long id;
    private Set children;

    public long getId() { return id; }
    private void setId(long id) { this.id=id; }

    private Set getChildren() { return children; }
    private void setChildren(Set children) { this.chil

    ....
    ....
}
```

这个类有一个`eg.Child`的实例集合。如果每一个子实例至多有一个父实例，那么最自然的映射是一个one-to-many的关联关系：

```
<hibernate-mapping>

    <class name="eg.Parent">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <set name="children" lazy="true">
            <key column="parent_id"/>
            <one-to-many class="eg.Child"/>
        </set>
    </class>
```

```

<class name="eg.Child">
  <id name="id">
    <generator class="sequence"/>
  </id>
  <property name="name"/>
</class>

```

```
</hibernate-mapping>
```

在以下的表定义中反应了这个映射关系：

```

create table parent ( id bigint not null primary key )
create table child ( id bigint not null primary key, r
alter table child add constraint childfk0 (parent_id)

```

如果父亲是必须的, 那么就可以使用双向one-to-many的关联了(请看后面父子关系的章节)。

```

<hibernate-mapping>

  <class name="eg.Parent">
    <id name="id">
      <generator class="sequence"/>
    </id>
    <set name="children" inverse="true" lazy="true"
      <key column="parent_id"/>
      <one-to-many class="eg.Child"/>
    </set>
  </class>

  <class name="eg.Child">
    <id name="id">
      <generator class="sequence"/>
    </id>
    <property name="name"/>
    <many-to-one name="parent" class="eg.Parent" c
  </class>

```

```
</hibernate-mapping>
```

请注意NOT NULL的约束:

```
create table parent ( id bigint not null primary key )
create table child ( id bigint not null
                    primary key,
                    name varchar(255),
                    parent_id bigint not null )
alter table child add constraint childfk0 (parent_id)
```

另外一方面,如果一个子实例可能有多个父实例,那么就应该使用many-to-many关联:

```
<hibernate-mapping>

  <class name="eg.Parent">
    <id name="id">
      <generator class="sequence"/>
    </id>
    <set name="children" lazy="true" table="childs">
      <key column="parent_id"/>
      <many-to-many class="eg.Child" column="chi
    </set>
  </class>

  <class name="eg.Child">
    <id name="id">
      <generator class="sequence"/>
    </id>
    <property name="name"/>
  </class>

</hibernate-mapping>
```

表定义:

```
create table parent ( id bigint not null primary key )
```

```
create table child ( id bigint not null primary key, r
create table childset ( parent_id bigint not null, chi
alter table childset add constraint childsetfk0 (paren
alter table childset add constraint childsetfk1 (chilo
```

## 6.13. <idbag>

如果你完全信奉我们对于“联合主键（composite keys）是个坏东西”，和“实体应该使用（无机的）自己生成的代用标识符（surrogate keys）”的观点，也许你会感到有一些奇怪，我们目前为止展示的多对多关联和值集合都是映射成为带有联合主键的表的！现在，这一点非常值得争辩；看上去一个单纯的关联表并不能从代用标识符中获得什么好处（虽然使用组合值的集合可能会获得一点好处）。不过，Hibernate提供了一个（一点点试验性质的）功能，让你把多对多关联和值集合应得到一个使用代用标识符的表去。

<idbag> 属性让你使用bag语义来映射一个List (或Collection)。

```
<idbag name="lovers" table="LOVERS" lazy="true">
  <collection-id column="ID" type="long">
    <generator class="hilo"/>
  </collection-id>
  <key column="PERSON1"/>
  <many-to-many column="PERSON2" class="eg.Person" c
</idbag>
```

你可以理解，<idbag>人工的id生成器，就好像是实体类一样！集合的每一行都有一个不同的人造关键字。但是，Hibernate没有提供任何机制来让你取得某个特定行的人造关键字。

注意<idbag>的更新性能要比普通的<bag>高得多！Hibernate可以有效的定位到不同的行，分别进行更新或删除工作，就如同处理一个list, map或者set一样。

在目前的实现中，还不支持使用identity标识符生成器策略。(In the current implementation, the identity identifier generation strategy is not supported.)

## Chapter 7. 组件 ( Components )

*Component*这个概念在Hibernate中几处不同的地方为了不同的目的被重复使用.

## 7.1. 作为依赖的对象 ( As Dependent Objects )

Component是一个被包含的对象, 它和它的所有者存储在同一张表中。也就是, 它是一个值类型, 而不是一个实体。Component术语和组成的面向对象概念相关(而并不是系统构架层次上的组件的概念)。举个例子, 你可以对一个人(Person)象以下这样来建模:

```
public class Person {
    private java.util.Date birthday;
    private Name name;
    private String key;
    public String getKey() {
        return key;
    }
    private void setKey(String key) {
        this.key=key;
    }
    public java.util.Date getBirthday() {
        return birthday;
    }
    public void setBirthday(java.util.Date birthday) {
        this.birthday = birthday;
    }
    public Name getName() {
        return name;
    }
    public void setName(Name name) {
        this.name = name;
    }
    .....
    .....
}
```

```
public class Name {
    char initial;
    String first;
    String last;
```

```

public String getFirst() {
    return first;
}
void setFirst(String first) {
    this.first = first;
}
public String getLast() {
    return last;
}
void setLast(String last) {
    this.last = last;
}
public char getInitial() {
    return initial;
}
void setInitial(char initial) {
    this.initial = initial;
}
}

```

现在,姓名(Name)是作为人(Person)的一个组成部分。需要注意的是:需要对姓名的持久化属性定义getter和setter方法,但是不需要实现任何的接口或声明标识符字段。

以下是这个例子的XML映射文件:

```

<class name="eg.Person" table="person">
  <id name="Key" column="pid" type="string">
    <generator class="uuid.hex"/>
  </id>
  <property name="birthday" type="date"/>
  <component name="Name" class="eg.Name"> <!-- class
    <property name="initial"/>
    <property name="first"/>
    <property name="last"/>
  </component>
</class>

```

人员(Person)表中将包括pid, birthday, initial, first和 last等字段。

就像所有的值类型一样, Component不支持共享引用。Component的值为空从语义学上来讲是专有的。每当重新加载一个包含组件的对象,如果component的所有字段为空,那么将Hibernate将假定整个component为空。对于绝大多数目的,这样假定是没有问题的。

Component的属性可以是Hibernate类型(包括Collections, many-to-one 关联, 及其它Component等等)。嵌套Component不应该作为特殊的应用被考虑(Nested components should not be considered an exotic usage)。Hibernate趋向于支持设计细致(fine-grained)的对象模型。

<component> 元素还允许有 <parent>子元素, 用来表明component类中的一个属性返回包含它的实体的引用。

```
<class name="eg.Person" table="person">
  <id name="Key" column="pid" type="string">
    <generator class="uuid.hex"/>
  </id>
  <property name="birthday" type="date"/>
  <component name="Name" class="eg.Name">
    <parent name="namedPerson"/> <!-- reference ba
    <property name="initial"/>
    <property name="first"/>
    <property name="last"/>
  </component>
</class>
```

## 7.2. In Collections

Hibernate支持component的集合(例如: 一个元素是“姓名”这种类型的数组)。你可以使用<composite-element>标签替代<element>标签来定义你的component集合。

```
<set name="someNames" table="some_names" lazy="true">
  <key column="id"/>
  <composite-element class="eg.Name"> <!-- class attribute -->
    <property name="initial"/>
    <property name="first"/>
    <property name="last"/>
  </composite-element>
</set>
```

注意, 如果你决定定义一个元素是联合元素的Set, 正确地实现equals()和hashCode()是非常重要的。

组合元素可以包含component但是不能包含集合。如果你的组合元素自身包含component, 必须使用<nested-composite-element>标签。这是一个相当特殊的案例 - 组合元素的集合自身可以包含component。这个时候你就应该考虑一下使用one-to-many关联是否会更恰当。尝试对这个组合元素重新建模为一个实体 - 但是需要注意的是, 虽然Java模型和重新建模前是一样的, 关系模型和持久性语义上仍然存在轻微的区别。

请注意如果你使用<set>标签, 一个组合元素的映射不支持可能为空的属性。当删除对象时, Hibernate必须使用每一个字段的来确定一条记录(在组合元素表中, 没有单个的关键字段), 如果有为null的字段, 这样做就不可能了。你必须作出一个选择, 要么在组合元素中使用不能为空的属性, 要么选择使用<list>, <map>, <bag> 或者 <idbag>而不是 <set>。

组合元素有个特别的案例, 是组合元素可以包含一个<many-to-one>元素。类似这样的映射允许你映射一个many-to-many关联表作为组合元素额外的字段。(A mapping like this allows you to map extra columns of a many-to-many association table to the composite element class.) 接下来的例子是从Order到Item的一个多对多的关联关系, 而 purchaseDate, price 和 quantity 是Item的关联属性。

```

<class name="eg.Order" .... >
  ....
  <set name="purchasedItems" table="purchase_items"
    <key column="order_id">
      <composite-element class="eg.Purchase">
        <property name="purchaseDate"/>
        <property name="price"/>
        <property name="quantity"/>
        <many-to-one name="item" class="eg.Item"/>
      </composite-element>
    </set>
  </class>

```

即使三重或多重管理都是可能的:

```

<class name="eg.Order" .... >
  ....
  <set name="purchasedItems" table="purchase_items"
    <key column="order_id">
      <composite-element class="eg.OrderLine">
        <many-to-one name="purchaseDetails" class="eg.Purchase"/>
        <many-to-one name="item" class="eg.Item"/>
      </composite-element>
    </set>
  </class>

```

在查询中，组合元素使用的语法是和关联到其他实体的语法一样的。

### 7.3. 作为一个Map的索引 ( As a Map Index )

`<composite-index>`元素允许你映射一个Component类作为Map的key，但是你必须确定你正确的在这个类中重写了`hashCode()` 和 `equals()`方法。

## 7.4. 作为联合标识符(As Composite Identifiers)

你可以使用一个component作为一个实体类的标识符。你的component类必须满足以下要求：

- 它必须实现java.io.Serializable接口
- 它必须重新实现equals()和hashCode()方法, 始终和组合关键字在数据库中的概念保持一致

你不能使用一个IdentifierGenerator产生组合关键字。作为替代应用程序必须分配它自己的标识符。

既然联合标识符必须在对象存储之前被分配，我们就不能使用unsaved-value 来把刚刚新建的实例和在先前的session保存的实例区分开来。

如果你希望使用saveOrUpdate()或者级联保存/更新(cascading save / update)，你应该实现 Interceptor.isUnsaved()。

使用<composite-id> 标签(它和<component> 标签有同样的属性和元素)代替<id>标签。下面有个联合标识符类的定义：

```
<class name="eg.Foo" table"F00S">
  <composite-id name="compId" class="eg.FooComposite"
    <key-property name="string"/>
    <key-property name="short"/>
    <key-property name="date" column="date_" type="date"/>
  </composite-id>
  <property name="name"/>
  . . . .
</class>
```

这时候,任何到F00S的外键也同样是联合的，在你其他类的映射文件中也必须同样定义。一个到Foo的定义应该像以下这样：

```
<many-to-one name="foo" class="eg.Foo">
<!-- the "class" attribute is optional, as usual -->
```

```
<column name="foo_string"/>
<column name="foo_short"/>
<column name="foo_date"/>
</many-to-one>
```

新的 `<column>` 标签同样被用于包含多个字段的自定义类型（This new column tag is also used by multi-column custom types）。事实上在各个地方它都是一个可选的字段属性。要定义一个元素是 `Foo` 的集合类，要这样写：

```
<set name="foos">
  <key column="owner_id"/>
  <many-to-many class="eg.Foo">
    <column name="foo_string"/>
    <column name="foo_short"/>
    <column name="foo_date"/>
  </many-to-many>
</set>
```

另一方面，`<one-to-many>` 元素通常不定义字段。

如果 `Foo` 自己包含集合，那么他们也需要使用联合外键。

```
<class name="eg.Foo">
  ....
  ....
  <set name="dates" lazy="true">
    <key>    <!-- a collection inherits the composition
              <column name="foo_string"/>
              <column name="foo_short"/>
              <column name="foo_date"/>
            </key>
    <element column="foo_date" type="date"/>
  </set>
</class>
```

## 7.5. 动态组件 ( Dynamic components )

你甚至可以映射`Map`类型的属性：

```
<dynamic-component name="userAttributes">
  <property name="foo" column="FOO"/>
  <property name="bar" column="BAR"/>
  <many-to-one name="baz" class="eg.Baz" column="BAZ"/>
</dynamic-component>
```

从`<dynamic-component>`映射的语义上来讲，它和`<component>`是相同的。这种映射类型的优点在于通过修改映射文件，就可以具有在部署时检测真实属性的能力。(利用一个DOM解析器，是有可能在运行时刻操作映射文件的。)

## Chapter 8. 操作持久化数据(Manipulating Persistent Data)

## 8.1. 创建一个持久化对象

对象（实体的实例）对一个特定的session来说，要么是一个瞬时（*transient*）对象，要么是持久化（*persistent*）对象。刚刚创建的对象当然是瞬时的（注：后文中transient object也称为临时对象）。session则提供了把瞬时实例保存（持久化）的服务：

```
DomesticCat fritz = new DomesticCat();
fritz.setColor(Color.GINGER);
fritz.setSex('M');
fritz.setName("Fritz");
Long generatedId = (Long) sess.save(fritz);
```

```
DomesticCat pk = new DomesticCat();
pk.setColor(Color.TABBY);
pk.setSex('F');
pk.setName("PK");
pk.setKittens( new HashSet() );
pk.addKitten(fritz);
sess.save( pk, new Long(1234) );
```

单参数的save()方法为fritz生成了一个唯一标识符，并赋给这个对象。双参数的形式则使用给定的标识符保存pk。我们一般不鼓励使用双参数的形式，因为这可能会（隐含）使主键赋予业务含义。它有用的时候是在一些特殊场合下，比如使用Hibernate来持久化一个BMP实体bean。

关联的对象可以用你喜欢的任何顺序持久化，除非有外键字段具有NOT NULL的约束。决不会有外键约束冲突的危险。然而，如果在save()对象的时候用错了顺序，会触犯NOT NULL约束。

## 8.2. 装载对象

如果你已知某个持久化实例的标识符，`Session`的`load()`方法让你取出它。第一种形式使用一个类对象作为参数，会把状态装载到另一个新创建的对象中去。第二个版本允许你给出一个实例，会在其中装载状态。把实例作为参数的形式在你准备把Hibernate和BMP实体bean一起使用的时候特别有用，它就是为此设计的。你也可以发现其他的用途（比如自己实现实例池等等）。

```
Cat fritz = (Cat) sess.load(Cat.class, generatedId);
```

```
// you need to wrap primitive identifiers
long pkId = 1234;
DomesticCat pk = (DomesticCat) sess.load( Cat.class, r
```

```
Cat cat = new DomesticCat();
// load pk's state into cat
sess.load( cat, new Long(pkId) );
Set kittens = cat.getKittens();
```

请注意如果没有匹配的数据库记录，`load()`方法可能抛出无法恢复的exception。如果类是通过代理映射的，`load()`方法返回一个对象，这是一个未初始化的代理，并且直到你调用该对象的某方法时才会去访问数据库。这种行为方式在你喜欢创建一个指向某对象的关联，又不想真的从数据库中装载它的时候特别有用。

如果你不确定是否有匹配的行存在，你应该使用`get()`方法，它会立刻访问数据库，如果没有对应的行，返回`null`。

```
Cat cat = (Cat) sess.get(Cat.class, id);
if (cat==null) {
    cat = new Cat();
    sess.save(cat, id);
}
return cat;
```

你可以用SQL `SELECT ... FOR UPDATE` 装载对象。下一节有关于Hibernate `LockMode` 的讨论。

```
Cat cat = (Cat) sess.get(Cat.class, id, LockMode.UPGRADE)
```

注意，任何关联的实例或者包含的集合都不会被做为 `FOR UPDATE` 返回。

任何时候都可以使用 `refresh()` 方法重新装载对象和它的集合。如果你使用数据库触发器更改了对象的某些属性，这就很有用。

```
sess.save(cat);  
sess.flush(); //force the SQL INSERT  
sess.refresh(cat); //re-read the state (after the trigger)
```

## 8.3. Querying

如果你不能确定你要寻找的对象的标示符，请使用Session的find()方法。Hibernate使用一种简单而强大的面向对象查询语言。

```
List cats = sess.find(
    "from Cat as cat where cat.birthdate = ?",
    date,
    Hibernate.DATE
);

List mates = sess.find(
    "select mate from Cat as cat join cat.mate as mate
    where cat.name = ?",
    name,
    Hibernate.STRING
);

List cats = sess.find( "from Cat as cat where cat.mate

List moreCats = sess.find(
    "from Cat as cat where " +
    "cat.name = 'Fritz' or cat.id = ? or cat.id = ?",
    new Object[] { id1, id2 },
    new Type[] { Hibernate.LONG, Hibernate.LONG }
);

List mates = sess.find(
    "from Cat as cat where cat.mate = ?",
    izi,
    Hibernate.entity(Cat.class)
);

List problems = sess.find(
    "from GoldFish as fish " +
    "where fish.birthday > fish.deceased or fish.birth
);
```

find()的第二个参数接受一个对象或者对象数组。第三个参数接受一个Hibernate类型或者类型的数组。这些指定的类型用来把给定的对象绑定到查询中的?占位符（实际上对应的是JDBC PreparedStatement的传入参数）。就像在JDBC中一眼，你应该优先使用这种参数绑定的方式，而非组装字符串。

Hibernate类定义了一些静态方法和常量，提供了访问大部分内置类型的手段。这些内置类型是net.sf.hibernate.type.Type的实例。

如果你知道你的查询会返回非常大量的对象，但是你不希望全部使用它们，你可以用iterate()方法获得更好的性能，它会返回一个java.util.Iterator。这个迭代器会在需要的时候装载对象，所使用的标识符来自一个前导的SQL查询。（一共是N+1次查询）

```
// fetch ids
Iterator iter = sess.iterate("from eg.Qux q order by c
while ( iter.hasNext() ) {
    Qux qux = (Qux) iter.next(); // fetch the object
    // something we couldnt express in the query
    if ( qux.calculateComplicatedAlgorithm() ) {
        // delete the current instance
        iter.remove();
        // dont need to process the rest
        break;
    }
}
```

很不幸，java.util.Iterator没有声明任何exception。所以，发生的任何SQL或者Hibernate的exception都会被包装在一个LazyInitializationException中（它是RuntimeException的子类）。

如果你预期大部分的对象已经装载过，存在于session的缓存中了，或者查询结果包含同样的对象很多次，那么iterator()方法也会获得更好的性能。（如果没有任何数据被缓存或者重复出现，则find()总是会更快。）下面是一个应该使用iterator()调用的查询例子：

```
Iterator iter = sess.iterate(
    "select customer, product " +
    "from Customer customer, " +
```

```
"Product product " +
"join customer.purchases purchase " +
"where product = purchase.product"
);
```

如果对上面的查询使用`find()`，会返回一个非常大的`JDBCResultSet`，包含很多重复的相同数据。

有时候Hibernate查询会每行返回多种对象，这种情况下，每行会返回一个数组，包含多个对象元素：

```
Iterator foosAndBars = sess.iterate(
    "select foo, bar from Foo foo, Bar bar " +
    "where bar.date = foo.date"
);
while ( foosAndBars.hasNext() ) {
    Object[] tuple = (Object[]) foosAndBars.next();
    Foo foo = tuple[0]; Bar bar = tuple[1];
    .....
}
```

### 8.3.1. 标量查询 ( Scalar query )

查询可以在`select`子句中指定类的属性。甚至可以调用SQL的统计函数。属性或者统计值被称为“标量(`scalar`)”结果。

```
Iterator results = sess.iterate(
    "select cat.color, min(cat.birthdate), count(c
    "group by cat.color"
);
while ( results.hasNext() ) {
    Object[] row = results.next();
    Color type = (Color) row[0];
    Date oldest = (Date) row[1];
    Integer count = (Integer) row[2];
    .....
}
```

```
Iterator iter = sess.iterate(  
    "select cat.type, cat.birthdate, cat.name from Dom  
);
```

```
List list = sess.find(  
    "select cat, cat.mate.name from DomesticCat cat"  
);
```

### 8.3.2. 查询接口 ( Query interface )

如果你需要为你的结果集设置边界 ( 你需要获取的最大行数与/或你希望获取的第一行 ) , 你应该得到一个`net.sf.hibernate.Query`的实例 :

```
Query q = sess.createQuery("from DomesticCat cat");  
q.setFirstResult(20);  
q.setMaxResults(10);  
List cats = q.list();
```

你甚至可以在映射文档中定义命名查询。( 记得用一个`CDATA`块把你的查询包含起来 , 否则在分析的时候可能引起误解。 )

```
<query name="eg.DomesticCat.by.name.and.minimum.weight"  
    from eg.DomesticCat as cat  
        where cat.name = ?  
        and cat.weight > ?  
] ]></query>
```

```
Query q = sess.getNamedQuery("eg.DomesticCat.by.name.a  
q.setString(0, name);  
q.setInt(1, minWeight);  
List cats = q.list();
```

查询界面支持使用命名参数。命名参数用`:name`的形式在查询字符串中表示。在`Query`中有方法把实际参数绑定到命名参数或者`JDBC`风格的`?`参数。和`JDBC`不同 , `Hibernate`的参数从`0`开始计数。使用命名参数有一些好处 :

- 命名参数不依赖于它们在查询字符串中出现的顺序
- 在同一个查询中可以使用多次
- 他们可读性好

```
//named parameter (preferred)
Query q = sess.createQuery("from DomesticCat cat where
q.setString("name", "Fritz");
Iterator cats = q.iterate();
```

```
//positional parameter
Query q = sess.createQuery("from DomesticCat cat where
q.setString(0, "Izi");
Iterator cats = q.iterate();
```

```
//named parameter list
List names = new ArrayList();
names.add("Izi");
names.add("Fritz");
Query q = sess.createQuery("from DomesticCat cat where
q.setParameterList("namesList", names);
List cats = q.list();
```

### 8.3.3. 可滚动迭代(Scrollable iteration)

如果你的JDBC驱动支持可滚动的`ResultSet,Query`接口可以获取一个`ScrollableResults` , 允许你在查询结果中灵活游走。

```
Query q = sess.createQuery("select cat.name, cat from
                        "order by cat.name");
ScrollableResults cats = q.scroll();
if ( cats.first() ) {

    // find the first name on each page of an alphabet
    firstNamesOfPages = new ArrayList();
    do {
```

```

        String name = cats.getString(0);
        firstNamesOfPages.add(name);
    }
    while ( cats.scroll(PAGE_SIZE) );

    // Now get the first page of cats
    pageOfCats = new ArrayList();
    cats.beforeFirst();
    int i=0;
    while( ( PAGE_SIZE > i++ ) && cats.next() ) pageOf
}

```

scroll()的行为方式与iterate()很类似，除了对象可以有选择的用get(int)初始化，而非整个行都一次性被初始化。

### 8.3.4. 过滤集合类(Filtering collections)

集合*filter*是一种特殊的查询，用于一个持久化集合或者数组。查询字符串可以引用this,意为当前的数组元素。

```

Collection blackKittens = session.filter(
    pk.getKittens(), "where this.color = ?", Color.BLA
);

```

返回的集合被认为是一个包(bag)。

请注意filter并不需要from子句（当然需要的话它们也可以加上）。Filter不限定返回它们自己的集合元素。

```

Collection blackKittenMates = session.filter(
    pk.getKittens(), "select this.mate where this.colc
);

```

### 8.3.5. 条件查询

HQL极为强大，但是有些人希望能够动态的使用一种面向对象API创建查

询，而非在他们的Java代码中嵌入字符串。对于那部分人来说，Hibernate提供了一种直观的Criteria查询API。

```
Criteria crit = session.createCriteria(Cat.class);
crit.add( Expression.eq("color", eg.Color.BLACK) );
crit.setMaxResults(10);
List cats = crit.list();
```

如果你对类似于SQL的语法不是感觉很舒服的话，用这种方法开始使用Hibernate可能更容易。这种API也比HQL更可扩展。程序可以提供它们自己的Criterion接口的实现。

### 8.3.6. 使用本地SQL的查询

你可以使用createSQLQuery()方法，用SQL来表达查询。你必须把SQL别名用大括号包围起来。

```
List cats = session.createSQLQuery(
    "SELECT {cat.*} FROM CAT AS {cat} WHERE ROWNUM<10"
    "cat",
    Cat.class
).list();
```

```
List cats = session.createSQLQuery(
    "SELECT {cat}.ID AS {cat.id}, {cat}.SEX AS {cat.se"
    "FROM CAT AS {cat} WHERE ROWNUM<10",
    "cat",
    Cat.class
).list();
```

和Hibernate查询一样，SQL查询也可以包含命名参数或者顺序参数。

## 8.4. 更改在当前session中保存或者装载的对象

持久化实例（就是通过session装载、保存、创建或者查询出的对象）可以被程序操作，所做的任何修改都会在Session清洗（*flushed*）的时候被持久化（参见后面的“flushing”部分）。所以最直接的更改一个对象的方法就是load()它，然后直接修改即可。

```
DomesticCat cat = (DomesticCat) sess.load( Cat.class,
cat.setName("PK");
sess.flush(); // changes to cat are automatically det
```

有些时候这种编程模式显得效率不高，因为它需要在同一个session中先使用SQL SELECT(来装载对象),又有一个SQL UPDATE（来把修改的状态写回）。因此，Hibernate提供了另一种方式。

## 8.5. 更改在以前session中保存或者装载的对象

很多程序需要在一个事务中获取对象，然后发送到界面层去操作，用一个新的事务来保存修改。（在高同步访问的环境中使用这种方式，经常使用附带版本的数据来保证事务独立性。）这种方法需要和上一节所描述的略微不同的编程模型。Hibernate支持这种模型，因为它提供了`Session.update()`方法。

```
// in the first session
Cat cat = (Cat) firstSession.load(Cat.class, catId);
Cat potentialMate = new Cat();
firstSession.save(potentialMate);

// in a higher tier of the application
cat.setMate(potentialMate);

// later, in a new session
secondSession.update(cat); // update cat
secondSession.update(mate); // update mate
```

如果拥有`catId`标识符的`Cat`在试图`update`它之前已经被`secondSession`装载了，会抛出一个异常。

对于给定的临时实例，当且仅当它们触及的其他临时实例需要保存的时候，应用程序应该对它们分别各自使用`update()`。（自动管理生命周期的对象(lifecycle object)除外。）

Hibernate用户曾经要求有一个通用的方法，可以为新建的临时实例生成标识符并保存，或者保存已经存在标识符的临时实例的改动。`saveOrUpdate()`方法就是用来提供这个功能的。

Hibernate通过对象的标识符的值（或`version`，或`timestamp`时间戳）来分辨这是一个“新”（未保存过的）实例，还是一个“已存在”（已经保存或者从先前的session中装载的）的实例。`id`映射中的`unsaved-value`（或`<version>`，或`<timestamp>`）用来指定哪个值被用于表示“新”实例。

```
<id name="id" type="long" column="uid" unsaved-value="
```

```
<generator class="hilo"/>
</id>
```

unsaved-value允许的取值包括：

- any - always save 永远保存
- none - always update 永远更新
- null - 当标识符是空的时候保存（默认情况）
- valid identifier value (合法的标识符值)- 当标识符是null或者这个给定的值时保存
- undefined - 对于version 或 timestamp来说的默认值。此时使用标识符检查。(原文: the default for version or timestamp, then identifier check is used.参见下文有进一步描述.)

```
// in the first session
Cat cat = (Cat) firstSession.load(Cat.class, catID);

// in a higher tier of the application
Cat mate = new Cat();
cat.setMate(mate);

// later, in a new session
secondSession.saveOrUpdate(cat); // update existing
secondSession.saveOrUpdate(mate); // save the new ins
```

saveOrUpdate()的用法和语义看来对初学者来说容易造成困惑。首先，如果你还没有试图在另一个新session中使用来自原session的实例，你根本就不需要使用update()或者saveOrUpdate()方法。有一些程序完全不需要使用这些方法。

通常，update()或saveOrUpdate()方法在下列情形下使用：

- 程序在前面的session中装载了对象
- 对象被传递到UI（界面）层

- 对该对象进行了一些修改
- 对象被传递回业务层
- 应用程序在第二个session中调用update()保存修改

saveOrUpdate()完成了如下工作：

- 如果对象已经在这个session中持久化过了，什么都不需要做
- 如果对象没有标识值，调用save()来保存它
- 如果对象的标识值与unsaved-value中的条件匹配，调用save()来保存它
- 如果对象使用了版本(version或timestamp),那么除非设置unsaved-value="undefined",版本检查会发生在标识符检查之前.
- 如果这个session中有另外一个对象具有同样的标识符，抛出一个异常

## 8.6. 把在先前的session中保存或装载的对象重新与新session建立关联(reassociate)

lock()方法是用来让应用程序把一个未修改的对象重新关联到新session的方法。

```
//just reassociate: 直接重新关联  
sess.lock(fritz, LockMode.NONE);  
//do a version check, then reassociate: 进行版本检查后关  
sess.lock(izi, LockMode.READ);  
//do a version check, using SELECT ... FOR UPDATE, the  
sess.lock(pk, LockMode.UPGRADE);
```

## 8.7. 删除持久化对象

使用`Session.delete()`会把对象的状态从数据库中移除。当然，你的应用程序可能仍然持有一个指向它的引用。所以，最好这样理解：`delete()`的用途是 把一个持久化实例变成临时实例。

```
sess.delete(cat);
```

你可以通过传递给`delete()`一个Hibernate 查询字符串来一次性删除很多对象。

你现在可以用你喜欢的任何顺序删除对象，不用担心外键约束冲突。当然，如果你搞错了顺序，还是有可能引发在外键字段定义的`NOT NULL`约束冲突。

## 8.8. 对象图 ( Graphs of objects )

要保存或者更新一个对象关联图中所有的所有对象，你必须做到：

- 保证每一个对象都执行`save()`、`saveOrUpdate()` 或 `update()`方法，或者，
- 在定义关联对象的映射时，使用`cascade="all"`或`cascade="save-update"`。

类似的，要删除一个关系图中的所有对象，必须：

- 对每一个对象都执行`delete()`，或者
- 在定义关联对象的映射时，使用`cascade="all"`、`cascade="all-delete-orphan"`或`cascade="delete"`。

建议：

- 如果子对象的生命期是绑定到父对象的生命期的，通过指定`cascade="all"`可以把它变成一个自动管理生命周期的对象 ( *lifecycle object* ) 。
- 否则，必须在应用程序代码中明确地执行`save()`和`delete()`。如果你想少敲一些代码，可以使用`cascade="save-update"`，然后只需明确地`delete()`。

### 8.8.1. 自动管理生命周期的对象 ( **lifecycle object** )

对一种关联 ( 多对一，或者集合 ) 使用`cascade="all"`映射，就把这种关联标记为一种父/子 ( *parent/child* ) 风格的关系，对父对象进行保存/更新/删除会导致对 ( 所有 ) 子对象的保存/更新/删除。但是这个比喻并不是特别确切。如果父对象解除了对某个子对象的关联，那这个子对象就不会被自动删除了。除非这是一个一对多的关联，并且标明了`cascade="all-delete-orphan"` ( 所有-删除-孤儿 ) 。级联操作的精确语义在下面列出：

- 如果父对象被保存，所有的子对象会被传递到`saveOrUpdate()`方法去执行

- 如果父对象被传递到update()或者saveOrUpdate()，所有的子对象会被传递到saveOrUpdate()方法去执行
- 如果一个临时的子对象被一个持久化的父对象引用了，它会被传递到saveOrUpdate()去执行
- 如果父对象被删除了，所有的子对象对被传递到delete()方法执行
- 如果临时的子对象不再被持久化的父对象引用，什么都不会发生（必要时，程序应该明确的删除这个子对象），除非声明了cascade="all-delete-orphan"，在这种情况下，成为“孤儿”的子对象会被删除。

### 8.8.2. 通过可触及性决定持久化（Persistence by Reachability）

Hibernate还没有完全实现“通过可触及性决定持久化”，后者暗示会对垃圾收集进行（效率不高的）持久化。但是，因为很广泛的呼声，Hibernate实现了一种意见，如果一个实体被一个持久化的对象引用，它也会被持久化。注明了cascade="save-update"的关联就是按照这种思路运作的。如果你希望在你的整个程序中都贯彻这个方法，你可以在<hibernate-mapping>元素的default-cascade属性中指定这种级联方式。

## 8.9. 清洗(Flushing) -- 这个词很难翻译，不能使用“刷新”，因为刷新一词已经被"refresh"使用了。有什么好的建议？

每件隔一段时间，`Session`会执行一些必需的SQL语句来把内存中的对象和JDBC连接中的状态进行同步。这个过程被称为清洗(*flush*)，默认会在下面的时间点执行：

- 在某些`find()`或者`iterate()`调用的时候
- 在`net.sf.hibernate.Transaction.commit()`的时候
- 在`Session.flush()`的时候

涉及的SQL语句会按照下面的顺序安排：

- 所有对实体进行插入的语句，其顺序按照对象执行`Session.save()`的时间顺序
- 所有对实体进行更新的语句
- 所有进行集合删除的语句
- 所有对集合元素进行删除，更新或者插入的语句
- 所有进行集合插入的语句
- 所有对实体进行删除的语句，其顺序按照对象执行`Session.delete()`的时间顺序

(有一个例外时，如果对象使用`native`方式进行ID生成的话，它们一执行`save`就会被插入。)

除非你明确地发出了`flush()`指令，关于`Session`合时会执行这些JDBC调用是完全无法保证的，只能保证它们执行的前后顺序。当然，Hibernate保证，`Session.find(..)`绝对不会返回已经失效的数据，也不会返回错误数据。

也可以改变默认的设置，来让清洗发生的不那么频繁。FlushMode类定义了三种不同的方式。大部分情况下，它们只由当你在处理“只读”的事务时才会使用，可能会得到一些（不是那么明显的）性能提高。

```
sess = sf.openSession();
Transaction tx = sess.beginTransaction();
sess.setFlushMode(FlushMode.COMMIT); //allow queries to
Cat izi = (Cat) sess.load(Cat.class, id);
izi.setName(iznizi);
// execute some queries....
sess.find("from Cat as cat left outer join cat.kittens
.....
tx.commit(); //flush occurs
```

## 8.10. 结束一个Session

结束一个session包括四个独立的步骤：

- 清洗session
- 提交事务
- 关闭session
- 处理异常

### 8.10.1. 清洗(Flush)session

如果你正在使用TransactionAPI，你就不用担心这个步骤。在事务提交的时候，隐含就会包括这一步。否则，你应该调用Session.flush()来确保你所有的修改都与数据库同步。

### 8.10.2. 提交事务

如果你正在使用Hibernate的Transaction API,代码类似这样：

```
tx.commit(); // flush the Session and commit the trans
```

如果你自行管理JDBC事务，你应该手工对JDBC连接执行commit()。

```
sess.flush();  
sess.connection().commit(); // not necessary for JTA
```

如果你决定不提交你的更改：

```
tx.rollback(); // rollback the transaction
```

或者:

```
// not necessary for JTA datasource, important otherwise  
sess.connection().rollback();
```

如果你回滚了事务,你应该立即关闭和取消当前session,确保Hibernate内部状态的完整性。

### 8.10.3. 关闭session

调用Session.close()就标志这个session进入了尾声。close()主要的含义就是与这个session相关的JDBC连接会被放弃。

```
tx.commit();  
sess.close();
```

```
sess.flush();  
sess.connection().commit(); // not necessary for JTA  
sess.close();
```

如果你自行管理连接,close()会返回连接的一个引用,你就可以手工把它关闭,或者返回它到连接池去。其他情况下,close()会把它返回到连接池去。

### 8.10.4. 处理异常

如果Session抛出了一个exception(包括任何SQLException),你应该立刻回滚这个事务,调用Session.close()来取消这个Session实例。Session中的一些特定方式会确保session不会处于一个不稳定不完整的状态。

建议采用下面的异常处理片断：

```
Session sess = factory.openSession();  
Transaction tx = null;  
try {  
    tx = sess.beginTransaction();  
    // do some work  
    ...  
}
```

```

        tx.commit();
    }
    catch (Exception e) {
        if (tx!=null) tx.rollback();
        throw e;
    }
    finally {
        sess.close();
    }
}

```

如果你是手工管理JDBC事务的，用下面这段：

```

Session sess = factory.openSession();
try {
    // do some work
    ...
    sess.flush();
    sess.connection().commit();
}
catch (Exception e) {
    sess.connection().rollback();
    throw e;
}
finally {
    sess.close();
}

```

如果你是从JTA中获得数据源的：

```

UserTransaction ut = .... ;
Session sess = factory.openSession();
try {
    // do some work
    ...
    sess.flush();
}
catch (Exception e) {
    ut.setRollbackOnly();
}

```

```
        throw e;
    }
    finally {
        sess.close();
    }
}
```

## 8.11. 拦截器(Interceptors)

Interceptor接口提供从session到你的应用程序的回调方法，让你的程序可以观察和在持久化对象保存/更改/删除或者装载的时候操作它的属性。一种可能的用途是用来监视统计信息。比如，下面的Interceptor会自动在一个Auditable创建的时候设置其createTimestamp,并且当它被更改的时候，设置其lastUpdateTimestamp属性。

```
package net.sf.hibernate.test;

import java.io.Serializable;
import java.util.Date;
import java.util.Iterator;

import net.sf.hibernate.Interceptor;
import net.sf.hibernate.type.Type;

public class AuditInterceptor implements Interceptor,

    private int updates;
    private int creates;

    public void onDelete(Object entity,
                        Serializable id,
                        Object[] state,
                        String[] propertyNames,
                        Type[] types) {
        // do nothing
    }

    public boolean onFlushDirty(Object entity,
                               Serializable id,
                               Object[] currentState,
                               Object[] previousState,
                               String[] propertyNames,
                               Type[] types) {
```

```

        if ( entity instanceof Auditable ) {
            updates++;
            for ( int i=0; i < propertyNames.length; i++ )
                if ( "lastUpdateTimestamp".equals( propertyNames[i] ) )
                    if ( currentTimestamp.equals( currentState[i] ) )
                        currentState[i] = new Date();
                    return true;
                }
            }
        }
    }
    return false;
}

public boolean onLoad(Object entity,
                    Serializable id,
                    Object[] state,
                    String[] propertyNames,
                    Type[] types) {
    return false;
}

public boolean onSave(Object entity,
                    Serializable id,
                    Object[] state,
                    String[] propertyNames,
                    Type[] types) {

    if ( entity instanceof Auditable ) {
        creates++;
        for ( int i=0; i<propertyNames.length; i++ )
            if ( "createTimestamp".equals( propertyNames[i] ) )
                if ( state[i] == null )
                    state[i] = new Date();
                return true;
            }
        }
    }
    return false;
}

public void postFlush(Iterator entities) {

```

```
        System.out.println("Creations: " + creates + "  
    }  
  
    public void preFlush(Iterator entities) {  
        updates=0;  
        creates=0;  
    }  
  
    .....  
    .....  
  
}
```

当session被创建的时候，就应该指定拦截器。

```
Session session = sf.openSession( new AuditInterceptor
```

## 8.12. 元数据(Metadata) API

Hibernate对所有的实体和值类型都需要一个非常丰富的元级别(meta-level)模型。有时候，这个模型对应用程序本身也会非常有用。比如说，应用程序可能使用Hibernate的元数据来实现一种“智能”的深度拷贝算法，来理解哪些对象应该被拷贝（比如，可变的值类型），那些不应该（不可变的值类型和可能的被关联的实体）。

Hibernate通过ClassMetadata接口，CollectionMetadata接口和Type对象树，暴露出元数据。可以通过SessionFactory获取metadata接口的实例。

```
Cat fritz = .....;
Long id = (Long) catMeta.getIdentifier(fritz);
ClassMetadata catMeta = sessionFactory.getClassMetadata(fritz);
Object[] propertyValues = catMeta.getPropertyValues(fritz);
String[] propertyNames = catMeta.getPropertyNames();
Type[] propertyTypes = catMeta.getPropertyTypes();
// get a Map of all properties which are not collections
// TODO: what about components?
Map namedValues = new HashMap();
for ( int i=0; i<propertyNames.length; i++ ) {
    if ( !propertyTypes[i].isEntityType() && !propertyTypes[i].isCollectionType() )
        namedValues.put( propertyNames[i], propertyValues[i] );
}
}
```

## Chapter 9. 父子关系(Parent Child Relationships)

刚刚接触Hibernate的人大多是从父子关系（parent / child type relationship）的建模入手的。父子关系的建模有两种方法。比较简便、直观的方法就是在实体类Parent和Child之间建立<one-to-many>的关联关系，从Parent指向Child，对新手来说尤其如此。但还有另一种方法，就是将Child声明为一个<composite-element>（组合元素）。可以看出在Hibernate中使用一对多关联比composite element更接近于通常parent / child关系的语义。下面我们会阐述如何使用双向可级联的一对多关联(*bidirectional one to many association with cascades*)去建立有效、优美的parent / child关系。这一点也不难！

## 9.1. 关于collections

在Hibernate下，实体类将collection作为自己的一个逻辑单元，而不是被容纳的多个实体。这非常重要！它主要体现为以下几点：

- 当删除或增加collection中对象的时候，拥有这个collection的实体对象的版本值会递增。
- 如果一个从collection中移除的对象是一个值类型(value type)的实例，比如composite element，那么这个对象的持久化状态将会终止，其在数据库中对应的记录会被删除。同样的，向collection增加一个value type的实例将会使之立即被持久化。
- 另一方面，如果从一对多或多对多关联的collection中移除一个实体，在缺省情况下这个对象并不会被删除。这个行为是完全合乎逻辑的 - 改变一个实体的内部状态不应该使与它关联的实体消失掉！同样的，向collection增加一个实体不会使之被持久化。

实际上，向Collection增加一个实体的缺省动作只是在两个实体之间创建一个连接而已，同样移除的时候也只是删除连接。这种处理对于所有的情况都是合适的。不适合所有情况的其实是父子关系本身，因为子对象是否存在依赖于父对象的生存周期。

## 9.2. 双向的一对多关系(Bidirectional one to many)

让我们从一个简单的例子开始，假设要实现一个从类Parent到类Child的一对多关系。

```
<set name="children">
  <key column="parent_id"/>
  <one-to-many class="Child"/>
</set>
```

如果我们运行下面的代码

```
Parent p = .....;
Child c = new Child();
p.getChildren().add(c);
session.save(c);
session.flush();
```

Hibernate就会产生下面的两条SQL语句:

- 一条INSERT语句，用于创建对象c对应的数据库记录
- 一条UPDATE语句，用于创建从对象p到对象c的连接

这样做不仅效率低，而且违反了列parent\_id非空的限制。

底层的原因是，对象p到对象c的连接（外键parent\_id）没有被当作是Child对象状态的一部分，也没有在INSERT的时候被创建。解决的办法是，在Child一端设置映射。

```
<many-to-one name="parent" column="parent_id" not-null
```

（我们还需要为类Child添加parent属性）

现在实体Child在管理连接的状态，为了使collection不更新连接，我们使用inverse属性。

```
<set name="children" inverse="true">
  <key column="parent_id"/>
  <one-to-many class="Child"/>
</set>
```

下面的代码是用来添加一个新的Child

```
Parent p = (Parent) session.load(Parent.class, pid);
Child c = new Child();
c.setParent(p);
p.getChildren().add(c);
session.save(c);
session.flush();
```

现在，只会有一条INSERT语句被执行！

为了让事情变得井井有条，可以为Parent加一个addChild()方法。

```
public void addChild(Child c) {
    c.setParent(this);
    children.add(c);
}
```

现在，添加Child的代码就是这样

```
Parent p = (Parent) session.load(Parent.class, pid);
Child c = new Child();
p.addChild(c);
session.save(c);
session.flush();
```

### 9.3. 级联 ( Cascades )

对每个对象调用`save()`方法很麻烦，我们可以用级联来解决这个问题。

```
<set name="children" inverse="true" cascade="all">
  <key column="parent_id"/>
  <one-to-many class="Child"/>
</set>
```

配置级联以后，代码就可以这样写：

```
Parent p = (Parent) session.load(Parent.class, pid);
Child c = new Child();
p.addChild(c);
session.flush();
```

同样的，保存或删除`Parent`对象的时候并不需要遍历其子对象。下面的代码会删除对象`p`及其所有子对象对应的数据库记录。

```
Parent p = (Parent) session.load(Parent.class, pid);
session.delete(p);
session.flush();
```

然而，这段代码

```
Parent p = (Parent) session.load(Parent.class, pid);
Child c = (Child) p.getChildren().iterator().next();
p.getChildren().remove(c);
c.setParent(null);
session.flush();
```

不会从数据库删除`c`；它只会删除与`p`之间的连接（并且会导致违反`NOT NULL`约束，在这个例子中）。你需要明确调用`Child`的`delete()`方法。

```
Parent p = (Parent) session.load(Parent.class, pid);
Child c = (Child) p.getChildren().iterator().next();
```

```
p.getChildren().remove(c);  
session.delete(c);  
session.flush();
```

在我们的例子中，如果我们规定没有父对象的话，子对象就不应该存在，如果将子对象从collection中移除，实际上我们是想删除它。要实现这种要求，就必须使用`cascade="all-delete-orphan"`。

```
<set name="children" inverse="true" cascade="all-delete-orphan">  
  <key column="parent_id"/>  
  <one-to-many class="Child"/>  
</set>
```

注意：即使在collection一方的映射中指定`inverse="true"`，在遍历collection的时候级联操作仍然会执行。如果你想要通过级联进行子对象的插入、删除、更新操作，就必须把它加到collection中，只调用`setParent()`是不够的。

## 9.4. 级联更新 ( Using cascading update() )

假设我们从Session中装入了一个Parent对象，用户界面对其进行了修改，然后我们希望在新的Session里面调用update()来更新它。对象Parent包含了子对象的集合，由于打开了级联更新，Hibernate需要知道哪些子对象是新的，哪些是数据库中已经存在的。我们假设Parent和Child对象的标识属性的类型为java.lang.Long。Hibernate会使用标识属性的值来判断哪些子对象是新的。(你也可以使用version 或 timestamp 属性，参见Section 8.5, “更改在以前session中保存或者装载的对象”。)

unsaved-value属性是用来表示新实例的标识属性值的，缺省为"null"，用在Long类型的标识类型再好不过了。如果我们使用原始类型作为标识类型的话，我们在配置Child类映射的时候就必须写：

```
<id name="id" type="long" unsaved-value="0">
```

(为版本和时间戳属性进行映射，也会有另一个叫做unsaved-value的属性。)

下面的代码会更新parent和child对象，并且插入newChild对象。

```
//parent and child were both loaded in a previous sess
parent.addChild(child);
Child newChild = new Child();
parent.addChild(newChild);
session.update(parent);
session.flush();
```

好的，对于自动生成标识的情况这样做很方便，但是自分配的标识和复合标识怎么办呢？这是有点麻烦，因为unsaved-value无法区分新对象（标识是用户指定的）和前一个Session装入的对象。在这种情况下，你可能需要给Hibernate一些提示，在调用update(parent)之前：

- 在这个类的<version> or <timestamp>属性映射上定义unsaved-value="null"或者unsaved-value="negative"。
- 在对父对象执行update(parent)之前，设定unsaved-value="none"并且显

式的调用save()在数据库创建新子对象

- 在对父对象执行update(parent)之前，设定unsaved-value="any"并且显式的调用update()更新已经装入的子对象

none是自分配标识和复合标识的unsaved-value的缺省值。

There is one further possibility. There is a new Interceptor method named isUnsaved() which lets the application implement its own strategy for distinguishing newly instantiated objects. For example, you could define a base class for your persistent classes.

还有一种可能情况，有一个名为isUnsaved()的拦截器(Interceptor)方法，它允许应用程序自己实现新实例的判断。比如，你可以自己定义一个持久类的祖先类：

```
public class Persistent implements Lifecycle {
    private boolean _saved = false;
    public boolean onSave(Session s) {
        _saved=true;
        return NO_VETO;
    }
    public void onLoad(Session s, Serializable id) {
        _saved=true;
    }
    .....
    public boolean isSaved() {
        return _saved;
    }
}
```

And implement isUnsaved()

(saved属性是不会被持久化的。)现在在onLoad()和onSave()外，还要实现isUnsaved()。

```
public Boolean isUnsaved(Object entity) {
    if (entity instanceof Persistent) {
        return new Boolean( !( (Persistent) entity ).i
```

```
    }
    else {
        return null;
    }
}

public boolean onLoad(Object entity,
    Serializable id,
    Object[] state,
    String[] propertyNames,
    Type[] types) {

    if (entity instanceof Persistent) ( (Persistent) e
    return false;
}

public boolean onSave(Object entity,
    Serializable id,
    Object[] state,
    String[] propertyNames,
    Type[] types) {

    if (entity instanceof Persistent) ( (Persistent) e
    return false;
}
```

## 9.5. 结论

这个问题往往让新手感到迷惑，它确实不太容易消化。不过，经过一些实践以后，你会感觉越来越顺手。父子对象模式已经被广泛的应用在Hibernate应用程序中。

在第一段中我们曾经提到另一个方案。复合元素的语义与父子关系是等同的，但是我们并没有详细讨论。很不幸复合元素还有两个重大限制:复合元素不能拥有collections，并且，除了用于惟一的父对象外，它们不能再作为其它任何实体的子对象。（但是，通过使用<idbag>映射，它们可能拥有代理主键。）

## Chapter 10. Hibernate查询语言(Query Language), 即HQL

Hibernate装备了一种极为有力的查询语言，（有意地）看上去很像SQL。但是别被语法蒙蔽，HQL是完全面向对象的，具备继承、多态和关联等特性。

## 10.1. 大小写敏感性(Case Sensitivity)

除了Java类和属性名称外，查询都是大小写不敏感的。所以，`SeLeCT` 和 `sSElect` 以及 `SELECT` 相同的，但是 `net.sf.hibernate.eg.FOO` 和 `net.sf.hibernate.eg.Foo` 是不同的，`foo.barSet` 和 `foo.BARSET` 也是不同的。

本手册使用小写的HQL关键词。有些用户认为在查询中使用大写的关键字更加易读，但是我们认为嵌入在Java代码中这样很难看。

## 10.2. from 子句

可能最简单的Hibernate查询是这样的形式：

```
from eg.Cat
```

它简单的返回所有`eg.Cat`类的实例。

大部分情况下，你需要赋予它一个别名 (*alias*)，因为你在查询的其他地方也会引用这个`Cat`。

```
from eg.Cat as cat
```

上面的语句为`Cat`赋予了一个别名`cat`。所以后面的查询可以用这个简单的别名了。`as`关键字是可以省略的，我们也可以写成这样：

```
from eg.Cat cat
```

可以出现多个类，结果是它们的笛卡尔积，或者称为“交叉”连接。

```
from Formula, Parameter
```

```
from Formula as form, Parameter as param
```

让查询中的别名服从首字母小写的规则，我们认为这是一个好习惯。这和Java对局部变量的命名规范是一致的。(比如，`domesticCat`).

## 10.3. 联合 ( Associations ) 和连接 ( joins )

你可以使用`join`定义两个实体的连接，同时指明别名。

```
from eg.Cat as cat
    inner join cat.mate as mate
    left outer join cat.kittens as kitten

from eg.Cat as cat left join cat.mate.kittens as kitten

from Formula form full join form.parameter param
```

支持的连接类型是从ANSI SQL借用的：

- 内连接, `inner join`
- 左外连接, `left outer join`
- 右外连接, `right outer join`
- 全连接, `full join` (不常使用)

`inner join`, `left outer join` 和 `right outer join` 都可以简写。

```
from eg.Cat as cat
    join cat.mate as mate
    left join cat.kittens as kitten
```

并且，加上 "fetch" 后缀的抓取连接可以让联合的对象随着它们的父对象的初始化而初始化，只需要一个`select`语句。这在初始化一个集合的时候特别有用。

```
from eg.Cat as cat
    inner join fetch cat.mate
    left join fetch cat.kittens
```

抓取连接一般不需要赋予别名，因为被联合的对象应该不会在`where`子句

( 或者任何其它子句 ) 中出现。并且，被联合的对象也不会出现在查询结果中直接出现。它们是通过父对象进行访问的。

请注意，目前的实现中，在一次查询中只会抓取一个集合（原文为：only one collection role may be fetched in a query ）。也请注意，在使用scroll()或者 iterate()方式调用的查询中，是禁止使用fetch构造的。最后，请注意full join fetch和right join fetch是没有意义的。

## 10.4. select子句

select子句选择在结果集中返回哪些对象和属性。思考一下下面的例子：

```
select mate
from eg.Cat as cat
     inner join cat.mate as mate
```

这个查询会选择出作为其它猫（cat）朋友（mate）的那些猫。当然，你可以更加直接的写成下面的形式：

```
select cat.mate from eg.Cat cat
```

你甚至可以选择集合元素，使用特殊的elements功能。下面的查询返回所有猫的小猫。

```
select elements(cat.kittens) from eg.Cat cat
```

查询可以返回任何值类型的属性，包括组件类型的属性：

```
select cat.name from eg.DomesticCat cat
where cat.name like 'fri%'

select cust.name.firstName from Customer as cust
```

查询可以用元素类型是Object[]的一个数组返回多个对象和/或多个属性。

```
select mother, offspr, mate.name
from eg.DomesticCat as mother
     inner join mother.mate as mate
     left outer join mother.kittens as offspr
```

或者实际上是类型安全的Java对象

```
select new Family(mother, mate, offspr)
from eg.DomesticCat as mother
```

```
join mother.mate as mate  
left join mother.kittens as offspr
```

上面的代码假定Family有一个合适的构造函数。

## 10.5. 统计函数(Aggregate functions)

查询可以返回属性的统计函数。

```
select avg(cat.weight), sum(cat.weight), max(cat.weight)
from eg.Cat cat
```

在select子句中，统计函数的变量也可以是集合。

```
select cat, count( elements(cat.kittens) )
from eg.Cat cat group by cat
```

下面是支持的统计函数列表：

- avg(...), sum(...), min(...), max(...)
- count(\*)
- count(...), count(distinct ...), count(all...)

distinct 和 all关键字的用法和语义与SQL相同。

```
select distinct cat.name from eg.Cat cat
select count(distinct cat.name), count(cat) from eg.Cat
```

## 10.6. 多态(polymorphism)

类似下面的查询：

```
from eg.Cat as cat
```

返回的实例不仅仅是`Cat`，也有可能是子类的实例，比如`DomesticCat`。Hibernate查询可以在`from`子句中使用任何Java类或者接口的名字。查询可能返回所有继承自这个类或者实现这个接口的持久化类的实例。下列查询会返回所有的持久化对象：

```
from java.lang.Object o
```

可能有多个持久化类都实现了`Named`接口：

```
from eg.Named n, eg.Named m where n.name = m.name
```

请注意，上面两个查询都使用了超过一个SQL的`SELECT`。这意味着`order by`子句将不会正确排序。（这也意味着你不能对这些查询使用`query.scroll()`。）

## 10.7. where子句

where子句让你缩小你要返回的实例的列表范围。

```
from eg.Cat as cat where cat.name='Fritz'
```

返回所有名字为'Fritz'的Cat的实例。

```
select foo
from eg.Foo foo, eg.Bar bar
where foo.startDate = bar.date
```

会返回所有的满足下列条件的Foo实例，它们存在一个对应的bar实例，其date属性与Foo的startDate属性相等。复合路径表达式令where子句变得极为有力。思考下面的例子：

```
from eg.Cat cat where cat.mate.name is not null
```

这个查询会被翻译为带有一个表间 (inner)join的SQL查询。如果你写下类似这样的语句：

```
from eg.Foo foo
where foo.bar.baz.customer.address.city is not null
```

你最终会得到的查询，其对应的SQL需要4个表间连接。

=操作符不仅仅用于判断属性是否相等，也可以用于实例：

```
from eg.Cat cat, eg.Cat rival where cat.mate = rival.mate

select cat, mate
from eg.Cat cat, eg.Cat mate
where cat.mate = mate
```

特别的，小写的id可以用来表示一个对象的惟一标识。（你可以使用它的属性名。）

```
from eg.Cat as cat where cat.id = 123  
  
from eg.Cat as cat where cat.mate.id = 69
```

第二个查询是很高效的。不需要进行表间连接！

组合的标示符也可以使用。假设Person有一个组合标示符，是由country和medicareNumber组合而成的。

```
from bank.Person person  
where person.id.country = 'AU'  
      and person.id.medicareNumber = 123456  
  
from bank.Account account  
where account.owner.id.country = 'AU'  
      and account.owner.id.medicareNumber = 123456
```

又一次，第二个查询不需要表间连接。

类似的，在存在多态持久化的情况下，特殊属性class用于获取某个实例的辨识值。在where子句中嵌入的Java类名将会转换为它的辨识值。

```
from eg.Cat cat where cat.class = eg.DomesticCat
```

你也可以指定组件（或者是组件的组件，依次类推）或者组合类型中的属性。但是在一个存在路径的表达式中，最后不能以一个组件类型的属性结尾。（这里不是指组件的属性）。比如，假若store.owner这个实体的address是一个组件

```
store.owner.address.city    //okay  
store.owner.address        //error!
```

“任意(any)”类型也有特殊的id属性和class属性，这可以让我们用下面的形式来表达连接（这里AuditLog.item是一个对应到<ant>的属性）。

