

Grails开发手册

Author:

Creation Date: 2015年03月30日

Last Updated: 2015年04月29日

Document Ref:

Version: 1.0

1. **Title, Subject, Last Updated Date, Reference Number**, **and** **Version** are marked by a Word Bookmark so that they can be easily reproduced in the header and footer of documents. When you change any of these values, be careful not to accidentally delete the bookmark. **You can make bookmarks visible by selecting Tools->Options…View and checking the Bookmarks option in the Show region.**
2. To add additional approval lines, press [Tab] from the last cell in the table above.
3. You can delete any elements of this cover page that you do not need for your document. For example, Copy Number is only required if this is a controlled document and you need to track each copy that you distribute.



## 文档控制

**更改记录**

| 日期 | 章节 | | 作者 | 版本 | 更改参考 |
| --- | --- | --- | --- | --- | --- |
|  |  | | |  |  |
| 2015/03/30 | 1.环境准备 | 王亦阳 | | 1.0 | 无前版本 |
| 2015/03/31 | 2.Grails知识点 | 王亦阳 | | 1.0 | 无前版本 |
| 2015/04/01 | 3.域类 | 王亦阳 | | 1.0 | 无前版本 |
| 2015/04/02 | 4.控制器 | 王亦阳 | | 1.0 | 无前版本 |
| 2015/04/06 | 5.视图 | 王亦阳 | | 1.0 | 无前版本 |
| 2015/04/07 | 6.URL映射 | 王亦阳 | | 1.0 | 无前版本 |
| 2015/04/08 | 7.国际化、CRUD基本功能开发 | 王亦阳 | | 1.0 | 无前版本 |
| 2015/04/09 | 8.Ajax、CRUD基本功能补充 | 王亦阳 | | 1.0 | 无前版本 |
| 2015/04/10 | 9.GORM | 王亦阳 | | 1.0 | 无前版本 |
| 2015/04/11 | 10.Service、集成和依赖管理 | 王亦阳 | | 1.0 | 无前版本 |
| 2015/04/12 | 11.插件 | 王亦阳 | | 1.0 | 无前版本 |
| 2015/04/28 | 12.使用database-migration插件 | 金鑫 | | 1.0 | 无前版本 |
| 2015/04/29 | 13.克隆项目、错误消息（i18n） | 王亦阳 | | 1.0 | 无前版本 |
|  |  |  | |  |  |
|  |  |  | |  |  |
|  |  |  | |  |  |
|  |  |  | |  |  |
|  |  |  | |  |  |
|  |  |  | |  |  |
|  |  |  | |  |  |
|  |  |  | |  |  |

4

内容目录

[文档控制 2](#_Toc418075339)

[1. 环境准备 4](#_Toc418075340)

[1.1. 下载软件 4](#_Toc418075341)

[1.2. 创建目录 4](#_Toc418075342)

[1.3. 安装JDK 4](#_Toc418075343)

[1.4. Posh-gvm安装 5](#_Toc418075344)

[1.5. Grails2.5安装 7](#_Toc418075345)

[1.6. IntelliJ IDEA安装 8](#_Toc418075346)

[1.7. IDEA使用 10](#_Toc418075347)

[1.8. 克隆项目 13](#_Toc418075348)

[2. Grails知识点 15](#_Toc418075349)

[2.1. 综述 15](#_Toc418075350)

[2.2. 域类 17](#_Toc418075351)

[2.3. 控制器 21](#_Toc418075352)

[2.4. 视图 30](#_Toc418075353)

[2.5. URL映射 36](#_Toc418075354)

[2.6. 国际化 42](#_Toc418075355)

[2.7. Ajax 45](#_Toc418075356)

[2.8. GORM 48](#_Toc418075357)

[2.9. Services 62](#_Toc418075358)

[2.10. 集成和依赖管理 66](#_Toc418075359)

[2.11. 插件 70](#_Toc418075360)

[3. CRUD基本功能开发 82](#_Toc418075361)

[3.1. 创建项目 82](#_Toc418075362)

[3.2. 配置数据库连接 82](#_Toc418075363)

[3.3. 创建域类 84](#_Toc418075364)

[3.4. 创建控制器 85](#_Toc418075365)

[3.5. 创建视图 87](#_Toc418075366)

[3.6. 运行程序 88](#_Toc418075367)

[4. 插件应用 89](#_Toc418075368)

[4.1. 使用database-migration插件 89](#_Toc418075369)

[5. 技术点总结 92](#_Toc418075370)

[5.1. 错误消息（i18n） 92](#_Toc418075371)

## 环境准备

本章节重点介绍Windows系统下Grails环境的安装和配置。

### 下载软件

Grails开发环境所需软件：

1. Jdk 7
2. PowerShell
3. GVM (the Groovy enVironment Manager)
4. Grails 2.5
5. Intellij IDEA 14.1

【请从相关网站上下载所需软件】

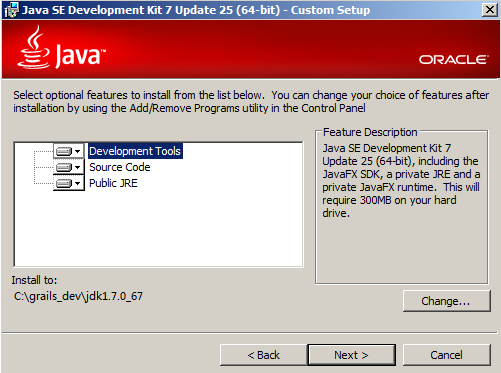
### 创建目录

为了使安装和说明更为简练，我们在根目录下创建一个名为grails\_dev的目录（如在C盘的根目录下C:\ grails\_dev），然后将后面所有的软件都装在这个目录下。

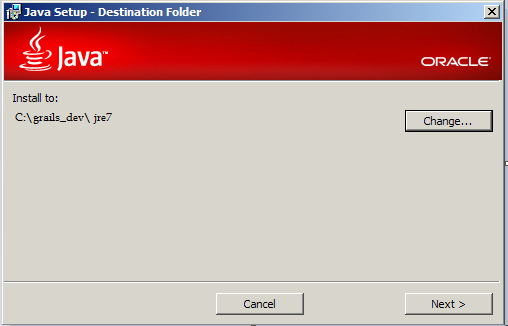
### 安装JDK

#### 软件安装

1. 双击打开安装程序，在第二步时改变安装路径，使其安装到grails\_dev目录下，点击Next进行安装，如C:\grails\_dev\jdk1.7.0\_67，如下图所示：



1. 当弹出安装JRE窗口时，同样需要更改默认安装路径，将其安装到grails\_dev目录下，如：C:\grails\_dev\ jre7，如下图所示：



#### 相关配置

1. 右键我的电脑，然后点击 属性—>系统属性—>高级—>环境变量。在系统变量中新建变量，变量名为JAVA\_HOME，变量值为JDK所在目录如：C:\grails\_dev\jdk1.7.0\_67。
2. 在系统变量中找到classpath，编辑其变量值，在最后一个分号“;”后添加.;%JAVA\_HOME%\lib;%JAVA\_HOME%\lib\tools.jar。如果没有classpath，则新建一个并添加变量值。
3. 在系统变量中找到Path，编辑其变量值，在最后一个分号“;”后添加%JAVA\_HOME%\bin;%JAVA\_HOME%\jre\bin，然后一直点击确定保存更改。

#### 测试

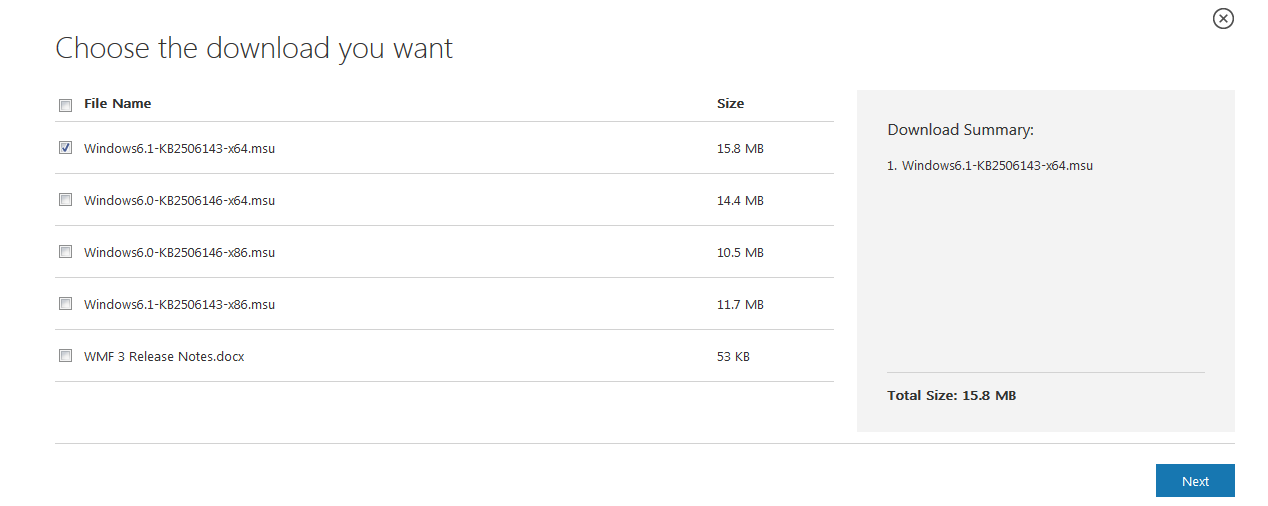
打开命令行窗口。输入java –version，如果可以查看到java version符合安装版本，则说明已正确配置。

### Posh-gvm安装

下面介绍在windows下安装posh-gvm，GVM的安装是为了方便管理安装Grails准备的，如果GVM安装不成功可直接按照1.5.3节进行安装Grails。

#### 环境准备

1. 安装PowerShell3.0版本（地址：http://www.microsoft.com/en-us/download/details.aspx?id=34595），选择相应的文件下载并安装。

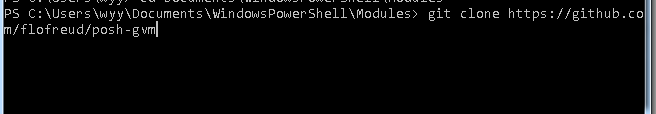


1. 安装完成后在“命令提示符”下输入“powershell”，再输入“$env:PSModulePath”，查看modules directory，根据查看的路径检查是否存在该路径，不存在创建该文件夹。



#### posh-gvm安装

1. 在“命令提示符”中，进入刚刚创建的modules directory文件夹（C:\Users\wyy\Documents\WindowsPowerShell\Modules），通过Git克隆posh-gvm。

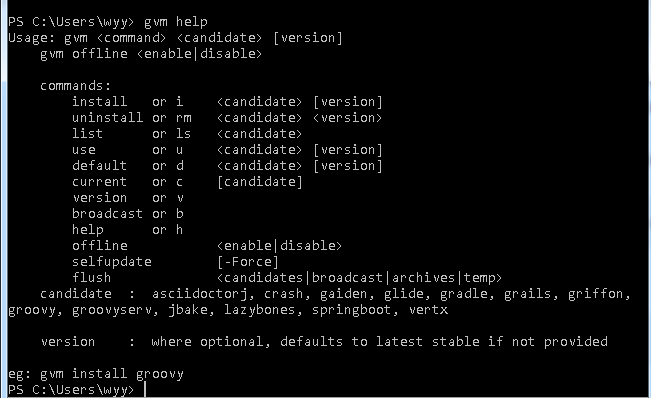


克隆地址：git clone <https://github.com/flofreud/posh-gvm>

1. 在powershell下 输入 “Import-Module posh-gvm” 指令导入posh-gvm模型，在执行过程中会出现“powershell-无法加载文件,因为在此系统中禁止执行脚本…”错误，则先执行一次“set-executionpolicy remotesigned”指令，再执行“Import-Module posh-gvm”指令。

#### 测试

在powershell下输入 “gvm help”，如出现如下效果，则安装成功。



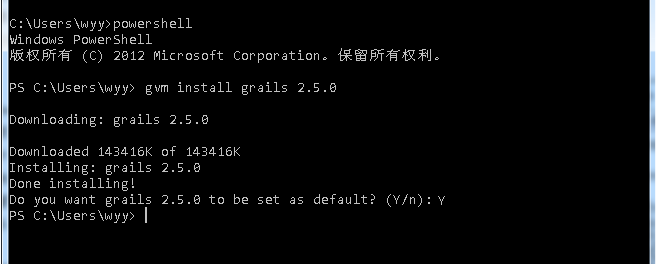
### Grails2.5安装

#### 安装前提条件

在安装Grails以前，你至少需要先安装1.6或者更高版本的JDK，并且设置名为JAVA\_HOME的全局环境变量来指向它。

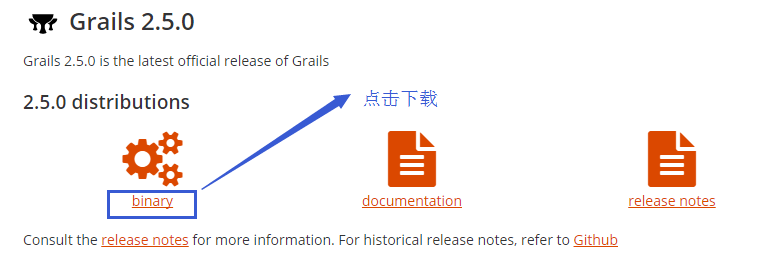
#### 安装方法一：通过GVM安装

在“命令提示符”下输入“powershell”，再输入“gvm install grails 2.5.0”下载安装。下载完成可在C:\user\计算机名\.posh\_gvm\grails下查看到下载安装的grails。

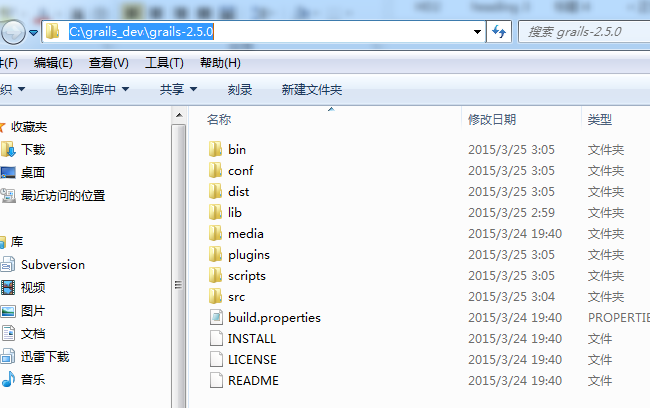


#### 安装方法二：解压缩安装

1. 下载 Grails（地址：http://www.grails.org/Download），找到图片对应的版本点击下载。



1. 解压到你指定的文件目录下（统一解压到C:\ grails\_dev下）。

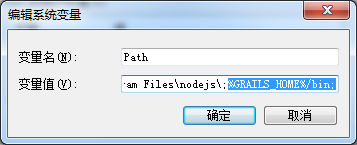


#### 相关配置

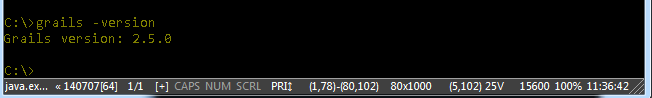
1. 在环境变量中添加GRAILS\_HOME,值为解压的文件目录（方法一：C:\user\计算机名\.posh\_gvm\grails\grails-2.5.0，方法二：C:\ grails\_dev\grails-2.5.0）。



1. 将GRAILS\_HOME的bin目录追加到系统的PATH变量中（%GRAILS\_HOME%/bin;）。



#### 测试

如果环境变量设置无误，此时可以打开“命令提示符”，输入grails -version 如果屏幕上显示如下提示，则说明安装成功。

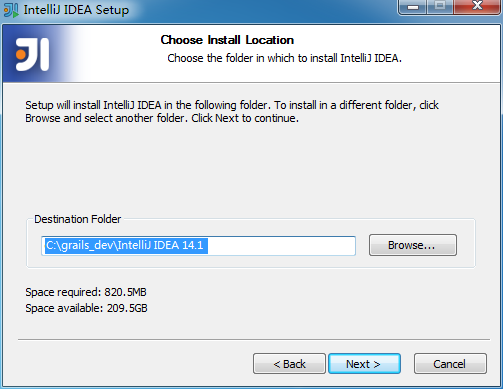
### IntelliJ IDEA安装

#### 环境准备

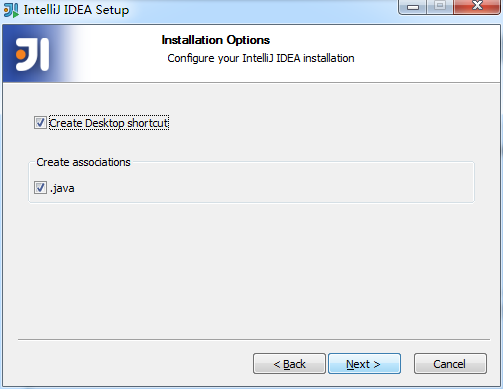
1. 下载 IDEA（地址：http://www.grails.org/Download），找到图片对应的版本点击下载



1. 解压安装IDEA，更改安装路径为C:\ grails\_dev \IntelliJ IDEA 14.1。



全部勾选，下一步，直到完成。



1. 启动IDEA，会出现下面弹窗，填入注册码即可。如下面提供的注册码都不能使用，请自行百度查询一个。

**User:** xring

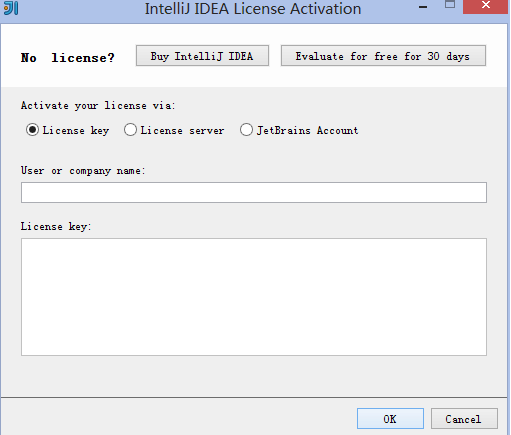
**Key:** 21423-V4P36-U7W8K-8CYUK-93HXA-MKGZ5

**User:** arix

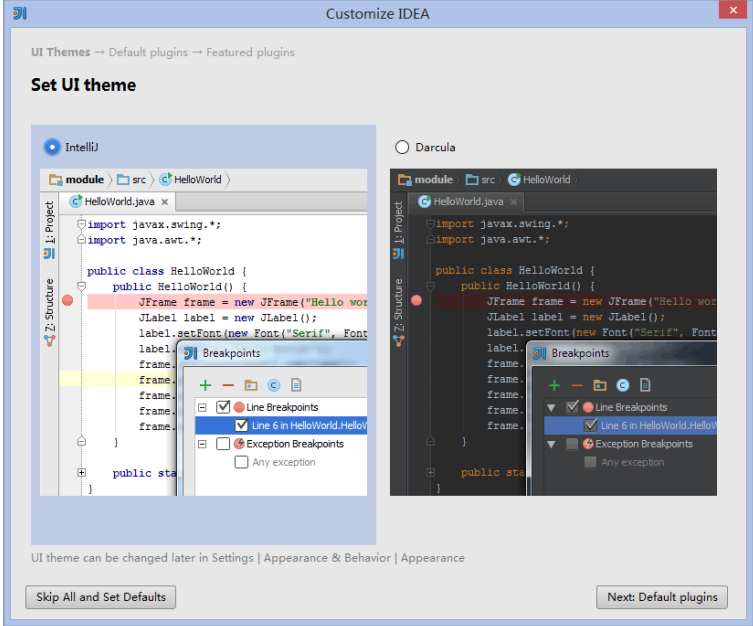
**Key:** 52998-LJT74-J7YEX-UPVT3-Q5GUF-5G4B5

