Table of Contents

1. 介绍.....11
2. 入门.....12
  2.1 下载和安装.....12
  2.2 从 Grails 1.0.x 升级.....12
    Groovy 1.6.....12
    Java 5.0.....13
    配置的变化.....13
    插件的变化.....13
    脚本的变化.....13
    命令行的变化.....14
    数据映射的变化.....14
    REST 支持.....15
  2.3 创建一个应用程序.....15
  2.4 一个 Hello World 例子.....15
  2.5 设置 IDE.....16
      IntelliJ IDEA.....16
      NetBeans 16
      Eclipse.....16
      TextMate.....17
  2.6 约定优于配置.....17
  2.7 运行应用程序.....17
  2.8 测试应用程序.....18
  2.9 部署应用程序.....18
  2.10 创建工件.....19
  2.10 支持的 Java EE 容器.....19
  2.11 生成一个应用.....20
3. 配置.....20
  3.1 基本配置.....20
  3.1.1 内置选项.....21
    生成 War 文件.....21
  3.1.2 日志.....22
      日志基础.....22
      顶级日志记录器.....22
      自定义输出目标.....23
      自定义布局.....24
      完整的堆栈日志跟踪.....25
      约定的日志记录方式.....25
  3.2 环境.....26
      多环境配置.....26
      针对不同环境的打包和运行.....27
      可编程的环境检测.....27
  3.3 数据源.....27
  3.3.1 数据源和环境.....29

3.3.2 JNDI 数据源.....29
3.3.3 自定数据库迁移.....29
3.4 外部配置.....30
3.5 版本管理.....31
版本管理基础.....31
运行期间检测版本.....31
4. 命令行.....31
4.1 创建 Gant 脚本.....33
默认任务（default）.....33
4.2 重用 Grails 脚本.....34
从其他脚本文件引入任务.....34
核心的 Grails 任务.....35
脚本结构.....36
4.3 钩子事件.....36
定义事件处理器.....37
触发事件.....38
公共事件.....38
4.4 自定义构建.....39
默认.....39
覆盖默认值.....40
可用的构建设置.....41
4.5 Ant 和 Maven.....41
Ant 集成.....42
Maven 集成.....43
准备.....43
创建一个 Grails Maven 项目.....44
给现有项目加入 Maven 支持.....45
添加 Grails 命令到 phase 中.....45
5. 对象关联映射 (GORM).....46
5.1 快速入门指南.....47
5.1.1 CRUD 基础.....47
Create.....47
Read.....48
Update.....48
Delete.....48
5.2 GORM 中进行 Domain 建模 48
5.2.1 GORM 中的关联.....49
5.2.1.1 One-to-one.....49
Example A.....49
Example B.....50
Example C.....50
5.2.1.2 One-to-many.....51
5.2.1.3 Many-to-many.....53
5.2.1.4 集合类型基础.....54
5.2.2 GORM 中的组合.....54

5.2.3 GORM 中的继承.....55
注意事项.....55
多态性查询.....56
5.2.4 Sets, Lists 和 Maps.....56
Sets 对象.....56
List 对象.....57
映射(Maps)对象.....57
集合类型和性能.....58
5.3 持久化基础.....59
5.3.1 保存和更新.....59
5.3.2 删除对象.....60
5.3.3 级联更新和删除.....60
设置了 belongsTo 的双向一对多.....61
单向一对多.....62
没有设置 belongsTo 的双向一对多.....62
设置了 belongsTo 的单向一对一.....62
5.3.4 立即加载和延迟加载.....62
配置立即加载.....63
使用批量加载 Using Batch Fetching.....63
5.3.5 悲观锁和乐观锁.....64
乐观锁.....64
悲观锁.....65
5.4 GORM 查询.....66
获取实例列表.....66
根据数据库标识符取回.....66
5.4.1 动态查询器.....66
方法表达式.....67
布尔逻辑(AND/OR).....68
查询关联.....69
分页和排序.....69
5.4.2 条件查询.....69
逻辑与（Conjunctions）和逻辑或（Disjunctions）.....70
查询关联.....70
投影(Projections)查询.....71
使用可滚动的结果.....71
在 Criteria 实例中设置属性.....72
立即加载的方式查询.....72
方法引用.....73
5.4.3 Hibernate 查询语言(HQL).....73
位置和命名参数.....73
多行查询.....74
分页和排序.....74
5.5 高级 GORM 特性.....74
5.5.1 事件和自动实现时间戳.....74

事件类型.....75
beforeInsert 事件.....75
beforeUpdate 事件.....75
beforeDelete 事件.....75
onLoad 事件.....76
自动时间戳.....76
5.5.2 自定义 ORM 映射.....76
5.5.2.1 表名和列名.....77
表名.....77
列名.....77
列类型.....77
一对一映射.....78
一对多映射.....79
多对多映射.....79
5.5.2.2 缓存策略.....80
设置缓存.....80
缓存实例.....81
缓存关联对象.....81
Caching Queries.....82
缓存用法.....82
5.5.2.3 继承策略.....83
5.5.2.4 自定义数据库标识符.....83
5.5.2.5 复合主键.....84
5.5.2.6 数据库索引.....85
5.5.2.7 乐观锁和版本定义.....85
5.5.2.8 立即加载和延迟加载.....86
延迟加载集合.....86
延迟加载单向关联.....86
5.5.2.9 自定义级联行为.....87
5.5.2.10 自定义 Hibernate 的类型.....88
5.5.3 缺省排序.....89
5.6 事务编程.....90
5.7 GORM 和约束.....91
影响字符串类型属性的约束.....92
影响数值类型属性的约束.....92
6.Web 层.....94
6.1 控制器(Controllers).....94
6.1.1 理解控制器(Controller)与操作(Action).....94
创建控制器(Controller).....94
创建操作(Action).....94
默认 Action.....95
6.1.2 控制器(Controller) 与作用域.....95
可用的作用域.....95
存取作用域.....96

使用 Flash 作用域.....96
6.1.3 Models(模型)与 Views(视图).....97
Returning the Model.....97
选择 View.....98
渲染响应.....98
6.1.4 重定向与链接.....100
Redirects.....100
6.1.5 Controller(控制器) 拦截器.....102
Before 拦截器.....102
After 拦截器.....103
拦截条件.....103
6.1.6 数据绑定.....104
绑定 Request 数据到 Model 上.....104
数据绑定和单向关联.....105
属于绑定与 Many-ended 关联.....105
数据绑定多个 domain 类.....106
数据绑定与类型转换错误.....106
数据绑定与安全关系.....107
6.1.7 XML 与 JSON 响应.....108
使用 render 方法输出 XML.....108
使用 render 方法输出 JSON.....109
自动 XML 列集(Marshalling).....109
自动 JSON 列集(Marshalling).....110
6.1.8 文件上传.....111
文件上传程序.....111
通过数据绑定上传文件.....111
6.1.9 命令对象.....112
声明命令对象.....112
使用命令对象.....112
命令对象与依赖注入.....113
6.1.10 处理重复的表单提交.....113
6.2 Groovy Server Pages.....114
6.2.1 GSP 基础.....115
6.2.1.1 变量与作用域.....115
6.2.1.2 逻辑和迭代.....116
6.2.1.3 页面指令.....117
6.2.1.4 表达式.....117
6.2.2 GSP 标签.....117
6.2.2.1 变量与作用域.....118
6.2.2.2 逻辑和迭代.....119
6.2.2.3 搜索和过滤.....119
6.2.2.4 链接和资源.....120
6.2.2.5 表单和字段.....121
表单基础.....121
表单字段.....121

多样的提交按钮.....121
6.2.2.6 标签作为方法调用.....122
来自 GSPs 中的标签当作方法调用.....122
来自控制器（Controllers）和标签库的标签作为方法调用.....122
6.2.3 视图(View)与模板(Templates).....122
模板基础.....123
共享模板.....123
模板命名空间.....123
在控制器(Controllers)和标签库中的模板.....124
6.2.4 使用 Sitemesh 布局 124
创建布局.....124
启用布局.....125
在控制器(Controller)中指定布局.....125
布局规约.....126
内联布局.....126
Server-Side 包含.....127
6.2.5 Sitemesh 内容块.....127
6.3 标签库.....128
6.3.1 变量与作用域 129
6.3.2 简单标签.....129
6.3.3 逻辑标签.....130
6.3.4 迭代标签.....131
6.3.5 标签命名空间.....132
6.3.6 使用 JSP 标签库.....132
6.4 URL 映射.....133
6.4.1 映射到控制器和操作.....133
6.4.2 嵌入式变量.....134
简单变量.....134
动态控制器(Controller)和操作(Action)名.....134
可选的变量.....135
任意变量.....135
动态解析变量.....135
6.4.3 映射到视图.....136
6.4.4 映射到响应代码.....136
6.4.5 映射到 HTTP 方法.....137
6.4.6 映射通配符.....137
6.4.7 自动重写链接.....138
6.4.8 应用约束.....138
6.5 Web 流(Flow).....139
概述.....139
创建流.....140
6.5.1 开始与结束状态.....140
6.5.2 操作(Action)状态和视图状态.....141
视图状态.....141

操作(Action)状态.....142
切换操作.....143
6.5.3 流(Flow)执行事件.....143
来自于一个视图状态的触发事件.....143
来自于一个操作(Action)的触发事件.....144
6.5.4 流(Flow)的作用域.....145
作用域基础.....145
流(Flow)的作用域和序列化.....146
6.5.5 数据绑定和验证.....147
6.5.6 子流程和会话.....148
6.6 过滤器.....149
6.6.1 应用过滤器.....149
6.6.2 过滤器(Filters)类型.....150
6.6.3 变量与作用域 151
6.7 Ajax.....152
6.7.1 用 Prototype 实现 Ajax.....152
6.7.1.1 远程链接.....152
6.7.1.2 内容更新.....153
6.7.1.3 远程表单提交 153
6.7.1.4 Ajax 事件.....154
6.7.2 用 Dojo 实现 Ajax.....154
6.7.3 用 GWT 实现 Ajax.....155
6.7.4 服务端的 Ajax.....155
内容为中心的 Ajax.....155
数据为中心的 Ajax 与 JSON.....156
数据为中心的 Ajax 与 XML.....156
脚本为中心的 Ajax 与 JavaScript.....157
6.8 内容协商.....157
配置 Mime 类型.....158
内容协商使用 Accept 报头.....158
内容协商与格式化请求参数.....159
内容协商与 URI 扩展.....159
测试内容协商.....160
7. 验证.....160
7.1 声明 Constraints(约束).....160
7.2 验证约束.....161
验证基础.....161
验证阶段.....162
7.3 客户端验证.....162
显示错误.....162
高亮错误.....163
取回输入值.....163
7.4 验证与国际化.....163
规约与 Message 编码.....164

显示消息.....164
7.5 验证非 Domain 与命令行对象.....165
Validateable 注解.....165
注册 Validateable 类.....166
8. Service 层.....166
创建 Service 166
8.1 声明式事务处理.....167
8.2 服务作用域.....167
8.3 依赖注入与服务.....168
依赖注入基础.....168
依赖注入与服务.....169
依赖注入与 Domain 类.....169
8.4 Using Services from Java.....169
9. 测试.....171
9.1 单元测试.....171
测试框架.....172
GrailsUnitTestCase - 模拟方法.....174
mockDomain(class, testInstances =).....175
mockForConstraintsTests(class, testInstances =).....176
mockLogging(class, enableDebug = false).....177
mockController(class).....177
mockTagLib(class).....178
9.2 集成测试.....178
测试控制器.....178
用应用测试控制器.....179
测试控制器 command 对象.....179
测试控制器和 render 方法.....180
模拟生成请求数据.....181
测试 Web Flows.....182
测试标签库.....183
使用 GroovyPagesTestCase 测试标签库.....184
测试 Domain 类.....185
9.3 功能测试.....185
10. 国际化.....186
10.1 理解消息绑定.....186
10.2 修改本地化.....186
10.3 读取信息.....187
视图中读取信息.....187
控制器和标签库中读取信息.....187
10.4 脚手架和 i18n.....188
11. 安全性.....188
Grails 可以自动做什么.....188
11.1 防止攻击.....188
SQL 注入.....188
钓鱼式攻击.....189

XSS-跨站脚本攻击.....189
HTML/URL 注入.....189
拒绝服务 DoS.....190
可推测 ID 号.....190
11.2 编码和解码对象.....190
编解码器类.....190
标准的编解码器.....191
定制编解码器 Custom Codecs.....193
11.3 认证.....193
11.4 安全插件.....194
11.4.1 Acegi.....194
11.4.2 JSecurity.....195
12. 插件.....195
12.1 创建和安装插件.....195
创建插件.....195
插件的安装和发布.....197
注意被排除的组件.....197
12.2 插件仓库.....198
在 Grails 插件的存储仓库（Repository）发布插件.....198
配置附加库.....198
12.3 理解插件的结构.....199
12.4 提供基础的工件.....200
增加新的脚本.....200
增加新的控制器，标签库或者服务.....200
Providing Views, Templates and View resolution.....201
Excluded Artefacts.....201
12.5 评估规约.....201
12.6 参与构建事件.....203
安装后进行配置和参与升级操作.....203
脚本事件.....203
12.7 运行时配置中的钩子 Hooking into Runtime Configuration.....203
跟 Grails 的 Spring 配置进行交互.....204
参与 web.xml 的生成.....204
在初始化完毕后进行配置.....205
12.8 运行时添加动态方法.....205
基础知识.....205
跟 ApplicationContext 交互.....206
12.9 参与自动重载.....207
监控资源的改变.....207
影响其他插件.....208
观察其他插件.....208
12.10 理解插件加载的顺序.....209
Controlling Plug-in Dependencies.....209
Controlling Load Order 210
13. Web 服务.....210

13.1 REST....210
URL 形式....211
XML 序列化 - 读取....211
XML 序列化 - 更新....212
13.2 SOAP....213
13.3 RSS 和 Atom....213
14. Grails 和 Spring....214
14.1 Grails 内部实现....214
Grails ApplicationContext....214
配置 Spring Beans....215
14.2 配置其他 Bean....215
使用 XML....215
引用现有的 Spring bean....215
使用 Spring DSL....216
14.3 运行时 Spring 与 Beans DSL....217
BeanBuilder 类....217
在 Spring MVC 中使用 BeanBuilder....218
从文件系统中加载 bean 定义....219
绑定变量....220
14.4 BeanBuilder DSL....220
使用构建器参数....220
配置 BeanDefinition (使用工厂方法)....220
使用工厂 bean（Factory beans）....221
运行时创建 bean 的引用....222
使用匿名内部 bean....222
抽象 bean 和父子 bean 定义....224
使用 Spring 命名空间....225
14.5 属性占位符配置....226
14.6 属性重载....227
15. Grails 与 Hibernate....227
15.1 通过 Hibernate 注解映射....228
15.2 进一步阅读....229
16. 脚手架....229
启动脚手架....230
动态脚手架....230
自定义生成的视图....231
生成控制器和视图....232
定制脚手架模板....233
17. 部署....233
"grails run-app"....233
"grails run-war"....233
WAR 文件....233
应用程序服务器....235

1. 介绍

当今的 Java Web 开发技术显得过于复杂，相对于它本身的需要来说。现在主流的 Java Web 框架也是异常复杂，而且没有很好的遵循 Don't Repeat Yourself (DRY) 法则。

因此我们要以一种新的思维方式来重新思考 Web 开发，Rails、Django 和 TurboGears 这样的动态框架给我们铺平了道路。Grails 建立在这些概念之上，它极大地降低了在 Java 平台上建立 Web 应用的复杂性。与那些框架不同的是，Grails 是构建在现有的像 Spring、Hibernate 这样的 Java 技术之上。

Grails 是个一栈式开发框架，它尝试通过核心技术和插件技术来解决许多 Web 开发难题。Grails 包含了如下内容：

- 由 Hibernate 构成的易于使用的 Object Relational Mapping (ORM)层
- 称为 Groovy Server Pages (GSP) 的展现层技术
- 基于 Spring MVC 的控制层
- 由基于 Groovy 的 Gant 工具构建的命令行脚本环境
- 一个内嵌的 Jetty 容器被配置用来快速重载应用
- Spring 容器内建的依赖注入技术
- 基于 Spring 的 MessageSource 核心概念的国际化 (i18n) 支持
- 基于 Spring 的抽象事务概念的事务服务层

所有这些都非常易于使用，这得益于 Groovy 语言的强大以及 Domain Specific Languages (DSLs) 的广泛使用。

本文档将带你从 Grails 入门开始，最终能够使用 Grails 框架建设 Web 应用程序。

2. 入门

2.1 下载和安装

让 Grails 运行起来的第一步是安装发行包。请按照如下步骤：

- 下载 Grails 的二进制发行包并解压到你指定的目录下
- 新增 GRAILS_HOME 环境变量并指向你解压发行包时选择的目录
    - Unix/Linux 系统上通常在你的 profile 文件中添加 export GRAILS_HOME=/path/to/grails 来设置环境变量

    - Windows 系统上则是在 我的电脑/属性/高级/环境变量 中添加相同的环境变量
- 现在需要添加 bin 目录到 PATH 环境变量中：
    - Unix/Linux 系统上在 profile 中继续添加 export PATH="$PATH:$GRAILS_HOME/bin"
    - Windows 系统上修改 我的电脑/属性/高级/环境变量 中的 Path 环境变量

如果 Grails 正常工作了那么你可以在终端窗口中键入 grails 命令并看到如下简单的输出：

```
Welcome to Grails 1.0 - http://grails.org/
Licensed under Apache Standard License 2.0
Grails home is set to: /Developer/grails-1.0
No script name specified. Use 'grails help' for more info
```

2.2 从 Grails 1.0.x 升级

尽管 Grails 开发团队试图将从 Grails 1.0.x 升级到 Grails 1.1 时带来的影响降到最低限度，但仍然有一些事项需要你认真考虑。重大变化描述如下。

Groovy 1.6

Grails 1.1 现在和 Groovy 1.6 协同工作并且不再支持针对 Groovy 1.5 的代码编译。如果你有一个使用 Groovy 1.5 编写的组件库，在将它用于 Grails 1.1 之前，你需要针对 Groovy 1.6 来重新编译它。

Java 5.0

Grails 1.1 现在不再支持 JDK 1.4，如果你希望正常使用 Grails，那么建议你继续使用 Grails 1.0.x 系列直到你能够升级你的 JDK。

配置的变化

1) 为了系统的一致性，设置项 grails.testing.reports.destDir 已经被重命名为 grails.project.test.reports.dir。

2) 下列设置已经从 grails-app/conf/Config.groovy 文件中移到了 grails-app/conf/BuildConfig.groovy 文件：

- grails.config.base.webXml
- grails.war.destFile
- grails.war.dependencies
- grails.war.copyToWebApp
- grails.war.resources

3) 自从 Java 5.0 成为基线起， grails.war.java5.dependencies 选项已不再被支持(见上文)。

4) jsessionid 的使用(现在被认为是有害的)默认是禁用的。如果你的应用程序需要用到 jsessionid，你可以重新启用它，在 grails-app/conf/Config.groovy 文件中添加如下设置：

```
grails.views.enable.jsessionid=true
```

5) 用来配置 Log4j 的语法已经改变了。看看用户指南的 日志 一章可以获得更多信息。

插件的变化

Grails 1.1 默认将不在你的 PROJECT_HOME/plugins 目录下储存插件。这可能导致你的应用程序出现编辑错误，解决办法是重新安装所有的插件或者在 grails-app/conf/BuildConfig.groovy 文件中设置下列属性：

```
grails.project.plugins.dir="./plugins"
```

脚本的变化

1) 如果你先前使用的是 Grails 1.0.3 或以下的版本，那么下边用于从 GRAILS_HOME 导入脚本的语法将不再被支持：

```
Ant.property(environment:"env")
grailsHome = Ant.antProject.properties."env.GRAILS_HOME"
includeTargets << new File ( "${grailsHome}/scripts/Bootstrap.groovy" )
```

取而代之的是 grailsScript 方法，它能导入一个命名的脚本：

```
includeTargets << grailsScript( "Bootstrap.groovy" )
```

2) 由于升级到了 Gant，所有对变量 Ant 的引用应该改为 ant。

3) 项目的根目录不再存于 classpath 中，像如下的资源加载方式将无法使用：

```
def stream = getClass().classLoader.getResourceAsStream("grails-app/conf/my-config.xml")
```

但是你可以使用 Java 文件 API 以及 basedir 属性来完成如上操作：

```
new File("${basedir}/grails-app/conf/my-config.xml").withInputStream { stream ->
    // read the file
}
```

命令行的变化

run-app-https 和 run-war-https 这两个命令已经被取消了，取而代之的是 run-app 命令带上特定的参数：

```
grails run-app -https
```

数据映射的变化

1) 枚举类型现在可以使用它们的 String 值来映射，而不再是 ordinal 值。当然你也可以通过如下方式改变你的映射来还原为旧的习惯：

```
static mapping = {
    someEnum enumType:"ordinal"
}
```

2) 双向的一对一关联现在可以使用在所有者端的一个单列和一个外键引用来映射。你不需要做任何修改，除非你想删除在相反端包含了重复数据的那一列。

REST 支持

接收到的 XML 请求现在不能被自动解析了。要启用对 REST 请求的解析，你需要在 URL 映射中使用 parseRequest 变量：

```
"/book"(controller:"book",parseRequest:true)
```

其次，你也可以使用新的 resource 变量来开启默认解析：

```
"/book"(resource:"book")
```

2.3 创建一个应用程序

要创建一个 Grails 应用程序你首先需要熟悉 grails 命令的使用，使用方式如下：

```
grails [命令名称]
```

假如你需要执行的命令是 create-app：

```
grails create-app helloworld
```

这将创建一个新的目录，其中包含了 helloworld 这个项目。你现在可以在终端里导航到这个目录：

```
cd helloworld
```

2.4 一个 Hello World 例子

要实现经典的"hello world!"例子你可以运行 create-controller 命令：

```
grails create-controller hello
```

这将在 grails-app/controllers 目录中创建一个名为 HelloController.groovy 的控制器（参见控制器一章获得更多内容）。

控制器能用来处理 web 请求并用来实现“hello world!”的例子，我们的实现代码如下：

```
class HelloController {
    def world = {
        render "Hello World!"
    }
}
```

完工。现在使用另一个称为 run-app 的新命令来启动容器：

```
grails run-app
```

这将在 8080 端口开启一个服务器，现在可以通过 http://localhost:8080/helloworld 这个 URL 来访问你的应用程序了。

你将看到如下截图所示的内容：

这是由 web-app/index.gsp 文件所呈现的 Grails 介绍页面。你会注意到它已经发现了你的控制器的存在，点击链接来访问控制器，我们可以看到浏览器窗口中打印除了“Hello World!”的文本。

2.5 设置 IDE

IntelliJ IDEA

目前用于 Groovy 和 Grails 开发的 IDE 中，最成熟、最全面的是 IntelliJ IDEA 7.0 和它的 JetGroovy 插件。在大型项目中，Grails 团队优先推荐使用 IDEA。

NetBeans

一个非常好的开源软件是 Sun 的 NetBeans，它提供了一个 Groovy/Grails 插件,可自动识别 Grails 项目并提供在 IDE 中运行 Grails 应用程序,代码自动完成,集成 Sun 的 Glassfish 服务器的能力。可以查看 Grails 站点由 NetBeans 团队编写的 NetBeans 集成功能描述