```
from eg.AuditLog log, eg.Payment payment  
where log.item.class = 'eg.Payment' and log.item.id =
```

注意上面查询中，`log.item.class`和`payment.class`会指向两个值，代表完全不同的数据库字段。

## 10.8. 表达式(Expressions)

where子句允许出现的表达式包括了你在SQL中使用的大多数情况：

- 数学操作+, -, \*, /
- 真假比较操作 =, >=, <=, <>, !=, like
- 逻辑操作 and, or, not
- 字符串连接 ||
- SQL标量 ( scalar ) 函数 , 例如 upper() 和 lower()
- 没有前缀的 ( )表示分组
- in, between, is null
- JDBC 传入参数?
- 命名参数 :name, :start\_date, :x1
- SQL 文字 'foo', 69, '1970-01-01 10:00:01.0'
- Java的public static final常量 比如 Color.TABBY

in 和 between 可以如下例一样使用:

```
from eg.DomesticCat cat where cat.name between 'A' and 'B'  
from eg.DomesticCat cat where cat.name in ( 'Foo', 'Bar' )
```

其否定形式为

```
from eg.DomesticCat cat where cat.name not between 'A' and 'B'  
from eg.DomesticCat cat where cat.name not in ( 'Foo', 'Bar' )
```

类似的，`is null`和`is not null`可以用来测试`null`值。

通过在Hibernate配置中声明HQL查询的替换方式，`Boolean`也是很容易在表达式中使用的：

```
<property name="hibernate.query.substitutions">true 1,
```

在从HQL翻译成SQL的时候,关键字`true`和`false`就会被替换成`1`和`0`。

```
from eg.Cat cat where cat.alive = true
```

你可以用特殊属性`size`来测试一个集合的长度，或者用特殊的`size()`函数也可以。

```
from eg.Cat cat where cat.kittens.size > 0
```

```
from eg.Cat cat where size(cat.kittens) > 0
```

对于排序集合，你可以用`minIndex`和`maxIndex`来获取其最大索引值和最小索引值。类似的，`minElement`和`maxElement`可以用来获取集合中最小和最大的元素，前提是必须是基本类型的集合。

```
from Calendar cal where cal.holidays.maxElement > curr
```

也有函数的形式（和上面的形式不同，函数形式是大小写不敏感的）：

```
from Order order where maxindex(order.items) > 100
```

```
from Order order where minelement(order.items) > 100000
```

SQL中的`any`，`some`，`all`，`exists`，`in`功能也是支持的，前提是必须把集合的元素或者索引集作为它们的参数（使用`element`和`indices`函数），或者使用子查询的结果作为参数。

```
select mother from eg.Cat as mother, eg.Cat as kit  
where kit in elements(foo.kittens)
```

```
select p from eg.NameList list, eg.Person p
where p.name = some elements(list.names)

from eg.Cat cat where exists elements(cat.kittens)

from eg.Player p where 3 > all elements(p.scores)

from eg.Show show where 'fizard' in indices(show.acts)
```

请注意这些设

施：size,elements,indices,minIndex,maxIndex,minElement,maxElement 都有一些使用限制：

- 在where子句中: 只对支持子查询的数据库有效
- 在select子句中：只有elements和indices有效

有序的集合(数组、list、map)的元素可以用索引来进行引用（只限于在where子句中）

```
from Order order where order.items[0].id = 1234

select person from Person person, Calendar calendar
where calendar.holidays['national day'] = person.birth
      and person.nationality.calendar = calendar

select item from Item item, Order order
where order.items[ order.deliveredItemIndices[0] ] = item

select item from Item item, Order order
where order.items[ maxindex(order.items) ] = item and
```

[]中的表达式允许是另一个数学表达式。

```
select item from Item item, Order order
where order.items[ size(order.items) - 1 ] = item
```

HQL也对一对多关联或者值集合提供内置的index()函数。

```
select item, index(item) from Order order
      join order.items item
where index(item) < 5
```

底层数据库支持的标量SQL函数也可以使用

```
from eg.DomesticCat cat where upper(cat.name) like 'FR'
```

假如以上的这些还没有让你信服的话，请想象一下下面的查询假若用SQL来写，会变得多么长，多么不可读：

```
select cust
from Product prod,
     Store store
     inner join store.customers cust
where prod.name = 'widget'
     and store.location.name in ( 'Melbourne', 'Sydney' )
     and prod = all elements(cust.currentOrder.lineItems)
```

提示：对应的SQL语句可能是这样的

```
SELECT cust.name, cust.address, cust.phone, cust.id, c
FROM customers cust,
     stores store,
     locations loc,
     store_customers sc,
     product prod
WHERE prod.name = 'widget'
     AND store.loc_id = loc.id
     AND loc.name IN ( 'Melbourne', 'Sydney' )
     AND sc.store_id = store.id
     AND sc.cust_id = cust.id
     AND prod.id = ALL(
     SELECT item.prod_id
     FROM line_items item, orders o
     WHERE item.order_id = o.id
           AND cust.current_order = o.id
     )
```



## 10.9. order by 子句

查询返回的列表可以按照任何返回的类或者组件的属性排序：

```
from eg.DomesticCat cat
order by cat.name asc, cat.weight desc, cat.birthdate
```

asc和desc是可选的，分别代表升序或者降序。

## 10.10. group by 子句

返回统计值的查询可以按照返回的类或者组件的任何属性排序：

```
select cat.color, sum(cat.weight), count(cat)
from eg.Cat cat
group by cat.color

select foo.id, avg( elements(foo.names) ), max( indices
from eg.Foo foo
group by foo.id
```

请注意：你可以在select子句中使用elements和indices指令，即使你的数据库不支持子查询也可以。

having子句也是允许的。

```
select cat.color, sum(cat.weight), count(cat)
from eg.Cat cat
group by cat.color
having cat.color in (eg.Color.TABBY, eg.Color.BLACK)
```

在having子句中允许出现SQL函数和统计函数，当然这需要底层数据库支持才行。（比如说,MySQL就不支持）

```
select cat
from eg.Cat cat
    join cat.kittens kitten
group by cat
having avg(kitten.weight) > 100
order by count(kitten) asc, sum(kitten.weight) desc
```

注意，group by子句和order by子句都不支持数学表达式。

## 10.11. 子查询

对于支持子查询的数据库来说，Hibernate支持在查询中嵌套子查询。子查询必须由圆括号包围（常常是在一个SQL统计函数中）。也允许关联子查询（在外部查询中作为一个别名出现的子查询）。

```
from eg.Cat as fatcat
where fatcat.weight > (
    select avg(cat.weight) from eg.DomesticCat cat
)

from eg.DomesticCat as cat
where cat.name = some (
    select name.nickName from eg.Name as name
)

from eg.Cat as cat
where not exists (
    from eg.Cat as mate where mate.mate = cat
)

from eg.DomesticCat as cat
where cat.name not in (
    select name.nickName from eg.Name as name
)
```

## 10.12. 示例

Hibernate查询可以非常强大复杂。实际上，强有力的查询语言是Hibernate的主要卖点之一。下面给出的示例与我在近期实际项目中使用的查询很类似。请注意你编写的查询大部分等都不会这么复杂！

下面的查询对特定的客户，根据给定的最小总计值（minAmount），查询出所有未付订单，返回其订单号、货品总数、订单总金额，结果按照总金额排序。在决定价格的时候，参考当前目录。产生的SQL查询，在ORDER,ORDER\_LINE,PRODUCT,CATALOG和PRICE表之间有四个内部连接和一个没有产生关联的字查询。

```
select order.id, sum(price.amount), count(item)
from Order as order
    join order.lineItems as item
    join item.product as product,
    Catalog as catalog
    join catalog.prices as price
where order.paid = false
    and order.customer = :customer
    and price.product = product
    and catalog.effectiveDate < sysdate
    and catalog.effectiveDate >= all (
        select cat.effectiveDate
        from Catalog as cat
        where cat.effectiveDate < sysdate
    )
group by order
having sum(price.amount) > :minAmount
order by sum(price.amount) desc
```

好家伙，真长！实际上，在现实生活中我并不是非常热衷于子查询，所以我的查询往往是这样的：

```
select order.id, sum(price.amount), count(item)
from Order as order
    join order.lineItems as item
```

```

    join item.product as product,
    Catalog as catalog
    join catalog.prices as price
where order.paid = false
    and order.customer = :customer
    and price.product = product
    and catalog = :currentCatalog
group by order
having sum(price.amount) > :minAmount
order by sum(price.amount) desc

```

下面的查询统计付款记录处于每种状态中的数量，要排除所有处于AWAITING\_APPROVAL状态的，或者最近一次状态更改是由当前用户做出的。它翻译成SQL查询后，在PAYMENT,PAYMENT\_STATUS和PAYMENT\_STATUS\_CHANGE表之间包含两个内部连接和一个用于关联的子查询。

```

select count(payment), status.name
from Payment as payment
    join payment.currentStatus as status
    join payment.statusChanges as statusChange
where payment.status.name <> PaymentStatus.AWAITING_AF
    or (
        statusChange.timeStamp = (
            select max(change.timeStamp)
            from PaymentStatusChange change
            where change.payment = payment
        )
        and statusChange.user <> :currentUser
    )
group by status.name, status.sortOrder
order by status.sortOrder

```

假若我已经把statusChange集合映射为一个列表而不是一个集合的话，查询写起来会简单很多。

```

select count(payment), status.name
from Payment as payment
    join payment.currentStatus as status

```

```
where payment.status.name <> PaymentStatus.AWAITING_AF  
      or payment.statusChanges[ maxIndex(payment.statusC  
group by status.name, status.sortOrder  
order by status.sortOrder
```

下面的查询使用了MS SQL Server的isNull()函数，返回当前用户所属的组织所有账户和未付支出。翻译为SQL查询后，在ACCOUNT, PAYMENT, PAYMENT\_STATUS, ACCOUNT\_TYPE, ORGANIZATION 和 ORG\_USER表之间有三个内部连接，一个外部连接和一个子查询。

```
select account, payment  
from Account as account  
      left outer join account.payments as payment  
where :currentUser in elements(account.holder.users)  
      and PaymentStatus.UNPAID = isNull(payment.currentS  
order by account.type.sortOrder, account.accountNumber
```

对某些数据库而言，我们可能不能依赖（关联的）子查询。

```
select account, payment  
from Account as account  
      join account.holder.users as user  
      left outer join account.payments as payment  
where :currentUser = user  
      and PaymentStatus.UNPAID = isNull(payment.currentS  
order by account.type.sortOrder, account.accountNumber
```

## 10.13. 提示和技巧 ( Tips & Tricks )

你不返回结果集也可以查询结果集的大小：

```
( (Integer) session.iterate("select count(*) from ..."))
```

要依据一个集合的大小对结果集排序，可以用下面的查询来对付一对多或多对多的关联：

```
select usr
from User as usr
     left join usr.messages as msg
group by usr
order by count(msg)
```

如果你的数据库支持子查询，你可以在查询的where子句中对选择的大小进行条件限制：

```
from User usr where size(usr.messages) >= 1
```

如果你的数据库不支持子查询，可以使用下列查询：

```
select usr.id, usr.name
from User usr
     join usr.messages msg
group by usr.id, usr.name
having count(msg) >= 1
```

因为使用了inner join,这个解决方法不能返回没有message的User.下面的方式就可以：

```
select usr
from User as usr
     left join usr.messages as msg
group by usr
having count(msg) = 0
```

JavaBean的属性可以直接作为命名的查询参数：

```
Query q = s.createQuery("from foo in class Foo where f  
q.setProperties(fooBean); // fooBean has getName() and  
List foos = q.list();
```

在Query接口中使用过滤器(filter),可以对集合分页：

```
Query q = s.createFilter( collection, "" ); // the tri  
q.setMaxResults(PAGE_SIZE);  
q.setFirstResult(PAGE_SIZE * pageNumber);  
List page = q.list();
```

集合元素可以使用查询过滤器(query filter)进行排序或者分组：

```
List orderedCollection = s.filter( collection, "order  
List counts = s.filter( collection, "select this.type,
```

不用初始化集合就可以得到其大小：

```
( (Integer) session.iterate("select count(*) from ....
```

## Chapter 11. 实例(A Worked Example)

我们用实例来演示上两章所讲述的概念。

## 11.1. 持久化类

下面的两个持久化类表示一个weblog,和在其中张贴的一个帖子。他们是标准的父/子关系模型,但是我们会用一个排序包 ( ordered bag)而非集合 (set)。

```
package eg;

import java.util.List;

public class Blog {
    private Long _id;
    private String _name;
    private List _items;

    public Long getId() {
        return _id;
    }
    public List getItems() {
        return _items;
    }
    public String getName() {
        return _name;
    }
    public void setId(Long long1) {
        _id = long1;
    }
    public void setItems(List list) {
        _items = list;
    }
    public void setName(String string) {
        _name = string;
    }
}
```

```
package eg;
```

```
import java.text.DateFormat;
import java.util.Calendar;

public class BlogItem {
    private Long _id;
    private Calendar _datetime;
    private String _text;
    private String _title;
    private Blog _blog;

    public Blog getBlog() {
        return _blog;
    }
    public Calendar getDatetime() {
        return _datetime;
    }
    public Long getId() {
        return _id;
    }
    public String getText() {
        return _text;
    }
    public String getTitle() {
        return _title;
    }
    public void setBlog(Blog blog) {
        _blog = blog;
    }
    public void setDatetime(Calendar calendar) {
        _datetime = calendar;
    }
    public void setId(Long long1) {
        _id = long1;
    }
    public void setText(String string) {
        _text = string;
    }
    public void setTitle(String string) {
        _title = string;
    }
}
```

```
}  
}
```

## 11.2. Hibernate 映射

下列的XML映射应该是很直白的。

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 2.0//EN"
    "http://hibernate.sourceforge.net/hibernate-ma

<hibernate-mapping package="eg">
  <class
    name="Blog"
    table="BLOGS"
    lazy="true">

    <id
      name="id"
      column="BLOG_ID">

      <generator class="native"/>

    </id>

    <property
      name="name"
      column="NAME"
      not-null="true"
      unique="true"/>

    <bag
      name="items"
      inverse="true"
      lazy="true"
      order-by="DATE_TIME"
      cascade="all">

      <key column="BLOG_ID"/>
```

```
        <one-to-many class="BlogItem"/>
    </bag>
</class>
</hibernate-mapping>
```

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 2.0//EN"
    "http://hibernate.sourceforge.net/hibernate-ma
<hibernate-mapping package="eg">
    <class
        name="BlogItem"
        table="BLOG_ITEMS"
        dynamic-update="true">
        <id
            name="id"
            column="BLOG_ITEM_ID">
            <generator class="native"/>
        </id>
        <property
            name="title"
            column="TITLE"
            not-null="true"/>
        <property
            name="text"
            column="TEXT"
            not-null="true"/>
```

```
<property
  name="datetime"
  column="DATE_TIME"
  not-null="true"/>

<many-to-one
  name="blog"
  column="BLOG_ID"
  not-null="true"/>

</class>

</hibernate-mapping>
```

## 11.3. Hibernate 代码

下面的类演示了我们可以使用Hibernate对这些类所进行的操作。

```
package eg;

import java.util.ArrayList;
import java.util.Calendar;
import java.util.Iterator;
import java.util.List;

import net.sf.hibernate.HibernateException;
import net.sf.hibernate.Query;
import net.sf.hibernate.Session;
import net.sf.hibernate.SessionFactory;
import net.sf.hibernate.Transaction;
import net.sf.hibernate.cfg.Configuration;
import net.sf.hibernate.tool.hbm2ddl.SchemaExport;

public class BlogMain {

    private SessionFactory _sessions;

    public void configure() throws HibernateException {
        _sessions = new Configuration()
            .addClass(Blog.class)
            .addClass(BlogItem.class)
            .buildSessionFactory();
    }

    public void exportTables() throws HibernateException {
        Configuration cfg = new Configuration()
            .addClass(Blog.class)
            .addClass(BlogItem.class);
        new SchemaExport(cfg).create(true, true);
    }
}
```

```
public Blog createBlog(String name) throws Hibernate

    Blog blog = new Blog();
    blog.setName(name);
    blog.setItems( new ArrayList() );

    Session session = _sessions.openSession();
    Transaction tx = null;
    try {
        tx = session.beginTransaction();
        session.save(blog);
        tx.commit();
    }
    catch (HibernateException he) {
        if (tx!=null) tx.rollback();
        throw he;
    }
    finally {
        session.close();
    }
    return blog;
}
```

```
public BlogItem createBlogItem(Blog blog, String t

    BlogItem item = new BlogItem();
    item.setTitle(title);
    item.setText(text);
    item.setBlog(blog);
    item.setDatetime( Calendar.getInstance() );
    blog.getItems().add(item);

    Session session = _sessions.openSession();
    Transaction tx = null;
    try {
        tx = session.beginTransaction();
        session.update(blog);
        tx.commit();
    }
}
```

```

        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
        return item;
    }

    public BlogItem createBlogItem(Long blogid, String title, String text) {

        BlogItem item = new BlogItem();
        item.setTitle(title);
        item.setText(text);
        item.setDatetime( Calendar.getInstance() );

        Session session = _sessions.openSession();
        Transaction tx = null;
        try {
            tx = session.beginTransaction();
            Blog blog = (Blog) session.load(Blog.class, blogid);
            item.setBlog(blog);
            blog.getItems().add(item);
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
        return item;
    }

    public void updateBlogItem(BlogItem item, String text) {

        item.setText(text);
    }

```

```

    Session session = _sessions.openSession();
    Transaction tx = null;
    try {
        tx = session.beginTransaction();
        session.update(item);
        tx.commit();
    }
    catch (HibernateException he) {
        if (tx!=null) tx.rollback();
        throw he;
    }
    finally {
        session.close();
    }
}

```

```

public void updateBlogItem(Long itemid, String text)

```

```

    Session session = _sessions.openSession();
    Transaction tx = null;
    try {
        tx = session.beginTransaction();
        BlogItem item = (BlogItem) session.load(BlogItem.class, itemid);
        item.setText(text);
        tx.commit();
    }
    catch (HibernateException he) {
        if (tx!=null) tx.rollback();
        throw he;
    }
    finally {
        session.close();
    }
}

```

```

public List listAllBlogNamesAndItemCounts(int max)

```

```

    Session session = _sessions.openSession();

```

```

Transaction tx = null;
List result = null;
try {
    tx = session.beginTransaction();
    Query q = session.createQuery(
        "select blog.id, blog.name, count(blog
        "from Blog as blog " +
        "left outer join blog.items as blogIte
        "group by blog.name, blog.id " +
        "order by max(blogItem.datetime)"
    );
    q.setMaxResults(max);
    result = q.list();
    tx.commit();
}
catch (HibernateException he) {
    if (tx!=null) tx.rollback();
    throw he;
}
finally {
    session.close();
}
return result;
}

```

```

public Blog getBlogAndAllItems(Long blogid) throws

```

```

    Session session = _sessions.openSession();
    Transaction tx = null;
    Blog blog = null;
    try {
        tx = session.beginTransaction();
        Query q = session.createQuery(
            "from Blog as blog " +
            "left outer join fetch blog.items " +
            "where blog.id = :blogid"
        );
        q.setParameter("blogid", blogid);
        blog = (Blog) q.list().get(0);
    }

```

```

        tx.commit();
    }
    catch (HibernateException he) {
        if (tx!=null) tx.rollback();
        throw he;
    }
    finally {
        session.close();
    }
    return blog;
}

```

```

public List listBlogsAndRecentItems() throws Hiber

```

```

    Session session = _sessions.openSession();
    Transaction tx = null;
    List result = null;
    try {
        tx = session.beginTransaction();
        Query q = session.createQuery(
            "from Blog as blog " +
            "inner join blog.items as blogItem " +
            "where blogItem.datetime > :minDate"
        );

        Calendar cal = Calendar.getInstance();
        cal.roll(Calendar.MONTH, false);
        q.setCalendar("minDate", cal);

        result = q.list();
        tx.commit();
    }
    catch (HibernateException he) {
        if (tx!=null) tx.rollback();
        throw he;
    }
    finally {
        session.close();
    }
}

```

```
    return result;  
  }  
}
```

## Chapter 12. 性能提升 ( Improving Performance )

我们已经为您展示了如何在对集合持久化时使用延迟装载 ( lazy initialization )。对于通常的对象引用，使用CGLIB代理可以达到类似的效果。我们也提到过Hibernate在Session级别缓存持久化对象。还有更多先进的缓存策略，你可以为每一个类单独配置。

这一章里，我们来教你如何使用这些特性，在必要的时候得到高得多的性能。

## 12.1. 用于延迟装载的代理

Hibernate使用动态字节码增强技术来实现持久化对象的延迟装载代理（使用优秀的CGLIB库）。

映射文件为每一个类声明一个类或者接口作为代理接口。建议使用这个类自身：

```
<class name="eg.Order" proxy="eg.Order">
```

运行时的代理应该是Order的子类。注意被代理的类必须实现一个默认的构造器，并且至少在包内可见。

在扩展这种方法来对应多形的类时，要注意一些细节，比如：

```
<class name="eg.Cat" proxy="eg.Cat">
    .....
    <subclass name="eg.DomesticCat" proxy="eg.DomesticCat">
        .....
    </subclass>
</class>
```

首先，Cat永远不能被强制转换为DomesticCat，即使实际上该实例就是一个DomesticCat实例。

```
Cat cat = (Cat) session.load(Cat.class, id); // instance
if ( cat.isDomesticCat() ) {                // hit t
    DomesticCat dc = (DomesticCat) cat;      // Error
    .....
}
```

其次，代理的==可能不再成立。

```
Cat cat = (Cat) session.load(Cat.class, id);
DomesticCat dc =
    (DomesticCat) session.load(DomesticCat.class, id);
```

```
System.out.println(cat==dc);
```

虽然如此，这种情况并不像看上去得那么糟。虽然我们有两个不同的引用来指向不同的代理对象，实际上底层的实例应该是同一个对象：

```
cat.setWeight(11.0); // hit the db to initialize the
System.out.println( dc.getWeight() ); // 11.0
```

第三，你不能对`final`的类或者具有`final`方法的类使用CGLIB代理。

最后，假如你的持久化对象在实例化的时候需要某些资源（比如，在实例化方法或者默认构造方法中），这些资源也会被代理需要。代理类实际上是持久化类的子类。

这些问题都来源于Java的单根继承模型的天生限制。如果你希望避免这些问题，你的每个持久化类必须抽象出一个接口，声明商业逻辑方法。你应该在映射文件中指定这些接口，比如：

```
<class name="eg.Cat" proxy="eg.ICat">
    .....
    <subclass name="eg.DomesticCat" proxy="eg.IDomesti
    .....
    </subclass>
</class>
```

这里`Cat`实现`ICat`接口，并且`DomesticCat`实现`IDomesticCat`接口。于是`load()`或者`iterate()`就会返回`Cat`和`DomesticCat`的实例的代理。（注意`find()`不会返回代理。）

```
ICat cat = (ICat) session.load(Cat.class, catid);
Iterator iter = session.iterate("from cat in class eg.
ICat fritz = (ICat) iter.next();
```

关系也是延迟装载的。这意味着你必须把任何属性声明为`ICat`类型，而非`Cat`。

某些特定操作不需要初始化代理

- equals(), 假如持久化类没有重载equals()
- hashCode(), 假如持久化类没有重载hashCode()
- 标识符的get方法

Hibernate会识别出重载了equals() 或者 hashCode()方法的持久化类。

在初始化代理的时候发生的异常会被包装成LazyInitializationException。

有时候我们需要保证在Session关闭前某个代理或者集合已经被初始化了。当然，我们总是可以通过调用cat.getSex()或者 cat.getKittens().size()之类的方法来确保这一点。但是这样程序可读性不佳，也不符合通常的代码规范。静态方法Hibernate.initialize()和Hibernate.isInitialized()给你的应用程序一个正常的途径来加载集合或代理。Hibernate.initialize(cat)会强制初始化一个代理,cat,只要它的Session仍然打开。Hibernate.initialize(cat.getKittens() )对kittens的集合具有同样的功能。

## 12.2. 第二层缓存(The Second Level Cache)s

HibernateSession是事务级别的持久化数据缓存。再为每个类或者每个集合配置一个集群或者JVM级别(SessionFactory级别)的缓存也是有可能的。你甚至可以插入一个集群的缓存。要小心，缓存永远不会知道其他进程可能对持久化仓库（数据库）进行的修改（即使他们可能设定为经常对缓存的数据进行失效）。

默认情况下，Hibernate使用EHCACHE进行JVM级别的缓存。但是，对JCS的支持现在已经被废弃了，未来版本的Hibernate将会去掉它。通过hibernate.cache.provider\_class属性，你也可以指定其他缓存，只要其实现了net.sf.hibernate.cache.CacheProvider接口。

**Table 12.1. Cache Providers**

Cache	Provider class	Type
Hashtable (not intended for production use)	net.sf.hibernate.cache.HashtableCacheProvider	memory
EHCACHE	net.sf.ehcache.hibernate.Provider	memory, disk
OSCache	net.sf.hibernate.cache.OSCacheProvider	memory, disk
SwarmCache	net.sf.hibernate.cache.SwarmCacheProvider	clustered (ip multicast)
JBoss TreeCache	net.sf.hibernate.cache.TreeCacheProvider	clustered (ip multicast), transactional

### 12.2.1. 映射(Mapping)

类或者集合映射的<cache>元素可能有下列形式：

```
<cache
    usage="transactional|read-write|nonstr
```



usage 指定了缓存策略：transactional, read-write, nonstrict-read-write 或者 read-only

另外 (推荐首选?), 你可以在hibernate.cfg.xml中指定<class-cache> 和 <collection-cache> 元素。

usage属性指明了缓存并发策略 ( *cache concurrency strategy* )。

### 12.2.2. 只读缓存

如果你的应用程序需要读取一个持久化类的实例，但是并不打算修改它们，可以使用read-only 缓存。这是最简单，也是实用性最好的策略。甚至在集群中，它也能完美地运作。

```
<class name="eg.Immutable" mutable="false">
    ....
    <jcs-cache usage="read-only"/>
</class>
```

### 12.2.3. 读/写缓存

如果应用程序需要更新数据，可能read-write缓存比较合适。如果需要可序列化事务隔离级别 ( serializable transaction isolation level )，这种缓存决不能使用。如果在JTA环境中使用这种缓存，你必须指定hibernate.transaction.manager\_lookup\_class属性的值，给出得到JTA TransactionManager的策略。在其它环境中，你必须确保在Session.close() 或者Session.disconnect()调用前，事务已经结束了。如果你要在集群环境下使用这一策略，你必须确保底层的缓存实现支持锁定(locking)。内置的缓存提供者并不支持。

```

<class name="eg.Cat" .... >
  <jcs-cache usage="read-write"/>
  ....
  <set name="kittens" ... >
    <jcs-cache usage="read-write"/>
    ....
  </set>
</class>

```

### 12.2.4. 不严格的读/写缓存

如果程序偶尔需要更新数据（也就是说，出现两个事务同时更新同一个条目的现象很不常见），也不需要十分严格的事务隔离，可能适用nonstrict-read-write缓存。如果在JTA环境中使用这种缓存，你必须指定hibernate.transaction.manager\_lookup\_class属性的值，给出得到JTA TransactionManager的策略。在其它环境中，你必须确保在Session.close()或者Session.disconnect()调用前，事务已经结束了。

### 12.2.5. 事务缓存（transactional）

transactional缓存策略提供了对全事务缓存提供,比如JBoss TreeCache的支持。这样的缓存只能用于JTA环境，你必须指定hibernate.transaction.manager\_lookup\_class。

没有一种缓存提供者能够支持所有的缓存并发策略。下面的表列出每种提供者与各种并发策略的兼容性。

**Table 12.2. 缓存并发策略支持(Cache Concurrency Strategy Support)**

Cache	read-only	nonstrict-read-write	read-write	transactional
Hashtable (not intended for production use)	yes	yes	yes	
EHCache	yes	yes	yes	
OSCache	yes	yes	yes	

SwarmCache	yes	yes		
JBoss TreeCache	yes			yes

## 12.3. 管理Session缓存

不管何时你传递一个对象给`save()`, `update()`或者 `saveOrUpdate()`, 或者不管何时你使用`load()`, `find()`, `iterate()`或者`filter()`取得一个对象的时候, 该对象被加入到`Session`的内部缓存中。当后继的`flush()`被调用时, 对象的状态会和数据库进行同步。如果你在处理大量对象并且需要有效的管理内存的时候, 你可能不希望发生这种同步, `evict()`方法可以从缓存中去掉对象和它的集合。

```
Iterator cats = sess.iterate("from eg.Cat as cat"); //
while ( cats.hasNext() ) {
    Cat cat = (Cat) iter.next();
    doSomethingWithACat(cat);
    sess.evict(cat);
}
```

`Session`也提供了一个`contains()`方法来判断是否一个实例处于这个`session`的缓存中。

要把所有的对象从`session`缓存中完全清除, 请调用`Session.clear()`。

For the JVM-level JCS cache, there are methods defined on `SessionFactory` for evicting the cached state of an instance, entire class, collection instance or entire collection role.

对于第二层缓存来说, 在`SessionFactory`中定义了一些方法来从缓存中清除一个实例、整个类、集合实例或者整个集合。

## 12.4. 查询缓存(Query Cache)

查询结果集也可以被缓存。只有当经常使用同样的参数进行查询时，这才会有些用处。要使用查询缓存，首先你要打开它，设置 `hibernate.cache.use_query_cache=true` 这个属性。这样会创建两个缓存区域——一个保存查询结果集(`net.sf.hibernate.cache.QueryCache`),另一个保存最近查询的表的时间戳(`net.sf.hibernate.cache.UpdateTimestampsCache`)。请注意查询缓存并不缓存结果集中包含实体的状态；它只缓存标识符属性的值和值类型的结果。所以查询缓存通常会和第二层缓存一起使用。

大多数查询并不会从缓存中获得什么好处，所以默认查询是不进行缓存的。要进行缓存，调用 `query.setCacheable(true)`。这个调用会让查询在执行时去从缓存中查找结果，或者把结果集放到缓存去。

如果你要对查询缓存的失效政策进行精确的控制，你必须调用 `query.setCacheRegion()` 来为每个查询指定一个命名的缓存区域。

```
List blogs = sess.createQuery("from Blog blog where bl
    .setEntity("blogger", blogger)
    .setMaxResults(15)
    .setCacheable(true)
    .setCacheRegion("frontpages")
    .list();
```

## Chapter 13. 理解集合类的性能 ( Understanding Collection Performance )

我们已经在集合类(collections)上面花了很多口舌了。这一章，我们要着重关注集合类在运行时的一些问题。

## 13.1. 分类 ( Taxonomy )

Hibernate定义了三种不同的基本集合类

- 值集合
- 一对多关联
- 多对多关联

这个分类方法是根据不同的表和外键关系来区分的，但是没有确切的告诉我们关系模型。要完整的了解关系结构和性能的区别，我们必须考虑Hibernate再更新或者删除集合类记录时的主键结构。这样的话，我们会得到如下的分类：

- 有序集合类
- 集合 ( sets )
- 包 ( bags )

所有的有序集合类 ( map,list,array)都有一个由<key>和<index>组合的主键字段。这种集合类的更新非常高效——主键有效地排了序，Hibernate要更新或者删除一个元素的时候可以很高效的找到它。

集合(Sets)有一个由<key>和某个元素字段组成的主键。对于某些元素类型，特别是组合元素以及大文本、二进制字段，效率会来的比较低；数据库无法对复杂的主键有效索引。另一方面，一对多关系或者多对多关系来说，特别是使用“人造”的标识符的时候，它的性能同样出色。（注：如果你希望SchemaExport为你创建一个<set>的主键，你必须把所有的字段都声明成non-null="true"。）

包(Bags)是最差劲的。因为包允许元素值重复，也没有索引字段，所以无法定义主键。Hibernate没有办法来区分重复的行。Hibernate的处理方法是，每当更改的时候，完全删除（用一个DELETE),再重新创建这个集合类。这可能是效率极低的。

请注意，对于一对多关联，“主键”可能不是数据库表的物理主键——当时就算考虑这种情况，上面分类描述仍然是正确的。（反映了Hibernate是如何

在不同的集合类中“定位”某条记录的。)

## 13.2. Lists, maps 和sets用于更新效率最高

根据我们上面的讨论，显然有序类型和大多数set可以在增加/删除/修改元素的时候得到最好的性能。

但是，在多对多关联，或者对值元素而言，有序集合类比集合(set)有一个好处。因为Set的结构，如果“改变”了一个元素,Hibernate并不会UPDATE这一行。对Set来说，只有INSERT和DELETE才有效。注意这一段描述一对多关联并不适用。

注意到数组无法延迟转载，我们可以得出结论，list, map和set是最高效的集合类型。（当然，我们警告过了，由于集合中的值的关系，set可能性能下降。）

Set可以被看作是Hibernate程序中最普遍的集合类型。

这个版本的Hibernate有一个没有写在文档中的功能。<idbag>可以对值集合和多对多关联实现bag语义，并且性能比上面任何类型都高！

### 13.3. Bag和list是反向集合类中效率最高的

好了，在你把bag扔到水沟里面再踩上一只脚之前，有一种情况下bag(包括list)要比set性能高得多。对于指明了inverse="true"的集合类(比如说，标准的双向一对多关联)，我们可以在不初始化(fetch)包元素的情况下就增加新元素！这是因为Collection.add()或者Collection.addAll()对bag或者List总是返回true的(与Set不同)。对于下面的代码来说，速度会快得多。

```
Parent p = (Parent) sess.load(Parent.class, id);
Child c = new Child();
c.setParent(p);
p.getChildren().add(c); //no need to fetch the collection
sess.flush();
```

## 13.4. 一次性删除(One shot delete)

有时候，一个一个的删除集合类中的元素是极度低效的。Hibernate没那么笨，如果你想要把整个集合都删除（比如说调用`list.clear()`），Hibernate只需要一个`DELETE`就搞定了。

假设我们在一个长度为20的集合类中新增加了一个元素，然后删除了两个。Hibernate会安排一个`INSERT`语句和两条`DELETE`语句（除非集合类是一个`bag`）。这当然是可以想见的。

但是，如果假设我们删除了18个元素，只剩下2个，然后新增3个。有两种处理方式：

- 把这18个元素一个一个的干掉，再新增三个
- 把整个集合类都咔嚓掉（只用一句`DELETE`语句），然后增加5个元素。

Hibernate还没那么聪明，知道第二种选择可能会比较快。（也许让Hibernate不要这么聪明也是好事，否则可能会引发意外的数据库触发器什么的。）

幸运的是，你可以强制使用第二种策略。你需要把原来的整个集合类都取消（取消其引用），然后返回一个新实例化的集合类，只包含需要的元素。有些时候这是非常有用的。

## Chapter 14. 条件查询(Criteria Query)

现在Hibernate也支持一种直观的、可扩展的条件查询API。目前为止，这个API还没有更成熟的HQL查询那么强大，也没有那么多查询能力。特别要指出，条件查询也不支持投影（projection）或统计函数（aggregation）。

## 14.1. 创建一个Criteria实例

`net.sf.hibernate.Criteria`这个接口代表对一个特定的持久化类的查询。`Session`是用来制造`Criteria`实例的工厂。

```
Criteria crit = sess.createCriteria(Cat.class);
crit.setMaxResults(50);
List cats = crit.list();
```

## 14.2. 缩小结果集范围

一个查询条件(Criterion)是net.sf.hibernate.expression.Criterion接口的一个实例。类net.sf.hibernate.expression.Expression定义了获得一些内置的Criterion类型。

```
List cats = sess.createCriteria(Cat.class)
    .add( Expression.like("name", "Fritz%") )
    .add( Expression.between("weight", minWeight, maxWeight) )
    .list();
```

表达式 ( Expressions ) 可以按照逻辑分组。

```
List cats = sess.createCriteria(Cat.class)
    .add( Expression.like("name", "Fritz%") )
    .add( Expression.or(
        Expression.eq( "age", new Integer(0) ),
        Expression.isNull("age")
    ) )
    .list();
```

```
List cats = sess.createCriteria(Cat.class)
    .add( Expression.in( "name", new String[] { "Fritz", "Fritz" } ) )
    .add( Expression.disjunction(
        .add( Expression.isNull("age") )
        .add( Expression.eq("age", new Integer(0) ) )
        .add( Expression.eq("age", new Integer(1) ) )
        .add( Expression.eq("age", new Integer(2) ) )
    ) )
    .list();
```

有很多预制的条件类型 ( Expression的子类 )。有一个特别有用，可以让你直接嵌入SQL。

```
List cats = sess.createCriteria(Cat.class)
    .add( Expression.sql("lower($alias.name) like lower('Fritz%')") )
    .list();
```

```
.list();
```

其中的{alias}是一个占位符，它将会被所查询实体的行别名所替代。(原文:The {alias} placeholder will be replaced by the row alias of the queried entity.)

## 14.3. 对结果排序

可以使用`net.sf.hibernate.expression.Order`对结果集排序.

```
List cats = sess.createCriteria(Cat.class)
    .add( Expression.like("name", "F%")
    .addOrder( Order.asc("name") )
    .addOrder( Order.desc("age") )
    .setMaxResults(50)
    .list();
```

## 14.4. 关联 ( Associations )

你可以在关联之间使用`createCriteria()`，很容易地在存在关系的实体之间指定约束。

```
List cats = sess.createCriteria(Cat.class)
    .add( Expression.like("name", "F%")
    .createCriteria("kittens")
        .add( Expression.like("name", "F%")
    .list();
```

注意，第二个`createCriteria()`返回一个`Criteria`的新实例，指向`kittens`集合类的元素。

下面的替代形式在特定情况下有用。

```
List cats = sess.createCriteria(Cat.class)
    .createAlias("kittens", "kt")
    .createAlias("mate", "mt")
    .add( Expression.eqProperty("kt.name", "mt.name")
    .list();
```

( `createAlias()` ) 并不会创建一个`Criteria`的新实例。 )

请注意，前面两个查询中`Cat`实例所持有的`kittens`集合类并没有通过`criteria`预先过滤！如果你希望只返回满足条件的`kittens`，你必须使用`returnMaps()`。

```
List cats = sess.createCriteria(Cat.class)
    .createCriteria("kittens", "kt")
        .add( Expression.eq("name", "F%") )
    .returnMaps()
    .list();
Iterator iter = cats.iterator();
while ( iter.hasNext() ) {
    Map map = (Map) iter.next();
    Cat cat = (Cat) map.get(Criteria.ROOT_ALIAS);
    Cat kitten = (Cat) map.get("kt");
```

}

## 14.5. 动态关联对象获取 ( Dynamic association fetching )

可以在运行时通过`setFetchMode()`来改变关联对象自动获取的策略。

```
List cats = sess.createCriteria(Cat.class)
    .add( Expression.like("name", "Fritz%") )
    .setFetchMode("mate", FetchMode.EAGER)
    .list();
```

这个查询会通过外连接(outer join)同时获得 `mate`和`kittens`。

## 14.6. 根据示例查询 ( Example queries )

`net.sf.hibernate.expression.Example`类允许你从指定的实例创造查询条件。

```
Cat cat = new Cat();
cat.setSex('F');
cat.setColor(Color.BLACK);
List results = session.createCriteria(Cat.class)
    .add( Example.create(cat) )
    .list();
```

版本属性，表示符属性和关联都会被忽略。默认情况下，null值的属性也被排除在外。

You can adjust how the `Example` is applied. 你可以调整示例(`Example`)如何应用。

```
Example example = Example.create(cat)
    .excludeZeroes()           //exclude zero valued p
    .excludeProperty("color") //exclude the property
    .ignoreCase()              //perform case insensit
    .enableLike();             //use like for string c
List results = session.createCriteria(Cat.class)
    .add(example)
    .list();
```

你甚至可以用示例对关联对象建立criteria。

```
List results = session.createCriteria(Cat.class)
    .add( Example.create(cat) )
    .createCriteria("mate")
        .add( Example.create( cat.getMate() ) )
    .list();
```

## Chapter 15. SQL查询

你也可以直接使用你的数据库方言表达查询。在你想使用数据库的某些特性的时候，这是非常有用的，比如Oracle中的CONNECT关键字。这也会扫清你把原来直接使用SQL/JDBC 的程序移植到Hibernate道路上的障碍。

## 15.1. 创建一个基于SQL的Query

和普通的HQL查询一样，SQL查询同样是从Query接口开始的。唯一的区别是使用`session.createSQLQuery()`方法。

```
Query sqlQuery = sess.createSQLQuery("select {cat.*} f  
sqlQuery.setMaxResults(50);  
List cats = sqlQuery.list());
```

传递给`createSQLQuery()`的三个参数是：

- SQL查询语句
- 表的别名
- 查询返回的持久化类

别名是为了在SQL语句中引用对应的类（本例中是`Cat`）的属性的。你也可以传递一个别名的`String`数组和一个对应的`Class`的数组进去，每行就可以得到多个对象。

## 15.2. 别名和属性引用

上面使用的`{cat.*}`标记是“所有属性的”的简写。你可以显式的列出需要的属性，但是你必须让Hibernate为每个属性提供SQL列别名。这些列的的占位表示符是以表别名为前导，再加上属性名。下面的例子中，我们从一个其它的表(`cat_log`) 中获取`Cat`对象，而非`Cat`对象原本在映射元数据中声明的表。注意你在`where`子句中也可以使用 属性别名。

```
String sql = "select cat.originalId as {cat.id}, cat.mate as {cat.mate}"
            + " from cat_log cat where {cat.mate} = :catId"
List loggedCats = sess.createSQLQuery(sql, "cat", Cat.class)
                    .setLong("catId", catId)
                    .list();
```

注意：如果你明确的列出了每个属性，你必须包含这个类和它的子类的属性! //??

## 15.3. 为SQL查询命名

可以在映射文档中定义SQL查询的名字，然后就可以像调用一个命名HQL查询一样直接调用命名SQL查询。

```
List people = sess.getNamedQuery("mySqlQuery")
    .setMaxResults(50)
    .list();
```

```
<sql-query name="mySqlQuery">
  <return alias="person" class="eg.Person"/>
  SELECT {person}.NAME AS {person.name}, {person}.AGE
  FROM PERSON {person} WHERE {person}.NAME LIKE 'Hib
</sql-query>
```

## Chapter 16. 继承映射(Inheritance Mappings)

## 16.1. 三种策略

Hibernate支持三种不同的基本继承映射策略。

- 每棵类继承树使用一个表(table per class hierarchy)
- 每个子类一个表(table per subclass)
- 每个具体类一个表(table per concrete class) (有一些限制)

甚至在一棵继承关系书中对不同的分支使用不同的映射策略也是可能的。但是和“每个具体类一个表”的映射有一样的限制。Hibernate不支持把<subclass>映射与<joined-subclass>在同一个<class>元素中混合使用。

假设我们有一个Payment接口，有不同的实现：CreditCardPayment, CashPayment, ChequePayment。“继承数共享一个表”的映射是这样的：

```
<class name="Payment" table="PAYMENT">
  <id name="id" type="long" column="PAYMENT_ID">
    <generator class="native"/>
  </id>
  <discriminator column="PAYMENT_TYPE" type="string"
  <property name="amount" column="AMOUNT"/>
  ...
  <subclass name="CreditCardPayment" discriminator-value="credit-card"
    ...
  </subclass>
  <subclass name="CashPayment" discriminator-value="cash"
    ...
  </subclass>
  <subclass name="ChequePayment" discriminator-value="cheque"
    ...
  </subclass>
</class>
```

只需要一个表。这种映射策略由一个大限制：子类定义的字段不能有NOT NULL限制。

“每个子类一个表”的映射是这样的：

```
<class name="Payment" table="PAYMENT">
  <id name="id" type="long" column="PAYMENT_ID">
    <generator class="native"/>
  </id>
  <property name="amount" column="AMOUNT"/>
  ...
  <joined-subclass name="CreditCardPayment" table="C
    <key column="PAYMENT_ID"/>
    ...
  </subclass>
  <joined-subclass name="CashPayment" table="CASH_PA
    <key column="PAYMENT_ID"/>
    ...
  </subclass>
  <joined-subclass name="ChequePayment" table="CHEQU
    <key column="PAYMENT_ID"/>
    ...
  </subclass>
</class>
```

需要四个表。三个子类表通过主键和超类表关联（所以实际上关系模型是一对一关联）。

注意Hibernate的“每子类一表”的实现并不需要一个特别的辨认字段。其他的对象/关系数据库映射工具使用另一种“每子类一表”实现，需要在超类表中有一个类型辨认字段。Hibernate的这种实现更加困难，但是从关系(数据库)的角度来看，这样做更加正确。

对这两种映射策略来说，指向Payment的关联是使用<many-to-one>进行映射的。

```
<many-to-one name="payment"
  column="PAYMENT"
  class="Payment"/>
```

“每个具体类一个表”的策略非常不同

```

<class name="CreditCardPayment" table="CREDIT_PAYMENT"
  <id name="id" type="long" column="CREDIT_PAYMENT_ID"
    <generator class="native"/>
  </id>
  <property name="amount" column="CREDIT_AMOUNT"/>
  ...
</class>

<class name="CashPayment" table="CASH_PAYMENT">
  <id name="id" type="long" column="CASH_PAYMENT_ID"
    <generator class="native"/>
  </id>
  <property name="amount" column="CASH_AMOUNT"/>
  ...
</class>

<class name="ChequePayment" table="CHEQUE_PAYMENT">
  <id name="id" type="long" column="CHEQUE_PAYMENT_ID"
    <generator class="native"/>
  </id>
  <property name="amount" column="CHEQUE_AMOUNT"/>
  ...
</class>

```

需要三个表。注意我们没有明确的定义Payment接口。我们用Hibernate的隐含多态（*implicit polymorphism*）机制代替。也要注意Payment的属性在三个子类中都进行了映射。

这种情形下，与Payment关联的多态关联被映射为<any>。

```

<any name="payment"
  meta-type="class"
  id-type="long">
  <column name="PAYMENT_CLASS"/>
  <column name="PAYMENT_ID"/>
</any>

```

如果我们定义UserType和meta-type来根据不同的标识字符串映射

到Payment，事情会更好一些。

```
<any name="payment"
      meta-type="PaymentMetaType"
      id-type="long">
  <column name="PAYMENT_TYPE"/> <!-- CREDIT, CASH or
  <column name="PAYMENT_ID"/>
</any>
```

对这个映射还有一点需要注意。因为每个子类都在各自独立的<class>元素中映射（并且Payment只是个接口），每个子类都可以和容易的成为另一个“每个类一个表”或者“每个子类一个表”的继承树！（并且你仍然可以对Payment接口使用多态查询。）

```
<class name="CreditCardPayment" table="CREDIT_PAYMENT"
  <id name="id" type="long" column="CREDIT_PAYMENT_I
    <generator class="native"/>
  </id>
  <discriminator column="CREDIT_CARD" type="string"/>
  <property name="amount" column="CREDIT_AMOUNT"/>
  ...
  <subclass name="MasterCardPayment" discriminator-v
  <subclass name="VisaPayment" discriminator-value="
</class>

<class name="NonelectronicTransaction" table="NONELECT
  <id name="id" type="long" column="TXN_ID">
    <generator class="native"/>
  </id>
  ...
  <joined-subclass name="CashPayment" table="CASH_PA
    <key column="PAYMENT_ID"/>
    <property name="amount" column="CASH_AMOUNT"/>
    ...
  </joined-subclass>
  <joined-subclass name="ChequePayment" table="CHEQU
    <key column="PAYMENT_ID"/>
    <property name="amount" column="CHEQUE_AMOUNT"
```

```
    ...  
    </joined-subclass>  
</class>
```

我们再一次没有明确的提到Payment。如果我们针对Payment接口执行查询——比如，`from Payment`——Hibernate自动返回CreditCardPayment实例（以及它的子类，因为它们也继承了Payment），CashPayment和Chequepayment，但是不会是NonelectronicTransaction的实例。

## 16.2. 限制

Hibernate假设关联严格的和一个外键字段相映射。如果一个外键具有多个关联，也是可以容忍的（你可能需要指定`inverse="true"`或者`insert="false" update="false"`），但是你不能为多重外键指定任何映射的关联。这意味着：

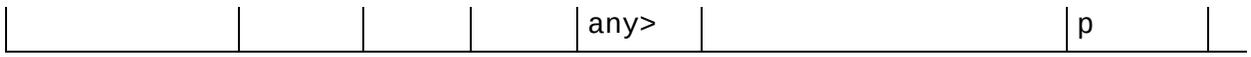
- 当更改一个关联的时候，永远是更新的同一个外键
- 当一个关联是延迟抓取(`fetch="lazy"`)的时候，只需要用一次数据库查询
- 当一个关联是提前抓取(`fetch="eager"`)的时候，使用一次`outer join`即可

特别要指出的是，使用“每个具体类一个表”的策略来实行多态的一对多关联是不支持的。（抓取这样的关联需要多次查询或者多次`join`。）

下面的表格列出了Hibernate中，“每个具体类一个表”策略与隐含多态机制的限制。

**Table 16.1. 继承映射时的隐含多态 ( Implicit polymorphism in inheritance mappings )**

继承策略 ( Inheritance strategy )	多态多对一	多态一对一	多态一对多	多态多对多	多态 load()/get()	多态查询	多态 join
每继承树一表	<many-to-one>	<one-to-one>	<one-to-many>	<many-to-many>	<code>s.get(Payment.class, id)</code>	<code>from Payment p</code>	<code>from Payment p</code>
每子类一表	<many-to-one>	<one-to-one>	<one-to-many>	<many-to-many>	<code>s.get(Payment.class, id)</code>	<code>from Payment p</code>	<code>from Payment p</code>
每类一表(隐含多态)	<any>	不支持	不支持	<many-to-	<i>use a query</i>	<code>from Payment</code>	不



## Chapter 17. 事务和并行 ( Transactions And Concurrency )

Hibernate本身并不是数据库，它只是一个轻量级的对象 - 关系数据库映射 ( object-relational ) 工具。它的事务交由底层的数据库连接管理，如果数据库连接有JTA的支持，那么在session中进行的操作将是整个原子性JTA事务的一部分。Hibernate可以看作是添加了面向对象语义的JDBC瘦适配器 ( thin adapter )。

## 17.1. 配置，会话和工厂（Configurations, Sessions and Factories）

SessionFactory的创建需要耗费大量资源，它是线程安全（threadsafe）的对象，在应用中它被所有线程共享。而Session的创建耗费资源很少，它不是线程安全的对象，对于一个简单商业过程（business process），它应该只被使用一次，然后被丢弃。举例来说，当Hibernate在基于servlet的应用中，servlet能够以下面的方式得到SessionFactory。

```
SessionFactory sf = (SessionFactory)getServletContext()
```

每次调用SessionFactory的service方法能够生成一个新的Session对象，然后调用Session的flush()，调用commit()提交它的连接，调用close()关闭它，最终丢弃它。

在无状态的session bean中，可以同样使用类似的方法。bean在setSessionContext()中得到SessionFactory的实例，每个商业方法会生成一个Session对象，调用它的flush()和close()，当然，应用不应该commit()connection. (把它留给JTA.)

这里需要理解flush()的含义。flush()将持久化存储与内存中的变化进行同步，但不是将内存的变化与持久化存储进行同步。所以在调用flush()并接着调用commit()关闭连接时，会话将仍然含有过时的数据，在这种情况下，继续使用会话的唯一的办法是将会话中的数据进行版本化。

接下来的几小节将讨论利用版本化的方法来确保事务原子性，这些“高级”方法需要小心使用。

## 17.2. 线程和连接 ( Threads and connections )

You should observe the following practices when creating Hibernate Sessions:

在创建Hibernate会话 ( Session ) 时，你应该留意以下的实践 ( practices ) ：

- 对于一个数据库连接，不要创建一个以上的Session或Transaction
- 在对于一个数据库连接、一个事务使用多个Session时，你尤其需要格外地小心。Session对象会记录下调入数据更新的情况，所以另一个Session对象可能会遇到过时的数据。
- Session不是线程安全的。如果确实需要在两个同时运行的线程中共享会话，那么你应该确保线程在访问会话时，线程对Session具有同步锁。

## 17.3. 乐观锁定 / 版本化 ( Optimistic Locking / Versioning )

许多商业过程需要一系列与用户进行交互的过程，数据库访问穿插在这些过程中。对于web和企业应用来说，跨一个用户交互过程的数据事务是不可接受的，因而维护各商业事务间的隔离 ( isolation ) 就成为应用层的部分责任。唯一满足高并发性以及高可扩展性的方法是使用带有版本化的乐观锁定。Hibernate为使用乐观锁定的代码提供了三种可能的方法。

### 17.3.1. 使用长生命周期带有自动版本化的会话

在整个商业过程中使用一个单独的Session实例以及它的持久化实例，这个Session使用带有版本化的乐观锁定机制，来确保多个数据库事务对于应用来说只是一个逻辑上的事务。在等待用户交互时，Session断开与数据库的连接。这个方法从数据库访问方面来看是最有效的，应用不需要关心对自己的版本检查或是重新与不需要序列化 ( transient ) 的实例进行关联。

```
// foo is an instance loaded earlier by the Session
session.reconnect();
foo.setProperty("bar");
session.flush();
session.connection().commit();
session.disconnect();
```

### 17.3.2. 使用带有自动版本化的多个会话

每个与持久化存储的交互出现在一个新的Session中，在每次与数据库的交互中，使用相同的持久化实例。应用操作那些从其它Session调入的不需要持久化实例的状态，通过使用Session.update()或者Session.saveOrUpdate()来重新建立与它们的关联。

```
// foo is an instance loaded by a previous Session
foo.setProperty("bar");
session = factory.openSession();
session.saveOrUpdate(foo);
```

```
session.flush();
session.connection().commit();
session.close();
```

### 17.3.3. 应用程序自己进行版本检查

每当一个新的Session中与持久化存储层出现交互的时候，这个session会在操作持久化实例前重新把它们从数据存储中装载进来。我们现在所说的方式就是你的应用程序自己使用版本检查来确保商业过程的隔绝性。（当然，Hibernate仍会为你更新版本号）。从数据库访问方面来看，这种方法是最没有效率的，与entity EJB方式类似。

```
// foo is an instance loaded by a previous Session
session = factory.openSession();
int oldVersion = foo.getVersion();
session.load( foo, foo.getKey() );
if ( oldVersion!=foo.getVersion() ) throw new StaleObjectException();
foo.setProperty("bar");
session.flush();
session.connection().commit();
session.close();
```

当然，如果在低数据并行（low-data-concurrency）的环境中，并不需要版本检查，你仍可以使用这个方法，只需要忽略版本检查。

## 17.4. 会话断开连接 ( Session disconnection )

The first approach described above is to maintain a single session for a whole business process that spans user think time. (For example, a servlet might keep a session in the user's HttpSession.) For performance reasons you should

上面提到的第一种方法是对于对一个用户的一次登录产生的整个商业过程维护一个Session。（举例来说，servlet有可能会在用户的HttpSession中保留一个Session）。为性能考虑，你必须

- 提交Transaction（或者JDBC连接），然后
- (在等待用户操作前，)断开Session与JDBC连接。

Session.disconnect()方法会断开会话与JDBC的连接，把连接返还给连接池（除非是你自己提供这个连接的）。

Session.reconnect()方法会得到一个新的连接（你也可以自己提供一个），重新开始会话。在重新连接后，你可以通过对任何可能被其它事务更新的对象调用Session.lock()方法，来强迫对你没有更新的数据进行版本检查。你不需要对正在更新的数据调用lock()。

这是一个例子：

```
SessionFactory sessions;
List fooList;
Bar bar;
....
Session s = sessions.openSession();

Transaction tx = null;
try {
    tx = s.beginTransaction();

    fooList = s.find(
        "select foo from eg.Foo foo where foo.Date = c
        // uses db2 date function
    );
```

```

        bar = (Bar) s.create(Bar.class);

        tx.commit();
    }
    catch (Exception e) {
        if (tx!=null) tx.rollback();
        s.close();
        throw e;
    }
    s.disconnect();

```

接下来：