**User:** liberal

**Key:**  68936-FTXB9-B92UI-Y7J2E-N5TN6-0TB37



根据喜好选择界面风格

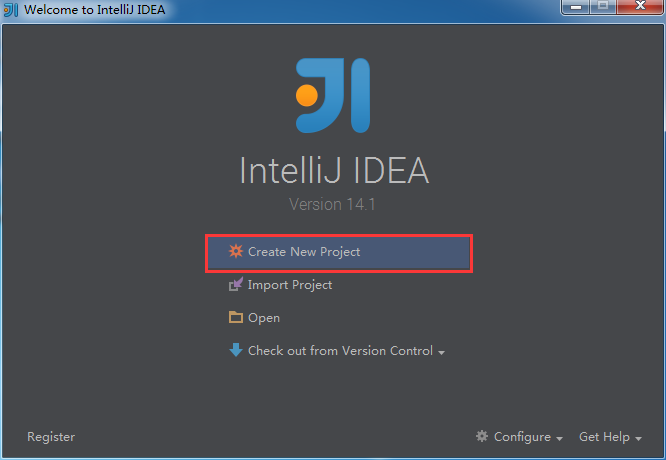


下一步，直接点击跳过即可。

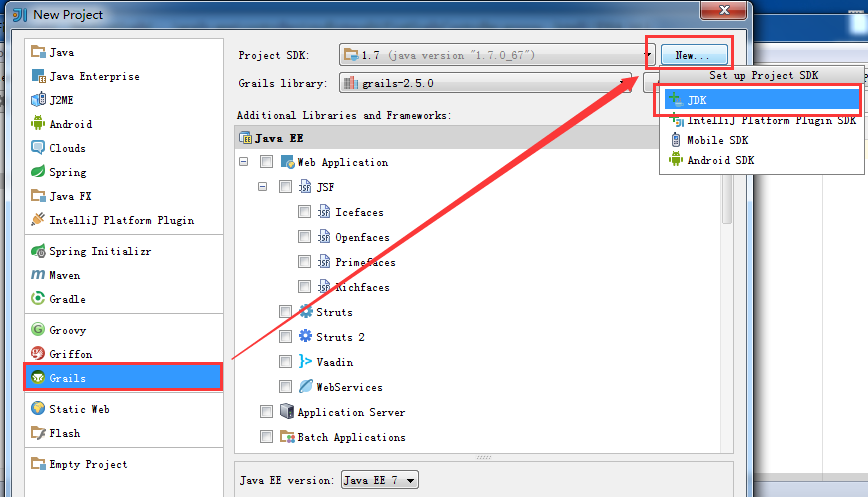
### IDEA使用

#### 创建项目

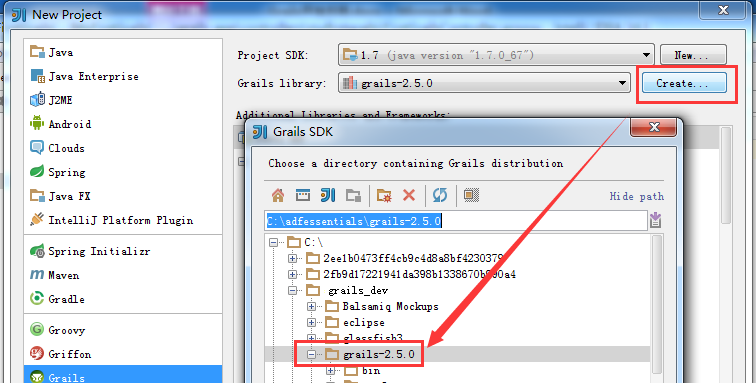
1. 启动软件后点击 Create New Project。



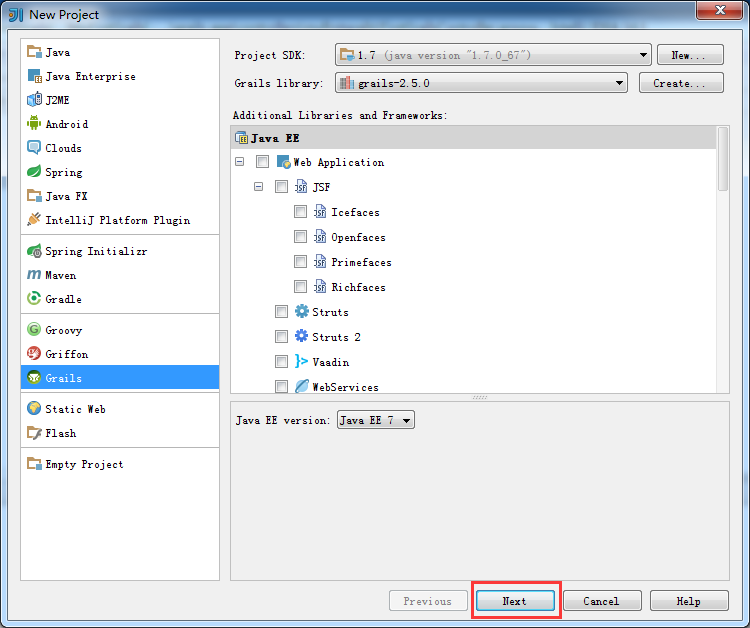
1. 选择左边栏Grails项目，然后选择Project SDK，New一个JDK（选择：C:\ grails\_dev \jdk1.7.0\_67）。



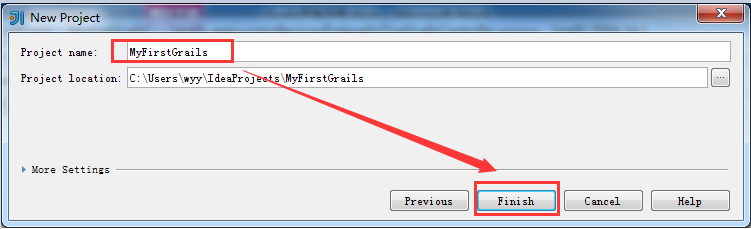
再选择Grails Library，点击Create（选择已安装的grails-2.5.0）。



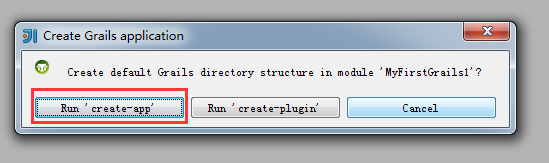
1. 直接点击Next。



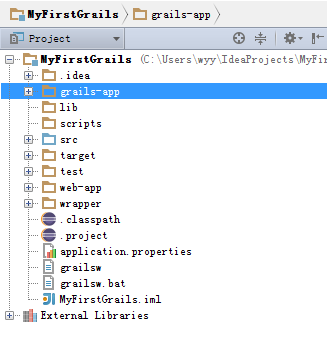
1. 输入Project Name，点击Finish



1. 创建成功后点击Run ‘create-app’。



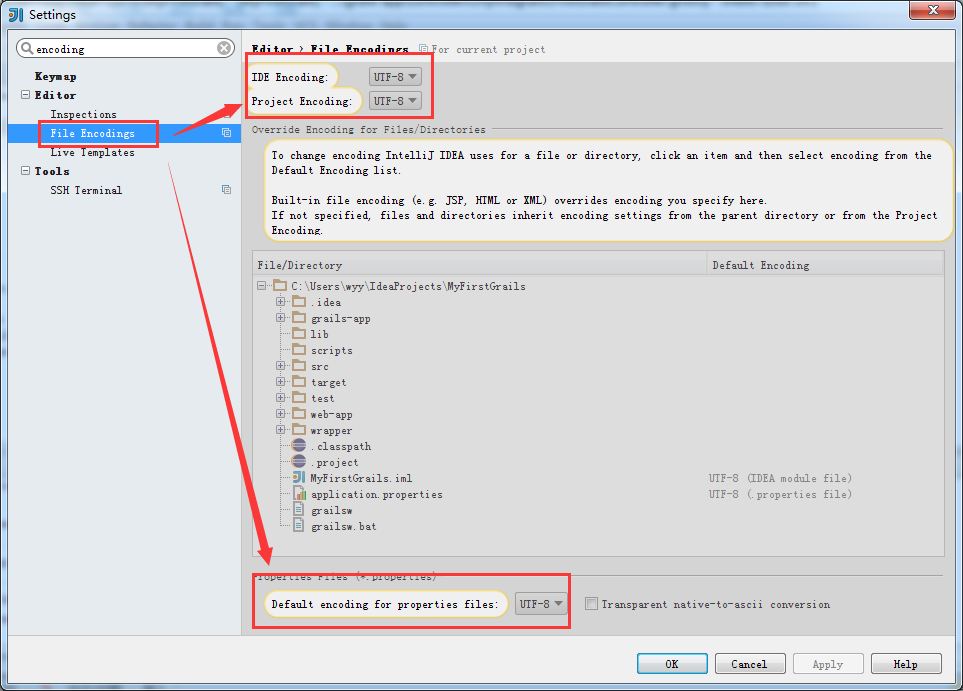
如出现下图的结构，则说明项目创建成功



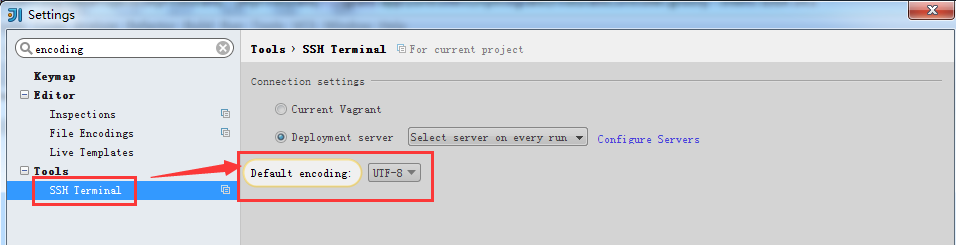
#### 配置

1. 编码配置：

点击File->Settings…在搜索框中输入encoding，选择File Encodings，设置“Project Encoding”和“Default encoding for Properties files”为“UTF-8”。



再点击选择SSH Terminal，设置“default encoding”为“UTF-8”。



#### IDEA概念介绍

下边给出Eclipse与IDEA的概念的对应关系：

|  | Eclipse |  | | IDEA |
| --- | --- | --- | --- | --- |
|  |  | | |  |
| Workspace | 工作区 | | Project | 项目 |
| Project | 项目 | | Module | 模块 |
| Project-specific JRE | 当前项目指定使用的JRE | | Module JDK | 当前模块指定使用的JDK |
| User library | 用户自定义库，所有project都可以引用 | | Grobal library | 用户自定义库，所有module都可以引用 |
| Classpath variable | classpath变量 | | Path variable | classpath变量 |
| Project dependency | 项目依赖 | | Module dependency | 模块依赖 |
| Library | 当前项目中的库 | | Module Library | 当前项目中的库 |

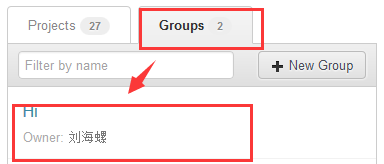
### 克隆项目

将项目代码克隆到本地，过程如下：

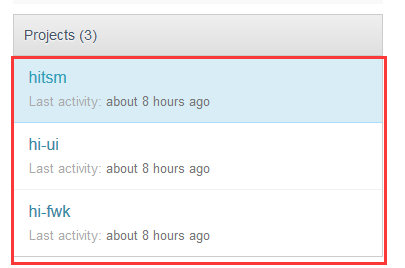
1. 进入IDEA的workspace，如：C:\IdeaProjects
2. 运行cmd并进入到C:\IdeaProjects下，输入如下命令，新建hi文件夹：

*mkdir hi;*

1. cd进入到hi文件夹内
2. 打开浏览器，输入URL <http://hismsdev.hand-china.com/gitlab/users/sign_in>，输入用户名密码并登录。
3. 进入网站后，在右侧找到Group并点击。然后点击下面的HismsAdf如下图：



1. 进入后，在右边的Project栏中找到Hitsm、hi-ui、hi-fwk，准备Clone。如下图：



1. 分别点击hitsm、hi-ui、hi-fwk
2. 在网页中找到地址栏，并点击选择HTTP



1. 在Git Shell中，输入如下命令克隆项目（注意确保当前目录为hi），如下为克隆hi-ui项目：

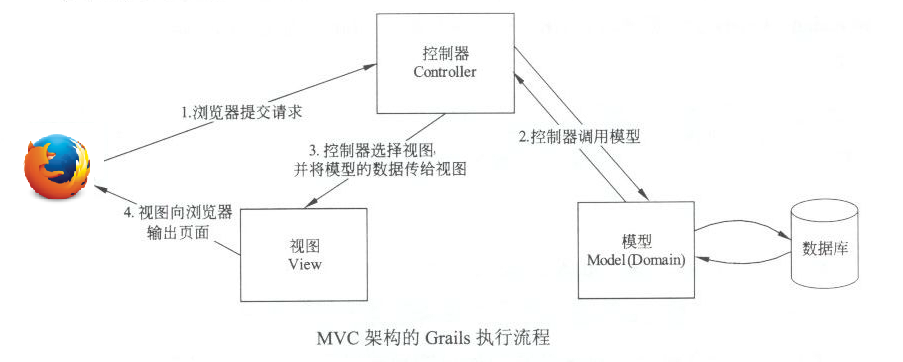
*git clone http://hismsdev.hand-china.com/gitlab/hi/hi-ui.git*

## Grails知识点

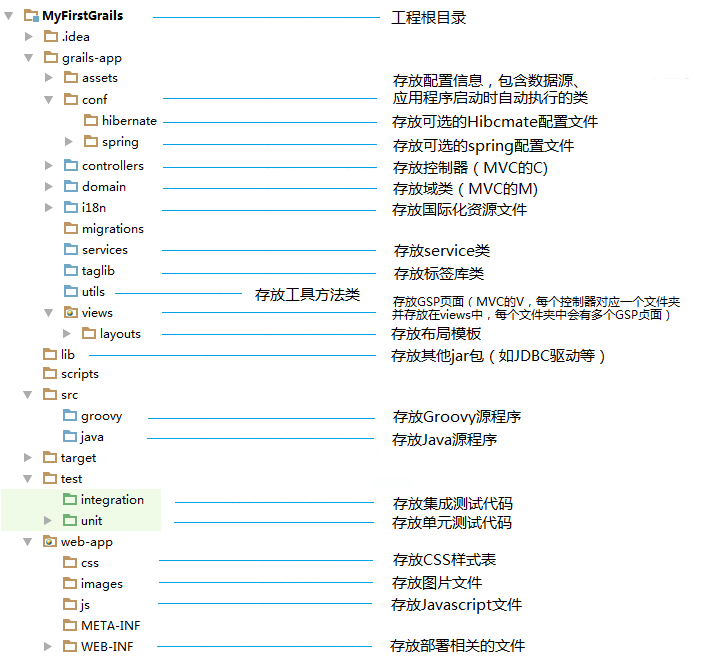
本章介绍Grails一些相关的知识点。

### 综述

#### Grails整体执行流程



#### Grails生成的项目目录结构及说明



#### 配置MySQL数据库

1. 在项目的lib文件夹中加入连库jar包。（如：mysql-connector-java-5.1.30-bin.jar）
2. 接下来要做的是配置Grails数据源使用的驱动程式的文件中定义的设置，基本上需要以下信息：driver class 、URL、usernam、password；数据源配置，在DataSource.groovy文件中简单的配置如下：

dataSource {

pooled = true

driverClassName = "com.mysql.jdbc.Driver" //链接驱动

username = "root" //账号

password = "root" //密码

loggingSql = true //输出sql日志

}

**DataSource.groovy标准属性有：**

1. driverClassName - JDBC驱动程序的类名
2. username -用户名用于建立一个JDBC连接
3. password -用于建立一个JDBC连接的密码
4. url -数据库的JDBC URL
5. dbCreate -是否根据领域类自动生成数据库-可以是'create-drop'、'create'、'update'或者'validate'
6. pooled -是否使用连接池（默认为true）
7. logSql -是否将SQL输出到字符终端
8. formatSql -格式化SQL
9. dialect - Hibernate用于跟数据库通讯的方言（dialect），可以是字符串或者类名。可以通过org.hibernate.dialect来查看所支撑的方言。

**其中：通过dbCreate属性来进行一些控制，其可选值如下:**

create -在启动时候，先删除已存在的，包括表、索引等，然后创建。

create-drop - 同create，不过在应用关闭的时候，也进行表删除。

update - 创建不存在的表和索引，并且在不删除表和数据的情况下更新表结构。注意此种情况于很多限制，比如你不能很好地处理重命名字段（旧有地字段依然保留）

validate - 不改变你数据库地任何信息，只是跟现有地数据库配置脚本进行比较，并且报告一个警告。

除了这里描述的标准属性，附加的驱动特性可以通过定义属性块作为数据源配置部分配置，例如：

environments{

production{

dataSource{

dbCreate="update"

url="jdbc:h2:prodDb;MVCC=TRUE;LOCK\_TIMEOUT=10000"

pooled=true

properties{

maxActive=-1

minEvictableIdleTimeMillis=1800000

timeBetweenEvictionRunsMillis=1800000

numTestsPerEvictionRun=3

testOnBorrow=true

testWhileIdle=true

testOnReturn=true

validationQuery="SELECT1"

}

}

}

}

### 域类

#### 瞬态特性的理解

salaryPaidYTD定义成瞬时变量，其值将不存入到数据库中：

class Company{

String name

Integer numberOfEmployees

BigDecimal salaryPaidYTD

static transients = ['salaryPaidYTD']

}

下面的例子中的get方法，虽然NetWorth没有定义，默认情况下Grails将通过创建相应的列在数据库中。但你可以用瞬变特性告诉Grails这个属性应该怎样处理：

class Company{

BigDecimal cash

BigDecimal receivables

BigDecimal capital

BigDecimal getNetWorth(){

cash + receivables + capital

}

static transients=['netWorth']

}

#### 派生属性

也是没有存储在数据库中的列，但它告诉数据库应该查询那些值来获得派生属性值，如下netWorth就是一个派生属性：

class Company{

BigDecimal cash

BigDecimal receivables

BigDecimal capital

BigDecimal netWorth

static mapping={

netWorth formula : 'CASH + RECEIVABLES + CAPITAL '

}

}数据库映射

如下例所示，定义了一个Person类，以及firstName、lastName、age三个属性：

class Person{

String firstName

String lastName

Integer age

static mapping={

id column:'person\_id'

firstName column:'person\_first\_name'

lastName column:'person\_last\_name'

age column:'person\_age'

version false

}

}

例子如果不定义mapping，数据库表结构会是这样子的：

计算机生成了可选文字: +++++++
1FieldlT）甲e1Null1Key1Default1Extral
+++++++
1id1bigint(20)1No1PRI1NU匕1auto--incrementl
Iversionlbigint(20)INO1INULL11
1age1int(11)1NO11NllLL11
I石rst_n袱1varchar(2551NO11NULL11
1last_n祝1varchar(255)1NO11NULL11
+++++++

使用了mapping是这样子的：

计算机生成了可选文字: +++++++
1Field11冲e1Null1KeyIDefaultIE丈tral
+++++++
1person_id1bigint(20)1NO1PRI1NULL1auto_incrementl
1perSOn_age1int(11)1NO11NULL11
1person少st_na爬Ivarchar(255)INO1INll比11
1person_bst_name1varchar(255)1NO11NllLL11
+++++++

#### GORM中的关联关系

1. 一对一的关系

class Car{

Engine engine

}

class Engine{

Car car

}

一个发动机属于一个汽车，但汽车不属于这个发动机（在有belongsTo的情况为级联删除）

class Car{

Engine engine

}

class Engine{

static belongsTo = [car:Car]