Eclipse

对于 Eclipse，Groovy Eclipse 插件提供了语法高亮和代码自动完成等功能。

> 在 Grails 的 Wiki 上有更多关于 Groovy Eclipse 插件的详细讨论。

Grails 为你自动创建了用于 Eclipse 的.project 以及 classpath 文件，所以要在 Eclipse 中导入一个 Grails 项目，只需在“Package Explorer”中点右键并选择“Import”，随后选择“Existing project into Workspace”并“Browse”你的项目位置。

接着顺序点击“Ok”和“Finish”即可完成项目的导入和安装。

Grails 也将自动安装一个项目对应的“Run Configuration”配置，随后可以在 Eclipse 的“Run”菜单中来使用它运行 Grails。

TextMate

由于 Grails 关注的是简洁性，所以我们可以使用一些更简单的编辑器，例如在 Mac 环境下的 TextMate，它对 Groovy/Grails 有着优秀的支持，可以从 Texmate bundles SVN 获得它。

2.6 约定优于配置

Grails 使用“约定优于配置”原则来配置自己。这通常意味着文件的名称和位置被用来替代明确的配置，因此你需要熟悉 Grails 提供的目录结构。

以下是大致目录结构并链接到相关的章节：

- grails-app - Groovy 源码的顶级目录
    - conf - 源配置。
    - controllers - Web 控制器 - MVC 模式中的 C 层。
    - domain - 应用域。
    - i18n - 国际化（i18n）支持。
    - services - 服务层。
    - taglib - 标记库。
    - views - 视图，包含了 Groovy 服务器页面（GSP）。
- scripts - Gant 脚本。
- src - 源文件目录
    - groovy - 其他的 Groovy 源文件
    - java -其他的 Java 源文件
- test - 单元测试和集成测试。

2.7 运行应用程序

Grails 应用程序可以使用 run-app 命令来运行在内置的 Jetty 服务器上，这个命令将默认在 8080 端口上启动一个服务器：

```
grails run-app
```

当然你也可以使用 server.port 变量来指定其他端口：

```
grails -Dserver.port=8090 run-app
```

更多关于 run-app 命令的信息可以在参考指南中找到。

2.8 测试应用程序

Grails 中的 create-* 系列命令可以为你在 test/integration 目录内创建集成测试代码框架。当然你还得自己来填写有效的逻辑测试代码，更多的信息可以参考 测试 一章。如果你要执

行测试代码，那么可以运行 test-app 命令：

```
grails test-app
```

Grails 也自动生成了用于 Ant 的 build.xml 文件，它可以委托 Grails 的 test-app 命令来运行测试代码：

```
ant test
```

当你使用如 CruiseControl 这样的持续集成平台来自动构建 Grails 应用程序时，使用 Ant 的方式将非常有用。

2.9 部署应用程序

Grails 应用程序是通过 Web 应用程序档(WAR 文件）的格式来部署的，它使用 war 命令来执行这个部署任务：

```
grails war
```

这将在你的项目根目录中产生一个 WAR 文件，你可以参照你的容器指南来部署它。

> 绝对不要使用 run-app 命令来作为部署的命令，因为它使 Grails 能够在运行期间自动重载，但这将带来许多性能和扩展性问题。

部署好 Grails 之后，你应该为你的容器 JVM 设置 -server 选项来分配足够的内容。一个较好的 VM 设置应该像这样：

```
-server -Xmx512M
```

2.10 创建工件

Grails 提供了许多像 create-controller 和 create-domain-class 这样方便的命令，可以使用它们来为你创建 控制器 等各种类型的工件。

> 这只是为了方便你进行开发，你也可以随意使用喜欢的 IDE 或文本编辑器来完成同样的工作。

例如，一个应用程序的基础部分是 域模型 ，我们可以像这样创建它：

```
grails create-domain-class book
```

这将在 grails-app/domain/Book.groovy 文件中创建一个域类，内容如下：

```
class Book {
}
```

还有许多类似 create-* 这样的命令，你可以在命令行参考指南中了解它们

2.10 支持的 Java EE 容器

Grails 支持相当广泛的容器，如下：

- Tomcat 5.5
- Tomcat 6.0
- GlassFish v1 (Sun AS 9.0)
- GlassFish v2 (Sun AS 9.1)
- Sun App Server 8.2
- Websphere 6.1
- Websphere 5.1
- Resin 3.2
- Oracle AS
- JBoss 4.2
- Jetty 6.1
- Jetty 5
- Weblogic 7/8/9/10

一些容器还有不少 Bug，但在多数情况下它们都能工作的很好。在 Grails 的 wiki 站点你能找到一份 开发中已知问题列表 。

2.11 生成一个应用

要使用 Grails 快速开始，经常用到的一个特性叫做 脚手架 ，它可以用来生成一个应用的骨架。要开始这样做，你可以使用 generate-* 这样的命令，如 generate-all 可以用来生成一个 控制器 以及相关的 视图 ：

```
grails generate-all Book
```

Show details

3. 配置

也许在这里谈论配置对于一个“约定优于配置”的框架来说，这可能比较奇怪，但这些配置通常都是一次性，我们最好还是先了解他们的大概。

由于 Grails 提供了默认设置，你确实可以在不做任何配置的情况下进行开发和应用。Grails 也内嵌了一个 Web 容器和一个称为 HSQLDB 的内存数据库，这意味着你甚至都不用安装数据库了。

不过，在将来某些情况下你还是会想要安装一个真正的数据库的，我们将在随后的一些章节进行描述。

3.1 基本配置

Grails 提供了一个名为 grails-app/conf/Config.groovy 的文件用来进行一般性配置。这个文件使用了 Groovy 的 ConfigSlurper 特性，除了它是由纯正的 Groovy 实现外，它与 Java 的 properties 文件是非常相似的，因此你可以在应用中重用定义的变量或者使用适合的 Java 类型！

你可以在这里添加你自己的配置，例如：

```
foo.bar.hello = "world"
```

配置完成后你就可以在你的应用程序里使用两种方式来访问这些设置了。最常用是通过 GrailsApplication 对象，它可以在控制器或标记库中作为一个变量来使用：

```
assert "world" == grailsApplication.config.foo.bar.hello
```

另一种方式是先获得对 ConfigurationHolder 类的引用，然后再通过它获得配置对象的引用：

```
import org.codehaus.groovy.grails.commons.*
...
def config = ConfigurationHolder.config
assert "world" == config.foo.bar.hello
```

3.1.1 内置选项

Grails 提供了下列配置选项：

- grails.config.locations - 资源（properties）文件或需要被合并到主配置文件中的附加 Grails 配置文件的位置
- grails.enable.native2ascii - 如果你不需要对 Grails 的 i18n 资源（properties）文件进行 native2ascii 的转换，那么就将该选项设为 false
- grails.views.default.codec - 用于设置 GSP 文件的默认编码体制——可以设置“none”、“html”或“base64”中的一个（默认值为：“none”）。为了降低 XSS 攻击的风险可以将改选项设为“html”。
- grails.views.gsp.encoding - 用于 GSP 源代码文件的文件编码（默认为“utf-8”）
- grails.mime.file.extensions - 是否使用文件扩展名来表示内容协商中的 MIME 类型
- grails.mime.types - 被支持的用于内容协商中的 MIME 类型对应表
- grails.serverURL - 一个用于描述绝对链接中服务器 URL 部分的字符串，其中包括了服务器名称。例如：grails.serverURL="http://my.yourportal.com"。具体内容请参考 createLink 一节。

生成 War 文件

- grails.war.destFile - 用来设置 war 命令将把生成的 WAR 文件放置在什么位置
- grails.war.dependencies - 一个包含了 Ant 构建器语法或 JAR 文件列表的闭包。允许你指定哪些库文件需要被包含在 WAR 文件中。
- grails.war.java5.dependencies - 一个 JAR 文件列表，这些 JAR 文件是需要被包含在用于 JDK 1.5 或以上版本的 WAR 文件里的。
- grails.war.copyToWebApp - 一个包含了 Ant 构建器语法的闭包，这些语法应该符合 Ant 的拷贝语法，例如“fileset()”。该功能允许你控制将“web-app”目录中的哪些内容包含到 WAR 文件中。
- grails.war.resources - 一个包含了 Ant 构建器语法的闭包。允许应用程序在正式生成 WAR 文件前做一些必要的事情。

要获得使用这些选项的更多信息，可以参考部署一章

3.1.2 日志

日志基础

Grails 使用它的通用配置方式来配置潜在的 Log4j 日志系统。要配置日志你需要修改位于 grails-app/conf 目录下的 Config.groovy 文件。

这个独特的 Config.groovy 文件允许你为 开发（development）、测试（test）和生产（production）环境（environments）分别进行日志的配置。Grails 将适当地处理 Config.groovy 文件并配置 Log4j。

从 1.1 版本的 Grails 开始，提供了一个 Log4j DSL，你可以像如下例子一样来配置 Log4j：

```
log4j = {
   error  'org.codehaus.groovy.grails.web.servlet',  //  controllers
          'org.codehaus.groovy.grails.web.pages' //  GSP
   warn   'org.mortbay.log'
}
```

实际上，每个方法都可以转化为一个日志级别，你可以把你想要记录日志的包名作为方法的参数。

以下是一些有用的日志记录器：

- org.codehaus.groovy.grails.commons - 记录核心工件的信息，如类加载等。
- org.codehaus.groovy.grails.web - 记录 Grails 的 Web 请求处理
- org.codehaus.groovy.grails.web.mapping - URL 映射的调试
- org.codehaus.groovy.grails.plugins - 记录插件活动情况
- org.springframework -查看 Spring 在做什么
- org.hibernate - 查看 Hibernate 在做什么

顶级日志记录器

顶级日志记录器会被所有其他日志记录器继承。你可以使用 root 方法来配置顶级日志记录器：

```
root {
   error()
   additivity = true
}
```

下边的例子用来配置顶级日志记录器去记录错误级别的信息，它的上方是默认的标准输出目标。你也可以将顶级日志记录器配置为将日志输出到多个已命名的输出目标：

```
appenders {
      file name:'file', file:'/var/logs/mylog.log'
}
root {
   debug 'stdout', 'file'
```

```
    additivity = true
}
```

这里的顶级日志记录器将日志记录到了两个输出目标——默认的“stdout”输出目标和一个“file”输出目标。

你也可以通过参数方式进入 Lorg4J 闭包的方式来配置顶级日志记录器：

```
log4j = { root ->
    root.level = org.apache.log4j.Level.DEBUG
    ...
}
```

闭包参数“root”是 org.apache.log4j.Logger 的一个实例，因此你可以查阅 Log4J 的 API 文档，找出哪些属性和方法对你有用。

自定义输出目标

使用 Log4j 你可以明确的定义输出目标。下边是默认可用的输出目标：

- jdbc - 用于将日志输出到 JDBC 连接的输出目标
- console - 用于将日志输出到标准输出的输出目标
- file - 用于将日志输出到文件的输出目标
- rollingFile - 用于将日志输出到滚动文件集的输出目标

例如你可以配置一个滚动文件输出目标：

```
log4j = {
    appenders {
        rollingFile name:"myAppender", maxFileSize:1024, fileName:"/tmp/logs/myApp.log"
    }
}
```

每个进入输出目标的参数都会对应到 Appender 类的一个属性。上边的例子设置了 RollingFileAppender 类的 name、maxFileSize 和 fileName 属性。

如果你愿意通过自己编程来创建输出目标或者你已经有自己的输出目标实现，那么你可以简单地调用 appender 方法以及输出目标实例：

```
import org.apache.log4j.*
log4j = {
    appenders {
        appender new RollingFileAppender(name:"myAppender", maxFileSize:1024,
```

```
fileName:"/tmp/logs/myApp.log")
        }
}
```

现在你可以将输出目标的名称作为一个唯一值设置到某个日志级别方法中，这样日志就记录到一个特定的输出目标中。这些在上一节讲述过：

```
error myAppender:"org.codehaus.groovy.grails.commons"
```

自定义布局

Log4j DSL 默认假设你想要使用 样板布局（PatternLayout） 日志格式。也有如下其他布局可用使用：

- xml - 创建一个 XML 布局日志文件
- html - 创建一个 HTML 布局日志文件
- simple - 创建一个简单的纯文本布局日志文件
- pattern - 创建一个样板布局日志文件

你可以使用 layout 设置来指定自定义的样板作为一个输出目标：

```
log4j = {
    appenders {
    console name:'customAppender', layout:pattern(conversionPattern: '%c{2} %m%n')
  }
}
```

这样的设置也可以用于内置的“stdout”输出目标，这样会将日志输出到控制台中：

```
log4j = {
  appenders {
    console name:'stdout', layout:pattern(conversionPattern: '%c{2} %m%n')
  }
}
```

完整的堆栈日志跟踪

当发生异常时，会产生大量来自 Java 和 Groovy 内部的堆栈日志信息。Grails 过滤了那些典型的无关信息，同时聚焦到非 Grails/Groovy 核心类的信息上。

当这种情况发生时，完整的追踪信息总是会写到 StackTrace 日志记录器。这些日志被记录到一个称为 stacktrace.log 的文件中 - 当然你也可以修改 Config.groovy 文件来进行你想

要的设置。例如，如果你更喜欢将完整的堆栈记录信息输出到标准输出，可以添加这样一行：

```
error stdout:"StackTrace"
```

你也可以将 grails.full.stacktrace 虚拟机属性设置为 true 来完全禁用堆栈跟踪过滤器：

```
grails -Dgrails.full.stacktrace=true run-app
```

约定的日志记录方式

所有的应用程序工件都有一个动态添加的 log 属性。这些工件类型包括 domain 类、控制器和标记库等。下边是一个使用例子：

```
def foo = "bar"
log.debug "The value of foo is $foo"
```

Grails 使用 grails.app.<工件类型>.ClassName 来作为日志记录器的命名。下边是一个如何配置日志记录器去记录不同 Grails 工件的日志的例子：

```
log4j = {
    // 为所有的应用程序工件设置
    info "grails.app"
    // 为一个特定的控制器设置
    debug "grails.app.controller.YourController"
    // 为一个特定的 domain 类设置
    debug "grails.app.domain.Book"
    // 为所有的标记库设置
    info "grails.app.tagLib"
}
```

工件名称（ <工件类型> ）也是按照约定命名的，一些常见的如下列表：

- bootstrap - 用于系统启动类
- dataSource - 用于数据源
- tagLib - 用于标记库
- service - 用于服务类
- controller - 用于控制器
- domain - 用于 domain 实体

3.2 环境

多环境配置

Grails 支持“多环境配置”的概念。grails-app/conf 中的 Config.groovy 和 DataSource.groovy 两个文件能够使用 ConfigSlurper 提供的语法来应用“多环境配置”的特性。以下例子是 Grails 提供的默认 DataSource 里的定义：

```
dataSource {
    pooled = false
    driverClassName = "org.hsqldb.jdbcDriver"
    username = "sa"
    password = ""
}
environments {
    development {
        dataSource {
            dbCreate = "create-drop" // 可选“create”、“createeate-drop”和“update”中的一个
            url = "jdbc:hsqldb:mem:devDB"
        }
    }
    test {
        dataSource {
            dbCreate = "update"
            url = "jdbc:hsqldb:mem:testDb"
        }
    }
    production {
        dataSource {
            dbCreate = "update"
            url = "jdbc:hsqldb:file:prodDb;shutdown=true"
        }
    }
}
```

注意配置文件的开头部分提供的是公共配置，紧接着的 environments 代码块则指定了用于独立环境配置的数据源信息，包括 dbCreate 和 url 属性。这样的语法也可以用与 Config.groovy 文件。

针对不同环境的打包和运行

Grails 的命令行 已经内建了针对特定环境来执行任何命令的能力。格式为：

grails [环境名] [命令名]

另外，已经有三个 Grails 的预制环境：dev、prod 和 test 分别用于 开发、生产和测试。例如要为 test 环境创建一个 WAR 包，你可以这样做：

grails test war

如果你有自建的其他环境需要使用，可以通过 grails.env 变量来设置并用于任何命令：

grails -Dgrails.env=UAT run-app

可编程的环境检测

在你的 Gant 脚本或系统启动类的代码中，你可以使用 Environment 类来检测环境：

```
import grails.util.Environment
...
switch(Environment.current) {
      case Environment.DEVELOPMENT:
        configureForDevelopment()
      break
      case Environment.PRODUCTION:
        configureForProduction()
      break
}
```

3.3 数据源

Grails 是基于 Java 技术构建的，因此要在其中安装数据源必然需要一些 JDBC（这种技术并不只支持 Java 数据库连接）的知识。

根本上来说，如果你正在使用的另一种数据库，而不是 Grails 内嵌的 HSQLDB，那么你就需要为它准备一个 JDBC 驱动。例如使用 MySQL 数据库，就需要 Connector/J

这个 JDBC 驱动。通常这些 JDBC 驱动都是以 JAR 文件格式发行的。将需要的 JAR 文件放到项目的 lib 目录下即可。

一旦你把 JAR 文件放到了正确的位置，你还需要熟悉位于 grails-

app/conf/DataSource.groovy 的 Grails 数据库描述文件。这个文件包含了数据源的定义，其中有下列这些设置：

- driverClassName - JDBC 驱动的类名
- username - 获得 JDBC 连接需要使用的用户名
- password - 获得 JDBC 连接需要使用的密码
- url - 数据库的 JDBC URL
- dbCreate - 是否从 domain 模型自动生成数据库
- pooled - 是否使用连接池（默认为 true）
- logSql - 是否启动 SQL 日志记录
- dialect - 用于表示与数据库通讯时应该使用的 Hibernate 方言的字符串或类。查看 org.hibernate.dialect 一文以便获得可用的方言。

一个用于 MySQL 数据库的典型配置可以像这样：

```
dataSource {
    pooled = true
    dbCreate = "update"
    url = "jdbc:mysql://localhost/yourDB"
    driverClassName = "com.mysql.jdbc.Driver"
    username = "yourUser"
    password = "yourPassword"
}
```

在配置数据源的时候不要在配置项之前包含类型声明或 def 关键字，否则 Groovy 会把它们当作本地变量定义并且不对它们进行处理。例如下边的例子就是无效的：

```
dataSource {
    boolean pooled = true // 类型声明导致它被当作是一个本地变量
    ...
}
```

3.3.1 数据源和环境

前边的配置范例假设你想要对所有的环境做一些配置，包括：生产、测试和开发等。

Grails 的数据源定义是“环境感知”的，因此你可以针对需要的环境这样配置：

```
dataSource {
```

```
        // 这里放置公共设置
}
environments {
  production {
     dataSource {
         url = "jdbc:mysql://liveip.com/liveDb"
     }
  }
}
```

3.3.2 JNDI 数据源

许多 Java EE 容器通常都支持通过 Java 命名与目录接口 （ JNDI ）来获取 数据源 实例。有时你可能需要通过 JNDI 去查找一个 数据源 。

Grails 支持像下边这样的 JNDI 数据源定义：

```
dataSource {
   jndiName = "java:comp/env/myDataSource"
}
```

JNDI 的名称格式在不同的容器中会有不同，但是在定义 数据源 的方式上是一致的。

3.3.3 自定数据库迁移

DataSource 的 dbCreate 属性是非常重要的，它会指示 Grails 在运行期间使用 GORM 类来自动生成数据库表。选项如下：

- create-drop - 当 Grails 运行的时候删除并且重新创建数据库。
- create - 如果数据库不存在则创建数据库，存在则不做任何修改。删除现有的数据。
- update - 如果数据库不存在则创建数据库，存在则对它进行修改更新。

  create-drop 和 create 都会删除所有存在的数据，因此请小心使用！

In 部署 模式下 dbCreate 默认被设置为“create-drop” ：

```
dataSource {
      dbCreate = "create-drop" // one of 'create', 'create-drop','update'
}
```

在每次应用程序重启时都会自动删除并重建数据库表。显然，这不应该用于生产环境。

> 尽管目前 Grails 还不支持 Rails 风格的开箱迁移特性，但有两个插件可以提供 Grails 类似的简单能力：LiquiBase 插件和 DbMigrate 插件都可以通过 grails list-plugins 命令获得。

3.4 外部配置

大多数情况下， grails-app/conf 目录下的 Config.groovy 默认配置文件是足够使用了，但可能有某些特殊情况让你想要在主应用程序框架 *之外* 维护一个配置文件。例如你使用 WAR 文件部署了系统，管理员会经常需要修改配置文件来改变系统的特性，但又要避免每次修改都得重新打包生成 WAR 文件。

为了支持这种外部配置文件的部署方案，你需要在 Config.groovy 文件的 grails.config.locations 设置中指明你的外部配置文件所在位置：

```
grails.config.locations = [ "classpath:${appName}-config.properties",
                            "classpath:${appName}-config.groovy",
                            "file:${userHome}/.grails/${appName}-config.properties",
                            "file:${userHome}/.grails/${appName}-config.groovy"]
```

上边的例子演示了从 classpath 和 USER_HOME 这些不同的位置来加载配置文件（包括 Java 属性（properties）文件和 ConfigSlurper 配置）。

最终所有的配置文件都被合并到了 GrailsApplication 对象的 config 属性中，就可以通过这个属性来获取配置信息了。

> Grails 也支持 Spring 中定义的属性占位（property place holder）概念和属性重载（property override）配置，更多信息请查看 Grails 和 Spring 一章。

3.5 版本管理

版本管理基础

Grails 已经内置了对版本管理的支持。当首次使用 create-app 命令创建应用程序的时候，应用程序的版本就被设置为 0.1 了。这个版本信息被记录在项目根目录下的应用程序元数据文件 application.properties 里边。

需要改变你的应用程序版本时你可以运行 set-version 命令：

```
grails set-version 0.2
```

版本信息被用在各种命令中，例如 war 命令就会将应用程序版本附加到创建的 WAR 文件末尾。

运行期间检测版本

你可以使用 Grails 对应用程序元数据的支持来检测应用程序版本，也就是使用 GrailsApplication 类。例如在 控制器 里你可以使用隐藏的 grailsApplication 变量：

```
def version = grailsApplication.metadata['app.version']
```

如果你需要获得的不是应用程序的版本而是 Grails 环境的版本，那么可以这样做：

```
def grailsVersion = grailsApplication.metadata['app.grails.version']
```

也可以使用 GrailsUtil 类：

```
import grails.util.*
def grailsVersion = GrailsUtil.grailsVersion
```

Show details

4. 命令行

Grails 的命令行系统是构建于 Gant 之上，Gant 就是使用 Groovy 对 Apache Ant 进行了简单的包装。

然而，Grails 通过约定规则以及 grails 命令的使用带来了一些改进。当你键入如下内容时：

```
grails [命令名称]
```

为了 Gant 脚本的执行，Grails 会在下列目录中做一次搜索：

- USER_HOME/.grails/scripts
- PROJECT_HOME/scripts
- PROJECT_HOME/plugins/*/scripts
- GRAILS_HOME/scripts

Grails 将把小写的命令名称（如 run-app）转换为单词连写的格式。因此如果键入的是

```
grails run-app
```

，那么 Grails 将会搜索下列文件：

- USER_HOME/.grails/scripts/RunApp.groovy
- PROJECT_HOME/scripts/RunApp.groovy
- PLUGINS_HOME/*/scripts/RunApp.groovy
- GLOBAL_PLUGINS_HOME/*/scripts/RunApp.groovy
- GRAILS_HOME/scripts/RunApp.groovy

如果找到多个同名的文件，Grails 将要求你选择执行其中的一个。当 Grails 执行一个 Gant 脚本的时候，它会首先调用定义在脚本文件中的“default”任务。如果找不到“default”任务，Grails 将退出并报错。