```

s.reconnect();

try {
    tx = sessions.beginTransaction();

    bar.setFooTable( new HashMap() );
    Iterator iter = fooList.iterator();
    while ( iter.hasNext() ) {
        Foo foo = (Foo) iter.next();
        s.lock(foo, LockMode.READ);    //check that fo
        bar.getFooTable().put( foo.getName(), foo );
    }

    tx.commit();
}
catch (Exception e) {
    if (tx!=null) tx.rollback();
    throw e;
}
finally {
    s.close();
}

```

从上面的例子可以看到Transaction和Session之间是多对一的关系。一个Session表示了应用程序与持久存储之间的一个对话，Transaction把这个

对话分隔成一个个具有原子性的单元。

## 17.5. 悲观锁定 ( Pessimistic Locking )

用户不需要在锁定策略上花费过多时间，通常我们可以选定一种隔离级别 ( isolationn level ) ，然后让数据库完成所有的工作。高级用户可能希望得到悲观锁定或者在新的事务开始时重新得到锁。

LockMode类定义了Hibernate需要的不同的锁级别。锁由以下的机制得到：

- LockMode.WRITE在Hibernate更新或插入一行数据时自动得到。
- LockMode.UPGRADE在用户通过SELECT ... FOR UPDATE这样的特定请求得到，需要数据库支持这种语法。
- LockMode.UPGRADE\_NOWAIT在用户通过SELECT ... FOR UPDATE NOWAIT这样的特定请求在Oracle数据库环境下得到。
- LockMode.READ在Hibernate在不断读 ( Repeatable Read ) 和序列化 ( Serializable ) 的隔离级别下读取数据时得到。也可以通过用户的明确请求重新获得。
- LockMode.NONE表示没有锁。所有对象在Transaction结束时切换到这种锁模式，通过调用update()或者saveOrUpdate()与会话进行关联的对象,开始时也会在这种锁模式。

“明确的用户请求”会以以下几种方式出现：

- 调用Session.load()，指定一种LockMode。
- 调用Session.lock()。
- 调用Query.setLockMode()。

如果在调用Session.load()时指定了UPGRADE或者UPGRADE\_NOWAIT，并且请求的对象还没有被会话调入，那么这个对象会以SELECT ... FOR UPDATE的方式调入。如果调用load()在一个已经调入的对象，并且这个对象调入时的锁级别没有请求时来得严格，Hibernate会对这个对象调用lock()。

Session.lock()会执行版本号检查的特定的锁模式是：READ，UPGRADE或者UPGRADE\_NOWAIT。（在UPGRADE或者UPGRADE\_NOWAIT，SELECT ... FOR

UPGRADE使用的情况下。 )

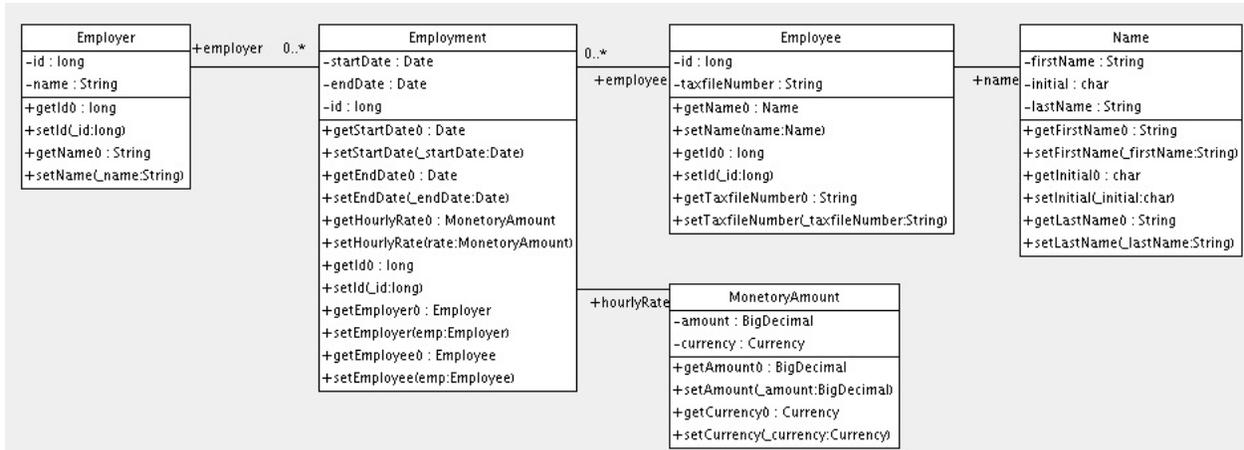
如果数据库不支持所请求的锁模式，Hibernate将会选择一种合适的受支持的锁模式替换（而不是抛出一个异常）。这确保了应用具有可移植性。

## Chapter 18. 映射实例(Mapping Examples)

本章将为你展示几个比较复杂的关联映射。

## 18.1. 雇员 / 雇主 ( Employer/Employee )

接下来关于Employer和Employee关系的模型使用了一个实体 ( entity ) 类 ( Employment ) 来表示这个关联，因为对于相同的雇主和雇员可能会有多个雇用时间段。对于雇用金额和雇员姓名，我们使用组件 ( component ) 来进行建模。



这是一个可行的映射文档：

```
<hibernate-mapping>
  <class name="Employer" table="employers">
    <id name="id">
      <generator class="sequence">
        <param name="sequence">employer_id_sec
      </generator>
    </id>
    <property name="name"/>
  </class>
  <class name="Employment" table="employment_periods">
    <id name="id">
      <generator class="sequence">
        <param name="sequence">employment_id_s
      </generator>
```

```

</id>
<property name="startDate" column="start_date"
<property name="endDate" column="end_date"/>

<component name="hourlyRate" class="MonetaryAmount"
  <property name="amount">
    <column name="hourly_rate" sql-type="NUMBER"/>
  </property>
  <property name="currency" length="12"/>
</component>

<many-to-one name="employer" column="employer_id"/>
<many-to-one name="employee" column="employee_id"/>

</class>

<class name="Employee" table="employees">
  <id name="id">
    <generator class="sequence">
      <param name="sequence">employee_id_seq</param>
    </generator>
  </id>
  <property name="taxfileNumber"/>
  <component name="name" class="Name">
    <property name="firstName"/>
    <property name="initial"/>
    <property name="lastName"/>
  </component>
</class>

</hibernate-mapping>

```

这里是由SchemaExport生成的表结构。

```

create table employers (
  id BIGINT not null,
  name VARCHAR(255),
  primary key (id)
)

```

```
)
```

```
create table employment_periods (  
    id BIGINT not null,  
    hourly_rate NUMERIC(12, 2),  
    currency VARCHAR(12),  
    employee_id BIGINT not null,  
    employer_id BIGINT not null,  
    end_date TIMESTAMP,  
    start_date TIMESTAMP,  
    primary key (id)
```

```
)
```

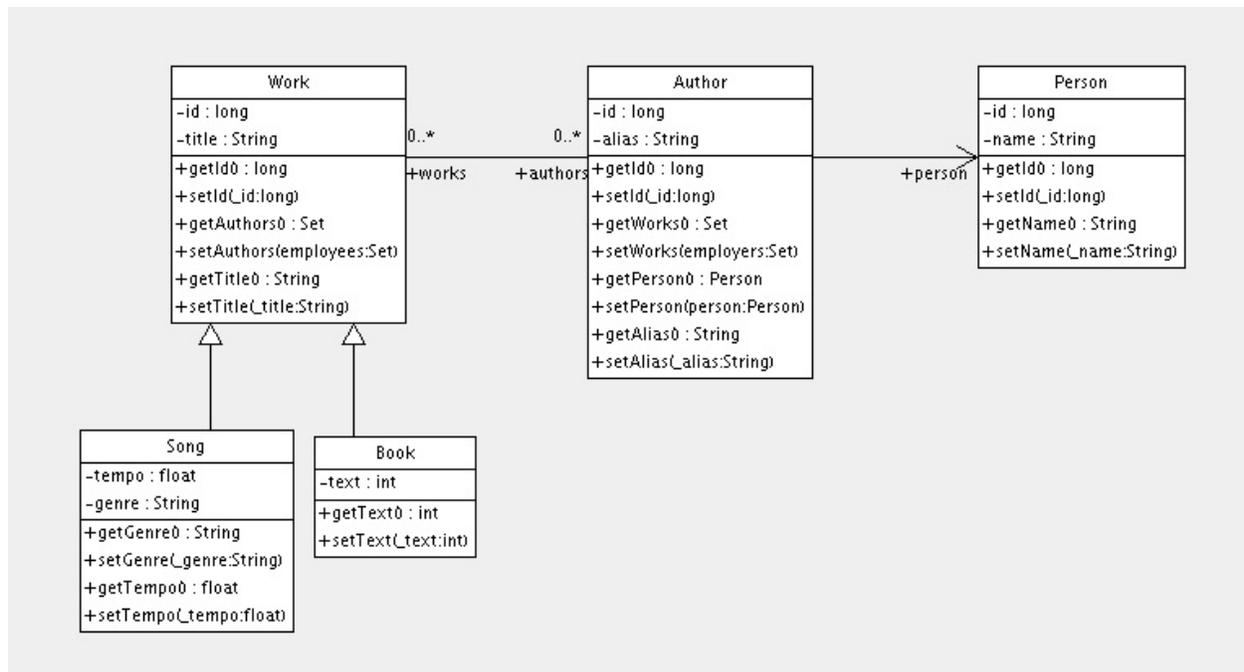
```
create table employees (  
    id BIGINT not null,  
    firstName VARCHAR(255),  
    initial CHAR(1),  
    lastName VARCHAR(255),  
    taxfileNumber VARCHAR(255),  
    primary key (id)
```

```
)
```

```
alter table employment_periods  
    add constraint employment_periodsFK0 foreign key (  
alter table employment_periods  
    add constraint employment_periodsFK1 foreign key (  
create sequence employee_id_seq  
create sequence employment_id_seq  
create sequence employer_id_seq
```

## 18.2. 作者 / 著作(Author/Work)

下面的例子是关于Work、Author和Person。我们用多对多关系来表示Work和Author之间的关联，用一对一的关系来表示Author和Person之间的关联。另外一种可行的方式是对Author扩展Person。



接下来的映射文档正确地表示这些关系：

```
<hibernate-mapping>
  <class name="Work" table="works" discriminator-val
    <id name="id" column="id">
      <generator class="native"/>
    </id>
    <discriminator column="type" type="character"/>
    <property name="title"/>
    <set name="authors" table="author_work" lazy="
      <key>
        <column name="work_id" not-null="true"
```

```

        </key>
        <many-to-many class="Author">
            <column name="author_id" not-null="true">
        </many-to-many>
    </set>

    <subclass name="Book" discriminator-value="B">
        <property name="text"/>
    </subclass>

    <subclass name="Song" discriminator-value="S">
        <property name="tempo"/>
        <property name="genre"/>
    </subclass>

</class>

<class name="Author" table="authors">

    <id name="id" column="id">
        <!-- The Author must have the same identifier as the work it belongs to -->
        <generator class="assigned"/>
    </id>

    <property name="alias"/>
    <one-to-one name="person" constrained="true"/>

    <set name="works" table="author_work" inverse="true">
        <key column="author_id"/>
        <many-to-many class="Work" column="work_id"/>
    </set>

</class>

<class name="Person" table="persons">
    <id name="id" column="id">
        <generator class="native"/>
    </id>
    <property name="name"/>

```

```
</class>
```

```
</hibernate-mapping>
```

在这个映射中有四个表。works, authors和persons分别存放著作、作者以及人的数据。author\_work是关联表,把作者与著作关联起来。以下是由SchemaExport生成的表结构。

```
create table works (
  id BIGINT not null generated by default as identity,
  tempo FLOAT,
  genre VARCHAR(255),
  text INTEGER,
  title VARCHAR(255),
  type CHAR(1) not null,
  primary key (id)
)

create table author_work (
  author_id BIGINT not null,
  work_id BIGINT not null,
  primary key (work_id, author_id)
)

create table authors (
  id BIGINT not null generated by default as identity,
  alias VARCHAR(255),
  primary key (id)
)

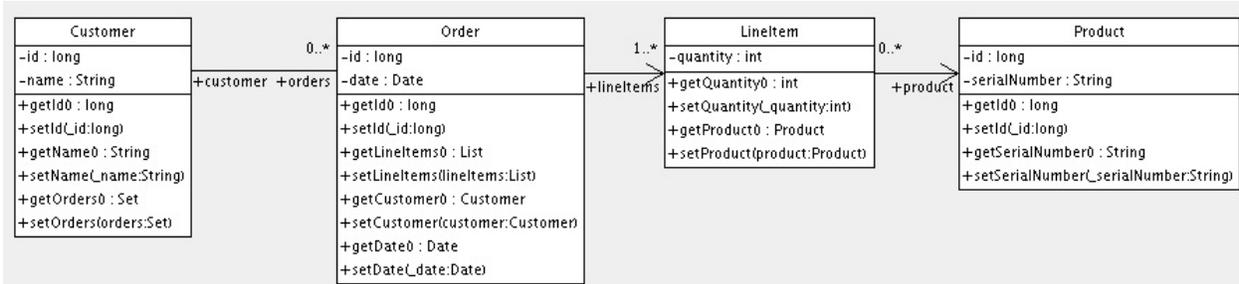
create table persons (
  id BIGINT not null generated by default as identity,
  name VARCHAR(255),
  primary key (id)
)

alter table authors
  add constraint authorsFK0 foreign key (id) referer
```

```
alter table author_work
  add constraint author_workFK0 foreign key (author_
alter table author_work
  add constraint author_workFK1 foreign key (work_id
```

## 18.3. 客户 / 订单 / 产品(Customer/Order/Product)

接下来的例子是关于Customer、Order、LineItem和Product。Customer和Order之间是一对多的关联。那么Order / LineItem / Product之间的关联怎么表示呢？我们可以把LineItem作为关联表来表示Order和Product之间多对多关联，在Hibernate里，它被称为组合元素（composite element）。



映射文档：

```
<hibernate-mapping>

  <class name="Customer" table="customers">
    <id name="id">
      <generator class="native"/>
    </id>
    <property name="name"/>
    <set name="orders" inverse="true" lazy="true">
      <key column="customer_id"/>
      <one-to-many class="Order"/>
    </set>
  </class>

  <class name="Order" table="orders">
    <id name="id">
      <generator class="native"/>
    </id>
    <property name="date"/>
    <many-to-one name="customer" column="customer_id"/>
    <list name="lineItems" table="line_items" lazy="true">
      <key column="order_id"/>
    </list>
  </class>

  <class name="LineItem" table="line_items">
    <id name="id">
      <generator class="native"/>
    </id>
    <property name="quantity"/>
    <property name="product"/>
  </class>

  <class name="Product" table="products">
    <id name="id">
      <generator class="native"/>
    </id>
    <property name="serialNumber"/>
  </class>

  <many-to-many name="Order-Product" class="LineItem" column="order_id"
    column2="product_id"/>

</hibernate-mapping>
```

```

        <index column="line_number"/>
        <composite-element class="LineItem">
            <property name="quantity"/>
            <many-to-one name="product" column="pr
        </composite-element>
    </list>
</class>

<class name="Product" table="products">
    <id name="id">
        <generator class="native"/>
    </id>
    <property name="serialNumber"/>
</class>

</hibernate-mapping>

```

customers、orders、line\_items和products分别存放客户、订单、订单项以及产品的数据。line\_items作为关联表，把订单和产品关联起来。

```

create table customers (
    id BIGINT not null generated by default as identit
    name VARCHAR(255),
    primary key (id)
)

create table orders (
    id BIGINT not null generated by default as identit
    customer_id BIGINT,
    date TIMESTAMP,
    primary key (id)
)

create table line_items (
    line_number INTEGER not null,
    order_id BIGINT not null,
    product_id BIGINT,
    quantity INTEGER,

```

```
        primary key (order_id, line_number)
    )
create table products (
    id BIGINT not null generated by default as identity,
    serialNumber VARCHAR(255),
    primary key (id)
)
alter table orders
    add constraint ordersFK0 foreign key (customer_id)
alter table line_items
    add constraint line_itemsFK0 foreign key (product_id)
alter table line_items
    add constraint line_itemsFK1 foreign key (order_id)
```

## Chapter 19. 工具箱指南

通过Hibernate项目中提供的几个命令行工具(他们也被当作项目的一部分不断得到维护),还有XDoclet,Middlegen和AndroMDA内置的对Hibernate的支持,可以在几个不同的环境(SQL,java代码,xml映射文件)中进行相互转换(roundtrip)。

Hibernate的主发行包中附带了最重要的工具(甚至在Hibernate内部也可以快速调用这个工具)：

- 从映射文件到DDL schema的生成器(也就是SchemaExport和hbm2ddl)

Hibernate项目直接提供的其他工具在一个单独的发行包中发布, *Hibernate Extensions*。这个发行包包含了下列任务的工具：

- 从映射文件到Java源代码的生成器(也就是CodeGenerator, hbm2java)
- 从已编译的Java类或者带有XDoclet标记的Java源代码生成映射文件(它们是MapGenerator,class2hbm)

实际上Hibernate Extensions里面还有一个工具：dd12hbm。但是它已经被废弃了,已经不再被维护了。Middlegen完成了同样的任务,并且更加出色。

对Hibernate提供支持的第三方工具有：

- Middlegen (从现有的数据库schema中生成映射文件)
- AndroMDA (使用MDA思想(Model-Driven Architecture, 模型驱动体系)的代码生成器,它从UML图和其XML/XMI等价形式中生成持久化类的代码)

这些第三方工具没有在这篇指南中说明。请查阅Hibernate网站得到关于它们目前的情况。(Hibernate主发行包中有关于整个网站的快照)

## 19.1. Schema 生成器 ( Schema Generation )

可以从你的映射文件使用一个命令行工具生成DDL。在Hibernate主发行包的hibernate-x.x.x/bin目录下有一个批处理文件。

生成的schema包含有对实体和集合类表的完整性引用约束（主键和外键）。涉及到的标示符生成器所需的表和sequence也会同时生成。

在使用这个工具的时候，你必须通过hibernate.dialect属性指定一个SQL方言(Dialect)。

### 19.1.1. 对schema定制化(Customizing the schema)

很多Hibernate映射元素定义了一个可选的length属性。你可以通过这个属性设置字段的长度。

有些tag接受not-null属性（用来在表字段上生成NOT NULL约束）和unique属性（用来在表字段上生成UNIQUE约束）。

有些tag接受index属性，用来指定字段的index名字。unique-key属性可以对成组的字段指定一个组合键约束(unit key constraint)。目前，unique-key属性指定的值并不会被当作这个约束的名字，它们只是在用来在映射文件内部用作区分的。

示例：

```
<property name="foo" type="string" length="64" not-null="true" />
<many-to-one name="bar" foreign-key="fk_foo_bar" not-null="true" />
<element column="serial_number" type="long" not-null="true" />
```

另外，这些元素还接受<column>子元素。在定义跨越多字段的类型时特别有用。

```
<property name="foo" type="string">
  <column name="foo" length="64" not-null="true" sql-type="VARCHAR" />
</property>
```

```

</property>

<property name="bar" type="my.customtypes.MultiColumnT
  <column name="fee" not-null="true" index="bar_idx"
  <column name="fi" not-null="true" index="bar_idx"/
  <column name="fo" not-null="true" index="bar_idx"/
</property>

```

sql-type属性允许用户覆盖默认的Hibernate类型到SQL数据类型的映射。

check属性允许用户指定一个约束检查。

```

<property name="foo" type="integer">
  <column name="foo" check="foo > 10"/>
</property>

<class name="Foo" table="foos" check="bar < 100.0">
  ...
  <property name="bar" type="float"/>
</class>

```

**Table 19.1. Summary**

属性 (Attribute)	值 ( Values )	解释 ( Interpretation )
length	true false	字段长度
not-null	true false	指明字段是否应该是非空的
unique	true false	指明是否该字段具有惟一约束
index	index_name	指明一个 ( 多字段 ) 的索引(index)的名字
unique-key	unique_key_name	指明多字段惟一约束的名字 ( 参见上面的说明 )
foreign-key	foreign_key_name	指明一个外键的名字, 它是为关联生成的。
sql-type	column_type	覆盖默认的字类型(只能用于<column>属性)

check	SQL 表达式	对字段或表加入SQL约束检查
-------	---------	----------------

### 19.1.2. 运行该工具

SchemaExport工具把DDL脚本写到标准输出，同时/或者执行DDL语句。

```
java -cp hibernate_classpaths net.sf.hibernate.tool.hbm2ddl.SchemaExport
options mapping_files
```

**Table 19.2. SchemaExport命令行选项**

选项	说明
--quiet	不要把脚本输出到stdout
--drop	只进行drop tables的步骤
--text	不执行在数据库中运行的步骤
--output=my_schema.ddl	把输出的ddl脚本输出到一个文件
--config=hibernate.cfg.xml	从XML文件读入Hibernate配置
--properties=hibernate.properties	从文件读入数据库属性
--format	把脚本中的SQL语句对齐和美化
--delimiter=x	为脚本设置行结束符

你甚至可以在你的应用程序中嵌入SchemaExport工具:

```
Configuration cfg = .....;
new SchemaExport(cfg).create(false, true);
```

### 19.1.3. 属性(Properties)

可以通过如下方式指定数据库属性:

- 通过-D<property>系统参数
- 在hibernate.properties文件中

- 位于一个其它名字的properties文件中,然后用 --properties参数指定

所需的参数包括:

**Table 19.3. SchemaExport 连接属性**

属性名	说明
hibernate.connection.driver_class	jdbc driver class
hibernate.connection.url	jdbc url
hibernate.connection.username	database user
hibernate.connection.password	user password
hibernate.dialect	方言(dialect)

#### 19.1.4. 使用Ant(Using Ant)

你可以在你的Ant build脚本中调用SchemaExport:

```
<target name="schemaexport">
  <taskdef name="schemaexport"
    classname="net.sf.hibernate.tool.hbm2ddl.SchemaExport"
    classpathref="class.path"/>

  <schemaexport
    properties="hibernate.properties"
    quiet="no"
    text="no"
    drop="no"
    delimiter=";"
    output="schema-export.sql">
    <fileset dir="src">
      <include name="**/*.hbm.xml"/>
    </fileset>
  </schemaexport>
</target>
```

## 19.1.5. 对schema的增量更新(Incremental schema updates)

SchemaUpdate工具对已存在的schema采用"增量"方式进行更新。注意SchemaUpdate严重依赖于JDBC metadata API,所以它并非对所有JDBC驱动都有效。

```
java -cp hibernate_classpaths net.sf.hibernate.tool.hbm2ddl.SchemaUpdate
options mapping_files
```

**Table 19.4.** SchemaUpdate命令行选项

选项	说明
--quiet	不要把脚本输出到stdout
--properties=hibernate.properties	从指定文件读入数据库属性

你可以在你的应用程序中嵌入SchemaUpdate工具:

```
Configuration cfg = ....;
new SchemaUpdate(cfg).execute(false);
```

## 19.1.6. 用Ant来增量更新schema(Using Ant for incremental schema updates)

你可以在Ant脚本中调用SchemaUpdate :

```
<target name="schemaupdate">
  <taskdef name="schemaupdate"
    classname="net.sf.hibernate.tool.hbm2ddl.SchemaUpdate"
    classpathref="class.path"/>

  <schemaupdate
    properties="hibernate.properties"
    quiet="no">
    <fileset dir="src">
      <include name="**/*.hbm.xml"/>
    </fileset>
  </schemaupdate>
</target>
```

```
</schemaupdate>  
</target>
```

## 19.2. 代码生成 ( Code Generation )

Hibernate代码生成器可以用来为Hibernate映射文件生成Java实现类的骨架。这个工具在Hibernate Extensions发行包中提供 ( 需要单独下载 )。

hbm2java解析映射文件，生成可工作的Java源代码文件。使用hbm2java，你可以“只”提供.hbm文件，不用担心要去手工编写Java文件。

```
java -cp hibernate_classpaths net.sf.hibernate.tool.hbm2java.CodeGenerator  
options mapping_files
```

**Table 19.5.** 代码生成器命令行选项

选项	说明
--output= <i>output_dir</i>	生成代码输出的根目录
--config= <i>config_file</i>	可选的hvm2java配置文件

### 19.2.1. 配置文件(可选)

配置文件提供了配置生成源代码的多个“渲染器(renderers)”的途径,也可以声明在全局范围生效的<meta>属性。详情请参见<meta>属性的部分。

```
<codegen>  
  <meta attribute="implements">codegen.test.IAuditab  
  <generate renderer="net.sf.hibernate.tool.hbm2java  
  <generate  
    package="autofinders.only"  
    suffix="Finder"  
    renderer="net.sf.hibernate.tool.hbm2java.Finde  
</codegen>
```

这个配置文件声明了一个全局的meta(元)属性“implements”，指定了两个渲染器，默认渲染器(BadicRender)和生成Finder ( 参见下面的“基本Finder生成器” ) 的渲染器。

定义第二个渲染器需要一个包名和后缀属性。

包名属性指定生成后的源代码应该保存的位置，覆盖在.hbm文件中指定的包范围。

后缀属性指定生成的文件的后缀。比如说，如果有一个Foo.java文件，应该变成FooFinder.java。

### 19.2.2. meta属性

<meta>标签时对hbm.xml文件进行的简单注解，工具可以用这个位置来保存/阅读和Hibernate内核不是直接相关的一些信息。

你可以用<meta>标签来告诉hbm2java只生成"protected"

下面的例子：

```
<class name="Person">
  <meta attribute="class-description">
    Javadoc for the Person class
    @author Frodo
  </meta>
  <meta attribute="implements">IAuditable</meta>
  <id name="id" type="long">
    <meta attribute="scope-set">protected</meta>
    <generator class="increment"/>
  </id>
  <property name="name" type="string">
    <meta attribute="field-description">The name c
  </property>
</class>
```

会生成类似下面的输出（为了有助于理解，节选部分代码）。注意Javadoc注释和声明成protected的set方法：

```
// default package

import java.io.Serializable;
import org.apache.commons.lang.builder.EqualsBuilder;
import org.apache.commons.lang.builder.HashCodeBuilder
```

```
import org.apache.commons.lang.builder.ToStringBuilder

/**
 *      Javadoc for the Person class
 *      @author Frodo
 */
public class Person implements Serializable, IAuditabl

    /** identifier field */
    public Long id;

    /** nullable persistent field */
    public String name;

    /** full constructor */
    public Person(java.lang.String name) {
        this.name = name;
    }

    /** default constructor */
    public Person() {
    }

    public java.lang.Long getId() {
        return this.id;
    }

    protected void setId(java.lang.Long id) {
        this.id = id;
    }

    /**
     * The name of the person
     */
    public java.lang.String getName() {
        return this.name;
    }
}
```

```

    public void setName(java.lang.String name) {
        this.name = name;
    }
}

```

**Table 19.6.** 支持的meta标签

属性	说明
class-description	插入到类的javadoc说明去
field-description	插入到field/property的javadoc说明去
interface	如果是true,生成interface而非class
implements	类要实现的接口
extends	类要继承的超类(若是subclass,则忽略该属性)
generated-class	重新指定要生成的类名
scope-class	class的scope
scope-set	set方法的scope
scope-get	get方法的scope
scope-field	实际属性字段(field)的scope
use-in-tostring	在toString()中包含此属性
bound	为属性增加propertyChangeListener支持
constrained	为属性增加vetoChangeListener支持
gen-property	如果是false,不会生成属性(谨慎使用)
property-type	覆盖属性的默认值.如果值是标签,则指定一个具体的类型而非Object(Use this with any tag's to specify the concrete type instead of just Object.)
finder-method	参见下面的"Basic finder generator"
session-	

method      参见下面的"Basic finder generator"

通过<meta>标签定义的属性在一个hbm.xml文件中是默认"继承"的。

这究竟是什么意思？如果你希望你所有的类都实现IAuditable接口，那么你只需要加一个<meta attribute="implements">IAuditable</meta> 在你hbm.xml文件的开头，就在<hibernate-mapping>后面。现在所有在hbm.xml文件中定义的类都会实现IAuditable了！（除了那些也特别指定了"implements"元属性的类，因为本地指定的元标签总是会覆盖任何继承的元标签）。

注意，这条规则对所有的<meta>标签都有效。也就是说它可以用来指定所有的字段都被声明成protected的，而非默认的private。这可以通过在<class>后面<meta attribute="scope-field">protected</meta>指定，那么这个类所有的field都会变成protected。

如果你不想让<meta>标签继承，你可以简单的在标签属性上指明inherit="false"，比如<meta attribute="scope-class" inherit="false">public abstract</meta>，这样"class-scope"就只会对当前类起作用，不会对其子类生效。

### 19.2.3. 基本的finder生成器（Basic finder generator）

目前可以让hbm2java为Hibernate属性生成基本的finder。这需要在hbm.xml文件中做两件事情。

首先是要标记出你希望生成finder的字段。你可以通过在property标签中的meta 块来定义：

```
<property name="name" column="name" type="string">
  <meta attribute="finder-method">findByName</meta>
</property>
```

find方法的名字就是meta标签中间的文字。

第二件事是为hbm2java建立下面格式的配置文件的：

```
<codegen>
```

```
<generate renderer="net.sf.hibernate.tool.hbm2java"
  <generate suffix="Finder" renderer="net.sf.hibernate"
</codegen>
```

然后用参数去调用：`hbm2java --config=xxx.xml`，`xxx.xml`就是你刚才创建的配置文件的名字。

有个可选的参数,作为一个在class级别的meta标签，格式如下：

```
<meta attribute="session-method">
  com.whatever.SessionTable.getSessionTable().getSes
</meta>
```

他是用来管理你如何使用*Thread Local Session*模式(在Hibernate 网站的 Design Patterns部分有文档)得到session的。

#### 19.2.4. 基于Velocity的渲染器/生成器(Velocity based renderer/generator)

目前可以使用velocity作为渲染机制的一个替代方案。下面的config.xml文件显示了如果配置hbm2java来使用velocity渲染器。

```
<codegen>
  <generate renderer="net.sf.hibernate.tool.hbm2jav
    <param name="template">pojo.vm</param>
  </generate>
</codegen>
```

名为template的参数是指向你希望你使用velocity macro文件的资源路径。这个文件必须在hbm2java的classpath中。所以要记住把pojo.vm所在的路径加入到你ant任务或者shell脚本中去。(默认的位置是./tools/src/velocity)

注意，当前的pojo.vm只生成java beans最基本的部分。他还没有默认的渲染器那么完整，也没有那么多功能——特别是大部分meta标签还不支持。

## 19.3. 映射文件生成器 ( Mapping File Generation )

映射文件的骨架可以从编译过的持久化类中使用MapGenerator工具生成。这工具是Hibernate Extensions发行包的一部分。

Hibernate映射生成器提供了从编译过的类中产生映射的机制。他使用Java反射来查找属性(*properties*)，然后使用启发式算法来从属性类型猜测合适的映射。生成出来的映射文件之应该看作是后续工作的起点。没有办法在没有用户修正的情况下生成完整的Hibernate映射。但是，这个工具还是替你做了很多非常琐碎和麻烦的工作。

类一个一个地加入到映射去。如果工具认为某个类不是*Hibernate*可持久化 (*persistable*) 的，就会把这些类剔除。

判断是否是*Hibernate*可持久化 (*persistable*) 的原则是：

- 必定不是一个原始类型
- 必定不是一个数组
- 必定不是一个接口
- 必定不是一个内部类
- 必定有一个默认的无参数的构造方法。

注意，接口和内部类实际上是可以通过Hibernate持久化的，但是一般来说用户不会使用。

对已经发现的类，MapGenerator会重复回溯到超类链条上去，以尽可能的把Hibernate可持久化的超类加入到对同一个数据库表的映射去。如果回溯过程中某个类出现了有个属性在下列备选UID名字 (*candidate UID names*) 名单中，回溯就会停止。

默认的备选UID属性名有：uid, UID, id, ID, key, KEY, pk, PK。

如果类中有两个方法，一个是setter,一个是getter，并且setter的单参数的属性和getter的无参数返回值得类型相同，并且setter返回void,就认为发现了一个属性。并且，setter的名字必须以set字符串开始，getter的名字必须以get

开始，或者以is开始并且属性类型是boolean。在上面的情况发生时，get和set之后的名字还必须匹配。这个匹配就是属性的名字，然后如果第二个字母是小写的话，会把其首字母变成小写。

用来决定每个属性的数据库类型的规则如下：

- 如果Java类型是Hibernate.basic()，则属性是该类型的一个普通字段。
- 对于hibernate.type.Type特定类型和PersistentEnum来说，也会使用一个普通字段。
- 如果属性类型是一个数组，那么会使用一个Hibernate数组，并且MapGenerator试图反映数组元素的类型。（attempts to reflect on the array element type.）
- 如果属性是java.util.List,java.util.Map或者java.util.Set，会使用对应的Hibernate类型，但是MapGenerator不能对这些类型进行进一步处理了。
- 如果属性的类型不是上面任何一种，MapGenerator把决定数据库类型的步骤留待所有的类都被处理之后再来做。在那时候，如果类在上面描述过的超类搜索过程中被发现了，这个属性会被认为是一个many-to-one的关联。如果类有人和属性，它则是一个组件(component)。否则它就是可序列化的（serializable），或者不是可持久化的。

### 19.3.1. 运行此工具

这个工具会把XML映射写入到标准输出或者/并且到一个文件中去。

在调用这个工具的时候，你必须把你编译过的类放到classpath中去。

```
java -cp hibernate_and_your_class_classpaths  
net.sf.hibernate.tool.class2hbm.MapGenerator options and classnames
```

有两种操作模式：命令行或者交互式。

交互式模式当你使用一个唯一的命令行参数--interact的时候启动。这个模式提供一个命令控制台。你可以用uid=xxx命令设置每个类的UID属性的名字，xxx就是UID属性名。其他可用的命令就是类名的全限定名，或

者“done”命令用来输出XML,并且结束。

在命令行模式下，下面的参数选项和所需处理的类的全限定名可以相互间隔使用。大多数选项会使用多次，每个只影响其后出现的类。

**Table 19.7. MapGenerator命令行选项**

选项	说明
<code>--quiet</code>	不把O-R 映射输出到stdout
<code>--setUID=uid</code>	设置备选UID名单
<code>--addUID=uid</code>	在备选UID名单前面增加一个新的uid
<code>--select=mode</code>	对后面的classes使用select选择的模式(mode)(比如, <i>distinct</i> 或者 <i>all</i> )
<code>--depth=&lt;small-int&gt;</code>	限制后面的类的组件数据递归层数
<code>--output=my_mapping.xml</code>	把O-R 映射输出到一个文件
<code>full.class.Name</code>	把这个类加入到映射中
<code>--abstract=full.class.Name</code>	参见下面的说明

abstract开关指定本工具忽略特定的超类,所以它的继承数上的类不会被映射到一个大表中去。比如，我们来看下面的类继承树：

```
Animal-->Mammal-->Human
```

```
Animal-->Mammal-->Marsupial-->Kangaroo
```

如果不使用--abstract开关，Animal的所有子类都会被放到一个巨大的表中去，包含所有类的所有属性，还有一个用于分辨子类的字段。如果Mammal被标记成abstract，Human和Marsupial会被映射到不同的<class>声明，并且会有各自单独的表。Kangaroo仍然会被认为是Marsupial的子类，除非Marsupial也标记为abstract的。

## Chapter 20. 最佳实践(Best Practices)

设计细颗粒度的持久类并且使用<component>来实现映射。

使用一个Address持久类来封装 street, suburb, state, postcode. 这将有  
利于代码重用和简化代码重构(refactoring)的工作。

对持久类声明标识符属性。

Hibernate中标识符属性是可选的，不过有很多原因来说明你应该使用  
标识符属性。我们建议标识符应该是“人造”的(自动生成，不涉及业务  
含义)，并且不是基本类型。为了最大的灵活性，应该使  
用java.lang.Long Or java.lang.String

为每个持久类写一个映射文件

不要把所有的持久类映射都写到一个大文件中。把 com.eg.Foo 映射  
到com/eg/Foo.hbm.xml中，在团队开发环境中，这一点显得特别有意  
义。

把映射文件作为资源加载

把映射文件和他们的映射类放在一起进行部署。

考虑把查询字符串放在程序外面

如果你的查询中调用了非ANSI标准的SQL函数，那么这条实践经验对  
你适用。把查询字符串放在程序外面可以让程序具有更好的可移植  
性。

使用绑定变量

就像在JDBC编程中一样，应该总是用占位符"?"来替换非常量值，不  
要在查询中用字符串值来构造非常量值！更好的办法是在查询中使用  
命名参数。

不要自己来管理JDBC connections

Hibernate允许应用程序自己来管理JDBC connections，但是应该作为最后没有办法的办法。如果你不能使用Hibernate内建的connections providers，那么考虑实现自己来实现  
`net.sf.hibernate.connection.ConnectionProvider`

### 考虑使用用户自定义类型(custom type)

`net.sf.hibernate.UserType`. This approach frees the application code from implementing transformations to / from a Hibernate type. 假设你有一个Java类型，来自某些类库，需要被持久化，但是该类没有提供映射操作需要的存取方法。那么你应该考虑实现`net.sf.hibernate.UserType`接口。这种办法使程序代码写起来更加自如，不再需要考虑类与Hibernate type之间的相互转换。

### 在性能瓶颈的地方使用硬编码的JDBC

在对性能要求很严格的一些系统中，一些操作(例如批量更新和批量删除)也许直接使用JDBC会更好，但是请先搞清楚这是否是一个瓶颈，并且不要想当然认为JDBC一定会更快。如果确实需要直接使用JDBC，那么最好打开一个Hibernate Session 然后从 Session 获得 connection，按照这种办法你仍然可以使用同样的transaction策略和底层的connection provider。

### 理解Session清洗 ( flushing )

Session会不时的向数据库同步持久化状态，如果这种操作进行的过于频繁，那么性能会受到一定的影响。有时候你可以通过禁止自动flushing尽量最小化非必要的flushing操作，或者更进一步，在一个特殊transaction中改变查询和其它操作的顺序。

### 在三层架构中，考虑使用 `saveOrUpdate()`

当使用一个servlet / session bean 的架构的时候, 你可以把已加载的持久对象在session bean层和servlet / JSP 层之间来回传递。使用新的session 来为每个请求服务，使用 `Session.update()` 或者 `Session.saveOrUpdate()` 来更新对象的持久状态。

### 在两层架构中，考虑使用 `session disconnection`.

当仅仅使用 servlet的时候，你可以在多个客户请求中复用同一个

session，只是要记得在把控制权交还给客户端之前disconnect掉session。

## 不要把异常看成可恢复的

这一点甚至比“最佳实践”还要重要，这是“必备常识”。当异常发生的时候，回滚Transaction，关闭Session。如果你不这样做的话，Hibernate无法保证内存状态精确的反应持久状态。尤其不要使用Session.load()来判断一个给定标识符的对象实例在数据库中是否存在，应该使用find()。

## 对于关联优先考虑lazy fetching

谨慎的使用主动外连接抓取(eager (outer-join) fetching)。对于大多数没有JVM级别缓存的持久对象的关联，应该使用代理(proxyes)或者具有延迟加载属性的集合(lazy collections)。对于被缓存的对象的关联，尤其是缓存的命中率非常高的情况下，应该使用outer-join="false"，显式的禁止掉eager fetching。如果那些特殊的确实适合使用outer-join fetch 的场合，请在查询中使用left join。

## 考虑把Hibernate代码从业务逻辑代码中抽象出来

把Hibernate的数据存取代码隐藏到接口(interface)的后面，组合使用DAO和Thread Local Session模式。通过Hibernate的UserType，你甚至可以用硬编码的JDBC来持久化那些本该被Hibernate持久化的类。(该建议更适用于规模足够大应用程序中，对于那些只有5张表的应用程序并不适合。)

### 1、安装iBATIS

下载路径:<http://www.ibatis.com/common/download.html>

解压缩到D:\develop\ibatis目录(以下称{iBATIS})

### 2、安装数据库HSQLDB

下载路径:<http://hsqldb.sourceforge.net/>

解压缩到D:\develop\hsqldb目录(以下称{hsqldb})

### 3、配置java环境

在D盘根目录下，增加文件 J.BAT:

```
SET JAVA_HOME=D:\j2sdk1.4.2_05
```

```
SET ANT_HOME=D:\develop\ant1.5
```

```
SET HSQLDB_HOME=D:\develop\hsqldb
```

```
SET CLASSPATH=.;%JAVA_HOME%\lib\dt.jar;%JAVA_HOME%\lib\tools.jar;%CLASSPATH%
```

```
SET CLASSPATH = %CLASSPATH%;%ANT_HOME%\lib\ant.jar
```

```
SET CLASSPATH = %CLASSPATH%;%HSQLDB_HOME%\lib\hsqldb.jar
```

```
SET PATH=%PATH%;%JAVA_HOME%\bin;%J2EE_HOME%\bin;%ANT_HOME%\bin
```

### 4、准备数据库脚本

在Alo SCM项目的sql目录下，新增文件crtbastbl.sql:

```
Create Table Brand(  
  ID integer not null PRIMARY KEY,  
  Name varchar(20)  
);
```

### 5、配置eclipse 3.0.1项目--Alo SCM

eclipse位置: E:\eclipse (以下称{eclipse})

菜单项: Project->Properties->Java Build Path

Libraries标签: Add External JARs->指向{iBATIC}目录->选择ibatis-common-2.jar,ibatis-dao-2.jar,ibatis-sqlmap-2.jar三个文件

件

## 6、配置数据库aloscsm

在{hsqldb}\bin目录下添加文件server.properties和runServer.bat，文件内容如下，

runServer.bat:

```
java -cp ../lib/hsqldb.jar org.hsqldb.Server
```

server.properties:

```
server.port=9001
```

```
server.database=../data/aloscsm
```

```
server.silent=true
```

```
server.no_system_exit=false
```

## 7、测试数据库运行

在{hsqldb}\bin目录下，运行runServer.bat，若出现以下文字，则说明数据库已经运行:

```
Opening database: ../data/aloscsm
```

```
HSQldb server 1.7.1 is running
```

```
Use SHUTDOWN to close normally. Use [Ctrl]+[C] to abort abruptly
```

```
Mon Feb 21 10:22:36 CST 2005 Listening for connections ...
```

你可以在{hsqldb}\data目录下，看到三个文件aloscsm.data、aloscsm.properties、aloscsm.script

## 8、编辑sqlMapConfig.xml文件

在Alo SCM项目目录下，创建sqlMapConfig.xml文件：

```
<?xml version="1.0" encoding="GBK" ?>
```

```
<!DOCTYPE sqlMapConfig PUBLIC "-//iBATIC.com//DTD SQL Map Config 2.0//EN"
```

```
"http://www.ibatis.com/dtd/sql-map-config-2.dtd">
```

```
<!-- Always ensure to use the correct XML header as above! -->
```

```
<sqlMapConfig>
```

```
<!-- The properties (name=value) in the file specified here can be used
```

placeholders in this config file (e.g. `??${driver}??`). The file is relative to the classpath and is completely optional. -->

```
<properties resource="conf/sqlMapConfigHSQL.properties " />
```

<!-- These settings control SqlMapClient configuration details, primarily to do

with transaction management. They are all optional (more detail later in

this document). -->

```
<settings cacheModelsEnabled="true"
  enhancementEnabled="true"
  lazyLoadingEnabled="true"
  maxRequests="32"
  maxSessions="10"
  maxTransactions="5"
  useStatementNamespaces="false" />
```

<!-- Type aliases allow you to use a shorter name for long fully qualified class names. -->

```
<typeAlias alias="brand" type="base.Brand"/>
```

<!-- Configure a datasource to use with this SQL Map using SimpleDataSource. Notice the use of the properties from the above resource -->

```
<transactionManager type="JDBC" >
  <dataSource type="SIMPLE">
    <property name="JDBC.Driver" value="${driver}"/>
    <property name="JDBC.ConnectionURL" value="${url}"/>
    <property name="JDBC.Username" value="${username}"/>
    <property name="JDBC.Password" value="${password}"/>
    <property name="JDBC.DefaultAutoCommit" value="true" />
    <property name="Pool.MaximumActiveConnections" value="10"/>
    <property name="Pool.MaximumIdleConnections" value="5"/>
    <property name="Pool.MaximumCheckoutTime" value="120000"/>
    <property name="Pool.TimeToWait" value="500"/>
    <property name="Pool.PingQuery" value="select 1 from Brand"/>
    <property name="Pool.PingEnabled" value="false"/>
    <property name="Pool.PingConnectionsOlderThan" value="1"/>
    <property name="Pool.PingConnectionsNotUsedFor" value="1"/>
  </dataSource>
```

```
</transactionManager>
<!-- Identify all SQL Map XML files to be loaded by this SQL map.
Notice the paths are relative to the classpath. For now,
we only have one?? -->
<sqlMap resource="map/Brand.xml" />
<sqlMap resource="map/Size.xml" />
<sqlMap resource="map/Type.xml" />
</sqlMapConfig>
```

### 9、编辑sqlMapConfigHSQL.properties文件

在 Alo SCM 项目的conf目录下，创建sqlMapConfigHSQL.properties文件:

```
driver=org.hsqldb.jdbcDriver
url=jdbc:hsqldb:hsql://localhost
username=sa
password=
```

### 10、编辑Brand.xml文件

在 Alo SCM 项目的map目录下，创建Brand.xml文件:

```
<?xml version="1.0" encoding="GBK"?>
<!DOCTYPE sqlMap PUBLIC "-//ibatis.com//DTD SQL Map 2.0//EN"
"http://www.ibatis.com/dtd/sql-map-2.dtd">
<sqlMap namespace="Brand">
  <resultMap id="get-brand-result" class="base.Brand">
    <result property="id" column="ID"/>
    <result property="name" column="NAME"/>
  </resultMap>
  <select id="getBrand" parameterClass="java.lang.Integer" resultClass="base.Brand">
    <![CDATA[
    SELECT
    ID,
    Name
    FROM Brand
    WHERE ID=#id#;
    ]]>
```

```
</select>

<insert id="insertBrand" parameterClass="base.Brand">
  <![CDATA[
    INSERT INTO Brand (ID, NAME)
    VALUES (#id#, #name#)
  ]]>
</insert>

<update id="updateBrand" parameterClass="base.Brand">
  <![CDATA[
    UPDATE Brand
      SET ID=#id#,
      NAME=#name#
    WHERE ID=#id#
  ]]>
</update>

<delete id="deleteBrand" parameterClass="base.Brand">
  <![CDATA[
    DELETE FROM Brand
    WHERE ID=#id#
  ]]>
</delete>

<select id="getBrandCount" resultClass="java.lang.Integer">
  <![CDATA[
    SELECT
    count(1)
    FROM Brand
  ]]>
</select>
</sqlMap>
```

## 11、编辑Brand.java文件

```
/*
```

```
* Created on 2005-2-1
```

```
*/
```

```
*/  
package base;  
  
import java.io.Serializable;  
  
/**  
 * @author robin  
 *  
 */  
public class Brand implements Serializable{  
    private int id;  
    private String name;  
  
    /**  
     * @return Returns the id.  
     */  
    public int getId() {  
        return id;  
    }  
    /**  
     * @param id The id to set.  
     */  
    public void setId(int id) {  
        this.id = id;  
    }  
    /**  
     * @return Returns the name.  
     */  
    public String getName() {  
        return name;  
    }  
    /**  
     * @param name The name to set.  
     */  
    public void setName(String name) {  
        this.name = name;  
    }  
}
```

```
    public static void main(String[] args) {  
    }  
}
```

## 12、编辑BrandDAO.java

```
/*  
 * Created on 2005-2-17  
 *  
 */  
package dao;  
  
import com.ibatis.common.resources.Resources;  
import com.ibatis.sqlmap.client.*;  
import java.io.Reader;  
import base.Brand;  
import java.sql.*;  
import util.KeyGenerator;  
  
/**  
 * @author robin  
 *  
 */  
public class BrandDAO {  
    private static SqlMapClient sqlMap;  
  
    public BrandDAO(){  
        try{  
            String resource = "./sqlMapConfig.xml";  
            Reader reader = Resources.getResourceAsReader (resource);  
            sqlMap = SqlMapClientBuilder.buildSqlMapClient(reader);  
        }catch(Exception e){  
            System.out.println(e.getMessage());  
        }  
    }  
  
    public void getLine(Integer id){  
        try{
```

```
        Brand brand=(Brand)sqlMap.queryForObject("getBrand", id);
        //System.out.println(brand.getName());
    }catch(SQLException sqle){
        System.out.println(sqle.getMessage());
    }
}
```

```
public void insertBrand(int id, String name){
    try{
        Brand brand = new Brand();
        brand.setId(id);
        brand.setName(name);
        sqlMap.insert("insertBrand",brand);
    }catch(SQLException sqle){
        System.out.println(sqle.getMessage());
    }
}
```

```
public void deleteBrand(int id){
    try{
        Brand brand = new Brand();
        brand.setId(id);
        sqlMap.delete("deleteBrand",brand);
    }catch(SQLException sqle){
        System.out.println(sqle.getMessage());
    }
}
```

```
public void updateBrand(int id, String name){
    try{
        Brand brand = new Brand();
        brand.setId(id);
        brand.setName(name);
        sqlMap.delete("updateBrand",brand);
    }catch(SQLException sqle){
        System.out.println(sqle.getMessage());
    }
}
```

```

}

/**
 * @return int: count.
 * This function is used to return count of this table.
 */
public int count(){
    Integer count=new Integer(0);
    try{
        //Brand brand = new Brand();
        count=(Integer)sqlMap.queryForObject("getBrandCount",null);
        //System.out.println(count.toString());
    }catch(SQLException sqle){
        System.out.println(sqle.getMessage());
    }
    return count.intValue();
}

public static void main(String[] arg0){
    BrandDAO a = new BrandDAO();
    KeyGenerator key;

    key = KeyGenerator.getInstance("brand");

    a.insertBrand(key.getNextKey(),"美成");
    a.count();
}
}

```

参考文档《iBATIS SQL Maps 入门教程》、《iBATIS SQL Maps 开发指南》

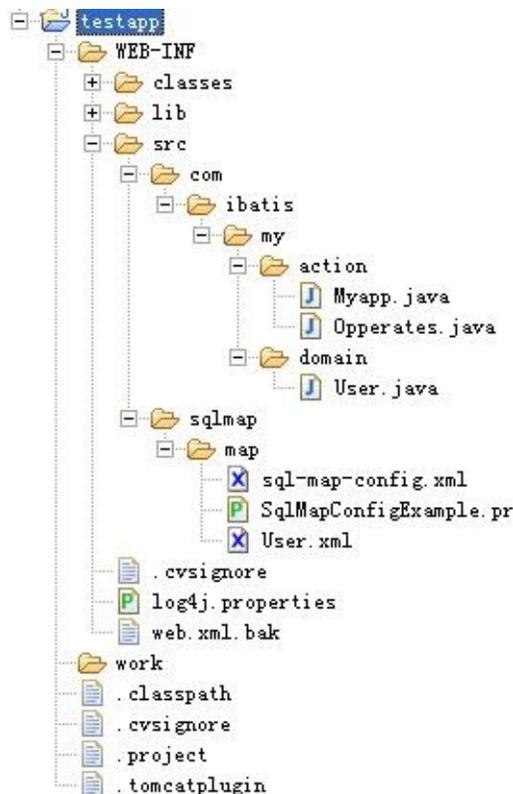
学习iBATIS的过程，以备查考。

## ibATIS入门 (实例代码)

首先建立表t\_user

```
CREATE TABLE public.t_user
(
  uid int4,
  name varchar(20),
  sex int4,
  age int4,
  addr varchar(50),
  zipcode varchar(6),
) WITH OIDS;
```

整个project的目录如下：



然后把所需要的包引入（可把iBAtIS\_JPetStore-4.0.5的lib目录下的包直接拿过来用，另外还要一个jdbc的包）。如下图：



新建一个sql-map-config.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sqlMapConfig
PUBLIC "-//iBATIS.com//DTD SQL Map Config 2.0//EN"
"http://www.ibatis.com/dtd/sql-map-config-2.dtd">

<sqlMapConfig>

  <properties
    resource="sqlmap/map/SqlMapConfigExample.properties" />

  <settings
    cacheModelsEnabled="true"
    enhancementEnabled="true"
    lazyLoadingEnabled="true"
    maxRequests="32"
    maxSessions="10"
    maxTransactions="5"
    useStatementNamespaces="false" />

  <transactionManager type="JDBC" >
    <dataSource type="SIMPLE">
      <property name="JDBC.Driver" value="${driver}"/>
      <property name="JDBC.ConnectionURL" value="${url}"/>
      <property name="JDBC.Username" value="${username}"/>
      <property name="JDBC.Password" value="${password}"/>
    </dataSource>
  </transactionManager>
</sqlMapConfig>
```

```
</transactionManager>
```

```
<sqlMap resource="sqlmap/map/User.xml" />  
</sqlMapConfig>
```

### 新建数据库资源文件SqlMapConfigExample.properties

```
driver=org.postgresql.Driver  
url=jdbc:postgresql://localhost/testapp  
username=zzz  
password=zzz
```

### 类user.java:

```
package com.ibatis.my.domain;  
public class User {  
    public int uid;  
    public String name;  
    public int sex;  
    public int age;  
    public String addr;  
    public String zipcode;  
  
    public String getAddr() {  
        return addr;  
    }  
    public void setAddr(String addr) {  
        this.addr = addr;  
    }  
    public int getAge() {  
        return age;  
    }  
    public void setAge(int age) {  
        this.age = age;  
    }  
    public String getName() {  
        return name;  
    }  
}
```

```

    }
    public void setName(String name) {
        this.name = name;
    }
    public int getSex() {
        return sex;
    }
    public void setSex(int sex) {
        this.sex = sex;
    }
    public int getUid() {
        return uid;
    }
    public void setUid(int uid) {
        this.uid = uid;
    }
    public String getZipcode() {
        return zipcode;
    }
    public void setZipcode(String zipcode) {
        this.zipcode = zipcode;
    }
}

```

#### 配置user.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sqlMap
PUBLIC "-//iBATIS.com//DTD SQL Map 2.0//EN"
"http://www.ibatis.com/dtd/sql-map-2.dtd">

<sqlMap namespace="User">
    <select id="getUser" parameterClass="java.lang.Integer">
        SELECT * FROM t_user WHERE uid=#uid#
    </select>
    <insert id="insertUser"
parameterClass="com.ibatis.my.domain.User">
        INSERT INTO

```

```
        t_user(uid, name, sex, age, addr, zipcode)
        VALUES (#uid#, #name#, #sex#, #age#, #addr#,
#zipcode#)
</insert>
</sqlMap>
```

### 建一个myapp.java

```
import java.io.Reader;

import com.ibatis.common.resources.Resources;
import com.ibatis.sqlmap.client.SqlMapClient;
import com.ibatis.sqlmap.client.SqlMapClientBuilder;

public class MyApp {
    private static final SqlMapClient sqlMap;
    static {
        try {
            String resource = "sqlmap/map/sql-map-config.xml";
            Reader reader = Resources.getResourceAsReader
(resource);
            sqlMap =
SqlMapClientBuilder.buildSqlMapClient(reader);
        }
        catch (Exception e) {
            e.printStackTrace();
            throw new RuntimeException ("Error initializing MyApp
class. Cause:"+e);
        }
    }
    public static SqlMapClient getSqlMapInstance () {
        return sqlMap;
    }
}
```

### 操作类Operates.java

```

package com.ibatis.my.action;

import com.ibatis.my.domain.User;
import com.ibatis.sqlmap.client.SqlMapClient;

public class Opperates {

    public static void main(String[] args) {
        User newUser= new User();
        newUser.setUid(5);
        newUser.setName("oring");
        newUser.setSex(2);
        newUser.setAge(2);

        newUser.setAddr("dfghg");
        newUser.setZipcode("301");
        SqlMapClient sqlMap = Myapp.getSqlMapInstance();
        try{

            Integer uid= new Integer(1);
            // User user = (User) sqlMap.queryForObject ("getUser",
uid);
            sqlMap.insert ("insertUser", newUser);
        }
        catch(Exception e){
            e.printStackTrace();
        }

    }
}

```

在Eclipse中选择Run As Java Application运行Opperates.java，即可看到数据库中添加了一条记录，如下：

```
statement: INSERT INTO t_user(uid, name, sex, age, addr, zipcode)
           VALUES (5, 'oring', 2, 2, 'dfghg', '301')
duration: 0.000 ms
statement: commit;begin;
duration: 32.000 ms
statement: rollback; begin;
duration: 0.000 ms
```

## Hibernate、iBATIS 与 BLOB

在存储图片、可执行文件等二进制信息时（当然直接放在文件系统上也数据就派上用场了。本文无太多深度可言，能为大家在开发过程中提供参

### Hibernate 与 SQL Server BLOB

BLOB 数据在 SQL Server 数据库中主要由 IMAGE 类型体现，最大容量存储方式不同于普通的数据类型，对于普通类型的数据系统直接在用户定储数据值，而对于 IMAGE 类型数据，系统开辟新的存储页面来存放这些；IMAGE 类型数据字段存放的仅是一个 16 字节的指针，该指针指向存放该 IMAGE 数据的页面。如果你对 Hibernate 还不熟息，请看[这里](#)。

新建名为“BLOB\_TEST”的表，字段分别是 INT 类型的“ID”和 IMAGE 类型的“MYBLOB”。从文件系统读取“sample.jpg”并转换成字节数组再放进 Blob 例。写入程序如下：

```
import java.io.*;

import net.sf.hibernate.*;
import net.sf.hibernate.cfg.*;

import bo.*;

public class Tester {

    public void DoTest() {
        InputStream in = null;
        BlobTest blobTest = null;
        Configuration cfg = null;
        SessionFactory sessions = null;
        Session session = null;
        Transaction tx = null;
        try {
```

```
//begin InputStream
in = new FileInputStream("d:/sample.jpg");
byte[] b = new byte[in.available()];
in.read(b);
in.close();

//begin BlobTest
blobTest = new BlobTest();
blobTest.setMyblob(b);

//begin Hibernate Session
cfg = new Configuration().configure();
sessions = cfg.buildSessionFactory();
session = sessions.openSession();
tx = session.beginTransaction();
session.save(blobTest);
tx.commit();
} catch (Exception e) {
e.printStackTrace();
} finally {
try {
session.close();
} catch (Exception e1) {
e1.printStackTrace();
}
}
}
}
```

取出程序如下：

```
import java.io.*;

import net.sf.hibernate.*;
import net.sf.hibernate.cfg.*;

import bo.*;

public class Tester {
```

```

public void DoTest() {
    OutputStream out = null;
    BlobTest blobTest = null;
    Configuration cfg = null;
    SessionFactory sessions = null;
    Session session = null;
    try {
        //begin Hibernate Session
        cfg = new Configuration().configure();
        sessions = cfg.buildSessionFactory();
        session = sessions.openSession();