}

class Car{

static hasOne = [engine:Engine] //一对一管理

}

在上面例子中我们使用了belongsTo设置来说明Engine属于Car，即保存和删除将会从Car级联到关联的Engine

**默认的级联行为是级联保存和更新，但不删除，除非belongsTo被设置**

1. 一对多关系

对于双向一对多，你可以通过改变many端列名来更改外键列。然而，对于单项关联，需要在关联端指定。

class Artist{

String name

static hasMany = [albums:Album]

}

class Album{

String title

static hasMany = [songs:Song]

static belongsTo =[artist:Artist]

}

class Song{

String itle

Integer duration

static belongsTo = Album

}

1. 多对多关系

Grails支持许多对多关系，通过在两端都定义hasMany上的关系，并且在某一端使用belongsTo：

class Book{

static belongsTo = Author

static hasMany =[authors:Author]

String title

}

class Author{

static hasMany = [books:Book]

String name

}

1. 继承关系

这有一个缺点，就是在数据库级别上不能有NOT-NULL约束。

class Person{

String firstName

String lastName

Integer age

}

class Employee extends Person{

String employeeNumber

String companyName

}

class Player extends Person{

String teamName

}

#### 嵌入对象

下面以车和发动机的关系为例：

class Car {

String make

String model

Engine engine

}

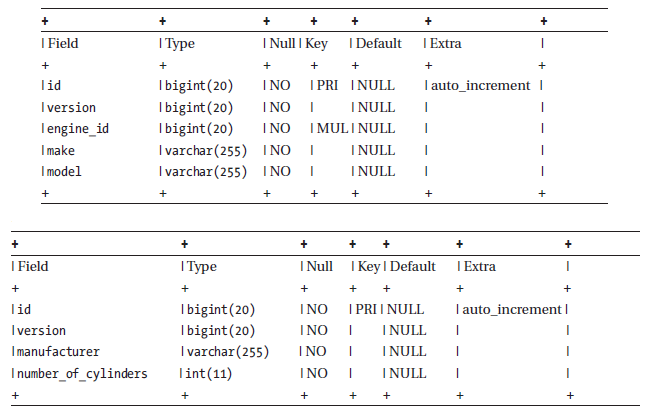
class Engine {

String manufacturer

Integer numberOfCylinders

}

通常情况下，car需要一个外键，将Engine指向他：



通过对象嵌入：

class Car {

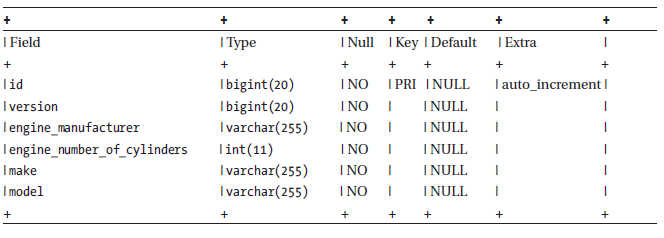
String make

String model

Engine engine

static embedded = ['engine']

}



### 控制器

Grails控制器是负责处理请求来在应用程序类。控制器接收到一个请求，有可能做一些工作的要求，并最终决定了未来会发生什么、接下来发生的事情，可能包括以下内容：

•执行另一个控制器的动作（可能但不一定在同一控制器）

•渲染一个视图

•渲染信息直接发送到响应

#### 控制器的定义

控制器是一个Grails app/controllers目录下定义的类。类名必须以controller结束。你不一定需要指定URL中执行的动作。如果没有具体指定动作，Grails将在指定的控制器执行默认动作。你可以使用下面的规则确定的默认动作：

•如果控制器只定义一个Action，它成为默认动作。

•如果控制器定义了一个Action，称为index，它成为默认动作。

•如果控制器定义了一个属性叫做DefaultAction，其value是默认动作的名称。

示例：

// Here the 'list' action is the default as there is only one action defined

class SampleController {

def list() {}

}

// In this example 'index' is the default by convention

class SampleController {

def list() {}

def index() {}

}

// Here 'list' is explicitly set as the default

class SampleController {

static defaultAction = 'list'

def list() {}

def index() {}

}

#### 日志

示例：

class SampleController {

def index() {

try {

// do something that might throw an exception

} catch (Exception e) {

log.error ('some message goes here', e)

}

}

}

根据org.apache.commons.logging.log API文档可以调用的方法：

public interface Log {

public void debug(Object msg);

public void debug(Object msg, Throwable t);

public void error(Object msg);

public void error(Object msg, Throwable t);

public void fatal(Object msg);

public void fatal(Object msg, Throwable t);

public void info(Object msg);

public void info(Object msg, Throwable t);

public void trace(Object msg);

public void trace(Object msg, Throwable t);

public void warn(Object msg);

public void warn(Object msg, Throwable t);

public boolean isDebugEnabled();

public boolean isErrorEnabled();

public boolean isFatalEnabled();

public boolean isInfoEnabled();

public boolean isTraceEnabled();

public boolean isWarnEnabled();

}

#### 访问请求属性

访问请求属性：

| 属性 | 描述 |
| --- | --- |
|  |  |
| actionName | 当前执行的action的名称 |
| actionUri | 当前执行的action的Uri |
| controllerName | 当前的控制器名称 |
| controllerUri | 当前控制器的uri |
| flash | flash作用域 |
| log | 打印日志 |
| params | 获取参数 |
| request | The HttpServletRequest object |
| response | The HttpServletResponse object |
| session | The HttpSession object |
| servletContext | The ServletContext object |

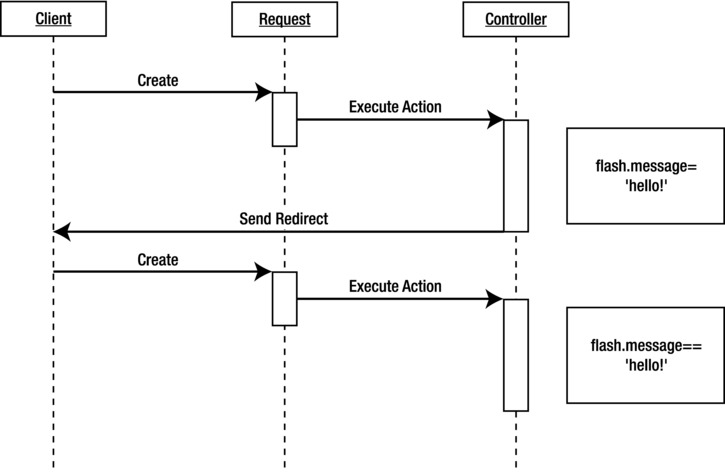
Java servlet请求属性和Grails控制器之间的差异：

| Java | Servlet Grails Controller |
| --- | --- |
|  |  |
| request.getAttribute(“myAttr”); | request.myAttr |
| request.setAttribute(“myAttr”, “myValue”); | request.myAttr = “myValue” |
| session.getAttribute(“mAttr”); | session.myAttr |
| session.setAttribute(“myAttr”, “myValue””); | session.myAttr = “myValue” |
| servletContext.getAttribute(“mAttr”); | servletContext.myAttr |
| servletContext.setAttribute(“myAttr”,“myValue””); | servletContext.myAttr = “myValue” |

#### 控制器的作用域

request，flash，session，servletContext四种作用域。其中flash：放入Flash对象仅在本次请求和下次请求中保持。

例如:当客户端创建一个请求，执行action，在controller中重定向后当前的request.message = "hello!"在下一次调用时会变为request.message == null，若使用flash，在重定向后仍能获取到该值：



#### 参数

1. 访问请求参数，如：

def userName = request.getParameter('userName')

log.info("User Name: ${userName}")

1. 请求参数类型转换

def counter = params.counter.toInteger()

或

def counter = params.int('counter')

包括了(Boolean, byte, char, short, int, long, float, and double)八种类型的转换。

另一种类型的转换方法是命名列表。该列表的方法是有用的在处理多个同名的请求参数。方法将返回一个列表的列表包含所有与指定的请求参数名称关联的值。

def index() {

for (name in params.list('name') {

// do something with name

}

}

1. 默认情况下，方法的参数名称对应一个请求参数的名称。如果由于某种原因，请求参数的方法参数需要有不同的名称，那么Grails.web.requestparameter注释可以用来明确的请求参数绑定到一个特定的参数：

import grails.web.RequestParameter

class AdminController {

// mainNumber will be initialized with the value

// of params.accountNumber

// accountType will be initialized with params.int('accountType')

def action(@RequestParameter('accountNumber') String mainNumber, int accountType) {

// …

}

}

#### 请求重定向

使用redirect控制方法可以Action重定向：

class OverviewController {

def login() {}

def find() {

if (!session.user)

redirect(action: 'login')

return

}

…

}

本质上，redirect方法使用的是HttpServletResponse对象的sendRedirect方法。

该重定向方法达到以下预期：

在同一个控制器类中的另一个闭包（action）： redirect(action: login)。一个Action名称（如不是当前控制器的Action那么需要控制器名称）： redirect(controller: 'home', action: 'index')。一个资源相对应用程序上下文路径的URI：redirect(uri: "/login.html")或一个完整的URL：redirect(url: "http://grails.org")。使用方法的params参量，可传递参数从一个动作（Action）到下一个动作（Action）：

redirect(action: 'myaction', params: [myparam: "myvalue"])

上面的参数通过params动态属性变得可用，params动态属性可以访问请求（request）参数。如果果指定参数与请求参数具有相同名称，请求参数被覆盖并使用控制器参数。既然params对象是一个映射，你可以用它来传递当前请求参数从一个action到下一个action： redirect(action: "next", params: params)。

最后，您还可以在目标URI中包含片段： redirect(controller: "test", action: "show", fragment: "profile")这将（依赖URL映射）重定向到类似“/myapp/test/show#profile”。

#### 模型和视图

模型是视图渲染时使用的一个模型映射。该映射的键符合视图中可访问的变量名称。map内的键对应的视图访问的变量名。

如：

def show() {

[book: Book.get(params.id)]

}

Grails会在本地寻找grails-app/views/book/show.gsp为这个show操作，为了呈现不同的视图，可以使用render方法：

def show() {

def map = [book: Book.get(params.id)]

render(view: "display", model: map)

}

在这种情况下，Grails将试图呈现grails-app/views/book/display.gsp视图。请注意，Grails自动在grails-app/views目录下的book目录找出该视图。这是很方便的，但如果你访问共享视图，需要你使用绝对路径，而不是相对路径。

def show() {

def map = [book: Book.get(params.id)]

render(view: "/shared/display", model: map)

}

#### 数据绑定

数据绑定是将传入的请求参数绑定到一个对象属性或对象实体视图上的行为。数据绑定将对所有来自请求参数进行必要的类型转换，它们通常通过表单提交来实现，并总是String类型，可是，Groovy 或者Java对象的属性很可能不是String类型，所以需要转换。

通常情况下做保存操作:

def save() {

def album = new Album()

album.genre = params.genre

album.title = params.title

album.save()

}

通过映射到一个域类的构造函数，用相应的请求参数初始化所有的域类的属性,这种方法更好：

def save() {

def album = new Album(params)

album.save()

}

同样在做更新操作：

def update() {

def album = Album.get(params.id)

album.properties = params

album.save()

}

验证操作：

def save() {

def album = Album.get(params.id)

album.properties = params

if(album.save()) {

redirect action: "show", id: album.id

} else {

render view: "edit", model: [album:album]

}

}

通过调用save()方法，从而引发验证，如果发生验证错误将返回到用户视图上。当包含一个或多个错误时，可以迭代显示它们：

album.errors.allErrors.each { println it.code }

如果只想检查是否已经有一个实例的任何错误，可以调用该实例的haserrors()方法

if(album.hasErrors()) println "Something went wrong!"

在视图中，您可以将这些使用<g:renderErrors>标签:

<g:renderErrors bean="${album}" />

**绑定多个域类**

比如有一个传入请求：/book/save?book.title=The%20Stand&author.name=Stephen%20King

Controller 获取

def b = new Book(params.book)

def a = new Author(params.author)

#### 命令对象

一个命令对象是一个用来连接数据绑定的类。通常对数据进行验证，这个数据可能不适合现有的域类。

1. 声明命令对象，如：

class AlbumCreateCommand {

String artist

String title

List songs = []

List durations = []

static constraints = {

artist blank:false

title blank:false

songs minSize:1, validator:{ val, obj ->

if(val.size() != obj.durations.size())

return "songs.durations.not.equal.size"

}

}

Album createAlbum() {

def artist = Artist.findByName(artist) ?: new Artist(name:artist)

def album = new Album(title:title)

songs.eachWithIndex { songTitle, i ->

album.addToSongs(title:songTitle, duration:durations[i])

}

return album

}

}

1. 使用命令对象 例：

def save(AlbumCreateCommand cmd) {

if(cmd.validate()) {

def album = cmd.createAlbum()

album.save()

redirect(action:"show", id:album.id)

}

else {

render(view:"create", model:[cmd:cmd])

}

}

页面获取错误信息：

<g:renderErrors bean="{cmd}" />

#### HTTP方法约束

一般指的是get、put、post等方法的约束，下面以get为例：

class SongController {

def delete() {

if(request.method == "GET") {

// do not delete in response to a GET request

// redirect to the list action

redirect action: "list"

} else {

// carry out the delete here...

}

}

}

#### 文件上传

1. 编程式文件上传

Grails支持使用Spring文件上传MultipartHttpServletRequest接口。对文件上传的第一步是创建一个这样的混合表单：

Upload Form: <br />

<g:uploadForm action="upload">

<input type="file" name="myFile" />

<input type="submit" />

</g:uploadForm>

该uploadForm标签方便地给<g:form>标签添加enctype="multipart/form-data"属性。有那么多的方式来处理文件上传。一个是直接使用Spring MultipartFile实例：

def upload() {

def f = request.getFile('myFile')

if (f.empty) {

flash.message = 'file cannot be empty'

render(view: 'uploadForm')

return

}

f.transferTo(new File('/some/local/dir/myfile.txt'))

response.sendError(200, 'Done')

}

当你使用MultipartFile接口获得一个InputStream，这可以方便地转移到其他目的地并直接操作文件等等。

1. 通过数据绑定上传文件

文件上传也可以使用数据绑定执行。考虑这个图片的域类：

class Image {

byte[] myFile

static constraints = {

// Limit upload file size to 2MB

myFile maxSize: 1024 \* 1024 \* 2

}

}

如果像下面的例子那样，您在构造函数中使用params对象创建一个图像，Grails将自动将文件的内容作为字节绑定到MyFile属性：

def img = new Image(params)

很重要的一点是你设置的size或maxsize限制，否则你的数据库可能因为列太小而不能保存文件。例如，无论是H2和MySQL的默认的BLOB属性的大小是255字节。另外，也可以通过改变myFile属性的类型来设置文件内容作为一个字符串，即将image变为String类型：

class Image {

String myFile

}

#### 拦截器

通常，它非常有助于基于request、session或者application状态的拦截过程。 这个可以通过动作拦截器来完成。当前有两种类型的拦截器:before和after。

1. 拦截前（beforeInterceptor）

该beforeInterceptor在执行Action之前进行拦截处理。如果返回false则拦截的动作将不会被执行。拦截器可在控制器中为所有Action定义，如：

def beforeInterceptor = [action: this.&auth, except: 'login']

private auth() {

if (!session.user) {

redirect(action: 'login')

return false

}

}

1. 拦截后（afterInterceptor）

使用afterInterceptor属性来定义一个在动作之后执行的拦截：

def afterInterceptor = { model, modelAndView ->

println "Current view is ${modelAndView.viewName}"

if (model.someVar) modelAndView.viewName = "/mycontroller/someotherview"

println "View is now ${modelAndView.viewName}"

}

### 视图

#### 基础知识

GSP中的一些属性：

| 属性 | 描述 |
| --- | --- |
|  |  |
| application | javax.servlet.ServletContext 实例 |
| applicationContext | Spring ApplicationContext 实例 |
| flash | flash 对象 |
| grailsApplication | GrailsApplication 实例 |
| out | 响应输出流 |
| params | params 对象用于检索请求参数 |
| request | HttpServletRequest 实例 |
| response | HttpServletResponse 实例 |
| session | HttpSession 实例 |
| webRequest | GrailsWebRequest 实例 |

#### 页面指令

它出现在页面顶端，可以设置content类型、引包、设置页面属性甚至是一个特定容器，一般是以<%@开头的，如：

<%@ page contentType="text/xml; charset=UTF-8" %>

<%@ page import="java.sql.Time" %>

#### 表达式

支持传统的JSP脚本块使用<% %>语法，如：

<html>

<body>

<% 3.times { %>

<p>I'm printed three times!</p>

<% } %>

</body>

</html>

在JSP中c:out标签：<p><c:out value="${album.title}" /></p>，在GSP中则可以使用Gstrings：<p>${album.title}</p>

#### 标签

GSP执行基本的操作有一些内置的标签,他们包括循环、分支、逻辑判断语句。

注：GSP还支持JSP自定义标签库，可以与标准的JSP标签库指令输入。

打开命令行窗口。输入java –version，如果可以查看到java version符合安装版本，则说明已正确配置。

1. Set标签：

<g:set>标签有两个属性：var属性和value属性，如：

<g:set var="albumTitle" value="${album.title}" />

默认情况下，<g:set>被认为是内页范围。也可以通过scope属性设定一个session范围：

<g:set scope="session" var="user" value="${user}" />

1. 逻辑标签：

<g:if>, <g:elseif>和<g:else>标签，与常规编程结构相同：

<g:if test="${album?.year < 1980 && album?.genre == 'Rock'}">

Classic rock

</g:if>

<g:elseif test="${album?.year >= 1980 && album?.genre == 'Rock'}">

Modern Rock

</g:elseif>

<g:else>

Other

</g:else>

因为某些原因要大写，你将使用一个表达式如下：

${album.title.toUpperCase()}，如果album或title空的话，会抛出NullPointerException，为了规避这种情况，使用?.来处理：

${album?.title?.toUpperCase()}

1. 迭代标签

第一个标签是<g:each>标签

<g:each in="${album.songs?}">

<span class="tag">${it.title}</span>

</g:each>

也可以定义var属性：

<g:each var="song" in="${album.songs?}">

<span class="song">${song.title}</span>

</g:each>

下一个迭代标签GSP提供的是<g:while> 标签，注意死循环。

<g:set var="i" expr="${album.songs?.size()}" />

<g:while test="${i > 0}">

<g:set var="i" expr="${i-1}" />

</g:while>

1. 过滤标签：collect, findAll和 grep
2. Collect标签

<g:collect> 标签允许你遍历对象集合的属性：

<ol>

<g:collect in="${albums}" expr="${it.title}">

<li>${it}</li>

</g:collect>

</ol>

上例的代码相当于scriptlet代码：

<ol>

<% albums.collect{ it.title }.each { %>

<li>${it}</li>

<%}%>

</ol>

1. findAll标签

<g:findAll in="${albums}" expr="${it.songs?.title.contains('Love')}">

<li>${it.title}</li>

</g:findAll>

1. grep标签

例如通过类过滤：

<g:grep in="${books}" filter="NonFictionBooks.class">

<p>Title: ${it.title}</p>

</g:grep>

或使用正则表达式：

<g:grep in="${books.title}" filter="~/.\*?Groovy.\*?/">

<p>Title: ${it}</p>

</g:grep>

上面的例子也很有趣，因为它使用GPath。GPath是在Groovy中 的XPath语言。

1. 链接标签

<g:link>标签可以创建一个简单的HTML锚标记，在其基础上提供的属性，其中包括以下内容：

•controller：连接到控制器的名称

•action：链接到动作的名称

•id：定义在URI的最后

• mapping：使用的URL映射名称

•params：任何参数传递以map形式

基本链接，如：

<g:link controller="album" action="list">list Albums</g:link>

<g:link action="show" id="l">Show album with id K/g:link>

带参数链接：

<g:link controller="album" action="list"

params="[max:10,order:'title']">Show first ten ordered by Title</g:link>

<g:link action="create"

params="${params}">Pass parameters from this action to next</g:link>

1. Form标签

默认情况下，<g:form>标签使用的表单提交POST方法：

<form action="/gTunes/user/register" method="POST" name="registerForm">

...