获得可用的命令及其帮助信息：

```
grails help
```

这个命令将输出 Grails 当前所知的命令列表和使用说明：

```
Usage (optionals marked with *):
grails [environment]* [target] [arguments]*
Examples:
grails dev run-app
grails create-app books
Available Targets (type grails help 'target-name' for more info):
grails bootstrap
grails bug-report
grails clean
grails compile
...
```

> 参考本使用指南左侧菜单中的命令行指南，可以获得更多的命令行的信息。

4.1 创建 Gant 脚本

你可以在项目的根目录下运行 create-script 命令来创建你自己的 Gant 脚本。例如如下命令：

```
grails create-script compile-sources
```

这将创建一个叫做 scripts/CompileSources.groovy 的脚本。Gant 脚本本身与规范的 Groovy 脚本非常相似，除了它支持“targets”的概念以及它们之间的依赖关系：

```
target(default:"default 任务是由 Grails 来执行的") {
    depends(clean, compile)
}
target(clean:"清除一些东西") {
    ant.delete(dir:"output")
}
target(compile:"编译一些源码") {
    ant.mkdir(dir:"mkdir")
    ant.javac(srcdir:"src/java", destdir:"output")
}
```

如上面的脚本所说明的，这个内置的 ant 变量可以访问 Apache Ant API。

在以前的 Grails 中（1.0.3 和以下），这个变量是 Ant，即第一个字母是大写的。

你也可以依赖其他的任务，只要在 default 任务中使用 depends 方法说明。

默认任务（default）

在上边的例子中，我们使用明确的名称“default”来指明一个任务。这是为一个脚本文件定义默认任务的一种方式。可选的另一种方式是使用 setDefaultTarget() 方法：

```
target("clean-compile": "对应用程序源文件执行清理并编译。") {
    depends(clean, compile)
}
target(clean:"清除文件") {
    ant.delete(dir:"output")
}
target(compile:"编译源码") {
    ant.mkdir(dir:"mkdir")
    ant.javac(srcdir:"src/java", destdir:"output")
}
setDefaultTarget("clean-compile")
```

这样将允许你从其他脚本中直接调用默认的任务。另外，尽管在这个例子中我们把调用 setDefaultTarget() 这一行放在了脚本文件的最后，但你可以把它放在任何位置，只要它位于它要引用的那个任务 *之后*（在这个例子中这个任务就是“clean-compile”）。

哪种方式更好？坦率地说，你可以使用你喜欢的那种方式——看起来这两种方式都没有什么突出的优势。我们应该讨论的一个问题是，如果你想要允许任何其他脚本都能调用你的

“default”任务，那么你应该把它移动到一个没有默认任务的共享脚本文件中。关于这些内容，我们将在下一章节进行更多讨论。

4.2 重用 Grails 脚本

Grails 带了许多开箱即用的命令行功能，你会发现这在你自己的脚本中那个会很有用（查看参考指南的命令行指南部分可以获得所有命令的详细信息）。尤其是使用 compile 、package 和 bootstrap 脚本。

下边的 bootstrap 脚本例子允许你启动一个 Spring 的 ApplicationContext 实例，通过它来访问数据源等(集成测试时可以这样用）：

```
includeTargets << grailsScript("_GrailsBootstrap")
target ('default': "Load the Grails interactive shell") {
    depends( configureProxy, packageApp, classpath, loadApp, configureApp )
  Connection c
    try {
        // 使用连接做一些事情
        c = appCtx.getBean('dataSource').getConnection()
    }
    finally {
        c?.close()
    }
}
```

从其他脚本文件引入任务

Gant 允许你从另一个 Gant 脚本文件中引入所有任务（除了“default”）。然后你就可以依赖或调用这些已经被定义在当前脚本文件中的任务了。实现的途径是 includeTargets 属性。使用左移操作符来简单的“附加”一个文件或类：

```
includeTargets << new File("/path/to/my/script.groovy")
includeTargets << gant.tools.Ivy
```

不用太担心关于使用一个类的语法，它是相当专业的。要是你感兴趣，可以看看 Gant 的文档。

核心的 Grails 任务

如你在本章开头部分所看到的例子，当使用 includeTargets 来包含核心的 Grails 任务时，

既没有使用基于文件的语法也没有使用基于类的语法。取而代之的，你应该使用 Grails 命令启动器提供的特殊的 grailsScript() 方法（注意这个方法在一般的 Gant 脚本中是不可用的，只有在 Grails 环境中才行）。

grailsScript() 方法的语法是非常简单易读的：简单的把你想要包含的 Grails 脚本文件的名称传入，不需要任何路径信息。以下是一个你可能想要重用的 Grails 脚本列表：

脚本	描述
_GrailsSettings	你确实应该包括这个！幸运的是，它已经被所有其他 Grails 脚本文件自动包括了（_GrailsProxy），因此你通常不必明确的包括它。
_GrailsEvents	如果你想要触发事件，你应该包括这个。添加一个 event(String eventName, List args) 方法。另外，这也被几乎所有其他 Grails 脚本文件包括。
_GrailsClasspath	安装编译、测试和运行用的 classpath。如果你想使用它们，就包含这个脚本。另外，这也由几乎所有其他 Grails 脚本包含。
_GrailsProxy	如果你需要访问互联网，为了避免遇到代理引起的问题请包含这个脚本。
_GrailsArgParsing	提供一个 parseArguments 任务，就像字面上的意思：当运行你的脚本的时候解析用户提供的参数。把参数添加到 argsMap 属性中。
_GrailsTest	包含所有共享的测试代码。如果你要添加额外的测试这将非常有用。
_GrailsRun	为你提供在配置好的 servlet 容器中运行应用程序时需要的一切，可以是正常的运行（runApp/runAppHttps），也可以是来自于一个 WAR 文件（runWar/runWarHttps）。

这些由 Grails 提供的脚本很值得对它们进行深入的分析，从而找出哪些类型的任务是可以使用的。任何脚本文件都是以“_”作为前缀以便进行重用。

> 在 Grails 1.1 版本之前，“_Grails...”这样的脚本文件是不可用的。而通常会包含对应命令脚本，例如“Init.groovy”或“Bootstrap.groovy”。
>
> 同样，在 Grails 1.0.4 版本之前，是无法使用 grailsScript() 方法的，你只能使用 includeTargets << new File(...) 并指明脚本的完整位置。（例如：$GRAILS_HOME/scripts）。

脚本结构

你可能对这些下划线词语作为 Grails 脚本的名称感到疑惑。用_internal_作为一个脚本或者用没有对应的“command”的其他单词，这些就是 Grails 的决定方式。因此无法运行例如“grails _grails-settings”这样的命令。这也就是为什么它们没有个默认的任务。

内部脚本是和代码共享重用相关的。实际上，我们建议在自己的脚本中使用类似的方式：把你的所有任务放入一个内部脚本中可以更容易的共享，然后提供简单的命令脚本来解析任何命令行参数并委托给内部脚本中的任务。假如你有一个脚本要运行一些功能测试——你可以将它们像这样分离：

```
./scripts/FunctionalTests.groovy:
includeTargets << new File("${basedir}/scripts/_FunctionalTests.groovy")
target(default: "为 这个 项目运行功能测试。") {
    depends(runFunctionalTests)
}
./scripts/_FunctionalTests.groovy:
includeTargets << grailsScript("_GrailsTest")
target(runFunctionalTests: "运行功能测试。") {
    depends(...)
    ...
}
```

以下是在编写脚本时常用的一些指导方案：

- 将脚本分为“command”脚本和内部脚本。
- 将大部分执行脚本放入内部脚本。
- 将参数解析放入“command”脚本。
- 要把参数传入一个任务，先创建一些脚本变量并在调用任务前将它们初始化。
- 为了避免名称冲突，可以为脚本变量分配闭包以替代任务。之后你可以直接将参数传入闭包。

4.3 钩子事件

Grails 提供了钩住脚本事件的能力。这里指的是当 Grails 的任务和插件脚本执行的时候能触发的一些事件。

这个机制是故意简单化和松散的规定的。可能的事件列表是不会以任何方式固定的，所以可以钩住那些被插件脚本触发的事件，在核心目标脚本中没有类似的事件。

定义事件处理器

事件处理器是定义在称为 _Events.groovy 的脚本文件中。Grails 会在以下位置搜索这些脚本：

- USER_HOME/.grails/scripts - 用户特定的事件处理器

- PROJECT_HOME/scripts - 应用程序特定的事件处理器
- PLUGINS_HOME/*/scripts - 插件特定的事件处理器
- GLOBAL_PLUGINS_HOME/*/scripts - 由全局插件提供的事件处理器

无论事件在何时被激发，*所有* 已经注册到该事件的处理器都会被执行。需要注意的是处理器的注册工作会由 Grails 自动进行，你只需要在相关的 _Events.groovy 文件中声明即可。

在 Grails 1.0.4 版本之前，脚本文件被命名为 Events.groovy，它没有前下划线。

事件处理器是分块定义在 _Events.groovy 文件中，使用“event”作为名称的开头部分。下边的例子可以被放在你的 /scripts 目录中来展示这个特性：

```
eventCreatedArtefact = { type, name ->
  println "Created $type $name"
}
eventStatusUpdate = { msg ->
  println msg
}
eventStatusFinal = { msg ->
  println msg
}
```

你可以看到这儿有三个处理器分别是：eventCreatedArtefact、eventStatusUpdate 和 eventStatusFinal。Grails 提供了一些标准的事件，它们在命令行参考指南中有描述。例如 compile 命令会激发下列事件：

- CompileStart - 当编译过程开始时，针对这几种类型的编译——源文件和测试文件
- CompileEnd - 当编译过程完成时，针对这几种类型的编译——源文件和测试文件

触发事件

要简单地触发一个包含 Init.groovy 脚本的事件并调用 event()闭包：

```
includeTargets << grailsScript("_GrailsEvents")
event("StatusFinal", ["Super duper plugin action complete!"])
```

公共事件

下表是一些可以被利用的公共事件：

事件	参数	描述

StatusUpdate	message	传入一个标志当前脚本状态或进展的字符串
StatusError	message	传入一个标志来自当前脚本的错误信息的字符串
StatusFinal	message	传入一个标志最终脚本状态消息的字符串，例如：当编译一个任务时，即使任务还没有退出脚本环境
CreatedArtefact	artefactType, artefactName	当一个 create-xxxx 脚本已执行完成并创建了一个工件时调用
CreatedFile	fileName	当一个项目的源码文件被创建时调用，但不包括那些由 Grails 管理的固定文件
Exiting	returnCode	当脚本环境即将正常的退出时调用
PluginInstalled	pluginName	在一个插件被安装之后调用
CompileStart	kind	当编译过程开始时调用，针对这几种类型的编译——源文件和测试文件
CompileEnd	kind	当编译过程完成时调用，针对这几种类型的编译——源文件和测试文件
DocStart	kind	当生成文档过程即将开始时调用——生成 javadoc 或 groovydoc 时
DocEnd	kind	当生成文档过程已经结束时调用——生成 javadoc 或 groovydoc 时
SetClasspath	rootLoader	在 classpath 初始化时调用以便插件可以通过 rootLoader.addURL(...)来扩大 classpath。注意这种扩大 classpath 是在事件脚本被加载 **之后**进行的，因此你不能使用这种方式来加载你的事件脚本需要导入的类，即使你可以通过名称来加载类。
PackagingEnd	none	当打包结束时调用（这个调用是在 Jetty 服务器被启动之前并在 web.xml 文件被生成之后）
ConfigureJetty	Jetty Server object	在 Jetty web 服务器的配置被初始化之后调用。

4.4 自定义构建

Grails 无疑是一个固执己见框架，并且它喜欢按照约定来进行配置，但这并不意味着你 *不能* 去配置它。在本章，我们将看到你可以如何去影响和修改标准的 Grails 构建。

默认

为了自定义一个构建，你首先需要知道你可以自定义些*什么*。Grails 构建配置的核心就是 grails.util.BuildSettings 类，它包含了大量有用的信息。它控制了哪些类被编译、应用程序依赖什么以及其他类似的设置。

以下是一个配置选项和它们的默认值的集录：

属性	配置选项	默认值
grailsWorkDir	grails.work.dir	$USER_HOME/.grails/<grailsVersion>
projectWorkDir	grails.project.work.dir	<grailsWorkDir>/projects/<baseDirName>
classesDir	grails.project.class.dir	<projectWorkDir>/classes
testClassesDir	grails.project.test.class.dir	<projectWorkDir>/test-classes
testReportsDir	grails.project.test.reports.dir	<projectWorkDir>/test/reports
resourcesDir	grails.project.resource.dir	<projectWorkDir>/resources
projectPluginsDir	grails.plugins.dir	<projectWorkDir>/plugins
globalPluginsDir	grails.global.plugins.dir	<grailsWorkDir>/global-plugins

BuildSettings 类也有一些其他属性，但是它们应该被只读处理：

属性	描述
baseDir	项目的位置。
userHome	用户的主目录。
grailsHome	正在使用的 Grails 的安装位置（也许为 null）。
grailsVersion	被项目使用的 Grails 的版本。
grailsEnv	当前的 Grails 环境。
compileDependencies	编译时项目依赖的 文件 实例列表。
testDependencies	测试时项目依赖的 文件 实例列表。
runtimeDependen	运行时项目依赖的 文件 实例列表。

cies
当然，如果你不能获得这些属性那么它们并没有多好。幸运的是这很容易实现：通过 grailsSettings 脚本变量可以得到一个 BuildSettings 实例用于你的脚本。你也可以在你的代码中通过使用 grails.util.BuildSettingsHolder 类来访问它，但是并不推荐这样做。

覆盖默认值

所有在第一个表中的属性都可以被一个系统属性或配置选项所覆盖——简单地使用“config option”名称。例如，要改变项目工作目录，你可以运行这个命令：

```
grails -Dgrails.project.work.dir=work compile
```

或者将这个选项添加到你的 grails-app/conf/BuildConfig.groovy 文件中：

```
grails.project.work.dir = "work"
```

注意默认值带有许多它们依赖的属性值，因此像这样设置项目工作目录也将迁移编译好的类、测试类、资源和插件。

如果你同时使用系统属性和配置选项将发生什么？当然是系统属性被采用了，因为它优先于 BuildConfig.groovy 文件，而后者优先于默认值。

BuildConfig.groovy 文件是 grails-app/conf/Config.groovy 的姐妹文件，——过去包含的选项仅仅影响构建，但是之后包含的就影响正在运行的应用程序了。这并不局限于第一个表中的选项：你会发现构建配置选项在文档中到处都是，比如其中一些就用来指定内嵌的 servlet 容器应该运行在哪个端口上或者决定哪些文件应该被打包到 WAR 文件中。

可用的构建设置

名称	描述
grails.server.port.http	指定内嵌的 servlet 容器应该运行的端口（ “run-app”和“run-war”命令使用）。整型。
grails.server.port.https	指定内嵌的 servlet 容器用于 HTTPS 的运行端口（ “run-app ~~https”和“run-war~~ https”）。整型。
grails.config.base.webXml	指定用于应用程序的自定义 web.xml 文件的路径（取代使用 web.xml 模板）。
grails.compiler.dependencies	将额外的依赖添加到编译器 classpath 的传统方式。设置它到一个包含“fileset()”入口的闭包。
grails.testing.patterns	一个 Ant 路径格式的列表，允许你控制哪些文件可以被包含在测

	试中。这些格式不应该包括测试用例后缀，它们将在下一个属性中设置。
grails.testing.nameSuffix	默认的，测试类都假定有一个“Tests”的后缀。你可以设置这个选项来改变它为你想要的任何内容。例如：另一个公共后缀是“Test”。
grails.war.destFile	一个包含了生成的 WAR 文件的文件路径的字符串，除了它的全名意外（包括扩展名）。例如，“target/my-app.war”。
grails.war.dependencies	一个包含“fileset()”入口的闭包，它允许你完全控制什么内容可以被放入 WAR 文件的“WEB-INF/lib”目录中。
grails.war.copyToWebApp	一个包含“fileset()”入口的闭包，它允许你完全控制什么内容可以被放入 WAR 文件的根目录中。它覆盖了包含“web-app”目录下所有内容的那种默认习惯。
grails.war.resources	一个闭包，它的第一个参数作为分段目录的位置。你可以使用任何 Ant 任务来做你想做的任何事。通常这用来在目录被打包成 WAR 之前从分段目录中删除文件。

4.5 Ant 和 Maven

如果你的团队或公司的所有其他项目都在使用像 Ant 或 Maven 这样的标准的构建工具进行构建的，当你使用 Grails 命令行来构建你的应用程序时你可能成为害群之马。幸运的是，今天你可以很容易的将 Grails 构建系统集成到正在使用的主要构建工具中（嗯，至少是在 Java 项目中使用的那种构建工具）。

Ant 集成

当你通过 create-app 命令来创建一个 Grails 应用程序时，Grails 会自动为你创建一个 Apache Ant 工具使用的 build.xml 文件，这个文件包含了下列的任务：

- clean - 清理 Grails 应用程序
- compile -编译你的应用程序的源码
- test - 运行单元测试
- run - 等同于“grails run-app”的功能
- war - 创建一个 WAR 文件
- deploy - 默认为空，但可以用它实现自动部署

这些任务都可以被 Ant 运行，例如：

ant war

为了实现依赖管理，构建文件已经被全面改进为使用 Apache Ivy，这意味着它可以自动下载所有需要的 Grails JAR 文件和其他以来的文件。你甚至不必在本地安装 Grails 就可以使用它了！这对于需要使用像 CruiseControl 或 Hudson 这样的持续集成系统进行自动构建时特别有用。

这里使用了 Grails 的 Ant task 来对现有的 Grails 构建系统进行钩子操作。这个任务允许你运行任何可用的 Grails 脚本，不只是由生成的构建文件所使用的那些。要使用某个任务，你必须先声明它：

```
<taskdef name="grailsTask"
      classname="grails.ant.GrailsTask"
      classpathref="grails.classpath"/>
```

这也引出了另外的问题：“grails.classpath”中应该是什么内容？这个任务本身是在“grails-bootstrap”这个 JAR 工件中的，因此至少这个工件需要在 classpath 中。同时也应该包含“groovy-all”这个 JAR。对于定义这个任务，你只需要使用这个！下表列出了可用的属性：

属性	描述	是否必填
home	构建时需要用到的 Grails 安装目录的位置。	除非 classpath 被指定否则必填。
classpathref	载入 Grails 的 Classpath。必须包含“grails-bootstrap”工件并且应该包含“grails-scripts”。	除非 home 被设置或者你使用 classpath 元素否则必填。
script	要运行的 Grails 脚本的名称，例如：“TestApp”。	必填。
args	要加入脚本中的参数，例如：“-unit -xml”。	不是必填。默认为“”。
environment	运行脚本时的 Grails 环境。	不是必填。默认为脚本的 default。
includeRuntimeClasspath	高级设置：如果设为 true 则将应用程序的运行时 classpath 添加到构建 classpath 中。	不是必填。默认为 true。

这个任务也支持下列内嵌元素，这些全都是标准的 Ant 路径结构：

- classpath - 构建 classpath（用来载入 Gant 和 Grails 脚本）。

- compileClasspath - 用来编译应用程序的类的 Classpath。
- runtimeClasspath - 用来运行应用程序并将程序打成 WAR 包的 Classpath。通常包含了@compileClasspath 中的一切。
- testClasspath - 用来编译和运行测试的 Classpath。通常包含了 runtimeClasspath 中的一切。

要如何填写这些路径信息完全取决于你。如果你正在使用 home 属性并且把你自己的依赖内容放在了 lib 目录中，那么你不需要使用以上任何一个路径。如果想看看使用它们的例子，那么就查看为一个新应用而生成的 Ant 构建文件吧。

Maven 集成

从 1.1 版本起，Grails 通过一个 Maven 插件提供了与 Maven 2 的集成。当前作为基础的 Maven 插件，特别是由 Octo 创建的这个版本是非常有效的，它做得非常出色。

准备

为了使用这个新的插件，你只需要安装和设置 Maven 2。这是因为 **你不再需要单独的安装 Grails 为了使用 Maven！**

> Grails 集成 Maven 2 已经针对 Maven 2.0.9 及以上版本进行了设计和测试。它将无法工作在更早期的版本中。

为了让你的生活更轻松，我们强烈推荐你添加一个用于 Grails 的插件组到 Maven 的设置文件中（$USER_HOME/.m2/settings.xml）：

```
<settings>
  ...
  <pluginGroups>
    <pluginGroup>org.grails</pluginGroup>
  </pluginGroups>
</settings>
```

另外，如果你已经有用于 Grails 设置的 Octo Maven 工具，那么你需要删除 com.octo.mtg 插件组。

创建一个 Grails Maven 项目

要简单地创建一个支持 Maven 的 Grails 项目只要运行下边的命令：

```
mvn archetype:generate -DarchetypeGroupId=org.grails \
   -DarchetypeArtifactId=grails-maven-archetype \
   -DarchetypeVersion=1.0-SNAPSHOT \
   -DarchetypeRepository=http://snapshots.repository.codehaus.org \
   -DgroupId=example -DartifactId=my-app
```

无论你想为你的应用选择哪个 group ID 和 artifact ID，一切内容格式都必须像上面写的那样。这将创建一个新的 Maven 项目以及一个 POM 文件和一系列其他文件。你不会看到有什么是像一个 Grails 应用。因此，下一步就要创建一个你要使用的项目结构了：

```
cd my-app
mvn initialize
```

现在你已经有一个可以使用的 Grails 应用了。插件已经集成到了标准的构建周期，因此你可以使用标准的 Maven 语法来构建和打包你的应用程序了： mvn clean ， mvn compile ， mvn test ， mvn package 。

你也可以利用许多已经被包装成 Maven 目标的 Grails 命令：

- grails:create-controller - 调用 create-controller 命令
- grails:create-domain-class - 调用 create-domain-class 命令
- grails:create-integration-test - 调用 create-integration-test 命令
- grails:create-pom - 为现有的 Grails 项目创建一个新的 Maven POM 文件
- grails:create-script - 调用 create-script 命令
- grails:create-service - 调用 create-service 命令
- grails:create-taglib - 调用 create-tag-lib 命令
- grails:create-unit-test - 调用 create-unit-test 命令
- grails:exec - 执行一个任意的 Grails 命令行脚本
- grails:generate-all - 调用 generate-all 命令
- grails:generate-controller - 调用 generate-controller 命令
- grails:generate-views - 调用 generate-views 命令
- grails:install-plugin - 调用 install-plugin 命令
- grails:install-templates - 调用 install-templates 命令
- grails:list-plugins - 调用 list-plugins 命令
- grails:package - 调用 package 命令
- grails:run-app - 调用 run-app 命令
- grails:uninstall-plugin - 调用 uninstall-plugin 命令

给现有项目加入 Maven 支持

创建一个全新的项目当然是一个很好的途径，但如果已经有一个项目了该怎么办呢？你应该不会愿意先创建一个新项目然后再把旧项目的内容拷贝进去的。解决方法是使用下列命令为现有项目创建一个 POM 文件：

```
mvn grails:create-pom -DgroupId=com.mycompany
```

当这个命令完成时，你就可以立即使用标准的语法了，如 mvn package 。需要注意的是当创建 POM 文件时你必须指定一个 group ID。

添加 Grails 命令到 phase 中

标准的 POM 文件被创建是为了让 Grails 将合适的核心 Grails 命令附加到它们对应的构建语法上，因此“compile”对应“compile”语法，“war”对应“package”语法。当你想要将一个插件的命令附加到一个特定的 phase 上时，这可能没有什么帮助。典型的例子是功能测试。你如何确保你的功能测试（无论正在使用你决定的哪个插件）是使用“integration-test” phase 来运行的？