        //begin BlobTest
        blobTest = new BlobTest();
        blobTest = (BlobTest) session.load(BlobTest.class, new Integer(23));

        //begin OutputStream
        out = new FileOutputStream("d:/sample.jpg");
        out.write(blobTest.getMyblob());
        out.flush();
        out.close();
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        try {
            session.close();
        } catch (Exception e1) {
            e1.printStackTrace();
        }
    }
}
}
}
}

```

## Hibernate 与 MySQL BLOB

MySQL 中的 BLOB 数据由四种类型体现，分别是 TINYBLOB 其容量为 1KB、BLOB 其容量为 64KB、MEDIUMBLOB 其容量为 16MB、LONGBLOB 其容量为 4GB。

新建名为“BLOB\_TEST”的表，字段分别是 INTEGER 类型的“ID”和 MEDIUMBLOB 类型的“MYBLOB”。从文件系统读取“sample.jpg”并转换再放进 BlobTest 对象实例。写入、取出程序和上面的 SQL Server 一样。

## Hibernate 与 Oracle BLOB

为了不使用“for update”锁住数据库，遂打算让 Oracle LONG RAW 类型最大容量 2GB。经过测试后发现，直接写 JDBC 代码可以保存，但 Hibernate 4K 大小内容，换成 Hibernate 3.0 beta3 也未能成功。偶然的机会在邮件列表 JDBC Driver 的问题，换成 Oracle 10g 的驱动后问题解决。

新建名为“BLOB\_TEST”的表，字段分别是 NUMBER 类型的“ID”和 BLOB 类型的“MYBLOB”。从文件系统读取“sample.jpg”并转换成字节数组再放入对象实例。写入、取出程序和 SQL Server 一样。

如果你一定要用 **Oracle BLOB** 类型，接着往下看：

Hibernate 处理 Oracle BLOB 类型较特殊，从文件系统读取“sample.jpg”对象实例的是 java.sql.Blob 类型，而不是字节数组。

```
import java.io.*;

import net.sf.hibernate.*;
import net.sf.hibernate.cfg.*;
import oracle.sql.*;

import bo.*;

public class Tester {

    public void DoTest() {
        BLOB blob = null;
        InputStream in = null;
```

```
OutputStream out = null;
BlobTest blobTest = null;
Configuration cfg = null;
SessionFactory sessions = null;
Session session = null;
Transaction tx = null;
try {
    //begin InputStream
    in = new FileInputStream("d:/sample.jpg");
    byte[] b = new byte[in.available()];
    in.read(b);
    in.close();

    //begin BlobTest
    blobTest = new BlobTest();
    blobTest.setMyblob(BLOB.empty_lob());

    //begin Hibernate Session
    cfg = new Configuration().configure();
    sessions = cfg.buildSessionFactory();
    session = sessions.openSession();
    tx = session.beginTransaction();
    session.save(blobTest);
    session.flush();
    session.refresh(blobTest, LockMode.UPGRADE);
    blob = (BLOB) blobTest.getMyblob();
    out = blob.getBinaryOutputStream();
    out.write(b);
    out.close();
    session.flush();
    tx.commit();
} catch (Exception e) {
    e.printStackTrace();
} finally {
    try {
        session.close();
    } catch (Exception e1) {
        e1.printStackTrace();
    }
}
}
```

取出程序和其他两种数据库操作几乎一样。

## iBATIS SQL Maps 与 SQL Server BLOB

建表过程和 Hibernate 操作 SQL Server 一样，如果你对 iBATIS SQL Map 请看[这里](#)。

映射文件如下：

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

<!DOCTYPE sqlMap
  PUBLIC "-//iBATIS.com//DTD SQL Map 2.0//EN"
  "http://www.ibatis.com/dtd/sql-map-2.dtd">

<sqlMap>

  <insert id="insertBlob" parameterClass="bo.BlobTest">
    <![CDATA[
      insert into blob_test (myblob) values (#myblob#)
    ]]>
    <selectKey resultClass="java.lang.Integer" keyProperty="id">
      <![CDATA[
        SELECT @@IDENTITY AS ID
      ]]>
    </selectKey>
  </insert>

  <resultMap id="get-blob-result" class="bo.BlobTest">
    <result property="id" column="id"/>
    <result property="myblob" column="myblob"/>
  </resultMap>

  <select id="getBlob" resultMap="get-blob-result"
```

```
parameterClass="bo.BlobTest">
  <![CDATA[
    select * from blob_test where id=#id#
  ]]>
</select>

</sqlMap>
```

写入程序如下：

```
import java.io.*;

import com.ibatis.sqlmap.client.*;
import com.ibatis.common.resources.*;

import bo.*;

public class Tester {

public void DoTest() {
byte[] b=null;
Reader reader = null;
InputStream in = null;
BlobTest blobTest = null;
SqlMapClient sqlMap = null;
String resource = "SqlMapConfig.xml";
try {
//begin InputStream
in = new FileInputStream("d:/sample.jpg");
b = new byte[in.available()];
in.read(b);
in.close();

//begin BlobTest
blobTest = new BlobTest();
blobTest.setMyblob(b);

//begin SqlMapClient
reader = Resources.getResourceAsReader(resource);
```

```
sqlMap = SqlMapClientBuilder.buildSqlMapClient(reader);
sqlMap.startTransaction();
sqlMap.insert("insertBlob", blobTest);
sqlMap.commitTransaction();
} catch (Exception e) {
    e.printStackTrace();
} finally {
    try {
        sqlMap.endTransaction();
    } catch (Exception e1) {
        e1.printStackTrace();
    }
}
}
```

取出程序如下：

```
import java.io.*;

import com.ibatis.sqlmap.client.*;
import com.ibatis.common.resources.*;

import bo.*;

public class Tester {

    public void DoTest() {
        Reader reader = null;
        OutputStream out = null;
        BlobTest blobTest = null;
        SqlMapClient sqlMap = null;
        String resource = "SqlMapConfig.xml";
        try {
            //begin BlobTest
            blobTest = new BlobTest();
            blobTest.setBlob(new Integer(21));

            //begin SqlMapClient
            reader = Resources.getResourceAsReader(resource);
```

```

sqlMap = SqlMapClientBuilder.buildSqlMapClient(reader);
blobTest = (BlobTest) sqlMap.queryForObject("getBlob", blobTest);

//begin OutputStream
out = new FileOutputStream("d:/sample.jpg");
out.write(blobTest.getMyblob());
out.close();
} catch (Exception e) {
e.printStackTrace();
} finally {
try {
sqlMap.endTransaction();
} catch (Exception e1) {
e1.printStackTrace();
}
}
}
}
}
}

```

## iBATIS SQL Maps 与 MySQL BLOB

这个主题很简单，需要注意映射文件 insert 元素主键生成方式，写入、！面的 SQL Server 一样：

```

<insert id="insertBlob" parameterClass="bo.BlobTest">
  <![CDATA[
    insert into blob_test (myblob) values (#myblob#)
  ]]>
  <selectKey resultClass="java.lang.Integer" keyProperty="id">
    <![CDATA[
      select last_insert_id();
    ]]>
  </selectKey>
</insert>

```

## iBATIS SQL Maps 与 Oracle BLOB

使用 Oracle LONG RAW 类型，注意映射文件 insert 元素主键生成方式，程序和上面的 SQL Server 一样：

```
<insert id="insertBlob" parameterClass="bo.BlobTest">
  <selectKey resultClass="int" keyProperty="id">
    <![CDATA[
      select hibernate_sequence.nextval from dual
    ]]>
  </selectKey>
  <![CDATA[
    insert into blob_test (id,myblob) values (#id#,#myblob#)
  ]]>
</insert>
```

如果你一定要用 Oracle BLOB 类型，接着往下看：

在 iBATIS 2.0.9 以前，处理 Oracle BLOB 类型相当麻烦，要自己实现 TypeHandlerCallback 接口。iBATIS 2.0.9 的出现使一切都简单了，写入、界面的 SQL Server 一样。

## mysql 中得到新插入数据的 ID ，在 ibatis 中的应用

### mysql 中得到新插入数据的 ID ，在 ibatis 中的应用

我们经常有这种需求，插入一条数据库记录时，需要马上得到这条记录的 AUTO\_INCREMENT ID 。很多数据库有内置支持，mysql 中一直没有找到办法，以为没有内置支持呢，所以用个比较笨的办法，有朋友问我这个问题时，我也是说的这个：

```
select max(ID) from mytable
```

返

回最大的 id 。这个语句得与插入记录的语句在同一个“事务”里，如果用 jdbc 事务，应该一个 connection commit 里。在数据量不大时，应用当然是没有问题。数据量非常大的情况我就不知道了，我认为应该是没有问题的。因为，一个“事务”就保证了 SQL 插入操作后与 SQL 取得 max(ID) 的操作之间，不可能再有别的数据库操作。我是这样以为，但还是没有底。。。

今天无意中发现，mysql 还是有内置支持的，有专门的一个函数：

```
LAST_INSERT_ID([expr])
```

google 这个函数名称，找到英文或者中文的 mysql 参考手册都有详细的说明。文档说明也特别提到，用这个函数得到最新的 id ，操作也还是需要与插入操作同一个 connection 范围内。应该说与原来的作法是类似的。

还

有一点意外的发现就是，ibatis 里的 sqlmap insert 操作，竟然返回新插入数据的自增 id 。这样我在程序里就不需要用二个独立的方法了，一个 sqlmap insert 就搞定了，真是爽。当然 sqlmap insert SQL 写法有不同：

```
<insert id="insertTopic" parameterClass="topic">  
insert into TOPIC (FORUM_ID, USERID, TITLE, SUMMARY,
```

```
CREATE_TIME)
values (#forumId#, #userid#, #title#, #summary#, now())
<selectKey resultClass="string" keyProperty="id">
select last_insert_id() as ID from TOPIC limit 1
</selectKey>
</insert>
```

现在我们来注意一下这行 `select last_insert_id() as ID from TOPIC limit 1`，不同的数据库，这地方有不同，都有自己特别的内置实现。请又特别注意 `limit 1` 这地方，看文档应该是返回一条记录的，就是返回 `id` 呀，但实际应用中我发现，竟然返回了 `n` 条记录。没有办法，我只有限制记录条数为 `1`，但结果也是对的。不知道这个算不算 `mysql` 的 `bug`。

如  
果大家仔细看 `last_insert_id()` 这个函数的文档，他举了个用这个函数实现 `sequence` 的例子，感觉很值得引起注意，暂时我是用不着，但有需要用得着的时候。可能有这样一种需要：需要得到自增 `id`，这个 `id` 我可能是用作数据库表的主键，也可能有别的用途。一般的想法，好像只有在数据库里写个存储过程，反正我是没有写过，关键是需要考虑多用户并发。。。

用 `last_insert_id()` 函数这样来模拟一个 `sequence` 实现：

```
mysql> CREATE TABLE sequence (id INT NOT NULL);
mysql> INSERT INTO sequence VALUES (0);

mysql> UPDATE sequence SET id=LAST_INSERT_ID(id+1);
```

因为“如果 `expr` 被作为一个参数传递给 `LAST_INSERT_ID()`，那么函数将返回这个参数的值，并且被设置为 `LAST_INSERT_ID()` 返回的下一个值”，这样每次 `update sequence` 后，`id` 肯定是自增过了，关键是这个数据库级的支持，不需要你再去考虑多用户并发问题。

## [Ibatis2.0使用说明（一）——入门实例篇](#)

本文章将从一个Ibatis的具体示例，帮助你快速了解IBatis框架。

一个简单的IBatis应用包含以下基本步骤：

### 一、 配置文件

1. 配置SqlMapConfig.properties文件

2. 配置SqlMapConfig.xml文件

3. 配置SqlMap.xml文件（可能有多个文件，一般情况下，可以一个表对应一个SqlMap.xml文件，文件名称可以与表名相同）

注意：上面所述的SqlMapConfig.xml文件必须在类路径中，SqlMapConfig.properties和SqlMap.xml文件可以在类路径中，也可以不在类路径中。当SqlMapConfig.properties和SqlMap.xml文件不在类路径中的时候，配置也不同，在本文中，这三个文件都放在类路径中。

### 二、 程序调用

1. 初始化SqlMapClient对象。

2. 运行Sql语句：你可以调用SqlMapClient对象的queryfor...()、insert()、update()、delete()来分别执行select、insert、update和delete操作。

好了，下面我们结合实例进行讲解：

### 三、 实例：

下面的例子是以mysql为例进行说明，建立了一个author表，为了方便调试代码，你可以将ibatis-common-2.jar、ibatis-dao-2.jar、ibatis-sqlmap-2.jar和lib目录下的所有的jar都加载到你的程序中，在后续的文章中，将会说明每个Jar的用途。

## （一）创建数据库和表

创建一个名字为IBatisExample的数据库

```
CREATE TABLE author (  
  auth_id int(8) NOT NULL auto_increment,  
  auth_name varchar(100) NOT NULL default "",  
  auth_age int(3) NOT NULL default '0',  
  auth_tel varchar(100) NOT NULL default "",  
  auth_address varchar(100) NOT NULL default "",  
  PRIMARY KEY (auth_id)  
) TYPE=MyISAM;  
INSERT INTO author VALUES (1, '作者一', 30, '025-12345678', '南京');  
INSERT INTO author VALUES (2, '作者二', 30, '025-12345678', '南京');
```

## （二）配置文件

### 1. 配置SqlMapConfig.properties文件

文件内容：

```
driver=org.gjt.mm.mysql.Driver  
url=jdbc:mysql://192.168.0.26:3306/IBatisExample?  
useUnicode=true&characterEncoding=GB2312  
username=root  
password=123456
```

### 2. 配置SqlMapConfig.xml文件

文件内容：

```
<?xml version="1.0" encoding="UTF-8" ?>  
<!DOCTYPE sqlMapConfig  
PUBLIC "-//iBATIS.com//DTD SQL Map Config 2.0//EN"  
"http://www.ibatis.com/dtd/sql-map-config-2.dtd">  
<!-- Always ensure to use the correct XML header as above! -->  
<sqlMapConfig>
```

<!-- The properties (name=value) in the file specified here can be used placeholders in this config

file (e.g. "\${driver}". The file is relative to the classpath and is completely optional. -->

```
<properties resource="SqlMapConfig.properties" />
```

<!-- These settings control SqlMapClient configuration details, primarily to do with transaction

management. They are all optional (more detail later in this document). -->

```
<settings
```

```
cacheModelsEnabled="true"
```

```
enhancementEnabled="true"
```

```
lazyLoadingEnabled="true"
```

```
maxRequests="32"
```

```
maxSessions="10"
```

```
maxTransactions="5"
```

```
useStatementNamespaces="false"
```

```
/>
```

<!-- Configure a datasource to use with this SQL Map using SimpleDataSource.

Notice the use of the properties from the above resource -->

```
<transactionManager type="JDBC" >
```

```
<dataSource type="SIMPLE">
```

```
<property name="JDBC.Driver" value="${driver}"/>
```

```
<property name="JDBC.ConnectionURL" value="${url}"/>
```

```
<property name="JDBC.Username" value="${username}"/>
```

```
<property name="JDBC.Password" value="${password}"/>
```

```
<property name="JDBC.DefaultAutoCommit" value="true" />
```

```
<property name="Pool.MaximumActiveConnections" value="10"/>
```

```
<property name="Pool.MaximumIdleConnections" value="5"/>
```

```
<property name="Pool.MaximumCheckoutTime" value="120000"/>
```

```
<property name="Pool.TimeToWait" value="500"/>
```

```

<property name="Pool.PingQuery" value="select 1 from author"/>
<property name="Pool.PingEnabled" value="false"/>
<property name="Pool.PingConnectionsOlderThan" value="1"/>
<property name="Pool.PingConnectionsNotUsedFor" value="1"/>
</dataSource>
</transactionManager>
<!-- Identify all SQL Map XML files to be loaded by this SQL map.
Notice the paths
are relative to the classpath. For now, we only have one... -->
<sqlMap resource="com/ibatis/sqlmap/author.xml" />
</sqlMapConfig>

```

### 3. 配置SqlMap.xml文件 这里我们命名为author.xml

```

<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE sqlMap
PUBLIC "-//iBATIS.com//DTD SQL Map 2.0//EN"
"http://www.ibatis.com/dtd/sql-map-2.dtd">
<sqlMap namespace="Author">
<typeAlias alias="Author" type="com.ibatis.beans.Author" />

<select id="getAuthor" parameterClass="int" resultClass="Author">
  SELECT auth_id as id,auth_name as name,auth_age as age,auth_tel
as telephone,auth_address as address FROM author WHERE auth_id
= #id#
</select>

<statement id="getAllAuthor" resultMap="authorResult">
  SELECT * FROM author
</statement>

<insert id="insertAuthor" parameterMap="authorParameter">
  INSERT INTO author (auth_name,auth_age,auth_tel,auth_address)

```

```
VALUES (?, ?, ?, ?)
```

```
</insert>
```

```
<update id="updateAuthor" parameterClass="Author">
```

```
UPDATE author set auth_name=#name# WHERE auth_id = #id#
```

```
</update>
```

```
<delete id="deleteAuthor" parameterClass="int">
```

```
delete from author WHERE auth_id = #id#
```

```
</delete>
```

```
</sqlMap>
```

### (三) 程序调用

由于源代码很长，所以这里我只给出一些简单的程序调用方法，所以如果有人想要源代码的话，可以留下你的邮箱。

1. 初始化一个SqlMapClient对象，代码如下：

```
public class SqlMapConf
```

```
{
```

```
    private static SqlMapClient sqlMapClient;
```

```
    static
```

```
{
```

```
    try
```

```
{
```

```
        System.out.println("sqlMapClient initing.....");
```

```
        String resource = "SqlMapConfig.xml";
```

```
        Reader reader = Resources.getResourceAsReader (resource);
```

```
        sqlMapClient = SqlMapClientBuilder.buildSqlMapClient(reader);
```

```
    }
```

```
    catch (Exception e)
```

```
{
```

```
        e.printStackTrace();
```

```
        throw new RuntimeException ("Error initializing MyAppSqlConfig  
class. Cause: " +e);
```

```
}  
}  
public static SqlMapClient getInstance()  
{  
    return sqlMapClient;  
}  
}
```

2. 然后要为Author表写一个bean，代码如下：

```
public class Author  
{  
    private int id;  
    private int age;  
    private String name;  
    private String address;  
    private String telephone;  
  
    public int getId()  
    {  
        return id;  
    }  
    public void setId(int id)  
    {  
        this.id=id;  
    }  
    public int getAge()  
    {  
        return age;  
    }  
    public void setAge(int age)  
    {  
        this.age=age;  
    }  
}
```

```

public String getName()
{
    return name;
}
public void setName(String name)
{
    this.name=name;
}

public String getAddress()
{
    return address;
}
public void setAddress(String address)
{
    this.address=address;
}
public String getTelephone()
{
    return telephone;
}
public void setTelephone(String telephone)
{
    this.telephone=telephone;
}
}

```

### 3. 程序调用：

这里将只示范一下getAuthor、insertAuthor1、updateAuthor和deleteAuthor的方法。

首先应该得到一个SqlMapClient实例：

```
SqlMapClient sqlMapClient = SqlMapConf.getInstance();
```

```
( 1 ) getAuthor :
Author author = (Author)sqlMapClient.queryForObject("getAuthor",
new Integer(1));
( 2 ) getAllAuthor
List authorList = (List)sqlMapClient.queryForList("getAllAuthor",
null);
( 3 ) insertAuthor :
Author author = new Author();
author.setName("作者三");
author.setAge(31);
author.setAddress("南京");
author.setTelephone("025-987654321");
sqlMapClient.insert(operaName, author);
( 4 ) updateAuthor
Author author = new Author();
author.setName("Updated");
author.setId(authorID);
sqlMapClient.update(operaName, author);
( 5 ) deleteAuthor
sqlMapClient.delete("deleteAuthor", new Integer(authorID));
```

这里只是做一个简单的例子，希望能够帮助快速的入门，而并没有对IBatis的原理进行剖析，不过通过这几个调用，我想你可能能够猜到IBatis的一部分运作原理了，关于IBatis的原理以及高级应用，请关注后续文章。

本文引用通告地址：

<http://blog.csdn.net/sunsnow8/services/trackbacks/246575.aspx>

## [Ibatis2.0使用说明\(二\)——配置篇\(1\)](#)

### 一、SQL Map XML 配置文件

SQL Map 使用XML 配置文件集中的配置不同的设置属性，包括 DataSource 的详细配置信息，SQL Map 和其他可选属性，如线程管理等。以下是SQL Map 配置文件的一个例子：

SqlMapConfig.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE sqlMapConfig
PUBLIC "-//iBATIS.com//DTD SQL Map Config 2.0//EN"
"http://www.ibatis.com/dtd/sql-map-config-2.dtd">
<!-- Always ensure to use the correct XML header as above! -->
<sqlMapConfig>
<!-- The properties (name=value) in the file specified here can be used
placeholders in this config
file (e.g. "${driver}". The file is relative to the classpath and is
completely optional. -->
<properties resource="
examples/sqlmap/maps/SqlMapConfigExample.properties " />
<!-- These settings control SqlMapClient configuration details,
primarily to do with transaction
management. They are all optional (more detail later in this
document). -->
<settings
cacheModelsEnabled="true"
enhancementEnabled="true"
lazyLoadingEnabled="true"
maxRequests="32"
maxSessions="10"
maxTransactions="5"
useStatementNamespaces="false"
/>
<!-- Type aliases allow you to use a shorter name for long fully
```

qualified class names. -->

```
<typeAlias alias="order" type="testdomain.Order"/>
```

<!-- Configure a datasource to use with this SQL Map using SimpleDataSource.

Notice the use of the properties from the above resource -->

```
<transactionManager type="JDBC" >
```

```
<dataSource type="SIMPLE">
```

```
<property name="JDBC.Driver" value="{driver}"/>
```

```
<property name="JDBC.ConnectionURL" value="{url}"/>
```

```
<property name="JDBC.Username" value="{username}"/>
```

```
<property name="JDBC.Password" value="{password}"/>
```

```
<property name="JDBC.DefaultAutoCommit" value="true" />
```

```
<property name="Pool.MaximumActiveConnections" value="10"/>
```

```
<property name="Pool.MaximumIdleConnections" value="5"/>
```

```
<property name="Pool.MaximumCheckoutTime" value="120000"/>
```

```
<property name="Pool.TimeToWait" value="500"/>
```

```
<property name="Pool.PingQuery" value="select 1 from  
ACCOUNT"/>
```

```
<property name="Pool.PingEnabled" value="false"/>
```

```
<property name="Pool.PingConnectionsOlderThan" value="1"/>
```

```
<property name="Pool.PingConnectionsNotUsedFor" value="1"/>
```

```
</dataSource>
```

```
</transactionManager>
```

<!-- Identify all SQL Map XML files to be loaded by this SQL map.

Notice the paths

are relative to the classpath. For now, we only have one... -->

```
<sqlMap resource="examples/sqlmap/maps/Person.xml" />
```

```
</sqlMapConfig>
```

下面详细讨论SQL Map 配置文件的各组成部分。

(一) <properties>元素

SQL Map 配置文件拥有唯一的<properties>元素，用于在配置文件中使⤵用标准的Java属性文件（name = value）。在属性文件中定

义的属性可以作为变量在SQL Map 配置文件及其包含的所有SQL Map 映射文件中引用。

例如，如果属性文件中包含属性：`driver=org.hsqldb.jdbcDriver`

SQL Map 配置文件及其每个映射文件都可以使用占位符`${driver}`来代表值`org.hsqldb.jdbcDriver`。

例如：`<property name="JDBC.Driver" value="${driver}"/>`

这个元素在开发，测试和部署各阶段都很有用。它可以使在多个不同的环境中重新配置应用和使用自动生成工具（如ANT）变得容易。

Java属性文件可以通过类路径导入，也可以通过有效的URL导入。

例如：`<properties url="file:///c:/config/my.properties" />`

## （二）<setting>元素

<setting>元素用于配置和优化利用XML配置文件创建的SqlMapClient实例。<settings>元素及其所有的属性都是可选的。下面列出了<setting>元素支持的属性和功能：

### 1. maxExecute

同时执行一个Sql statement的最大线程数，大于这个值的线程将阻塞直到另一个线程退出。不同的DBMS 有不同的限制值。

例子：`maxExecute="256"`

缺省值：512

一般情况下，这个值要大于10，而且要同时比maxSessions和maxTransactions参数设定的值大。通常情况下，减少最大同时访问次数可以提高执行的效率。

### 2. maxSessions

是指在一个给定时间内处于活动状态的session（或客户端）的数量。这个值一般要大于或等于maxTransactions的参数值，同时要小于maxRequests的参数值

例子：maxSessions="64"

缺省值：128

### 3. maxTransaction

同时进入SqlMapClient.startTransaction()的最大线程数。大于这个值的线程将阻塞直到另一个线程退出。不同的DBMS有不同的限制值。这个值应该总是小于或等于maxSessions，而且要比maxRequests小的多。通常情况下，减少这个值可以提高执行的效率。

例子：maxTransaction="16"

缺省值：32

### 4. cacheModelsEnabled

启用或禁用SqlMapClient所有的cache models。调试程序时有用。

例子：cacheModelsEnabled="true"

缺省值：true

### 5. lazyLoadingEnabled

启用或禁用SqlMapClient所有的lazy loading。调试程序时有用。

例子：lazyLoadingEnabled="true"

缺省值：true

### 6. enhancementEnabled

This setting enables runtime bytecode enhancement to facilitate optimized JavaBean property access as well as enhanced lazy loading.

例子 enhancementEnabled="true"

缺省值：false (disabled)

### 7. useStatementNamespaces

这个选项如果启用，你就必须使用全限定名来引用mapped statements，这个全名是由sqlMap的名字和statement的名字组成

的。

例如：`queryForObject("sqlMapName.statementName");`

例子：`useStatementNamespaces="false"`

缺省值：`false (disabled)`

### （三）<typeAlias>元素

<typeAlias>允许你指定别名。这样你就可以通过你指定的短名字来代替冗长的名字了。

例如：`<typeAlias alias="shortname"`

`type="com.long.class.path.Class"/>`

下面是在SqlMap中预定义的别名：

#### 1. Transaction Manager Aliases

JDBC `com.ibatis.sqlmap.engine.transaction.jdbc.JdbcTransactionCon`

JTA `com.ibatis.sqlmap.engine.transaction.jta.JtaTransactionConfig`

EXTERNAL `com.ibatis.sqlmap.engine.transaction.external.ExternalTr`

#### 2. Data Source Factory Aliases

SIMPLE `com.ibatis.sqlmap.engine.datasource.SimpleDataSourceFact`

DBCP `com.ibatis.sqlmap.engine.datasource.DbcpDataSourceFactory`

JNDI `com.ibatis.sqlmap.engine.datasource.JndiDataSourceFactory`

### （四）<transactionManager>元素

注意：在Sql Map 1.0版本中，允许同时配置多个数据源，这样会导致一些问题。所以从2.0的版本开始，只允许配置一个数据源，如果你有配置多个数据源的需求的话，建议你使用多个不同配置的属性文件，或者在build Sql Map的时候，作为一个参数传进来。

<transactionManager>元素允许你为SQL Map配置事务管理服务。

type属性用来指明使用哪种事务管理，值既可以是一个类名字，也可以是一个别名。

在框架中已经包含了三种事务管理：JDBC, JTA 和 EXTERNAL

## 1. JDBC

允许用JDBC通过Connection的commit()和rollback()方法来控制事务。

## 2. JTA

这种事务管理使用一个JTA的全局事务将SQL Map的activities作为一个wider scope事务的一部分而包含进来，这个事务可能包含其他的数据库或事务源。这个配置需要一个UserTransaction属性来通过JNDI resource设置本地的user transaction。

## 3. EXTERNAL

允许你自己管理事务。你仍然可以配置一个数据源，但是事务不会在框架生命周期中被提交或回滚。这意味着你必须要用自己的程序来控制事务。这个设置对于非事务型数据库是非常有用的。

### (五) <datasource>元素

<datasource>标签及其属性是<transactionManager>的配置的一部分，它也是用于配置你的SQL Map所使用的数据源的。

目前本框架提供三种数据源工厂，但是你也可以自己写一个。下面是每一种数据源工厂的配置举例：

#### 1. SimpleDataSourceFactory

SimpleDataSourceFactory 为池化的DataSource提供了一个基本的实现，适用于在没有容器提供数据源的情况。

```
<transactionManager type="JDBC">
  <dataSource type="SIMPLE">
    <property name="JDBC.Driver" value="org.postgresql.Driver"/>
    <property name="JDBC.ConnectionURL"
value="jdbc:postgresql://server:5432/dbname"/>
    <property name="JDBC.Username" value="user"/>
    <property name="JDBC.Password" value="password"/>
  </dataSource>
</transactionManager>
```

```

<!-- OPTIONAL PROPERTIES BELOW -->
<property name="Pool.MaximumActiveConnections" value="10"/>
<property name="Pool.MaximumIdleConnections" value="5"/>
<property name="Pool.MaximumCheckoutTime" value="120000"/>
<property name="Pool.TimeToWait" value="10000"/>
<property name="Pool.PingQuery" value="select * from dual"/>
<property name="Pool.PingEnabled" value="false"/>
<property name="Pool.PingConnectionsOlderThan" value="0"/>
<property name="Pool.PingConnectionsNotUsedFor" value="0"/>
</dataSource>
</transactionManager>

```

## 2. DbcpDataSourceFactory

DbcpDataSourceFactory 实现使用Jakarta DBCP ( Database Connection Pool ) 的DataSource

API 提供连接池服务。适用于Web 容器不提供DataSource 服务的情况，或执行一个单独的

应用。 DbcpDataSourceFactory 中必须要配置的参数例子如下：

```

<transactionManager type="JDBC">
  <dataSource type="DBCP">
    <property name="JDBC.Driver" value="{driver}"/>
    <property name="JDBC.ConnectionURL" value="{url}"/>
    <property name="JDBC.Username" value="{username}"/>
    <property name="JDBC.Password" value="{password}"/>
    <!-- OPTIONAL PROPERTIES BELOW -->
    <property name="Pool.MaximumActiveConnections" value="10"/>
    <property name="Pool.MaximumIdleConnections" value="5"/>
    <property name="Pool.MaximumWait" value="60000"/>
    <!-- Use of the validation query can be problematic.
    If you have difficulty, try without it. -->
    <property name="Pool.ValidationQuery" value="select * from
ACCOUNT"/>

```

```
<property name="Pool.LogAbandoned" value="false"/>
<property name="Pool.RemoveAbandoned" value="false"/>
<property name="Pool.RemoveAbandonedTimeout"
value="50000"/>
</datasource>
</transactionManager>
```

### 3. JndiDataSourceFactory

JndiDataSourceFactory 在应用服务器的容器中从JNDI Context 中查找DataSource 实现。当使用应用服务器，并且服务器提供了容器管理的连接池和相关的DataSource 实现的情况下，可以使用JndiDataSourceFactory。使用JDBC DataSource 的标准方法是通过JNDI 来查找。

JndiDataSourceFactory必须要配置的属性如下：

```
<transactionManager type="JDBC" >
<dataSource type="JNDI">
<property name="DataSource"
value="java:comp/env/jdbc/jpetstore"/>
</dataSource>
</transactionManager>
```

注意：上面的配置是使用标准的JDBC事务管理。但是，在一个容器管理的数据源中，你也可能想为全局的事务做如下配置：

```
<transactionManager type="JTA" >
<property name="UserTransaction"
value="java:ctx/con/UserTransaction"/>
<dataSource type="JNDI">
<property name="DataSource"
value="java:comp/env/jdbc/jpetstore"/>
</dataSource>
</transactionManager>
```

注意：UserTransaction属性指向一个JNDI的位置，你可以通过这个JNDI找到一个UserTransaction实例。这个在JTA的事务管理中是需要的，这样可以使你的SQL MAP参与到包含有其他数据库和事务源的事务中。

#### （六）<sql-map>元素

sqlMap元素用于包括SQL Map 映射文件和其他的SQL Map 配置文件。每个SqlMapClient对象使用的SQL Map 映射文件都要在此声明。映射文件作为stream resource 从类路径或URL中读入。您必须相对于类路径或URL来指定所有的SQL Map 文件。下面是几个例子：

```
<!-- CLASSPATH RESOURCES -->
<sqlMap resource="com/ibatis/examples/sql/Customer.xml" />
<sqlMap resource="com/ibatis/examples/sql/Account.xml" />
<sqlMap resource="com/ibatis/examples/sql/Product.xml" />

<!-- URL RESOURCES -->
<sqlMap url="file:///c:/config/Customer.xml" />
<sqlMap url="file:///c:/config/Account.xml" />
<sqlMap url="file:///c:/config/Product.xml" />
```

本文引用通告地址：

<http://blog.csdn.net/sunsnow8/services/trackbacks/246576.aspx>

## Ibatis2.0使用说明(二)——配置篇(2)

# sqlMap所包含的标签：

```
<sqlMap id="Product">

    <cacheModel id="productCache" type="LRU">

        <flushInterval hours="24"/>

        <property name="size" value="1000" />

    </cacheModel>

    <typeAlias alias="product" type="com.ibatis.example.Product" />

    <parameterMap id="productParam" class="product">

        <parameter property="id"/>

    </parameterMap>

    <resultMap id="productResult" class="product">

        <result property="id" column="PRD_ID"/>

        <result property="description" column="PRD_DESCRIPTION"/>

    </resultMap>

    <select id="getProduct" parameterMap="productParam"
resultMap="productResult" cacheModel="product-cache">

        select * from PRODUCT where PRD_ID = ?

    </select>

</sqlMap>
```

## The SQL Map XML File

( <http://www.ibatis.com/dtd/sql-map-config-2.dtd>)

### 一、 Mapped Statements

Mapped statements可以是任何一个SQL statement，并且可以指定输入参数的map和输出结果的map。

简单的情况下，mapped statement可以直接指定一个类来作为输入参数和输出结果参数。mapped statement也可以使用cache model在内存中缓冲经常使用的数据。

```
<statement          id="statementName"

                    [parameterClass="some.class.Name"]

                    [resultClass="some.class.Name"]

                    [parameterMap="nameOfParameterMap"]

                    [resultMap="nameOfResultMap"]

                    [cacheModel="nameOfCache"]

>

    select * from PRODUCT where PRD_ID = [?|#propertyName#] order by
    [$simpleDynamic$]

</statement>
```

在上面的statement的配置中，放在[]中的配置参数是可选的，所以下面的Mapped Statement是完全合法的。

```
<statement id="insertTestProduct">
```

```
    insert into PRODUCT (PRD_ID, PRD_DESCRIPTION) values (1, "Shih  
Tzu")
```

```
</statement>
```

下面逐一介绍各个标签的含义以及使用方法：

# 1. Statement 类型

下表中列出了所有的Statement，以及他们的属性和所支持的特征。

Statement Element	Attributes	Child Elements	Methods
<statement>	id	All dynamic elements	insert
	parameterClass		update
	resultClass		delete
	parameterMap		All query methods
	resultMap		
	cacheModel		
	xmlResultName		
<insert>	id	All dynamic elements	insert
	parameterClass		update
	parameterMap	<selectKey>	delete
<update>	id	All dynamic elements	insert
	parameterClass		update
	parameterMap		delete

<delete>	id	All dynamic elements	insert
	parameterClass		update
	parameterMap		delete
<select>	id	All dynamic elements	All query methods
	parameterClass		
	resultClass		
	parameterMap		
	resultMap		
	cacheModel		
<procedure>	id	All dynamic elements	insert
	parameterClass		update
	resultClass		delete
	parameterMap		All query methods
	resultMap		
	xmlResultName		

<statement id="statementName"

[parameterClass="some.class.Name"]

[resultClass="some.class.Name"]

[parameterMap="nameOfParameterMap"]

[resultMap="nameOfResultMap"]

[cacheModel="nameOfCache"]

>

select \* from PRODUCT where PRD\_ID = [?  
|#propertyName#] order by [\$simpleDynamic\$]

</statement>

## 2. the SQL

sql语句无疑是map中最重要的组成部分，你可以使用任何的sql语句，只要你的数据库和JDBC驱动支持就行。你也可以使用数据库和驱动支持的任何函数。因为你是将sql语句写在了XML文档中，为了区别Sql中的"<>"和XML中的"<>"，在写Sql时，可以用<![CDATA[ SQL ]]>来写。

### 3. 自增

很多关系型数据库都支持主键的自增，SQL Map可以通过<insert>标签中的<selectKey>来支持自增。预生成（如Oracle）和后生成（如Sql Server）都支持，下面是例子：

```
<!--Oracle SEQUENCE Example -->
```

```
<insert id="insertProduct-ORACLE"  
parameterClass="com.domain.Product">
```

```
    <selectKey resultClass="int" keyProperty="id" >
```

```
        SELECT STOCKIDSEQUENCE.NEXTVAL AS ID FROM  
DUAL
```

```
    </selectKey>
```

```
        insert into PRODUCT (PRD_ID,PRD_DESCRIPTION) values  
(#id#,#description#)
```

```
</insert>
```

```
<!-- Microsoft SQL Server IDENTITY Column Example -->
```

```
<insert id="insertProduct-MS-SQL"  
parameterClass="com.domain.Product">
```

```
    insert into PRODUCT (PRD_DESCRIPTION) values
```

(#description#)

```
<selectKey resultClass="int" keyProperty="id" >
```

```
    SELECT @@IDENTITY AS ID
```

```
</selectKey>
```

```
</insert>
```

## 4. 存储过程：

通过<procedure>标签来支持存储过程，下面的例子显示了如何使用带有输出参数的存储过程。

```
<parameterMap id="swapParameters" class="map" >
    <parameter property="email1" jdbcType="VARCHAR"
javaType="java.lang.String" mode="INOUT"/>
    <parameter property="email2" jdbcType="VARCHAR"
javaType="java.lang.String" mode="INOUT"/>
</parameterMap>

<procedure id="swapEmailAddresses"
parameterMap="swapParameters" >
    {call swap_email_address (?, ?)}
</procedure>
```

调用上面的存储过程会在数据表的两列间交换Email地址，而且在对应的参数对象中的也会交换。记住：当parameter mapping的mode为INOUT或OUT的时候，

你输入的参数对象才会改变。很显然，不变的参数对象也是不会改变的，比如说String对象。

记住：一定要使用标准的JDBC存储过程的语法。请参见JDBC CallableStatement documentation以获取更多信息。

本文引用通告地址：

<http://blog.csdn.net/sunsnow8/services/trackbacks/246578.aspx>

## Ibatis2.0使用说明(二)——配置篇 (3)

# statement中的参数简介：

## 1. parameterClass

parameterClass 属性的值是Java类的全限定名（即包括类的包名）。parameterClass属性是可选的，目的是限制输入参数的类型为指定的Java类。虽然Parameter-class属性是可选的，建议你为每一个SQL都指定parameterClass。如果不指定parameterClass参数，任何带有合适属性（get/set方法）的Java Bean都可以作为输入参数。如果你使用了parameterMap，那么你就不需要再使用parameterClass属性了。

下面是例子：

例1：

```
<insert id="insertAuthor2" parameterClass="Author">
```

```
    INSERT INTO author  
(auth_name,auth_age,auth_tel,auth_address) VALUES  
(#name#,#age#,#telephone#,#address#)
```

```
</insert>
```

在上面的语句中，你指定的parameterClass=Author，那么在你的Author类中要有name,age,telephone和address属性，并且要有相应的get和set方法

例2：

你可以使用基本类型作为parameterClass，如：

```
<delete id="deleteAuthor" parameterClass="int">
```

```
    delete from author WHERE auth_id = #id#
```

```
</delete>
```

例3：

你可以使用HashMap作为parameterClass，如：

```
<insert id="insertAuthor3" parameterClass="java.util.HashMap">
```

```
    INSERT INTO author  
(auth_name,auth_age,auth_tel,auth_address) VALUES  
(#name#,#age#,#telephone#,#address#)
```

```
</insert>
```

这时候，在你调用insertAuthor3的时候，你首先应该给传入的Map对象赋值，调用代码如下：

```
HashMap paramMap = new HashMap();  
  
paramMap.put("name", "作者三");  
  
paramMap.put("age", new Integer(31));  
  
paramMap.put("address", "南京");  
  
paramMap.put("telephone", "025-987654321");  
  
sqlMapClient.insert("insertAuthor3", paramMap);
```

## 2. parameterMap

parameterMap 定义一系列有次序的参数用于匹配PreparedStatement 的JDBC值符号。

parameterMap属性很少使用，parameterMap 属性的值等于指定的parameterMap元素的name属性值。通常（和缺省的）的方法是使用inline parameters。

注意！动态mapped statement 只支持inline parameter，不支持parameter map。关于动态mapped statement，将在后文中介绍。

例如：

```
<parameterMap id="authorParameter" class="Author">
    <parameter property="name" jdbcType="VARCHAR"
javaType="java.lang.String" mode="INOUT"/>
    <parameter property="age" jdbcType="INTEGER"
javaType="java.lang.Integer" mode="INOUT"/>
    <parameter property="telephone" jdbcType="VARCHAR"
javaType="java.lang.String" mode="INOUT"/>
    <parameter property="address" jdbcType="VARCHAR"
javaType="java.lang.String" mode="INOUT"/>
</parameterMap>
```

```
<insert id="insertAuthor1" parameterMap="authorParameter">
```

```
    INSERT INTO author  
(auth_name,auth_age,auth_tel,auth_address) VALUES (?, ?, ?, ?)
```

```
</insert>
```

上面的例子中，parameterMap的参数按次序匹配SQL语句中的值符号(?)。因此，第一个"?"号将被"name"属性的值替换，而第二个"?"号将被"age"属性的值替换，依此类推。

记住：使用parameterMap的时候，SQL中的参数用"?"来代替，并且每个"?"的顺序要与parameterMap中的定义完全匹配；如果使用parameterClass，那么SQL中的参数用"#parameterName#"来代替，如果传入的参数类为Bean，那么要有get和set这个参数名的方法。

### 3. resultClass

resultClass 属性可以让您指定一个Java 类，根据ResultSetMetaData 将其自动映射到JDBC ResultSet。只要是JavaBean 的属性、方法名称和ResultSet的列名匹配，属性自动赋值列值。

例1：

```
<select id="getAuthor1" parameterClass="int" resultClass="Author">
```

```
    SELECT auth_id as id,auth_name as name,auth_age as  
    age,auth_tel as telephone,auth_address as address FROM author  
    WHERE auth_id = #id#
```

```
</select>
```

在上面的语句中，你指定的resultClass=Author，那么在你的Author类中要有id,name,age,telephone和address属性，并且要有相应的get和set方法。

如果你写成：

```
<select id="getAuthor1" parameterClass="int" resultClass="Author">
```

```
    SELECT auth_id,auth_name,auth_age,auth_tel,auth_address  
    FROM author WHERE auth_id = #id#
```

```
</select>
```

那么在你的Author类中，要

有auth\_id,auth\_name,auth\_age,auth\_tel,auth\_address属性，并且要有相应的get和set方法。

例2：

你还可以使用基本类型作为resultClass，如：

```
<statement id="getAuthorNumber" resultClass="int">  
    <![CDATA[SELECT count(auth_id) as totalAuthor FROM  
author]]>  
</statement>
```

例3：

你还可以使用HashMap作为resultClass，如：

```
<select id="getAuthor4" resultClass="java.util.HashMap">  
    SELECT a.auth_id as authorid,a.auth_name as  
    authname,a.auth_age as authage,a.auth_tel as  
    authtel,a.auth_address as authaddress,b.art_title as arttitle FROM  
    author a, article b WHERE a.auth_id=b.art_author  
</select>
```

下面是调用代码：

```
List authorList = (List)sqlMapClient.queryForList("getAuthor4",null);  
showMethod("getAllAuthor");  
for(int i=0;i<authorList.size();i++)
```

```
{  
  
    HashMap authMap = (HashMap)authorList.get(i);  
  
    System.out.println("auth_id="+authMap.get("authid")+";  
auth_name="+authMap.get("authname")+";  
auth_age="+authMap.get("authage")+";  
auth_tel="+authMap.get("authtel")+";  
auth_address="+authMap.get("authaddress")+"; auth_article="+au  
  
}
```

但是，使用resultClass 的自动映射存在一些限制，无法指定输出字段的数据类型，无法自动装入相关的数据（复杂属性），并且因为需要ResultSetMetaData的信息，会对性能有轻微的不利影响。但使用resultMap，这些限制都可以很容易解决。

## 4. resultMap:

使用resultMap 属性可以控制数据如何从结果集中取出，以及哪一个属性匹配哪一个字段。不象上面使用resultClass 属性的自动映射方法，resultMap属性可以允许指定字段的数据类型，NULL的替代值。

例如：

```
<resultMap id="authorResult" class="Author">
    <result property="id" column="auth_id"/>
    <result property="age" column="auth_age"/>
    <result property="name" column="auth_name"/>
    <result property="telephone" column="auth_tel"/>
    <result property="address" column="auth_address"/>
</resultMap>
```

```
<select id="getAuthor3" resultMap="authorResult">
    SELECT auth_id,auth_name,auth_age,auth_tel,auth_address
    FROM author WHERE auth_address like #0address0#
</select>
```

或

```
<statement id="getAllAuthor" resultMap="authorResult">
```

```
    SELECT * FROM author
```

```
</statement>
```

在上面的语句中，你指定的resultClass=Author，那么在你的Author类中要有id,name,age,telephone和address属性，并且要有相应的get和set方法。

通过resultMap的定义，查询语句得到的ResultSet被映射成Author对象。resultMap定义"auth\_id"属性值将赋予"auth\_id"字段值，而"telephone"属性值将赋予"auth\_tel"字段值，依次类推。

注意：在resultMap中所指定的字段必须是下面的select中的子集。

也就是说，你不能写成

```
<select id="getAuthor3" resultMap="authorResult">
```

```
    SELECT auth_address FROM author WHERE auth_address like  
    #%address%#
```

```
</select>
```

但是你可以在resultMap中去掉<result property="id" column="auth\_id"/>这行，仍然可以执行下面的语句：

```
<select id="getAuthor3" resultMap="authorResult">
```

```
    SELECT auth_id,auth_name,auth_age,auth_tel,auth_address  
    FROM author WHERE auth_address like #address%#
```

```
</select>
```

这样的话，你就无法取得auth\_id的值。

## 5. cacheModel

定义查询mapped statement 的缓存。每一个查询mapped statement 可以使用不同或相同的cacheModel。

```
<cacheModel id="author-cache" imlementation="LRU">
    <flushInterval hours="24"/>
    <flushOnExecute statement="insertProduct"/>
    <flushOnExecute statement="updateProduct"/>
    <flushOnExecute statement="deleteProduct"/>
    <property name="size" value="1000" />
</cacheModel>

<select id="getAuthor3" resultMap="authorResult"
cacheModel="author-cache">
    SELECT auth_id,auth_name,auth_age,auth_tel,auth_address
FROM author WHERE auth_address like #%address%#
</select>
```

上面的配置说明：“getAuthor3”的缓存使用WEAK引用类型，当你通过调用“getAuthor3”的时候，Ibatis将会把结果缓存起来。每24小时缓存刷新一次，或当更新的操作(即上面配置的insertProduct、updateProduct或deleteProduct)发生时刷新。

当你对某些表中的记录操作频繁时，可以考虑使用缓冲，但是如果数据量过大的话，最好另想办法。

## 6. xmlResultName

当映射结果指向一个XML文档的时候，xmlResultName的值是指那个XML文档的root标签的名字。例如：

```
<select id="getAuthor1" parameterClass="int" resultClass="Author"
xmlResultName="author">
```

```
    SELECT auth_id as id,auth_name as name,auth_age as
    age,auth_tel as telephone,auth_address as address FROM author
    WHERE auth_id = #id#
```

```
</select>
```

上面的select将产生如下的XML对象：

```
<author>
```

```
    <id>1</id>
```

```
    <name>作者三</name>
```

```
    <age>31</age>
```

```
    <telephone>025-987654321</telephone>
```

```
    <address>南京</address>
```

```
</author>
```

本文引用通告地址：

<http://blog.csdn.net/sunsnow8/services/trackbacks/246579.aspx>

## Ibatis2.0使用说明(二)——配置篇(4)

## 五、Parameter Maps and Inline Parameters

```
<parameterMap id="parameterMapName" [class="Author"]>  
    <parameter property ="propertyName" [jdbcType="VARCHAR"]  
    [javaType="string"]  
    [nullValue="NUMERIC"] [null="-9999999"]/>  
    <parameter ..... />  
    <parameter ..... />  
</parameterMap>
```

括号[]中是可选的属性。parameterMap 元素的id 属性作为唯一标识，在同一个SQL Map XML 文件中不能重名。一个parameterMap 可包含任意多的property 元素。

## (一) property

property属性是指传入mapped statement中的JavaBean参数对象的属性名。这个属性名可以使用多次，这要看在这个statement中，这个属性名要出现多少次。例如：

```
<parameterMap id="authorParameter3" class="Author">
```

```
    <parameter property="name" jdbcType="VARCHAR"  
    javaType="java.lang.String" mode="INOUT"/>
```

```
    <parameter property="name" jdbcType="VARCHAR"  
    javaType="java.lang.String" mode="INOUT"/>
```

```
</parameterMap>
```

```
<update id="updateAuthor2" parameterMap="authorParameter2">
```

```
    UPDATE author set auth_name=? WHERE auth_name = ?
```

```
</update>
```

但是如果使用这样的方法的话，调用代码应该是：

```
Author author = new Author();
```

```
author.setName("作者三");
```

```
sqlMapClient.update("updateAuthor2", paraMap);
```

那么它其实执行的是：

```
UPDATE author set auth_name='作者三' WHERE auth_name = '作者三'
```

这样的话，就根本没有了意义，因为，你只能传进一个Author对象，而这个Author对象的name属性值将会被用在整个Sql语句中，而一般的情况下是不应该相同的，也就是说，我们的本意可能是想：

```
UPDATE author set auth_name='作者N' WHERE auth_name = '作者三'
```

方法倒是有的，不过我觉得不太好。

```
<parameterMap id="authorParameter2" class="java.util.HashMap">
```

```
    <parameter property="name1" jdbcType="VARCHAR"
    javaType="java.lang.String" mode="INOUT"/>
```

```
    <parameter property="name2" jdbcType="VARCHAR"
    javaType="java.lang.String" mode="INOUT"/>
```

```
</parameterMap>
```

```
<update id="updateAuthor2" parameterMap="authorParameter2">
```

```
    UPDATE author set auth_name=? WHERE auth_name = ?
```

```
</update>
```

调用代码为：

```
HashMap paraMap = new HashMap();
```

```
paraMap.put("name1", "作者N");
```

```
paraMap.put("name2", "作者三");
```

```
sqlMapClient.update("updateAuthor2", paraMap);
```

如果你想到更好的方法解决这个问题，请不吝赐教。

## (二) jdbcType

jdbcType用于指明数据库的字段类型。如果不说明字段类型的话，一些JDBC驱动程序就无法确定要操作的字段类型。例如：PreparedStatement.setNull(int parameterIndex, int sqlType)方法，要求指定数据类型。如果不指定数据类型，某些Driver可能指定为Types.Other或Types.Null。但是，不能保证所有的Driver都表现一致。对于这种情况，SQL Map API允许使用parameterMap元素的jdbcType属性指定数据类型。

正常情况下，只有当字段可以为NULL或日期时间类型时才需要type属性。因为Java只有一个Date类型（java.util.Date），而大多数SQL数据库有多个 - 通常至少有3种。因此，需要指定字段类型是DATE还是DATETIME。

Type属性可以是JDBC Types类中定义的任意参数的字符串值。虽然如此，还是有某些类型不支持（即BLOB）。

注意！大多数JDBC Driver只有在字段可以为NULL时需要指定type属性。因此，对于这些Driver，只是在字段可以为NULL时才需要指定type属性。

注意！当使用Oracle Driver时，如果没有给可以为NULL的字段指定type属性，当试图给这些字段赋值NULL时，会出现"Invalid column. type"错误。

### (三) javaType

javaType用于指明作为参数传递的java bean的属性的类型。通常情况下，这可以通过反射机制从java bean中获取类型，但是一些特定的映射，比如说MAP和XML的映射就无法将类型信息传递给框架了。如果java type没有设置而且框架无法获知类型的话，那么这个类型会被指定为Object。

## (四) nullValue

属性 nullValue 的值可以是对于 property 类型来说合法的任意值，用于指定 NULL 的替换值。就是说，当 Java Bean 的属性值等于指定值时，相应的字段将赋值 NULL。这个特性允许在应用中给不支持 null 的数据类型（即 int，double，float 等）赋值 null。当这些数据类型的属性值匹配 nullValue 值（即匹配 -9999）时，NULL 将代替 nullValue 值写入数据库。

例如：

```
<parameterMap id="authorParameter" class="Author">
```

```
    <parameter property="name" jdbcType="VARCHAR"
javaType="java.lang.String" nullValue="NO_ENTRY"
mode="INOUT"/>
```

```
    <parameter property="age" jdbcType="INTEGER"
javaType="java.lang.Integer" nullValue="-999" mode="INOUT"/>
```

```
    <parameter property="telephone" jdbcType="VARCHAR"
javaType="java.lang.String" nullValue="NO_ENTRY"
mode="INOUT"/>
```

```
    <parameter property="address" jdbcType="VARCHAR"
javaType="java.lang.String" nullValue="NO_ENTRY"
mode="INOUT"/>
```

```
</parameterMap>
```

```
<insert id="insertAuthor1" parameterMap="authorParameter">
```

```
    INSERT INTO author  
(auth_name,auth_age,auth_tel,auth_address) VALUES (?,?,,?)
```

```
</insert>
```

您可以在另一个SQL Map XML 文件中引用parameterMap。例如，要在另一个文件中引用上面的parameterMap，可以使用名称"Product.insert-product-param"。

## (五) Inline Parameter Maps

使用Inline Parameter Maps，可以把Java Bean的属性名称嵌在mapped-statement的定义中（即直接写在SQL语句中）。

例如：

```
<insert id="insertAuthor1" parameterClass="Author">
```

```
    INSERT INTO author  
    (auth_name,auth_age,auth_tel,auth_address) VALUES  
    (#name#,#age#,#telephone#,#address#)
```

```
</insert>
```

这样，在你的Author类中，要有name,age,telephone,address的属性以及相应的get和set方法，这样做可以避免使用另外定义parameterMap的麻烦。

你也可以在内嵌参数中指定数据类型和nullValue，例如：

```
<insert id="insertAuthor1" parameterClass="Author">
```

```
    INSERT INTO author  
    (auth_name,auth_age,auth_tel,auth_address) VALUES  
    (#name:VARCHAR:NO_ENTRY#,#age:INTEGER:-999#,#telepho
```

```
</insert>
```

注意！在内嵌参数中，要指定NULL的替代值，必须要先指定数据类型。

注意！如需要在查询时也使用NULL替代值，必须同时在resultMap中定义。

注意！如果您需要指定很多的数据类型和NULL替代值，可以使用外部的parameterMap元素，这样会使代码更清晰。

## 六、Result Maps

在SQL Map 架构中，Result Map 是极其重要的组件。在执行查询Mapped Statement 时，resultMap 负责将结果集的列值映射成Java Bean 的属性值。resultMap 的结构如下：

```
<resultMap id="resultMapName" class="some.domain.Class"
[extends="parent-resultMap"]>

    <result property="propertyName" column="COLUMN_NAME"

        [columnIndex="1"] [javaType="int"]
[jdbcType="NUMERIC"]

        [nullValue="-999999"] [select="someOtherStatement"]

    />

<result ...../>

<result ...../>

<result ...../>

</resultMap>
```

括号[]中是可选的属性resultMap 也有class 属性，是Java 类的全限定名（即包括包的名称）或该类的别名。该Java 类初始化并

根据定义填充数据。

Extends 是可选的属性，可以设定成以为基础的另外一个resultMap 的名字。和在Java 中继承一个类相似，父resultMap 的属性将作为子resultMap 的一部分。父resultMap 的属性总是加到子resultMap 属性的前面，并且父resultMap 必须要在子resultMap 之前定义。父resultMap 和子resultMap 的class 属性不一定要一致，它们可以没有任何关系。

resultMap 可以包括任意多的property 映射，将查询结果集的列值映射成Java Bean 的属性。属性的映射按它们在resultMap中定义的顺序进行。属性class 必须符合Java Bean 规范，每一属性都必须拥有get/set 方法。

注意！ResultSet 的列值按它们在resultMap 中定义的顺序读取。

## (一) property

property属性是指从mapped statement中返回的JavaBean对象的属性名。这个属性名也可以使用多次。

## (二) column

column属性值是ResultSet中的列名字，即字段名，取得的这个字段的值将赋给property所指的bean属性。

### (三) columnIndex

可选属性，用于改善性能。属性columnIndex 的值是ResultSet 中用于赋值Java Bean属性的字段次序号。在99%的应用中，不太可能需要牺牲可读性来换取性能。使用columnIndex，某些JDBC Driver可以大幅提高性能，某些则没有任何效果。

## ( 四 ) jdbcType

同ParameterMap中的jdbcType

## ( 五 ) javaType

同ParameterMap中的javaType

## (六) nullValue

属性nullValue指定数据库中NULL的替代值。因此，如果从ResultSet中读出NULL值，JavaBean属性将被赋值为属性nullValue指定的替代值。

如果数据库中存在NULLABLE 属性的字段，但您想在您的应用程序中用指定的常量代替NULL，您可以这样做：