</form>

也可以定义<g:form>标签使用一个URL属性：使用key:value的map形式：

<g:form url="[controller:'user', action:'register']">

...

</g:form>

1. textField标签

<g:textField name="login" value="${user?.login}"></g:textField>

1. 复选框和单选按钮

checkBox：

<g:checkBox name="aBooleanValue" value="${true}" />

radio：

<p><g:radio name="myGroup" value="1" checked="${someValue == 1}" />Radio 1 </p>

<p><g:radio name="myGroup" value="2" checked="${someValue == 2}" /> Radio 2 </p>

1. select标签

<g:select name="genre"  
 from="${['Rock', 'Blues', 'Jazz']}"

value="${album.genre}" />

<select name="genre">

<option value="Rock" selected="selected">Rock</option>

<option value="Blues">Blues</option>

<option value="Jazz">Jazz</option>

</select>

1. 日期标签

使用<g:datePicker>标签（自动执行类型转换到日期）

<g:datePicker name="myDate" value="${new Date()}" />

precision来确定精度，下面的例子只渲染了年，月，日

<g:datePicker name="myDate" value="${new Date()}" precision="day" />

dateFormat格式标签：

def dateFormat = { attrs, body ->

out << new java.text.SimpleDateFormat(attrs.format).format(attrs.date)

}

以上使用Java的SimpleDateFormat类来格式化日期，然后把它写入响应。标签可以在GSP中使用如下：

<g:dateFormat format="dd-MM-yyyy" date="${new Date()}" />

1. haserrors标签

Grails提供 <g:hasErrors>标签，支持以下属性：

•bean：一个bean实例来检查错误

•field：字段的名称来检查错误

•model：一种替代指定bean；整个模型（map）可以检查

<g:hasErrors>标签通常与<g:hasErrors>结合使用

1. eacherror标签

<g:hasErrors bean="${album}">

<ul class="errors">

<g:eachError bean="${album}">

<li>${it.defaultMessage}</li>

</g:eachError>

</ul>

</g:hasErrors>

#### 视图和模板

Grails 使用在一个视图名字前放置一个下划线来标识为一个模板的规约

例如：grails-app/views/book 视图中的一个模板，你可以使用 render标签：

<g:render template="bookTemplate" model="[book:myBook]" />

在控制器中返回一个模板

def show = {

de b = Book.get(params.id)

render(template:"bookTemplate", model:[book:b])

}

#### 标签库

创建一个以规约 TagLib 结尾的一个 Groovy 类，并把它放置于 grails-app/taglib 目录里。

1> 简单标签

例： 日期格式化标签

def dateFormat = { attrs, body ->

out << new java.text.SimpleDateFormat(attrs.format).format(attrs.date)

}

标签能像下列这样在 GSP 中使用：

<g:dateFormat format="dd-MM-yyyy" date="${new Date()}" />

2> 逻辑标签

例：标签检查用户是否为管理人员

def isAdmin = { attrs, body ->

def user = attrs['user']

if(user != null && checkUserPrivs(user)) {

out << body()

}

}

页面使用标签：

<g:isAdmin user="${myUser}">

// some restricted content

</g:isAdmin>

3> 迭代标签

例：我们检查一个 times 属性，假如存 在，把它转换为一个数字，然后使用 Groovy 的 times 方法

def repeat = { attrs, body ->

attrs.times?.toInteger().times { num ->

out << body(num)

}

}

页面使用标签：

<g:repeat times="3">

<p>Repeat this 3 times! Current repeat = ${it}</p>

</g:repeat>

4> 自定义标签

class GtunesTagLib {

static namespace = 'gt'

def repeat = { attrs, body ->

// retrieve the 'times' attribute and convert it to an int

int n = attrs.int('times')

// render the body n times, passing a 1 based

// counter into the body each time

n.times { counter ->

out << body(counter + 1)

}

}

}

页面使用标签：

<gt:repeat times="3">

Hello number ${it}

</gt:repeat>

页面不再使用 <g:repeat/>，而应该使用新定义的：<gt:repeat/>

### URL映射

#### 默认URL映射

默认URL映射配置在Grails应用程序是简单的,它包括两部分，第一部分的URL对应的控制器的名称，第二部分可选部分，$action?和$id是可选部分，可定义多个，也可以不要，如：

class UrlMappings {

static mappings = {

"/$controller/$action?/$id?"{

constraints {

// apply constraints here

}

}

"500"(view:'/error')

}

}

#### 包含静态文本的URL映射

静态文本必须为一个特定的映射应用为URL的一部分

class UrlMappings {

static mappings = {

"/showAlbum/$controller/$action?/$id?" {

constraints {

// apply constraints here

}

}

}

}

#### 不包含控制器和动作的URL映射

class UrlMappings {

static mappings = {

"/showAlbum/$id" {

controller = 'album'

action = 'show'

}

// ...

}

}

或者使用

class UrlMappings {

static mappings = {

"/showAlbum/$id"(controller:'album', action:'show')

// ...

}

}

#### 嵌入参数的URL映射

如下，我们要查询一个artistName为Rush的连接

class UrlMappings {

static mappings = {

"/showArtist/$artistName"(controller:'artist', action:'show')

// ...

}

}

那么我们在控制器中应该这样定义

class ArtistController {

def show() {

def artist = Artist.findByName(params.artistName)

// do whatever is appropriate with the artist...

}

}

#### 指定参数的URL映射

除了在URL中嵌入参数，任意的请求参数可以指定为一个特定的映射，不显示在URL属性。

class UrlMappings {

static mappings = {

"/showArtist/$artistName"(controller:'artist', action:'show') {

format = 'simple'

}

"/showArtistDetail/$artistName"(controller:'artist', action:'show') {

format = 'detailed'

}

// ...

}

}

#### 映射到视图

如下例，如果想映射根 URL /到一个位于 grails-app/views/index.gsp 的GSP

class UrlMappings {

static mappings = {

"/"(view:'/index')

// ...

}

}

具体给定的控制器（Controller）中的一个视图，可以这样使用

static mappings = {

"/find"(view:'query', controller:'search')

}

#### 带约束的URL映射

例如一个博客写的Grails应用。一个典型的博客系统的URL格式可能像/grailsblogs/2009/01/15/new\_grails\_release，如下：

class UrlMappings {

static mappings = {

"/grailsblogs/$year/$month/$day/$entry\_name?" {

controller = 'blog'

action = 'display'

constraints {

year matches: /[0-9]{4}/

month matches: /[0-9]{2}/

day matches: /[0-9]{2}/

}

}

// ...

}

}

**映射通配符**

1. 匹配所有 images 路径下像/image/logo.jpg 这样的 jpg

static mappings = {

"/images/\*.jpg"(controllers:"image")

}

也可以通过一个变量来达到同样的效果

static mappings = {

"/images/$name.jpg"(controllers:"image")

}

2. 使用双通配符来匹配多于一个层次之外的

static mappings = {

"/images/\*\*.jpg"(controllers:"image")

}

这个映射不但匹配/image/logo.jpg 而且匹配/image/other/logo.jpg

3．用一个双通配符变量

static mappings = {

// will match /image/logo.jpg and /image/other/logo.jpg

"/images/$name\*\*.jpg"(controllers:"image")

}

#### 映射到HTTP

URL映射也可以配置到基于HTTP方法的映射（GET，POST，PUT和DELETE）。这是RESTful API非常有用的，并且基于HTTP方法限制映射，如：

class UrlMappings {

static mappings = {

"/artist/$artistName" {

controller = 'artist'

action = [GET: 'show',

PUT: 'update',

POST: 'save',

DELETE: 'delete']

}

// ...

}

}

**映射到响应代码**

Grails 同样允许映射到一个 HTTP 响应代码到控制器，操作或视图，例：

static mappings = {

"500"(controller:"errors", action:"serverError")

"404"(controller:"errors", action:"notFound")

"403"(controller:"errors", action:"forbidden")

}

或者只想提供一个错误的页面

static mappings = {

"500"(view:"/errors/serverError")

"404"(view:"/errors/notFound")

"403"(view:"/errors/forbidden")

}

#### 反向映射

你可以链接任何使用grails提供的g:link标签创建的url映射，只需要简单引用要链接的控制器和动作：

<td>

<g:link action='show' controller='artist'id="${artist.id}">

${artist.name}</g:link>

</td>

在UrlMappings中，只要告诉链接标签到哪个控制器和动作，以及要提供的参数，来调用自定义映射。

class UrlMappings {

static mappings = {

"/showArtist/$artistName"(controller:'artist', action:'show')

// ...

}

}

#### 指定名称的URL映射

有很多情况下，多个URL映射匹配到给定的请求，这时候需要用Named URL Mapping，如:

class UrlMappings {

static mappings = {

name artistDetails: "/showArtist/$artistName" {

controller = "artist"

action = "show"

}

}

}

链接标签支持一个可选的属性名映射，可用于指定用于这种查询映射的名称:

<g:link mapping="artistDetails"

params="[artistName:${artist.name.replaceAll(' ', '\_')}">

${artist.name}

</g:link>

或者

<link:artistDetails artistName="${artist.name.replaceAll(' ', '\_')}">

${artist.name}

</link:artistDetails>

#### URL映射中重定向

Grails2.3开始，可以定义用于指定重定向的URL映射。当一个URL映射指定了重定向，任何时候映射匹配传入的请求，带有映射提供的信息来启动重定向与。

当一个URL映射指定重定向，映射必须要么提供一个字符串描述的重定向，要么提供一个映射来描述要重定向的目标。映射被结构化，就像在控制器中映射作为参数传递给redirect方法的映射一样。

"/viewBooks"(redirect: '/books/list')

"/viewAuthors"(redirect: [controller: 'author', action: 'list'])

"/viewPublishers"(redirect: [controller: 'publisher', action: 'list', permanent:true])

请求参数将被包括在重定向里（这个请求参数是原始请求的一部分）。

### 国际化

#### 定位信息

通过利用底层的Spring MVC国际化支持，Grails支持开箱即用国际化（i18n）使用。通过Grails，可以自定义显示在视图中的文本。它使用Locale来为用户裁剪信息。Locale是由语言代码 国家代码组成。例如，"en\_US"是美国英语的语言码，而"en\_GB"代表英国英语。

1. 定义用户信息

在项目的grails-app/i18n目录中，定义一个纯文本属性，如：

# messages.properties

app.name=gTunes

book.title=The Definitive Guide To Grails

favorite.language=Groovy

favorite.framework=Grails

1. 检索消息

在一个标准的Java和Groovy程序，可以使用java.util.resourcebundle类从属性文件中检索值，java.util.resourcebundle类负责加载属性文件并提供一个API来检索文件中定义的属性的值。Grails提供了一个称为消息，GSP标签将会在grails-app/i18n目录信息的文件检索属性值。

// JavaMessages.java

import java.util.ResourceBundle;

public class JavaMessages {

public static void main(String[] args) {

String appName = bundle.getString("app.name");

ResourceBundle bundle = ResourceBundle.getBundle("messages");

System.out.println("application name is " + appName);

}

}

// GroovyMessages.groovy

def messages = ResourceBundle.getBundle('messages')

def appName = messages.getString('app.name')

println "application name is ${appName}

"

在GSP这样使用

<body>

...

<g:message code="gtunes.welcome"/>

...

</body>

#### URL映射的国际化

如一个请求 http://localhost:8080/gTunes/?lang=es，通过自定义URL映射来定义请求参数：

class UrlMappings {

static mappings = {

"/store/$lang"(controller:'store')

// ...

}

}

#### 使用参数化的消息

通过使用{0}这样的占位符来输出参数，在资源文件中定义：

gtunes.purchased.songs=You have purchased [{0}] songs.

调用资源文件，并传递参数97：

import java.util.ResourceBundle;

import java.text.MessageFormat;

public class JavaMessages {

public static void main(String[] args) {

ResourceBundle bundle = ResourceBundle.getBundle("messages");

String songsPurchased = bundle.getString("gtunes.purchased.songs");

String message = MessageFormat.format(songsPurchased, 97);

System.out.println("message: " + message);

}

}

在页面中：

<div>

<g:message code="gtunes.purchased.songs" args="[97]"/>

</div>

默认URL映射配置在Grails应用程序是简单的,它包括两部分，第一部分的URL对应的控制器的名称，第二部分可选部分，$action?和$id是可选部分，可定义多

1. 使用参数化的信息进行验证

如：定义了一个有约束的域类

class Person {

String firstName

String lastName

Integer age

static constraints = {

firstName size: 2..30, blank: false

lastName size: 2..30, blank: false

age min: 0

}

}

在页面中进行验证

<h1><g:message code="default.create.label" args="[entityName]" /></h1>

<g:if test="${flash.message}">

<div class="message" role="status">${flash.message}</div>

</g:if>

<g:hasErrors bean="${personInstance}">

<ul class="errors" role="alert">

<g:eachError bean="${personInstance}" var="error">

<li <g:if test="${error in org.springframework.validation.FieldError}">data-fieldid="${error.field}"</g:if>>

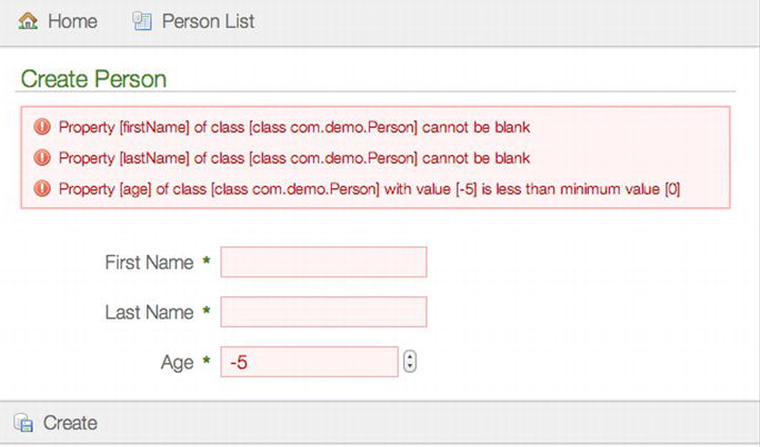
<g:message error="${error}"/></li>

</g:eachError>

</ul>

</g:hasErrors>

当输入错误时，会调用系统自带的一些验证信息：



1. 使用MessageSource

String getMessage(String code, Object[] args, Locale locale)

String getMessage(String code, Object[] args, String defaultMessage, Locale locale)

String getMessage(MessageSourceResolvable resolvable, Locale locale)

例：

package com.gtunes

class StoreService {

def messageSource

def someServiceMethod() {

def msg = messageSource.getMessage('gtunes.my.music', null, null)

// ...

}

...

}

1. 在控制器和标签库读取消息

在一个控制器读取消息是简单的，因为你可以当方法一样调用标签：

def show() {

def msg = message(code: "my.localized.content", args: ['Juan', 'lunes'])

}

同样的技术也可以用在标签库，但如果你的标签库使用自定义的命名空间，那么你必须使用前缀g.调用：

def myTag = { attrs, body ->

def msg = g.message(code: "my.localized.content", args: ['Juan', 'lunes'])

}

### Ajax

默认情况下，Grails提供了jQuery库。通过插件系统提供了诸如Prototype,GWT等其他框架的支持。也可以通过安装grails支持的其他js插件来替换jQuery。因为Grails的自适应标签库的支持这可以工作。rails插件系统对以下一些Ajax库的支持（不限于）：jQuery、Prototype、Dojo、YUI、MooTools。

#### Action中的Ajax

通过一个例子来介绍Ajax在Grails中的应用，首先在控制器中创建一个showTime，来显示当前时间：

def showTime() {

render "The time is ${new Date()}"

}

在页面中我们在<head>中加入<g:javascript library="jquery" />，这样你将告诉Grails使用jQuery的Ajax，下面为页面的主体部分：

<g:link action="showTime" elementId="timeLink">Show the time!</g:link>

<div id="time"></div>

<r:script>

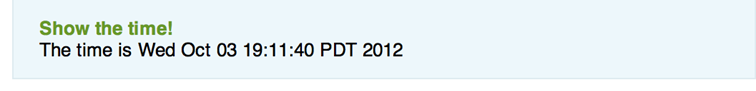
$('#timeLink').click(function() {

$('#time').load(this.href); return false;

});

</r:script>

每当你点击链接时，当前时间都会被显示在<div>中。



#### 修改Ajax支持

jQuery是Grails应用程序的默认的JavaScript库，但如果使用其他的库，可以通过下面方法来导入，首先修改buildconfig.groovy包括插件的依赖：

grails.project.dependency.resolution = {

...

plugins {

runtime ":prototype:1.0"

...

}

}

再修改<g:javascript>标签： <g:javascript library="prototype" />，这样就完成了替换Ajax的支持。

#### Ajax在服务器端

有一些不同方式实现Ajax，其通常细分为：

 内容为中心的Ajax - 只使用一个远程调用的HTML结果来更新页面

 数据为中心的Ajax – 实际上你从服务器上发送XML或JSON响应，以编程方式更新页面

 脚本为中心的Ajax - 服务器端发送的Javascript流在传送过程中被赋值

在Ajax部分的很多例子的示例涵盖了内容为中心的 Ajax，在这主要是更新页面。但你也可能需要使用数据为中心或以脚本为中心的Ajax。本指南涵盖了不同风格的Ajax。

1. 内容为中心的Ajax

内容为中心的Ajax涉及从服务器发送回来一些HTML，并且通常由调用一个render方法渲染一个模板来实现：

def showBook() {

def b = Book.get(params.id)

render(template: "bookTemplate", model: [book: b])

}

在客户端上调用这个涉及到使用remoteLink标签：

<g:remoteLink action="showBook" id="${book.id}"

update="book${book.id}">Update Book</g:remoteLink>

<div id="book${book.id}">

<!--existing book mark-up -->

</div>

1. 数据中心的Ajax与JSON

数据为中心的Ajax通常包括在客户端给响应赋值和以编程方式进行更新。对于Grails的JSON响应您通常会使用Grails的JSON marshalling（json编组）能力：

import grails.converters.JSON

def showBook() {

def b = Book.get(params.id)

render b as JSON

}

然后在客户端使用一个Ajax事件处理程序解析传入的JSON请求：

<g:javascript>

function updateBook(data) {

$("#book" + data.id + "\_title").html( data.title );

}

</g:javascript>

<g:remoteLink action="showBook" id="${book.id}" onSuccess="updateBook(data)">

Update Book

</g:remoteLink>

<g:set var="bookId">book${book.id}</g:set>

<div id="${bookId}">

<div id="${bookId}\_title">The Stand</div>

</div>

1. 数据中心的Ajax与XML

在服务器端使用XML也同样简单：

import grails.converters.XML

def showBook() {

def b = Book.get(params.id)

render b as XML

}

然而，由于DOM是复杂的，客户端变得更复杂：

<g:javascript>

function updateBook(data) {

var id = $(data).find("book").attr("id");

$("#book" + id + "\_title").html( $(data).find("title").text() );

}

</g:javascript>

<g:remoteLink action="showBook" id="${book.id}" onSuccess="updateBook(data)">

Update Book

</g:remoteLink>

<g:set var="bookId">book${book.id}</g:set>

<div id="${bookId}">

<div id="${bookId}\_title">The Stand</div>

</div>

1. 脚本为中心的Ajax和JavaScript

脚本为中心的Ajax包括返回Javascript并且脚本在客户端被赋值。这样的一个例子在下面可以看到：

def showBook() {

def b = Book.get(params.id)

response.contentType = "text/javascript"

String title = b.title.encodeAsJavaScript()

render "$('#book${b.id}\_title').html('${title}');"