恐怕不是：所有事情都是可能的。在这个例子中，你可以使用额外的“execution”块来将命令联合到一个 phase 上：

```
<plugin>
    <groupId>org.grails</groupId>
    <artifactId>grails-maven-plugin</artifactId>
    <version>1.0-SNAPSHOT</version>
    <extensions>true</extensions>
    <executions>
     <execution>
      <goals>
      ...
      </goals>
     </execution>
     <!-- 添加 "functional-tests" 命令到 "integration-test" phase -->
     <execution>
      <id>functional-tests</id>
      <phase>integration-test</phase>
      <goals>
       <goal>exec</goal>
      </goals>
      <configuration>
```

```
        <command>functional-tests</command>
      </configuration>
    </execution>
  </executions>
</plugin>
```

这也展示了 grails:exec 目标，它可以用来运行任何 Grails 命令。简单的将命令的名字作为 command 系统特性，还可以通过 args 特性来选择性地指定参数：

```
mvn grails:exec -Dcommand=create-webtest -Dargs=Book
```

Show details

5. 对象关联映射 (GORM)

Domain 类是任何商业应用的核心。 他们保存事务处理的状态，也处理预期的行为。 他们通过关联联系在一起, one-to-one 或 one-to-many。

GORM 是 Grails 对象关联映射 (GORM)的实现。在底层，它使用 Hibernate 3 (一个非常流行和灵活的开源 ORM 解决方案)，但是因为 Groovy 天生的动态性，实际上，对动态类型和静态类型两者都支持，由于 Grails 的规约，只需要很少的配置涉及 Grails domain 类的创建。

你同样可以在 Java 中编写 Grails domain 类。 请参阅在 Hibernate 集成上如果在 Java 中编写 Grails domain 类， 不过，它仍然使用动态持久方法。下面是 GORM 实战预览:

```
def book = Book.findByTitle("Groovy in Action")
book
  .addToAuthors(name:"Dierk Koenig")
  .addToAuthors(name:"Guillaume LaForge")
  .save()
```

5.1 快速入门指南

domain 类可以使用 create-domain-class 命令来创建:

```
grails create-domain-class Person
```

这将在 grails-app/domain/Person.groovy 位置上创建类，如下:

```
class Person {
```

```
}
```

> 如果在 DataSource 上设置 dbCreate 属性为"update", "create" or "create-drop", Grails 会为你自动生成/修改数据表格。

你可以通过添加属性来自定义类:

```
class Person {
        String name
        Integer age
        Date lastVisit
}
```

一旦你拥有一个 domain 类，可以尝试通过在 shell 或 console 上输入:

```
grails console
```

这会载入一个交互式 GUI，便于你键入 Groovy 命令。

5.1.1 CRUD 基础

尝试执行一些基础的 CRUD (Create/Read/Update/Delete) 操作。

Create

为了创建一个 domain 类，可以使用 Groovy new 操作符, 设置它的属性并调用 save:

```
def p = new Person(name:"Fred", age:40, lastVisit:new Date())
p.save()
```

save 方法将使用底层的 Hibernate ORM 持久你的类到数据库中。

Read

Grails 会为你的 domain 类显式的添加一个隐式 id 属性，便于你检索:

```
def p = Person.get(1)
assert 1 == p.id
```

get 方法通过你指定的数据库标识符，从 db 中读取 Person 对象。 你同样可以使用 read 方法加载一个只读状态对象:

```
def p = Person.read(1)
```

在这种情况下，底层的 Hibernate 引擎不会进行任何脏读检查，对象也不能被持久化。注意,假如你显式的调用 save 方法，对象会回到 read-write 状态.

Update

更新一个实体, 设置一些属性，然后，只需再次调用 save:

```
def p = Person.get(1)
p.name = "Bob"
p.save()
```

Delete

删除一个实体使用 delete 方法:

```
def p = Person.get(1)
p.delete()
```

5.2 GORM 中进行 Domain 建模

当构建 Grails 应用程序时，你必须考虑你要试图解决的问题域。 比如，你正在构建一个 Amazon 书店，你要考虑 books, authors, customers 和 publishers 等等.

这些在 GORM 中被当做 Groovy 类 来进行建模，因此， Book 类可能拥有 title, release date,ISBN 等等。 在后面章节将展示如何在 GORM 中进行 domain 建模。

创建 domain 类，你可以运行 create-domain-class ，如下:

```
grails create-domain-class Book
```

将会创建 grails-app/domain/Book.groovy 类:

```
class Book {
}
```

> 如果你想使用 packages 你可以把 Book.groovy 类移动到 domain 目录的子目录下，并按照 Groovy (和 Java)的 packaging 规则添加正确的 package 。

上面的类将会自动映射到数据库中名为 book 的表格 (与类名相同). 可以通过 ORM Domain Specific Language 定制上面的行为。

现在，你可以把这个 domain 类的属性定义成 Java 类型。 例如:

```
class Book {
        String title
        Date releaseDate
        String ISBN
}
```

每个属性都会被映射到数据库的列，列名的规则是所有列名小写，通过下划线分隔。 比如 releaseDate 映射到 release_date 列。 SQL 类型会自动检测来自 Java 的类型，但可以通过 Constraints 或 ORM DSL 定制。

5.2.1 GORM 中的关联

关联定义了 domain 类之间的相互作用。除非在两端明确的指定,否则关联只存在被定义的一方。

5.2.1.1 One-to-one

one-to-one 关联是最简单的种类，它只是把它的一个属性的类型定义为其他 domain 类。考虑下面的例子:

Example A

```
class Face {
   Nose nose
}
class Nose {
}
```

在这种情况下， 拥有一个 Face 到 Nose 的 one-to-one 单向关联。为了使它双向关联，需要定义另一端，如下:

Example B

```
class Face {
   Nose nose
}
class Nose {
        Face face
}
```

这就是双向关联。不过, 在这种情况下，关联的双方并不能级联更新。

考虑下这样的变化:

Example C

```
class Face {
   Nose nose
}
class Nose {
      static belongsTo = [face:Face]
}
```

在这种情况下，我们使用 belongsTo 来设置 Nose "属于" Face。结果是，我们创建一个 Face 并 save 它，数据库将 *级联* 更新/插入 Nose:

```
new Face(nose:new Nose()).save()
```

上面的示例，face 和 nose 都会被保存。注意，逆向 *不为* true，并会因为一个临时的 Face 导致一个错误:

```
new Nose(face:new Face()).save() // will cause an error
```

belongsTo 另一个重要的意义在于，假如你删除一个 Face 实体， Nose 也会被删除:

```
def f = Face.get(1)
f.delete() // both Face and Nose deleted
```

如果没有 belongsTo ，deletes 将**不**被级联，并会得到一个外键约束错误，除非你明确的删除 Nose:

```
// error here without belongsTo
def f = Face.get(1)
f.delete()
// no error as we explicitly delete both
def f = Face.get(1)
f.nose.delete()
f.delete()
```

你可以保持上面的关联为单向，为了保证级联保存/更新,可以像下面这样:

```
class Face {
   Nose nose
}
```

```
class Nose {
        static belongsTo = Face
}
```

注意，在这种情况下，我们没有在 belongsTo 使用 map 语法声明和明确命名关联。Grails 会把它当做单向。.下面的图表概述了 3 个示例:

5.2.1.2 One-to-many

one-to-many 关联是，当你的一个类，比如 Author ，拥有许多其他类的实体，比如 Book 。 在 Grails 中定义这样的关联可以使用 hasMany :

```
class Author {
   static hasMany = [ books : Book ]
   String name
}
class Book {
        String title
}
```

在这种情况下，拥有一个单向的 one-to-many 关联。 Grails 将默认使用一个连接表映射这样的关联。

> ORM DSL 允许使用外键关联作为映射单向关联的替代

对于 hasMany 设置，Grails 将自动注入一个 java.util.Set 类型的属性到 domain 类。用于迭代集合:

```
def a = Author.get(1)
a.books.each {
        println it.title
}
```

> Grails 中默认使用的 fetch 策略是 "lazy", 意思就是集合将被延迟初始化。 如果你不小心，这会导致 n+1 问题 。
>
> 如果需要"eager" 抓取 ，需要使用 ORM DSL 或者指定立即抓取作为 query 的一部分

默认的级联行为是级联保存和更新，但不删除，除非 belongsTo 被指定:

```
class Author {
   static hasMany = [ books : Book ]
   String name
}
class Book {
      static belongsTo = [author:Author]
      String title
}
```

如果在 one-to-many 的多方拥有 2 个同类型的属性，必须使用 mappedBy 指定哪个集合被映射:

```
class Airport {
      static hasMany = [flights:Flight]
      static mappedBy = [flights:"departureAirport"]
}
class Flight {
      Airport departureAirport
      Airport destinationAirport
}
```

如果多方拥有多个集合被映射到不同的属性，也是一样的:

```
class Airport {
      static hasMany = [outboundFlights:Flight, inboundFlights:Flight]
      static mappedBy = [outboundFlights:"departureAirport", inboundFlights:"destinationAirport"]
}
class Flight {
      Airport departureAirport
      Airport destinationAirport
}
```

5.2.1.3 Many-to-many

Grails 支持 many-to-many 关联，通过在关联双方定义 hasMany ，并在关联拥有方定义 belongsTo :

```
class Book {
  static belongsTo = Author
```

```
  static hasMany = [authors:Author]
  String title
}
class Author {
  static hasMany = [books:Book]
  String name
}
```

Grails 在数据库层使用一个连接表来映射 many-to-many，在这种情况下，Author 负责持久化关联，并且是唯一可以级联保存另一端的一方 。

例如，下面这个可以进行正常级联保存工作:

```
new Author(name:"Stephen King")
        .addToBooks(new Book(title:"The Stand"))
        .addToBooks(new Book(title:"The Shining"))
        .save()
```

而下面这个只保存 Book 而不保存 authors!

```
new Book(name:"Groovy in Action")
        .addToAuthors(new Author(name:"Dierk Koenig"))
        .addToAuthors(new Author(name:"Guillaume Laforge"))
        .save()
```

这是所期待的行为，就像 Hibernate，只有 many-to-many 的一方可以负责管理关联。

> 当前，Grails 的 Scaffolding 特性**不**支持 many-to-many 关联， 你必须自己编写关联的管理代码

5.2.1.4 集合类型基础

除了关联不同 domain 类外, GORM 同样支持映射基本的集合类型。比如，下面的类创建一个 nicknames 关联， 它是一个 String 的 Set 实体:

```
class Person {
   static hasMany = [nicknames:String]
}
```

GORM 将使用一个链接表，来映射上面的关联。你可以使用 joinTable 参数来改变各式各样的连接表映射:

```
class Person {
   static hasMany = [nicknames:String]
   static mapping = {
     hasMany joinTable:[name:'bunch_o_nicknames', key:'person_id', column:'nickname', type:"text"]
   }
}
```

上面的示例映射到表后看上去像这样:

bunch_o_nicknames Table

```
-----------------------------------------
| person_id      |    nickname         |
-----------------------------------------
|  1             |     Fred            |
-----------------------------------------
```

5.2.2 GORM 中的组合

除了 association 之外， Grails 支持组合概念。在这种情况下，并不是把类映射到分离的表格，而是将这个类"embedded"到当前的表格内。 例如:

```
class Person {
      Address homeAddress
      Address workAddress
      static embedded = ['homeAddress', 'workAddress']
}
class Address {
      String number
      String code
}
```

所产生的映射看上去像这样:

> 如果你在 grails-app/domain 目录中定义了一个单独的 Address 类，你同样会得到一个表格。如果你不想这样，你可以 利用 Groovy 在单个文件定义多个类的能力，让 grails-app/domain/Person.groovy 文件中的 Person 类包含 Address 类。

5.2.3 GORM 中的继承

GORM 支持从抽象类的继承和具体持久化 GORM 实体的继承。例如:

```
class Content {
    String author
}
class BlogEntry extends Content {
   URL url
}
class Book extends Content {
   String ISBN
}
class PodCast extends Content {
   byte[] audioStream
}
```

上面的示例，我们拥有一个 Content 父类和各式各样带有更多指定行为的子类。

注意事项

在数据库层， Grails 默认使用一个类一个表格的映射附带一个名为 class 的识别列，因此，父类 (Content) 和它的子类(BlogEntry, Book 等等.), 共享 **相同**的表格。

一个类一个表格的映射有个负面的影响，就是你 **不能** 有非空属性一起继承映射。 另一个选择是使用每个子类一个表格 ，你可以通过 ORM DSL 启用。

不过, 过分使用继承与每个子类一个表格会带来糟糕的查询性能，因为，过分使用链接查询。总之，我们建议：假如你打算使用继承，不要滥用它，不要让你的继承层次太深。

多态性查询

继承的结果是你有能力进行多态查询。比如，在 Content 使用 list 方法，超类将返回所有 Content 子类:

```
def content = Content.list() // list all blog entries, books and pod casts
content = Content.findAllByAuthor('Joe Bloggs') // find all by author
def podCasts = PodCast.list() // list only pod casts
```

5.2.4 Sets, Lists 和 Maps

Sets 对象

默认情况下，在中 GORM 定义一个 java.util.Set 映射，它是无序集合，不能包含重复元素。 换句话，当你有:

```
class Author {
  static hasMany = [books:Book]
}
```

GORM 会将 books 注入为 java.util.Set 类型。问题在于存取时，这个集合的无序的，可能不是你想要的。为了定制序列，你可以设置为 SortedSet:

```
class Author {
  SortedSet books
  static hasMany = [books:Book]
}
```

在这种情况下，需要实现 java.util.SortedSet ，这意味着，你的 Book 类必须实现 java.lang.Comparable:

```
class Book implements Comparable {
  String title
  Date releaseDate = new Date()
  int compareTo(obj) {
     releaseDate.compareTo(obj.releaseDate)
  }
}
```

上面的结果是，Author 类的中的 books 集合将按 Book 的 releasedate 排序。

List 对象

如果你只是想保持对象的顺序，添加它们和引用它们通过索引，就像 array 一样，你可以定义你的集合类型为 List:

```
class Author {
  List books
  static hasMany = [books:Book]
}
```

在这种情况下当你向 books 集合中添加一个新元素时,这个顺序将会保存在一个从 0 开始的列表索引中,因此你可以:

```
author.books[0] // get the first book
```

这种方法在数据库层的工作原理是:为了在数据库层保存这个顺序,Hibernate 创建一个叫做 books_idx 的列,它保存着该元素在集合中的索引.

当使用 List 时,元素在保存之前必须先添加到集合中,否则 Hibernate 会抛出异常 (org.hibernate.HibernateException: null index column for collection):

```
// This won't work!
def book = new Book(title: 'The Shining')
book.save()
author.addToBooks(book)
// Do it this way instead.
def book = new Book(title: 'Misery')
author.addToBooks(book)
author.save()
```

映射(Maps)对象

如果你想要一个简单的 string/value 对 map,GROM 可以用下面方法来映射:

```
class Author {
   Map books // map of ISBN:book names
}
def a = new Author()
a.books = ["1590597583":"Grails Book"]
a.save()
```

这种情况 map 的键和值都必须是字符串.

如果你想用一个对象的 map,那么你可以这样做:

```
class Book {
  Map authors
  static hasMany = [authors:Author]
}
def a = new Author(name:"Stephen King")
```

```
def book = new Book()
book.authors = [stephen:a]
book.save()
```

static hasMany 属性定义了 map 中元素的类型,map 中的 key **必须** 是字符串.

集合类型和性能

Java 中的 Set 是一个不能有重复条目的集合类型. 为了确保添加到 Set 关联中的条目是唯一的，Hibernate 首先加载数据库中的全部关联. 如果你在关联中有大量的条目，那么这对性能来说是一个巨大的浪费.

这样做就需要 List 类型, 因为 Hibernate 需要加载全部关联以维持供应. 因此如果你希望大量的记录关联，那么你可以制作一个双向关联以便连接能在反面被建立。例如思考一下代码:

```
def book = new Book(title:"New Grails Book")
def author = Author.get(1)
book.author = author
book.save()
```

在这个例子中关联链接被 child (Book)创建，因此没有必要手动操作集合以使查询更少和高效代码。由于 Author 有大量的关联的 Book 实例，如果你写入像下面的代码，你可以看到性能的影响：

```
def book = new Book(title:"New Grails Book")
def author = Author.get(1)
author.addToBooks(book)
author.save()
```

5.3 持久化基础

关于 Grails 要记住的很重要的一点就是，Grails 的底层使用 Hibernate 来进行持久化. 如果您以前使用的是 ActiveRecord 或者 iBatis 您可能会对 Hibernate 的"session"模型感到有点陌生.

本质上,Grails 自动绑定 Hibernate session 到当前正在执行的请求上.这允许你像使用 GORM 的其他方法一样很自然地使用 save 和 delete 方法.

5.3.1 保存和更新

下面看一个使用 save 方法的例子:

```
def p = Person.get(1)
p.save()
```

一个主要的不同是当你调用 save 的**时候** Hibernate 不会执行任何 SQL 操作. Hibernate 通常将 SQL 语句分批,最后执行他们.对你来说,这些一般都是由 Grails 自动完成的,它管理着你的 Hibernate session.

也有一些特殊情况,有时候你可能想自己控制那些语句什么时候被执行,或者用 Hibernate 的术语来说,就是什么时候 session 被"flushed".要这样的话,你可以对 save 方法使用 flush 参数:

```
def p = Person.get(1)
p.save(flush:true)
```

请注意，在这种情况下，所有暂存的 SQL 语句包括以往的保存将同步到数据库。这也可以让您捕捉任何被抛出的异常，这在涉及乐观锁高度并发的情况下是很常用的：

```
def p = Person.get(1)
try {
	p.save(flush:true)
}
catch(Exception e) {
	// deal with exception
}
```

5.3.2 删除对象

下面是 delete 方法的一个例子:

```
def p = Person.get(1)
p.delete()
```

默认情况下在执行 delete 以后 Grails 将使用事务写入, 如果你想在适当的时候删除，这时你可以使用 flush 参数:

```
def p = Person.get(1)
p.delete(flush:true)
```

使用 flush 参数也允许您捕获在 delete 执行过程中抛出的任何异常. 一个普遍的错误就是违犯数据库的约束, 尽管这通常归结为一个编程或配置错误. 下面的例子显示了当您违犯了数据库约束时如何捕捉 DataIntegrityViolationException:

```
def p = Person.get(1)
try {
    p.delete(flush:true)
}
catch(org.springframework.dao.DataIntegrityViolationException e) {
    flash.message = "Could not delete person ${p.name}"
    redirect(action:"show", id:p.id)
}
```

注意 Grails 没有提供 deleteAll 方法,因为删除数据是 discouraged 的，而且通常可以通过布尔标记/逻辑来避免.

如果你确实需要批量删除数据,你可以使用 executeUpdate 法来执行批量的 DML 语句:

```
Customer.executeUpdate("delete Customer c where c.name = :oldName", [oldName:"Fred"])
```

5.3.3 级联更新和删除

在使用 GORM 时,理解如何级联更新和删除是很重要的.需要记住的关键是 belongsTo 的设置控制着哪个类"拥有"这个关联.

无论是一对一,一对多还是多对多,如果你定义了 belongsTo ,更新和删除将会从拥有类到被它拥有的类(关联的另一方)级联操作.

如果你 *没有* 定义 belongsTo 那么就不能级联操作,你将不得不手动保存每个对象.

下面是一个例子:

```
class Airport {
    String name
    static hasMany = [flights:Flight]
}
class Flight {
    String number
    static belongsTo = [airport:Airport]
}
```

如果我现在创建一个 Airport 对象,并向它添加一些 Flight 它可以保存这个 Airport 并级联

保存每个 flight,因此会保存整个对象图:

```
new Airport(name:"Gatwick")
        .addToFlights(new Flight(number:"BA3430"))
        .addToFlights(new Flight(number:"EZ0938"))
        .save()
```

相反的,如果稍后我删除了这个 Airport 所有跟它关联的 Flight 也都将会被删除:

```
def airport = Airport.findByName("Gatwick")
airport.delete()
```

然而,如果我将 belongsTo 去掉的话,上面的级联删除代码就了. **不能工作**. 为了更好地理解, take a look at the summaries below that describe the default behaviour of GORM with regards to specific associations.

设置了 belongsTo 的双向一对多

```
class A { static hasMany = [bees:B] }
class B { static belongsTo = [a:A] }
```

如果是双向一对多，在多的一端设置了 belongsTo，那么级联策略将设置一的一端为"ALL"，多的一端为"NONE".

单向一对多

```
class A { static hasMany = [bees:B] }
class B {  }
```

如果是在多的一端没有设置 belongsTo 单向一对多关联，那么级联策略设置将为"SAVE-UPDATE".

没有设置 belongsTo 的双向一对多

```
class A { static hasMany = [bees:B] }
class B { A a }
```

如果是在多的一端没有设置 belongsTo 的双向一对多关联，那么级联策略将为一的一端设置为"SAVE-UPDATE" 为多的一端设置为"NONE".

设置了 belongsTo 的单向一对一

```
class A {  }
```

```
class B { static belongsTo = [a:A] }
```

如果是设置了 belongsTo 的单向一对一关联，那么级联策略将为有关联的一端(A->B)设置为"ALL"，定义了 belongsTo 的一端(B->A)设置为"NONE".

请注意，如果您需要进一步的控制级联的行为，您可以参见 ORM DSL.

5.3.4 立即加载和延迟加载

在 GORM 中,关联默认是 lazy 的.最好的解释是例子:

```
class Airport {
    String name
    static hasMany = [flights:Flight]
}
class Flight {
    String number
    static belongsTo = [airport:Airport]
}
```

上面的 domain 类和下面的代码:

```
def airport = Airport.findByName("Gatwick")
airport.flights.each {
    println it.name
}
```

GORM GORM 将会执行一个单独的 SQL 查询来抓取 Airport 实例,然后再用一个额外的 *for each* 查询逐条迭代 flights 关联.换句话说,你得到了 N+1 条查询.

根据这个集合的使用频率,有时候这可能是最佳方案.因为你可以指定只有在特定的情况下才访问这个关联的逻辑.

配置立即加载

一个可选的方案是使用立即抓取,它可以按照下面的方法来指定:

```
class Airport {
    String name
    static hasMany = [flights:Flight]
    static mapping = {
        flight fetch:"join"
```

```
    }
}
```

在这种情况下 Airport 实例对应的 flights 关联会被一次性全部加载进来(依赖于映射). 这样的好处是执行更少的查询,但是要小心使用,因为使用太多的 eager 关联可能会导致你将整个数据库加载进内存.