```
<resultMap id="get-product-result" class="Author">  
    <result property="id" column="auth_id"/>  
    <result property="age" column="auth_age"/>  
    <result property="name" column="auth_name" nullValue="you  
have no name"/>  
</resultMap>
```

在上例中，如果取得的记录中auth\_name字段的值为NULL，那么在赋给java bean的时候，name属性将被赋为"you have no name"。

## (七) select 复杂属性

如果在一个类与另一个类之间是关联关系的话，那么当你用JDBC取得记录的时候，这个关联关系是如何实现的呢？例如这样的关系：一个作者可能会有多个文章发表，那么作者与文章之间就是很强的关联关系，而且是一对多的关系，在Author的Bean中是这样写的：

```
public class Author
{
    private int id;
    ....
    private List articleList;

    public int getId()
    {
        return id;
    }

    public void setId(int id)
    {
```

```
        this.id=id;
    }

    ... ..

    public List getArticleList()
    {
        return articleList;
    }

    public void setArticleList(List articleList)
    {
        this.articleList=articleList;
    }
}
```

当你执行一条sql语句从数据表author中取出相应的数据的时候，

在上面的java bean中，articleList如何赋值呢？这时候，就需要使用select属性。

## 1 . 1:1关系：

我们先假设在author和article之间使用1:1的关系，虽然在真实世界中是不正确的，我们只是做个例子，那么Author和article的bean代码如下：

```
public class Author
{
    private int id;

    private int age;

    private String name;

    private String address;

    private String telephone;

    private Article article;

    public int getId()
    {
```

```
        return id;
    }

    public void setId(int id)
    {
        this.id=id;
    }

    public int getAge()
    {
        return age;
    }

    public void setAge(int age)
    {
        this.age=age;
    }

    public String getName()
    {
```

```
    return name;
}

public void setName(String name)
{
    this.name=name;
}

public String getAddress()
{
    return address;
}

public void setAddress(String address)
{
    this.address=address;
}

public String getTelephone()
```

```
{
```

```
    return telephone;
```

```
}
```

```
public void setTelephone(String telephone)
```

```
{
```

```
    this.telephone=telephone;
```

```
}
```

```
public Article getArticle()
```

```
{
```

```
    return this.article;
```

```
}
```

```
public void setArticle(Article article)
```

```
{
```

```
    this.article=article;
```

```
    }  
}  
  
public class Article  
{  
    private int id;  
    private String title;  
    private Date createtime;  
    private int author;  
  
    public int getId()  
    {  
        return id;  
    }  
  
    public void setId(int id)  
    {
```

```
    this.id=id;
}

public String getTitle()
{
    return title;
}

public void setTitle(String title)
{
    this.title=title;
}

public Date getCreatetime()
{
    return createtime;
}

public void setCreatetime(Date createtime)
{
```

```
        this.createtime=createtime;
    }

    public int getAuthor()
    {
        return author;
    }

    public void setAuthor(int author)
    {
        this.author=author;
    }
}
```

在author.xml的配置如下：

```
<resultMap id="linkResultMap1" class="Author">
    <result property="id" column="auth_id"/>
    <result property="age" column="auth_age"/>
    <result property="name" column="auth_name"/>
</resultMap>
```

```

    <result property="telephone" column="auth_tel"/>

    <result property="address" column="auth_address"/>

    <result property="article" column="auth_id"
select="getLinkArticle1"/>

</resultMap>

<select id="getAuthor5" resultMap="linkResultMap1"
parameterClass="int">

    SELECT * FROM author WHERE auth_id = #id#

</select>

<select id="getLinkArticle1" resultClass="com.ibatis.beans.Article"
parameterClass="int">

    SELECT art_id as id,art_title as title,art_createtime as
createtime,art_author as author FROM article WHERE art_id = #id#

</select>

```

调用代码如下：

```

Author author =
(Author)sqlMapClient.queryForObject("getAuthor5", new Integer(1));

System.out.println(author.getName()+"'s article is

```

```
:"+author.getArticle().getTitle());
```

你可以看到，对于Author类中的article属性，IBatis是将取得的记录的auth\_id字段值作为参数传入到id="getLinkArticle1"的语句中，并将取得的结果封装成Article对象，并赋给Author的article属性。上面的调用代码实际执行的两条sql语句是：

```
SELECT * FROM author WHERE auth_id = 1
```

```
SELECT art_id as id,art_title as title,art_createtime as  
createtime,art_author as author FROM article WHERE art_id = 1
```

在第二条语句中，将取得的记录封装成Article对象，并赋给Author的article属性，所以，你可以直接使用author.getArticle().getTitle()获得文章的标题。

上面的方法显示了如何实现1:1的关联关系，但是上面的方法并不好，原因是可能会执行很多次查询！

### (1) 避免 N+1 Selects (1:1)

如果上面的配置如下：

```
<resultMap id="linkResultMap1" class="Author">
```

```
    <result property="id" column="auth_id"/>
```

```
    <result property="age" column="auth_age"/>
```

```
    <result property="name" column="auth_name"/>
```

```
<result property="telephone" column="auth_tel"/>

<result property="address" column="auth_address"/>

<result property="article" column="auth_id"
select="getLinkArticle1"/>

</resultMap>

<select id="getAuthor5" resultMap="linkResultMap1"
parameterClass="int">

    SELECT * FROM author WHERE auth_id > #id#

</select>

<select id="getLinkArticle1" resultClass="com.ibatis.beans.Article"
parameterClass="int">

    SELECT art_id as id,art_title as title,art_createtime as
createtime,art_author as author FROM article WHERE art_id = #id#

</select>
```

调用代码如下：

```
Author author =
(Author)sqlMapClient.queryForList("getAuthor5", new
Integer(1));
```

如果SELECT \* FROM author WHERE auth\_id > 1的记录有N条，那么将对id="getLinkArticle1"的语句执行N次查询，这样所有的查询总数将为N+1次，执行效率会很低。

这时，可以使用下面的联合查询的方法来解决：

```
<resultMap id="linkResultMap2" class="Author">
    <result property="id" column="auth_id"/>
    <result property="age" column="auth_age"/>
    <result property="name" column="auth_name"/>
    <result property="telephone" column="auth_tel"/>
    <result property="address" column="auth_address"/>
    <result property="article.title" column="art_title"/>
</resultMap>

<select id="getAuthor6" resultMap="linkResultMap2"
parameterClass="int">
    <![CDATA[ SELECT
a.auth_id,a.auth_age,a.auth_name,a.auth_tel,a.auth_address,b.art_title
FROM author a,article b WHERE a.auth_id > #id# and a.auth_id =
b.art_id]]>
```

```
</select>
```

调用代码为：

```
Author author = (Author)sqlMapClient.queryForList("getAuthor6",  
new Integer(1));
```

这样只用一条Sql语句就可以解决。

## 2 . 1:M与M:N关系：

下面我们讨论1:M的关系，一个author可能有多个article，所以，author与article之间是一对多的关系。那么我们在Author类中加入如下代码(在省略号间的是要加入的代码)：

```
public class Author  
  
{  
  
    ... ..  
  
    private List articleList;  
  
    public List getArticleList()  
  
    {  
  
        return articleList;  
  
    }  
}
```

```
public void setArticleList(List articleList)
{
    this.articleList=articleList;
}
... ..
}
```

配置如下：

```
<resultMap id="linkResultMap3" class="Author">
    <result property="id" column="auth_id"/>
    <result property="age" column="auth_age"/>
    <result property="name" column="auth_name"/>
    <result property="telephone" column="auth_tel"/>
    <result property="address" column="auth_address"/>
    <result property="articleList" column="auth_id"
select="getLinkArticle3"/>
```

```
</resultMap>
```

```
<select id="getAuthor7" resultMap="linkResultMap3"  
parameterClass="int">
```

```
    SELECT * FROM author WHERE auth_id = #id#
```

```
</select>
```

```
<select id="getLinkArticle3" resultClass="com.ibatis.beans.Article"  
parameterClass="int">
```

```
    SELECT art_id as id,art_title as title,art_createtime as  
createtime,art_author as author FROM article WHERE art_author =  
#id#
```

```
</select>
```

调用代码为：

```
Author author = (Author)sqlMapClient.queryForObject("getAuthor7",  
new Integer(1));
```

```
System.out.println(author.getName()+"的文章有：");
```

```
for(int i=0;i<author.getArticleList().size();i++)
```

```
{
```

```
    int num=i+1;
```

```
Article art = (Article)author.getArticleList().get(i);

System.out.println(num+" "+art.getTitle());

}
```

从上面的实现可以看出，你只需要在bean中加入一个java.util.List(或java.util.Collection)类型的articleList来表示所有的文章列表即可，调用部分没有什么变化，IBaits会自动从Article中取得的记录封装成Article对象并加入到一个List对象中，然后将这个List对象赋值给Author类的articleList属性。

### ( 1 ) 避免 N+1 Selects (1:M and M:N)

1:M和M:N的情况与1:1的情况相似，也会出现N+1 Selects 的情况，但是到目前位置，还没有其他的方法来解决这个问题，希望在不久的将来能够解决。

## 3 . 多个复杂参数属性

你可能已经注意到了，上面的例子中，在resultMap中只指明了一个column属性用于id="getLinkArticle"的statement关联。其实，Ibatis允许你指明多个column属性与id="getLinkArticle"的statement关联，语法很简单，{param1=column1, param2=column2, ..., paramN=columnN}。下面是一个例子：

```
<resultMap id="linkResultMap4" class="Author">
```

```
<result property="id" column="auth_id"/>
```

```
<result property="age" column="auth_age"/>
```

```
<result property="name" column="auth_name"/>
```

```
<result property="telephone" column="auth_tel"/>
```

```
<result property="address" column="auth_address"/>
```

```
<result property="articleList" column="{id=auth_id,address=auth_address}"  
select="getLinkArticle4"/>
```

```
</resultMap>
```

```
<select id="getAuthor8" resultMap="linkResultMap4" parameterClass="int">
```

```
    SELECT * FROM author WHERE auth_id = #id#
```

```
</select>
```

```
<select id="getLinkArticle4" resultClass="com.ibatis.beans.Article"  
parameterClass="int">
```

```
    SELECT art_id as id,art_title as title,art_createtime as createtime,art_author as  
author FROM article WHERE art_author = #id# and art_publish_add=#address#
```

```
</select>
```

你也可以只写字段的名称，只要按照所关联的statement中所对应的字段

顺序即可，象这样：

```
{auth_id,auth_address}
```

例子我在Mysql的环境中运行没有通过，报出的错误是：**Column'{auth\_id,auth\_address}' not found.**

这个错误是与JDBC驱动无关的，而是在做XML解析的时候发生的错误，不知道是什么原因，如果您有这样的成功经历的话，希望能共同分享，我的Email是：[tenwang1977@163.com](mailto:tenwang1977@163.com)

注意：有些JDBC驱动不支持同时打开多个ResultSets(单个连接)。所以说，这样的驱动不能完成复杂对象的映射，因为JDBC驱动需要多个ResultSets的连接，这时候，只能使用一个关联查询解决问题。

如果你使用*Microsoft SQL Server 2000*的JDBC驱动的话，你需要在配置url的时候，在url后加上*SelectMethod=Cursor*。

## 4 . 在Parameter Maps and Result Maps中支持的参数

Java Type	JavaBean/Map Property Mapping	Result Class / Parameter Class***	Type Alias**
boolean	YES	NO	boolean
java.lang.Boolean	YES	YES	boolean

byte	YES	NO	byte
java.lang.Byte	YES	YES	byte
short	YES	NO	short
java.lang.Short	YES	YES	short
int	YES	NO	Int/ Integer
java.lang.Integer	YES	YES	Int/ Integer
long	YES	NO	long
java.lang.Long	YES	YES	long
float	YES	NO	float
java.lang.Float	YES	YES	float
double	YES	NO	double
java.lang.Double	YES	YES	double
java.lang.String	YES	YES	string

java.util.Date	YES	YES	date
java.math.BigDecimal	YES	YES	decimal
* java.sql.Date	YES	YES	N/A
* java.sql.Time	YES	YES	N/A
* java.sql.Timestamp	YES	YES	N/A

## 七、缓存Mapped Statement Result

```
<cacheModel id="product-cache" type="LRU" readOnly="true"
serialize="false">
```

```
<flushInterval hours="24"/>
```

```
<flushOnExecute statement="insertProduct"/>
```

```
<flushOnExecute statement="updateProduct"/>
```

```
<flushOnExecute statement="deleteProduct"/>
```

```
<property name="cache-size" value="1000" />
```

```
</cacheModel>
```

上面的cache model 创建了一个名为“product-cache”的缓存，使用“最近最少使用”（LRU）实现，每24小时，缓冲区将刷新一次，而且在执行insertProduct、updateProduct和deleteProduct的statement时，缓冲区也将刷新，设定的时间可以设定为hours, minutes, seconds 或 milliseconds。一些Cache的实现需要附加的属性，比如说上例中的cache-size

属性，cache的大小指明了可以存放在cache中的实体的个数。type属性的名称要么是全限定的类名，要么是缓存实现的别名。Cache Model 使用插件的形式来支持不同的缓存算法。它的

实现在cache-model 元素的type属性中指定（如上所示）。

## (一) Read-Only 与 Read/Write

Ibatis支持只读和可读写的Cache，只读的Cache可以在所有的用户间共享，所以它可以提供更大的操作空间。但是从只读缓冲中读取的对象不能够被修改。如果你要对你取得的对象进行修改的话，那么只能用可读写的缓冲。readOnly="true"为只读缓冲；readOnly="false"为可读写缓冲。

## (二) Serializable Read/Write Caches

要使用Serializable Read/Write Caches，设置readOnly="false"，serialize="true"。默认情况下，采用的是readOnly="true"，serialize="false"。

## (三) 缓冲类型

目前包括以下的4个缓冲类型实现：

### 1. “MEMORY”

(`com.ibatis.db.sqlmap.cache.memory.MemoryCacheController`)

MEMORY cache 实现使用reference 类型来管理cache 的行为。垃圾收集器可以根据reference 类型判断是否要回收cache 中的数据。MEMORY 实现适用于没有统一的对象重用模式的应用，或内存不足的应用。

MEMORY 实现可以这样配置：

```
<cache-model name="product-cache" implementation="MEMORY">
```

```
  <flush-interval hours="24"/>
```

```
  <flush-on-execute statement="insertProduct"/>
```

```
  <flush-on-execute statement="updateProduct"/>
```

```
  <flush-on-execute statement="deleteProduct"/>
```

```
  <cache-property name="reference-type" value="WEAK" />
```

```
</cache-model>
```

MEMORY cache 实现只认识一个<cache-property>元素。这个名为“reference-type”属性的值必须是STRONG，SOFT 和WEAK 三者其一。这三个值分别对应于JVM 不同的内存reference 类型。

### (1) WEAK (缺省)

大多数情况下，WEAK类型是最佳选择。如果不指定类型，缺省类型就是WEAK。它能大大提高常用查询的性能。但是对于当前不被使用的查询结果数据，将被清除以释放内存用来分配其他对象。

### (2) SOFT

在查询结果对象数据不被使用，同时需要内存分配其他对象的情况下，SOFT类型将减少内存不足的可能性。然而，这不是最具侵入性的reference类型，结果数据依然可能被清除。

### (3) STRONG

确保查询结果数据一直保留在内存中，除非Cache被刷新（例如，到了刷新的时间或执行了更新数据的操作）。

对于下面的情况，这是理想的选择：

1” 结果内容数据很少

2” 完全静态的数据

3” 频繁使用的数据

优点是对于这类查询性能非常好。缺点是，如果需要分配其他对象，内存无法释放（可能是更重要的数据对象）。

## 2. “LRU” (com.ibatis.db.sqlmap.cache.lru.LruCacheController)

LRU Cache 实现用“最近最少使用”原则来确定如何从Cache 中清除对象。当Cache溢出时，最近最少使用的对象将被从Cache 中清除。

```
<cache-model name="product-cache" implementation="LRU">  
  
    <flush-interval hours="24"/>  
  
    <flush-on-execute statement="insertProduct"/>  
  
    <flush-on-execute statement="updateProduct"/>  
  
    <flush-on-execute statement="deleteProduct"/>  
  
    <cache-property name="cache-size" value="1000" />  
  
</cache-model>
```

值得注意的是，这里指的对象可以是任意的，从单一的String 对象到Java Bean 的ArrayList 对象都可以。因此，不要Cache 太多的对象，以免内存不足。

## 3. “FIFO” (com.ibatis.db.sqlmap.cache.fifo.FifoCacheController)

FIFO Cache 实现用“先进先出”原则来确定如何从Cache 中清除

对象。对于短时间内持续引用特定的查询而后很可能不再使用的情况，FIFO Cache 是很好的选择。

```
<cache-model name="product-cache" implementation="FIFO">  
    <flush-interval hours="24"/>  
    <flush-on-execute statement="insertProduct"/>  
    <flush-on-execute statement="updateProduct"/>  
    <flush-on-execute statement="deleteProduct"/>  
    <cache-property name="cache-size" value="1000" />  
</cache-model>
```

值得注意的是，这里指的对象可以是任意的，从单一的String对象到Java Bean的ArrayList对象都可以。因此，不要Cache太多的对象，以免内存不足。

#### 4. “OSCACHE”

**(com.ibatis.db.sqlmap.cache.oscache.OSCacheController)**

OSCACHE Cache 实现是OSCache2.0 缓存引擎的一个Plugin。它具有高度的可配置性，分布式，高度的灵活性。

```
<cache-model name="product-cache" implementation  
="OSCACHE">
```

```
<flush-interval hours="24"/>
```

```
<flush-on-execute statement="insertProduct"/>
```

```
<flush-on-execute statement="updateProduct"/>
```

```
<flush-on-execute statement="deleteProduct"/>
```

```
</cache-model>
```

OSCACHE 实现不使用cache-property 元素。而是在类路径的根路径中使用标准的oscache.properties 文件进行配置。

在oscache.properties 文件中，您可以配置Cache 的算法（和上面讨论的算法很类似），Cache 的大小，持久化方法（内存，文件等）和集群方法。

要获得更详细的信息，请参考OSCache 文档。OSCache 及其文档可以从OpenSymphony网站上获

取：<http://www.opensymphony.com/oscache/>

本文引用通告地址：

<http://blog.csdn.net/sunsnow8/services/trackbacks/246582.aspx>

## 基于struts+spring+ibatis的轻量级J2EE开发

---

吴高峰 ( shuwgf@21cn.com )  
常德卷烟厂信息技术部  
2005年2月

大多数IT组织都必须解决三个主要问题：1. 帮助组织减少成本 2. 提高效率。完成这些问题一般都需要实现对多个业务系统的数据和业务逻辑系统集成工程，以便联结业务流程、实现数据的访问与共享。

JpetStore 4.0是ibatis的最新示例程序，基于Struts MVC框架（注：非持久化层。该示例程序设计优雅，层次清晰，可以学习以及作为一个高基础上，采用Spring对其中间层（业务层）进行改造。使开发量进一步好处...

### 1. 前言

JpetStore 4.0是ibatis的最新示例程序。ibatis是开源的持久层产品，包含ibatis 2.0 框架。JpetStore示例程序很好的展示了如何利用ibatis来开发petStore有如下特点：

- ibatis数据层
- POJO业务层
- POJO领域类
- Struts MVC
- JSP 表示层

以下是本文用到的关键技术介绍，本文假设您已经对Struts，SpringFramework中的参考资料。

- Struts 是目前Java Web MVC框架中不争的王者。经过长达五年的，且占有了MVC框架中最大的市场份额。但是Struts某些技术特性上，Struts 2 这些设计更精密，扩展性更强的框架，Struts受到了前所未有的选择。本文的原型例子JpetStore 4.0就是基于Struts开发的，在定义Action类，并且在form bean类的定义上也是开创性的，令人耳目一新。
- Spring Framework 实际上是Expert One-on-One J2EE Design and Spring Framework的功能非常多。包含AOP、ORM、DAO、Context

虑，JpetStore 4.0用的是更成熟的Struts和JSP；DAO由于目前Hibernate用的就是ibatis。因此最需要用的是AOP、ORM、Context。Context，非常强大。目前AOP应用最成熟的还是在事务管理上。

- ibatis 是一个功能强大实用的SQL Map工具，不同于其他ORM工具对于ORM工具，它的SQL语句是根据映射定义生成的。ibatis 以SQL提供了更大的自由空间。有ibatis代码生成的工具，可以根据DDL自

## 2. JpetStore简述

### 2.1. 背景

最初是Sun公司的J2EE petstore，其最主要目的是用于学习J2EE，但是后来E petstore来比较各应用服务器的性能。微软推出了基于.Net平台的 Petstore改良的基于struts的轻便框架J2EE web应用程序，相比来说，JpetStore实践和模式，避免了很多"反模式"，如使用存储过程，在java代码中嵌入SQL。JpetStore 4.0。

### 2.2. JpetStore开发运行环境的建立

#### 1、开发环境

- Java SDK 1.4.2
- Apache Tomcat 4.1.31
- Eclipse-SDK-3.0.1-win32
- HSQLDB 1.7.2

#### 2、Eclipse插件

- EMF SDK 2.0.1：Eclipse建模框架，lomboz插件需要，可以使用run
  - lomboz 3.0：J2EE插件，用来在Eclipse中开发J2EE应用程序
  - Spring IDE 1.0.3：Spring Bean配置管理插件
  - xmlbuddy\_2.0.10：编辑XML，用免费版功能即可
  - tomcatPluginV3：tomcat管理插件
  - Properties Editor：编辑java的属性文件,并可以预览以及自动存盘,省
- 麻烦。

#### 3、示例源程序

- ibatis示例程序JpetStore 4.0 <http://www.ibatis.com/jpetstore/jpetstore>
- 改造后的源程序（+spring）（源码链接）

## 2.3. 架构

图1 JpetStore架构图



图1 是JPetStore架构图，更详细的内容请参见JPetStore的白皮书。参见JPetStore 4.0的具体实现图（见图2），思路一下子就豁然开朗了。前言中提到了form bean类上。

struts Action类只有一个：BeanAction。没错，确实是一个！与传统的struts Action类不同，它其实是一个通用类，利用反射原理，根据URL来决定调用formbean的哪个方法。这降低了对struts的依赖（与struts以及WEB容器有关的几个类都放在commons-logging包中），我们会很容易的把它移植到新的框架如JSF，spring。

这样重心就转移到form bean上了，它已经不是普通意义上的form bean了，它有了自己的方法，而且已经具有了行为，从这个意义上来说，它更像一个BO(Business Object)。它利用反射原理，根据URL来决定调用form bean的哪个方法（行为）。form bean

```
public String myActionMethod() {
    //..work
    return "success";
}
```

方法的返回值直接就是字符串，对应的是forward的名称，而不再是Action

BeanAction类代劳了。

另外，程序还提供了ActionContext工具类，该工具类封装了request、request attributes和 application attributes中的数据存取操作，简单而线程安全解耦。

在这里需要特别指出的是，BeanAction类是对struts扩展的一个有益尝试，一直在发展中。

## 图2 JpetStore 4.0具体实现

图2 JpetStore 4.0具体实现



### 2.4. 代码剖析

下面就让我们开始进一步分析JpetStore4.0的源代码，为下面的改造铺路。

- BeanAction.java是唯一一个Struts action类，位于com.ibatis.struts射机制，把控制转移到form bean的某个方法来处理。详细处理过程
- Form bean类位于com.ibatis.jpstore.presentation包下，命名规则BaseBean类实际继承于ActionForm，因此，Form bean类就是Str框架自动填充。而实际上，JpetStore4.0扩展了struts中ActionForm为（方法）由BeanAction根据配置（struts-config.xml）的URL来让。

Struts-config.xml的配置里有3种映射方式，来告诉BeanAction把控以这个请求连接为例http://localhost/jpetstore4/shop/viewOrder.do

### 1. URL Pattern

```
<action path="/shop/viewOrder" type="com.ibatis.struts.B
name="orderBean" scope="session"
validate="false">
  <forward name="success" path="/order/ViewOrder.jsp"/>
</action>
```

此种方式表示，控制将被转发到"orderBean"这个form bean对象的以"/"分隔的最后一部分。

### 2. Method Parameter

```
<action path="/shop/viewOrder" type="com.ibatis.struts.B
name="orderBean" parameter="viewOrder" scope="session"
validate="false">
  <forward name="success" path="/order/ViewOrder.jsp"/>
</action>
```

此种方式表示，控制将被转发到"orderBean"这个form bean对象的参数表示form bean类上的方法。"parameter"参数优先于"path"参数

### 3. No Method call

```
<action path="/shop/viewOrder" type="com.ibatis.struts.B
name="orderBean" parameter="*" scope="session"
validate="false">
  <forward name="success" path="/order/ViewOrder.jsp"/>
</action>
```

此种方式表示，form bean上没有任何方法被调用。如果存在"name"属性，把控制转发到"success"。否则，如果name为空，则直接转发。

这就相当于struts内置的org.apache.struts.actions.ForwardAction的

```
<action path="/shop/viewOrder" type="org.apache.struts.acti
parameter="/order/ViewOrder.jsp " scope="session" valida
</action>
```

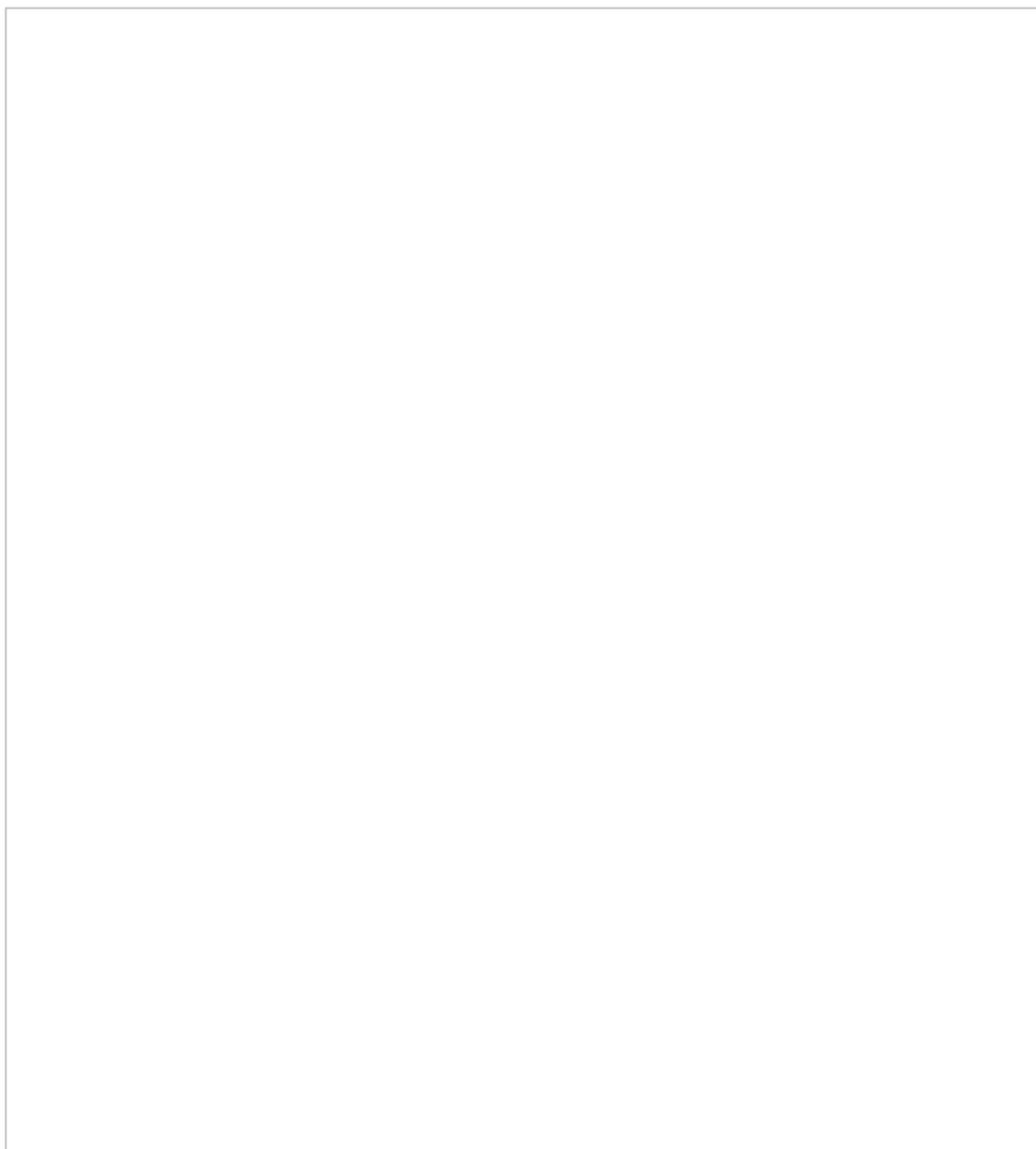
- Service类位于com.ibatis.jpjpetstore.service包下，属于业务层。这些Service类通过Form bean类来调用。
- com.ibatis.jpjpetstore.persistence iface包下的类是DAO接口，属于持久层。这些类通过Form bean类来调用。 DaoConfig类是工具类（DAO工厂类），Service类通过DaoConfig类来调用，实现了如图2中{耦合2}的解耦。
- com.ibatis.jpjpetstore.persistence.sqlmapdao包下的类是对应DAO接口的实现类。这些实现类继承BaseSqlMapDao类，而BaseSqlMapDao类是SqlMapDao接口。这些实现类的配置文件存放在com.ibatis.jpjpetstore.persistence.sqlmapdao.sc
- Domain类位于com.ibatis.jpjpetstore.domain包下，是普通的javabeans。这些类属于业务层和数据层，用于在不同层之间传输数据。

剩下的部分就比较简单了，请看具体的源代码，非常清晰。

#### 2.5. 需要改造的地方

JpetStore4.0的关键就在struts Action类和form bean类上，这也是其精髓。此次改造中我们要保留下来，即控制层一点不变，表现层获取相应业务数据。需要特别关注的改动是业务层和持久层，幸运的是JpetStore4.0设计非常清晰。

1. 业务层和数据层用Spring BeanFactory机制管理。
  2. 业务层的事务由spring 的aop通过声明来完成。
  3. 表现层 ( form bean ) 获取业务类的方法改由自定义工厂类来实现 ( /
- ### 3. JPetStore的改造
- #### 3.1. 改造后的架构



其中红色部分是要增加的部分，蓝色部分是要修改的部分。下面就让我

### 3.2. Spring Context的加载

为了在Struts中加载Spring Context，一般会在struts-config.xml的最后添

```
<plug-in className="org.springframework.web.struts.ContextLoader"
<set-property property="contextConfigLocation"
value="/WEB-INF/applicationContext.xml" />
</plug-in>
```

Spring在设计时就充分考虑到了与Struts的协同工作，通过内置的Struts  
这里我们一点也不改动JPetStore的控制层(这是JpetStore4.0的精华之  
xt。我们利用的是spring framework 的BeanFactory机制,采用自定义的  
以看出Spring有多灵活，它提供了各种不同的方式来使用其不同的部分

具体的来说，就是在com.ibatis.spring包下创建CustomBeanFactory类  
录下。以下就是该类的全部代码，很简单：

```
public final class CustomBeanFactory {
    static XmlBeanFactory factory = null;
    static {
        Resource is = new
InputStreamResource( CustomBeanFactory.class.getResourceAsStream
        factory = new XmlBeanFactory(is);
    }
    public static Object getBean(String beanName){
        return factory.getBean(beanName);
    }
}
```

实际上就是封装了Spring 的XMLBeanFactory而已，并且Spring的配置  
actory.getBean("someBean")来获得需要的对象了(例如someBean)，而  
1}的解耦。

CustomBeanFactory类在本文中只用于表现层的form bean对象获得spring的ApplicationContext.xml中。但是，为什么不把表现层的form bean类也配置在Spring中？问题的答案就在于form bean类是struts自动创建的：在一次请求中，struts判断，如果ActionForm类是struts自动创建的，那么struts就会把用户提交的表单数据保存到ActionForm对象中。因此formbean类的对象就可以，所以只有他们在spring中配置。

所以，很自然的，我们就创建了CustomBeanFactory类，在表现层来衔接struts和spring的解耦。

### 3.3. 表现层

通过分析到，struts和spring是在表现层衔接起来的，那么表现层就要做一些事情。以AccountBean类为例：

上

原来的源代码如下

```
private static final AccountService accountService = AccountService.  
private static final CatalogService catalogService = CatalogService.
```

改造后的源代码如下

```
private static final AccountService accountService = (AccountService) Spring.  
private static final CatalogService catalogService = (CatalogService) Spring.
```

其他的几个presentation类以同样方式改造。这样，表现层就完成了。如果没有看出什么特别之处的好处啊？你还是额外实现了一个工厂类。别着急发现：

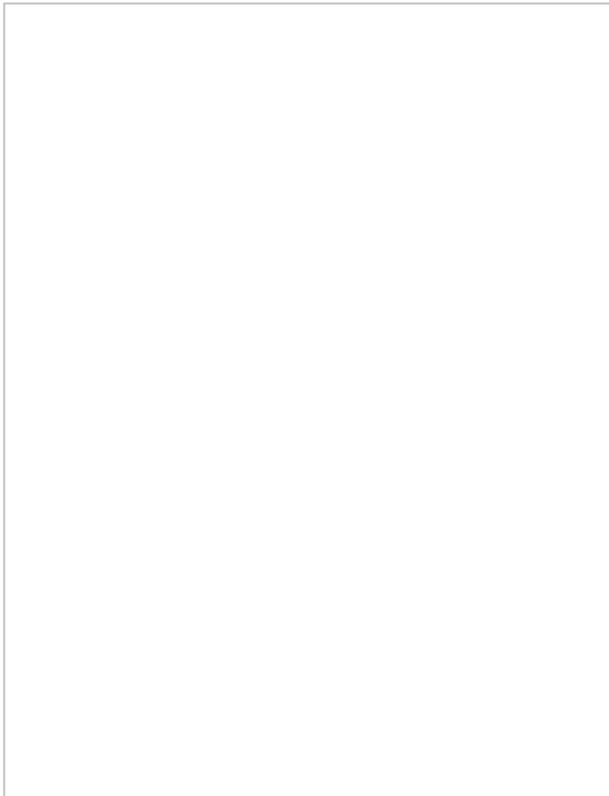
- presentation类仅仅面向service类的接口编程，具体AccountService

配置文件里配置。（本例中，为了最大限度的保持原来的代码不作因为如此的方便和自然，当然您也可以不这么做。

- CustomBeanFactory这个工厂类为什么会如此简单，因为其直接使个DI容器，其设计哲学是提供一种无侵入式的高扩展性的框架。为机制，通过动态调用的方式避免硬编码方式的约束，并在此基础上制的实现基础。org.springframework.beans包中包括了这些核心jry类。

### 3.4. 持久层

在讨论业务层之前，我们先看一下持久层，如下图所示：



在上文中，我们把iface包下的DAO接口归为业务层，在这里不需要做修现类，并在spring的配置文件中配置起来。

#### 1、修改基类

所有的DAO实现类都继承于BaseSqlMapDao类。修改BaseSqlMapDao

---

```

public class BaseSqlMapDao extends SqlMapClientDaoSupport {
    protected static final int PAGE_SIZE = 4;
    protected SqlMapClientTemplate smcTemplate = this.getSqlMapCli
    public BaseSqlMapDao() {
        }
    }
}

```

使BaseSqlMapDao类改为继承于Spring提供的SqlMapClientDaoSupport类，并实现SqlMapClientTemplate。关于SqlMapClientTemplate类的详细说明请参照

## 2、修改DAO实现类

所有的DAO实现类还是继承于BaseSqlMapDao类，实现相应的DAO接口，以AccountSqlMapDao类为例，部分代码如下：

```

    public List getUsernameList() {
        return smcTemplate.queryForList("getUsernameList", null);
    }
    public Account getAccount(String username, String password) {
        Account account = new Account();
        account.setUsername(username);
        account.setPassword(password);
        return (Account) smcTemplate.queryForObject("getAccountByUse
    }
    public void insertAccount(Account account) {
        smcTemplate.update("insertAccount", account);
        smcTemplate.update("insertProfile", account);
        smcTemplate.update("insertSignon", account);
    }
}

```

就这么简单，所有函数的签名都是一样的，只需要查找替换就可以了！

## 3、除去工厂类以及相应的配置文件

除去DaoConfig.java这个DAO工厂类和相应的配置文件dao.xml，因为L

#### 4、DAO在Spring中的配置 ( applicationContext.xml )

```
<bean id="dataSource"
    class="org.springframework.jdbc.datasource.DriverManager
    <property name="driverClassName">
        <value>org.hsqldb.jdbcDriver</value>
    </property>
    <property name="url">
        <value>jdbc:hsqldb:hsqldb://localhost/xdm</value>
    </property>
    <property name="username">
        <value>sa</value>
    </property>
    <property name="password">
        <value></value>
    </property>
</bean>
<!-- ibatis sqlMapClient config -->
<bean id="sqlMapClient"
    class="org.springframework.orm.ibatis.SqlMapClientFactor
    <property name="configLocation">
        <value>
            classpath:com\ibatis\jpetstore\persistence\sqlma
        </value>
    </property>
    <property name="dataSource">
        <ref bean="dataSource"/>
    </property>
</bean>
<!-- Transactions -->
<bean id="TransactionManager"
    class="org.springframework.jdbc.datasource.DataSourceTra
    <property name="dataSource">
        <ref bean="dataSource"/>
    </property>
</bean>
<!-- persistence layer -->
<bean id="AccountDao"
    class="com.ibatis.jpetstore.persistence.sqlmapdao.Accoun
    <property name="sqlMapClient">
        <ref local="sqlMapClient"/>
    </property>
</bean>
```

具体的语法请参照附录中的"Spring中文参考手册"。在这里只简单解释-

1. 我们首先创建一个数据源dataSource，在这里配置的是hsqldb数据库  
acle.jdbc.driver.OracleDriver"，URL的值类似于"jdbc:oracle:thin:@wuc  
现在我们就可以去掉properties目录下database.properties这个配置文件  
erties resource="properties/database.properties"/>对它的引用。

2. sqlMapClient节点。这个是针对ibatis SqlMap的SqlMapClientFactory  
类。configLocation属性配置了ibatis映射文件的名称。dataSource属性  
O都默认使用了该数据源，除非在DAO的配置中另外显式指定。

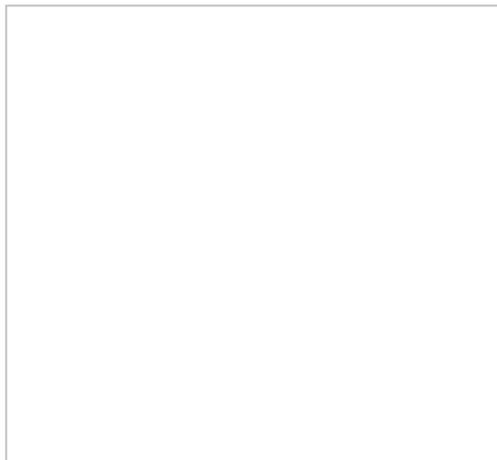
3. TransactionManager节点。定义了事务，使用的是DataSourceTrans

4. 下面就可以定义DAO节点了，如AccountDao，它的实现类是com.iba  
o，使用的SQL配置从sqlMapClient中读取，数据库连接没有特别列出，  
e。

这样，我们就把持久层改造完了，其他的DAO配置类似于AccountDao。  
->AccountSqlMapDao实现。

### 3.5. 业务层

业务层的位置以及相关类，如下图所示：



在这个例子中只有3个业务类，我们以OrderService类为例来改造，这个

## 1、在ApplicationContext配置文件中增加bean的配置：

```
<bean id="OrderService"
      class="org.springframework.transaction.interceptor.TransactionInterceptor"
      <property name="transactionManager">
        <ref local="TransactionManager"></ref>
      </property>
      <property name="target">
        <bean class="com.ibatis.jpetstore.service.OrderService"
              <property name="itemDao">
                <ref bean="ItemDao"/>
              </property>
              <property name="orderDao">
                <ref bean="OrderDao"/>
              </property>
              <property name="sequenceDao">
                <ref bean="SequenceDao"/>
              </property>
            </bean>
          </property>
      <property name="transactionAttributes">
        <props>
          <prop key="insert*">PROPAGATION_REQUIRED</prop>
        </props>
      </property>
    </bean>
```

定义了一个OrderService，还是很容易懂的。为了简单起见，使用了嵌套的bean，分别引用了ItemDao，OrderDao，SequenceDao。该bean的interceptor属性会自动创建一个事务advisor，该advisor包括一个基于事务属性的

## 2、业务类的修改

以OrderService为例：

```
public class OrderService {
```

```

    /* Private Fields */
    private ItemDao itemDao;
    private OrderDao orderDao;
    private SequenceDao sequenceDao;

    /* Constructors */

    public OrderService() {
    }

    /**
     * @param itemDao 要设置的 itemDao。
     */
    public final void setItemDao(ItemDao itemDao) {
        this.itemDao = itemDao;
    }

    /**
     * @param orderDao 要设置的 orderDao。
     */
    public final void setOrderDao(OrderDao orderDao) {
        this.orderDao = orderDao;
    }

    /**
     * @param sequenceDao 要设置的 sequenceDao。
     */
    public final void setSequenceDao(SequenceDao sequenceDao) {
        this.sequenceDao = sequenceDao;
    }
    //剩下的部分
    .....
}

```

红色部分为修改部分。Spring采用的是Type2的设置依赖注入，所以我不需要再在构造函数中设置itemDao, OrderDao, SequenceDao的值由spring在运行期间注入。构造函数就负责事务在配置中声明), daoManager.startTransaction();等与事务相关的逻辑精简了很多！可以更关注业务的实现。

#### 4. 结束语

ibatis是一个功能强大实用的SQL Map工具，可以直接控制SQL,为系统提供事务支持。JpetStore 4.0,设计优雅，应用了迄今为止很多最佳实践和设计模式，非常有序。JpetStore 4.0是基于struts的，本文在此基础上，最大程度保持了原代码的兼容性，引入了Spring。在您阅读了本文以及改造后的源代码后，会深切的

，业务对象的依赖注入，一致的数据存取框架和声明式的事务处理，统一化的，Spring有分层的体系结构，这意味着您能选择仅仅使用它任何一

## 参考资料

- **jpetstore相关各种资料和源程序** <http://www.ibatis.com/jpetstore/jpetstore.html>
- **Spring中文参考手册**<http://www.jactiongroup.net/reference/html/index.html>
- **Spring 开发指南** 夏昕
- **Struts** <http://struts.apache.org/>

## ibatis中执行pl/sql语句块的测试

配置文件：

```
<?xml version="1.0" encoding="GBK"?>
<!DOCTYPE sqlMap PUBLIC "-//iBATIS.com//DTD SQL Map
2.0//EN"
"http://www.ibatis.com/dtd/sql-map-2.dtd">
<sqlMap namespace="Test">
<update id="update"><![CDATA[
declare
n_count number;
begin
select count(*) into n_count from t_account;
update t_auth set s_authdesc='记录数：'||n_count;
end;
]]> </update>
</sqlMap>
```

测试代码：

```
public class Test
{
    public static void main(String[] args)
    {
        SqlMapClient sqlMap = SqlMapConfig.getSqlMap();
        try
        {
            sqlMap.startTransaction();
            sqlMap.update("Test.update", null);
            sqlMap.commitTransaction();
        }
        catch (SQLException e)
        {
            e.printStackTrace();
        }
        finally
        {
            try
            {
                sqlMap.endTransaction();
            }
            catch (SQLException e)
            {
                e.printStackTrace();
            }
        }
    }
}
```

}

## [pojo+xDoclet生成ibatis映射文件](#)

XDoclet实现基本原理是，通过在Java代码加入特定的JavaDoc tag，从而为其添加特定的附加语义，之后通过XDoclet工具对代码中JavaDoc Tag进行分析，自动生成与代码对应的配置文件，在Ejb、hibernate、struts中使用得比较广泛了，ibatis比较草根，没有这个棉袄穿，怎么办呢，大过年的，为了让ibatis也有新衣服穿，我只好弄一个了。如果你不了解xDoclet，推荐阅读一下此文：

<http://www-900.ibm.com/developerWorks/cn/java/j-xdoclet/index.shtml?ca=dwcn-newsletter-java>

我的思路是力求简单，原先想写一个xdt模板文件，订制一些标签在POJO中使用，我偷个懒，看见hibernate的衣服出自名设计师，很是羡慕，我来个仿版（所谓的A货），借用它的标签，另一个重要原因是，很多IDE已经支持hibernate标签的编辑工作了。

我只借用4个标签，要作少量扩展（后面实例说明）：

- 1 @hibernate.class
- 2 @hibernate.discriminator
- 3 @hibernate.property - 简单属性，对应数据表字段
- 4 @hibernate.component - 复杂属性,ibatis中对应另一个select子句

现在请出例子猫豆MM，User.java:

```
package org.chage.pojo;
import java.util.List;
/**
 * @hibernate.class table="D_USER"
 * @hibernate.discriminator column="USERID" property="id" type="long"
 */
public class User {
    private Long id;
    private String username;
    private List roles;

    /**
     * @hibernate.property column="USER_ID"
     */
}
```

```

public Long getId() { return id; }
/**
 * @hibernate.property column="USER_NAME" update="true"
 */
public Long getUsername() { return username; }
/**
 * @hibernate.component column="USER_ID" select="selectRoles"
 */
public List getRoles(){ return roles; }
//以下省略了setter方法
.....
}

```

蓝色标记的为新增标签，从字面意思大家就可以理解了吧，第一处是为了指明主键字段对应的属性，(上帝保佑，你采取唯一主键最好业务无关，这不奢侈吧) update="true"是为了标明update时需要参与更新的字段,@hibernate.component后面的是为了实现一对一或一对多的映射。

下面是build.xml中相关部分：

```

<target name="init">
  <path id="xdoclet.path">
    <fileset dir="${xdocletlib}">
      <include name="*.jar" />
    </fileset>
  </path>
  <taskdef classname="xdoclet.DocletTask" classpathref="xdoclet.path"
name="doclet" />
</target>
<target name="sqlmap" depends="init">
  <doclet destdir="${sqlmap.dir}" excludedtags="@version,@author,@todo"
force="true" verbose="true">
    <fileset dir="${src.dir}">
      <include name="**/${pojo.java}.java" />
    </fileset>
    <template destinationFile="{0}.xml"
templateFile="${template.dir}/gensqlmap.xdt" subTaskName="Generate
SqlMap xml...">

```

```
</template>
</doclet>
</target>
```

最后，给出这个xdt模板：

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE sqlMap PUBLIC "-//iBATIS.com//DTD SQL Map 2.0//EN"
"http://www.ibatis.com/dtd/sql-map-2.dtd">

<sqlMap namespace="<XDtClass:className/>">

  <typeAlias alias="<XDtClass:className/>" type="
<XDtClass:fullClassName/>" />

  <resultMap id="<XDtClass:className/>Result" class="
<XDtClass:className/>">
    <XDtMethod:forAllClassMethods>
      <XDtMethod:ifIsGetter>
        <XDtMethod:ifHasMethodTag tagName="hibernate.property" >
          <result property="<XDtMethod:propertyName/>" column="
<XDtMethod:methodName/>" tagName="hibernate.property"
paramName="column"/>" />
        </XDtMethod:ifHasMethodTag>
        <XDtMethod:ifHasMethodTag tagName="hibernate.component">
          <result property="<XDtMethod:propertyName/>" column="
<XDtMethod:methodName/>" tagName="hibernate.component"
paramName="column"/>" select="<XDtMethod:methodName/>"
tagName="hibernate.component" paramName="select"/>" />
        </XDtMethod:ifHasMethodTag>
      </XDtMethod:ifIsGetter>
    </XDtMethod:forAllClassMethods>
  </resultMap>

  <select id="get<XDtClass:className/>" resultMap="
<XDtClass:className/>Result" parameterClass="<XDtClass:classTagValue
tagName='hibernate.discriminator' paramName='type'/>">
    select
```

```

    <XDtMethod:forAllClassMethods>
      <XDtMethod:ifIsGetter>
        <XDtMethod:ifHasMethodTag tagName="hibernate.property" >
          <XDtMethod:methodTagValue tagName="hibernate.property"
paramName="column"/>,
        </XDtMethod:ifHasMethodTag>
      </XDtMethod:ifIsGetter>
    </XDtMethod:forAllClassMethods>
    from <XDtClass:classTagValue tagName='hibernate.class'
paramName='table'/>
    where <XDtClass:classTagValue tagName='hibernate.discriminator'
paramName='column'/> = #value#
  </select>

  <update id="insert<XDtClass:className/>" parameterClass="
<XDtClass:className/>">
    insert into <XDtClass:classTagValue tagName='hibernate.class'
paramName='table'/>
    (
      <XDtMethod:forAllClassMethods>
        <XDtMethod:ifIsGetter>
          <XDtMethod:ifHasMethodTag tagName="hibernate.property" >
            <XDtMethod:methodTagValue tagName="hibernate.property"
paramName="column"/>,
          </XDtMethod:ifHasMethodTag>
        </XDtMethod:ifIsGetter>
      </XDtMethod:forAllClassMethods>
    )values(
      <XDtMethod:forAllClassMethods>
        <XDtMethod:ifIsGetter>
          <XDtMethod:ifHasMethodTag tagName="hibernate.property" >
            #<XDtMethod:propertyName/>#,
          </XDtMethod:ifHasMethodTag>
        </XDtMethod:ifIsGetter>
      </XDtMethod:forAllClassMethods>
    )
  </update>

```

```

    <update id="update<XDtClass:className/>" parameterClass="
<XDtClass:className/>">
      update <XDtClass:classTagValue tagName='hibernate.class'
paramName='table'/> set
      <XDtMethod:forAllClassMethods>
        <XDtMethod:ifIsGetter>
          <XDtMethod:ifHasMethodTag tagName="hibernate.property" >
            <XDtMethod:ifMethodTagValueEquals tagName="hibernate.property"
paramName="update" value="true">
              <XDtMethod:methodTagValue tagName="hibernate.property"
paramName="column"/> = #<XDtMethod:propertyName/>#,
            </XDtMethod:ifMethodTagValueEquals>
          </XDtMethod:ifHasMethodTag>
        </XDtMethod:ifIsGetter>
      </XDtMethod:forAllClassMethods>
      where <XDtClass:classTagValue tagName='hibernate.discriminator'
paramName='column'/> = #<XDtClass:classTagValue
tagName='hibernate.discriminator' paramName='property'/>#
    </update>

```

```

    <update id="del<XDtClass:className/>" parameterClass="
<XDtClass:classTagValue tagName='hibernate.discriminator'
paramName='type'/>">
      delete from <XDtClass:classTagValue tagName='hibernate.class'
paramName='table'/>
      where <XDtClass:classTagValue tagName='hibernate.discriminator'
paramName='column'/> = #value#
    </update>
</sqlMap>

```

有一个郁闷之处是，在insert和update子句中，字段列表最后面多了一个逗号，有空了再说吧，快过年了，人心散了，队伍.....