}

要记住的重要一点是设置的contentType为text/ javascript。如果你在客户端使用Prototype，返回的JavaScript由于contentType设置会自动被赋值。

显然在这种情况下，你同意的客户端API同时你又不希望客户端得变化与服务器不一致，这样是危险的。这是Rails有一些像RJS之类的原因之一。虽然Grails的目前还没有如RJS的一个功能，但有一个动态的JavaScript插件，提供了类似的功能。

1. Ajax请求和非Ajax请求的响应

很简单，Grails可以使用同样的控制器动作处理Ajax请求和非Ajax请求。Grails添加isXhr（）方法给HttpServletRequest，它可以被用来识别Ajax请求。例如，你可以为Ajax请求使用一个模板来渲染页面片段，为Ajax请求或为常规HTTP请求渲染整个页面：

def listBooks() {

def books = Book.list(params)

if (request.xhr) {

render template: "bookTable", model: [books: books]

} else {

render view: "list", model: [books: books]

}

}

### GORM

Grails控制器是负责处理请求来在应用程序类。控制器接收到一个请求，有可能做一些工作的要求，并最终决定了未来会发生什么、接下来发生的事情，可能包

#### 持久性的基础

通过利用底层的Spring MVC国际化支持，Grails支持开箱即用国际化（i18n）使用。通过Grails，可以自定义显示在视图中的文本。它使用Locale来为用户裁剪信息。Locale是由语言代码 国家代码组成。例如，”en\_US”是美国英语的语言码，而”en\_GB”代表英国英语。

1. 读取对象

Grails自动在你的domain类中添加了一个隐含的id属性，你可以用它来取回对象：

def p = Person.get(1)或def p = Person.get(params.id)

这里使用get方法，它会根据数据库标识符从数据库中取回一个Person对象。也可以使用read（）方法以只读方式来获取Person对象。

def p = Person.read(1)或def p = Person.read(params.id)

在这种情况下，底层的Hibernate引擎不会做任何脏检查，并且对象不会被持久化。注意如果你显式调用save方法，对象被放回读写状态。

此外，您还可以通过使用load方法加载一个实例的代理：

def p = Person.load(1)

这导致没有数据库访问，直到一个非getId （）方法被调用。Hibernate初始化代理实例，或者如果没有根据给定id找到对象，将抛出异常。

另外：读取对象时，我们还可能需要检索，如：

// get all the albums; careful, there might be many!

def allAlbums = Album.list()

// get the ten most recently created albums

def topTen = Album.list(max:10, sort:'dateCreated', order:'desc')

// get the total number of albums

def totalAlbums = Album.count()

其中：

•max: 返回的最大实例数

•offset:返回偏移相对于0的第一个结果

•sort:排序的属性名称

•order: 结果的顺序，升序或降序

**保存、更新、删除**

新建一个实例

def album = new Album(params)

album.save()

更新一个实例

def album = Album.get(1)

album.title = "The Changed Title"

album.save()

一些老的版本数据库在update操作时需要指定insert：album.save(insert:true)

使用delete来删除一个实例:

def album = Album.get(1)

album.delete()

1. Associations

如果我们想要一个特定顺序的集合可以使用SortedSet，需要实现Comparable接口来存放SortedSet元素：

class Album {

...

SortedSet songs

}

class Song implements Comparable {

...

int compareTo(o) {

if(this.trackNumber > o.trackNumber) {

return 1

} else if(this.trackNumber < o.trackNumber) {

return -1

}

return 0

}

}

或者，可以指定排序的顺序

class Song {

...

static mapping = {

sort "trackNumber"

}

}

你可能不想为每个查询都进行排序，在这种情况下，可以这样：

static mapping = {

songs sort: "trackNumber"

}

GOR允许使用addTo\* 和 removeFrom\*动态方法来新增或删除一个实例：

new Album(title:"Odelay",

artist:beck,

year:1996)

.addToSongs(title:"Devil's Haircut",

artist:beck, duration:342343)

.addToSongs(title:"Hotwax", artist:beck, duration:490583)

...

.save()

1. 持久性

当你定义了一个belongsTo的级联关系，删除Master时会将Detail也删除，这时可以通过cascade来指定想要级联的操作，如下，只有save和update操作，当Master执行删除时Detail将不会删除

class Album {

...

static mapping = {

songs cascade:'save-udpate'

}

}

#### 查询

GORM提供了动态查询器、criteria、Hibernate面向对象查询语言HQL等一些强大查询方式。根据你查询的复杂程度，你有以下几种强大的、柔性的选项：

•Dynamic Finders----动态查询器

•Where Queries-------where查询

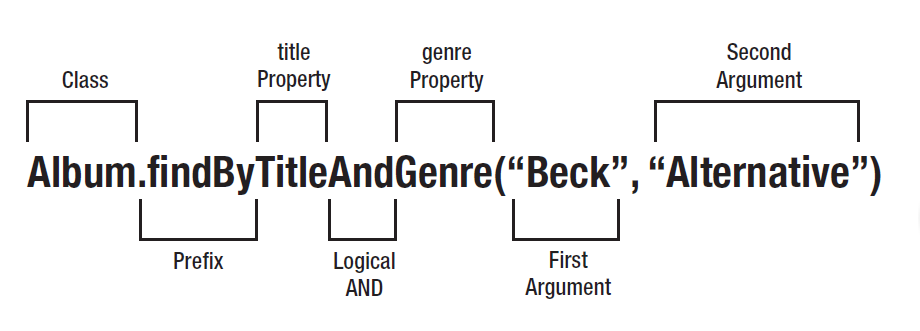
•Criteria Queries----- Criteria查询

•Hibernate Query Language (HQL)----Hibernate查询语言

此外，Groovy使用GPath和sort、 findAll等方法操作集合的能力结合GORM形成了一个强大的组合。

1. 动态查询

动态查询是GORM中最强大的查询，如下是一个findBy查询结构：



如：

// retrieve an album where the title contains 'Shake'

def album = Album.findByTitleLike('%Shake%')

// get an album created in last 10 days

def today = new Date()

def last10Days = Album.findByDateCreatedBetween(today-10,today)

// first album that is not 'Rock'

def somethingElse = Album.findByGenreNotEqual('Rock')

可用的动态查询方法：

| 表达式 | 参数 | | 示例 |
| --- | --- | --- | --- |
|  |  | | |
| Between | 2 | Album.findByDateCreatedBetween(today-10,today) | |
| Equals | 1 | Album.findByTitleEquals(‘Aha Shake Heartbreak’) | |
| GreaterThan | 1 | Album.findByDateCreatedGreaterThan(lastMonth) | |
| GreaterThanOrEqual | 1 | Album.findByDateCreatedGreaterThanOrEqual(lastMonth) | |
| Ilike | 1 | Album.findByTitleIlike(‘shake’) | |
| InList | 1 | Album.findByTitleInList([‘Aha Shake Heartbreak’, ‘Odelay’]) | |
| IsNull | 0 | Album.findByGenreIsNull() | |
| IsNotNull | 0 | Album.findByGenreIsNotNull() | |
| LessThan | 1 | Album.findByDateCreatedLessThan(lastMonth) | |
| LessThanOrEqual | 1 | Album.findByDateCreatedLessThanOrEqual(lastMonth) | |
| Like | 1 | Album.findByTitleLike(‘Shake’) | |
| NotEqual | 1 | Album.findByTitleNotEqual(‘Odelay’) | |
| Rlike | 1 | Album.findByTitleRlike(/F.\*/) | |

类似的GORM还提供了findallby \*，所有的实例匹配的方法将放入java.util.list中，countby \*方法，它返回找到的实例总和

1. Criteria查询

Grails提供各个领域类createCriteria的静态方法，有四种方法可以调用：

•get：定位查询的唯一实例

•list：返回一个查询的实例列表

•scroll：返回的查询scrollableresults实例

•count：返回查询结果的总和

如：

def c = Album.createCriteria()

def results = c.list {

eq('genre', 'Alternative')

between('dateCreated', new Date()-30, new Date())

}

1. 投影查询

投影被用于定制查询结果. 要使用投影你需要在criteria builder树里定义一个"projections"节点. projections节点内可用的方法等同于 Hibernate 的 Projections 类中的方法:

def criteria = Album.createCriteria()

def count = criteria.get {

projections {

countDistinct('name')

}

songs {

eq('genre', 'Alternative')

}

}

当多个字段在投影中指定，将返回一个值的列表。否则返回一个单一的值。

1. 分离查询

分离Criteria是criteria查询，它不与任何给定的数据库会话/连接相关。从Grails2.0开始支持独立Criteria，独立Criteria查询有许多用途，包括允许您创建可重复使用的通用标准查询，执行子查询和执行批量更新/删除。

**创建分离查询：**

使用分离查询的主要点是grails.gorm.DetachedCriteria类，它接收一个域类作为唯一的参数传递给它的构造函数：

import grails.gorm.\*

…

def criteria = new DetachedCriteria(Person)

一旦你获得一个detached criteria实例的引用，您可以执行在where查询或criteria查询来建立相应的查询。建立一般的criteria查询，你可以使用build方法：

def criteria = new DetachedCriteria(Person).build {

eq 'lastName', 'Simpson'

}

请注意，在DetachedCriteria实例上的build方法并不变化原始对象，而是返回一个新的查询。换句话说，你必须使用build方法返回值来获得突变的Criteria对象：

def criteria = new DetachedCriteria(Person).build {

eq 'lastName', 'Simpson'

}

def bartQuery = criteria.build {

eq 'firstName', 'Bart'

}

**执行分离查询**

| 方法 | 描述 |
| --- | --- |
|  |  |
| **List** | 列出所有匹配的实体 |
| **Get** | 返回一个匹配的结果 |
| **Count** | 指望所有匹配的记录 |
| **exists** | 返回true，如果任何匹配的记录存在 |
| **deleteAll** | 删除所有匹配的记录 |
| **updateAll（Map）** | 更新所有匹配的记录与给定的属性 |

不同于常规的criteria，分离查询Detached Criteria是延迟加载的，定义的时候不执行查询。一旦一个独立的标准语句已经构造，则有一些有用的查询方法可用，方法归纳于下表中：

如：

def list = query.list() // all results

def only10 = query.list(max:10) // pagination

def album = query.get() // one result

int total = query.count() // number of results

boolean alreadyExists = query.exists() // check for existence

def rockAlbums = query.findAllByGenre('Rock') // dynamic finders

def punkAlbums = query.list { // combine with other criteria

like('genre', 'Punk')

}

**分离查询中的Lazy Evaluation**

def showAlbums() {

def rockAlbums = new DetachedCriteria(Album).build {

eq('genre', 'Rock')

}

[rockAlbums: rockAlbums]

}

页面中：

<g:each in="${rockAlbums}" var="album">

<p>${album.title}</p>

</g:each>

这样可以减少数据库的访问来提高性能

**分离查询中的批处理**

DetachedCriteria类可用于执行批处理操作，如批量更新和删除。例如，下面的查询将更新所有的人姓“辛普森”有姓“布洛格斯”：

def criteria = new DetachedCriteria(Person).build {

eq 'lastName', 'Simpson'

}

int total = criteria.updateAll(lastName:"Bloggs")

请注意，在DetachedCriteria实例中有jion的不能进行批处理操作。

批量删除记录你可以使用deleteAll方法：

def criteria = new DetachedCriteria(Person).build {

eq 'lastName', 'Simpson'

}

int total = criteria.deleteAll()

1. Where查询

**基本查询**

Where方法接收一个闭包，这看起来非常类似于Groovy的正规集合方法。闭包使用Groovy语法来定义criteria逻辑，例如：

def query = Person.where {

firstName == "Bart"

}

Person bart = query.find()

每个Groovy操作符映射到一个正规的criteria方法。下表提供了Groovy的操作符到方法的映射：

| 操作符 | 查询方法 | 描述 | |
| --- | --- | --- | --- |
|  |  | | |
| == | eq | | 等于 |
| ！= | ne | | 不等于 |
| > | gt | | 大于 |
| < | lt | | 小于 |
| > = | ge | | 大于或等于 |
| <= | le | | 小于或等于 |
| 在 | inList | | 给定列表内包含 |
| ==〜 | like | | 就像一个给定的字符串 |
| =〜 | iLike | | 不区分大小写喜欢 |
| == | eq | | 等于 |
| ！= | ne | | 不等于 |
| > | gt | | 大于 |
| < | lt | | 小于 |

可以使用正规的Groovy比较操作符和逻辑来制定复杂的查询：

def query = Person.where {

(lastName != "Simpson" && firstName != "Fred") || (firstName == "Bart" && age > 9)

}

def results = query.list(sort:"firstName")

**子查询**

如下查询持续时间最大歌曲：

def longSongs = Song.where {

duration > avg(duration)

}

下表列出了可用的子查询：

| 方法 | 描述 |
| --- | --- |
|  |  |
| **avg** | 所有值的平均值 |
| **sum** | 所有值的总和 |
| **max** | 的最大值 |
| **min** | 最小值 |
| **count** | 所有值的计数 |
| **property** | 检索得到的实体的属性 |

**其他函数**

| 方法 | 描述 |
| --- | --- |
|  |  |
| second | 日期属性的秒 |
| minute | 日期属性的分 |
| hour | 日期属性中的小时 |
| day | 日期属性月份的一天 |
| month | 日期属性的月份 |
| year | 日期属性的一年 |
| lower | 一个字符串属性转换为大写 |
| upper | 一个字符串属性转换为小写 |
| length | 一个字符串属性的长度 |
| trim | 修剪字符串属性 |

如：

def query = Album.where {

year(dateCreated) == 2012

}

1. 分页

查询完成后会有很多信息被显示出来，这时我们来制定分页，这样显示的效果会更加舒适，如：我们返回前10个结果：

def results = Album.list(max:10)

我们也可以添加其他参数，如：

def results = Album.findAllByGenre("Alternative", [sort:'dateCreated', order:'desc'])

在GSP页面我通过<g:paginate>标签来设置翻页：

<g:paginate prev="Back" next="Forward" total="${Album.count()}" />

#### 配置GORM

通过利用底层的Spring MVC国际化支持，Grails支持开箱即用国际化（i18n）使用。通过Grails，可以自定义显示在视图中的文本。它使用Locale来为用户裁剪信息。Locale是由语言代码 国家代码组成。例如，"en\_US"是美国英语的语言码，而"en\_GB"代表英国英语。

1. SQL日志

在grails-app/conf/DataSource.groovy中，配置logSql=true，这样在程序执行时会输出sql语句：

dataSource {

...

logSql = true

}

如果想看到实际的值输出，需要在grails-app/conf/Config.groovy配置：

log4j = {

logger {

trace "org.hibernate.SQL", "org.hibernate.type"

}

}

1. 方言

每一个数据库都有自己的方言，下面以MySql为例，在grails-app/conf/DataSource.groovy中配置方言。

dataSource {

...

dialect = org.hibernate.dialect.MySQL5InnoDBDialect

}

1. 其他

还可以在grails-app/conf/DataSource.groovy中配置：

Hibernate的二级缓存

hibernate {

cache.use\_second\_level\_cache = true

cache.use\_query\_cache = false

cache.provider\_class = 'net.sf.ehcache.hibernate.EhCacheRegionFactory '

}

事务级别：

hibernate {

hibernate.connection.isolation=4

}

在Config.groovy中对所以得域类进行设置，如设定所以数据库映射都不使用version和autoTimestamp：

grails.gorm.default.mapping = {

version false

autoTimestamp false

}

对constraints设定：

grails.gorm.default.constraints = {

nullable true

blank false

size 1..20

}

#### GORM的语义

1. Hibernate Session

模型中持久会话的概念使用org.hibernate.session类。如：

def album = new Album(...)

album.save()

当创建一个实例后调用save()方法，由于Hibernate的write-behind机制，新的实例并没有真正存入数据库中，当使用flush时会将实例同步到数据库中，如：

def album2 = new Album(..)

album2.save(flush:true)

所有的Session都会消耗内存，这时我们需要手动管理，SessionFactory的有一个方法，称为currentsession()，可以用来获得会话绑定到当前线程：

def sessionFactory

...

def index() {

def session = sessionFactory.currentSession()

}

作为替代，你可以使用withsession方法，在任何域类是可用的：

def index() {

Album.withSession { session ->

...

session.clear()

}

}

当执行完操作时通过session.clear()来释放，也可以使用discard()方法，其中clear()方法从会话中删除所有持续的实例，使用discard()只删除你不再需要的实例。

1. Session的自动冲洗

当下例情况之一发生时，Session会自动冲洗：

•每当运行查询；

•直接控制器的动作完成后，如果没有异常抛出；

•直接在事务提交之前。

如：

def album = Album.get(1)

album.title = "Change It"

这将title更改为Change It，并保存到数据库中，虽然没有执行save()。

在datasource.groovy中你可以指定hibernate.flush.mode如：

hibernate.flush.mode="manual"

hibernate.flush.mode设置的可能值如下：

•manual：当你允许时才flush，只有当flush:true参数才会传递给save()或delete()。

•commit：只有在事务提交才flush。

•auto：flush在事务提交之前并且在运行查询时。

1. GORM中的Transactions

Grails使用Spring的PlatformTransactionManager，如果抛出一个异常，事务范围内的所有更改将被回滚，通过调用setrollbackonly()方法

def save() {

Album.withTransaction { status ->

def album = Album.get(params.id)

album.title = "Changed Title"

album.save(flush:true)

...

// something goes wrong

if(hasSomethingGoneWrong()) {

status.setRollbackOnly()

}

}

}

如果你有JDBC 3.0一个compliant database，可以指定回滚到某个特定点而不是回滚整个事务

def save() {

Album.withTransaction { status ->

def album = Album.get(params.id)

album.title = "Changed Title"

album.save(flush:true)

def savepoint = status.createSavepoint()

...

// something goes wrong

if(hasSomethingGoneWrong()) {

status.rollbackToSavepoint(savepoint)

// do something else

}

}

}

#### GORM性能调优

1. Eager 和 Lazy Associations对比

Grails默认会使用Lazy，如下，当你定义一个list，输出其中的姓名

def albums = Album.list()

for(album in albums) {

println album.artist.name

}

若在域类中使用

class Album {

...

static mapping = {

artist fetch:'join'

}

}

并将def albums = Album.list()改为def albums = Album.list(fetch:[artist:'join'])，这样使用联接查询可以通过降低多个SQL SELECT语句以一个SELECT语句中使用SQL连接解决n + 1的问题。然而，连接查询也可以是昂贵的，这取决于连接数和被从数据库里的数据量。

也可以使用FETCH参数通过map作为最后一个参数：

def albums = Album.findAllByGenre("Alternative", [fetch:[artist:'join']])

使用标准的查询，您可以使用连接方法：

def albums = Album.withCriteria {

...

join 'artist'

}

1. 批量抓取

如下例：如果你有23首歌曲，设置了batchSize为10，那么数据库将执行3个查询

class Song {

...

static mapping = {

batchSize 10

}

}

1. 二级缓存

在datasource.groovy缓存配置改变

hibernate {

cache.use\_second\_level\_cache=true

cache.use\_query\_cache=true

cache.provider\_class='net.sf.ehcache.hibernate.EhCacheRegionFactory '

}

有四种基本的缓存策略：

•read-only：应用程序不需要创建后修改数据，那么使用这种策略。

•nonstrict-read-write：应用程序很少修改的数据和事务更新时使用。

•read-write：应用程序需要用户频繁修改的数据时使用。

•transactional：事务缓存提供了充分的交易行为而且没有机会dirty reads。

1. 查询缓存

查询缓存也可以在datasource.groovy中使用hibernate.cache.use\_query\_cache设置，在使用中，你必须指定cache:true才能使用，如：

def albums = Album.list(cache:true)

def albums = Album.findAllByGenre("Alternative", [cache:true])

def albums = Album.withCriteria {

...

cache true

}

1. 继承策略

默认情况下GORM类使用table-per-hierarchy（表层次结构）来继承映射。这有一个缺点，就是在数据库级别上不能有NOT-NULL约束。如果您更喜欢使用table-per-subclass继承策略你可以这样做如下：

class Payment {

Integer amount

static mapping = {

tablePerHierarchy false

}

}

class CreditCardPayment extends Payment {

String cardNumber

}

#### 乐观锁和悲观锁

在默认情况下GORM使用乐观锁定，并自动注入一个版本属性到每一个类，它又反过来在数据库级别映射一个version列，可以禁用version方法：

class Person {

static mapping = {

version false

}

}

又如在更新操作时，我们给他上一个悲观锁，这样防止多线程其他用户修改同一条数据，在提交数据时会自动解锁

def update() {

def album = Album.get(params.Id)

album.lock() // lock the Instance

...