关联也可以用 ORM DSL 将关联声明为 non-lazy

使用批量加载 Using Batch Fetching

虽然立即加载适合某些情况,它并不总是可取的,如果您所有操作都使用立即加载,那么您会将整个数据库加载到内存中,导致性能和内存的问题.替代立即加载是使用批量加载.实际上,您可以在"batches"中配置 Hibernate 延迟加载. 例如:

```
class Airport {
    String name
    static hasMany = [flights:Flight]
    static mapping = {
        flight batchSize:10
    }
}
```

在这种情况下,由于 batchSize 参数,当您迭代 flights 关联, Hibernate 加载 10 个批次的结果. 例如，如果您一个 Airport 有 30 个 s, 如果您没有配置批量加载，那么您在对 Airport 的查询中只能一次查询出一个结果，那么要执行 30 次查询以加载每个 flight. 使用批量加载，您对 Airport 查询一次将查询出 10 个 Flight，那么您只需查询 3 次. 换句话说, 批量加载是延迟加载策略的优化. 批量加载也可以配置在 class 级别:

```
class Flight {
    ...
    static mapping = {
        batchSize 10
    }
}
```

5.3.5 悲观锁和乐观锁

乐观锁

默认的 GORM 类被配置为乐观锁。乐观锁实质上是 Hibernate 的一个特性，它在数据库里一个特别的 version 字段中保存了一个版本号.

version 列读取包含当前你所访问的持久化实例的版本状态的 version 属性:

```
def airport = Airport.get(10)
println airport.version
```

当你执行更新操作时，Hibernate 将自动检查 version 属性和数据库中 version 列，如果他们不同，将会抛出一个 StaleObjectException 异常，并且当前事物也会被回滚.

这是很有用的，因为它允许你不使用悲观锁(有一些性能上的损失)就可以获得一定的原子性。由此带来的负面影响是，如果你有一些高并发的写操作的话，你必须处理这个异常。这需要刷出(flushing)当前的 session:

```
def airport = Airport.get(10)
try {
	airport.name = "Heathrow"
	airport.save(flush:true)
}
catch(org.springframework.dao.OptimisticLockingFailureException e) {
	// deal with exception
}
```

你处理异常的方法取决于你的应用. 你可以尝试合并数据,或者返回给用户并让他们来处理冲突.

作为选择，如果它成了问题，你可以求助于悲观锁.

悲观锁

悲观锁等价于执行一个 SQL "SELECT * FOR UPDATE" 语句并锁定数据库中的一行. 这意味着其他的读操作将会被锁定直到这个锁放开.

在 Grails 中悲观锁通过 lock 方法执行:

```
def airport = Airport.get(10)
airport.lock() // lock for update
```

```
airport.name = "Heathrow"
airport.save()
```

一旦当前事物被提交，Grails 会自动的为你释放锁. 可是,在上述情况下我们做的事情是从正规的 SELECT“升级”到 SELECT ..FOR UPDATE 同时其它线程也会在调用 get()和 lock()之间更新记录。

为了避免这个问题，你可以使用静态的 lock 方法，就像 get 方法一样传入一个 id:

```
def airport = Airport.lock(10) // lock for update
airport.name = "Heathrow"
airport.save()
```

这个只有 SELECT..FOR UPDATE 时候可以使用.

> 尽管 Grails 和 Hibernate 支持悲观所，但是在使用 Grails 内置默认的 HSQLDB 数据库时**不支持**。如果你想测试悲观锁，你需要一个支持悲观锁的数据库，例如 MySQL.

你也可以使用 lock 方法在查询中获得悲观锁。例如使用动态查询：

```
def airport = Airport.findByName("Heathrow", [lock:true])
```

或者使用 criteria:

```
def airport = Airport.createCriteria().get {
    eq('name', 'Heathrow')
    lock true
}
```

5.4 GORM 查询

GORM 提供了从动态查询器到 criteria 到 Hibernate 面向对象查询语言 HQL 的一系列查询方式.

Groovy 通过 GPath 操纵集合的能力, 和 GORM 的像 sort,findAll 等方法结合起来，形成了一个强大的组合.

但是，让我们从基础开始吧.

获取实例列表

如果你简单的需要获得给定类的所有实例，你可以使用 list 方法:

```
def books = Book.list()
```

list 方法支持分页参数:

```
def books = Book.list(offset:10, max:20)
```

也可以排序:

```
def books = Book.list(sort:"title", order:"asc")
```

这里，Here, the sort 参数是您想要查询的 domain 类中属性的名字，argument is the name of the domain class property that you wish to sort on, and the order 参数要么以 argument is either asc for **asc** 结束 ending or 要么以 desc for **desc** 结束 ending.

根据数据库标识符取回

第二个取回的基本形式是根据数据库标识符取回，使用 get 方法:

```
def book = Book.get(23)
```

你也可以根据一个标识符的集合使用 getAll 方法取得一个实例列表:

```
def books = Book.getAll(23, 93, 81)
```

5.4.1 动态查询器

GORM 支持 **动态查找器** 的概念 . 动态查找器看起来像一个静态方法的调用，但是这些方法本身在代码中实际上并不存在.

而是在运行时基于一个给定类的属性,自动生成一个方法. 比如例子中的 Book 类:

```
class Book {
    String title
    Date releaseDate
    Author author
}
class Author {
    String name
}
```

Book 类有一些属性，比如 title, releaseDate 和 author. 这些都可以按照"方法表达式"的格式被用于 findBy 和 findAllBy 方法:

```
def book = Book.findByTitle("The Stand")
book = Book.findByTitleLike("Harry Pot%")
```

```
book = Book.findByReleaseDateBetween( firstDate, secondDate )
```

```
book = Book.findByReleaseDateGreaterThan( someDate )
```

```
book = Book.findByTitleLikeOrReleaseDateLessThan( "%Something%", someDate )
```

方法表达式

在 GORM 中一个方法表达式由前缀,比如 findBy 后面跟一个表达式组成，这个表达式由一个或多个属性组成。基本形式是:

```
Book.findBy([Property][Comparator][Boolean Operator])?[Property][Comparator]
```

用'?' 标记的部分是可选的. 每个后缀都会改变查询的性质。例如:

```
def book = Book.findByTitle("The Stand")
book =  Book.findByTitleLike("Harry Pot%")
```

在上面的例子中，第一个查询等价于等于后面的值, 第二个因为增加了 Like 后缀, 它等价于 SQL 的 like 表达式.

可用的后缀包括:

- InList - list 中给定的值

- LessThan - 小于给定值
- LessThanEquals - 小于或等于给定值
- GreaterThan - 大于给定值
- GreaterThanEquals - 大于或等于给定值
- Like - 价于 SQL like 表达式
- Ilike - 类似于 Like,但不是大小写敏感
- NotEqual - 不等于
- Between - 于两个值之间 (需要两个参数)
- IsNotNull - 不为 null 的值 (不需要参数)
- IsNull - 为 null 的值 (不需要参数)

你会发现最后三个方法标注了参数的个数，他们的示例如下:

```
def now = new Date()
def lastWeek = now - 7
def book = Book.findByReleaseDateBetween( lastWeek, now )
books = Book.findAllByReleaseDateIsNull()
books = Book.findAllByReleaseDateIsNotNull()
```

布尔逻辑(AND/OR)

方法表达式也可以使用一个布尔操作符来组合两个 criteria:

```
def books =
   Book.findAllByTitleLikeAndReleaseDateGreaterThan("%Java%", new Date()-30)
```

在这里我们在查询中间使用 And 来确保两个条件都满足, 但是同样地你也可以使用 Or:

```
def books =
   Book.findAllByTitleLikeOrReleaseDateGreaterThan("%Java%", new Date()-30)
```

At the moment 此时, 你最多只能用两个 criteria 做动态查询, 也就是说，该方法的名称只能含有一个布尔操作符. 如果你需要使用更多的, 你应该考虑使用 Criteria 或 HQL.

查询关联

关联也可以被用在查询中:

```
def author = Author.findByName("Stephen King")
def books = author ? Book.findAllByAuthor(author) : []
```

在这里如果 Author 实例不为 null 我们在查询中用它取得给定 Author 的所有 Book 实例.

分页和排序

跟 list 方法上可用的分页和排序参数一样,他们同样可以被提供为一个 map 用于动态查询器的最后一个参数:

```
def books =
  Book.findAllByTitleLike("Harry Pot%", [max:3,
                                         offset:2,
                                         sort:"title",
                                         order:"desc"])
```

5.4.2 条件查询

Criteria 是一种类型安全的、高级的查询方法，它使用 Groovy builder 构造强大复杂的查询.它是一种比使用 StringBuffer 好得多的选择.

Criteria 可以通过 createCriteria 或者 withCriteria 方法来使用. builder 使用 Hibernate 的 Criteria API, builder 上的节点对应 Hibernate Criteria API 中 Restrictions 类中的静态方法. 用法示例:

```
def c = Account.createCriteria()
def results = c {
    like("holderFirstName", "Fred%")
    and {
        between("balance", 500, 1000)
        eq("branch", "London")
    }
    maxResults(10)
    order("holderLastName", "desc")
}
```

逻辑与（Conjunctions）和逻辑或（Disjunctions）

如前面例子所演示的，你可以用 and { } 块来分组 criteria 到一个逻辑 AND:

```
and {
    between("balance", 500, 1000)
    eq("branch", "London")
}
```

逻辑 OR 也可以这么做:

```
or {
      between("balance", 500, 1000)
      eq("branch", "London")
}
```

你也可以用逻辑 NOT 来否定:

```
not {
      between("balance", 500, 1000)
      eq("branch", "London")
}
```

查询关联

关联可以通过使用一个跟关联属性同名的节点来查询. 比如我们说 Account 类有关联到多个 Transaction 对象:

```
class Account {
   ...
   def hasMany = [transactions:Transaction]
   Set transactions
   ...
}
```

我们可以使用属性名 transaction 作为 builder 的一个节点来查询这个关联:

```
def c = Account.createCriteria()
def now = new Date()
def results = c.list {
      transactions {
          between('date',now-10, now)
      }
}
```

上面的代码将会查找所有过去 10 天内执行过 transactions 的 Account 实例. 你也可以在逻辑块中嵌套关联查询:

```
def c = Account.createCriteria()
def now = new Date()
def results = c.list {
    or {
```

```
        between('created',now-10,now)
        transactions {
            between('date',now-10, now)
        }
    }
}
```

这里,我们将找出在最近 10 天内进行过交易或者最近 10 天内新创建的所有用户.

投影(Projections)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 c = Account.createCriteria()
def numberOfBranches = c.get {
        projections {
                countDistinct('branch')
        }
}
```

使用可滚动的结果

Y 你可以通过调用 scroll 方法来使用 Hibernate 的 ScrollableResults 特性:

```
def results = crit.scroll {
    maxResults(10)
}
def f = results.first()
def l = results.last()
def n = results.next()
def p = results.previous()
def future = results.scroll(10)
def accountNumber = results.getLong('number')
```

下面引用的是 Hibernate 文档中关于 ScrollableResults 的描述:

> 结果集的迭代器（iterator）可以以任意步进的方式前后移动，而 Query / ScrollableResults 模式跟 JDBC 的 PreparedStatement/ ResultSet 也很像，其接口方法名的语意也跟 ResultSet 的类似.

不同于 JDBC，结果列的编号是从 0 开始.

在 Criteria 实例中设置属性

如果在 builder 树内部的一个节点不匹配任何一项特定标准，它将尝试设置为 Criteria 对象自身的属性。因此允许完全访问这个类的所有属性。下面的例子是在 Criteria Criteria 实例上调用 setMaxResults 和 setFirstResult:

```
import org.hibernate.FetchMode as FM
        ...
        def results = c.list {
                maxResults(10)
                firstResult(50)
                fetchMode("aRelationship", FM.EAGER)
        }
```

立即加载的方式查询

在 Eager and Lazy Fetching 立即加载和延迟加载 这节，我们讨论了如果指定特定的抓取方式来避免 N+1 查询的问题。这个 criteria 查询也可以做到:

```
def criteria = Task.createCriteria()
def tasks = criteria.list{
    eq "assignee.id", task.assignee.id
    join 'assignee'
    join 'project'
    order 'priority', 'asc'
}
```

注意这个 join 方法的用法. This method indicates the criteria API that a JOIN query should be used to obtain the results.

方法引用

如果你调用一个没有方法名的 builder，比如:

```
c { ... }
```

默认的会列出所有结果，因此上面代码等价于:

```
c.list { ... }
```

方法	描述
list	这是默认的方法。它会返回所有匹配的行。
get	返回唯一的结果集，比如，就一行。criteria 已经规定好了，仅仅查询一行。这个方法更方便，免得使用一个 limit 来只取第一行使人迷惑。
scroll	返回一个可滚动的结果集
listDistinct	如果子查询或者关联被使用，有一个可能就是在结果集中多次出现同一行，这个方法允许只列出不同的条目，它等价于 CriteriaSpecification 类的 DISTINCT_ROOT_ENTITY

5.4.3 Hibernate 查询语言(HQL)

GORM 也支持 Hibernate 的查询语言 HQL,在 Hibernate 文档中的 Chapter 14. HQL: The Hibernate Query Language 可以找到它非常完整的参考手册。

GORM 提供了一些使用 HQL 的方法，包括 find, findAll 和 executeQuery. 下面是一个查询的例子:

```
def results =
    Book.findAll("from Book as b where b.title like 'Lord of the%'")
```

位置和命名参数

上面的例子中传递给查询的值是硬编码的，但是，你可以同样地使用位置参数:

```
def results =
    Book.findAll("from Book as b where b.title like ?", ["The Shi%"])
```

或者甚至使用命名参数:

```
def results =
    Book.findAll("from Book as b where b.title like :search or b.author like :search", [search:"The Shi%"])
```

多行查询

如果你需要将查询分割到多行你可以使用一个行连接符:

```
def results = Book.findAll("\
from Book as b, \
   Author as a \
where b.author = a and a.surname = ?", ['Smith'])
```

Groovy 的多行字符串对 HQL 查询无效

分页和排序

使用 HQL 查询的时候你也可以进行分页和排序。要做的只是简单指定分页和排序参数作为一个散列在方法的末尾调用:

```
def results =
    Book.findAll("from Book as b where b.title like 'Lord of the%' order by b.title asc",
            [max:10, offset:20])
```

5.5 高级 GORM 特性

接下来的章节覆盖更多高级的 GORM 使用 包括 缓存、定制映射和事件。

5.5.1 事件和自动实现时间戳

GORM 支持事件注册，只需要将事件作为一个闭包即可，当某个事件触发，比如删除，插入，更新。The following is a list of supported events 下面就是所支持事件的列表:

- beforeInsert - 对象持久到数据之前执行
- beforeUpdate - 对象被更新之前执行
- beforeDelete - 对象被删除之前执行
- afterInsert - 对象持久到数据库之后执行
- afterUpdate - 对象被更新之后执行
- afterDelete - 对象被删除之后执行
- onLoad - 对象从数据库中加载之后执行

为了添加一个事件需要在你的领域类中添加相关的闭包。

事件类型

beforeInsert 事件

当一个对象保存到数据库之前触发

```
class Person {
  Date dateCreated
  def beforeInsert = {
```

```
    dateCreated = new Date()
  }
}
```

beforeUpdate 事件

当一个对象被更新之前触发

```
class Person {
  Date dateCreated
  Date lastUpdated
  def beforeInsert = {
    dateCreated = new Date()
  }
  def beforeUpdate = {
    lastUpdated = new Date()
  }
}
```

beforeDelete 事件

当一个对象被删除以后触发.

```
class Person {
  String name
  Date dateCreated
  Date lastUpdated
  def beforeDelete = {
    new ActivityTrace(eventName:"Person Deleted",data:name).save()
  }
}
```

onLoad 事件

当一个对象被加载之后触发:

```
class Person {
  String name
  Date dateCreated
  Date lastUpdated
  def onLoad = {
```

```
    name = "I'm loaded"
  }
}
```

自动时间戳

上面的例子演示了使用事件来更新一个 lastUpdated 和 dateCreated 属性来跟踪对象的更新。事实上，这些设置不是必须的。通过简单的定义一个 lastUpdated 和 dateCreated 属性，GORM 会自动的为你更新。

如果，这些行为不是你需要的，可以屏蔽这些功能。如下设置:

```
class Person {
  Date dateCreated
  Date lastUpdated
  static mapping = {
    autoTimestamp false
  }
}
```

5.5.2 自定义 ORM 映射

Grails 的域对象可以映射到许多遗留的模型通过 关系对象映射域语言。接下来的部分将带你领略它是可能的通过 ORM DSL。

> 这是必要的，如果你高兴地坚持以约定来定义 GORM 对应的表，列名等。你只需要这个功能，如果你需要定制 GORM 映射到遗留模型或进行缓存

自定义映射是使用静态的 mapping 块定义在你的域类中的:

```
class Person {
  ..
  static mapping = {
  }
}
```

5.5.2.1 表名和列名

表名

类映射到数据库的表名可以通过使用 table 关键字来定制:

```
class Person {
  ..
  static mapping = {
     table 'people'
  }
}
```

在上面的例子中，类会映射到 people 表来代替默认的 person 表.

列名

同样，也是可能的定制某个列到数据库。比如说，你想改变列名例子如下:

```
class Person {
  String firstName
  static mapping = {
     table 'people'
     firstName column:'First_Name'
  }
}
```

在这个例子中，你定义了一个 column 块，此块包含的方法调用匹配每一个属性名称 (in this case firstName). 接下来使用命名的 column, 来指定字段名称的映射.

列类型

GORM 还可以通过 DSL 的 type 属性来支持 Hibernate 类型. 包括特定 Hibernate 的 org.hibernate.usertype.UserType 的子类, which allows complete customization of how a type is persisted. 比如，有一个 PostCodeType 你可以象下面这样使用:

```
class Address {
   String number
   String postCode
   static mapping = {
     postCode type:PostCodeType
```

```
  }
}
```

另外如果你想将它映射到 Hibernate 的基本类型而不是 Grails 的默认类型，可以参考下面代码:

```
class Address {
  String number
  String postCode
  static mapping = {
    postCode type:'text'
  }
}
```

上面的例子将使 postCode 映射到数据库的 SQL TEXT 或者 CLOB 类型.

See the Hibernate documentation regarding Basic Types for further information.

一对一映射

在关联中，你也有机会改变外键映射联系，在一对一的关系中，对列的操作跟其他常规的列操作并无二异，例子如下:

```
class Person {
  String firstName
  Address address
  static mapping = {
     table 'people'
     firstName column:'First_Name'
        address column:'Person_Adress_Id'
  }
}
```

默认情况下 address 将映射到一个名称为 address_id 的外键. 但是使用上面的映射，我们改变外键列为 Person_Adress_Id.

一对多映射

在一个双向的一对多关系中，你可以象前节中的一对一关系中那样改变外键列，只需要在多的一端中改变列名即可。然而，在单向关联中，外键需要在关联自身中（即一的一端-译者注）指定。比如，给定一个单向一对多联系 Person 和 Address 下面的代码会改变 address 表中外键:

```
class Person {
  String firstName
  static hasMany = [addresses:Address]
  static mapping = {
     table 'people'
     firstName column:'First_Name'
        addresses column:'Person_Address_Id'
  }
}
```

如果你不想在 address 表中有这个列,可以通过中间关联表来完成，只需要使用 joinTable 参数即可:

```
class Person {
  String firstName
  static hasMany = [addresses:Address]
  static mapping = {
     table 'people'
     firstName column:'First_Name'
     addresses joinTable:[name:'Person_Addresses', key:'Person_Id', column:'Address_Id']
  }
}
```

多对多映射

默认情况下, Grails 中多对多的映射是通过中间表来完成的. 以下面的多对多关联为例:

```
class Group {
       ...
       static hasMany = [people:Person]
}
class Person {
       ...
       static belongsTo = Group
```

```
    static hasMany = [groups:Group]
}
```

在上面的例子中 Grails 将会创建一个 group_person 表包含外键 person_id 和 group_id 对应 person 和 group 表. 假如你需要改变列名，你可以为每个类指定一个列映射.

```
class Group {
  ...
  static mapping = {
    people column:'Group_Person_Id'
  }
}
class Person {
  ...
  static mapping = {
    groups column:'Group_Group_Id'
  }
}
```

你也可以指定中间表的名称:

```
class Group {
  ...
  static mapping = {
    people column:'Group_Person_Id',joinTable:'PERSON_GROUP_ASSOCIATIONS'
  }
}
class Person {
  ...
  static mapping = {
    groups column:'Group_Group_Id',joinTable:'PERSON_GROUP_ASSOCIATIONS'
  }
}
```

5.5.2.2 缓存策略

设置缓存

Hibernate 本身提供了自定义二级缓存的特性. 这就需要在 grails-app/conf/DataSource.groovy 文件中配置:

```
hibernate {
   cache.use_second_level_cache=true
   cache.use_query_cache=true
   cache.provider_class='org.hibernate.cache.EhCacheProvider'
}
```

当然，你也可以按你所需来定制设置，比如，你想使用分布式缓存机制.

想了解更多 Hibernate 的二级缓存，参考 Hibernate documentation 相关文档.

缓存实例

假如要在映射代码块中启用缺省的缓存，可以通过调用 cache 方法实现:

```
class Person {
  ..
  static mapping = {
     table 'people'
     cache true
  }
}
```

上面的例子中将配置一个读-写(read-write)缓存包括 lazy 和 non-lazy 属性.假如你想定制这些特性，你可以如下所示:

```
class Person {
  ..
  static mapping = {
     table 'people'
     cache usage:'read-only', include:'non-lazy'
  }
}
```

缓存关联对象

就像使用 Hibernate 的二级缓存来缓存实例一样，你也可以来缓存集合（关联），比如:

```
class Person {
  String firstName
  static hasMany = [addresses:Address]
  static mapping = {
     table 'people'
```

```
    version false
    addresses column:'Address', cache:true
  }
}
class Address {
  String number
  String postCode
}
```

上面的例子中，我们在 addresses 集合启用了一个读-写缓存，你也可以使用:

```
cache:'read-write' // or 'read-only' or 'transactional'
```

更多配置请参考缓存用法.

Caching Queries

You can cache queries such as dynamic finders and criteria. To do so using a dynamic finder you can pass the cache argument:

```
def person = Person.findByFirstName("Fred", cache:true)
```

> Note that in order for the results of the query to be cached, you still need to enable caching in your mapping as discussed in the previous section.

You can also cache criteria queries:

```
def people = Person.withCriteria {
      like('firstName', 'Fr%')
      cache true
}
```

缓存用法

下面是不同缓存设置和他们的使用方法:

- read-only - 假如你的应用程序需要读但是从不需要更改持久化实例，只读缓存或许适用.
- read-write - 假如你的应用程序需要更新数据，读-写缓存或许是合适的.
- nonstrict-read-write - 假如你的应用程序仅偶尔需要更新数据（也就是说，如果这是极不可能两笔交易，将尝试更新同一项目同时）并且时进行），并严格交易隔离，

是不是需要一个 nonstrict-read-write 可能是适宜的.

- transactional - transactional 缓存策略提供支持对全事务缓存提供比如 JBoss 的 TreeCache. 这个缓存或许仅仅使用在一个 JTA 环境，同时你必须在 grails-app/conf/DataSource.groovy 文件中的 hibernate 配置中 hibernate.transaction.manager_lookup_class.

5.5.2.3 继承策略

默认情况下 GORM 类使用 table-per-hierarchy 来映射继承的. 这就有一个缺点就是在数据库层面，列不能有 NOT-NULL 的约束。如果你更喜欢 table-per-subclass 你可以使用下面方法:

```
class Payment {
   Long id
   Long version
   Integer amount
   static mapping = {
      tablePerHierarchy false
   }
}
class CreditCardPayment extends Payment  {
   String cardNumber
}
```

在祖先 Payment 类的映射设置中，指定了在所有的子类中，不使用 table-per-hierarchy 映射.

5.5.2.4 自定义数据库标识符

你可以通过 DSL 来定制 GORM 生成数据库标识，缺省情况下 GORM 将根据原生数据库机制来生成 ids，这是迄今为止最好的方法，但是仍存在许多模式，不同的方法来生成标识。

为此，Hibernate 特地定义了 id 生成器的概念，你可以自定义它要映射的 id 生成器和列，如下:

```
class Person {
  ..
  static mapping = {
     table 'people'
```

```
    version false
    id generator:'hilo', params:[table:'hi_value',column:'next_value',max_lo:100]
  }
}
```

在上面的例子中，我们使用了 Hibernate 内置的'hilo'生成器，此生成器通过一个独立的表来生成 ids.

想了解更多不同的 Hibernate 生成器请参考 Hibernate 文档

注意，如果你仅仅想定制列 id，你可以这样:

```
class Person {
  ..
  static mapping = {
    table 'people'
    version false
    id column:'person_id'
  }
}
```

5.5.2.5 复合主键

GORM 支持复合标识（复合主键--译者注）. 概念（标识由两个或者更多属性组成，这不是我们建议的方法，但是如果你想这么做，这也是可能的:

```
class Person {
  String firstName
  String lastName
  static mapping = {
    id composite:['firstName', 'lastName']
  }
}
```

上面的代码将通过 Person 类的 firstName 和 lastName 属性来创建一个复合 id。当你后面需要通过 id 取一个实例时，你必须用这个对象的原型:

```
def p = Person.get(new Person(firstName:"Fred", lastName:"Flintstone"))
println p.firstName
```

5.5.2.6 数据库索引

To get the best performance out of your queries it is often necessary to tailor the table index definitions. How you tailor them is domain specific and a matter of monitoring usage patterns of your queries. 为得到最好的查询性能，通常你需要调整表的索引定义。如何调整它们是跟特定领域和要查询的用法模式相关的。使用 GORM 的 DSL 你可以指定那个列需要索引：

```
class Person {
  String firstName
  String address
  static mapping = {
     table 'people'
     version false
     id column:'person_id'
     firstName column:'First_Name', index:'Name_Idx'
     address column:'Address', index:'Name_Idx, Address_Index'
  }
}
```

5.5.2.7 乐观锁和版本定义

就像在 乐观锁和悲观锁 部分讨论的，默认情况下，GORM 使用乐观锁和在每一个类中自动注入一个 version 属性，此属性将映射数据库中的一个 version 列.

如果你映射的是一个遗留数据库（已经存在的数据库--译者注），这将是一个问题，因此可以通过如下方法来关闭这个功能:

```
class Person {
  ..
  static mapping = {
     table 'people'
     version false
  }
}
```

> 如果你关闭了乐观锁 你将自己负责并发更新并且存在用户丢失数据的风险 (due to data overriding) 除非你使用 悲观锁

5.5.2.8 立即加载和延迟加载

延迟加载集合

就像在 立即加载和延迟加载, 部分讨论的，默认情况下，GORM 集合使用延迟加载的并且可以通过 fetchMode 来配置, 但如果你更喜欢把你所有的映射都集中在 mappings 代码块中，你也可以使用 ORM 的 DSL 来配置获取模式:

```
class Person {
  String firstName
  static hasMany = [addresses:Address]
  static mapping = {
     addresses lazy:false
  }
}
class Address {
  String street
  String postCode
}
```

延迟加载单向关联

在 GORM 中，one-to-one 和 many-to-one 关联缺省是非延迟加载的.这在有很多实体（数据库记录-译者注）的时候，会产生性能问题，尤其是关联查询是以新的 SELECT 语句执行的时候. 此时你应该将 one-to-one 和 many-to-one 关联的延迟加载象集合那样进行设置:

```
class Person {
        String firstName
        static belongsTo = [address:Address]
        static mapping = {
                address lazy:true // lazily fetch the address
        }
}
class Address {
        String street
        String postCode
}
```

这里我们设置 Person 的 address 属性为延迟加载.

5.5.2.9 自定义级联行为

正如 级联更新 这节描述的，控制更新和删除的主要机制是从关联一端到 belongsTo 静态属性的一端。

然而，通过 cascade 属性，ORM DSL 可以让你访问 Hibernate 的 transitive persistence 能力。

有效级联属性的设置包括：

- create - 创建从关联端到另一端的级联
- merge - 合并 detached 联合
- save-update - 只级联保存和更新
- delete - 只级联删除
- lock - 关联的悲观锁是否被级联
- refresh - 级联 refreshes
- evict - cascades evictions (equivalent to discard() in GORM) to associations if set
- all - 级联所有操作
- delete-orphan - Applies only to one-to-many associations and indicates that when a child is removed from an association then it should be automatically deleted

  获得级联样式更好的理解和用法的介绍，请阅读 Hibernate 文档的 transitive persistence 章节

使用上述的值定义一个或多个级联属性(逗号分隔)：

```
class Person {
  String firstName
  static hasMany = [addresses:Address]
  static mapping = {
     addresses cascade:"all,delete-orphan"
  }
}
class Address {
  String street
  String postCode
}
```

5.5.2.10 自定义 Hibernate 的类型

在较早的章节看到可以(通过 embedded 属性) 把一个表分成多个对象。 你也可以通过 Hibernate 的自定义用户类型实现相同的效果。这不是领域类本身，而是 java 或者 groovy 类。所有这些类型都有一个继承自 org.hibernate.usertype.UserType org.hibernate.usertype.UserType 的"meta-type"类。

Hibernate 参考手册 有一些自定义类型资料，在这里我们将重点放在如何在 Grails 中映射。让我们看一个使用老式的(Java 1.5 以前)枚举类型安全的领域类：

```
class Book {
  String title
  String author
  Rating rating
  static mapping = {
     rating type: RatingUserType
  }
}
```

我们所要做的是声明 rating 的枚举类型和在自定义映射 UserType 中设置属性的类型。这是你想使用自定义类型所必须做的。你也可以使用其他列的设置，比如使用"column"来改变列名和使用"index"把它添加到 index。

自定义类型不局限于只是一个列，他们可以映射到多列。在这种情况下，你必须在映射中明确地定义那列使用，因为 Hibernate 只能为一列使用属性名。 幸运的是，Grails 可以为属性映射多列：

```
class Book {
  String title
  Name author
  Rating rating
  static mapping = {
     name type: NameUserType, {
        column name: "first_name"
        column name: "last_name"
     }
     rating type: RatingUserType
  }
}
```

上面的例子将为 author 属性创建"first_name"和"last_name"列。You'll be pleased to

know that you can also use some of the normal column/property mapping attributes in the column definitions. For example:

```
column name: "first_name", index: "my_idx", unique: true
```

The column definitions do *not* support the following attributes: type, cascade, lazy, cache, and joinTable.

One thing to bear in mind with custom types is that they define the *SQL types* for the corresponding database columns. That helps take the burden of configuring them yourself, but what happens if you have a legacy database that uses a different SQL type for one of the columns? In that case, you need to override column's SQL type using the sqlType attribute:

```
class Book {
  String title
  Name author
  Rating rating
  static mapping = {
     name type: NameUserType, {
        column name: "first_name", sqlType: "text"
        column name: "last_name", sqlType: "text"
     }
     rating type: RatingUserType, sqlType: "text"
  }
}
```

Mind you, the SQL type you specify needs to still work with the custom type. So overriding a default of "varchar" with "text" is fine, but overriding "text" with "yes_no" isn't going to work.

5.5.3 缺省排序

你可以使用像 list 方法中的参数来排序对象：

```
def airports = Airport.list(sort:'name')
```

当然，你也可以定义一个排序的声明：

```
class Airport {
      ...
```

```
    static mapping = {
        sort "name"
    }
}
```

必要的话你可以配置这个排序：

```
class Airport {
    ...
    static mapping = {
        sort name:"desc"
    }
}
```

另外，您也可以在关联中配置排序：

```
class Airport {
    ...
    static hasMany = [flights:Flight]
    static mapping = {
        flights sort:'number'
    }
}
```

5.6 事务编程

Grails 是构建在 Spring 的基础上的，所以使用 Spring 的事务来抽象处理事务编程，但 GORM 类通过 withTransaction 方法使得处理更简单，方法的第一个参数是 Spring 的 TransactionStatus 对象.

典型的使用场景如下:

```
def transferFunds = {
    Account.withTransaction { status ->
        def source = Account.get(params.from)
        def dest = Account.get(params.to)
        def amount = params.amount.toInteger()
        if(source.active) {
            source.balance -= amount
            if(dest.active) {
                dest.amount += amount
```

```
                }
                else {
                        status.setRollbackOnly()
                }
        }




    }


}
```

在上面的例子中，如果目的账户没有处于活动状态，系统将回滚事务，同时如果有任何异常抛出在事务的处理过程中也将会自动回滚。

假如你不想回滚整个事务，你也可以使用"save points"来回滚一个事务到一个特定的点。你可以通过使用 Spring 的 SavePointManager 接口来达到这个目的.

The withTransaction 方法为你处理 begin/commit/rollback 代码块作用域内的逻辑。

5.7 GORM 和约束

尽管约束是 验证 章节的内容, 但是在此涉及到约束也是很重要的，因为一些约束会影响到数据库的生成。

Grails 通过使用领域类的约束来影响数据库表字段（领域类所对于的属性）的生成，还是可行的。

考虑下面的例子，假如我们有一个域模型如下的属性.

```
String name
String description
```

默认情况下，在 MySql 数据库中，Grails 将会定义这个列为...

```
column name | data type
 description | varchar(255)
```

但是，在业务规则中，要求这个领域类的 description 属性能够容纳 1000 个字符，在这种情况下，如果我们是使用 SQL 脚本，那么我们定义的这个列可能是：

```
column name | data type
 description | TEXT
```

现在我们又想要在基于应用程序的进行验证，_要求在持久化任何记录之前_，确保不能超过 1000 个字符。在 Grails 中，我们可以通过 constraints. 来完成，我们将在领域类中新增如下的约束声明.

```
static constraints = {
      description(maxSize:1000)
}
```

这个约束条件将会提供我们所需的基于应用程序的验证并且也将生成上述示例所示的数据库信息。下面是影响数据库生成的其他约束的描述。

影响字符串类型属性的约束

- inList
- maxSize
- size

如果 maxSize 或者 size 约束被定义, Grails 将根据约束的值设置列的最大长度.

通常, 不建议在同一个的领域类中组合使用这些约束. 但是, 如果你非要同时定义 maxSize 和 size 约束的话, Grails 将设置列的长度为 maxSize 约束和 size 上限约束的最少值. (Grails 使用两者的最少值，因此任何超过最少值的长度将导致验证错误.)

如果定义了 inList 约束 (maxSize 和 size 未定义), 字段最大长度将取决于列表（list）中最长字符串的的长度. 以"Java"、"Groovy"和"C++"为例, Grails 将设置字段的长度为 6（"Groovy"的最长含有 6 个字符）.

影响数值类型属性的约束

- min
- max
- range

如果定义了 max、min 或者 range 约束, Grails 将基于约束的值尝试着设置列的精度. (设置的结果很大程度上依赖于 Hibernate 跟底层数据库系统的交互程度.)

通常来说, 不建议在同一领域类的属性上组合成双的 min/max 和 range 约束，但是如果这些约束同时被定义了，那么 Grails 将使用约束值中的最少精度值. (Grails 取两者的最少值，是因为任意超过最少精度的长度将会导致一个验证错误.)

- scale

如果定义了 scale 约束, 那么 Grails 会试图使用基于约束的值来设置列的 标度（scale）. 此规则仅仅应用于浮点数值 (比如，java.lang.Float,java.Lang.Double, java.lang.BigDecimal 及其相关的子类). (设置的结果同样也是很大程度上依赖于 Hibernate 跟底层数据库系统的交互程度.)

约束定义着数值的最小/最大值, Grails 使用数字的最大值来设置其精度. 切记仅仅指定 min/max 约束中的一个，是不会影响到数据库的生成的 (因为可能会是很大的负值，比如当 max 是 100)，，除非指定的约束值要比 Hibernate 默认的精度（当前是 19）更高.比如...

```
someFloatValue(max:1000000, scale:3)
```

将产生:

```
someFloatValue DECIMAL(19, 3) // precision is default
```

但是

```
someFloatValue(max:12345678901234567890, scale:5)
```

将产生:

```
someFloatValue DECIMAL(25, 5) // precision = digits in max + scale
```

和

```
someFloatValue(max:100, min:-100000)
```

将产生:

```
someFloatValue DECIMAL(8, 2) // precision = digits in min + default scale
```

Show details

6.Web 层

6.1 控制器(Controllers)

一个控制器(Controllers)处理请求并创建或准备响应 ，是请求范围。 换句话说，会为每个 request 创建一个新的实体。 一个控制器(Controller)可以生成响应或委托给视图。 创建一个控制器(Controller)只需要创建一个以 Controller 结尾的类。并放置于 grails-app/controllers 目录下。

默认的 URL Mapping 设置确保控制器(Controllers)名字的第一个部分被映射到 URI 上，每个在控制器(Controllers)中定义的操作(Action)被映射到控制器(Controller)名字 URI 中的 URI。

6.1.1 理解控制器(Controller)与操作(Action)

创建控制器(Controller)

可以通过 create-controller 创建控制器(Controllers)。例如，你可以在 Grails 项目的根目录尝试运行下面命令:

```
grails create-controller book
```

这条命令将会在 grails-app/controllers/BookController.groovy 路径下创建一个控制器(Controller):

```
class BookController { ... }
```

BookController 默认被映射到 /book URI(相对于你应用程序根目录).

> create-controller 命令只不过是方便的工具，你同样可以使用你喜欢的文本编辑器或 IDE 更容易的创建控制器(Controller)

创建操作(Action)

一个控制器(Controllers) 可以拥有多个属性，每个属性都可以被分配一个代码块。所有这样的属性都被映射到 URI:

```
class BookController {
   def list = {
```

```
        // do controller logic
        // create model



        return model
    }
}
```

默认情况下，由于上面示例属性名被命名为 list 所以被映射到/book/list URI。

默认 Action

一个控制器(Controller)具有默认 URI 概念，即被映射到控制器(Controller)的根 URI。默认情况下，默认的 URI 是/book。 默认的 URI 通过下面的规则来规定:

- 如果只存在一个操作(Action)， 控制器(Controller)默认的 URI 映射为这个。
- 如果定义了 index 操作(Action)用于处理请求，并且没有操作(Action)在 URI /book 中指定
- 作为选择，你还可以明确的通过 defaultAction 属性来设置。 property:

```
def defaultAction = "list"
```

6.1.2 控制器(Controller) 与作用域

可用的作用域

作用域本质上就是 hash 对象，它允许你存储变量。 下面的作用域在控制器(Controller)中可以使用:

- servletContext - 也被称为 application 作用域, 这个作用域允许你在整个 web 应用程序中共享状态。 servletContext 对象为一个 javax.servlet.ServletContext 实体
- session - session 允许关联某个给定用户的状态，通常使用 Cookie 把一个 session 与一位客户关联起来。session 对象为一个 HttpSession 实体
- request -request 对象只允许存储当前的请求对象 。 request 对象为一个 HttpServletRequest 实体
- params - 可变的请求参数 map(CGI)。
- flash - 见下文。

存取作用域

作用域可以通过上面的变量名与 Groovy 数组索引操作符结合来进行存取。甚至是 Servlet API 提供的类，像 HttpServletRequest:

```
class BookController {
    def find = {
        def findBy = params["findBy"]
        def appContext = request["foo"]
        def loggedUser = session["logged_user"]
    }
}
```

你设置可以使用.操作符来存取作用域中的变量，这是语法更加清楚:

```
class BookController {
    def find = {
        def findBy = params.findBy
        def appContext = request.foo
        def loggedUser = session.logged_user
    }
}
```

这是 Grails 统一存取不同作用域的一种方式。

使用 Flash 作用域

Grails 支持 flash 作用域的概念，它只用于临时存储用于这个请求到下个请求的属性，然后，这个属性就会被清除对于重定向前直接设置消息是非常有用的，例如:

```
def delete = {
    def b = Book.get( params.id )
    if(!b) {
        flash.message = "User not found for id ${params.id}"
        redirect(action:list)
    }
    ... // remaining code
}
```

6.1.3 Models(模型)与 Views(视图)

Returning the Model

一个 model 本质上就是一个 map，在视图渲染时使用。map 中的 keys 转化为变量名，用于视图的获取。第一种方式是明确的 return 一个 model:

```
def show = {
      [ book : Book.get( params.id ) ]
}
```

如果没有明确的 model 被 return，控制器(Controller)的属性将会被视为 model。 所以允许你这样编写代码:

```
class BookController {
   List books
   List authors
   def list = {
         books = Book.list()
         authors = Author.list()
   }
}
```

> 这可能由于实际上控制器(Controller)是 prototype（原型）范围。换句话说，每个请求都会创建一个新的控制器(Controller)。 否则，像上面的代码，就不会是线程安全的。

上面示例中，books 和 authors 属性在视图中都是可用的。

一个更高级的方式就是 return 一个 Spring ModelAndView 类的实体:

```
import org.springframework.web.servlet.ModelAndView
def index = {
   def favoriteBooks = ... // get some books just for the index page, perhaps your favorites



   // forward to the list view to show them
   return new ModelAndView("/book/list", [ bookList : favoriteBooks ])
}
```

选择 View

在之前的 2 个示例中，都没有指定哪个 view 用于渲染。因此，Grails 怎么知道哪个 view 被选取?答案在于规约。对于 action:

```
class BookController {
        def show = {
                [ book : Book.get( params.id ) ]
        }
}
```

Grails 会自动查找位于 grails-app/views/book/show.gspview (事实上， Grails 会首先查找 JSP，因为，Grails 同样可以与 JSP 一起使用).

假如，你想渲染其他 view, render 方法在这里就能帮助你:

```
def show = {
        def map = [ book : Book.get( params.id ) ]
    render(view:"display", model:map)
}
```

这种情况下，Grails 将会尝试渲染位于 grails-app/views/book/display.gsp 的 view。注意，Grails 自动描述位于 book 文件夹中的 grails-app/views 路径位置的视图。很方便，但是，如果你拥有某些共享的视图需要存取，作为替代使用:

```
def show = {
        def map = [ book : Book.get( params.id ) ]
    render(view:"/shared/display", model:map)
}
```

在这种情况下，Grails 将尝试渲染 grails-app/views/shared/display.gsp 位置上的视图。

渲染响应

有时它很容易的渲染来自创建控制器小块文本或者代码的响应(通常使用 Ajax 应用程序)。因为，使用高度灵活的 render 方法:

```
render "Hello World!"
```

上面的代码，在响应中写入 "Hello World!"文本， 其他的示例包括:

```
// write some markup
render {
```

```
  for(b in books) {
    div(id:b.id, b.title)
  }
}
// render a specific view
render(view:'show')
// render a template for each item in a collection
render(template:'book_template', collection:Book.list())
// render some text with encoding and content type
render(text:"<xml>some xml</xml>",contentType:"text/xml",encoding:"UTF-8")
```

如果，你打算使用 Groovy 的 MarkupBuilder 来产生 html，可以使用 render 来避免 html 元素与 Grails 标签之间的命名冲突。例如:

```
def login = {
        StringWriter w = new StringWriter()
        def builder = new groovy.xml.MarkupBuilder(w)
        builder.html{
            head{
                title 'Log in'
            }
            body{
                h1 'Hello'
                form{
                }
            }
        }



        def html = w.toString()
        render html
    }
```

实际上调用 form 标签 (将返回一些文本，而忽略 MarkupBuilder). 为了正确的输出 <form>元素，使用下面这些:

```
def login = {
        // ...
        body{
            h1 'Hello'
```

```
        builder.form{
        }
    }
    // ...
}
```

6.1.4 重定向与链接

Redirects

使用 redirect 方法，Actions(操作)可在所有的控制器(Controller)中重定向:

```
class OverviewController {
        def login = {}
        def find = {
            if(!session.user)
                redirect(action:login)
            ...
        }
    }
```

redirect 方法内部使用 HttpServletResonse 对象的 sendRedirect 方法。

redirect 方法可以选择如下用法之一:

- 同一个控制器(Controller)类中的其他闭包:

```
// 调用同一个类的 login action
        redirect(action:login)
```

- 一个控制器(Controller)和一个操作(Action)的名字:

```
// 重定向到 home 控制器(Controller)的 index action
        redirect(controller:'home',action:'index')
```

- 相对于应用程序上下文路径的一个 URI 资源:

```
// 明确的重定向到 URI
        redirect(uri:"/login.html")
```

- 或者一个完整的 URL:

```
// 重定向到一个 URL
```

```
redirect(url:"http://grails.org")
```

使用方法的 params 参数，参数可以选择性的从一个 action 传递到下一个:

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

通过 params 动态属性，这些方法变得可用，同样也接受 request 参数。如果指定一个名字与 request 参数的名字相同的参数，则 request 参数被隐藏，控制器(Controller)参数被使用。

因为 params 对象也是一个 map，可以使用它把当前的 request 参数，从一个 action 传递到下一个:

```
redirect(action:"next", params:params)
```

最后，你也可以在一个目标 URI 上包含一个片段(fragment):

```
redirect(controller: "test", action: "show", fragment: "profile")
```

将(依靠 URL mappings) 导航到/myapp/test/show#profile"。

h4. 链接

Actions 同样可以被链接。链接允许 model 在一个操作(Action)到下一个操作(Action)中保留。例如下面调用 first action :

```
class ExampleChainController {
                def first = {
                   chain(action:second,model:[one:1])
                }
                def second  = {
                   chain(action:third,model:[two:2])
                }
                def third = {
                    [three:3])
                }
            }
```

model 的结果:

```
[one:1, two:2, three:3]
```

通过 chainModel map,这个 model 在 chain 中会被随后的 控制器(controller)操作(actions)存取. 这个动态属性只存在于随后调用 chain 方法的操作(actions)中:

```
class ChainController {
                def nextInChain = {
                   def model = chainModel.myModel
                   ...
                }
            }
```

Like the redirect method you can also pass parameters to the chain method:

```
chain(action:"action1", model:[one:1], params:[myparam:"param1"])
```

6.1.5 Controller(控制器) 拦截器

通常，它用于拦截基于每个 request（请求）,session(会话)或应用程序状态的数据处理，这可以通过 action(操作)拦截器来实现。目前有两种拦截器类型: before 和 after.

> 假如你的拦截器可能被用于更多的 controller(控制器), 几乎肯定会写一个更好的 Filter(过滤器). Filters(过滤器) 可以应用于多个 controllers(控制器)或 URIs, 无需改变任何 controller(控制器)逻辑.

Before 拦截器

beforeInterceptor 在 action (操作)被执行前进行数据处理拦截 . 假如它返回 false，那么 ，被拦截的 action (操作)将不会被执行. 拦截器可以像下面这样被定义为拦截一个 controller(控制器)中所有的 action (操作):

```
def beforeInterceptor = {
    println "Tracing action ${actionUri}"
}
```

上面是在 controller(控制器)定义主体内被声明. 它会在所有 action(操作)之前被执行，并且不会干扰数据处理. 一个普通的使用情形是为了验证:

```
def beforeInterceptor = [action:this.&auth,except:'login']
// defined as a regular method so its private
def auth() {
    if(!session.user) {
        redirect(action:'login')
        return false
    }
```