本文引用通告地址：

[http://blog.csdn.net/\\_chage/services/trackbacks/275265.aspx](http://blog.csdn.net/_chage/services/trackbacks/275265.aspx)

## [自己搞的在oralce下动态生成IBATIS的sqlMap文件小工具](#)

如果需要完整的代码或建议可以发到zhoubikui@eyou.com。

metadata类，可以在oralce下的取得相应的数据表信息,并生成文件sqlMap文件

\* 目前可以生成IBATIS从数据库表到IBATIS sqlMAP类型主要有SELECT，INSERT，UPDATE

\* 因项目太急，只是临时自己使用，没有开发RESULTMAP和PARAMETERMAP生成

\* 下步计划开发这个工具，并把它图形化，先听一下各位的意见，希望与各位交流。

\* 调用该程序，只需调用getMetaData（）方法即可

\* 数据库配置见cn/com/mofit/demo/system/dao/maps/sqlMap-config.xml，

\* 修改当前目录下的jdbc.properties数据库配置就可使用

其主类实现方法是：

```
package cn.com.mofit.util.jdbc;
```

```
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.io.Reader;
import java.sql.Connection;
import java.sql.DatabaseMetaData;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.SQLException;
import java.sql.Statement;
import java.sql.Types;
import java.util.HashMap;
import java.util.Iterator;
import java.util.List;
```

```

import java.util.Vector;

import org.springframework.dao.DataAccessException;
import org.springframework.orm.ibatis.SqlMapClientTemplate;

import cn.com.mofit.util.spring.orm.ibatis.SqlMapDaoSupportPlus;

import com.ibatis.common.resources.Resources;
import com.ibatis.sqlmap.client.SqlMapClient;
import
com.ibatis.sqlmap.engine.builder.xml.XmlSqlMapClientBuilder;

/**
 *
 * @author 周必奎
 * 2004-10-15
 * @email:zhoubikui@eyou.com
 * @deprecated metadata类，可以在oralce下的取得相应的数据表
信息,并生成文件sqlMap文件
 * 目前可以生成IBATIS从数据库表到IBATIS sqlMAP类型主要有
SELECT，INSERT，UPDATE
 * 因项目太急，只是临时自己使用，没有开发RESULTMAP和
PARAMETERMAP生成
 * 下步计划开发这个工具，并把它图形化，先听一下各位的意
见，希望与各位交流。
 * 调用该程序，只需调用getMetaData（）方法即可
 * 数据库配置见cn/com/mofit/demo/system/dao/maps/sqlMap-
config.xml，
 * 修改当前目录下的jdbc.properties数据库配置就可使用
 */
public class RsMetaDataOracle {
/**
 * filePath SQLMAP文件生成的路径名，是绝对路径

```

```
*/
private String filePath = "c:/";

/*
 * mapTablename 要映射的数据库的表名
 */
private String mapTablename = "BK_BILL";

//System.getProperty("user.dir") + "/config/sqlmap/";
private static SqlMapClientTemplate sqlTemp;
static {
    try {
        SqlMapDaoSupportPlus sqlsu = new SqlMapDaoSupportPlus();

        String resource = "cn/com/mofit/demo/system/dao/maps/sqlMap-
        config.xml";
        Reader read;
        read = Resources.getResourceAsReader(resource);

        XmlSqlMapClientBuilder xmlBuilder = new
        XmlSqlMapClientBuilder();
        SqlMapClient sqlMap = xmlBuilder.buildSqlMap(read);

        sqlsu.setSqlMapClient(sqlMap);
        sqlsu.afterPropertiesSet();
        sqlTemp = sqlsu.getSqlMapClientTemplate();

    } catch (IOException e1) {
        e1.printStackTrace();
    } catch (Exception e) {

        e.printStackTrace();
    }
}
```

```
}
```

```
private static SqlMapClientTemplate getSqlMapTempInstance() {  
return sqlTemp;  
}
```

```
public void getMetaData() throws DataAccessException {  
try {  
//DaoCommon.startTransaction();  
SqlMapClientTemplate sqlTemp = RsMetaDataOracle  
.getSqlMapTempInstance();  
  
//SqlMap sqlMap = DaoCommon.getSqlMap(this);  
Connection conn = sqlTemp.getDataSource().getConnection();  
Statement stmt = conn.createStatement();  
List list = getTableNames();  
  
for (Iterator iter = list.iterator(); iter.hasNext()🤔) {  
String element = (String) iter.next();  
  
ResultSet rs = stmt.executeQuery("select * from " + element);  
ResultSetMetaData rsmd = rs.getMetaData();  
int numberOfColumns = rsmd.getColumnCount();  
if (element.startsWith(mapTablename.toUpperCase())) {  
  
File file = new File(filePath);  
  
if (!file.exists()) {  
file.mkdir();  
}  
  
file = new File(filePath + element.toLowerCase() + ".xml");
```

```

String xml = "<?xml version=\"1.0\" encoding=\"GBK\" ?>\n";
xml += "<!DOCTYPE sql-map\n";
xml += "PUBLIC \"-//iBATIS.com//DTD SQL Map Config 2.0//EN\"
\n";
xml += "\"http://www.ibatis.com/dtd/sql-map-2.dtd\">\n";
xml += ("<sql-map namespace=\"\" + element.toLowerCase() +
\">\n");
xml += getXml(rsmd, numberOfColumns, element);
xml += "\n</sql-map>";

```

```

FileWriter writer = new FileWriter(file);
writer.write(xml);
writer.flush();
writer.close();
}
}
} catch (DataAccessException e) {
e.printStackTrace();
} catch (SQLException e) {

```

```

e.printStackTrace();
} catch (IOException e) {

```

```

e.printStackTrace();
}
}

```

```

private String getXml(ResultSetMetaData rsmd, int
numberOfColumns,
String tableName) throws SQLException {
String result = "";
result += (createfindSql(rsmd, numberOfColumns, tableName));

```

```

result += (createInsertSql(rsmd, numberOfColumns, tableName));
result += (createUpdateSql(rsmd, numberOfColumns, tableName));
return result;
}

```

```

private String createfindSql(ResultSetMetaData rsmd, int
numberOfColumns,
String tableName) throws SQLException {
String result;
result = "<!--
=====\\n
mapped-statement find
\\n===== --
>";

```

```

result += ("\\n<select id=\\\"find\" + tableName.toLowerCase() + "Dao\\\"
resultClass=\\\"java.util.HashMap\\\">");

```

```

result += ("\\n select $listfield$ from " + tableName + "\\n <dynamic
prepend=\\\"where\\\">");

```

```

result += createWhereSql(rsmd, numberOfColumns, "and", 1);
result += "\\n </dynamic>";
result += "\\n</select>\\n\\n\\n";
return result;
}

```

```

private String createColumnsString(ResultSetMetaData rsmd)
throws SQLException {
String result = "";
int numberOfColumns = rsmd.getColumnCount();
for (int i = 1; i <= numberOfColumns; i++) {
String colName = rsmd.getColumnName(i);

```

```
String name = rsmd.getColumnTypeName□;
result += (((i == 1) ? "\n " : "\n ") + colName + ",");
}
```

```
return result.substring(1, result.length() - 1);
}
```

```
private String createWheremapSql(ResultSetMetaData rsmd, int
numberOfColumns)
throws SQLException {
String result = "";
```

```
for (int i = 1; i <= numberOfColumns; i++) {
String colName = rsmd.getColumnNames□;
String name = rsmd.getColumnTypeName□;
result += (((i == 1) ? "\n " : "\n and ") + colName
+ "=#" + colName + "#");
}
```

```
return result;
}
```

```
private String createInsertSql(ResultSetMetaData rsmd, int
numberOfColumns,
String tableName) throws SQLException {
String result = "<!--
===== \n
mapped-statement insert
\n===== --
>";
result += ("\n<insert id=\"insert\" + tableName.toLowerCase() +
"Dao\" parameterClass=\"java.util.HashMap\">");
result += ("\n insert into " + tableName + "( \n"
```

```

+ createColumnsString(rsmd) + ") "

+ "\n <dynamic prepend=\"values(\">");

result += createWhereSql(rsmd, numberOfColumns, ",", 3) + "));
result += "\n </dynamic>";
result += "\n</insert>\n\n\n";
return result;

}

private String createUpdateSql(ResultSetMetaData rsmd, int
numberOfColumns,
String tableName) throws SQLException {
String result = "<!--
=====
mapped-statement update
\n===== --
>";
result += ("\n<update id=\"update\" + tableName.toLowerCase() +
\"Dao\" parameterClass=\"java.util.HashMap\">");
result += ("\n update " + tableName
+ "\n <dynamic prepend=\"set\"> "
+ createWhereSql(rsmd, numberOfColumns, ",", 4)

+ "\n </dynamic> \n <dynamic prepend=\"where\">");

result += createWhereSql(rsmd, numberOfColumns, "and", 1);
result += "\n </dynamic>";
result += "\n</update>\n\n\n";
return result;
}

```

```

private String createWhereSql(ResultSetMetaData rsmd, int
numberOfColumns,
String prepend, int detail) throws SQLException {
String result = "";

for (int i = 1; i <= numberOfColumns; i++) {
String colName = rsmd.getColumnName(i);

result += ("\n <isPropertyAvailable prepend=\"\" property=\""
+ colName.toLowerCase() + "\" >");

result += ("\n <isNotNull prepend=\"" + prepend
+ "\" property=\"" + colName.toLowerCase() + "\" >");

switch (detail) {
case 1: //where 语句
result += ("\n " + colName + "=#"
+ colName.toLowerCase() + "#");

break;

case 2: //insert的语句
result += ("\n " + colName.toLowerCase());

break;

case 3: //insert 准备的
result += ("\n #" + colName.toLowerCase() + "#");

break;

case 4: //修改的set语句

```

```
result += ("\n "  
+ colName  
+ "#"  
+ colName.toLowerCase()  
+ (Types.VARCHAR == rsmd.getColumnType□ ? ":VARCHAR"  
: "")) + "#");
```

```
break;
```

```
default:
```

```
break;
```

```
}
```

```
result += ("\n </isNotNull>")  
+ "\n </isPropertyAvailable>";  
}
```

```
return result;
```

```
}
```

```
private void getType(ResultSetMetaData rsmd, int i, HashMap  
colMap)  
throws SQLException {  
switch (rsmd.getColumnType□) {  
case Types.VARCHAR:  
colMap.put("COLUMNTYPE", rsmd.getColumnTypeName□ + "("  
+ rsmd.getPrecision□ + ")");  
break;  
case 2:  
colMap.put("COLUMNTYPE", rsmd.getColumnTypeName□ + "("  
+ rsmd.getPrecision□ + "," + rsmd.getScale□ + ")");  
break;  
default:
```

```
colMap.put("COLUMNNTYPE", rsmd.getColumnTypeName());
break;
}
}
```

```
private List getTableNames() throws DataAccessException {
List result = new Vector();
```

```
//SqlMap sqlMap = DaoCommon.getSqlMap(this);
Connection conn;
try {
conn = getSqlMapTempInstance().getDataSource().getConnection();
DatabaseMetaData dbmd = conn.getMetaData();
ResultSet rs = dbmd.getTableTypes();
String[] types = { "TABLE" };
rs = dbmd.getTables(null, dbmd.getUserName(), "%", types);
```

```
while (rs.next()) {
result.add(rs.getString("TABLE_NAME"));
}
```

```
rs.close();
```

```
} catch (SQLException e) {
```

```
e.printStackTrace();
}
```

```
return result;
}
```

```
public String getFilePath() {
return filePath;
```

```

}

public void setFilePath(String filePath) {
this.filePath = filePath;
}

public String getMapTablename() {
return mapTablename;
}

public void setMapTablename(String mapTablename) {
this.mapTablename = mapTablename;
}
}

```

SQL-CONFIG文件配置是：

```

<?xml version="1.0" encoding="GB2312" ?>
<!DOCTYPE sqlMapConfig
PUBLIC "-//iBATIS.com//DTD SQL Map Config 2.0//EN"
"http://www.ibatis.com/dtd/sql-map-config-2.dtd">

<sqlMapConfig>
  <properties
resource="cn/com/mofit/demo/system/dao/maps/jdbc.properties"/>
  <!-- debug环境下，将其设为false. 正式运行时应设为true,启用缓存 -->
  <settings
    cacheModelsEnabled="false"
  />
  <transactionManager type="JDBC">
    <dataSource type="SIMPLE">
      <property name="JDBC.Driver" value="{jdbc.driverClassName}"/>
      <property name="JDBC.ConnectionURL" value="{jdbc.url}"/>

```

```
<property name="JDBC.Username" value="{jdbc.username}"/>
<property name="JDBC.Password" value="{jdbc.password}"/>
<property name="Pool.MaximumActiveConnections"
value="10"/>
<property name="Pool.MaximumIdleConnections" value="5"/>
<property name="Pool.MaximumCheckoutTime"
value="120000"/>
<property name="Pool.TimeToWait" value="500"/>
<property name="Pool.PingQuery" value="select 1 from
ACCOUNT"/>
<property name="Pool.PingEnabled" value="false"/>
<property name="Pool.PingConnectionsOlderThan"
value="1"/>
<property name="Pool.PingConnectionsNotUsedFor"
value="1"/>
</dataSource>
</transactionManager>
<!-- 非常简洁，将用到的sqlMap文件列到这儿就行了 -->
<sqlMap resource="cn/com/mofit/demo/system/dao/maps/User.xml"
/>
<sqlMap resource="cn/com/mofit/demo/bank/dao/maps/Bank.xml" />
</sqlMapConfig>
```

jdbc.properties文件配置：

jdbc.driverClassName=oracle.jdbc.driver.OracleDriver

jdbc.url=jdbc:oracle:thin:IP:1521:SID

jdbc.username=

jdbc.password=

jdbc.maxActive=3

jdbc.maxIdle=1

jdbc.maxWait=5000

本文引用通告地址：

<http://blog.csdn.net/ZHBK/services/trackbacks/137867.aspx>

# Apache xml-rpc入门

作者：yi5 转贴自：yi5\_yuyu  
文章录入：yuyu3007

由于最近做的一个项目需要，使用了apache xml-rpc，顺便整理一下使用

xml-rpc是一套允许运行在不同操作系统、不同环境的程序实现基于inter的规范和一系列的实现。这种远程过程调用使用http作为传输协议，xml作的编码格式。xml-rpc的定义尽可能的保持了简单，但同时能够传送、处理的数据结构。

关于xml-rpc更详细的信息，请参阅<http://www.xmlrpc.com>。

## 1，客户程序

Apache xml-rpc提供两种客户类：

org.apache.xmlrpc.XmlRpcClient：使用java.net.URLConnection

org.apache.xmlrpc.XmlRpcClientLite：自身提供轻量级的http client

如果您需要完全的http支持（例如：代理，重定向等等），你应该使用XmlRpcClient。反之，如果您不需要完全的http支持并且更注重性能，那么你应该仔细选择客户类。在某些平台上，可能XmlRpcClient更快，但是在某些平台上XmlRpcClientLite更快。

这两个客户类提供相同的接口。

在客户端使用apache xml-rpc是非常简单的，只需要完成下面的简单

```
// 建立xml-rpc客户
XmlRpcClient client = new XmlRpcClient("http://" + server + ":" + port);

// 设置调用参数
Vector params = new Vector();
params.addElement(name);

// 调用并取得结果
String result = (String) client.execute("hello.sayHello", params);
```

如果您需要进行异步调用，并使用executeAsync()方法。

## 2，登记Handler Object

org.apache.xmlrpc.XmlRpcServer和org.apache.xmlrpc.WebServer都提供方法registerHandlerObject：

```

addHandler (String name, Object handler);
removeHandler (String name);

```

3, 在servlet环境中使用xml-rpc  
典型的代码如下所示：

```

XmlRpcServer xmlrpc = new XmlRpcServer ();
xmlrpc.addHandler ("examples", new ExampleHandler ());
...
byte[] result = xmlrpc.execute (request.getInputStream ());
response.setContentType ("text/xml");
response.setContentLength (result.length());
OutputStream out = response.getOutputStream();
out.write (result);
out.flush ();

```

请注意：execute方法不会返回任何异常，因为所有错误都被编码成xml

4, 使用内建的http server  
代码如下：

```

XmlRpc.setDriver("org.apache.xerces.parsers.SAXParser");

//start the server
System.out.println("Starting XML-RPC Server.....");

WebServer server = new WebServer(8585);
//register our handler class

server.addHandler("hello", new HelloHandler());

server.start();

```

5, Apache xml-rpc支持的类型  
这些类型适用于xml-rpc的参数和返回类型，同时，如果参数或者返回类型的  
话，也适用于集合元素。

XML-RPC data type	Data Types generated by the Parser	Types expected by the Invoker as parameters of RPC handlers
<i4> or <int>	java.lang.Integer	int
<boolean>	java.lang.Boolean	boolean

<string>	java.lang.String	java.lang.String
<double>	java.lang.Double	double
<dateTime.iso8601>	java.util.Date	java.util.Date
<struct>	java.util.Hashtable	java.util.Hashtable
<array>	java.util.Vector	java.util.Vector
<base64>	byte[ ]	byte[ ]

6 , 使用内建http server的简单例子  
a , 建立handler object

```

        /*
        * 创建日期 2004-5-12
        *
        * 更改所生成文件模板为
        * 窗口 > 首选项 > Java > 代码生成 > 代码和注释
        */
        package helloxmlrpc;

        import java.util.Vector;

        /**
        * @author fyun
        *
        * 更改所生成类型注释的模板为
        * 窗口 > 首选项 > Java > 代码生成 > 代码和注释
        */
        public class HelloHandler {
        public String sayHello(String name) {
            return "Hello " + name;
        }
        }

```

b , 登记并启动server

```

        /*
        * 创建日期 2004-5-12
        *

```

\* 更改所生成文件模板为  
\* 窗口 > 首选项 > Java > 代码生成 > 代码和注释

```
*/
```

```
package helloxmlrpc;
```

```
/**
```

```
* @author fyun
```

```
*
```

\* 更改所生成类型注释的模板为

\* 窗口 > 首选项 > Java > 代码生成 > 代码和注释

```
*/
```

```
import org.apache.xmlrpc.*;
```

```
public class HelloServer {
```

```
public static void initServer() {
```

```
try {
```

```
XmlRpc.setDriver("org.apache.xerces.parsers.SAXParser")
```

```
//start the server
```

```
System.out.println("Starting XML-RPC Server.....");
```

```
WebServer server = new WebServer(8585);
```

```
//register our handler class
```

```
server.addHandler("hello", new HelloHandler());
```

```
server.start();
```

```
System.out.println("Now accepting requests.....");
```

```
} catch (ClassNotFoundException e) {
```

```
System.out.println("Could not locate SAX Driver");
```

```
}
```

```
}
```

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

```
initServer();
```

```
}
```

```

    }

    c , 客户程序
    /*
    * 创建日期 2004-5-12
    *
    * 更改所生成文件模板为
    * 窗口 > 首选项 > Java > 代码生成 > 代码和注释
    */
    package helloxmlrpc;

    /**
    * @author fyun
    *
    * 更改所生成类型注释的模板为
    * 窗口 > 首选项 > Java > 代码生成 > 代码和注释
    */
    import java.io.IOException;
    import org.apache.xmlrpc.XmlRpc;
    import org.apache.xmlrpc.XmlRpcClient;
    import java.net.MalformedURLException;
    import org.apache.xmlrpc.XmlRpcException;

    public class HelloClient {
    public static void invoke(String server, String port, String name
        try {
            //Use the Apache Xerces SAX Driver
            XmlRpc.setDriver("org.apache.xerces.parsers.SAXParser");

            //Specify the server
            XmlRpcClient client = new XmlRpcClient("http://" + server + ":"

            //create request
            Vector params = new Vector();
            params.addElement(name);

            //make a request and print the result
            String result = (String) client.execute("hello.sayHello", paran
                System.out.println("hello.sayHello: " + result);

```

```

        } catch (ClassNotFoundException e) {
            System.out.println("Could not locate SAX Driver");
        } catch (MalformedURLException e) {
            System.out.println(
                "Incorrect URL fro xml-rpc server foramt:" + e.getMessage()
            )
        } catch (XmlRpcException e) {
            e.printStackTrace();
            System.out.println("XmlRpcException :" + e.getMessage())
        } catch (IOException e) {
            System.out.println("IOException:" + e.getMessage());
        } catch (Exception e){
            e.printStackTrace();
        }
    }

    public static void main(String[] args){
        if( args == null || args.length < 2 ){
            System.out.println("Usage: java HelloClient [server] [port] [yourn
                System.exit(1);
            }
            invoke(args[0], args[1], args[2]);
        }
    }
}

```

## 7, 使用servlet的例子

1, handler object不变

2, 建立XmlRpcFacade  
package helloxmlrpc;

```

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.io.OutputStream;
import org.apache.xmlrpc.XmlRpcServer;

```

```

public class XmlRpcFacade {
    private static XmlRpcServer xmlrpc;
    static{

```

```

        xmlrpc = new XmlRpcServer();

        //登记你的handler object
        xmlrpc.addHandler("hello", new HelloHandler());
    }

    public void execute(HttpServletRequest request, HttpServletResponse res
        s
        IOException {
        byte[] result = xmlrpc.execute(request.getInputStream());

        response.setContentType("text/xml; charset=GB2312");
        response.setContentLength(result.length);
        OutputStream out = response.getOutputStream();
        out.write(result);
        out.flush();
        out.close();
    }
}

3 , 建立servlet
package hellpxmlrpc;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;

public class XmlRpcServlet extends HttpServlet {
    private static final String CONTENT_TYPE = "text/html; charset=
        private XmlRpcFacade facade;

    public void init() throws ServletException {
        facade = new XmlRpcFacade();
    }

    //Process the HTTP Get request
    public void doGet(HttpServletRequest request, HttpServletResponse resp
        ServletException, IOException {
        this.doService(request, response);
    }
}

```

```

    }

    public void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        this.doService(request, response);
    }

    public void doService(HttpServletRequest request,
        HttpServletResponse response) throws ServletException,
        IOException {
        facade.execute(request, response);
    }

    //Clean up resources
    public void destroy() {
    }
}

```

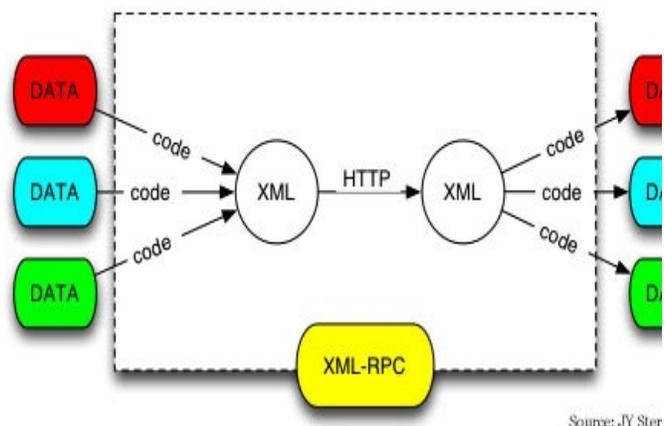
4, 客户程序和内建http server类似, 只需将先下面这句  
`XmlRpcClient client = new XmlRpcClient("http://" + server + ":" + port);`  
 改为  
`XmlRpcClient client = new XmlRpcClient(<servletURL>);`  
 即可

希望这篇文档能对你有小小帮助。更详细的信息可以到<http://ws.apache.org> 解。

## XML-RPC 之 Apache XML-RPC 5

作者：王恩建 来源：<http://www.sento>

XML-RPC 是工作在 Internet 上的远程过程调用协议。通俗点讲，就是 XML 文件。XML-RPC 具体的规范说明请参考[这里](#)。



图片来自XML-RPC官方网站

XML-RPC 规范定义了六种数据类型，下表是这六种数据类型与 Java 的

XML-RPC	Java
<i4> 或 <int>	int
<boolean>	boolean
<string>	java.lang.Str
<double>	double
<dateTime.iso8601>	java.util.Da
<struct>	java.util.Hasht
<array>	java.util.Vec
<base64>	byte[ ]

XML-RPC 规范的各种平台都有具体实现，XML-RPC 规范的 Java 实现 [Apache XML-RPC](#)。

## XML-RPC 服务端实现

先定义一个简单业务对象 MyHandler ，远程客户端将调用该对象的方法

```
package net.sentom.xmlrpc;

public class MyHandler {

    public String sayHello(String str){
        return "Hello," + str;
    }
}
```

然后定义一个 Servlet 名叫 MyXmlRpcServer ，远程客户端通过 HTTP

```
package net.sentom.xmlrpc;
import java.io.IOException;
import java.io.OutputStream;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.apache.xmlrpc.XmlRpcServer;

public class MyXmlRpcServer extends HttpServlet {
    public void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        XmlRpcServer xmlrpc = new XmlRpcServer();
        xmlrpc.addHandler("myHandler", new MyHandler());
        byte[] result = xmlrpc.execute(request.getInputStream());
        response.setContentType("text/xml");
        response.setContentLength(result.length);
        OutputStream out = response.getOutputStream();
        out.write(result);
        out.flush();
    }
}
```

```
}  
}
```

需要特别说明是：

```
xmlrpc.addHandler("myHandler", new MyHandler());
```

为了便于理解，这里可以看成普通的：

```
MyHandler myHandler = new MyHandler();
```

最后在web.xml文件中加入以下几行：

```
<servlet>  
    <servlet-name>MyXmlRpcServer</servlet-name>  
    <servlet-class>net.sentom.xmlrpc.MyXmlRpcServer</servlet-class>  
</servlet>  
<servlet-mapping>  
    <servlet-name>MyXmlRpcServer</servlet-name>  
    <url-pattern>/MyXmlRpcServer</url-pattern>  
</servlet-mapping>
```

## XML-RPC 客户端实现

客户端相对简单一些，先来一个 Java 客户端实现 MyXmlRpcClient:

```
package net.sentom.xmlrpc;  
import java.io.IOException;  
import java.net.MalformedURLException;  
import java.util.Vector;  
import org.apache.xmlrpc.XmlRpcClient;  
import org.apache.xmlrpc.XmlRpcException;  
  
public class MyXmlRpcClient {  
    public static void main(String[] args) {
```

```
try {
    XmlRpcClient xmlrpc = new XmlRpcClient("http://localhost:
    Vector params = new Vector();
    params.addElement("Tom");
    String result = (String) xmlrpc.execute("myHandler.sayHello'
    System.out.println(result);
} catch (MalformedURLException e) {
    System.out.println(e.toString());
} catch (XmlRpcException e) {
    System.out.println(e.toString());
} catch (IOException e) {
    e.printStackTrace();
}
}
```

http://localhost:8080/XMLRPC/MyXmlRpcServer 为 MyXmlRpcS

```
String result = (String) xmlrpc.execute("myHandler.sayHello",params);
```

再来一个 Python 客户端实现

```
import xmlrpclib
url = 'http://localhost:8080/XMLRPC/MyXmlRpcServer';
server = xmlrpclib.Server(url);
print server.myHandler.sayHello('Tom');
```

参考资料

XML-RPC HOWTO  
Python 2.4 Documentation

## [BossConnector笔记]第2章XML-RPC的HelloWorld

作者： 来自： 阅读次数： 10 [大 中 小]

---

\*\*\*\*\*

作者：陈刚，程序员，广西省桂林人，广西师范大学  
届毕业。

blog： <http://blog.csdn.net/glchengang>

Email： [glchengang@yeah.net](mailto:glchengang@yeah.net)

\*\*\*\*\*

# 第2章 XML-RPC的Hello World 例

## 2.1 前言

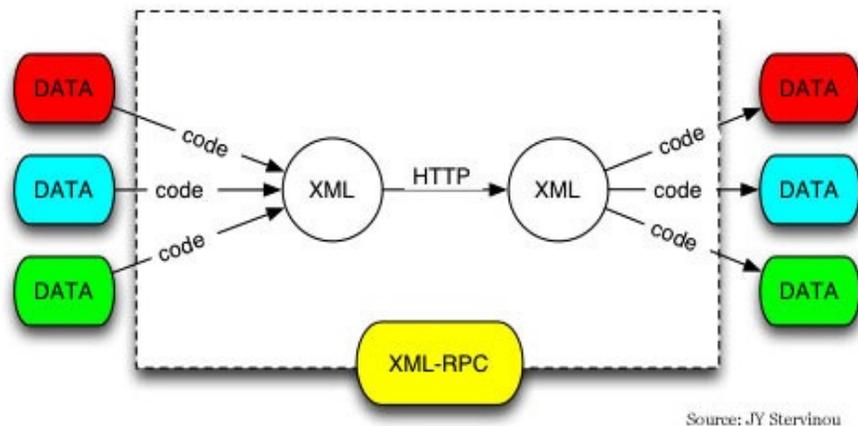
经过研究以及和BOSS系统的开发方沟通，发现此BOSS系统采用纯HTTP + POST+XML的方式来进行信息沟通：XML是数据载体、HTTP是传递的协议、POST是传递的方式。也就是说Web Service没什么关系。Web Service现在有两种不同实现方式和XML - RPC，XML-RPC（RPC是远程调用的意思）是微软源的产品，其实现的低层也是基于HTTP + POST+XML。我本来打算用纯Servlet就解决这个项目了，但老大说这扩展性太差，因为福建是纯HTTP、也许广东就是SOAP的方式，而且其中还存在一些构架设计等等方面的限制。

最后，我得到的要求和提示就是基于XML - RPC来实现这个项目，但XML - RPC是远程调用的方式，而福建BOSS系统并未采用，所以需要修改一下XML - RPC的源代码（老大花了一周多改好了，而我花了一周多去研读和完整，水平不是一般的呀，郁闷一下）。修改后的XML - RPC，在远程调用之外，再提供了一个纯HTTP + POST的现实方式。

注：本篇不涉及到修改XML - RPC的源代码。

## 2.2 XML-RPC简介

XML - RPC是Apache中Web Service方面的一个项目，地址：<http://ws.apache.org/xmlrpc/>，其中的RPC - remote procedure call是远程调用的意思，它本身是通过HTTP传输，用XML做传递信息的载体。XML - RPC是一个规范化的软件框架，也是Web Service的最早的实现，不过现在似乎比XML - RPC更流行。



上图反映了XML - RPC的传输方式：数据（DATA）和的方法被封装到一个XML中，然后由XML - RPC简单 + POST到服务端。服务端也装有XML - RPC来接收XML文件，并将接收到的XML信息进行分解，然后根据记载的信息调用服务端的相应的方法。



## 2.3 XML - RPC的下载

### 1、我的环境

WindowsXP SP2 + JDK1.4.2\_6 + Eclipse3.0.1

### 2、XML - RPC

1 版本：v1.2-b1

1 文件名：xmlrpc-1.2-b1.zip（下载后还要解压一到xmlrpc-1.2-b1.jar）

1 下载：<http://apache.justdn.org/ws/xmlrpc/>

1 其它：为了追踪XML - RPC的代码执行过程，的源代码包一起下了，文件名：xmlrpc-1.2-b1-src

### 3、SAX实现包：xerces

因为XML-RPC中涉及到XML文件的解析等操作，它采用SAX方式，所以它需要一个SAX实现包的支持。SAX实现包有很多种，我选择了xerces

1 版本：v1.4.4（版本老了点，是2001年的，不  
了）

1 文件名：Xerces-J-bin.1.4.4.zip（下载后还要解压  
到xerces.jar）

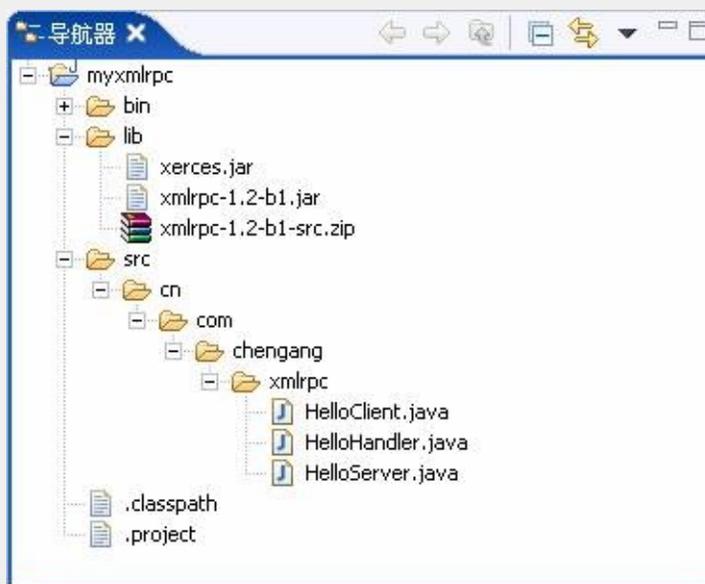
1 下载：<http://xml.apache.org/dist/xerces-j>

## 2.4 Hello World实例

我一直认为实例和代码是最好的说明，还是少说废话，简单的Hello World实例来体会一下吧。

### 2.4.1 创建项目

由于我用的是Eclipse，所以先创建一个普通的JAVA项目myxmlrpc，然后将xerces.jar、xmlrpc-1.2-b1.jar复制到lib目录。再创建一个包：cn.com.chengang.xmlrpc，以后就在这个包下写程序。如下图：



### 2.4.2 创建服务端程序

## 1、HelloServer

```
package cn.com.chengang.xmlrpc;

import org.apache.xerces.parsers.SAXParser;

import org.apache.xmlrpc.WebServer;

import org.apache.xmlrpc.XmlRpc;

public class HelloServer {

public static void main(String[] args) {

//使用Xerces的XML解析器

XmlRpc.setDriver(SAXParser.class);

System.out.println("启动一个WEB SERVER, 端口号 : 8989");
```

```
WebServer server = new WebServer(8989);  
  
    server.start();  
  
//将HelloHandler类的实例绑定到WEB SERVER上  
    该处理类的id标识，在客户端调用时要用得到  
  
    server.addHandler("hello_id", new HelloHandle  
  
        }  
  
    }
```

说明：

这个文件主要是启动了一个Web Server，并将一个HelloHandler类加到server中。

## 2、HelloHandler

```
package cn.com.chengang.xmlrpc;
```

```
public class HelloHandler {  
  
    public String sayHello(String name) {  
  
        return "Hello World, " + name;  
  
    }  
  
}
```

说明：

HelloHandler类不需要实现或继承任何接口抽象类，它是做为服务器端的处理程序，所有处理逻辑都在Handler实现。我们可以写上很多的Handler类，并且一个Handler可以有许多的方法（这里有一个sayHello方法）。

## 2.4.3 创建客户端程序

### 1、HelloClient

```
package cn.com.chengang.xmlrpc;
```

```
import java.io.IOException;

import java.util.Vector;

import org.apache.xerces.parsers.SAXParser;

import org.apache.xmlrpc.XmlRpc;

import org.apache.xmlrpc.XmlRpcClient;

import org.apache.xmlrpc.XmlRpcException;

public class HelloClient {

public static void main(String args[]) throws XmlRpcE
IOException {

//使用 Apache Xerces SAX 解析器

XmlRpc.setDriver(SAXParser.class);

//定位远程服务器
```

```
XmlRpcClient client = new XmlRpcClient("http://127.0.0.1:8989");
```

//创建调用请求，方法的参数列表用一个Vector对象存储。

```
Vector params = new Vector();
```

```
params.addElement("ChenGang");
```

//发出请求，并返回结果，execute需要两个参数，第一个参数用“Handler的标识名.方法名”，第二参数是一个刚刚创建的对象

```
String result = (String) client.execute("hello_id.sayHello", params);
```

```
System.out.println("服务器的返回值: " + result);
```

```
}
```

```
}
```

说明：

1 客户端的程序比较简单，主要是通过XmlRpcClient调用的服务器端Handler的方法名（Handler的标识），法的参数（params），传递给服务端。

1 在这里127.0.0.1是指本机，因为服务端和客户端都运行的，共用同一台电脑。实际运行时，应该是分属在上的，这时把127.0.0.1改成真实的服务器IP即可

1 8989就是HelloServer类中启动Web Server的端

1 有人说这里并没有看到XML文件的影子呀。  
client.execute("hello\_id.sayHello", params);这一句已经封的处理过程，表面上它只是方法名和方法参数，但内部会将这两者处理成一个XML文件，然后POST到服务

1 params.addElement("ChenGang");的参数不能是中  
报错。暂未找到解决方法。

## 2.4.4 运行

### 1、启动服务

以Application方式来运行HelloServer。Eclipse的“控制台”  
下图：



## 2、运行客户端

以Application方式来运行HelloClient。Eclipse的“控制台”  
下图：



## 2.5 XML-RPC原理分析

### 2.5.1 XML - RPC所传输的XML文件

在上面的例程中说了，XML - RPC将远程方法的调用封装成一个XML文件传给服务器端的，那么我们就来看看XML的样子是怎么样的。

#### 1、启动监听器

在上一篇所说的SOAP中有一个监听器，我们把它运行个监听器可以监听本机上发送到某一个端口的信息。这的入口类为TcpTunnelGui，启动此监听器的命令如

```
java org.apache.soap.util.net.TcpTunnelGui 8070 localhost
```

如果报以下错误

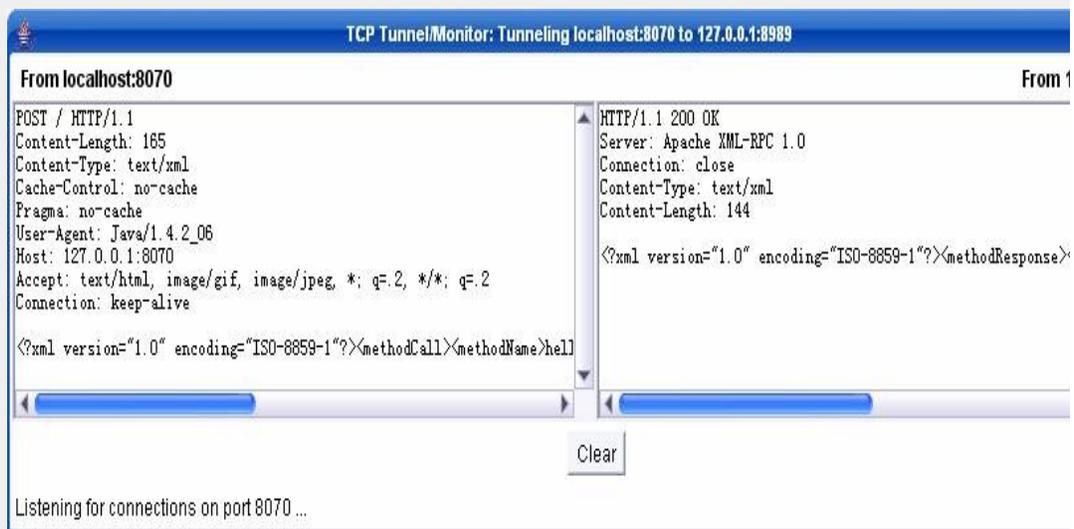
```
Exception in thread "main" java.lang.NoClassDefFoundError: org/apache/soap/util/net/TcpTunnelGui
```

这是因为soap.jar没有定义在classpath变量中，你可以将入到系统变量classpath，也可以在命令中用-cp参数指定soap.jar的位置。

```
java -cp "D:\soap-2_3_1\lib\soap.jar"
org.apache.soap.util.net.TcpTunnelGui 8070 localhost
```

这样所有发向8070端口的信息将被截获，然后这个监听器获取的信息转发到8989端口。此监听器的界面如下，左边信息，右边是从服务端接收的信息。

注意：HelloClient类中的服务器端口号要由8989成8070。HelloServer还是原来的8989，不用改。



## 2、客户端发出的XML信息

POST / HTTP/1.1

Content-Length: 165

Content-Type: text/xml

Cache-Control: no-cache

Pragma: no-cache

User-Agent: Java/1.4.2\_06

Host: 127.0.0.1:8070

Accept: text/html, image/gif, image/jpeg, \*; q=.2, \*/\*;

Connection: keep-alive

```
<?xml version="1.0" encoding="ISO-8859-1"?><method
<methodName>hello_id.sayHello</methodName><pa
<param><value>ChenGang</value></param></para
```

```
</methodCall>
```

XML的信息是一行，我把它格式化一下，如下

```
<?xml version="1.0" encoding="ISO-8859-1"?>  
  
  <methodCall>  
  
    <methodName>hello_id.sayHello</methodName>  
  
    <params>  
  
      <param>  
  
        <value>ChenGang</value>  
  
      </param>  
  
    </params>  
  
  </methodCall>
```

### 3、服务端反馈的信息

HTTP/1.1 200 OK

Server: Apache XML-RPC 1.0

Connection: close

Content-Type: text/xml

Content-Length: 144

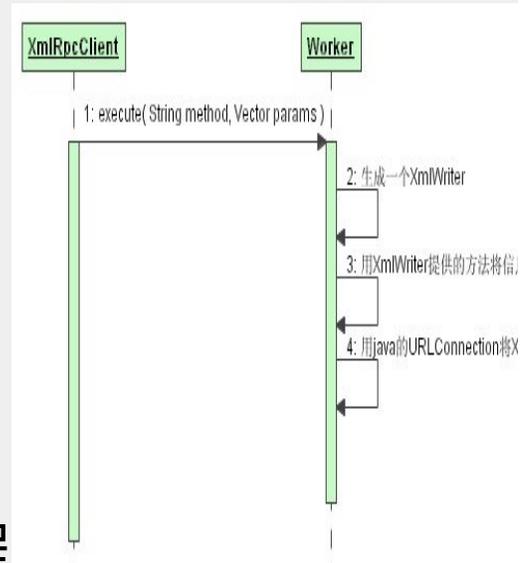
```
<?xml version="1.0" encoding="ISO-8859-1"?><methodResponse>  
  <params><param><value>Hello World, ChenGang</value></param></params></methodResponse>
```

XML的信息是一行，我把它格式化一下，如下

```
<?xml version="1.0" encoding="ISO-8859-1"?>
```

```
<methodResponse>
```

```
<params>
  <param>
    <value>Hello World, ChenGang</value>
  </param>
</params>
</methodResponse>
```



## 2.5.2 客户端的处理流程

在HelloClient的代码中，我们看到是通过XML-RPC XmlRpcClient类来处理客户端的请求的，XmlRpcClient处理逻辑如上，Worker是XmlRpcClient的一个内部类。W

是将“要调用的远程方法以及方法参数”写成一个XML文件的格式在上面“2.5.1 2、客户端发出的XML信息”中了。

### 2.5.3 服务器端的处理流程(省略)

我修改的XML - RPC源代码不是这个版本的，而是从CVS上直接check下来的，最新的开发中，其设计已经有些变化。比如，客户端的XML发送，原版是用HTTP+GET的，最新的则将它封成了一个接口：XmlRpcTransport，现在是HTTP + POST，另一个实现则是EMAIL的方式，灵活性就更大了。

## 2.6 参考资料

xml-rpc入门例程及一个通用服务器（CSDN），  
址：<http://dev.csdn.net/develop/article/39/article/38/3830>

## [原创]xml-rpc入门例程及一个通用服务器

### 一，准备过程

远程过程调用RPC，基于XML的传输方式，当然低层API，就不用我们操心了，但必须的软件还是要有的，先给个列表清单

JDK1.4.2 不用说了

Xerces解析器 到<http://xml.apache.org/>上去下载吧，

XML-RPC开发包, <http://ws.apache.org/xmlrpc/>上可以下得到

将以上所有的jar包放到开发环境的classpath中。

### 二，Hello World

XML-RPC如果想跑起来，最后需要四个组件，WEB server, 服务器类，处理类，客户类

#### 1.WEB Server.

在我们已经下载的XML-RPC包中就有一个轻型级的WEB SERVER。

在程序中，我们只需要简单的用以下语句就可以启动。

//建立一个对象，传输一个端口

```
WebServer server = new
```

```
WebServer(Integer.parseInt("8989"));
```

//启动

```
server.start();
```

#### 2.编写处理类

处理类相当RMI中的远程类，只不过在这里更轻量类，无须任何接口。

在这个类中包含一个或多个公有方法以供远程的客户端来调用。

```
public class HelloHandler {
```

```
public String sayHello(String name) {
```

```
return "Hello " + name;
}

}
```

### 3.服务器

负责调用以上代码来启动用服务器，同时还要将远程对象绑定到该服务器上。

```
import java.io.IOException;
//引入必须的包，当然你的xml-rpc的包应该在
classpath中
import org.apache.xmlrpc.WebServer;
import org.apache.xmlrpc.XmlRpc;
public class HelloServer {
    /**
    主方法
    */
    public static void main(String[] args) {

        try {
            // 使用Xerces的XML解析器

            XmlRpc.setDriver("org.apache.xerces.parsers.SAXI
            // 给出提示，并在端8989上启动服务器
            System.out.println("Starting XML-RPC
            Server...");
            WebServer server = new
            WebServer(Integer.parseInt("8989"));
            server.start();

            // 将HelloHandler类的实例绑定到WEB
            SERVER上，hello是该处理类的标识，在客户端
            调用时要用得到
            server.addHandler("hello", new
            HelloHandler());
```

```

        System.out.println(
            "Registered HelloHandler class to
            \"hello\");

    } catch (ClassNotFoundException e) {
        System.out.println("Could not locate SAX
        Driver");
    } catch (Exception e) {
        System.out.println("Could not start
        server: " +
            e.getMessage());
    }
}
}
}

```

#### 4. 客户端

根据“标识名.方法名”来定位远程的处理方法。

```

import java.io.IOException;
import java.net.MalformedURLException;
import java.util.Vector;

//导入必须的包
import org.apache.xmlrpc.XmlRpc;
import org.apache.xmlrpc.XmlRpcClient;
import org.apache.xmlrpc.XmlRpcException;

```

```

public class HelloClient {

```

```

    public static void main(String args[]) {

```

```

        String yourname="liu xiaobai";

```

```

        try {

```

```

            // 使用 Apache Xerces SAX 解析器

```

```

            XmlRpc.setDriver("org.apache.xerces.parsers.SAXI

```

```
        // 定位远程服务器，http://主机地址:端口号，8989是上文服务器启动时用的端口
        XmlRpcClient client =
            new
        XmlRpcClient("http://localhost:8989/");

        // 创建调用请求，方法的参数列表如果一个Vector对象来存储。
        Vector params = new Vector();
        params.addElement(yourname);

        // 发出请求，并返回结果,execute需要两个参数，第一个参数用“标识名.方法名”，第二参数是一个刚刚建立的向量对象

        String result =
            (String)client.execute("hello.sayHello",
        params);
        System.out.println("Response from
        server: " + result);

        } catch (ClassNotFoundException e) {
            System.out.println("Could not locate SAX
        Driver");
        } catch (MalformedURLException e) {
            System.out.println(
                "Incorrect URL for XML-RPC server
        format: " +
                e.getMessage());
        } catch (XmlRpcException e) {
            System.out.println("XML-RPC Exception:
        " + e.getMessage());
        } catch (IOException e) {
            System.out.println("IO Exception: " +
```

```
e.getMessage());  
    }  
}  
}
```

5,编译以上代码，要确保解析器，XMP-RPC开发包的jar文件在classpath中。

运行服务器

```
java HelloServer
```

运行客户端

```
java HelloClient
```

6.一个通用的XML服务器

功能描述：通过配置文件来配置要加载的处理器

1.配置文件的名称及位置

名字：config.properties

类文件的根目录中，如果你类是默认包，那么就是同一个目录；如果类的包名是com.hello,那么该文件应该与com目录放在同一级中。

内容：

```
#标识名=类名  
hello=javaxml2.HelloHandler
```

2.通用的源代码

```
import java.io.*;  
import org.apache.xmlrpc.*;  
import java.util.Properties;  
import java.util.Enumeration;  
import java.util.Hashtable;  
public class MyLightXMLServer  
{  
    private WebServer server;
```

```

private int port;
private String configfile;

public MyLightXMLServer(int port,String
config)
{
    this.port=port;
    this.configfile=config;
}
//启动服务器
public void start() throws
IOException,ClassNotFoundException,Exception
{
    XmlRpc.setDriver("org.apache.xerces.parsers.SAXI
System.out.println("starting up xml-rpc
server...");
    server=new WebServer(port);
    //调用注册函数
    registerHandlers(this.getHandlers());
    server.start();
}

public void registerHandlers(Properties
handlers) throws Exception
{
    Enumeration
enum=handlers.propertyNames();
    while(enum.hasMoreElements())
    {
        String temp=(String)enum.nextElement();
        String tempcls=(String)handlers.get(temp);
        Class cls=Class.forName(tempcls);
        server.addHandler(temp,cls.newInstance());
    }
}
}

```

```

public Properties getHandlers()
{
    try
    {
        Properties properties=new Properties();
        properties.load(new FileInputStream(new
File("config.properties")));

        return properties;
    }
    catch(Exception e)
    {
        e.printStackTrace();
    }
    return null;
}

public static void main(String args[])
{
    String port="8989";
    String configfile="";
    MyLightXMLServer server=new
MyLightXMLServer(Integer.parseInt(port),configfile)
    try
    {
        server.start();
    }
    catch(Exception e)
    {e.printStackTrace();}
}
}

```

将MyLightXMLServer .java编译并运行之后，该服务器会在配置文件中加载该处理器

然后可以直接执行HelloClient

```
java HelloClient
```

本文引用通告地址：

<http://blog.csdn.net/lxblg/services/trackbacks/11023>

## XML配置文件的读取处理

2004-12-12 板桥里人 【打印】 【关闭】 【双击滚屏】

---

### XML配置文件的读取处理

Java和XML是黄金组合,网上已经有很多文章介绍,XML作为电子商务中数  
经有其不可替代的作用,但是在平时系统开发中,我们不一定都用到数据交互  
无法使用XML了?

当然不是,现在已经有一个新趋势,java程序的配置文件都开始使用XML格式  
使用类似windows的INI格式.(Java中也有 Propertiesy这样的类专门处理这样  
配置文件).使用XML作为Java的配置文件有很多好处,从Tomcat的安装配置文  
的配置文件中,我们已经看到XML的普遍应用,让我们也跟随流行趋势用XML  
来.

现在关键是如何读取XML配置文件?有好几种XML解析器:主要有DOM和SAX  
区别网上文章介绍很多.

在apache的XML项目组中,目前有Xerces Xalan Cocoon几个开发XML相关打  
project.Tomcat本身使用的是 Sun 的 JAXP,而其XSL Taglib project中使用Xe  
器.

好了,上面都是比较烦人的理论问题,还是赶快切入XML的配置文件的读取

在我们的程序中,通常要有一些根据主机环境确定的变量.比如数据库访问代  
码,不同的主机可能设置不一样.只要更改XML配置文件,就可以正常运行.

```
<myenv>

<datasource>
<dbhost>localhost</dbhost>
<dbname>sqlname</dbname>
<dbuser>username</dbuser>
<dbpassword>password</dbpassword>
</datasource>

</myenv>
```

上面这个myenv.xml配置文件一般是放在tomcat的WEB-INF/classes目录下.

我们编制一个Java程序直接读取,将dbhost dbuser dbpassword提取出来供其问数据库用.

目前使用SAX比较的多,与DOM主要区别是 SAX是一行一行读取XML文件适合比较大文件,DOM是一次性读入内存,显然不能对付大文件.这里我们解析,由于SAX解析器不断在发展,网上有不少文章是针对老版本的.如果你使可以参考 使用SAX处理XML文档一文.这里的程序是根据其改进并且经过得来的.

对上面myenv.xml读取的Java程序:

```
import org.xml.sax.Attributes;
import org.xml.sax.helpers.DefaultHandler;
import org.xml.sax.SAXException;
import java.util.Properties;

//使用DefaultHandler的好处 是 不必陈列出所有方法,
public class ConfigParser extends DefaultHandler {

    ///定义一个Properties 用来存放 dbhost dbuser dbpassword的值
    private Properties props;

    private String currentSet;
    private String currentName;
    private StringBuffer currentValue = new StringBuffer();

    //构建器初始化props
    public ConfigParser() {

        this.props = new Properties();
    }

    public Properties getProps() {
```

```

return this.props;
}

//定义开始解析元素的方法. 这里是将<xxx>中的名称xxx提取出来.
public void startElement(String uri, String localName, String qName, Attribute
throws SAXException {
currentValue.delete(0, currentValue.length());
this.currentName =qName;

}

//这里是将<xxx></xxx>之间的值加入到currentValue

public void characters(char[] ch, int start, int length) throws SAXException {

currentValue.append(ch, start, length);

}

//在遇到</xxx>结束后,将之前的名称和值一一对应保存在props中

public void endElement(String uri, String localName, String qName) throws
SAXException {

props.put(qName.toLowerCase(), currentValue.toString().trim());
}

}

```

上面的这个解析程序比较简单吧? 其实解析XML就是这么简单.

现在我们已经将dbhost dbuser dbpassword的值localhost sqlname username p;取了出来.但是这只是在在解析器内部,我们的程序还不能访问.需要再编制

```

import java.util.Properties;
import javax.xml.parsers.SAXParser;

```

```

import javax.xml.parsers.SAXParserFactory;
import java.net.URL;

public class ParseXML{

//定义一个Properties 用来存放 dbhost dbuser dbpassword的值
private Properties props;

//这里的props
public Properties getProps() {
return this.props;
}

public void parse(String filename) throws Exception {

//将我们的解析器对象化
ConfigParser handler = new ConfigParser();

//获取SAX工厂对象
SAXParserFactory factory = SAXParserFactory.newInstance();
factory.setNamespaceAware(false);
factory.setValidating(false);

//获取SAX解析
SAXParser parser = factory.newSAXParser();

//得到配置文件myenv.xml所在目录. tomcat中是在WEB-INF/classes
//下例中BeansConstants是用来存放xml文件中配置信息的类,可以自己代替
URL confURL = BeansConstants.class.getClassLoader().getResource(filename

try
{
//将解析器和解析对象myenv.xml联系起来,开始解析
parser.parse(confURL.toString(), handler);
//获取解析成功后的属性 以后 我们其他应用程序只要调用本程序的props就
出属性名称和值了
props = handler.getProps();
}finally{
factory=null;
}
}

```

```
parser=null;
handler=null;
}

}

}
```

由于我们的XML文件是使用最简单的形式,因此解析器相对简单,但是这已付我们的配置文件了.

判断一个程序系统的先进性,我们先看看他的配置文件,如果还在使用老套的这样类似.ini的文件,我们也许会微微一笑,他又落伍了.....

责任丝

XML配置文件的读取(sax)

作者: yi5      转贴自: yi5\_yuyu      点击数: 12      文章录入: yuyu3007

在最近的一个MIS项目中, 为了避免硬编码, 我需要把一些配置信息写在一个配置文·  
好像都是xml文件了, 再用传统ini文件是不是有点落伍了?

ok, 就用xml做配置文件吧.

我的配置文件reportenv.xml如下, 比较简单:

```
<?xml version="1.0" encoding="utf-8"?>
<reportenv>
<datasource>
<username>sqlname</username>
<password>password</password>
</datasource>
</reportenv>
```

现在的问题是我用什么来读取配置信息?

现在流行的是dom4j和sax, 我以前一直用dom4j. 可是weblogic workshop自带  
第一步: ConfigParser.java

```
/*
 * Create Date: 2005-6-13
 * Create By: 板桥里人
 * purpose: xml配置文件属性读取器
 */
package com.infoearth.report;

import org.xml.sax.Attributes;
import org.xml.sax.helpers.DefaultHandler;
import org.xml.sax.SAXException;
import java.util.Properties;

public class ConfigParser extends DefaultHandler {

    ////定义一个Properties 用来存放属性值
    private Properties props;

    private String currentSet;
    private String currentName;
    private StringBuffer currentValue = new StringBuffer();

    //构建器初始化props
    public ConfigParser() {
```

```

        this.props = new Properties();
    }

    public Properties getProps() {
        return this.props;
    }

    //定义开始解析元素的方法. 这里是将<xxx>中的名称xxx提取出来.
    public void startElement(String uri, String localName, S
throws SAXException {
        currentValue.delete(0, currentValue.length());
        this.currentName =qName;
    }

    //这里是将<xxx></xxx>之间的值加入到currentValue
    public void characters(char[] ch, int start, int length)
        currentValue.append(ch, start, length);
    }

    //在遇到</xxx>结束后,将之前的名称和值一一对应保存在props中
    public void endElement(String uri, String localName, Str
        props.put(qName.toLowerCase(), currentValue.toString
    }

}

```

第二步:ParseXML.java

```

/*
 * Create Date: 2005-6-13
 * Create By: 板桥里人 李春雷修改
 * purpose:xml配置文件属性读取器(通用),
 */

package com.infoearth.report;

import java.util.Properties;
import javax.xml.parsers.SAXParser;
import javax.xml.parsers.SAXParserFactory;
import java.net.URL;

public class ParseXML{

```

```

//定义一个Properties 用来存放属性值
private Properties props;

public Properties getProps() {
    return this.props;
}

public void parse(String filename) throws Exception {
    //将我们的解析器对象化
    ConfigParser handler = new ConfigParser();
    //获取SAX工厂对象
    SAXParserFactory factory = SAXParserFactory.newInstance();
    factory.setNamespaceAware(false);
    factory.setValidating(false);
    //获取SAX解析
    SAXParser parser = factory.newSAXParser();
    try{
        //将解析器和解析对象xml联系起来,开始解析
        parser.parse(filename, handler);
        //获取解析成功后的属性
        props = handler.getProps();
    }finally{
        factory=null;
        parser=null;
        handler=null;
    }
}
}
}

```

第三步:ReadConfigXml.java

```

/*
 * Create Date: 2005-6-13
 * Create By: 李春雷
 * purpose:xml配置文件属性读取器
 */

package com.infoearth.report;

import java.util.Properties;

public class ReadConfigXml
{
    private Properties props;
}

```

```

    public ReadConfigXml(String url){
ParseXML myRead = new ParseXML();
    try {
        myRead.parse(url);
                props = new Properties();
                props = myRead.getProps();
    } catch (Exception e) {
        e.printStackTrace();
    }
    }
    public String getUsername(){
        return props.getProperty("username");
    }
    public String getPassword(){
        return props.getProperty("password");
    }
}
}

```

ok, 搞定了, 读取的时候如下:

```

ReadConfigXml xmlread = new ReadConfigXml("reportenv.xml");
String username = xmlread.getUsername();
String password = xmlread.getPassword();

```

前两个类实现了xml文档属性设置的任意读取. 只要是xml的属性值, 都读到了prop  
第三个类是我针对我的xml文件写的, 似乎有点多余. 呵呵. 其实有难言之隐. 因为不足了一下.

另外, 感谢j道, 感谢板桥里人.