}

#### 事件和自动时间戳

GORM支持把事件注册作为一个方法，并当certain事件发生时被触发，如删除，插入和更新时。下面是所支持的事件的列表：

•tbeforeInsert - 对象持久到数据之前执行

•tbeforeUpdate - 对象被更新之前执行

•tbeforeDelete - 对象被删除之前执行

•tbeforeValidate—对象校验前执行

•tafterInsert - 对象持久到数据库之后执行

•tafterUpdate - 对象被更新之后执行

•tafterDelete - 对象被删除之后执行

•tonLoad - 对象从数据库中加载之后执行

如：

class Album {

...

def onLoad() {

new AuditLogEvent(type:"read", data:title).save()

}

def beforeSave() {

new AuditLogEvent(type:"save", data:title).save()

}

def beforeValidate() {

// run before validation happens

}

def beforeValidate(List propertiesBeingValidated) {

// run before validation happens and propertiesBeingValidated

// is a List containing the names of all of the properties which

// are about to be validated.

}

}

Grails还支持automatic time stamps如果你提供名称为datecreated或lastUpdated的属性，Grails会在每次实例保存或更新时自动填充值。然而，如果你喜欢手动管理这些特性，可以禁用autotimestamp方法，如：

class Album {

...

static mapping = {

autoTimestamp false

}

}

### Services

Grails中定义了服务层的概念。Grails团队不鼓励在控制器中签入核心应用程序逻辑，因为它不能重复使用和关注点分离。Grails的服务是把应用程序中大部分逻辑放在其中，留下控制器负责处理重定向等请求流。

#### 创建服务

你可以在应用根目录下运行create-service命令来创建一个Grails服务创建服务：

grails create-service helloworld.simple

如果没有指定创建服务脚本的包，Grails的自动使用应用程序的名称作为包名。上面的例子将在grails-app/services/helloworld/SimpleService.groovy该位置创建一个服务。服务的名称约定以Service结尾，除此之外服务是一个纯Groovy类：

package helloworld

class SimpleService {

}

#### 服务和依赖注入

Grails服务的一个重要方面是使用Spring框架的依赖注入功能的能力。Grails支持“按照惯例依赖注入”。换句话说，你可以使用一个服务类名的属性名，来自动将它们注入控制器、标签库等等。

作为一个例子，考虑一个名为BookService的服务，如果你在控制器中定义了一个名为BookService属性，如下所示：

class BookController {

def bookService

}

在这种情况下，根据其配置的作用域，Spring容器将自动注入该服务的一个实例。所有的依赖注入是通过名称来完成。您也可以像如下指定类型：

class AuthorService {

BookService bookService

}

#### Action中的服务

一般来说，你应该尽量保持你的Grails控制器和taglibs紧凑简洁，不应该让很多复杂的业务出现在控制器或标签库，而应该将他提取到service中，这个service必须以XXXService命名格式。

1. 配置服务bean的属性

例如当一个字符串他指向一个RESTful来获取某些内容，我们可以将它定义到bean中，在grails-app/conf/Config.groovy来配置他：

beans {

albumArtService {

artworkRequestUrl ='http://itunes.apple.com/search? entity=album&attribute=albumTerm'

}

如果有多个服务在不同的包中定义却有相同名称，关联一个服务的bean的默认名称是有问题的。例如，考虑一个应用程序中定义了一个名为com.demo.ReportingService一个服务类和应用程序使用一个插件名为ReportingUtilities和插件提供了一个名为com.reporting.util.ReportingService服务类的情况。每一个默认的bean名称会互相冲突。Grails的通过使用插件名作为前缀来更改默认的bean，以解决这个问题。

1. 高速缓存服务的方法

在grails-app/conf/BuildConfig.groovy定义一个缓存机制：

grails.project.dependency.resolution = {

plugins {

compile ':cache:1.0.0'

}

}

在使用这个插件，将通过注释依赖的方法来使用它：

import grails.plugin.cache.Cacheable

class AlbumArtService {

@Cacheable('albumArt')

def getAlbumArt(String artist, String album) {

...

}

}

#### 事务管理

如前所述，服务往往封装的业务操作是处理几个域对象。如果在执行中发生异常，你可能不希望任何早期的变化被提交到数据库。

所有的服务都是默认事务的。要禁用此设置可以设置transactional属性为false：

class CountryService {

static transactional = false

}

您也可以通过withTransaction方法使用编程式事务：

class StoreService {

// turn off automatic transaction management

static transactional = false

void someServiceMethod() {

Album.withTransaction {

// everything in this closure is happening within a transaction

// which will be committed when the closure completes

}

}

}

显式控制事务回滚：

class StoreService {

// turn off automatic transaction management

static transactional = false

void someServiceMethod() {

Album.withTransaction { tx ->

// do some work with the database

// if the transaction needs to be rolled back for

// any reason, call setRollbackOnly() on the

// TransactionStatus argument...

tx.setRollbackOnly()

}

}

}

**自定义事务配置**

Grails中还提供了@Transactional和@NotTransactional注解，在你需要更细粒度的控制事务在每个方法级别或需要指定一个传播水平的情况。例如，如果当一个类使用@Transactional可以使用@ NotTransactional注释可以用来标记一个特定的方法以跳过这个方法。

#### 服务作用域

默认情况下，访问服务方法是不同步的，所以没有什么能阻止这些方法的并发执行。事实上，因为该服务是独立的，并可以同时使用，你应该在服务中存储状态非常小心。或采取简单的（更好）的方式并且在服务中从不存储状态。

您可以通过将一个服务放在特定作用域下来改变这种行为。支持的范围是：

•prototype -被注入到另一个类时每次都创建一个新的服务

•request-一个新的服务将每个请求创建

•flash -一种新的服务将只对当前和下一请求创建

•flow -在网络流服务将存在于流动的范围

•conversation -在网络中流动的服务将存在于对话的范围。即根流和它的子流

•session -一个服务是对用户会话的范围内创建

•singleton（默认） -该服务只有一个实例存在过

如果您的服务是flash，flow或conversation 作用域，它必须实现java.io.Serializable接口，并仅在webflow上下文中可用。

为了使其中一个作用域，给你的类添加一个静态scope属性，其属性值是上面任何一个，例如：

static scope = "flow"

#### Exposing服务

在grails-app/conf/BuildConfig.groovy声明：

grails.project.dependency.resolution = {

// ...

plugins {

runtime ':jmx:0.7'

// ...

}

}

在service中通过expose来使用JMX

package com.gtunes

class StoreService {

static transactional = true

static expose = ['jmx']

int getNumberOfAlbums() {

Album.count()

}

int getNumberOfAlbumsForGenre(String genre) {

Album.countByGenre(genre)

}

}

如果你想使用JMX和Axis2，可以这样调用：

static expose = ['jmx', 'axis2']

### 集成和依赖管理

#### Grails和配置

通过利用底层的Spring MVC国际化支持，Grails支持开箱即用国际化（i18n）使用。通过Grails，可以自定义显示在视图中的文本。它使用Locale来为用户裁剪信息。Locale是由语言代码 国家代码组成。例如，"en\_US"是美国英语的语言码，而"en\_GB"代表英国英语。

1. 配置基础知识

config.groovy是一个groovy脚本，类似于普通Java属性文件，如：

grails.mime.file.extensions = true

一个布尔属性称为grails.mime.file.extensions设置为true。访问此设置，使用在控制器和视图中可用的grailsapplication对象的配置属性：

assert grailsApplication.config.grails.mime.file.extensions == true

Groovy还支持组采用分块设置：

grails.mime {

file.extensions = true

types = [html: 'text/html']

}

1. 环境的具体配置

如在 DataSource.groovy可以指定环境的特定设置：

environments {

development {

grails.serverURL = "http://localhost:8080"

} production {

grails.serverURL = "http://www.gtunes.com"

}

}

1. 配置日志记录

Grails利用其自身的配置机制来提供对Log4j 日志系统的配置，因此你所需要做的只是将log4j配置添加到grails-app/conf/Config.groovy配置文件中。

log4j = {

error 'codehaus.groovy.grails.web.servlet', // controllers

'codehaus.groovy.grails.web.pages', // GSP

'codehaus.groovy.grails.web.sitemesh', // layouts

'codehaus.groovy.grails.web.mapping.filter', // URL mapping

'codehaus.groovy.grails.web.mapping', // URL Mapping

'codehaus.groovy.grails.commons',// core / classloading

'codehaus.groovy.grails.plugins',// plugins

'codehaus.groovy.grails.orm.hibernate' // hibernate integration

}

以下是按照优先级降序（由高到低）排列的标准日志级别：

off、fatal、error、war、info、debug、trace、all

比如你有一个'org.example.domain'记录器，其配置如下：

warn 'org.example.domain'

那么级别是'warn'、'error'或者'fatal'的消息都将会输出，而其他级别的都将被忽略。在我们继续记录器以前，我们需要对'off'和'all'的级别做一个小关注，它们都只能在配置中使用，你不能使用它们记录任何日志信息。因此如果你将级别配置为'off'，那么将不会有任何信息输出，而配置为'all'意味着你将看到所有的日志信息。

你也可以创建你自己的自定义log4j Appenders，如下例设置了一个附加文件附加写入日志文件的代码。

log4j {

appenders {

rollingFile name:"myl\_og",

file:"/var/log/gtunes.log", maxFileSize:"lMB",

layout: pattern(conversionPattern: '%c{2} %m%n')

}

}

下面是log4j可用的Appenders

•jdbc：通过org.apache.log4j.jdbc.ddbcappender日志数据库连接。

•null：通过org.apache.log4j.varia.nullappender什么都不做！

•console：通过org.apache.log4j.consoleappender在控制台输出日志。

•file：通过org.apache.log4j.fileappender记录到一个文件。

•rollingfile：通过org.apache.log4j.rollingfileappender记录到一个文件，会自动备份，当达到设置最大值时会重新创建。

多数情况下，Log4j DSL使用缺省的PatternLayout，除此之外，还有以下布局可以选择：

•XML：创建一个XML日志文件

•HTML；创建一个HTML日志文件

•simple：一个简单的文本日志

•pattern：pattern布局

当一个异常发生时，可能有大量的来自Java和Groovy内部栈跟踪信息，这其实是很恼人的。Grails的过滤器将这些不相干的细节屏蔽了，并且将栈的跟踪信息限制非Grails/Groovy的类包范围。

当异常发生时，完整的跟踪信息总是被记录到StackTrace记录器中，此记录器缺省将内容输出到一个名为stacktrace.log的文件中。

log4j {

appenders {

rollingFile name:"stacktraceLog",

file:"/var/log/unfiltered-stacktraces.log",

maxFileSize:"1MB",

layout: pattern(conversionPattern: '%c{2} %m%n')

}

error stacktraceLog:"StackTrace"

}

1. 外部配置

在部署过程中，config.groovy文件编译war包。这虽然有它的优点，但你可能想logging信息配置在war报外，这时你可以使用Grails的外部配置机制。如：

grails.config.locations = ["file:${userHome}/gtunes-logging.groovy"]

#### 依赖解析

Grails提供了依赖解析的DSL来处理插件和JAR的依赖处理。你可以在grails-app/conf/BuildConfig.groovy中设置，主要包括了继承，库，依赖，和插件:

grails.project.dependency.resolution = {

inherits("global") {

...

}

repositories {

...

}

dependencies {

...

}

plugins {

...

}

}

1. 继承的依赖关系

默认情况下，每一个Grails应用程序继承了一些框架的依赖。配置如下：

inherits "global"

在BuildConfig.groovy文件中，你可以通过excludes方法来排除掉指定的依赖，比如：

inherits("global") {

excludes "oscache", "ehcache"

}

1. 声明库

应用程序有时候会需要Grails不提供的依赖关系，如依赖的lib中的jar文件等，Grails需要知道到哪里找到这些依赖关系，如：

grails.project.dependency.resolution = {

repositories {

grailsPlugins()

grailsHome()

grailsCentral()

mavenCentral()

...

}

}

1. 依赖

通常情况下，依赖是不变的。或者说，对于给定了group、name和version组合的jar（或者插件）来说，就意味着将永远不变。Grails的依赖管理系统就是利用这点来缓存依赖以避免每次从远程存储库中下载。但有时候这并不是可取的。比如很多开发者使用的是快照 （例如依赖一个以“-SNAPSHOT”结尾的版本号）规约，这样就是在同一个版本下，也可以一次一次的更新变化。多依赖可在一次调用中指定：

grails.project.dependency.resolution = {

...

dependencies {

// specify dependencies here under either 'build', 'compile', 'runtime', 'test', or provided' scopes, e.g.

runtime 'mysql:mysql-connector-java:5.1.21'

runtime 'commons-httpclient:commons-httpclient:3.1'

// or ...

runtime 'mysql:mysql-connector-java:5.1.21',

'commons-httpclient:commons-httpclient:3.1'

// or...

runtime [[group: 'mysql', name: 'mysql-connector-java', version: '5.1.21'],

[group: 'commons-httpclient', name: 'commons-httpclient', version: '3.1']]

}

...

}

1. 插件

为一个插件指定依赖的方式跟你为一个应用的方式一致。当一个插件被安装以后，此应用将自动继承插件的依赖：

grails.project.dependency.resolution = {

...

plugins {

runtime ":hibernate:$grailsVersion"

runtime ":jquery:1.7.1"

runtime ":resources:1.1.6"

runtime ":searchable:0.6.4"

compile ":cache:1.0.0"

compile ":rest-client-builder:1.0.2"

build ":tomcat:$grailsVersion"

}

}

### 插件

扩展的平台可能听起来很复杂，但插件可以从平凡简单到令人难以置信的强大。如果你知道如何建立一个Grails应用程序，你就会知道如何创建一个插件的共享数据模型或一些静态资源。

#### 插件的基础

Grails有一组默认的插件：GORM and the Grails MVC 框架。通过grails list-plugins指令可以查看Grails的中央库，下载最新发布的插件，查看插件的详细信息：grails plugin-info spring-security-core

插件安装

在grails-app/conf/BuildConfig.groovy文件中，我们导入spring-security-core插件：

grails.project.dependency.resolution = {

...

plugins {

compile ':spring-security-core:1.2.7.3'

...

}

}

1. 本地插件

当然，你创建的插件不一定住在中央Grails库。Grails插件封装成简单的ZIP文件，如果你从别处下载一个插件zip文件，你可以指定他在磁盘的位置来进行安装：

grails install-plugin ~/grails-spring-security-core-1.2.7.3.zip

也可以从网站上下载安装：

$ grails install-plugin http://plugins.mydomain.com/spring-security-core/spring-security-core-1.2.7.3.zip

1. 插件的创建

可以运行下面的指令来创建一个插件：

grails create-plugin simple-cache

它会创建一个SimpleCacheGrailsPlugin.groovy文件在项目根目录下。通过以下特 殊的属性，你还可以提供插件的一些额外的信息：

• title - 用一句话来简单描述你的插件

• author - 插件的作者

• authorEmail - 插件作者的电子邮箱

• description - 插件的完整特性描述

• documentation - 插件文档的 URL

如：

class SimpleCacheGrailsPlugin {

// the plugin version

def version = "0.1"

// the version or versions of Grails the plugin is designed for

def grailsVersion = "2.0 > \*"

// the other plugins this plugin depends on

def dependsOn = [:]

// resources that are excluded from plugin packaging

def pluginExcludes = [

"grails-app/views/error.gsp"

]

// TODO Fill in these fields

def title = "Simple Cache Plugin" // Headline display name of the plugin

def author = "Your name"

def authorEmail = ""

def description = '''\

Brief summary/description of the plugin.

'''

// URL to the plugin's documentation

def documentation = "http://grails.org/plugin/simple-cache"

...

}

1. 发布插件

要发布插件，你需要一个命令行窗口，并且进入到插件的根目录，输入：

grails package-plugin

这将创建一个 grails-+插件名称+版本的 zip 文件，package-plugin 命 令还将生成

plugin.xml，在此文件中包含机器可读的插件信息，比如插件的名称、版本、 作

者等等。

**Grails 2.3版本以上**

为了让你的插件可供使用的Grails应用程序运行Maven的安装命令：

grails maven-install

这将安 装插件到你的本地Maven的缓存。然后在应用程序中使用该插件在你的声明对插件的依赖grails-app/conf/BuildConfig.groovy文件：

compile ": example:0.1"

**在 Grails 插件的存储仓库（ Repository ）发布插件**

发布插件到Grails 插件的存储仓库，这样通过list-plugins 命令就可以看到你的插件了。

grails list-plugins

此命令将列出 Grails 插件存储库的所有插件，当然了也可以用 plugin-info 来查看指定插件的信息：

grails plugin-info [plugin-name]

这将输出更多的详细信息，这些信息都是维护在插件描述文件中的。

当你有访问 Grails 插件存储库的权限时，要发行你的插件，只需要简单执行

release-plugin 即可：

grails release-plugin

这将自动地将改动提交到 SVN 和创建标签（svn 的 tagging），并且通过

list-plugins 命令你可以看到这些改动

1. 插件的结构

如前所提到的，一个插件除了包含一个插件描述文件外，几乎就是一 个常规的Grails 应用。尽管如此，当安装以后，插件的结构还是有些许的差别。比如一个插件目录的结构如下：

+ grails-app

+ controllers

+ domain

+ taglib

etc.

+ lib

+ src

+ java

+ groovy

+ web-app

+ js

+ css

从本质上讲，当一个插件被安装到 Grails 工程以后，grails-app 下 边的内容将被拷贝到以 plugins/example-1.0/grails-app(以 example 为例)目录中，这些内容不会被拷贝到工程的源文件主目录，即插件永远不会跟工程的主目录树有任何接口上的关系。然而， 那些在特定插件目录中 web-app 目录下的静态资源将会被拷贝到主工程的web-app 目 录下，比如 web-app/plugins/example-1.0/js.因此，要从正确的地方引用这些静态资源也就成为插件的责任。

比如，你要在GSP 中引用一个 JavaScript 文 件，你可以这样：

<g:createLinkTo dir="/plugins/example/js" file="mycode.js" />

这样做当然可以， 但是当你开发插件并且单独运行插件的时候，将产生相对链接

（link）的问题。为了应对这种变化即不管插件是单独运行还是在 Grails 应用中运行，特地新增一个特别的 pluginContextPath 变 量，用法如下：

<g:createLinkTo dir="${pluginContextPath}/js" file="mycode.js" />

这样在运行期间 pluginContextPath 变量将会等价于/ 或 /plugins/example，

这取决于插件是单独运行还是被安装在 Grails 应用中。

在 lib 和 src/java 以及 src/groovy 下的 Java、Groovy 代码将被编译到当前工程的 web-app/WEB-INF/classes 下边，因此在运行时也不会出 现类找不到的问题。

1. 插件的引入

在项目中我们要

#### 提供基础工作

通过利用底层的Spring MVC国际化支持，Grails支持开箱即用国际化（i18n）使用。通过Grails，可以自定义显示在视图中的文本。它使用Locale来为用户裁剪信息。Locale是由语言代码 国家代码组成。例如，”en\_US”是美国英语的语言码，而”en\_GB”代表英国英语。

1. 添加新的脚本

一个插件可以简单地通过提供在其scripts目录相关的Gant脚本中添加一个新的脚本：

+ MyPlugin.groovy

+ scripts <-- additional scripts here

+ grails-app

+ controllers

+ services

+ etc.