```
}
def login = {
    // display login page
}
```

上面的代码定义了一个名为 auth 的方法. 使用一个方法，是为了让它不会作为一个 action(操作)而暴露于外界(即. 它是 private). 随后，beforeInterceptor 定义用于'except' login actions(操作)之外的所有 actions(操作)的拦截，并告知执行'auth' 方法. 'auth' 方法是使用 Groovy 的方法指针语法来引用 ,在方法内部，它自己会检测是否一个用户在 session(会话)内，否则，重定向到 login action(操作) 并返回 false, 命令被拦截的 actions(操作)不被执行 .

After 拦截器

为了定义一个在 actions(操作)之后执行的拦截,可以使用 afterInterceptor 属性:

```
def afterInterceptor = { model ->
    println "Tracing action ${actionUri}"
}
```

after 拦截器把结果 model 作为参数，所以，可以执行 model 或 response 的 post 操作.

after 拦截器 也可以在渲染之前修改 Spring MVC ModelAndView 对象. 在这种情况下, 上面的示例变成:

```
def afterInterceptor = { model, modelAndView ->
    println "Current view is ${modelAndView.viewName}"
    if(model.someVar) modelAndView.viewName = "/mycontroller/someotherview"
    println "View is now ${modelAndView.viewName}"
}
```

通过当前 action(操作)，允许基于被返回的 model 改变视图. 注意，如果 action(操作)被拦截调用 redirect 或 render， modelAndView 可能为 null.

拦截条件

Rails 用户非常熟悉验证示例 ，以及如何在'except'条件的使用下执行拦截 (拦截器在 Rails 中被称为'过滤器', 这个术语与 Java 领域中的 servlet 过滤器术语有冲突):

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

除了被指定的 actions(操作)，它执行所有 actions(操作)的拦截. 一组 actions(操作)列表同样可以像下面这样被定义:

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

其他被支持的条件是'only', 它只对被指定的 actions(操作)执行拦截:

```
def beforeInterceptor = [action:this.&auth,only:['secure']]
```

6.1.6 数据绑定

数据绑定是"绑定"进入的请求参数到一个对象的属性或者一个完整对象图的行为. 数据绑定将处理所有来自请求参数必要的类型装换,典型的传送通过表单提交，始终是字符串,尽管 Groovy 或 Java 对象的属性可能不一定是.

Grails 使用 Spring's 底层的数据绑定能力来完成数据绑定.

绑定 Request 数据到 Model 上

这里有 2 种方式来绑定请求参数到 domain 类的属性上. 第一种涉及使用 domain 类的隐式构造函数:

```
def save = {
  def b = new Book(params)
  b.save()
}
```

这里的数据绑定发生在代码 new Book(params)内.通过传递 params 对象给 domain 类的构造函数, Grails 自动识别来自请求参数的绑定 . 因此，假如你有一个这样进入的请求 :

```
/book/save?title=The%20Stand&author=Stephen%20King
```

title 和 author 请求参数将会自动被设置到 domain 类上. 假如，你需要在一个已存在的实体上执行数据绑定,那么你可以使用 properties 属性:

```
def save = {
  def b = Book.get(params.id)
  b.properties = params
  b.save()
}
```

这个和使用隐式构造函数是完全一样的效果.

数据绑定和单向关联

如果你有 one-to-one 或 many-to-one 关联,你同样可以使用 Grails 的数据绑定能力更新这些关系. 例如，如果你有这样的请求参数:

/book/save?author.id=20

Grails 将自动检测请求参数上的 .id 后缀，并查找给定 id 的 Author 实体 ，随后像这样进行数据绑定:

```
def b = new Book(params)
```

属于绑定与 Many-ended 关联

假如你有一个 one-to-many 或 many-to-many 关联,依赖关联类型,有不同的方法用于数据绑定.

假如你有一个以 Set 基本的关联 (默认用于 hasMany) ,那么简单的方式加入一个关联是简单的传送一组标识符列表. 考虑下面 <g:select> 示例的用法:

```
<g:select name="books"
        from="${Book.list()}"
        size="5" multiple="yes" optionKey="id"
        value="${author?.books}" />
```

它生成一个选择框 ,允许你选择多个值. 在这种情况下,如果你提交表单,Grails 将自动利用来自选择框的标识符加入 books 关联.

不过, 假如,你有一个更新关联对象的属性的方案,这个方法将不会工作. 作为替代,你需要使用下标操作符:

```
<g:textField name="books[0].title" value="the Stand" />
<g:textField name="books[1].title" value="the Shining" />
```

不过, 如果，你想要更新在相同顺序中的渲染标记,对于基于 Set 的关联是危险的 . 这是因为 Set 没有顺序的概念, 所以,你引用的 books0 和 books1 不能确保关联的顺序在服务器端的正确性，除非你自己应用明确排序 .

如果你使用基于 List 的关联就不会存在这个问题，因为 List 拥有确定的顺序并使供索引来引用. 这同样适用于基于 Map 的关联.

还要注意 ,假如你绑定的关联长度为,你引用的元素超出了关联的长度:

```
<g:textField name="books[0].title" value="the Stand" />
<g:textField name="books[1].title" value="the Shining" />
<g:textField name="books[2].title" value="Red Madder" />
```

随后, Grails 在确定的位置自动为你创建一个实体. 如果你"跳过"中间的某些元素：

```
<g:textField name="books[0].title" value="the Stand" />
<g:textField name="books[1].title" value="the Shining" />
<g:textField name="books[5].title" value="Red Madder" />
```

随后,Grails 会自动在中间创建实体 . 例如,如果关联的长度为 2,在上面的情况下,Grails 会创建 4 个额外的实体.

数据绑定多个 domain 类

它可能通过来自 params 对象来绑定多个 domain 对象.

例如,你有一个进入的请求:

```
/book/save?book.title=The%20Stand&author.name=Stephen%20King
```

需要注意的是,上面请求不同之处在于拥有 author.前缀或 book 前缀. 这是用于分离哪个参数属于哪个类型. Grails 的 params 对象就像 多维 hash ,你可以索引来分离唯一的参数子集来绑定.

```
def b = new Book(params['book'])
```

注意,我们如何使用 book.title 的第一圆点前面的前缀参数来隔离唯一的参数绑定. 我们同样可以这样来使用 Authordomain 类 :

```
def a = new Author(params['author'])
```

数据绑定与类型转换错误

有时,当执行数据绑定时,它可能不会将一种指定的 String 转换为指定的目标类型. 你会得到类型转换错误. Grails 会保留类型转换错误在 Grails domain 类的 errors 属性中 . 例如这里:

```
class Book {
    ...
    URL publisherURL
}
```

这里，我们有一个 Bookdomain 类 ,它使用 Java 的 java.net.URL 来表示 URLs.现在,我们有

一个像这样的请求参数:

```
/book/save?publisherURL=a-bad-url
```

在这种情况下,它不可能将 字符串 a-bad-url 绑定到 publisherURL 属性上,一个类型匹配错误会发生. 你可以像这样来检查它们:

```
def b = new Book(params)
if(b.hasErrors()) {
   println "The value ${b.errors.getFieldError('publisherURL').rejectedValue} is not a valid URL!"
}
```

虽然,我们没有覆盖错误代码 (更多信息查看 Validation), 你需要的类型转换错误的错误消息在 grails-app/i18n/messages.properties 内. 你可以使用像下面这样的普通错误消息来处理：

```
typeMismatch.java.net.URL=The field {0} is not a valid URL
```

或更具体点:

```
typeMismatch.Book.publisherURL=The publisher URL you specified is not a valid URL
```

数据绑定与安全关系

当批量更新来自请求参数的属性,你必须小心，避免客户端绑定恶意数据到 domain 类上, 并持久化到数据库.你可以使用下标操作符限制捆绑在某个给定 domain 类的属性:

```
def p = Person.get(1)
p.properties['firstName','lastName'] = params
```

在这种情况下,只有 firstName 和 lastName 属性将被捆绑.

另一种实现这个的方式是使用 domain 类作为数据绑定目标,你可以使用 Command Objects. 另外还有一个更加灵活 bindData 方法.

The bindData 方法具有同样的数据绑定能力,但,是对于任意的对象:

```
def p = new Person()
bindData(p, params)
```

当然,bindData 方法同样允许你排除某些你不想更新的参数:

```
def p = new Person()
bindData(p, params, [exclude:'dateOfBirth'])
```

或只包含某些属性:

```
def p = new Person()
bindData(p, params, [include:['firstName','lastName]])
```

6.1.7 XML 与 JSON 响应

使用 render 方法输出 XML

Grails 支持一些不同的方法来产生 XML 和 JSON 响应. 第一个是通过 render 方法.

render 方法可以传递一个代码块来实现 XML 中的标记生成器:

```
def list = {
    def results = Book.list()
    render(contentType:"text/xml") {
        books {
            for(b in results) {
                book(title:b.title)
            }
        }
    }
}
```

这段代码的结果会像这样:

```
<books>
    <book title="The Stand" />
    <book title="The Shining" />
</books>
```

注意,你必须小心的是避免使用标记生成器带来的命名冲突. 例如,这段代码会产生一个错误:

```
def list = {
    def books = Book.list()  // naming conflict here
    render(contentType:"text/xml") {
        books {
            for(b in results) {
                book(title:b.title)
            }
        }
    }
```

```
}
```

问题在于,这里的局部变量 books, Groovy 会把它当做一个方法来调用.

使用 render 方法输出 JSON

render 同样被用于输出 JSON:

```
def list = {
    def results = Book.list()
    render(contentType:"text/json") {
        books {
            for(b in results) {
                book(title:b.title)
            }
        }
    }
}
```

在这种情况下,结果大致相同:

```
[
    {title:"The Stand"},
    {title:"The Shining"}
]
```

同样的命名冲突危险适用于 JSON 生成器.

自动 XML 列集(Marshalling)

（译者注：在此附上对于列集(Marshalling)解释：对函数参数进行打包处理得过程，因为指针等数据，必须通过一定得转换，才能被另一组件所理解。可以说列集(Marshalling)是一种数据格式的转换方法。）

Grails 同样支持自动列集(Marshalling) domain 类 为 XML,通过特定的转换器.

首先，导入 grails.converters 类包到你的 controller(控制器):

```
import grails.converters.*
```

现在，你可以使用下列高度易读的语法来自动转换 domain 类为 XML:

```
render Book.list() as XML
```

输出结果看上去像下面这样:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<list>
  <book id="1">
    <author>Stephen King</author>
    <title>The Stand</title>
  </book>
  <book id="2">
    <author>Stephen King</author>
    <title>The Shining</title>
  </book>
</list>
```

一个使用转换器的替代方法是使用 Grails 的 codecs 特性. codecs 特性提供了 encodeAsXML 和 encodeAsJSON 方法:

```
def xml = Book.list().encodeAsXML()
render xml
```

更多的 XML 列集(Marshalling)信息见 REST

自动 JSON 列集(Marshalling)

Grails 同样支持自动列集(Marshalling)为 JSON 通过同样的机制. 简单替代 XML 为 JSON:

```
render Book.list() as JSON
```

输出结果看上去像下面这样:

```
[
       {"id":1,
        "class":"Book",
        "author":"Stephen King",
        "title":"The Stand"},
       {"id":2,
        "class":"Book",
        "author":"Stephen King",
        "releaseDate":new Date(1194127343161),
        "title":"The Shining"}
]
```

作为替代，你可以使用 encodeAsJSON 达到相同的效果.

6.1.8 文件上传

文件上传程序

Grails 通过 Spring 的 MultipartHttpServletRequest 接口来支持文件上传. 上传文件的第一步就是像下面这样创建一个 multipart form:

```
Upload Form: <br />
      <g:form action="upload" method="post" enctype="multipart/form-data">
            <input type="file" name="myFile" />
            <input type="submit" />
      </g:form>
```

这里有一些方法来处理文件上传. 第一种方法是直接与 Spring 的 MultipartFile 实体:

```
def upload = {
   def f = request.getFile('myFile')
   if(!f.empty) {
     f.transferTo( new File('/some/local/dir/myfile.txt') )
     response.sendError(200,'Done');
   }
   else {
      flash.message = 'file cannot be empty'
      render(view:'uploadForm')
   }
}
```

这显然很方便,通过 MultipartFile 接口可以直接获得一个 InputStream，用来转移到其他目的地和操纵文件等等.

通过数据绑定上传文件

文件上传同样可以通过数据绑定来完成。例如，假定你有一个像下面这样 Image domain 类:

```
class Image {
  byte[] myFile
}
```

现在，假如你创建一个 image 并像下面这个示例一样传入 params 对象，Grails 将自动把文件的内容当作一个 byte 绑定到 myFile 属性:

```
def img = new Image(params)
```

它同样可以设置文件的内容为一个 string，通过改变 image 的 myFile 属性类型为一个 String 类型:

```
class Image {
  String myFile
}
```

6.1.9 命令对象

Grails 控制器(controllers)支持命令对象概念.一个命令对象类似于 Struts 中的一个 formbean,它们在当你想要写入属性子集来更新一个 domain 类情形时是非常有用的 . 或在没有 domain 类需要的相互作用，但必须使用 data binding 和 validation 特性 .

声明命令对象

命令对象通常作为一个控制器直接声明在控制器(controller)类定义下的同一个源文件中. 例如:

```
class UserController {
      ...
}
class LoginCommand {
  String username
  String password
  static constraints = {
        username(blank:false, minSize:6)
        password(blank:false, minSize:6)
  }
}
```

上面的示例证明你可以提供 约束给命令对象,就象你在 domain 类中的用法一样.

使用命令对象

为了使用命令对象，控制器可以随意指定任何数目的命令对象参数。必须提供参数的类型以至于 Grails 能知道什么样的对象被创建，写入和验证.

在控制器(controller)的操作被执行之前，Grails 将自动创建一个命令对象类的实体，用相

应名字的请求参数写入到命令对象属性，并且命令对象将被验证，例如:

```
class LoginController {
  def login = { LoginCommand cmd ->
        if(cmd.hasErrors()) {
              redirect(action:'loginForm')
        }
        else {
           // do something else
       }
  }
}
```

命令对象与依赖注入

命令对象可以参与依赖注入。这有利于一些定制的验证逻辑与 Grails 的 services 的结合。：

```
class LoginCommand {
    def loginService
        String username
        String password



        static constraints = {
              username(validator: { val, obj ->
                    obj.loginService.canLogin(obj.username, obj.password)
              })
        }
}
```

上面示例，命令对象与一个来自 Spring 的 ApplicationContext 注入名字 bean 结合.

6.1.10 处理重复的表单提交

Grails 已经内置支持处理重复表单提交, 通过使用"同步令牌模式". 首先,你得在 form 标签上定义一个令牌:

```
<g:form useToken="true" ...>
```

随后,在你的控制器(controller)代码中使用 withForm 方法来处理有效和无效的请求:

```
withForm {
  // good request
}.invalidToken {
  // bad request
}
```

如果你只提供了 withForm 方法而没有链接 invalidToken 方法,那么,默认情况下,Grails 将会无效的令牌存储在 flash.invalidToken 变量中并导航请求回到原始页面. 这可以在页面中检测到:

```
<g:if test="${flash.invalidToken}">
  Don't click the button twice!
</g:if>
```

> withForm 标签利用了 session ,因此,如果在群集中使用,要求会话密切关联.

6.2 Groovy Server Pages

Groovy Servers Pages (或者简写为 GSP)Grails 的视图技术。它被设计成像 ASP 和 JSP 这样被使用者熟悉的技术，但更加灵活和直观.

GSP 存在于 Grails 的 grails-app/views 目录中，他们通常会自动渲染（通过规约），或者像这样通过 render 方法:

```
render(view:"index")
```

GSP 使典型的混合标记和 GSP 标签,辅助页面渲染.

> 虽然，它可能会在你的 GSP 页面中内置 Groovy 逻辑,Although it is possible to have Groovy logic embedded in your GSP and doing this will be covered in this document the practice is strongly discouraged. Mixing mark-up and code is a **bad** thing and most GSP pages contain no code and needn't do so.

一个 GPS 通常拥有一个"model"，它是变量集被用于视图渲染。通过一个控制器 model 被传递到 GSP 视图。例如，考虑下列控制器的操作:

```
def show = {
      [book: Book.get(params.id)]
}
```

这个操作将查找一个 book 实体，并创建一个包含关键字为 Book 的 model,这个关键字可在随后的 GSP 视图中应用:

```
<%=book.title%>
```

6.2.1 GSP 基础

在下一节，我们将通过 GSP 基础知识让你知道它能做什么。首先，我们将涵盖基础语法，对于 JSP 和 ASP 用户是非常熟悉的.

GSP 支持使用<% %>来嵌入 Groovy 代码(这是不推荐的):

```
<html>
  <body>
    <% out << "Hello GSP!" %>
  </body>
</html>
```

同样，你可以使用<%= %>语法来输出值:

```
<html>
  <body>
    <%="Hello GSP!" %>
  </body>
</html>
```

GSP 同样支持服务器端 JSP 样式注释，像下列示例显示的这样:

```
<html>
  <body>
      <%-- This is my comment --%>
    <%="Hello GSP!" %>
  </body>
</html>
```

6.2.1.1 变量与作用域

在 <% %> 中你当然可以声明变量:

```
<% now = new Date() %>
```

然后，在页面中的之后部分可以重复使用 :

```
<%=now%>
```

然而, 在 GSP 中存在着一些预先定义的变量，包括:

- 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

6.2.1.2 逻辑和迭代

使用 <% %> 语法，你当然可以使用这样的语法进行嵌套循环等等操作:

```
<html>
  <body>
    <% [1,2,3,4].each { num -> %>
      <p><%="Hello ${num}!" %></p>
    <%}%>
  </body>
</html>
```

同样可以分支逻辑:

```
<html>
  <body>
    <% if(params.hello == 'true' )%>
    <%="Hello!"%>
    <% else %>
    <%="Goodbye!"%>
  </body>
</html>
```

6.2.1.3 页面指令

GSP 同样支持少许的 JSP 样式页面指令.

import 指令允许在页面中导入类。然而，它却很少被使用，因为 Groovy 缺省导入和 GSP 标签:

```
<%@ page import="java.awt.*" %>
```

GSP 同样支持 contentType@ 指令:

```
<%@ page contentType="text/json" %>
```

contentType@指令允许 GSP 使用其他的格式来渲染.

6.2.1.4 表达式

尽管 GSP 也支持 <%= %> 语法，而且很早就介绍过，但在实际当中却很少应用，因为此用法主要是为 ASP 和 、JSP 开发者所保留的。 而 GSP 的表达式跟 JSP EL 表达式很相似的，跟 Groovy GString 的 ${expr} 用法也很像:

```
<html>
  <body>
    Hello ${params.name}
  </body>
</html>
```

尽管如此，跟 JSP EL 不同的是， 你可以在${..}括号中使用 Groovy 表达式.${..}中的变量缺省情况下是**非**转义，因此变量的任何 HTML 字符串内容被直接输出到页面，要减少这种 Cross-site-scripting (XSS)攻击的风险， 你可以设置 grails-app/conf/Config.groovy 中的 grails.views.default.codec 为 HTML 转化方式:

```
grails.views.default.codec='html'
```

其他可选的值是'none' (缺省值)和'base64'.

6.2.2 GSP 标签

现在，JSP 遗传下来的缺点已经被取消，下面的章节将涵盖 GSP 的内置标签，它是定义 GSP 页面最有利的方法.

标签库 部分涵盖怎么添加你自己的定制标签库.

所有 GSP 内置标签以前缀 g:开始。 不像 JSP，你不需要指定任何标签库的导入.假如,一个标签以 g:开始,它被自动认为是一个 GSP 标签.一个 GPS 标签的示例看起来像这样:

```
<g:example />
```

GSP 标签同样可以拥有主体,像这样:

```
<g:example>
  Hello world
</g:example>
```

表达式被传递给 GSP 标签属性，假如没有使用表达式,将被认为是一个 String 值:

```
<g:example attr="${new Date()}">
  Hello world
</g:example>
```

Maps 同样能被传递给 GSP 标签属性，通常使用一个命名参数样式语法:

```
<g:example attr="${new Date()}" attr2="[one:1, two:2, three:3]">
  Hello world
</g:example>
```

注意，对于 String 类型属性值，你必须使用单引号:

```
<g:example attr="${new Date()}" attr2="[one:'one', two:'two']">
  Hello world
</g:example>
```

在介绍完基本的语法之后，下面我们来讲解 Grails 中默认提供的标签.

6.2.2.1 变量与作用域

变量可以在 GSP 中使用 set 标签来定义:

```
<g:set var="now" value="${new Date()}" />
```

这里， 我们给 GSP 表达式结果赋予了一个名为 now 的变量 (简单的构建一个新的 java.util.Date 实体)。 你也可以在<g:set>主体中定义一个变量:

```
<g:set var="myHTML">
  Some re-usable code on: ${new Date()}
```

```
</g:set>
```

变量同样可以被放置于下列的范围内:

- page - 当前页面范围 (默认)
- request - 当前请求范围
- flash - flash 作用域，因此它可以在下一次请求中有效
- session - 用户 session 范围
- application - 全局范围.

选择变量被放入的范围可以使用 scope 属性:

```
<g:set var="now" value="${new Date()}" scope="request" />
```

6.2.2.2 逻辑和迭代

GSP 同样支持迭代逻辑标签，逻辑上通过使用 if , , else 和 elseif 来支持典型的分支情形。 :

```
<g:if test="${session.role == 'admin'}">
  <%-- show administrative functions --%>
</g:if>
<g:else>
  <%-- show basic functions --%>
</g:else>
```

GSP 用 each each 和 while 标签来处理迭代:

```
<g:each in="${[1,2,3]}" var="num">
  <p>Number ${num}</p>
</g:each>
<g:set var="num" value="${1}" />
<g:while test="${num < 5 }">
   <p>Number ${num++}</p>
</g:while>
```

6.2.2.3 搜索和过滤

假如你拥有对象集合，你经常需要使用一些方法来排序和过滤他们。 GSP 支持 findAll 和 grep 来做这些工作:

```
Stephen King's Books:
<g:findAll in="${books}" expr="it.author == 'Stephen King'">
    <p>Title: ${it.title}</p>
</g:findAll>
```

expr 属性包含了一个 Groovy 表达式，它可以被当作一个过滤器来使用。谈到过滤器，grep 标签通过类来完成与过滤器类似的工作:

```
<g:grep in="${books}" filter="NonFictionBooks.class">
    <p>Title: ${it.title}</p>
</g:grep>
```

或者使用一个正则表达式:

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

上面的示例同样有趣，因为它使用了 GPath.Groovy 的 GPath 等同与 XPath 语言。实际上 books 集合是 books 集合的实体。不过，假设每个 books 拥有一个 title,你可以使用表达式 books.title 来获取 Book titles 的 list!

6.2.2.4 链接和资源

GSP 还拥有特有的标签来帮助你管理连接到控制器和操作. link 标签允许你指定控制器和操作配对的名字，并基于 URL Mappings 映射来自动完成连接。即使你去改变！一些 link 的示例如下:

```
<g:link action="show" id="1">Book 1</g:link>
<g:link action="show" id="${currentBook.id}">${currentBook.name}</g:link>
<g:link controller="book">Book Home</g:link>
<g:link controller="book" action="list">Book List</g:link>
<g:link url="[action:'list',controller:'book']">Book List</g:link>
<g:link action="list" params="[sort:'title',order:'asc',author:currentBook.author]">
    Book List
</g:link>
```

6.2.2.5 表单和字段

表单基础

GSP 支持许多不同标签来帮助处理 HTML 表单和字段，最基础的是 form 标签， form 标签是一个控制器/操作所理解的正规的 HTML 表单标签版本。 url 属性允许你指定映射到哪个控制器和操作:

```
<g:form name="myForm" url="[controller:'book',action:'list']">...</g:form>
```

我们创建个名为 myForm 的表单，它被提交到 BookController 的 list 操作。除此之外，适用于所有不同的 HTML 属性.