## XML入门精解之文件格式定义 ( DTD )

DTD实际上可以看作一个或多个XML文件的模板，这些XML文件中的元素属性、元素的排列方式/顺序、元素能够包含的内容等，都必须符合DTD中XML文件中的元素，即我们所创建的标记，是根据我们应用的实际情况来创建一份完整性高、适应性广的DTD是非常困难的，因为各行各业都有他行业特点，所以DTD通常是以某种应用领域为定义的范围，如：医学、建筑政。DTD定义的元素含盖范围越广泛，那么就越复杂。

DTD可以是一个完全独立的文件，也可以在XML文件中直接设定。所为外部DTD（在XML文件中调用另外已经编辑好的DTD）和内部DTD（在XML文件中直接设定DTD）两种。比如，有几十家相互联系的、合作伙伴关系的公司他们相互之间的交换电子文档都是用XML文档。那么我们可以将这些XML放在某个地方，让所有交换的XML文档都使用此DTD，这是最方便的做法用于公司内部XML文件使用。

### 内部DTD

内部DTD是在XML文件的文件序言区域中定义的。语法：

```
!DOCTYPE element-name[.....  
  
]
```

!DOCTYPE : 表示开始设定DTD，注意DOCTYPE是大写

Element-name : 指定此DTD的根元素的名称，一个XML文件只能有一个根元素。注意，如果XML文件使用了DTD，那么文件中的根元素就在这里

[.....] : 在[ ]标记里面定义XML文件使用元素，然后用 ] 结束DTD

下面，我们来看一下怎样给XML文件定义DTD，请见例1。

例1中的DTD定义区可以看作是一个DTD定义的大概框架，为其他XML文件定义DTD，结构和例1的DTD差不多，只是可能需要添加、删除或者更改一些

在DTD定义的中间是元素设定，这是一个DTD的最主要部分，其主要语

**!ELEMENT element-name element-definition**

**!ELEMENT** : 表示开始元素设置，注意此处ELEMENT关键字是

**element-name** : 表示要设置的元素的名称。

**element-definition** : 指明要对此元素进行怎样的定义，就是说元素之间能够包含什么内容，是其他元素还是一般性的文字。

在例1中，大家可以看到，**!ELEMENT 参考资料(书籍)** 这个元素设“参考资料”这个元素，并且它是作为“书籍”这个元素的父元素。**!ELEMENT 名称,作者,价格)** 这个元素设定声明了“书籍”这个元素，并且它是作为“名称”、“价格”这三个元素的父元素。而**!ELEMENT 名称(#PCDATA)** 这个明了“名称”这个元素，但是此元素仅仅包含一般文字，是基本元素，这是A关键字定义的。

在元素设置中，如果元素包含多个子元素，如：**!ELEMENT 书籍(名称,作者,价格)** 这种含多个子元素的声明，那么“名称”、“作者”、“价格”这些标记在必须以上面排列的顺序出现，每个标记必须而且只能够出现一次。如果在，按照**!ELEMENT 书籍ANY**，这样在元素下就可以包含任意被设定过出现的次数和顺序也不受限制，并且在该元素下，除了可以包含子元素以外含一般的文字。有时候，在XML文件中，一个标记可能多次出现（或者不么我们除了在它们的父元素中用ANY关键字之外，还可以在元素的旁边加号来控制标记出现的次数。这些符号见表1。

表1

符号 代表标记出现的次数

? 不出现或只出现一次

\* 不出现或可出现多次

+ 必须出现一次以上

无符号 只能出现一次

例如：**!ELEMENT 参考资料(书籍)ANY**，**!ELEMENT 名称(#PCDATA)** 这个元素

例如：`!ELEMENT 参考资料(书籍,报纸,杂志,网站)` 这个元素以“”标记在XML文件中可以不出现或者出现多次；“报纸”标记必须出现一次以“”标记可以不出现或只出现一次；而“网站”标记必须出现而且只能出现

在一些父元素的声明中，有可能它包含的子元素是在多个子元素中选用，那么我们声明此父元素时，就可以把它声明成选择性元素，例如：`!ELEMENT 配偶(妻子|丈夫)`。可供选择的子元素用“|”分隔，这样，我们在XML文  
样写：

配偶

丈夫 张三/ 丈夫

/ 配偶

只从中选择一个子元素。

在我们的XML文件中，还可能包括很多“空元素”，即：元素是单独存  
/ 元素 这样的结束标记。那么在DTD中是用EMPTY关键字来声明的  
`!ELEMENT 元素名 EMPTY`。在XML文件中，空元素不需要结束标记，  
空元素名 这样的写法。

在DTD中，还可以声明一些称为Entity的东西，让DTD和XML文件使  
以把Entity看作是一个常量，它有一定的值。在DTD中，Entity的声明  
`!ENTITY entity-name entity-definition`。例如：我们在DTD中声明!  
C "(#PCDATA)"，那么在后面的元素设定中，就可以使用这个Entity来  
DATA)"这个字符串，如：`!ELEMENT 作者(#PCDATA)`可以写成!  
者 &&PC;。引用Entity的时候，必须要在Entity名称前面加上“&&”符  
上“;”符号。

在例1中，`!ATTLIST 价格 货币单位 CDATA#REQUIRED` 这一句  
的属性，关于元素属性设置的语法为：`!ATTLIST element-name attribu  
Type Default-value`。其中，`!ATTLIST` 是开始属性的设定（注意大  
ment-name是指明此属性设定是针对什么元素的；attribute-name是设

名称；Type是该属性的属性值的类别，属性值有多种，可以是一般的文字属性值中取一种等，属性值的种类见表2。Default-value是指该属性的内有四种不同的属性内定值（见表3）。

下面我们举几个例子来看一下几个常用的元素属性的设定。例

```
!ATTLIST 姓名 性别 (男|女) "男"
```

此元素属性设定是为“姓名”这个元素设定一个名为“性别”的属性，此值类别是Enumerated，取值范围为“男”或者“女”（用“|”分隔）。如果在XML文件中没有为此属性赋值，那么就取值为“男”，因为属性内定值是一个字符

```
!ATTLIST 姓名 号码 ID #REQUIRED
```

该属性设定是为“姓名”元素设定一个名为“号码”的属性，属性值类别是ID，表示在XML文件中为此属性赋值的时候，值在此XML文件中是唯一的，如在XML文件中出现下面的XML语句：

```
姓名 号码="1234567" 张三/ 姓名
```

```
姓名 号码="1234567" 李四/ 姓名
```

注意：“号码”属性的值重复了，这样，在解析过程中将会出现错误信息。属性内定值为#REQUIRED，表示这个属性在XML文件中必须出现，否则解析会发生错误。

```
!ATTLIST 电话号码 国家代码 CDATA #FIX "86"
```

该属性设定是为“电话号码”这个元素设定一个名为“国家代码”的属性，属性值是一般的文字。在“电话号码”标记中不能够设定该属性，因为这个属性是固定值的属性（#FIX关键字），解析器会自动地将该属性以及值“86”加到“电话号码”标记中。

表2

## 属性值类别 描述

`CDATA` 属性值仅仅是一般的文字。

`enumerated` 列出该属性的取值范围，一次只能有一个属性值能够赋予。

`nmToken` 表示属性值只能由字母、数字、下划线、`.`、`:`、`-`这些符号

`nmTokens` 表示属性值能够由多个`nmToken`组成，每个`nmToken`之间用

`id` 该属性在XML文件中是唯一的，常用来表示人的身份证号码。

`idref` 表示该属性值是参考了另一个`id`属性。

`idrefs` 表示该属性值是参考了多个`id`属性，这些`id`属性的值用空格隔

`entity` 表示该属性的设定值是一个外部的`entity`，如一个图片文件

`entities` 该属性值包含了多个外部`entity`，不同的`entity`之间用空格隔

`notation` 属性值是在DTD中声明过的`notation`（声明用什么应用软件解读文件，如图片）。

在XML的规范中，还规定了两个内定的属性，即：`xml:space`和XML内定的属性名称以`xml:`开头，而你自已定义的属性名不能以`xml:`开头，否则发生错误。

我们前面已经讲过，空白格在XML文件是有含义的，`xml:space`属性解析器将XML文件中的空白格传给应用程序后的处理方法。`xml:space`是`enumerated`类型的属性，只能够在`default`和`preserve`之间取值。`xml:space="default"`表示解析器将空白格传递给应用程序后，由应用程序内定的方法来处理这如果没有设定`xml:space`属性，则解析器会默认用`default`来设定该属性。`xml:space="preserve"`是表示解析器将空白格传递给应用程序后，要求应用程序保留空白格。

`xml:lang`属性是用来设置标记中的文字信息是使用哪种语言，ISO-639-1同语言的代表缩写，如：`xml:lang="en"`表示英文；`xml:lang="la"`表示拉丁语；`xml:lang="zh"`表示中文资料；`xml:lang="zh-CN"`表示中文（简体）；`xml:lang="zh-TW"`表示中文（繁体）。系统内定是`xml:lang="en"`，即标记中的信息都是英文。

## 外部DTD

外部DTD是一个独立于XML文件的文件，实际上也是一个文本文件，其文件扩展名为`.dtd`。因为外部DTD独立于XML文件，那么它可以供多个XML

TD为文件扩展名。因为外部DTD独立于XML文件，那么它可以供多个XML文件使用。就像用同一个模板可以写出多个不同内容的文件一样，这多个XML文件因使用同一个外部DTD，所以它们的结构大致相同。

外部DTD的创建方式、语法和内部DTD是一样的，把例1的内部DTD按外部DTD来写，文件如下所示。文件存为后缀名为 .dtd的文件。

```
?xml version="1.0" encoding="GB2312" ?
```

```
!ELEMENT 参考资料(书籍*)
```

```
!ELEMENT 书籍 (名称,作者,价格)
```

```
!ELEMENT 名称 (#PCDATA)
```

```
!ELEMENT 作者 (#PCDATA)
```

```
!ELEMENT 价格 (#PCDATA)
```

```
!ATTLIST 价格 货币单位 CDATA #REQUIRED
```

除了没有内部DTD中的!DOCTYPE 参考资料[.....] 语句外，其他都有关元素数目、排列顺序、空元素设定、选择性元素、Entity声明、属性内部DTD是一样的。

XML文件使用!DOCTYPE element-name SYSTEM DTD-URL  
!DOCTYPE element-name PUBLIC DTD-name DTD-URL 来引用创建  
DTD文件。

### 表3

属性内定值 描述

#required 表示在标记中必须出现此属性。

#implied 标记中可以不出现此属性。

#fix 属性的值是固定的某个值。

字符串 标记中如没有指定属性的值，那么此字符串就是此属性的值。

此语句必须位于XML文件的文件序言区，其中，!DOCTYPE 表示开外部DTD；element-name是指该DTD的根元素的名称；SYSTEM是指文件是私有的，即我们自己创建的，没有公开发行，只是个人或在公司内合作单位之间使用；而PUBIC关键字是指该外部DTD是公用的，经过了PUBLIC的DTD都有一个逻辑名称——DTD-name，我们必须在调用时指明名称。DTD-URL是用URL的方式指明外部DTD文件的位置。例如，我们的文件存放在URL为：http://www.xml.com/这个地方，文件名为ckzl.dtd。

文件中的声明如下：

```
?xml version="1.0" encoding="GB2312" ?
```

```
!DOCTYPE 参考资料SYSTEM "http://www.xml.com/ckzl.dtd"
```

...

### Schema简介

DTD的语法相当复杂，并且它不符合XML文件的标准，自成一个体系介绍也仅仅是作了一个简介，目的是帮助大家能读懂DTD文件以及在必要时DTD文件，因为现在很多的XML应用是建立在DTD之上的。

另外，一个代替DTD的就是W3C定义的Schema，Schema相对于XML的好处是XML Schema文档本身也是XML文档，而不是像DTD一样使用自创的语法。这就方便了用户和开发者，因为可以使用相同的工具来处理XML Schema XML信息，而不必专门为Schema使用特殊工具。Schema简单易懂，懂XML、规则的人都可以立刻理解它。Schema的概念提出已久，但W3C的标准，相应的应用支持尚未完善，但采用Schema已成为XML发展的一个趋势。

### 例1

DTD定义区：

```
?xml version="1.0" encoding="GB2312" ?
```

```
!DOCTYPE 参考资料 [
!ELEMENT 参考资料 (书籍)
!ELEMENT 书籍 (名称,作者,价格)
!ELEMENT 名称 (#PCDATA)
!ELEMENT 作者(#PCDATA)
!ELEMENT 价格 (#PCDATA)
!ATTLIST 价格 货币单位 CDATA#REQUIRED
]
```

参考资料

书籍

名称XML 入门精解/ 名称

作者 张三/ 作者

价格货币单位="人民币"20.00/ 价格

/ 书籍

书籍

名称XML 语法/ 名称

!-- 即将出版 --

作者 李四/ 作者

价横币单位="人民币"18.00/ 价格

/ 书籍

/ 参考资料

# JAVA的XML编程

<http://www.chinaunix.net> 作者:[zlzj2010](#) 发表于: 2003-07-31 10:24:01

XML作为全球通用的结构化语言，越来越受人们青睐，各种开发平台（比如Microsoft Studio系列、Oracle系列、Inprise Borland系列等）也都把支持XML开发作为宣传口号之一。由于笔者所从事的电子政务开发较早的引入了XML,所以尝到了许多甜头，在许多项目中利用XML数据交换信息，省去了许多麻烦事，不用制定繁琐的数据格式，利用XML数据易于表达，也利于一线开发者跟踪调试。

笔者先前也曾发表过相关的文章，比如《简析Delphi中的XML编程》一文，有兴趣的读者可以到Google网（<http://www.google.com>）去搜索一下，有很多媒体转载。今天笔者想探讨的是关于JAVA中的XML编程，希望对正在或想要学习XML编程的新老读者有所帮助。

在XML应用中，最常用也最实用的莫过于XML文件的读写，所以笔者通过一个简单的XML文件读写来作简要分析。可以在任何文本编辑器中先建立如下结构的XML文件，类似于HTML结构，但XML语义比较严格，起始标记必须配对，比如" 学生花名册" 与" /学生花名册" 对应，空格多少可不必在意，但一般都以缩格形式书写，便于阅读。把此文件命名为Input.xml，可以在任何支持XML的浏览器中打开测试一下，如果输入正确，在浏览中可以看到此文件的树形表示结构。如果您还对XML结构感到比较陌生，建议先看看《简析Delphi中的XML编程》一文中关于XML文件的说明。

Input.xml

[code:1:af65f1d5b3]

```
<?xml version="1.0" encoding="GB2312"
?>
  <学生花名册>
    <学生 性别 = "男">
```

```
<姓名>李华</姓名>
  <年龄>14</年龄>
<电话>6287555</电话>
  </学生>
  <学生 性别 = "男">
  <姓名>张三</姓名>
  <年龄>16</年龄>
<电话>8273425</电话>
  </学生>
</学生花名册>
```

```
[/code:1:af65f1d5b3]
```

准备工作做完后，接着就开始写实质性的JAVA代码了。为保存从XML文件读入的信息，需要先建一个简单的Bean来保存学生信息，命名为StudentBean，代码如下所示：

StudentBean.java

```
[code:1:af65f1d5b3]
```

```
public class StudentBean {
  private String sex; //学生性别
  private String name; //学生姓名
  private int age; //学生年龄
  private String phone; //电话号码

  public void setSex(String s) {
    sex = s;
  }
  public void setName(String s) {
    name = s;
  }
  public void setAge(int a) {
    age = a;
  }
  public void setPhone(String s) {
    phone = s;
  }
  public String getSex() {
    return sex;
  }
}
```

```

    }
    public String getName() {
        return name;
    }
    public int getAge() {
        return age;
    }
    public String getPhone() {
        return phone;
    }
}

```

[/code:1:af65f1d5b3]  
 之后写XML的测试类，  
 TESTXml.java  
 [code:1:af65f1d5b3]

```

import java.io.*; //Java基础包，包含各种IO
                  操作
import java.util.*; //Java基础包，包含各种标
                  准数据结构操作
import javax.xml.parsers.*; //XML解析器接
                  □
import org.w3c.dom.*; //XML的DOM实现
import org.apache.crimson.tree.XmlDocu
                  ment; //写XML文件要用到

    public class XMLTest {
        Vector student_Vector;
        XMLTest() {

        }
    }

```

//为了保存多个学生信息，还得借助一个集合类(并不是单纯意义上的集合，JAVA中的集合是集合框架的概念，包含向量、列表、哈希表等)，这里采用Vector向量类。定义在XMLTest测试类中，命名为student\_Vector。然后定义两个方

法readXMLFile和writeXMLFile，实现读写操作。代码如下：

```
private void readXMLFile(String inFile) throws Exception {
//为解析XML作准备，创建DocumentBuilderFactory实例,指定DocumentBuilderFactory
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
DocumentBuilder db = null;
    try {
        db = dbf.newDocumentBuilder();
    }
    catch (ParserConfigurationException pce) {
        System.err.println(pce); //出异常时输出异常信息，然后退出，下同
        System.exit(1);
    }

    Document doc = null;
    try {
        doc = db.parse(inFile);
    }
    catch (DOMException dom) {
        System.err.println(dom.getMessage());
        System.exit(1);
    }
    catch (IOException ioe) {
        System.err.println(ioe);
        System.exit(1);
    }
//下面是解析XML的全过程，比较简单，先取根元素"学生花名册"
    Element root = doc.getDocumentElement();
```

```

        //取"学生"元素列表
        NodeList students = root.getElementsByTagName("学生");
        for (int i = 0; i < students.getLength(); i++) {
            //依次取每个"学生"元素
            Element student = (Element) students.item(i);
            //创建一个学生的Bean实例
            StudentBean studentBean = new StudentBean();
            //取学生的性别属性
            studentBean.setSex(student.getAttribute("性别"));
            //取"姓名"元素，下面类同
            NodeList names = student.getElementsByTagName("姓名");
            if (names.getLength() == 1) {
                Element e = (Element) names.item(0);
                Text t = (Text) e.getFirstChild();
                studentBean.setName(t.getNodeValue());
            }

            NodeList ages = student.getElementsByTagName("年龄");
            if (ages.getLength() == 1) {
                Element e = (Element) ages.item(0);
                Text t = (Text) e.getFirstChild();
                studentBean.setAge(Integer.parseInt(t.getNodeValue()));
            }
        }
    }
}

```

```
        NodeList phones = student.getDocumentElementsByTagName("电话");
        if (phones.getLength() == 1) {
            Element e = (Element) phones.item(0);
            Text t = (Text) e.getFirstChild();
            studentBean.setPhone(t.getNodeValue());
        }
    }
}
```

```
student_Vector.add(studentBean);
    }
}
```

```
private void writeXMLFile(String outfile) throws Exception {
//为解析XML作准备，创建DocumentBuilderFactory实例,指定DocumentBuilderFactory
    DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
    DocumentBuilder db = null;
    try {
        db = dbf.newDocumentBuilder();
    }
    catch (ParserConfigurationException pce) {
        System.err.println(pce);
        System.exit(1);
    }
}
```

```
    Document doc = null;
    doc = db.newDocument();
```

//下面是建立XML文档内容的过程，先建立根元

```
        素"学生花名册"
Element root = doc.createElement
    ("学生花名册");
    //根元素添加上文档
    doc.appendChild(root);

    //取学生信息的Bean列表
    for (int i = 0; i < student_Vector.si
        ze(); i++) {
        //依次取每个学生的信息
        StudentBean studentBean = (
StudentBean) student_Vector.get(i);
        //建立"学生"元素，添加到根元素
        Element student = doc.create
            Element("学生");
            student.setAttribute("性别", stu
                dentBean.getSex());
                root.appendChild(student);
                //建立"姓名"元素，添加到学生下面，下同
                Element name = doc.createEle
                    ment("姓名");
                    student.appendChild(name);
                    Text tName = doc.createTextNo
                        ode(studentBean.getName());
                        name.appendChild(tName);

                    Element age = doc.createElem
                        ent("年龄");
                        student.appendChild(age);
                        Text tAge = doc.createTextNo
                            de(String.valueOf(studentBean.
                                getAge()));
                                age.appendChild(tAge);

                    Element phone = doc.createEl
                        ement("电话");
                        student.appendChild(phone);
```

```

        Text tPhone = doc.createTextNode
Node(studentBean.getPhone());
        phone.appendChild(tPhone);
    }
    //把XML文档输出到指定的文件
    FileOutputStream outputStream = ne
w FileOutputStream(outFile);
    OutputStreamWriter outWriter =
new OutputStreamWriter(outputStream);
    ((XmlDocument) doc).write(outWr
iter, "GB2312");
    outWriter.close();
    outputStream.close();
}

```

//最后加入测试主函数，如下：

```

public static void main(String[] args) t
hrows Exception {
    //建立测试实例
    XMLTest xmlTest = new XMLTest(
    );
    //初始化向量列表
    xmlTest.student_Vector = new Vec
tor();

    System.out.println("开始读Input.x
ml文件");
    xmlTest.readXMLFile("Input.xml")
    ;

    System.out.println("读入完毕,开始写
Output.xml文件");
    xmlTest.writeXMLFile("Output.xml
l");
    System.out.println("写入完成");
    System.in.read();
}

```

```
}/code:1:af65f1d5b3]
```

了，保存好StudentBean和XMLTest，把Input.xml保存到工作目录下。如果您输入很仔细，没敲错字母的话，可以看到"写入完成"了，去瞧瞧Output.xml文件和Input.xml文件是不是一样吧。如果您在调试过程中发现有什么问题，欢迎通过E-Mail：nbDeveloper@hotmail.com与笔者取得联系。 [/code]

[【发表回复】](#) [【查看CU论坛原帖】](#) [【关闭】](#)

---

[zlzi2010](#) 回复于：2003-07-31 10:25:58

转自中文java技术网，

在测试运行是对代码做了一些小得改动.程序在我得机子上运行通过，

---

[coco520](#) 回复于：2003-07-31 10:37:21

ok,thx

---

[foreverlee0619](#) 回复于：2003-07-31 23:01:39

thanx

---

[hdcola](#) 回复于：2003-08-01 08:40:40

这只是用dom的，再写一个用sax的罢。  
还可以给大家推荐一下xml实体化的一些类库  
呀 😊

---

[jinijxta](#) 回复于：2003-08-03 14:11:03

JDOM分析还要好用一些,DOM分析XML文档还是比较麻烦.XML什么都好,就是语义问题不是很好处理.很多XML应用都是强制规定了语义的,如SVG,SMIL等

---

[zlzi2010](#) 回复于 : 2003-08-05 11:52:10

当前SAX API有两个版本。我们用第二版（见资源）来做示例。第二版中的类名和方法名与第一版都有出入，但是代码的结构是一样的。

SAX是一套API，不是一个解析器，所以这个代码在XML解析器中是通用的。要让示例跑起来，你将需要一个支持SAX v2的XML解析器。我用Apache的Xerces解析器。（见资源）参照你的解析器的getting-started文档来获得调用一个SAX解析器的资料。

SAX API 的说明书通俗易懂。它包含了很多的详细内容。而使用SAX API的主要任务就是创建一个实现ContentHandler接口，一个供XML解析器调用以将分析XML文档时所发生的SAX事件分发给处理程序的回调接口。

方便起见，SAX API也提供了一个已经实现了ContentHandler接口的DefaultHandler适配器类。

一旦实现了ContentHandler或者扩展了DefaultHandler类，你只需直接将XML解析器解析一个特定的文档即可。

我们的第一个例子扩展DefaultHandler将每个SAX事件打印到控制台。这将给你一个初步的印象，以说明什么SAX事件将会发生及以怎样的顺序发生。

作为开始，以下是将在我们的第一个示例中用到的XML文档样本：

```
<?xml version="1.0"?>  
<simple date="7/7/2000" >
```

```
<name> Bob </name>
<location> New York </location>
</simple>
```

```
[code:1:8cb40ad65c]package xmltest;
import javax.xml.parsers.*;
import org.xml.sax.*;
import org.xml.sax.helpers.*;
import java.io.*;
```

```
public class SaxTest extends DefaultHand
ler {
```

```
    // 重载DefaultHandler类的方法
    // 以拦截SAX事件通知。
```

```
    //
```

```
    // 关于所有有效事件，见org.xml.sax.Co
    ntentHandler
```

```
    //
```

```
public void startDocument( ) throws SA
XException {
    System.out.println( "SAX Event: STAR
T DOCUMENT" );
}
```

```
public void endDocument( ) throws SAX
Exception {
    System.out.println( "SAX Event: END
DOCUMENT" );
}
```

```
public void startElement( String names
```

```

        namespaceURI,
        String localName,
        String qName,
        Attributes attr ) throws SAXException {
    System.out.println( "SAX Event: START ELEMENT[ " +
        localName + " ]" );

    // 如果有属性，我们也一并打印出来...
    for ( int i = 0; i < attr.getLength(); i++ ){
        System.out.println( " ATTRIBUTE: " +
            attr.getLocalName(i) +
            " VALUE: " +
            attr.getValue(i) );
        }

    }

public void endElement( String namespaceURI,
    String localName,
    String qName ) throws SAXException {
    System.out.println( "SAX Event: END ELEMENT[ " +
        localName + " ]" );
    }

public void characters( char[] ch, int start, int length )
    throws SAXException {

    System.out.print( "SAX Event: CHARACTERS[ " );

```

```

        try {
            OutputStreamWriter outw = new OutputStreamWriter(System.out);
            outw.write( ch, start,length );
            outw.flush();
        } catch (Exception e) {
            e.printStackTrace();
        }

        System.out.println( " )" );

    }

public static void main( String[] argv ){

    System.out.println( "Example1 SAX Events:" );
        try {

            // SAXParserFactory spFactory = SAXParserFactory.newInstance();
            // SAXParser sParser = spFactory.newSAXParser();

            // 建立SAX 2解析器 . . .
            XMLReader xr = XMLReaderFactory.createXMLReader("org.apache.xerces.parsers.SAXParser");

            // 安装ContentHandler . . .
            xr.setContentHandler( new SaxTest(
                ) );

            // 解析文件 . . .

```

```
xr.parse( new InputSource(  
new FileReader( "exampleA.xml"  
)) );
```

```
}catch ( Exception e ) {  
e.printStackTrace();  
}
```

```
}
```

```
}
```

```
[/code:1:8cb40ad65c]
```

需下载xml解析器<http://xml.apache.org/dist/xerces-j/Xerces-J-bin.2.5.0.zip>

jdom解析xml:

<http://chinaunix.net/forum/viewtopic.php?p=865198#865198>  
建议下载jdom8得包

## Log4J 学习笔记 ( 1 )

本文由 Hilton 所撰写 版权归属于 Hilton  
如需转载请来信告知 hitonyang@yahoo.com.cn



说实话，除了log4j的功能外，我更喜欢它的logo.

下面的这篇笔记，主要是"borrow from"Log4J的随机文档"Short introduction to log4j"，

由Ceki Gülcü 写于March 2002，其它参考文档见文后。

- 1、log4j已经被移植到C, C++, C#, Perl, Python, Ruby, Eiffel 几种语言。
- 2、log4j有三种主要的组件：记录器, 存放器, 布局
- 3、记录器（记录器可不关心log数据存放的事哟）

log4j允许程序员定义多个记录器，每个记录器有自己的名字，记录器之间通过名字来表明隶属关系（或家族关系）。列如，记录器a.b, 与记录器a.b.c之间是父子关系，而记录器a与a.b.c之间是祖先与后代的关系，父子关系是祖先与后代关系的特例。通过这种关系，可以描述不同记录器之间的逻辑关系。

有一个记录器叫根记录器，它永远存在，且不能通过名字检索或引用，可以通过Logger.getRootLogger()方法取得它，而一般记录器通过Logger.getLogger(String name)方法。下面是Logger类的基本方法。

```
package org.apache.log4j;
```

```
public class Logger {
```

```
// Creation & retrieval methods:
```

```
public static Logger getRootLogger();
```

```
public static Logger getLogger(String name);
```

```
// printing methods:
public void debug(Object message);
public void info(Object message);
public void warn(Object message);
public void error(Object message);
public void fatal(Object message);

// generic printing method:
public void log(Level l, Object message);
}
```

记录器还有一个重要的属性，就是级别。（这好理解，就象一个家庭中，成员间存在辈份关系，但不同的成员的身高可能不一样，且身高与辈份无关）程序员可以给不同的记录器赋以不同的级别，如果某个成员没有被明确值，就自动继承最近的一个有级别长辈的级别值。根记录器总有级别值。例如：

记录器名	赋予的级别值	继承的级别值
root	Proot	Proot
X	Px	Px
X.Y	none	Px
X.Y.Z	none	Px

程序员可以自由定义级别。级别值之间存在偏序关系，如上面几种级别就有关系DEBUG

每一条要输出的log信息，也有一个级别值。

前面的Logger类中，就预定义了 DEBUG, INFO, WARN, ERROR , FATAL几种级别，由于与方法绑定，让人易产生误解，其实这几个方法只不过表明了要记录的log信息的级别。当调用log()方法时，log信息的级别就需要在通过参数明确指定。

如果一条log信息的级别，大于等于记录器的级别值，那么记录器就会记录它。如果你觉得难以理解，可参考下例。

```
// get a logger instance named "com.foo"
Logger logger = Logger.getLogger("com.foo");
```

```
// Now set its level. Normally you do not need to set the
```

```
// level of a logger programmatically. This is usually done
// in configuration files.
logger.setLevel(Level.INFO);

Logger barlogger = Logger.getLogger("com.foo.Bar");

// This request is enabled, because WARN >= INFO.
logger.warn("Low fuel level.");

// This request is disabled, because DEBUG < INFO.
logger.debug("Starting search for nearest gas station.");

// The logger instance barlogger, named "com.foo.Bar",
// will inherit its level from the logger named
// "com.foo" Thus, the following request is enabled
// because INFO >= INFO.
barlogger.info("Located nearest gas station.");

// This request is disabled, because DEBUG < INFO.
barlogger.debug("Exiting gas station search");
```

有几个有趣的情况，一是当一个记录器实例化后，再一次用相同的名字调用getLogger()会返回对它的引用，这非常有利于用同一个记录器在不同代码或类中记录log信息，另一个是与自然界中祖先先于后代出现不同，一个记录器的祖先可以比后代记录出现的晚，但会自动根据名字之间的关系建立这种家族关系。

#### 4、存放器

在log4j中，log信息通过存放器输出到目的地。支持的存放器有console, files, GUI components, remote socket servers, JMS, NT Event Loggers, remote UNIX Syslog daemons。通过file存放器，log信息可以被输出到不同的文件中（即不同的目的地）。log信息可被异步存放。

一个记录器可以有多个存放器，可以通过方法addAppender来增加存放器。一条log信息如果可被这个记录器处理，则记录器会把这条信息送往每个它所拥有的存放器。

每个记录器有一个继承开关，其开关决定记录器是/否继承其父记录

器的存放器，注意，如果继承则只继承其父记录器，而不考虑更远的祖先的情况。参考下表：

记录器	增加的存放器	继承的存放器	输出的目的地	备注
root	A1	not applicable	A1	The root logger is anonymous but can be accessed with the <code>Logger.getRootLogger()</code> method. There is no default appender attached to root.
x	A-x1, A-x2	TRUE	A1, A-x1, A-x2	Appenders of "x" and root.
x.y	none	TRUE	A1, A-x1, A-x2	Appenders of "x" and root.
x.y.z	A-xyz1	TRUE	A1, A-x1, A-x2, A-xyz1	Appenders in "x.y.z", "x" and root.
security	A-sec	FALSE	A-sec	No appender accumulation since the additivity flag is set to false.
security.access	none	TRUE	A-sec	Only appenders of "security" because the additivity flag in "security" is set to

||false.||

## 5、布局

布局负责格式化输出的log信息。log4j的PatternLayout可以让程序以类似C语言printf的格式化模板来定义格式。

6、log4j可据程序员制定的标准自动提供一些log信息，这对那类需要频繁log的对象的情况很帮助。对象的自动log，具有继承性。

参考文献：

1、log4j--新的日志操作方法，scriptskychen，[http://www.cn-java.com/target/news.php?news\\_id=2590](http://www.cn-java.com/target/news.php?news_id=2590)

## Log4J 学习笔记 ( 2 )

本文由 Hilton 所撰写 版权归属于 Hilton  
如需转载请来信告知 hitonyang@yahoo.com.cn



前面主要记了一些原理，这次是实务。

- 1、研究发现，一个系统中4%的代码是用来作logging的。
- 2、Log4J的配置文件(Configuration File)就是用来设置记录器的级别、输出器和布局的，它可接key=value格式的设置或xml格式的设置信息。通过配置，可以创建出Log4J的运行环境。

Log4J运行时，不对环境做任何假定，尤其是没有默认的存放器。

- 3、有几种方式可以配置Log4J

- 1) 在程序中调用BasicConfigurator.configure()方法；
- 2) 配置放在文件里，通过命令行参数传递文件名字，通过PropertyConfigurator.configure(args[x])解析并配置；
- 3) 配置放在文件里，通过环境变量传递文件名等信息，利用log4j默认初始化过程解析并配置；
- 4) 配置放在文件里，通过应用服务器配置传递文件名等信息，利用一个特殊的servlet来完成配置。

看下面的例子：

```
import com.foo.Bar;

// Import log4j classes.

import org.apache.log4j.Logger;

import org.apache.log4j.BasicConfigurator;

public class MyApp {

    // Define a static logger variable so that it references the
    // Logger instance named "MyApp".

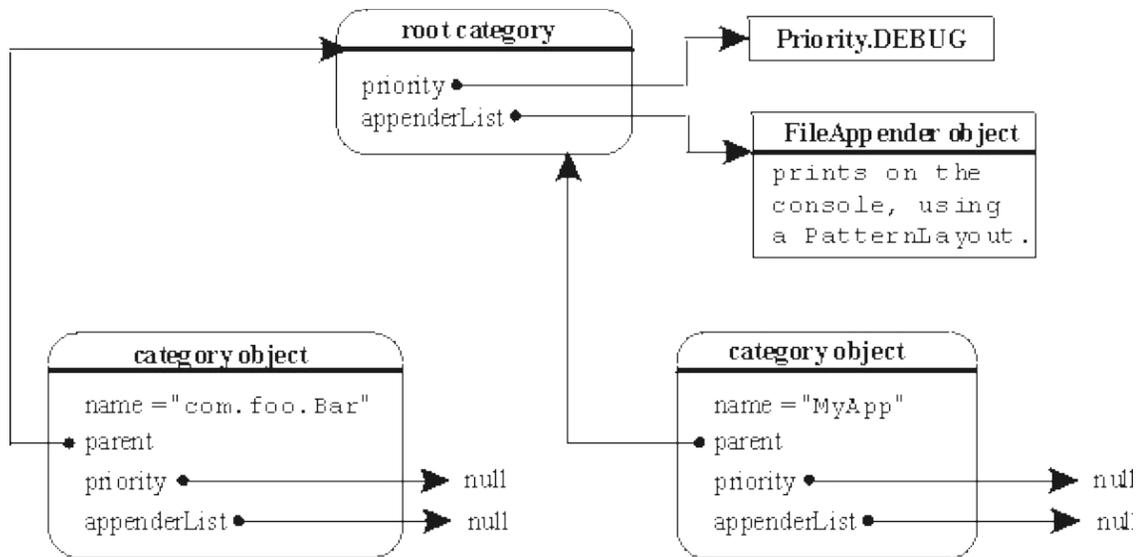
    static Logger logger = Logger.getLogger(MyApp.class);

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

```
// Set up a simple configuration that logs on the console.  
BasicConfigurator.configure();  
  
logger.info("Entering application.");  
  
Bar bar = new Bar();  
  
bar.doIt();  
  
logger.info("Exiting application.");  
  
}  
  
}
```

```
package com.foo;  
  
import org.apache.log4j.Logger;  
  
public class Bar {  
  
    static Logger logger = Logger.getLogger(Bar.class);  
  
    public void doIt() {  
  
        logger.debug("Did it again!");  
  
    }  
  
}
```

BasicConfigurator.configure给根记录器增加一个ConsoleAppender，输出格式通过PatternLayout设为"%-4r [%t] %c %x - %m%n"，还有根记录器的默认级别是Level.DEBUG。记录器之间的关系如下图：



输出结果如下：

```

0 [main] INFO MyApp - Entering application.
36 [main] DEBUG com.foo.Bar - Did it again!
51 [main] INFO MyApp - Exiting application.
  
```

下面的代码结合配置信息，会得到与上述程序一样的结果。

```

import com.foo.Bar;

import org.apache.log4j.Logger;

import org.apache.log4j.PropertyConfigurator;

public class MyApp {

    static Logger logger = Logger.getLogger(MyApp.class.getName())

    public static void main(String[] args) {

        // BasicConfigurator replaced with PropertyConfigurator.
        PropertyConfigurator.configure(args[0]);

        logger.info("Entering application.");

        Bar bar = new Bar();

        bar.doIt();
    }
}
  
```

```
        logger.info("Exiting application.");
    }
}
```

配置文件的内容如下：

```
# Set root logger level to DEBUG and its only appender to A1.
log4j.rootLogger=DEBUG, A1
```

```
# A1 is set to be a ConsoleAppender.
log4j.appender.A1=org.apache.log4j.ConsoleAppender
```

```
# A1 uses PatternLayout.
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
log4j.appender.A1.layout.ConversionPattern=%-4r [%t] %-5p %
%x - %m%n
```

利用配置文件，可以很方便地修改配置。如下例

```
log4j.rootLogger=debug, stdout, R
```

```
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
```

```
# Pattern to output the caller's file name and line number.
#log4j.appender.stdout.layout.ConversionPattern=%5p [%t]
(%F:%L) - %m%n
```

```
# Print the date in ISO 8601 format
log4j.appender.stdout.layout.ConversionPattern=%d [%t] %-5p
%c - %m%n
```

```
log4j.appender.R=org.apache.log4j.RollingFileAppender
log4j.appender.R.File=example.log
```

```
log4j.appender.R.MaxFileSize=100KB
# Keep one backup file
log4j.appender.R.MaxBackupIndex=1
```

```
log4j.appender.R.layout=org.apache.log4j.PatternLayout
```

```
log4j.appender.R.layout.ConversionPattern=%p %t %c - %m%  
  
# Print only messages of level WARN or above in the package  
com.foo.  
log4j.logger.com.foo=WARN
```

对于tomcat4，利用环境变量传递参数的，可参见以下几个例子

unix设置

```
export CATALINA_OPTS="-Dlog4j.configuration=foobar.txt" <==  
用PropertyConfigurator解析
```

```
export CATALINA_OPTS="-Dlog4j.debug -  
Dlog4j.configuration=foobar.xml" <===用DOMConfigurator解析
```

以下是windows设置

```
set CATALINA_OPTS =-Dlog4j.configuration=foobar.lcf -  
Dlog4j.configuratorClass=com.foo.BarConfigurator <===用  
com.foo.BarConfigurator解析
```

```
set CATALINA_OPTS =-Dlog4j.configuration=file:/c:/foobar.lcf  
配置文件位置如果没有明确指明，则要放在WEB-INF/classes目录下。
```

#### 4、用servlet配置log4j

以下都是参考冰之火的文章，抄来放在这儿，并做了一些必要的修改。说明的是，下面的代码需要自己写并发布，下的jar中没有这个类。待我后写一个，也放上来。我也写了两个，在笔记（三）中。

在Application目录下的web.xml文件加入以后代码

```
<servlet>  
  
  <servlet-name>log4jlog4j-init</servlet-name>  
  
  <servlet-class>com.apache.jakarta.log4j.Log4jInit</servlet-cl  
  
  <init-param>  
  
    <param-name>log4j</param-name>  
  
    <param-value>/WEB-INF/log4j.properties</param-value>  
  
  </init-param>  
  
  <load-on-startup>1</load-on-startup>
```

```
</servlet>
```

这段代码的意思是说，在Tomcat启动时加载com.apache.jakarta.log4j.Log4jInit这个名叫Log4jInit.class这个类文件。

其中Log4jInit.class的源代码如下

```
package com.apache.jakarta.log4j;

import org.apache.log4j.PropertyConfigurator;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class Log4jInit extends HttpServlet {

    public void init() {

        String prefix = getServletContext().getRealPath("/");
        String file = getInitParameter("log4j");
        // if the log4j-init-file is not set, then no point in tr
        System.out.println(".....log4j start");
        if(file != null) {
            PropertyConfigurator.configure(prefix+file);
        }
    }

    public void doGet(HttpServletRequest req, HttpServletResponse res
```

```
}  
  
}
```

在加载的过程中，程序会读取/WEB-INF/log4j.properties这个文件。  
配置文件讲解如下：

```
# Set root logger level to DEBUG and its only appender to A1  
#log4j中有五级logger  
#FATAL 0  
#ERROR 3  
#WARN 4  
#INFO 6  
#DEBUG 7  
#配置根Logger，其语法为：  
#log4j.rootLogger = [ level ] , appenderName, appenderName  
log4j.rootLogger=INFO, A1 ,R  
#这一句设置以为着所有的log都输出  
#如果为log4j.rootLogger=WARN, 则意味着只有WARN,ERROR,FAT  
#被输出，DEBUG,INFO将被屏蔽掉。  
# A1 is set to be a ConsoleAppender.  
#log4j中Appender有几层如控制台、文件、GUI组件、甚至是套接口、  
#器、NT的事件记录器、UNIX Syslog守护进程等  
#ConsoleAppender输出到控制台  
log4j.appender.A1=org.apache.log4j.ConsoleAppender  
# A1 使用的输出布局，其中log4j提供4种布局。  
org.apache.log4j.HTMLLayout (以HTML表格形式布局)  
#org.apache.log4j.PatternLayout (可以灵活地指定布局模式)，  
#org.apache.log4j.SimpleLayout (包含日志信息的级别和信息字符串)，  
#org.apache.log4j.TTCCLayout (包含日志产生的时间、线程、类别  
等信息)  
  
log4j.appender.A1.layout=org.apache.log4j.PatternLayout  
#灵活定义输出格式 具体查看log4j javadoc  
org.apache.log4j.PatternLayout  
#d 时间 ....  
log4j.appender.A1.layout.ConversionPattern=%-d{yyyy-MM-dc
```

```
HH:mm:ss} [%c]-[%p] %m%n
#R 输出到文件 RollingFileAppender的扩展，可以提供一种日志的备
能。
log4j.appender.R=org.apache.log4j.RollingFileAppender
#日志文件的名称
log4j.appender.R.File=log4j.log
#日志文件的大小
log4j.appender.R.MaxFileSize=100KB
# 保存一个备份文件
log4j.appender.R.MaxBackupIndex=1

log4j.appender.R.layout=org.apache.log4j.TTCCLayout
#log4j.appender.R.layout.ConversionPattern=%-d{yyyy-MM-d
HH:mm:ss} [%c]-[%p] %m%n
```

配置根Logger，其语法为：

```
log4j.rootLogger = [ level ], appenderName, appenderName,
```

level 是日志记录的优先级

appenderName就是指定日志信息输出到哪个地方。您可以同时指定多个输出目的地。

配置日志信息输出目的地Appender，其语法为

```
log4j.appender.appenderName =
fully.qualified.name.of.appender.class
log4j.appender.appenderName.option1 = value1
...
log4j.appender.appenderName.option = valueN
```

Log4j提供的appender有以下几种：

org.apache.log4j.ConsoleAppender（控制台），

org.apache.log4j.FileAppender（文件），

org.apache.log4j.DailyRollingFileAppender（每天产生一个日志文件），

org.apache.log4j.RollingFileAppender（文件大小到达指定尺寸的时候产生一个新的文件），

org.apache.log4j.WriterAppender ( 将日志信息以流格式发送到任意指定的地方 )

配置日志信息的格式 ( 布局 ) , 其语法为 :

```
log4j.appender.appenderName.layout =  
fully.qualified.name.of.layout.class  
log4j.appender.appenderName.layout.option1 = value1  
....  
log4j.appender.appenderName.layout.option = valueN
```

Log4j提供的layout有以下几种 :

org.apache.log4j.HTMLLayout ( 以HTML表格形式布局 ) ,  
org.apache.log4j.PatternLayout ( 可以灵活地指定布局模式 ) ,  
org.apache.log4j.SimpleLayout ( 包含日志信息的级别和信息字符串 ) ,  
org.apache.log4j.TTCCLayout ( 包含日志产生的时间、线程、类别等信息 )

## Log4J 学习笔记 (3)

本文由 Hilton 所撰写 版权归属于 Hilton  
如需转载请来信告知 hitonyang@yahoo.com.cn

---

我这儿有两个程序，一个是普通的java程序，实现了一个“九九表”  
tomcat是4.1.12，J2SE是1.3.1，log4j的版本是1.2.8.

### 一、九九表。

环境设置：需要将log4j-1.2.8.jar放入CLASSPATH变量中。

Hello.java文件的内容如下：

```
import org.apache.log4j.*;
public class Hello{
    static Logger logger = Logger.getLogger(Hello.class);
    public static void main(String[] args) {
        int i,j;
//        BasicConfigurator.configure();
        PropertyConfigurator.configure(args[0]);

        logger.info("Entering application.");
        for(i=1;i<10;i++){
            logger.debug(""+i);
            for (j=1;j<=i;j++){
                logger.warn(""+j);
                System.out.print(i*j);
                System.out.print("\t");
            }
            System.out.println("");
        }
        logger.info("Exiting application.");
    }
}
```

log4j的配置文件log4j.inf的内容如下：

```
log4j.rootLogger=WARN, stdout, R
```

```
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
```

```

# Pattern to output the caller's file name and line number.
#log4j.appender.stdout.layout.ConversionPattern=%5p [%t] (%F:%L)
# Print the date in ISO 8601 format
log4j.appender.stdout.layout.ConversionPattern=%d [%t] %-5p %c -

log4j.appender.R=org.apache.log4j.RollingFileAppender
log4j.appender.R.File=example.log

log4j.appender.R.MaxFileSize=100KB
# Keep one backup file
log4j.appender.R.MaxBackupIndex=1

log4j.appender.R.layout=org.apache.log4j.PatternLayout
log4j.appender.R.layout.ConversionPattern=%p %t %c - %m%n

# Print only messages of level WARN or above in the package com.f
log4j.logger.com.foo=WARN

```

运行：

```

javac Hello.java
java Hello log4j.inf

```

结果会在屏幕上显示，并在example.log文件中记录。

## 二、servlet

环境设置：将log4j-1.2.8.jar及servlet.jar放入环境变量CLASSPATH中  
\$TOMCAT\_HOME/common/lib目录下。

假定，有一个部署到tomcat的应用叫myweb。

servlet程序Log4jInit.java的位置在\$TOMCAT\_HOME/webapps/my  
INF/classes/com/hedong/learning/log4j/目录下，内容如下：

```

package com.hedong.learning.log4j;
import org.apache.log4j.*;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
public class Log4jInit extends HttpServlet {
    public void init() {
        String prefix = getServletContext().getRealPath("/");
        String file = getInitParameter("log4j");
        // if the log4j-init-file is not set, then no point in tr
        System.out.println(".....log4j start");
        if(file != null) {

```

```

        PropertyConfigurator.configure(prefix+file);
    }
}
public void doGet(HttpServletRequest req, HttpServletResponse res
}
}
}

```

在Log4jInit.java所在目录下编译它:

```
javac Log4jInit.java
```

myweb的设置文件web.xml在\$TOMCAT\_HOME/webapps/myweb/分。

```

</web-app>
.....
<servlet>
  <servlet-name>log4j-init</servlet-name>
  <servlet-class>com.hedong.learning.log4j.Log4jInit</servlet-c
  <init-param>
    <param-name>log4j</param-name>
    <param-value>WEB-INF/log4j.properties</param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>
</web-app>

```

同时，在这个目录下建一个文件名叫log4j.properties，内容如下：

```

log4j.rootLogger=INFO, A1 , R
log4j.appender.A1=org.apache.log4j.ConsoleAppender
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
log4j.appender.A1.layout.ConversionPattern=%-d{yyyy-MM-dd HH:mm:s
log4j.appender.R=org.apache.log4j.RollingFileAppender
log4j.appender.R.File=$TOMCAT_HOME/webapps/dbweb/logs/log4j.log<-
log4j.appender.R.MaxFileSize=100KB
log4j.appender.R.MaxBackupIndex=1
log4j.appender.R.layout=org.apache.log4j.PatternLayout
log4j.appender.R.layout.ConversionPattern=%p %t %c - %m%n

```

记住，别忘了创建目录\$TOMCAT\_HOME/webapps/dbweb/logs/，如

创建目录\$TOMCAT\_HOME/webapps/dbweb/test/，然后在这个目录

```
<%@ page contentType="text/html; charset=GB2312" %>
<%@ page import="org.apache.log4j.*" %>
<%
    Logger logger = Logger.getLogger("test.jsp");
    logger.info("befor say hi");
%>
<h1> Hi</h1>
<%
    logger.info("after say hi");
%>
```

然后，重新启动tomcat，通过浏览器访问这个jsp页面，如  
http://yourdomain.com:8080/myweb/test/test.jsp，如果一切正常  
务器上的\$TOMCAT\_HOME/webapps/dbweb/logs/log4j.log文件中  
INFO HttpProcessor[8080][4] test.jsp - befor say hi  
INFO HttpProcessor[8080][4] test.jsp - after say hi  
。在默认的情况下，tomcat的屏幕输出被重定向到\$TOMCAT\_HOME/  
最后也应看到上述的输出。

# 实例介绍Struts + Spring + Hibernate 开发

## 一 介绍

本文并不想介绍Struts, Spring, Hibernate的原理系统架构等, 本文地目的是通过一个较复杂地实例介绍如何整合Struts, Spring, Hibernate, 网上现有的例子虽然也能达到目的, 但功能都比较单一, 复杂的例子时会有意想不到的麻烦。本文对读者假设已经具备了以上框架的基础知识。以及那些已经了解Struts, Spring, Hibernate的基本概念, 但是还没有亲身在较复杂的项目中体验Struts + Spring + Hibernate的开发人员。

### 1 Struts

虽然不打算过多介绍Struts的原理, 但是大概介绍一下还是有必要的。Struts本身就是 MVC 在这里负责将用户数据传入业务层, 以及 将业务层处理的结果返回给用户, 此系统属于较简单WEB应用, 采用了OpenSessionInView模式处理LazyLoad问题, 这样我们可以在用户视图中使用 get, set方法来方便地获取关联对象。为了处理庞大的Action和ActionForm问题, 在此我们准备使用DynaActionForm (DynaValidatorForm)和DispatchAction以及 动态验证框架 来解决。及使用Tile来解决框架问题。使用自定义标签处理分页和身份验证问题。

### 2 Spring

Spring Framework最得以出名的是与Hibernate的无缝链接, 虽然Spring 对Hibernate提供了90%以上的封装, 使我们不必去关心Session 的建立, 关闭, 以及事务使我们能够专心的关注业务逻辑。但是一些特殊情况如 有时需要Query以及Criteria 对象, 分页等, Spring不能给我们提供支持, 总不能每次都在你的DAO上写个HibernateCallBackup()吧? Spring的作用不是把Hibernate再封装一层, 而是让你接触不到Hibernate的API, 而是帮助你管理好Session和Transaction。

在这里解决方法是：首先写一个IBase的接口，和一个BaseDao的实现。在实现中仿照HibernateTemplate，将其功能一一实现，同时考虑到Spring未能支持的地方，我们不得已只好自己来管理Session，因此加入public Session openSession()，public Query getQuery(String sql)，public Criteria getCriteria(Class clazz)，以及分页的方法。然后为每一个Entity都建立继承于以上类的IEntity，与EntityDao。这里可以根据需求对Entity加入特殊的方法实现，如在StudentsDao.java中加入类似用户身份验证等。以上就是数据访问层。接下来在Service层中通过对dao的引用完成业务逻辑方法。在下面的例子中我们分别为学生模块，教师模块，管理员模块构建Service层，StudentsServiceImpl，TeachersServiceImpl，AdminServiceImpl

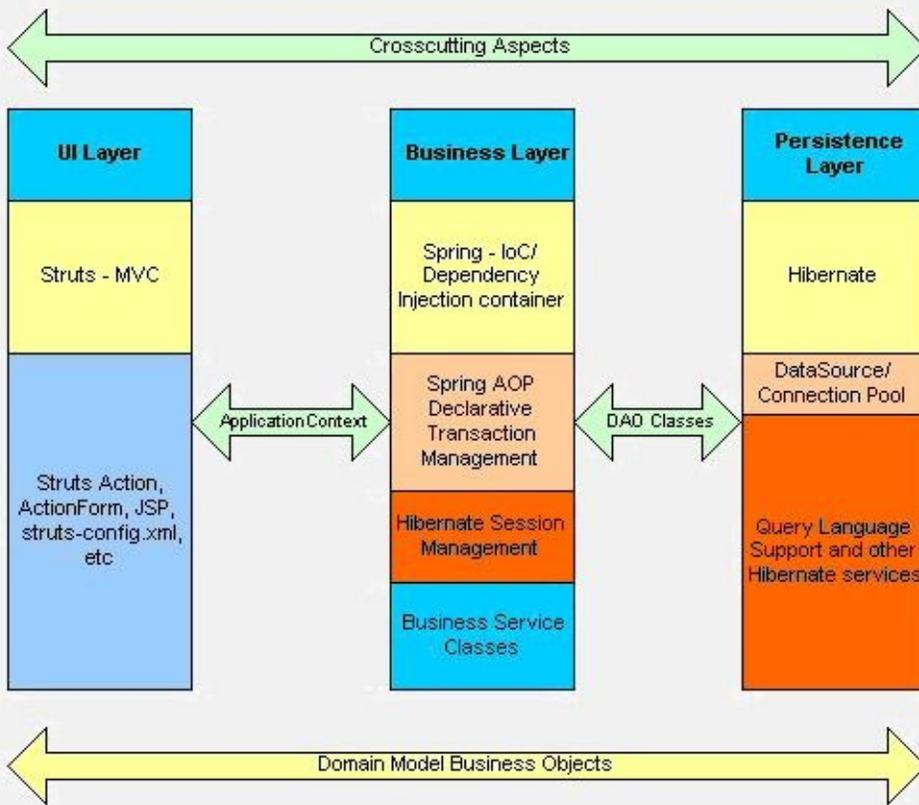
### **3 Hibernate**

有了Spring的封装，我们要对Hibernate做的就是正确实现对象关系的映射。由于此处处于系统的最底层，准确无误的实现对象之间的关联关系映射将起着至关重要的作用。

总之，理解了Struts，Spring，Hibernate地原理以及之间的关系之后，剩下的工作就如同在以Spring为核心的Struts为表现的框架中堆积木。

下图可以更好的帮助我们理解Struts，Spring，Hibernate之间的关系。

## Application Server Container



## 二 案例简述：

设计思路主要源于 大学选修课，该系统可以方便处理学生在课程选报，学分查询，成绩查询，以及成绩发布等。

系统以班级为核心，一门课程可以对应多个班级，一名教师也可以带不同的班级，学生可以选报不同课程所对应的班级，班级自身有目前人数，和最大人数，以及上课时间，上课地点的属性。

学生在选报班级之后，班级的人数会自动加一，直到等于最大人数时，其他学生将会有人数已满的错误提示。同理如果学生选择了同一课程的不同班级，也将收到错误提示。学生有密码，系别，学分，地址，电话等属性。

教师在系统中主要负责成绩发布，教师可以对其所带的班级的学生的成绩修改，系统会以成绩是否大于等于60来判断学生是否通过考试，如果通过会将该课程的学分累加到学生学分，同样如果教师二次修改了成绩，而且小于60，系统会在学生学分数上扣掉该课程的分。

课程在系统中具体体现为班级，自身带有学分属性。  
系有编号，名称的属性，同时可以作为联系教师，课程，学生的桥梁。

### 功能模块

- 1 身份验证模块：根据用户名，密码，用户类别 转发用户到不同的模块。
- 1 学生模块：查看课程，查看班级，选报课程，查看已选课程，成绩查询。
- 1 教师模块：录入成绩
- 1 管理员模块：对学生，教师，课程，班级，系增，删，查，改。

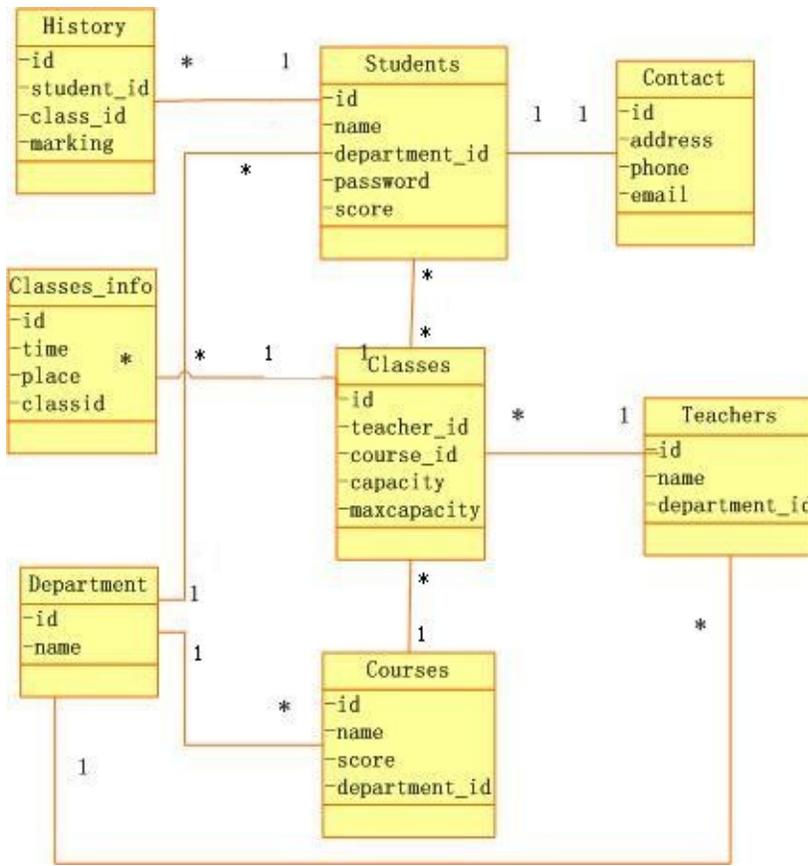
## 三 具体实践

### 代码下载

<http://www.blogjava.net/Files/limq/StudentManger.rar>

#### 1 对象关系映射：

首先，将库表映射为数据模型（SQL在源码中查看），转换后的数据模型如下图：



由此我们可以看出一下关联关系：

- 1 Students 和 Contact（联系方式）一对一关系。
- 2 Students 和 History（选课历史）一对多关系
- 3 Students 和 Classes 多对多关系。
- 4 Classes 和 Classes\_info 一对多关系。
- 5 Classes 和 Teachers 多对一关系。
- 6 Classes 和 Courses 多对一关系。
- 7 Course 和 Department（系）多对一关系。
- 8 Teachers 和 Department 多对一关系。
- 9 Students 和 Department 多对一关系。

在Hibernate中将以上关系一一映射，如Students 和 History 一对多关系

Students.cfg.xml.：

```

1 <set name="history"
2     table="history"

```

```
3         cascade="all"
4         inverse="true"
5         lazy="true" >
6         <key column="student_id"/>
7         <one-to-many class="limq.hibernate.vo.History"
8                 />
9         <SPAN>set>
10
```

同样在History.cfg.xml中加入：

```
1 <many-to-one name="student"
2         class="limq.hibernate.vo.Students"
3         column="student_id" >
4     <SPAN>many-to-one>
5
```

用过MyEclipse开发Hibernate的就知道，MyEclipse会帮助我们生成持久对象和抽象对象，我们要在 Students.java 中加入对History的引用

```
private Set history=new HashSet();
```

```
public Set getHistory() {
    return history;
}
```

```
public void setHistory(Set history) {
    this.history = history;
}
```

同时，在AbstractHistory.java 中删除student\_id 以及对应的get，set方法，History.java 中加入

```
private Students student;
public Students getStudent() {
```

```
        return student;
    }
```

```
public void setStudent(Students student) {
    this.student = student;
}
```

具体内容请查看 源代码。

## 2 DAO 数据访问层

首先，编写IBaseDao与BaseDao，其中IBaseDao代码如下：

```
1 package limq.hibernate.dao;
2
3 import java.util.Collection;
4 import java.util.List;
5 import net.sf.hibernate.Criteria;
6 import net.sf.hibernate.Query;
7 import net.sf.hibernate.Session;
8 import limq.exception.DaoException;
9
10 public interface IBaseDao {
11
12     public Session openSession();
13
14     public int getTotalCount( String hql) throws Exception;
15
16     public Query getQuery(String sql) throws Exception;
17
18     public Criteria getCriteria(Class clazz) throws Exception;
19
20     public int getTotalPage(int totalCount,int pageSize);
21
22     public void create(Object entity);
23
24     public void update(Object entity);
25
26     public void delete(Object entity) throws DaoException;
```

```

27
28 public void deleteAll(Class clazz) throws DaoException;
29
30 public void deleteAll(Collection entities) throws DaoException;
31
32 public Object loadByKey(Class clazz, String keyName, Object keyValue)
;
33
34 public List find(String queryString) throws DaoException;
35
36 public List find(String queryString, Object param) throws DaoException;
37
38 public List find(String queryString, Object[] params) throws DaoExcepti
on;
39
40 }
41

```

## BaseDao继承

继承org.springframework.orm.hibernate.support.HibernateDaoSupport  
实现以上的方法  
如：

```

1 public void create(Object entity) {
2     try {
3         getHibernateTemplate().save(entity);
4
5     } catch (Exception e) {
6         log.error("保存 " + entity.getClass().getName() + " 实例到数据库失
败", e);
7
8     }
9 }
10 /**
11  * 获得session
12  */
13 public Session openSession() {

```

```

14     return SessionFactoryUtils.getSession(getSessionFactory(), false);
15 }
16
17 /**
18  * 获得Query对象
19  */
20 public Query getQuery(String sql) throws Exception{
21     Session session = this.openSession();
22     Query query = session.createQuery(sql);
23     return query;
24 }
25 /**
26  * 获得Criteria对象
27  */
28 public Criteria getCriteria(Class clazz) throws Exception{
29
30     Session session=this.openSession();
31     Criteria criteria = session.createCriteria(clazz);
32     return criteria;
33 }
34

```

可以看到，这里即充分利用了Spring对Hibernate的支持，还弥补了Spring的不足。最后分别为每个持久对象建立Interface，以及DAO，使其分别继承IBaseDao与BaseDao。

如IDepartment，DepartmentDao

```

1 public interface IDepartment extends IBaseDao {}
2
3 public class DepartmentDao extends BaseDao implements IBaseDao {}
4

```

### 3 Service 层

在这里需要认真思考每个业务逻辑所能用到的持久层对象和DAO，还要完成配置Spring框架，首先我一起看看applications-

## service.xml