+ lib

在 grails-app 相关的目录树下，可以增加新的控制器、标签库、服务等，不过

要 注意：当插件被安装后，将从其被安装的地方加载，而不是被拷贝到当前主

应用工程的相应目录。

+ ExamplePlugin.groovy

+ scripts

+ grails-app

+ controllers <-- additional controllers here

+ services <-- additional services here

+ etc. <-- additional XXX here

+ lib

1. 查看模板和视图解析

当一个插件提供了一个控制器，它也可以提供默认的视图来呈现。这是通过插件模块化您的应用程序的绝佳方式。Grails的视图解析机制将首先查找应用程序中的视图，如果失败将尝试寻找插件中的视图。这意味着，你可以覆盖在应用程序的视图，创建相应的GSP在插件grails-app/views目录中。

例如，由 “亚马逊”的插件提供一个名为BookController的控制器。如果正在执行的动作列表，Grails会首先寻找一个叫做grails-app/views/book/list.gsp视图，如果失败它会在插件中寻找视图。

但是，如果该插件还提供了模板，那么下面的语法可能是必要的：

<g:render template="fooTemplate" plugin="amazon"/>

注意使用插件属性，它包含模板所在的插件的名称。如果没有指定，那么Grails会找相对于应用程序的模板。

#### 评估规约

在得以继续查看基于规约所能提供的运行时配置以前， 有必要了解一下怎样来评估插件的这些基本规约。本质上，每一个插件都有一 个隐含的GrailsApplication 接口的实例变量：application。GrailsApplication 提供了在工程内评估这些规约的方法并且保存着所有类 的相互引用，这些类都实现了 GrailsClass 接口。

一个 GrailsClass 代 表着一个物理的 Grails 资源， 比如一个控制器或者一个标签库。如果要获取所有 GrailsClass 实例，你可以这样：

application.allClasses.each { println it.name }

在 GrailsApplication 实例中有一些特殊的属性可以方便的操作你感兴趣 的人工制品（artefact）类型，比如你要获取所有控制器的类，可以如此：

application.controllerClasses.each { println it.name }

这些动态方法的规约如下：

• \*Classes - 获取特定人工制品名称的所有类，比如

application.controllerClasses。

• get\*Class - 获取特定人工制品的特定类，比如

application.getControllerClass("ExampleController")。

• is\*Class - 如果给定的类是指定的人工制品类型，那么返回 true， 比如 application.isControllerClass(ExampleController.class)。

• add\*Class - 为给定的人工制品类型新增一个类并且返回新增的GrailsClass 实例，比如application.addControllerClass(ExampleController.class)

GrailsClass 接口本身也提供了很多有用的方法以允许你进一步 的评估和了解

这些规约，他们包括：

• getPropertyValue - 获取给定属性的初始值

• hasProperty - 如果类含有指定的属性，那么返回 true

• newInstance - 创建一个类的新实例

• getName - 如果可以的话，返回应用类的逻辑名称，此名称不含后缀部分

• getShortName - 返回类的简称，不包含包前缀

• getFullName - 返回应用类的完整名称，包含后缀部分和包的名称

• getPropertyName - 将类的名称返回为属性名称

• getLogicalPropertyName - 如果可以的话，返回应用类的逻辑属性名称 ，此名称不包含后缀部分

• getNaturalName - 返回属性名称的自然语言的术语（比如将'lastName'变为 'Last Name'）

• getPackageName - 返回包的名称定

#### 参与构建事件

1. 安装后进行配置和参与升级操作

Grails 插件可以在安装完后进行配置并且可以参与应用的升级过程（通过upgrade 命令），这是由 scripts 目录下两个特定名称的脚本来 完成的：\_Install.groovy和 \_Upgrade.groovy。\_Install.groovy是在插件安装完成后被执行的，而\_Upgrade.groovy 是用户每次通过 upgrade 命令来升级他的应用时被执行的。这些是一个普通的 Gant 脚本，因此你完全可以使用 Gant 的强大特性。另外pluginBasedir 被加入到 Gant 的标准变量中，其指向安装插件的根目录。

以下的\_Install.groovy 示例脚本将在 grails-app 目录下创建一个新的目录，并且安装一个配置模板，如下：

Ant.mkdir(dir:"${basedir}/grails-app/jobs")

Ant.copy(file:"${pluginBasedir}/src/samples/SamplePluginConfiguration.

groovy",

todir:"${basedir}/grails-app/conf")

// To access Grails home you can use following code:

// Ant.property(environment:"env")

// grailsHome = Ant.antProject.properties."env.GRAILS\_HOME"

1. 脚本事件

将插件和命令行的脚本事件关联起来还是有可能的，这些事件在执行 Grails 的任务和插件事件的时候被触发。

比如你希望在更新的时候，显示更新状态（如"Tests passed", "Server running"），并且创建文件或者人工制品。一个插件只能 通过 Events.groovy 脚本来监听那些必要的事件。

#### 参与运行时的配置

Grails提供了很多的钩子函数来处理系统的不同部分，并且通过惯例的形式来执行运行时配置。

1. 跟Grails的Spring配置进行交互

可以使用 doWithSpring 闭包来跟 Grails 运行 时的配置进行交互，例如下面的代码片段是取自于 Grails 核心插件 i18n 的一部分，提供国际化支持：

import org.springframework.web.servlet.i18n.CookieLocaleResolver

import org.springframework.web.servlet.i18n.LocaleChangeInterceptor

import org.springframework.context.support.ReloadableResourceBundleMessageSource

class I18nGrailsPlugin {

def version = "0.1"

def doWithSpring = {

messageSource(ReloadableResourceBundleMessageSource) {

basename = "WEB-INF/grails-app/i18n/messages"

}

localeChangeInterceptor(LocaleChangeInterceptor) {

paramName = "lang"

}

localeResolver(CookieLocaleResolver)

}

}

它是通过 Spring Bean Builder 语法来完成的

1. 参与 web.xml 的生成

Grails是在加载的时候生成 WEB-INF/web.xml 文件，因此插件不能直接修改此文件，但他们可以参与此文件的生成。本质上一个插件可以通过doWithWebDescriptor 闭包来完成此功能，此闭包的参数是web.xml，是作为XmlSlurper GPathResult 类型传入的。来自 ControllersPlugin 的示例：

def doWithWebDescriptor = { webXml ->

def mappingElement = webXml.'servlet-mapping'

mappingElement + {

'servlet-mapping' {

'servlet-name'("grails")

'url-pattern'("\*.dispatch")

}

}

}

此处插件得到最后一个<servlet-mapping>元素的引用，并且在其后添加Grails' servlet，这得益于 XmlSlurper 可以通过闭包以编程的方式修改XML的能力。

1. 在初始化完毕后进行配置

有时候在Spring的ApplicationContext被创建以后做一些运行时配置是有意义的，这种情况下，你可以定义 doWithApplicationContext 闭 包，如下例：

class SimplePlugin {

def name="simple"

def version = 1.1

def doWithApplicationContext = { appCtx ->

SessionFactory sf = appCtx.getBean("sessionFactory")

// do something here with session factory

}

}

#### 运行时添加动态方法

1. 基础知识

Grails插件允许在运行时注册Grails管辖类或者其他类的动态方法，但新的方法只能通过 doWithDynamicMethods 闭包来增加。

对Grails管辖类来说，比如controllers、tag libraries 等等，可以增加方法，构造函数等，这是通过 ExpandoMetaClass 机制做到的，比如访问每个控制器的MetaClass的代码如下所示：

class ExamplePlugin {

def doWithDynamicMethods = { applicationContext ->

application.controllerClasses.metaClass.each { metaClass ->

metaClass.myNewMethod = {-> println "hello world" }

}

}

}

此处通过隐含的application对象来获取所有控制器类的MetaClass实例，并且为每一个控制器增加一个myNewMethod的方法。

或者，已经知道要处理的类的类型，那只需要在此类的 metaClass 属性上增加一个方法即可。例如，添加一个新的方法的swapCase到java.lang.String中：

class ExamplePlugin {

def doWithDynamicMethods = { applicationContext ->

String.metaClass.swapCase = {->

def sb = new StringBuffer()

delegate.each {

sb << (Character.isUpperCase(it as char) ?

Character.toLowerCase(it as char) :

Character.toUpperCase(it as char))

}

sb.toString()

}

assert "UpAndDown" == "uPaNDdOWN".swapCase()

}

}

跟ApplicationContext交互

doWithDynamicMethods闭包的参数是Spring的ApplicationContext实例，允许和该应用上下文实例中的对象进行交互。比如实现一个跟Hibernate交互的方法，可以联合着HibernateTemplate来使用SessionFactory实例，代码如下：

import org.springframework.orm.hibernate3.HibernateTemplate

class ExampleHibernatePlugin {

def doWithDynamicMethods = { applicationContext ->

application.domainClasses.metaClass.each { metaClass ->

metaClass.static.load = { Long id->

def sf = applicationContext.sessionFactory

def template = new HibernateTemplate(sf)

template.load(delegate, id)

}

}

}

}

另外因为Spring容器具有自动装配和依赖注入的能力，可以在运行时实现更强大的动态构造器，此构造器使用applicationContext来装配对象及其依赖，例：通过查找 Spring 的原型 beans（prototyped beans）来替代缺省的构造器

class MyConstructorPlugin {

def doWithDynamicMethods = { applicationContext ->

application.domainClasses.each { domainClass ->

domainClass.metaClass.constructor = {->

return applicationContext.getBean(domainClass.name)

}

}

}

}

#### 参与自动装载

1. 监控资源的改变

当资源发生改变的时候，监控并且重新加载这些变化是非常有意义的。这也是Grails为什么要在运行时实现复杂的应用程序重新加载。查看如下Grails的ServicesPlugin的一段简单的代码片段：

class ServicesGrailsPlugin {

…

def watchedResources = "file:./grails-app/services/\*Service.groovy"

…

def onChange = { event ->

if(event.source) {

def serviceClass = application.addServiceClass(event.source)

def serviceName = "${serviceClass.propertyName}"

def beans = beans {

"$serviceName"(serviceClass.getClazz()) { bean ->

bean.autowire = true

}

if(event.ctx) {

event.ctx.registerBeanDefinition(serviceName,

beans.getBeanDefinition(serviceName))

}

}

}

}

首先定义了watchedResources集合，此集合可能是String或者String的List，包含着要监控的资源的引用或者模式。如果要监控的资源是Groovy文件，那当它被改变的时候，此文件将会自动被重新加载，而且被传给onChange闭包的参数event。

event.source -该事件的来源，无论是重装的类或Spring的资源

event.ctx - Spring的ApplicationContext的实例

event.plugin -用于管理资源的插件对象（通常是这个）

event.application -本的grailsApplication实例

event.manager -该GrailsPluginManager实例

通过这些对象，你可以评估这些惯例,而且基于这些惯例你可以将这些变化适当的应用到ApplicationContext中。在上述的"Services"示例中， 当一个 service类变化时，一个新的 service 类被重新注册到 ApplicationContext 中。

1. 影响其他插件

当一个插件变化时，插件不但 要有相应地反应，而且有时还会“影响”另外的插件。

以 Services 和 Controllers 插件为例，当一个 service 被重新加载的时候， 除非也重新加载 controllers，否则将加载过的 service 自动装配到旧的controller 类的时候，将会发生问题。为了避免这种情况发生，可以指定将要受到 “影响”的另外一个插件，这意味着当一个插件监测到改变的时候，它将先重新加载自身，然后重新加载它所影

响到的所有插件。看 ServicesGrailsPlugin 的代码片段：

def influences = ['controllers']

观察其他插件

如果你想观察一个特殊的插件 的变化但又不需要监视插件的资源，那你可以使用"observe"属性：

def observe = ["hibernate"]

在此示例中，当一个 Hibernate 的领域类变化的时候，你将收到从 hibernate

插件传递过来的事件。

#### 插件的加载顺序

1. 控制插件依赖性

插件经常依赖于其他插件的存在，并可以根据他人的存在适应。这两个属性来实现。首先是所谓的dependsOn。例如，如下片段来自于Hibernate的插件：

class HibernateGrailsPlugin {

def version = 1.0

def dependsOn = [dataSource:1.0,

domainClass:1.0,

i18n:1.0,

core: 1.0]

}

上述示例所演示的，Hibernate插件依赖于4个插件： dataSource, domainClass, i18n,以及 core。

依赖将Hibernate的插件之前被加载，如果所有的依赖关系不加载，那么这个插件将不会加载。

该dependsOn属性还支持指定版本范围一个小型的表达语言。语法的几个例子可以看到如下：

def dependsOn = [foo: "\* > 1.0"]

def dependsOn = [foo: "1.0 > 1.1"]

def dependsOn = [foo: "1.0 > \*"]

1. 控制加载顺序

如果所依赖的插件不能被解析的话，则依赖于此的插件将被放弃并且不会被加载，这就是所谓的“强”依赖。然而我们可以 通过使用 loadAfter 来定义一个“弱”依赖，示例如下：

def loadAfter = ['controllers']

此处如果 controllers 插件存在的话，插件将在 controllers 之后被加载，否则的话将被单独加载。插件也可以适应于其他已存在的插件，以 Hibernate 插件的doWithSpring 闭包代码为例：

if(manager?.hasGrailsPlugin("controllers")) {

openSessionInViewInterceptor(OpenSessionInViewInterceptor) {

flushMode = HibernateAccessor.FLUSH\_MANUAL

sessionFactory = sessionFactory

}

grailsUrlHandlerMapping.interceptors <<

openSessionInViewInterceptor

}

controllers插件如果被加载的话，Hibernate 插件仅仅注册一个OpenSessionInViewInterceptor。 变量 manager 是 GrailsPluginManager 接口的一个实例， 并且提供同其他插件交互的方法，而且 GrailsPluginManager@ 本身存在与任何一个插件中。

#### 制作一个插件

例：创建一个插件MyPlugin，执行每个controller时候输出 “HelloWord”

1. 创建MyPlugin插件

grails create-plugin MyPlugin

1. 添加业务

在MyPlugin项目下的MyPluginGrailsPlugin.groovy文件中添加逻辑代码，即在doWithDynamicMethods方法中添加如下代码：

def doWithDynamicMethods = { applicationContext ->

application.controllerClasses.metaClass.each { metaClass ->

metaClass.myNewMethod = {->

println " HelloWord "

}

}

}

1. 安装插件

进入项目MyPlugin目录中运行Maven的安装命令

grails maven-install

1. 添加插件依赖

在项目中的 \grails-app\conf\ BuildConfig.groovy 文件中添加插件的依赖

plugins {

compile ":my-plugin:0.1"

}

可以在插件项目的\target\ pom.xml文件中查看artifactId

<artifactId>my-plugin</artifactId>

1. 使用插件

可以直接在controller方法中使用myNewMethod()

def index() {

myNewMethod()

}

当掉用index时候 控制台就会打印出 “HelloWord”

#### 插件中的模板

如果我们需要在自定义插件中使用模板，来构建自己的controller及view，那么按照如下步骤操作：

1. 创建完成插件项目后打开conf下的BuildConfig.groovy。在plugin中加入：

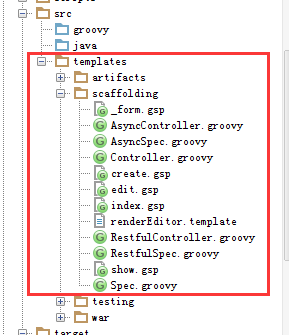
plugins {

compile ":scaffolding:2.1.2"

...

}

1. 在命令窗口输入：grails install-template执行完后，会在src下生成脚手架模板，我们可以根据自己的需求在模板中进行修改：



1. 在项目中引入插件：

打开conf下的BuildConfig.groovy在文件最下方，通过引入当前项目目录下插件的方式进行引入，如：

grails.plugin.location.'my-test-plugin' = "../MyTestPlugin"

‘my-test-plugin’可在在插件项目的\target\ pom.xml文件中查看artifactId，等号后面是插件的相对文件目录。

## CRUD基本功能开发

本章节使用Grails指令来完成一个CRUD（增删改查）基本功能，包括以下几个步骤：

1. 创建项目
2. 配置数据库链接
3. 创建域类
4. 创建controller和view
5. 运行程序

说明：

1. 为方便操作，首先在C盘根目录下创建example文件夹，本章节的项目将被创建在该目录下。
2. 命名规范：

1>项目名称首字母小写，项目路径不要出现中文或空格。

2>域类名称，第一个字母必需大写，域类名称不能重名，即使包名称不一样，类名称相同，这种情况也是不行。

3>服务类的文件名与类的名称必需一致，而且必须以Service结尾，不然在grails里无法正确识别。

1. 指令介绍：

创建项目 grails create-app

编译项目 grails compile

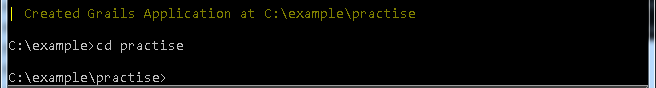
运行项目 grails run-app

打war包 grails grails war

### 创建项目

打开“命令提示符”，进入到C:\example中，输入grails create-app practise

等待创建成功，创建完成后进入到practise项目中：cd practise



### 配置数据库连接

#### 加载驱动

在网站上下载连库驱动包mysql-connector-java-5.1.30-bin.jar，加入到项目的lib文件夹下。（下载地址：<http://download.csdn.net/download/stc8902/7176105>）

#### 修改配置

进入C:\example\song\grails-app\conf\DataSource.groovy文件中，修改以下配置：

dataSource {

pooled = true

driverClassName = "com.mysql.jdbc.Driver"

username = "root"

password = "root"

}

hibernate {

cache.use\_second\_level\_cache = true

cache.use\_query\_cache = true

cache.region.factory\_class = 'org.hibernate.cache.ehcache.SingletonEhCacheRegionFactory' // Hibernate 4

singleSession = true

flush.mode = 'manual'

}

environments {

development {

dataSource {

dbCreate = "update"

url = "jdbc:mysql://localhost:3306/Practise\_dev"

}

}

test {

dataSource {

dbCreate = "update"

url = "jdbc:mysql://localhost:3306/Practise\_test"

}

}

production {

dataSource {

dbCreate = "update"

url = "jdbc:mysql://localhost:3306/Practise\_prod"

properties {

…

}

}

}

}

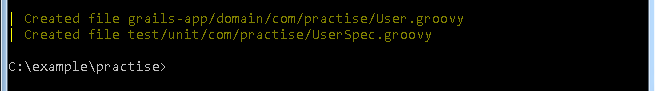
**说明：**

1. 在dataSource 中可加入“loggingSql = true”，这样在程序执行时可查看到所执行的sql语句。
2. dbCreate在第二章配置MySql数据库中有详细介绍，这里采用“update”
3. 首次连库会报错，是由于没有创建表空间，（在开发中我们会创建三个表空间，分别对应开发环境、测试环境和生产环境，这里我们只需要开发环境）所以先在数据库中创建表空间：CREATE DATABASE Practise\_dev;、CREATE DATABASE Practise\_test;、CREATE DATABASE Practise\_prod;。

### 创建域类

#### 创建User域类

回到“命令提示符”，在song项目中输入：grails create-domain-class com.practise.User，这时会在grails-appdomain\com\practise文件夹下生成User.groovy文件。



#### 编辑User域类

创建成功后进入C:\example\practise\grails-app\domain\com\practise打开User.groovy文件，编辑如下内容：

package com.practise

import org.grails.databinding.BindingFormat

class User {

String id

String username

String passwordHash

String address

@BindingFormat('yyyy-MM-dd HH:mm:ss')