```
1 xml version="1.0" encoding="UTF-8"?>
2 DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
3 "http://www.springframework.org/dtd/spring-beans.dtd">
4 <beans>
5   <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSourc
e" destroy-method="close">
6     <property name="driverClassName">
7       <value>com.mysql.jdbc.Driver< SPAN>value>
8     < SPAN>property>
9     <property name="url">
10      <value>jdbc:mysql://localhost:3306/Student< SPAN>value>
11    < SPAN>property>
12    <property name="username">
13      <value>root< SPAN>value>
14    < SPAN>property>
15    <property name="password">
16      <value>< SPAN>value>
17    < SPAN>property>
18  < SPAN>bean>
19  <bean id="sessionFactory" class="org.springframework.orm.hibernate.Lo
calSessionFactoryBean">
20    <property name="dataSource">
21      <ref local="dataSource"/>
22    < SPAN>property>
23    <property name="mappingResources">
24      <list>
25        <value>limq/hibernate/vo/Admins.hbm.xml< SPAN>value>
26        <value>limq/hibernate/vo/Classes.hbm.xml< SPAN>value>
27        <value>limq/hibernate/vo/Courses.hbm.xml< SPAN>value>
28        <value>limq/hibernate/vo/Students.hbm.xml< SPAN>value>
29        <value>limq/hibernate/vo/ClassesInfo.hbm.xml< SPAN>value>
30        <value>limq/hibernate/vo/Contact.hbm.xml< SPAN>value>
31        <value>limq/hibernate/vo/Department.hbm.xml< SPAN>value>
```

```
32     <value>limq/hibernate/vo/History.hbm.xml< SPAN>value>
33     <value>limq/hibernate/vo/Teachers.hbm.xml< SPAN>value>
34     < SPAN>list>
35     < SPAN>property>
36     <property name="hibernateProperties">
37     <props>
38     <prop key="hibernate.dialect">net.sf.hibernate.dialect.MySQLDialect
< SPAN>prop>
39     <prop key="hibernate.show_sql">>true< SPAN>prop>
40     < SPAN>props>
41     < SPAN>property>
42     < SPAN>bean>
43     <bean id="myTransactionManager" class="org.springframework.orm.hibe
rnative.HibernateTransactionManager">
44     <property name="sessionFactory">
45     <ref local="sessionFactory"/>
46     < SPAN>property>
47     < SPAN>bean>
48
49     <bean id="hibernateInterceptor" class="org.springframework.orm.hiberna
te.HibernateInterceptor">
50     <property name="sessionFactory">
51     <ref bean="sessionFactory"/>
52     < SPAN>property>
53     < SPAN>bean>
54     <bean id="studentDaoTarget" class="limq.hibernate.dao.StudentsDao">
55     <property name="sessionFactory">
56     <ref bean="sessionFactory"/>
57     < SPAN>property>
58     < SPAN>bean>
59     <bean id="teacherDaoTarget" class="limq.hibernate.dao.TeachersDao">
60     <property name="sessionFactory">
61     <ref bean="sessionFactory"/>
62     < SPAN>property>
63     < SPAN>bean>
64     <bean id="courseDaoTarget" class="limq.hibernate.dao.CoursesDao">
65     <property name="sessionFactory">
```

```
66     <ref bean="sessionFactory"/>
67     <SPAN>property>
68 <SPAN>bean>
69 <bean id="classDaoTarget" class="limq.hibernate.dao.ClassesDao">
70     <property name="sessionFactory">
71         <ref bean="sessionFactory"/>
72     <SPAN>property>
73 <SPAN>bean>
74 <bean id="departmentDaoTarget" class="limq.hibernate.dao.Department
Dao">
75     <property name="sessionFactory">
76         <ref bean="sessionFactory"/>
77     <SPAN>property>
78 <SPAN>bean>
79 <bean id="adminDaoTarget" class="limq.hibernate.dao.AdminDao">
80     <property name="sessionFactory">
81         <ref bean="sessionFactory"/>
82     <SPAN>property>
83 <SPAN>bean>
84 <bean id="studentDao" class="org.springframework.aop.framework.Prox
yFactoryBean">
85     <property name="proxyInterfaces">
86         <value>limq.hibernate.dao.IStudents<SPAN>value>
87     <SPAN>property>
88     <property name="interceptorNames">
89         <list>
90             <value>hibernateInterceptor<SPAN>value>
91             <value>studentDaoTarget<SPAN>value>
92         <SPAN>list>
93     <SPAN>property>
94 <SPAN>bean>
95 <bean id="teacherDao" class="org.springframework.aop.framework.Prox
yFactoryBean">
96     <property name="proxyInterfaces">
97         <value>limq.hibernate.dao.ITeachers<SPAN>value>
98     <SPAN>property>
99     <property name="interceptorNames">
```

```
100     <list>
101     <value>hibernateInterceptor< SPAN>value>
102     <value>teacherDaoTarget< SPAN>value>
103     < SPAN>list>
104     < SPAN>property>
105 < SPAN>bean>
106 <bean id="courseDao" class="org.springframework.aop.framework.ProxyFactoryBean">
107     <property name="proxyInterfaces">
108     <value>limq.hibernate.dao.ICourses< SPAN>value>
109     < SPAN>property>
110     <property name="interceptorNames">
111     <list>
112     <value>hibernateInterceptor< SPAN>value>
113     <value>courseDaoTarget< SPAN>value>
114     < SPAN>list>
115     < SPAN>property>
116 < SPAN>bean>
117 <bean id="classDao" class="org.springframework.aop.framework.ProxyFactoryBean">
118     <property name="proxyInterfaces">
119     <value>limq.hibernate.dao.IClasses< SPAN>value>
120     < SPAN>property>
121     <property name="interceptorNames">
122     <list>
123     <value>hibernateInterceptor< SPAN>value>
124     <value>classDaoTarget< SPAN>value>
125     < SPAN>list>
126     < SPAN>property>
127 < SPAN>bean>
128 <bean id="departmentDao" class="org.springframework.aop.framework.ProxyFactoryBean">
129     <property name="proxyInterfaces">
130     <value>limq.hibernate.dao.IDepartment< SPAN>value>
131     < SPAN>property>
132     <property name="interceptorNames">
133     <list>
```

```
134     <value>hibernateInterceptor< SPAN>value>
135     <value>departmentDaoTarget< SPAN>value>
136     < SPAN>list>
137     < SPAN>property>
138     < SPAN>bean>
139     <bean id="adminDao" class="org.springframework.aop.framework.Proxy
FactoryBean">
140     <property name="proxyInterfaces">
141     <value>limq.hibernate.dao.IAdmin< SPAN>value>
142     < SPAN>property>
143     <property name="interceptorNames">
144     <list>
145     <value>hibernateInterceptor< SPAN>value>
146     <value>adminDaoTarget< SPAN>value>
147     < SPAN>list>
148     < SPAN>property>
149     < SPAN>bean>
150
151     <bean id="studentManagerTarget" class="limq.spring.service.StudentsSe
rviceImpl">
152     <property name="studentsDao">
153     <ref bean="studentDao"/>
154     < SPAN>property>
155     <property name="coursesDao">
156     <ref bean="courseDao"/>
157     < SPAN>property>
158     <property name="classesDao">
159     <ref bean="classDao"/>
160     < SPAN>property>
161     <property name="departmentsdao">
162     <ref bean="departmentDao"/>
163     < SPAN>property>
164     < SPAN>bean>
165     <bean id="teacherManagerTarget" class="limq.spring.service.TeachersSe
rviceImpl">
166     <property name="teachersDao">
167     <ref bean="teacherDao"/>
```

```
168 < SPAN>property>
169 <property name="coursesDao">
170 <ref bean="courseDao"/>
171 < SPAN>property>
172 <property name="classesDao">
173 <ref bean="classDao"/>
174 < SPAN>property>
175 <property name="studentsDao">
176 <ref bean="studentDao"/>
177 < SPAN>property>
178 < SPAN>bean>
179 <bean id="adminManagerTarget" class="limq.spring.service.AdminServiceImpl">
180 <property name="adminDao">
181 <ref bean="adminDao"/>
182 < SPAN>property>
183 <property name="teachersDao">
184 <ref bean="teacherDao"/>
185 < SPAN>property>
186 <property name="coursesDao">
187 <ref bean="courseDao"/>
188 < SPAN>property>
189 <property name="classesDao">
190 <ref bean="classDao"/>
191 < SPAN>property>
192 <property name="studentsDao">
193 <ref bean="studentDao"/>
194 < SPAN>property>
195 <property name="departmentsdao">
196 <ref bean="departmentDao"/>
197 < SPAN>property>
198 < SPAN>bean>
199
200 <bean id="studentManager" class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
201 <property name="transactionManager">
202 <ref bean="myTransactionManager"/>
```

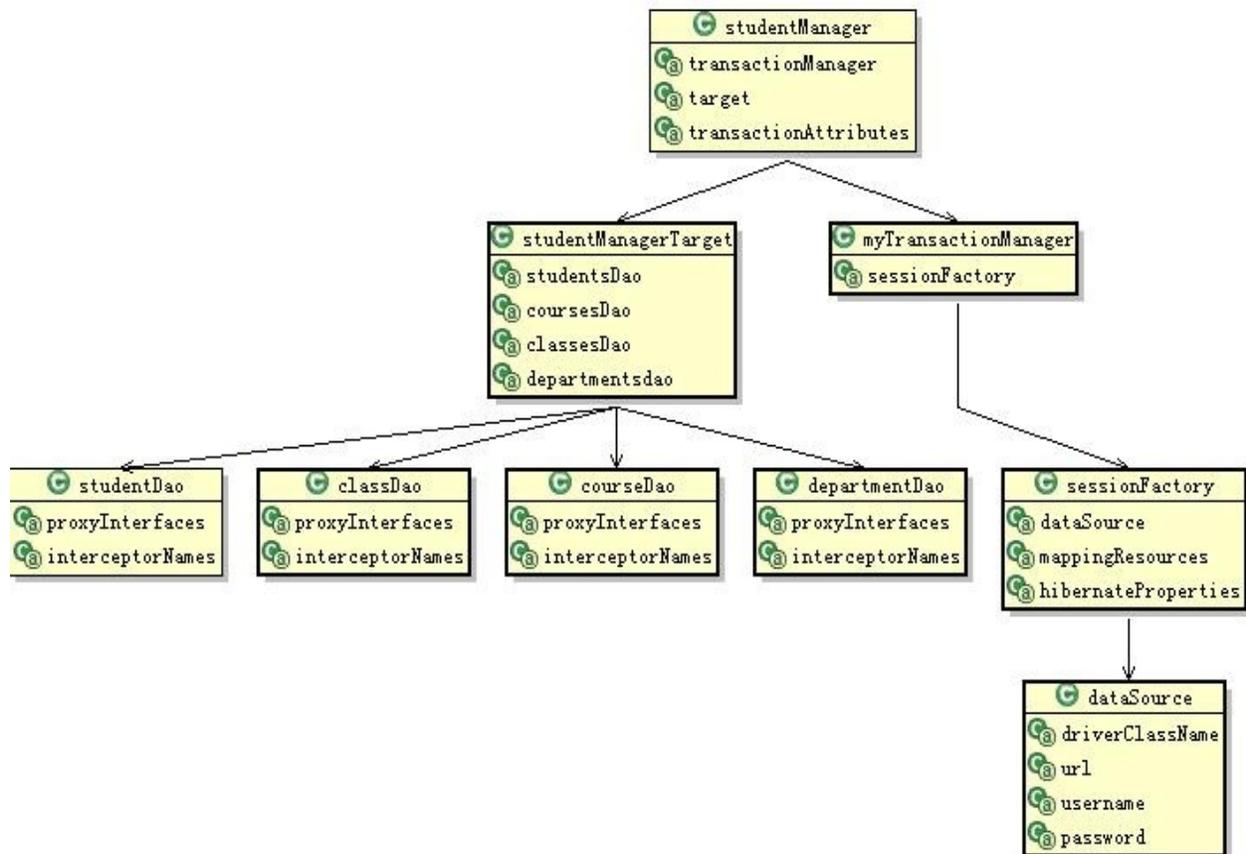
```
203 < SPAN>property>
204 <property name="target">
205   <ref bean="studentManagerTarget"/>
206 < SPAN>property>
207 <property name="transactionAttributes">
208   <props>
209     <prop key="get*">PROPAGATION_SUPPORTS< SPAN>prop>
210     <prop key="*">PROPAGATION_REQUIRED< SPAN>prop>
211   < SPAN>props>
212 < SPAN>property>
213 < SPAN>bean>
214 <bean id="teacherManager" class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
215   <property name="transactionManager">
216     <ref bean="myTransactionManager"/>
217   < SPAN>property>
218     <property name="target">
219       <ref bean="teacherManagerTarget"/>
220   < SPAN>property>
221     <property name="transactionAttributes">
222       <props>
223         <prop key="get*">PROPAGATION_SUPPORTS< SPAN>prop>
224         <prop key="*">PROPAGATION_REQUIRED< SPAN>prop>
225       < SPAN>props>
226     < SPAN>property>
227   < SPAN>bean>
228 <bean id="adminManager" class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
229   <property name="transactionManager">
230     <ref bean="myTransactionManager"/>
231   < SPAN>property>
232     <property name="target">
233       <ref bean="adminManagerTarget"/>
234   < SPAN>property>
235     <property name="transactionAttributes">
236       <props>
237         <prop key="get*">PROPAGATION_SUPPORTS< SPAN>prop>
```

```

238     <prop key="*">PROPAGATION_REQUIRED< SPAN>prop>
239     < SPAN>props>
240     < SPAN>property>
241     < SPAN>bean>
242 < SPAN>beans>
243

```

以StudentsServiceImpl以为例，下图演示了如何利用Spring的Ioc与Hibernate的结合。可以看到分别将studentDao,classDao,coursesDao,departmentDao,注入studentManager.



```

1 IStudentsService.java
2 public interface IStudentsService {
3
4     public boolean validate(String username,String pasword);
5     public Classes[] getClassesFromCourse(Courses courses);
6     public Department getDepFromID(Integer id);

```

```
7 public Courses getCourseFromID(Integer id);
8 public Classes getClassFromID(Integer id);
9 public Students getStudentFromName(String name);
10 public boolean ifEnrolSameCourse(Classes clazz, Students stu);
11 public void selectClasses(Students stu, Classes clazz, Date date);
12 public boolean ifMoreThanCap(Classes clazz);
13 public void updateStudent(Students stu, Contact contact);
14 public HashMap getCourse(PageInfo pageinfo) throws Exception;
15 public HashMap getStudentHistory(PageInfo pageinfo, String stu_name)
throws Exception;
```

```
16
```

```
17 }
```

```
18
```

```
19 实现StudentsServiceImpl.java
```

```
20 public class StudentsServiceImpl implements IStudentsService {
```

```
21
```

```
22     private Logger log = Logger.getLogger(this.getClass());
```

```
23
```

```
24     private IStudents studentsDao;
```

```
25
```

```
26     private ICourses coursesDao;
```

```
27
```

```
28     private IClasses classesDao;
```

```
29
```

```
30     private IDepartment departmentsdao;
```

```
31
```

```
32     /**
```

```
33     * 验证用户名密码
```

```
34     *
```

```
35     * @param username
```

```
36     *     用户名
```

```
37     * @param password
```

```
38     *     密码
```

```
39     */
```

```
40
```

```
41     public boolean validate(String username, String password) {
```

```
42
```

```
43     String password2 = studentsDao.getPasswordFromUsername( usernam
e);
44     if (password.equals(password2))
45         return true;
46     else
47         return false;
48
49 }
50
51 /**
52  * 查找所有课程
53  *
54  */
55 public Courses[] getAllCourses() {
56
57     List list = null;
58     try {
59
60         list = coursesDao.find("select c from Courses as c ");
61     } catch (Exception e) {
62     }
63
64     return (Courses[]) list.toArray(new Courses[0]);
65 }
66
67 /**
68  * 分页显示所有课程
69  *
70  * @param pageinfo
71  */
72 public HashMap getCourse(PageInfo pageinfo) throws Exception {
73
74     HashMap hp = new HashMap();
75     String hsql = "select c from Courses as c order by c.id";
76     Query query = coursesDao.getQuery(hsql);
77     int totalCount = pageinfo.getTatalCount();
78     int totalPage = pageinfo.getTotalpage();
```

```

79     int start = pageinfo.getStart();
80     totalCount = totalCount == -1 ? coursesDao.getTotalCount(hsql)
81         : totalCount;
82     totalPages = totalPages == -1 ? coursesDao.getTotalPage(totalCount,
83         pageinfo.getPageSize()) : totalPages;
84     query.setFirstResult(start);
85     query.setMaxResults(pageinfo.getPageSize());
86     List list = query.list();
87     hp.put("courses", (Courses[]) list.toArray(new Courses[0]));
88     hp.put("totalCount", new Integer(totalCount));
89     hp.put("totalPage", new Integer(totalPage));
90     return hp;
91 }
92 /**
93  * 分页显示所有选课历史
94  * @param pageinfo
95  * @param stu_name
96  */
97 public HashMap getStudentHistory(PageInfo pageinfo, String stu_name)
98     throws Exception {
99     HashMap hp = new HashMap();
100    Students stu = this.getStudentFromName(stu_name);
101    Integer stu_id = stu.getId();
102    Criteria criteria = coursesDao.getCriteria(History.class);
103    criteria.createCriteria("student").add(Expression.eq("name", stu_name));
104    int totalCount = pageinfo.getTotalCount();
105    int totalPages = pageinfo.getTotalPage();
106    int start = pageinfo.getStart();
107    totalCount = totalCount == -1 ? criteria.list().size() : totalCount;
108    totalPages = totalPages == -1 ? studentsDao.getTotalPage(totalCount,
109        pageinfo.getPageSize()) : totalPages;
110    criteria.setFirstResult(start);
111    criteria.setMaxResults(pageinfo.getPageSize());
112    criteria.addOrder(Order.asc("id"));
113    List list = criteria.list();
114    hp.put("history", (History[]) list.toArray(new History[0]));

```

```
115     hp.put("totalCount", new Integer(totalCount));
116     hp.put("totalPage", new Integer(totalPage));
117     return hp;
118 }
119 /**
120  * 根据课程查找班级
121  * @param course
122  *     课程实体
123  * @return 返回该课程下所有班级
124  */
125 public Classes[] getClassesFromCourse(Courses course) {
126     return coursesDao.getClassesFromCourse(course);
127 }
128
129 /**
130  * 根据主键查找系
131  * @param id
132  *     主键ID
133  */
134 public Department getDepFromID(Integer id) {
135     return (Department) departmentsdao
136         .loadByKey(Department.class, "id", id);
137 }
138
139 /**
140  * 根据主键查找课程
141  * @param id
142  *     主键ID
143  */
144 public Courses getCourseFromID(Integer id) {
145     return (Courses) coursesDao.loadByKey(Courses.class, "id", id);
146 }
147 /**
148  * 根据主键查找班级
149  * @param id
150  *     主键ID
151  */
```

```

152 public Classes getClassFromID(Integer id) {
153     return (Classes) classesDao.loadByKey(Classes.class, "id", id);
154 }
155
156 /**
157  * 根据姓名查找学生
158  * @param name
159  */
160 public Students getStudetFromName(String name) {
161     return (Students) studentsDao.loadByKey(Students.class, "name", na
me);
162 }
163
164 /**
165  * 检查学生是否选报了同一课程的班级
166  * @param clazz
167  *     所选报的班级
168  * @param stu
169  *     学生实体
170  * @return true 该生选报同一课程的班级
171  * @return false 没有报过该课程的班级，可以选报
172  *
173  */
174 public boolean ifEnrolSameCourse(Classes clazz, Students stu) {
175
176     Courses cour = clazz.getCourse();
177
178     Classes[] classes = (Classes[]) stu.getClasses()
179         .toArray(new Classes[0]);
180     for (int i = 0; i < classes.length; i++) {
181
182         Courses c1 = classes[i].getCourse();
183
184         if (c1.getId().equals(cour.getId()))
185             return true;
186     }
187     return false;

```

```
188 }
189
190 /**
191  * 检查课程的目前人数
192  * @param clazz
193  *     检查班级人数是否已满
194  * @param clazz
195  *     班级实体
196  * @return true 班级人数已满
197  * @return false 班级人数未滿
198  *
199  */
200 public boolean ifMoreThanCap(Classes clazz) {
201     Integer capacity = clazz.getCapacity();
202     Integer maxcapacity = clazz.getMaxcapacity();
203     if (capacity.intValue() < maxcapacity.intValue()) {
204         clazz.setCapacity(Integer.valueOf(capacity.intValue() + 1));
205         //classesDao.update(clazz);
206         return false;
207     } else
208         return true;
209
210 }
211
212 /**
213  * 数据库插入选择班级的记录
214  * @param stu
215  *     学生
216  * @param clazz
217  *     所选择的班级
218  */
219 public void selectClasses(Students stu, Classes clazz, Date date)
220 {
221     stu.getClasses().add(clazz);
222     clazz.getStudents().add(stu);
223     History his = new History();
224     his.setEnrolTime(date);
```

```
225     his.setStudent(stu);
226     his.setClasses(clazz);
227     his.setScore(clazz.getCourse().getScore());
228     his.setMarking(new Double(0));
229     try{
230         String cour_name=new String(clazz.getCourse().getName().getBytes("
GBK"));
231         his.setCourseName(cour_name);
232     }catch( java.io.UnsupportedEncodingException e){e.printStackTrace();
}
233     stu.getHistory().add(his);
234 }
235
236 public void updateSudent(Students stu,Contact contact){
237
238     studentsDao.update(stu);
239     studentsDao.update(contact);
240
241 }
242 public IStudents getStudentsDao() {
243     return studentsDao;
244 }
245 public void setStudentsDao(IStudents studentsDao) {
246     this.studentsDao = studentsDao;
247 }
248 public IClasses getClassesDao() {
249     return classesDao;
250 }
251 public void setClassesDao(IClasses classesDao) {
252     this.classesDao = classesDao;
253 }
254 public ICourses getCoursesDao() {
255     return coursesDao;
256 }
257 public void setCoursesDao(ICourses coursesDao) {
258     this.coursesDao = coursesDao;
259 }
```

```

260 public IDepartment getDepartmentsdao() {
261     return departmentsdao;
262 }
263 public void setDepartmentsdao(IDepartment departmentdao) {
264     this.departmentsdao = departmentdao;
265 }
266 }
267
268

```

## 4 UI层

这里我们选择Struts，首先配置 web.xml

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <web-app xmlns="http://java.sun.com/xml/ns/j2ee" xmlns:xsi="http
://www.w3.org/2001/XMLSchema-instance" version="2.4" xsi:schema
Location="http://java.sun.com/xml/ns/j2ee http://java.sun.com/xml/n
s/j2ee/web-app_2_4.xsd">
3 <context-param>
4 <param-name>contextConfigLocation< SPAN>param-name>
5 <param-value>/WEB-INF/classes/applications-service.xml< SPA
N>param-value>
6 < SPAN>context-param>
7 <context-param>
8 <param-name>log4jConfigLocation< SPAN>param-name>
9 <param-value>/WEB-INF/log4j.properties< SPAN>param-value>
10 < SPAN>context-param>
11 <filter>
12 <filter-name>hibernateFilter< SPAN>filter-name>
13 <filter-class>org.springframework.orm.hibernate.support.OpenSe
ssionInViewFilter< SPAN>filter-class>
14 < SPAN>filter>
15 <filter-mapping>
16 <filter-name>hibernateFilter< SPAN>filter-name>
17 <url-pattern>/*< SPAN>url-pattern>
18 < SPAN>filter-mapping>

```

```
19 <filter>
20   <filter-name>Set Character Encoding< SPAN>filter-name>
21   <filter-class>limq.struts.SetCharacterEncodingFilter< SPAN>filter-class>
22 </SPAN>filter>
23 <filter-mapping>
24   <filter-name>Set Character Encoding< SPAN>filter-name>
25   <url-pattern>/*< SPAN>url-pattern>
26 </SPAN>filter-mapping>
27 <servlet>
28   <servlet-name>SpringContextServlet< SPAN>servlet-name>
29   <servlet-class>org.springframework.web.context.ContextLoaderServlet< SPAN>servlet-class>
30   <load-on-startup>1< SPAN>load-on-startup>
31 </SPAN>servlet>
32 <servlet>
33   <servlet-name>action< SPAN>servlet-name>
34   <servlet-class>org.apache.struts.action.ActionServlet< SPAN>servlet-class>
35   <init-param>
36     <param-name>config< SPAN>param-name>
37     <param-value>/WEB-INF/struts-config.xml< SPAN>param-value>
38 </SPAN>init-param>
39   <init-param>
40     <param-name>debug< SPAN>param-name>
41     <param-value>3< SPAN>param-value>
42 </SPAN>init-param>
43   <init-param>
44     <param-name>detail< SPAN>param-name>
45     <param-value>3< SPAN>param-value>
46 </SPAN>init-param>
47   <load-on-startup>0< SPAN>load-on-startup>
48 </SPAN>servlet>
49 <servlet-mapping>
50   <servlet-name>action< SPAN>servlet-name>
51   <url-pattern>*.do< SPAN>url-pattern>
```

```
52 < SPAN>servlet-mapping>
53 < SPAN>web-app>
54
55
```

其中注意这几句,

```
1 <filter>
2   <filter-name>hibernateFilter< SPAN>filter-name>
3 <filter-class>org.springframework.orm.hibernate.support.OpenSessi
onInViewFilter< SPAN>filter-class>
4 < SPAN>filter>
5 <filter-mapping>
6   <filter-name>hibernateFilter< SPAN>filter-name>
7   <url-pattern>/*< SPAN>url-pattern>
8 < SPAN>filter-mapping>
9
```

由于我们使用了 `lazy = "true"` , 如果想在UI层使用实体对象关联来获得其他对象时就会有这样的提示 :

`org.hibernate.LazyInitializationException: failed to lazily initialize a collection`

Spring 中引入了 `OpenSessionInView` 模式可以处理以上问题 , 即在 `web.xml` 中加入以上代码。

接下来建立抽象 `BaseAction` , 和 `BaseDispatchAction` , 其中后者与前者相似目的为减少 `Action` 的数量

```
1 abstract class BaseAction extends Action {
2
3   private IStudentsService studentsService;
4   private ITeachersService teachersService;
5   private IAdminService adminService;
6   public void setServlet(ActionServlet actionServlet) {
7     super.setServlet(actionServlet);
```

```

8 ServletContext servletContext = actionServlet.getServletContext();
9 WebApplicationContext wac = WebApplicationContextUtils
10     .getRequiredWebApplicationContext(servletContext);
11
12     this.studentsService = (IStudentsService) wac.getBean("studentManager");
13     this.adminService = (IAdminService) wac.getBean("adminManager");
14     this.teachersService = (ITeachersService) wac.getBean("teacherManager");
15 }
16 public IStudentsService getStudentsService() {
17     return studentsService;
18 }
19 public ITeachersService getTeachersService() {
20     return teachersService;
21 }
22 public void setTeachersService(ITeachersService teachersService) {
23     this.teachersService = teachersService;
24 }
25 public IAdminService getAdminService() {
26     return adminService;
27 }
28 public void setAdminService(IAdminService adminService) {
29     this.adminService = adminService;
30 }
31 }
32

```

BaseDispatchAction与之类似，请查看源码。其他Action都从这两个类继承。

下面就以查看课程下的班级为例演示Struts与Spring的使用：

```

1 CoursesAction.java
2 /**
3  * 查看课程下的班级

```

```

4  */
5  public ActionForward viewClassFromCourse(ActionMapping mapping,
6      ActionForm form, HttpServletRequest request,
7      HttpServletResponse response) throws Exception {
8      Integer cour_id = Integer.valueOf((request.getParameter("cour_id")));
9      Courses cour = super.getStudentsService().getCourseFromID(cour_id);
10     Classes[] clazz =(Classes[])cour.getClasses().toArray(new Classes[0]);
11     request.setAttribute("clazz", clazz);
12     return mapping.findForward("success");
13 }
14

```

这里从上一个页面获得课程编号 `cour_id`, 然后通过 `StudentsServiceImpl` 中的

```

1 public Courses getCourseFromID(Integer id) {
2     return (Courses) coursesDao.loadByKey(Courses.class, "id", id);
3 }
4

```

方法查到 `Courses` 实例, 利用 `Courses` 和 `Classes` 的关联关系得到 `Classes[]`, 在将其放入

`Request`. 通过 `mapping.findForward("success")`, 转发到 `select_course_Content.jsp`

### ***CustomRequestProcessor.java*** 介绍

```

1 public class CustomRequestProcessor extends RequestProcessor {
2     protected boolean processPreprocess(HttpServletRequest request,
3         HttpServletResponse response) {
4         boolean continueProcessing = true;
5         HttpSession session = request.getSession();
6         String uri =request.getRequestURI();
7         if ( session == null || session.getAttribute("userName") == null ) {
8             continueProcessing = false;
9             if(uri.endsWith("login.do")) return true;
10            try{
11                response.sendRedirect("/StudentManger/login.jsp" );

```

```
12     }catch( Exception ex ){
13         log.error( "Problem sending redirect from processPreprocess()" );
14     }
15 }
16 return continueProcessing;
17 }
18 }
19
```

为了验证用户操作权限，这里扩展了Struts 的RequestProcessor来判断Session如果Session和userName都不空则程序继续，否则重定向到login.jsp。要想扩展RequestProcessor类，需在Struts的配置文件中加入

```
1 <controller
2 contentType="text/html;charset=UTF-8"
3 locale="true"
4 nocache="true"
5 processorClass="limq.struts.CustomRequestProcessor"/>
6
```

呵呵，当然在正规使用时仅仅这样验证是不够的。欢迎你把自己修改方法告诉我。

#### 4分页处理：

下面重点讨论一下Hibernate的分页处理方式。

Hibernate 中处理查询主要有 Query ， Criteria ， 分别以 HSQL或编程方式实现，

本例对这两种方法都有相关处理。由于在Spring中无法直接使用Query和Criteria对象

所以只有先从Spring那里借一个Session，等使用完了在还给Sping处理。读者应该还记得在BaseDao中有这样的语句方便我们获取Session及其他对象：

```

1 public Query getQuery(String sql) throws Exception{
2     Session session = this.openSession();
3     Query query = session.createQuery(sql);
4     return query;
5 }
6
7 public Criteria getCriteria(Class clazz) throws Exception{
8
9     Session session=this.openSession();
10    Criteria criteria = session.createCriteria(clazz);
11    return criteria;
12 }
13

```

Service层以查询所有课程与学生选课记录为例处理Query与Criteria：

```

1 StudentsServiceImpl.java
2 public HashMap getCourse(PageInfo pageinfo) throws Exception {
3
4     HashMap hp = new HashMap();
5     String hsql = "select c from Courses as c order by c.id";
6     Query query = coursesDao.getQuery(hsql);
7     int totalCount = pageinfo.getTotalCount();
8     int totalPage = pageinfo.getTotalpage();
9     int start = pageinfo.getStart();
10    totalCount = totalCount == -1 ? coursesDao.getTotalCount(hsql)
11        : totalCount;
12    totalPage = totalPage == -1 ? coursesDao.getTotalPage(totalCount,
13        pageinfo.getPageSize()) : totalPage;
14    query.setFirstResult(start);
15    query.setMaxResults(pageinfo.getPageSize());
16    List list = query.list();
17    hp.put("courses", (Courses[]) list.toArray(new Courses[0]));
18    hp.put("totalCount", new Integer(totalCount));
19    hp.put("totalPage", new Integer(totalPage));
20    return hp;

```

```

21 }
22
23 public HashMap getStudentHistory(PageInfo pageinfo, String stu_name)
24     throws Exception {
25     HashMap hp = new HashMap();
26     Students stu = this.getStudetFromName(stu_name);
27     Integer stu_id = stu.getId();
28     Criteria criteria = coursesDao.getCriteria(History.class);
29     criteria.createCriteria("student").add(Expression.eq("name", stu_name)
30 );
31     int totalCount = pageinfo.getTatalCount();
32     int totalPage = pageinfo.getTotalpage();
33     int start = pageinfo.getStart();
34     totalCount = totalCount == -1 ? criteria.list().size() : totalCount;
35     totalPage = totalPage == -1 ? studentsDao.getTotalPage(totalCount,
36 pageinfo.getPageSize()) : totalPage;
37     criteria.setFirstResult(start);
38     criteria.setMaxResults(pageinfo.getPageSize());
39     criteria.addOrder(Order.asc("id"));
40     List list = criteria.list();
41     hp.put("history", (History[]) list.toArray(new History[0]));
42     hp.put("totalCount", new Integer(totalCount));
43     hp.put("totalPage", new Integer(totalPage));
44     return hp;
45 }
46 PageIngfo.java
47 public class PageInfo {
48     int pageNo=0;
49     int totalpage=-1;
50     int tatalCount=-1;
51     int pageSize=0;
52     int start=0;
53     ....
54

```

可以看到getCourse和getStudentHistory有很多相似之处，Hibernate为Query和Criteria提供了针对不同数据库的解决分页方法，Query需要我们写HSQL，Criteria不但可以应付带有条件的查询，还不用我们自己写HSQL，PageInfo是含有分页信息的普通java类。

再看看Struts是如何调用getStudentHistory的，

```
1 PageAction.java
2 public class PageAction extends BaseDispatchAction{
3     public ActionForward execute(ActionMapping mapping,
4         ActionForm form,
5         HttpServletRequest request,
6         HttpServletResponse response)
7     throws Exception {
8         String pageNo=request.getParameter("pageNo");
9         String totalcount=request.getParameter("totalcount");
10        String totalpage=request.getParameter("totalpage");
11        int pagesize=2;//每页的大小
12        PageInfo page =new PageInfo();
13        page.setPageSize(pagesize);
14        HashMap hp=null;
15        History[] historys = null;
16        String stu_name=null;
17        HttpSession session = request.getSession();
18        stu_name = (String) session.getAttribute("userName");
19        if(pageNo == null || totalcount == null || totalpage==null){
20            //第一次发送请求
21            page.setPageNo(1);
22            hp=super.getStudentsService().getStudentHistory(page,stu_name);
23            page.setTatalCount(((Integer)hp.get("totalCount")).intValue());
24            page.setTotalpage(((Integer)hp.get("totalPage")).intValue());
25        }else{
26            page.setPageNo(Integer.parseInt(pageNo));
27            page.setTatalCount(Integer.parseInt(totalcount));
28            page.setTotalpage(Integer.parseInt(totalpage));
29            hp=super.getStudentsService().getStudentHistory(page,stu_name);
30
```

```

31     }
32     historys =(History[]) hp.get("history");
33     request.setAttribute("history",historys);
34     request.setAttribute("pageinfo",page);
35     return mapping.findForward("success");
36 }
37 }
38

```

在stu\_his\_Content.jsp中避免代码重复使用了自定义标志来处理分页

```

1 @ page contentType="text/html;charset=UTF-8" language="java" %>
2 @ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean" %>
3 @ taglib uri="/WEB-INF/struts-html.tld" prefix="html" %>
4 @ taglib uri="/WEB-INF/struts-logic.tld" prefix="logic" %>
5 @ page import="limq.hibernate.vo.*"%>
6 @ taglib uri="/WEB-INF/MyTag.tld" prefix="mytag"%>
7 @ page import="limq.common.*"%>
8 <html:html locale="true">
9 <body>
10
11 PageInfo pageinfo =(PageInfo) request.getAttribute("pageinfo");
12 History[] historys = (History[])request.getAttribute("history");
13 %>
14 <table width="550" border="1" cellspacing="0" align="center" cellpadding="0">
15 <tr>
16 <td><bean:message key="class.id"/>< SPAN>td>
17 <td><bean:message key="course.name"/>< SPAN>td>
18 <td><bean:message key="enrol.time"/>< SPAN>td>
19 <td><bean:message key="score"/>< SPAN>td>
20 <td><bean:message key="marking"/>< SPAN>td>
21 < SPAN>tr>
22
23 for(int i=0;i<historys.length;i++){
24 History his=historys[i];

```

```

25 %>
26 <tr>
27   <td>his.getClasses().getId()%>< SPAN>td>
28   <td>his.getCourseName()%>< SPAN>td>
29   <td>his.getEnrolTime()%>< SPAN>td>
30   <td>his.getScore()%>< SPAN>td>
31   <td>his.getMarking()%>< SPAN>td>
32 < SPAN>tr>
33
34 }
35 %>
36 < SPAN>table>
37 <mytag:page pageinfo="" action="getHistory.do"/>
38 < SPAN>body>
39 < SPAN>html:html>
40

```

标志处理类如下：

```

1 PageTag.java
2
3 public class PageTag extends SimpleTagSupport {
4
5   private PageInfo pageinfo = null;
6   private String action = null;
7
8   public String getAction() {
9     return action;}
10  public void setAction(String action) {
11    this.action = action;
12  }
13  public PageInfo getPageinfo() {
14    return pageinfo;
15  }
16  public void setPageinfo(PageInfo pageinfo) {
17    this.pageinfo = pageinfo;

```

```

18 }
19
20 public void doTag() throws JspException, IOException {
21     JspWriter out = getJspContext().getOut();
22
23     int totalpage = pageinfo.getTotalpage();
24     int totalcount = pageinfo.getTatalCount();
25     int pageNo = pageinfo.getPageNo();
26     int addPageNo = pageNo + 1;
27     int minPageNo = pageNo - 1;
28
29     out.println("<table border=400 align=center cellpadding=0 cellspacing=0
30 >");
31     out.print("<tr>");
32     out.println("<td align=left>");
33
34         if (pageNo > 1) {
35             out
36                 .println("<a href='&'/StudentManger/" +
37                     action
38                     + "?pageNo=1" + "&totalpage=" + totalpage + "&totalcount="
39                     + totalcount + ">");
40         }
41         out.print("<td align=right>");
42         if (pageNo > 1) {
43             out.println("<a href='&'/StudentManger/" + action + "?pageNo="
44                 + minPageNo + "&totalpage=" + totalpage + "&totalcount="
45                 + totalcount + ">");
46         }
47         out.print("<td align=right>");
48         out.println("<td align=right>");
49         if (pageNo < totalpage) {
50             out.println("<a href='&'/StudentManger/" + action

```

```

        ction + "?pageNo="
51         + addPageNo + "&totalpage=" +
        totalpage + "&totalcount="
52         + totalcount + "\">");
53     }
54     out.print("下页");
55     out.println(" ");
56     if (pageNo < totalpage) {
57
58         out.println("/StudentManger/" + act
        ion + "?pageNo="
59         + totalpage + "&totalpage=" + t
        otalpage + "&totalcount="
60         + totalcount + "\">");
61
62     }
63     out.print("末页");
64     out.println(" ");
65     out.println("
");
66
67 }
68
69 }
70

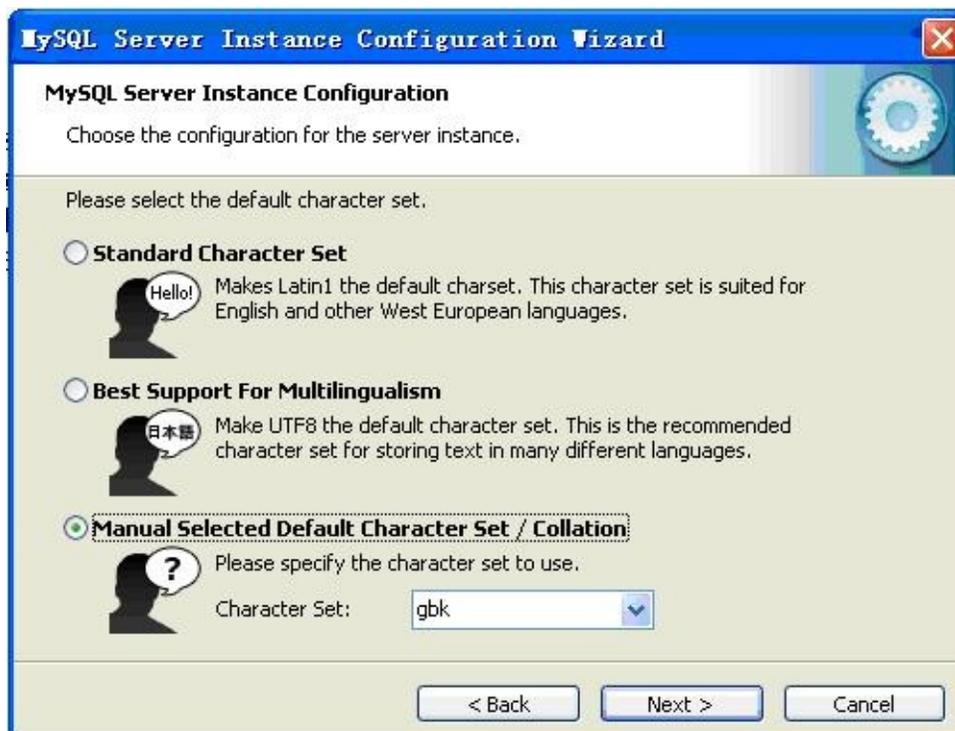
```

## 5 中文乱码问题：

1 数据库：MYSQL 4.1（或以上版本）4.1直接支持Unicode，以下版本支持的不好。

2 驱动：MySQL JDBC Driver的3.0.16（或以上版本）

3 在数据库中做如下设定



4 在建立表时同样加上ENGINE=MyISAM DEFAULT CHARSET=gbk

```
1 CREATE TABLE `students` (  
2   `id` int(20) NOT NULL default '0',  
3   `name` varchar(20) NOT NULL default "",  
4   `department_id` int(11) default NULL,  
5   `password` varchar(20) default NULL,  
6   `score` double(15,3) default NULL,  
7   PRIMARY KEY (`id`)  
8 ) ENGINE=MyISAM DEFAULT CHARSET=gbk  
9
```

5 配置hibernate.cfg.xml

```
1 <property name="connection.url">jdbc:mysql://localhost:3306/Student< SPAN>property>  
2 <property name="dialect">net.sf.hibernate.dialect.MySQLDialect< SPAN>property>  
3 <property name="connection.password">< SPAN>property>  
4 <property name="connection.driver_class">com.mysql.jdbc.Driver< SPAN>property>
```

**robbin**：MySQL JDBC Driver的3.0.16也是一个分水岭，3.0.16版本会取数据库本身的编码，然后按照该编码转换，这种方式 and Oracle的JDBC Driver是一样的。例如你的数据库是GBK编码的话，JDBC Driver就会把数据库里面的取出来的字符串按照GBK往unicode转换，送给JVM。因此正确的设置数据库本身的编码就尤为重要。

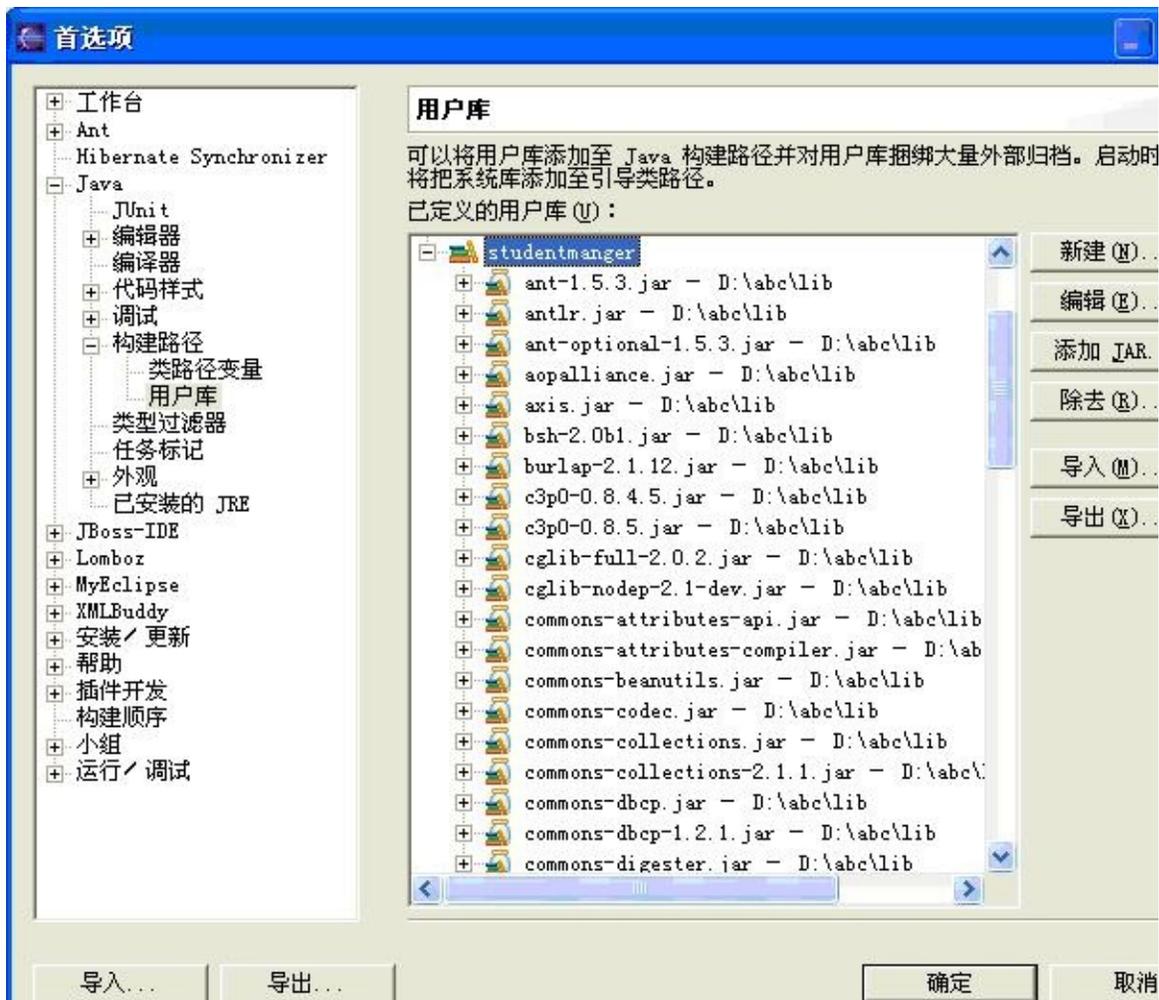
MySQL JDBC Driver3.0.16以下的版本则不然，它不会那么智能的根据数据库编码来确定如何转换，它总是默认使用ISO8859-1，因此你必须使用 `characterEncoding=GBK`来强制他把数据库中取出来的字符串按照GBK来往unicode转换。因此，使用什么数据库版本，不管是3.x，还是4.0.x还是4.1.x，其实对我们来说不重要，重要的有二：

- 1) 正确的设定数据库编码，MySQL4.0以下版本的字符集总是默认ISO8859-1，MySQL4.1在安装的时候会让你选择。如果你准备使用UTF-8，那么在创建数据库的时候就要指定好UTF-8(创建好以后也可以改，4.1以上版本还可以单独指定表的字符集)
- 2) 使用3.0.16以上版本的JDBC Driver，那么你就不需要再写什么 `characterEncoding=UTF-8`

## 6 开发工具介绍

### MyEclipse 3.8

首先添加用户库，如下图将Struts, Spring, Hibernate 的库添加到用户库中



如果出现环境问题可能你的Struts包有问题，请到<http://struts.apache.org/download.cgi>下载[struts-1.2.7-lib.zip](http://struts.apache.org/download.cgi)。具体使用参考<http://www.laliluna.de/struts-hibernate-integration-tutorial-en.html>

## 总结

本文至此已将Struts+Spring+Hibernate的大致思路以及本人所遇到的难点，重点介绍完了。

其中管理员我只完成了对学生的部分，其他功能大同小异，有兴趣的读者不妨动手试试。最后建议初学者不要直接使用Spring对Hibernate的封装，而是从Hibernate学起，先要学会自己管理Session，Transaction，然后在用Spring，这样理解会更深刻。同时如果你有好的建议，或问题请联系我

QQ 39315890

Email: [mill\\_lm@yaho.com.cn](mailto:mill_lm@yaho.com.cn)

参考：

**Spring Framework**之最佳实践：

<http://www.gpowersoft.com/tech/Spring/>

**Hibernate+Spring** 对DAO的处理实例

[http://www.javaeye.com/viewtopic.php?](http://www.javaeye.com/viewtopic.php?t=7923&start=0&postdays=0&postorder=asc&highlight=)

[t=7923&start=0&postdays=0&postorder=asc&highlight=](http://www.javaeye.com/viewtopic.php?t=7923&start=0&postdays=0&postorder=asc&highlight=)

**Hibernate**实现分页查询的原理分析

[http://forum.javaeye.com/viewtopic.php?](http://forum.javaeye.com/viewtopic.php?t=261&start=0&postdays=0&postorder=asc&highlight=)

[t=261&start=0&postdays=0&postorder=asc&highlight=](http://forum.javaeye.com/viewtopic.php?t=261&start=0&postdays=0&postorder=asc&highlight=)

**Spring**的DAO设计实践

[http://www.hibernate.org.cn/viewtopic.php?](http://www.hibernate.org.cn/viewtopic.php?t=8224&start=0&postdays=0&postorder=asc&highlight=)

[t=8224&start=0&postdays=0&postorder=asc&highlight=](http://www.hibernate.org.cn/viewtopic.php?t=8224&start=0&postdays=0&postorder=asc&highlight=)

用 **OpenSessionInViewInterceptor** 的思路解决**Hibernate Lazy**问题

[http://www.javaeye.com/viewtopic.php?](http://www.javaeye.com/viewtopic.php?t=14631&start=0&postdays=0&postorder=asc&highlight=)

[t=14631&start=0&postdays=0&postorder=asc&highlight=](http://www.javaeye.com/viewtopic.php?t=14631&start=0&postdays=0&postorder=asc&highlight=)

## 用缓冲技术提高JSP应用的性能和稳定性

### 一、概述

在Web应用中，有些报表的生成可能需要数据库花很长时间才能计算出来；有的网站提供天气信息，它需要访问远程服务器进行SOAP调用才能得到温度信息。所有这一切都属于复杂信息的例子。在Web页面中加入过多的复杂信息可能导致Web服务器、数据库服务器负荷过重。JSP代码块缓冲为开发者带来了随意地增加各种复杂信息的自由。

JSP能够在标记库内封装和运行复杂的Java代码，它使得JSP页面文件更容易维护，使得非专业开发人员使用JSP页面文件更加方便。现在已经有许多标记库，它们或者是商业产品，或者是源代码开放产品。但这些产品中的大多数都只是用标记库的形式实现原本可以用一个简单的Java Scriptlet实现的功能，很少有产品以某种创造性的方式使用定制标记，提供在出现JSP定制标记库之前几乎不可能实现的用法。

OSCache标记库由OpenSymphony设计，它是一种开创性的JSP定制标记应用，提供了在现有JSP页面之内实现快速内存缓冲的功能。虽然已经有一些供应商在提供各种形式的缓存产品，但是，它们都属于面向特定供应商的产品。OSCache能够在任何JSP 1.1兼容的服务器上运行，它不仅能够为所有用户缓冲现有JSP代码块，而且能够以用户为单位进行缓冲。OSCache还包含一些提高可伸缩性的高级特性，比如：缓冲到磁盘，可编程的缓冲刷新，异常控制，等等。另外，正如OpenSymphony的其他产品，OSCache的代码也在一个开放源代码许可协议之下免费发行。

本文以一个假想的拍卖网站设计过程为例，介绍OSCache的工作过程。这个假想的Web网站将包含：一个报告最近拍卖活动的管理页面；一个功能完整、带有各种宣传信息的主页；一个特殊的导航条，它包含了用户所有尚未成交的拍卖活动信息。

### 二、管理页面

拍卖网站包含一个管理报表，数据库服务器需要数秒时间才能创建

这样一个报表。报表生成时间长这一点很重要，因为我们可能让多个管理员监视系统运行情况，同时又想避免管理员每次访问时都重新生成这个报表。为了实现这一点，我们将把整个页面封装到一个应用级的缓冲标记之内，这个缓冲标记每隔1小时刷新。其他供应商提供的一些产品也具有类似的功能，只是OSCache比它们做得更好。

为简单计，我们将不过多地关注格式问题。在编写管理页面时，我们首先把标记库声明加入到页面：

```
<%@ taglib uri="cachetags" prefix="cache"
%>
```

接下来我们要用cache标记来包围整个页面。cache标记的默认缓冲时间是1小时。

```
<cache:cache> .... 复杂的管理报表 ....
</cache:cache>
```

现在管理页面已经被缓冲。如果管理员在页面生成后的一个小时内再次访问同一页面，他看到的将是以前缓存的页面，不需要由数据库服务器再次生成这个报表。

### 三、主页

拍卖网站的主页显示网站活动情况，宣传那些即将结束的拍卖活动。我们希望显示出正在进行的拍卖活动数量，当前登录用户数量，在短期内就要结束的拍卖活动的清单，以及当前时间。这些信息有着不同的时间精确度要求。网站上的拍卖活动通常持续数天，因此我们可以把缓冲有效拍卖活动数量的时间定为6个小时。用户数量的变化显然要频繁一些，但这里我们将把这个数值每次缓冲15分钟。最后，我们希望页面中显示的当前时间总是精确的页面访问时间。

在主页中声明标记库之后，我们首先以不带缓冲的方式直接输出当

前日期：

```
现在是：<%=new java.util.Date()%>
```

接下来，我们要显示一个清单，列出那些将在短期内结束的拍卖活动：

```
<cache:cache><ul>
<%
// 构造一个包含最近拍卖活动的
Iterator Iterator auctions = ....
while (auctions.hasMore())
{
    Auction auction = (Auction)auctions.next();
%>
    <li><%=auction%></li>
<%
}
%>
</ul> </cache:cache>
```

最后，我们希望显示出正在进行的拍卖活动的数量，这个数字需要缓冲6小时。由于cache标记需要的是缓冲数据的秒数，我们把6小时转换成21600秒：

```
<cache:cache time="21600">
<%
//查询数据库得到拍 卖活动总数
int auctionCount = ....
%>
本网站正在进行的拍卖活动有
<%=auctionCount%>个!
</cache>
```

可以看到，我们只用少量的代码就构造出了一个带有复杂缓冲系统的主页。这个缓冲系统对页面各个部分分别进行缓冲，而且各个部分的缓冲时间完全符合它们各自的信息变化频繁程度。由于有了缓冲，现在我们可以将更多的内容放入主页中；而在以前没有缓冲的情况下，主页中放入过多的内容会导致页面访问速度变慢，甚至可能给数据库服务器带来过重的负载。

#### 四、导航条

假设在规划网站的时候，我们决定在左边导航条的下方显示购物车内容。我们将显示出用户所拍卖的每一种商品的出价次数和当前报价，以及所有那些当前用户出价最高的商品的清单。

我们利用会话级的缓冲能力在导航条中构造上述功能。把下面的代码放入模板或者包含文件，以便网站中的其他页面引用这个导航条：

```
<cache:cache key="navbar" scope="session"
time="300">
<%
//提取并显示当前的出价信息
%>
</cache:cache>
```

在这里我们引入了两个重要的属性，即key和scope。在本文前面的代码中，由于cache标记能够自动为代码块创建唯一的key，所以我们不需要手工设置这个key属性。但在这里，我们想要从网站的其余部分引用这个被缓冲的代码块，因此我们显式定义了该cache标记的key属性。第二，scope属性用来告诉cache标记当前代码块必须以用户为单位缓冲，而不是为所有用户缓冲一次。

在使用会话级缓冲时应该非常小心，应该清楚：虽然我们可以让复杂的导航条减少5倍或10倍的服务器负载，但它将极大地增加每个会话所需要的内存空间。在CPU能力方面增加可能的并发用户数量无疑很理想，但是，一旦在内存支持能力方面让并发用户数量降低到了CPU的限制之下，这个方案就不再理想。

正如本文前面所提到的，我们希望从网站的其余部分引用这个缓冲的代码块。这是因为，当一个用户增加了一个供拍卖的商品、或者出价竞购其他用户拍卖的商品时，我们希望刷新缓冲，使得导航条下一次被读取时具有最新的内容。虽然这些数据可能因为其他用户的活动而改变，但如果用户在网站上执行某个动作之后看到自己的清单仍未改变，他可能会感到非常困惑。

OSCache库提供的flush标记能够刷新缓冲内容。我们可以把下面的代码加入到处理用户动作且可能影响这一区域的页面之中：

```
<cache:flush key="navbar" scope="session" />
```

当用户下次访问它时，navbar缓冲块将被刷新。

至此为止，我们这个示例网站的构造工作已经完成且可以开始运行。下面我们来看看OSCache的异常处理能力。即使缓冲的内容已经作废，比如在缓冲块内出现了Java异常，OSCache标记库仍旧允许我们用编程的方法显示这些内容。有了这种异常控制功能，我们可以拆除数据库服务器和Web服务器之间的连接，而网站仍能够继续运行。JSP 1.2规范引入了TryCatchFinally接口，这个接口允许标记本身检测和处理Java异常。因此，标记可以结合这种异常处理代码，使得JSP页面更简单、更富有条理。

OpenSymphony正在计划实现其他的缓冲机制以及一个可管理性更好的主系统，它将使我们能够对缓冲使用的RAM和磁盘空间进行管理。一旦有了这些功能，我们就能够进一步提高网站的响应速度和可靠性。

**【结束语】** OSCache能够帮助我们构造出更丰富多彩、具有更高性能的网站。有了OSCache标记库的帮助，现在我们能够用它解决一些影响网站响应能力的问题，比如访问量高峰期、数据库服务器负荷过重等。