Date dateCreated//这是一个特殊名字，Grails将自动在第一次向数据库保存实例的时候填上这个值

@BindingFormat('yyyy-MM-dd HH:mm:ss')

Date lastUpdated//同上，Grails将在每次把更新后的记录存回数据库的时候填充这个日期

//对于字段的一些约定

static constraints = {

//字符、数字或下划线开头，点号只能分隔字符或数字，其他内容可由字符、数字、下划线、减号组成，但必须以字符、数字或下划线结束。

id bindable: true, matches: /^[\w\d\_]+(?:(?:[\w\d].[\w\d])?[\w\d\_-]+[\w\d\_])\*/

// \/:\*?"<>|以及控制字符不能作为名称

username nullable: false, blank: false, matches: /[^\\\/:\*?"<>|\x00-\x1F\x7F]+/

address nullable: true

}

//对于数据库表的映射约定

static mapping = {

table "Users"

//主键生成策略

id generator: 'uuid'

version false

}

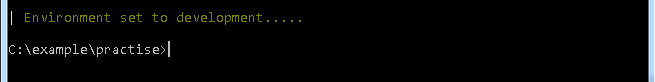
}

**说明：**

1. 对于字段是否为必输项需要在constraints中设置，默认为必输项。
2. 在mapping中我们可以定义表名、列名以及是否需要version字段等

#### 编译项目

回到“命令提示符”，在song项目中输入：grails compile

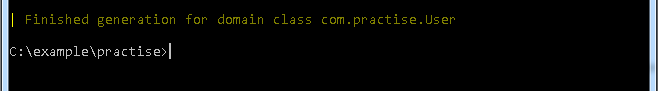


若出现图片内容则说明编译通过，否则根据报错内容进行改正。

### 创建控制器

在“命令提示符”中继续输入：grails generate-controller com.practise.User，这时

会在grails-app\controllers\com\practise下生成UserController.groovy文件。



**说明：**

在生成的UserController.groovy文件中，主要包含了6个action和一个notFound方法

默认的action：index，当我们进入到页面时，首先会进入到index页面

def index(Integer max) {

params.max = Math.min(max ?: 10, 100)

respond User.list(params), model:[userInstanceCount: User.count()]

}

Grails会帮我们自动生成index中的内容，现在目前的功能是用来做分页功能。

下面show action是响应index页面选中的行，将其内容返回到show页面。

def show(User userInstance) {

respond userInstance

}

在create中将页面获取的参数来新建一个User

def create() {

respond new User(params)

}

下面将执行保存操作，第一行@Transactional指明该操作是事务的。前两个判断是在判断数据是否为空和字段是否满足约束条件，若符合条件将执行保存操作（userInstance.save），并通过flash.message在页面输出提示信息，其中显示的信息内容参照上一章国际化来理解。

@Transactional

def save(User userInstance) {

if (userInstance == null) {

notFound()

return

}

if (userInstance.hasErrors()) {

respond userInstance.errors, view:'create'

return

}

userInstance.save flush:true

request.withFormat {

form multipartForm {

flash.message = message(code: 'default.created.message', args: [message(code: 'user.label', default: 'User'), userInstance.id])

redirect userInstance

}

'\*' { respond userInstance, [status: CREATED] }

}

}

点击编辑按钮时，首先会从页面获取当前行信息，然后修改信息执行update操作，其方法与save类似，不再描述。

def edit(User userInstance) {

respond userInstance

}

删除操作，同样是获取当前行，通过userInstance.delete执行删除，并在页面显示相应信息。

@Transactional

def delete(User userInstance) {

if (userInstance == null) {

notFound()

return

}

userInstance.delete flush:true

request.withFormat {

form multipartForm {

flash.message = message(code: 'default.deleted.message', args: [message(code: 'User.label', default: 'User'), userInstance.id])

redirect action:"index", method:"GET"

}

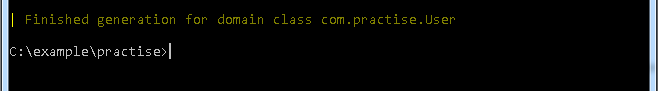
'\*'{ render status: NO\_CONTENT }

}

}

### 创建视图

在“命令提示符”中继续输入：grails generate-views com.practise.User，这时会在grails-app\views\user下生成五个gsp页面分别为：模板页面\_form.gsp、主页index.gsp、新建页面create.gsp、编辑页面edit.gsp、详细信息页面show.gsp，这时已经完成一个简单的增删改查功能。



说明：

新建和编辑页面都调用了模板页面：

<fieldset class="form">

<g:render template="form"/>

</fieldset>

下面以模板页面的passwordHash字段进行简要说：

在div中来捕获错误信息，required表明此字段是必输字段；

label中，messsge用来显示字段名称，在国际化一节中已进行说明，span用来输出必输项的“\*”号；

最后的textField中将required删除，在提交操作时仍提示该字段必输，所以要设置字段不必输，必须在域类中指定该字段不必输。

<div class="fieldcontain ${hasErrors(bean: userInstance, field: 'passwordHash', 'error')} required">

<label for="passwordHash">

<g:message code="user.passwordHash.label" default="Password Hash" />

<span class="required-indicator">\*</span>

</label>

<g:textField name="passwordHash" required="" value="${userInstance?.passwordHash}"/>

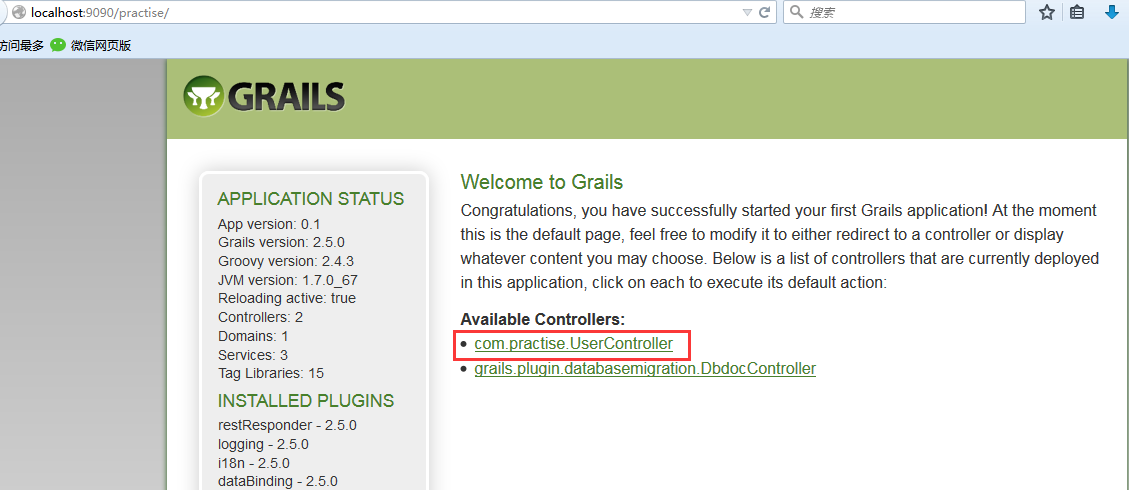
</div>

### 运行程序

在“命令提示符”中继续输入：grails run-app，若提示端口被占用，输入grails -Dserver.port=9090 run-app，更改一个端口运行程序



打开浏览器输入<http://localhost:9090/practise>，点击com.practise.UserController，进入页面。



## Master-Detail功能开发

本章节说明在IDEA中的master-Detail功能开发。下面以菜单功能进行说明：

### 创建Domain

创建Menu Domain Class：

class Menu {

String code

String name

String description

String statusCode = 'ENABLED'

Integer createdBy

Integer updatedBy

Date dateCreated

Date lastUpdated

static hasMany = [menuEntities: MenuEntry]

static constraints = {

code nullable: false, unique: true, size: 3..20

name nullable: false

description nullable:true, maxSize: 500

}

static mappedBy = [menuEntities: "menu"]

static mapping = {

table 'fwk\_menus'

}

}

创建MenuEntry Domain Class：

class MenuEntry {

String name

String description

Menu subMenu

Function subFunctionGroup

Integer displayOrder

String statusCode = 'ENABLED'

Integer createdBy

Integer updatedBy

Date dateCreated

Date lastUpdated

static belongsTo =[menu:Menu]

static constraints = {

name nullable: false

description nullable:true, maxSize: 500

displayOrder nullable:true

subMenu nullable:true

subFunctionGroup nullable:true

}

static mapping = {

table 'fwk\_menu\_entries'

}

}

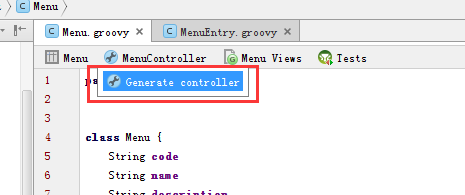
说明：

1. 在Menu中定义了static hasMany = [menuEntities: MenuEntry]，确保menu和menuEntry的一对多关系。
2. static mappedBy = [menuEntities: "menu"]确保级联显示，根据menu\_id查询出detail信息
3. 在MenuEntry定义了static belongsTo =[menus:Menu]，确保级联删除

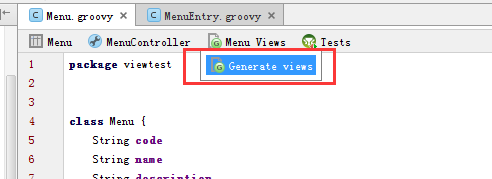
### 创建Master的controller、view

#### 创建controller、view

创建MenuController：在Menu.groovy中点击Generate controller，生成MenuController.groovy：



再点击Generate views生成页面，



#### 页面调整

1. 添加i18n语言描述，如下，其中对于domain中的验证信息，参照6.1错误消息（i18n）进行添加。

menu.label = 菜单

menu.code.blank=编码不能为空！

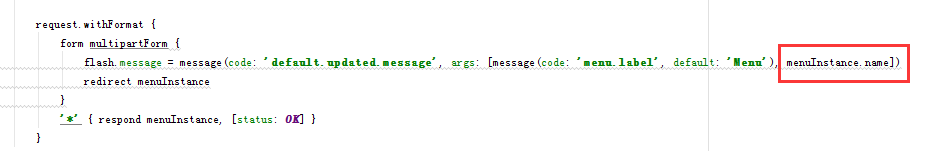
menu.code.unique=编码必须唯一！

menu.code.size.toosmall =编码长度过小！

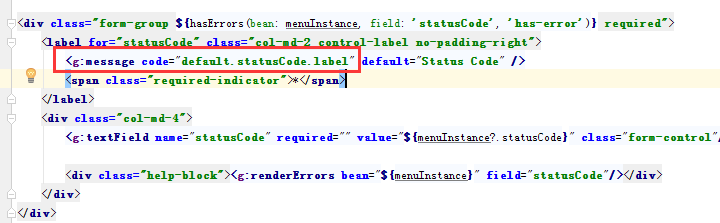
menu.code.size.toobig =编码长度过大！

menu.name.blank=名称不能为空！

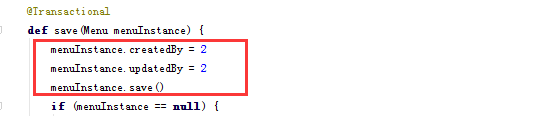
修改controller中提示信息传递的参数，如menuInstance.id改为menuInstance.name：



1. 页面一些公用的名称都使用default的，如编码、名称、描述等，详细参照项目中的i18n中的定义。



1. 删除页面中的createBy、updateBy字段，通过在controller中赋值。



### 创建Detail的controller、view

#### 创建controller、view

同创建Menu的controller、view方法一样：创建MenuEntryController：在MenuEntry.groovy中点击Generate controller，生成MenuEntryController.groovy；再点击Generate views生成页面。

#### 页面调整

1. 多语言的调节参照上一节。
2. 添加菜单项列表：

打开menu的show页面，在基本信息的table下添加<br>，再将如下代码添加到br下：

<div class="widget-header bordered-bottom bordered-blue">

<span class="widget-caption">${message(code: 'entries.list.label',default: 'Entries List')}</span>

<g:link class="btn btn btn-blue" controller="menuEntry" action="create" params="['menu.id': menuInstance?.id]"><g:message code="default.button.create.label" default="Create" /></g:link>

</div>

<div class="widget-body">

<!—此处预留添加detail的table-->

</div>

复制menuEntry的index页面的table，粘贴到预留的空白处（其中新建的link的controller、action、params需要修改）

1. 调整页面，保留需要显示的字段

注意：有下拉选择的，需要在select标签添加optionValue="name"，用来显示id对应的名称。

1. 修改detail页面返回链接，（menuEntry下的create、edit、show页面返回链接）将原有link中的的action替换为：

controller="menu" action="show" id ="${menuEntryInstance?.menu?.id}"

其中需要修改controller、id。

## 插件应用

本章节重点介绍在开发Hitsm系统时，有关插件的开发或使用的技术点总结。

### 使用database-migration插件

1. 在BuidConfig.groovy配置文件中添加插件依赖

plugins {

runtime ":database-migration:1.4.0"

}

注意：在使用此插件时，将DataSource.groovy 中 dbCreate = "update"去掉

1. 创建数据库脚本文件（默认名称changelog.groovy）在项目根目录下

grails dbm-generate-changelog changelog.groovy

默认创建文件的路径为 grails-app/migrations

可以手动创建，文件格式为

databaseChangeLog = {

}

1. 创建一张表

databaseChangeLog = {

changeSet(author: "jinxin", id: "create\_table\_school") {

createTable(tableName: "school") {

column(autoIncrement: "true", name: "id", type: "bigint") {

constraints(nullable: "false", primaryKey: "true", primaryKeyName: "schoolPK")

}

column(name: "version", type: "bigint") {

constraints(nullable: "false")

}

column(name: "schoolname", type: "varchar(100)") {

constraints(nullable: "false")

}

}

}

}

1. 使用如下命令更新数据库

grails dbm-update

1. 添加子文件

使用命令创建子文件2015-4-12-initial-database.groovy，文件会自动包含到changelog.groovy文件中

grails dbm-gorm-diff 2015-4-12-initial-database.groovy --add

手动创建子文件2015-4-12-initial-database.groovy 内容如下：

databaseChangeLog = {

changeSet(author: "jinxin", id: "create\_table\_school") {

createTable(tableName: "school") {

column(autoIncrement: "true", name: "id", type: "bigint") {

constraints(nullable: "false", primaryKey: "true", primaryKeyName: "schoolPK")

}

column(name: "version", type: "bigint") {

constraints(nullable: "false")

}

column(name: "schoolname", type: "varchar(100)") {

constraints(nullable: "false")

}

}

}

}

在changelog.groovy文件中添加如下语句

databaseChangeLog = {

include file: '2015-4-12-initial-database.groovy '

}

1. 配置数据脚本文件路径，在grails-app/conf/Config.groovy文件中添加

grails.plugin.databasemigration.changelogLocation = "grails-app/migrations"

1. 配置数据脚本文件名称，在grails-app/conf/Config.groovy文件中添加

grails.plugin.databasemigration.changelogFileName = "changelog.groovy"

1. 配置数据脚本在运行项目时自动运行，在grails-app/conf/Config.groovy文件中添加

grails.plugin.databasemigration.updateOnStart = true

grails.plugin.databasemigration.updateOnStartFileNames =['changelog.groovy']

1. 初始化数据

在changelog.groovy文件中添加

databaseChangeLog = {

include file: 'data.groovy'

}

同文件夹中创建data.groovy文件，内容如下

changeSet(id: "20140611\_2\_init\_data", author: "jinxin", runAlways: true) {

comment('Init Languages data')

loadData(encoding: "UTF-8", tableName: 'book', file: (new File(System.getProperty("user.dir") +"/grails-app/migrations/users.csv")).path) {

column(name: 'id', type: "NUMERIC")

column(name: 'version', type: "STRING")

column(name: 'name', type: "STRING")

column(name: age, type: "STRING")

column(name: 'address', type: "STRING")

}

}

同文件夹中创建users.csv 内容如下：

id,version,name,age,address

1,1,应用管理服务中心,2,11

1. 需要更多的配置可参照文档

<http://grails-plugins.github.io/grails-database-migration/docs/manual/guide/single.pdf>

## 技术点总结

本章节重点介绍在开发Hitsm系统时遇到的一些技术点的总结和使用,这些功能在开发其他功能模块也会用到。

### 错误消息（i18n）

在开发中，我们在domain中设置字段的一些验证规则，如唯一验证：

class Menu {

String code

...

static constraints = {

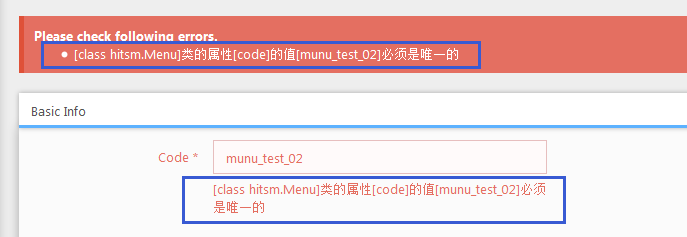
code nullable:false, unique: true, size: 3..30

...

}

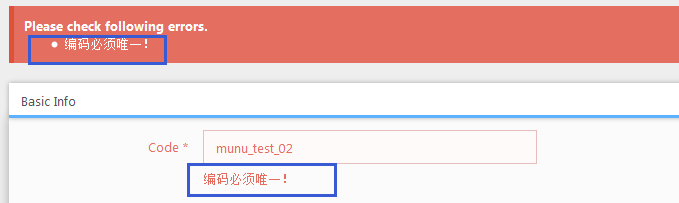
}

在页面显示这些规则给出的并不是我们想要的提示信息，如下唯一的验证信息：



这时，我们只需要在i18n文件夹下的messages\_zh\_CN.properties文件中添加：

menu.code.unique=编码必须唯一！



同样可对其它语言进行设置。

对于其它验证规则的提示信息，按照下表“**错误代码**”进行修改即可：

| 名称 | 示例 | 描述 | | | 错误代码 |
| --- | --- | --- | --- | --- | --- |
|  |  | |  | | |
| blank | login(blank:false) | 验证String值是否为空 | | className.propertyName.blank | |
| creditCard | cardNumber(creditCard:true) | 信用卡号格式验证 | | className.propertyName. creditCard.invalid | |
| email | homeEmail(email:true) | Email格式 | | className.propertyName.email.invalid | |
| inList | login(inList:['Joe', 'Fred']) | 必须在给出的值选 | | className.propertyName.not.inList | |
| min | duration(min:1) | 设置最小值 | | className.propertyName.min.notmet | |
| max | age(max:99) | 设置最大值 | | className.propertyName.max.exceeded | |
| minSize | children(minSize:5) | 设置集合或数字属性最小值 | | className.propertyName.minSize.exceeded | |
| maxSize | children(maxSize:25) | 设置集合或数字属性最大值 | | className.propertyName.maxSize.exceeded | |
| matches | login(matches:/[a-zA-Z]/) | 正则表达式验证 | | className.propertyName.matches.invalid | |
| notEqual | login(notEqual:'Bob') | 设置不能等于 | | className.propertyName.notEqual | |
| nullable | age(nullable:false) | 设置不能为空 | | className.propertyName.nullable | |
| range | age(range:16..59) | 设置一个常规的有效值范围 | | className.propertyName.range.toosmall  className.propertyName.range.toobig | |
| scale | salary(scale:2) | 设置小数点范围 | | N/A | |
| size | children(size:5..15) | 限制一个集合或数组的大小 | | className.propertyName.size.toosmall  className.propertyName.size.toobig | |
| unique | login(unique:true) | 必须唯一 | | className.propertyName.unique | |
| url | homePage(url:true) | 如果为true则显示为链接 | | className.propertyName.url.invalid | |