表单字段

同构造简单的表单一样，GSP 支持如下不同字段类型的定制:

- textField - 'text'类型输入字段
- checkBox - 'checkbox'类型输入字段
- radio - 'radio'类型输入字段
- hiddenField - 'hidden'类型输入字段
- select - 处理 HTML 选择框

上面的每一个都允许 GSP 表达式作为值:

```
<g:textField name="myField" value="${myValue}" />
```

GSP 同样包含上面标签的扩张助手版本， 比如 radioGroup (创建一组 radio 标签), localeSelect, currencySelect 和 timeZoneSelect(选择各自的地区区域, 货币 和时间区域). .

多样的提交按钮

处理多样的提交按钮这样由来已久的问题，同样可以通过 Grails 的 actionSubmit 标签优雅的处理。它就像一个正规提交，但是，允许你指定一个可选的操作来提交:

```
<g:actionSubmit value="Some update label" action="update" />
```

6.2.2.6 标签作为方法调用

GSP 标签和其他标签技术一个主要不同在于，来自 controllers(控制器)，标签库 或者 GSP 视图中的 GPS 标签可以被当作任意的正规标签或者当作方法被调用.

来自 GSPs 中的标签当作方法调用

当作为方法被调用时，标签的返回值被当作 String 实体直接被写入响应中。 因此，示例中的 createLinkTo 能等同的看做方法调用:

```
Static Resource: ${createLinkTo(dir:"images", file:"logo.jpg")}
```

当你必须在一个属性内使用一个标签时是特别有用的:

```
<img src="${createLinkTo(dir:'images', file:'logo.jpg')}" />
```

I 在视图技术中，标签内嵌套标签的特性是不被支持的，这样变得十分混乱，往往使得像 Dreamweaver 这样 WYSWIG 的工具产生不利的效果以至于在渲染标签时:

```
<img src="<g:createLinkTo dir="images" file="logo.jpg" />" />
```

来自控制器（Controllers）和标签库的标签作为方法调用

你同样可以调用来自控制器和标签库的标签。标签可以不需要内部默认的 g:namespace 前缀来调用，并返回 String 结果:

```
def imageLocation = createLinkTo(dir:"images", file:"logo.jpg")
```

然而，你同样可以用命名空间前缀来避免命名冲突:

```
def imageLocation = g.createLinkTo(dir:"images", file:"logo.jpg")
```

假如你有一个自定义命名空间,，你可以使用它的前缀来替换（例如，使用 FCK Editor plugin:

```
def editor = fck.editor()
```

6.2.3 视图(View)与模板(Templates)

除了 views 之外, Grails 还有模板的概念. 模板有利于分隔出你的视图在可维护的块中，并与 Layouts 结合提供一个高度可重用机制来构建视图.

模板基础

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，你可能有个位于 grails-app/views/book/_bookTemplate.gsp 的模板处理渲染 Books:

```
<div class="book" id="${book?.id}">
  <div>Title: ${book?.title}</div>
  <div>Author: ${book?.author?.name}</div>
</div>
```

为了渲染来自 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]" />
```

注意，我们是怎么样使用 render 标签的 model 属性来使用传入的一个 model。假如，你有多个 Book 实体，你同样可以使用 render 标签为每个 Book 渲染模板 :

```
<g:render template="bookTemplate" var="book" collection="${bookList}" />
```

共享模板

在早先的示例中，我们有一个特定于 BookController 模板，它的视图位于 grails-app/views/book.然而，你可能想横跨你的应用来共享模板。

在这种情况下,你可以把他们放置于 grails-app/views 视图根目录或者位于这个位置的任何子目录，然后在模板属性在模板名字之前使用一个 /来指明相对模板路径 .例如,假如你有个名为 grails-app/views/shared/_mySharedTemplate.gsp 模板, 你可以像下面这样引用它:

```
<g:render template="/shared/mySharedTemplate" />
```

你也可以使用这个技术从任何视图或控制器（Controllers）来引用任何目录下的模板:

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

模板命名空间

因为模板使用如此频繁,它有一个模板命名空间, 名为 tmpl, 他使模板的使用变得容易. 考虑下面例子的使用模式:

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

这个想下面这样通过 tmpl 命名空间表示 :

```
<tmpl:bookTemplate book="${myBook}" />
```

在控制器(Controllers)和标签库中的模板

你同样可以使用控制器 render 方法渲染模板控制器中，它对 Ajax 引用很有用:

```
def show = {
   def b = Book.get(params.id)
      render(template:"bookTemplate", model:[book:b])
}
```

在控制器(controller)中的 render 方法最普通的行为是直接写入响应。 假如，你需要获得模板作为一个 String 的结果作为替代，你可以使用 render 标签:

```
def show = {
   def b = Book.get(params.id)
      String content = g.render(template:"bookTemplate", model:[book:b])
      render content
}
```

注意， g. 命名空间的用法，它告诉 Grails 我们想使用标签作为方法调用来代替 render 方法.

6.2.4 使用 Sitemesh 布局

创建布局

Grails 利用了 Sitemesh，一个装饰引擎，来支持视图布局。布局位于 grails-app/views/layouts 目录中。一个典型的布局如下:

```
<html>
    <head>
      <title><g:layoutTitle default="An example decorator" /></title>
      <g:layoutHead />
    </head>
    <body onload="${pageProperty(name:'body.onload')}">
        <div class="menu"><!--my common menu goes here--></menu>
            <div class="body">
                <g:layoutBody />
            </div>
        </div>
    </body>
</html>
```

关键的元素是 layoutHead, layoutTitle 和 layoutBody 标签的用法，这里是他们所做的:

- layoutTitle - 输出目标页面的 title
- layoutHead - 输出目标页面 head 标签内容
- layoutBody - 输出目标页面 body 标签内容

上面的示例同样表明 pageProperty tag 可被用于检查和返回目标页面的外观.

启用布局

这里有一些方法来启用一个布局.简单的在视图中添加 meta 标签:

```
<html>
   <head>
         <title>An Example Page</title>
      <meta name="layout" content="main"></meta>
   </head>
   <body>This is my content!</body>
</html>
```

在这种情况下，一个名为 grails-app/views/layouts/main.gsp 将被用于布局这个页面。假如，我们使用来自早前部分的布局，输出看上去像下列这样:

```
<html>
    <head>
      <title>An Example Page</title>
    </head>
    <body onload="">
     <div class="menu"><!--my common menu goes here--></div>
           <div class="body">
                        This is my content!
           </div>
    </body>
</html>
```

在控制器(Controller)中指定布局

另一种用于指定布局的方式是通过在控制器(controller)中为 "layout"属性指定布局的名字, 假如你有个这样的控制器(controller):

```
class BookController {
   static layout = 'customer'
```

```
    def list = {  ... }
}
```

你可以创建一个 grails-app/views/layouts/customer.gsp 布局，应用于所有 BookController 中委派的视图 . "layout"属性值可能包含相对于 grails-app/views/layouts/目录的路径结构 . 例如:

```
class BookController {
    static layout = 'custom/customer'
    def list = {  ... }
}
```

视图的显然可通过 grails-app/views/layouts/custom/customer.gsp 模板.

布局规约

第二种关联布局的方法是使用"布局规约"，假如你有个这样的控制器:

```
class BookController {
    def list = {  ... }
}
```

你可以创建一个名为 grails-app/views/layouts/book.gsp 的布局,根据规约,它将被应用于 BookController 的所有视图中。

换句话说，你可以创建一个名为 grails-app/views/layouts/book/list.gsp 的布局,它将只被应用于 BookController 中的 list 操作，

如果你同时使用了以上提到的两种布局的话,那当 list 操作被执行的时候,那么操作将根据优先级的顺序来使用布局.

内联布局

通过 applyLayout 标签 Grails 同样支持 Sitemesh 的内联布局概念。 applyLayout 标签可以被用于应用一个布局到一个模板，URL 或者内容的任意部分。事实上，通过"decorating"你的模板允许你更进一步的积木化你的视图结构.

一些使用示例如下:

```
<g:applyLayout name="myLayout" template="bookTemplate" collection="${books}" />
<g:applyLayout name="myLayout" url="http://www.google.com" />
```

```
<g:applyLayout name="myLayout">
The content to apply a layout to
</g:applyLayout>
```

Server-Side 包含

当 applyLayout 标签被以用于引用布局外内容 applying layouts to，假如你想简单的在当前页面包含外部内容，你可以使用 include:

```
<g:include controller="book" action="list"></g:include>
```

你甚至可以结合 include 标签和 applyLayout 标签来添加灵活性:

```
<g:applyLayout name="myLayout">
  <g:include controller="book" action="list"></g:include>
</g:applyLayout>
```

最后,你也可以在控制器(controller)或标签库把 include 标签作为方法调用 :

```
def content = include(controller:"book", action:"list")
```

最后的内容有 include 标签的返回值提供 .

6.2.5 Sitemesh 内容块

虽然，这对于装饰全部页面非常有用,有时,你需要装饰站点的部分独自的页面。为了实现这个可以使用内容块. 在开始时，你需要使用 <content>标签分隔装饰页面 :

```
<content tag="navbar">
... draw the navbar here...
</content>
<content tag="header">
... draw the header here...
</content>
<content tag="footer">
... draw the footer here...
</content>
<content tag="body">
... draw the body here...
</content>
```

随后,在布局内部，你可以引用这些组件并为每个引用单个布局:

```
<html>
	<body>
		<div id="header">
			<g:applyLayout name="headerLayout"><g:pageProperty
name="page.header"></g:applyLayout>
		</div>
		<div id="nav">
			<g:applyLayout name="navLayout"><g:pageProperty
name="page.navbar"></g:applyLayout>
		</div>
		<div id="body">
			<g:applyLayout name="bodyLayout"><g:pageProperty
name="page.body"></g:applyLayout>
		</div>
		<div id="footer">
			<g:applyLayout name="footerLayout"><g:pageProperty
name="page.footer"></g:applyLayout>
		</div>
	</body>
</html>
```

6.3 标签库

像 Java Server Pages JSP) 一样，GSP 支持定制 tag 库的概念.不同于 JSP,Grails 标签库机制是简单的,优雅的,在运行时完全可重载的.

创建一个标签库是相当简单的,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:

```
class SimpleTagLib {
}
```

现在,为了创建一个标签,简单的创建属性并赋值一个带有两个参数的代码块：标签属性和主体内容:

```
class SimpleTagLib {
	def simple = { attrs, body ->
    }
}
```

attrs 属性是一个简单的标签属性 map，同时 body 是另一可调用的代码块，它返回主体内容:

```
class SimpleTagLib {
       def emoticon = { attrs, body ->
         out << body() << attrs.happy == 'true' ? " :-)" : " :-("
    }
}
```

正如以上所显示的,这里有个隐式的 out 变量,它引用了输出 Writer,可以用来附加内容到响应中. 然后,你可以在你的 GSP 内简单的引用这个标签而不需要任何导入:

```
<g:emoticon happy="true">Hi John</g:emoticon>
```

6.3.1 变量与作用域

在标签库的作用域中包含了一些预先定义好的变量:

- actionName - 当前执行的操作(action)名
- controllerName - 当前执行的控制器(controller)名
- flash - The flash 对象
- grailsApplication - The GrailsApplication 实体
- out - The response writer for writing to the output stream
- pageScope - pageScope 对象引用,用于 GSP 渲染(即. binding)
- params - The params 对象,用于取得请求参数
- pluginContextPath - 插件上下文路径 ,它包含标签库
- request - HttpServletRequest 实体
- response - HttpServletResponse 实体
- servletContext - javax.servlet.ServletContext 实体
- session - HttpSession 实体

6.3.2 简单标签

作为演示，早先的示例只不过是写了个没有主体只有输出内容的简单标签。另一个示例是一个 dateFormat 样式标签:

```
def dateFormat = { attrs, body ->
       out << new java.text.SimpleDateFormat(attrs.format).format(attrs.date)
```

```
}
```

上面使用了 Java 的 SimpleDateFormat 类来格式化一个 date，然后把它写入响应。随后，这个标签能像下列这样在 GSP 中使用:

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

有时。你需要用简单的标签把 HTML 标签（mark-up）写入到响应中。一个方法是直接嵌套内容:

```
def formatBook = { attrs, body ->
   out << "<div id="${attrs.book.id}">"
   out << "Title : ${attrs.book.title}"
      out << "</div>"
}
```

虽然，这个方法可能很诱人，但不是非常的简洁。一个更好的方法将是复用 render 标签:

```
def formatBook = { attrs, body ->
   out << render(template:"bookTemplate", model:[book:attrs.book])
}
```

然后，这个单独的 GSP 模板做了实际的渲染工作.

6.3.3 逻辑标签

一旦一组条件满足，你同样可以在标签的主体中创建仅仅用来输出的逻辑标签。一个这样的例子可能是一组安全标签:

```
def isAdmin = { attrs, body ->
    def user = attrs['user']
    if(user != null && checkUserPrivs(user)) {
        out << body()
    }
}
```

上面的标签检查用户是否为管理人员，如果他/她有正确设置的访问权限只输出主体内容:

```
<g:isAdmin user="${myUser}">
   // some restricted content
</g:isAdmin>
```

6.3.4 迭代标签

迭代标签同样普通，因为你可以多次调用主体:

```
def repeat = { attrs, body ->
   attrs.times?.toInteger().times { num ->
      out << body(num)
   }
}
```

在这个示例中，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:

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

注意，我们是怎么样在这个示例中使用隐式的 it 变量来引用当前的数字。这个过程是因为在迭代内部我们调用了传递进入当前值的主体:

```
out << body(num)
```

那个值然后被作为默认的 it 变量传递给标签，然而，假如你有嵌套标签便会导致冲突，因此，你将可能替换主体使用的变量名:

```
def repeat = { attrs, body ->
      def var = attrs.var ? attrs.var : "num"
   attrs.times?.toInteger().times { num ->
      out << body((var):num)
   }
}
```

这里，我们检查是否存在一个 var 属性，如果存在的话，将其作为 body 调用的参数:

```
out << body((var):num)
```

> 注意，变量名围绕的圆括号的使用.假如你省略,Groovy 会认为你使用了一个 String 关键字,而不是引用这个变量它自己.

现在，我们可以改变这个标签的使用方法，如下:

```
<g:repeat times="3" var="j">
<p>Repeat this 3 times! Current repeat = ${j}</p>
```

```
</g:repeat>
```

注意，我们是怎么样使用 var 属性来定义 j 变量名，随后，我们可能在标签主体类引用这个变量.

6.3.5 标签命名空间

默认情况下，标签被添加到默认的 Grails 命名空间，并在 GSP 页面中和 g: 前缀一起使用。然而，你可以指定一个不同的命名空间，通过在你的 TagLib 类中添加一个静态属性:

```
class SimpleTagLib {
    static namespace = "my"
    def example = { attrs ->
        ...
    }
}
```

这里，我们指定了一个命名空间 my，因此，稍后在 GPS 页面中标签库中的标签引用会像这样:

```
<my:example name="..." />
```

前缀和静态的命名空间属性值一样.命名空间对于插件特别有用.

命名空间内的标签可以作为方法调用，使用命名空间作为前缀来执行方法调用:

```
out << my.example(name:"foo")
```

可用于 GSP，控制器或者标签库.

6.3.6 使用 JSP 标签库

除了 GSP 提供的简单标签库机制, 你也可以在 GSP 中使用 JSP 标签.通过 taglib 指令来简单声明你需要的 JSP 标签:

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

随后,你可以想任何其他标签一样来使用它:

```
<fmt:formatNumber value="${10}" pattern=".00"/>
```

额外的好处是，你可以把 JSP 标签当方法调用：

${fmt.formatNumber(value:10, pattern:".00")}

6.4 URL 映射

到目前为止，贯穿整个文档用于 URLs 的规约默认为 /controller/action/id . 然而，这个规约不是硬性的写入 Grails 中，实际上，它是通过一个位于 grails-app/conf/UrlMappings.groovy 的 URL 映射类所控制.

UrlMappings 类包含一个名为 mappings 单一属性，并被赋予一个代码块:

```
class UrlMappings {
   static mappings = {
   }
}
```

6.4.1 映射到控制器和操作

为了创建简单的映射，只需简单的使用相对 URL 作为方法名，并指定控制器和操作的命名参数来映射:

```
"/product"(controller:"product", action:"list")
```

在这种情况下，我们建立 URL/product 到 ProductController 的 list 操作的映射。你当然可以省略操作定义，来映射控制器默认的操作:

```
"/product"(controller:"product")
```

一个可选的语法是把在块中被赋值的控制器和操作传递给方法:

```
"/product" {
      controller = "product"
      action = "list"
}
```

你使用哪一个句法很大程度上依赖于个人偏好.

6.4.2 嵌入式变量

简单变量

早前的部分说明，怎样使用具体的"标记"来映射普通的 URLs。在 URL 映射里讲过，标记是在每个斜线(/)字符之间的顺序字符。 一个具体的标记就像/product 这样被良好定义。然而，很多情况下，标记的值直到运行时才知道是什么。在这种情况下，你可以在 URL 中使用变量占位符，例如:

```
static mappings = {
  "/product/$id"(controller:"product")
}
```

在这种情况下，通过嵌入一个$id 变量作为第 2 个标记，Grails 将自动映射第 2 个标记到一个名为 id 的参数(通过 params 对象得到). 例如给定的 URL/product/MacBook，下面的代码将渲染"MacBook"到响应中:

```
class ProductController {
    def index = { render params.id }
}
```

当然你可以构建更多复杂的映射示例。例如传统的 blog URL 格式将被映射成下面这样:

```
static mappings = {
   "/$blog/$year/$month/$day/$id"(controller:"blog", action:"show")
}
```

上面的映射允许你这样:

```
/graemerocher/2007/01/10/my_funky_blog_entry
```

在 URL 里单独的标记将再次被映射到带有 year, month, day, id 等等可用值的 params 对象中.

动态控制器(Controller)和操作(Action)名

变量同样可以被用于动态构造控制器和操作名。实际上，默认的 Grails URL 映射使用这样的技术:

```
static mappings = {
   "/$controller/$action?/$id?"()
}
```

这里，控制器(controller)名，操作(action)名和 id 名，隐式的从嵌入在 URL 中的 controller, action 和 id 中获得:

```
static mappings = {
    "/$controller" {
          action = { params.goHere }
    }
}
```

可选的变量

默认映射另一个特性就是能够在一个变量的末尾附加一个？，使它成为一个可选的标记。这个技术更进一步的示例能够运用于 blog URL 映射，使它具有更灵活性的连接 :

```
static mappings = {
   "/$blog/$year?/$month?/$day?/$id?"(controller:"blog", action:"show")
}
```

下列 URLs 的所有映射将与放置于 params 对象中的唯一关联的参数匹配:

```
/graemerocher/2007/01/10/my_funky_blog_entry
/graemerocher/2007/01/10
/graemerocher/2007/01
/graemerocher/2007
/graemerocher
```

任意变量

你同样可以传递来自于 URL 映射的任意参数给控制器，把他们设置在块内传递给这个映射:

```
"/holiday/win" {
    id = "Marrakech"
    year = 2007
}
```

在这个 params 对象得到的这个变量将被传递给这个控制器.

动态解析变量

硬编码任意变量是有用的，但是，有时你需要基于运行时因素来计算变量名。这个同样可能通过给变量名分配一个块:

```
"/holiday/win" {
    id = { params.id }
    isEligible = { session.user != null } // must be logged in
}
```

上述情况，当 URL 实际被匹配，块中的代码将被解析，因此可以被用于结合所有种类的逻辑处理.

6.4.3 映射到视图

如果你想决定一个 URL 一个 view,而无需涉及一个控制器或者操作，你也可以这样做。 例如，如果你想映射根 URL / 到一个位于 grails-app/views/index.gsp 的 GSP，你可以这样使用:

```
static mappings = {
     "/"(view:"/index")  // map the root URL
}
```

换句话说，假如你需要一个具体给定的控制器(Controller)中的一个视图，你可以这样使用:

```
static mappings = {
   "/help"(controller:"site",view:"help") // to a view for a controller
}
```

6.4.4 映射到响应代码

Grails 同样允许你映射一个 HTTP 响应代码到控制器，操作或视图。所有你需要做的是使用一个方法名来匹配你所感兴趣的响应代码:

```
static mappings = {
   "500"(controller:"errors", action:"serverError")
   "404"(controller:"errors", action:"notFound")
   "403"(controller:"errors", action:"forbidden")
}
```

或者换句话说，假如你只不过想提供定制的错误页面:

```
static mappings = {
   "500"(view:"/errors/serverError")
   "404"(view:"/errors/notFound")
   "403"(view:"/errors/forbidden")
}
```

6.4.5 映射到 HTTP 方法

URL 映射同样可以配置成基于 HTTP 方法 (GET, POST, PUT or DELETE)的 map。这个对于 RESTful APIs 和基于 HTTP 方法的约束映射是非常有用的.

作为一个示例，下面的映射为 ProductControllerURL 提供一个 RESTful API URL 映射:

```
static mappings = {
  "/product/$id"(controller:"product"){
     action = [GET:"show", PUT:"update", DELETE:"delete", POST:"save"]
  }
}
```

6.4.6 映射通配符

Grails 的 URL 映射机制同样支持通配符映射。例如，考虑下面的映射:

```
static mappings = {
      "/images/*.jpg"(controller:"image")
}
```

这个映射将匹配所有 images 路径下像/image/logo.jpg 这样的 jpg。当然你可以通过一个变量来达到同样的效果:

```
static mappings = {
      "/images/$name.jpg"(controller:"image")
}
```

然而，你可以使用双通配符来匹配多于一个层次之外的:

```
static mappings = {
      "/images/**.jpg"(controller:"image")
}
```

这样的话，这个映射将不但匹配/image/logo.jpg 而且匹配/image/other/logo.jpg。更好的是你可以使用一个双通配符变量:

```
static mappings = {
      // will match /image/logo.jpg and /image/other/logo.jpg
      "/images/$name**.jpg"(controller:"image")
}
```

这样的话，它将储存路径，从 params 对象获得命名参数里的 name 通配符：

```
def name = params.name
println name // prints "logo" or "other/logo"
```

如果你使用通配符 URL mappings,那么你可以排除某些来自 Grails 的 URL mapping 进程 URIs. 实现这个你可以在 UrlMappings.groovy 类中设置 excludes：

```
class UrlMappings = {
	static excludes = ["/images/**", "/css/**"]
	static mappings = {
		…
	}
}
```

这样,Grails 不为匹配任何以 /images 或 /css 开头的 URLs.

6.4.7 自动重写链接

URL 映射另一个重要的特性是自动定制 link 标签的行为。以便改变这个映射而不需要改变所有的连接.

通过一个 URL 重写技术做到这点，从 URL 映射反转连接设计:

```
static mappings = {
  "/$blog/$year?/$month?/$day?/$id?"(controller:"blog", action:"show")
}
```

如果，你像下列一样使用连接标签:

```
<g:link controller="blog" action="show" params="[blog:'fred', year:2007]">My Blog</g:link>
<g:link controller="blog" action="show" params="[blog:'fred', year:2007, month:10]">My Blog - October 2007 Posts</g:link>
```

Grails 将自动重写 URL 通过适当的格式:

```
<a href="/fred/2007">My Blog</a>
<a href="/fred/2007/10">My Blog - October 2007 Posts</a>
```

6.4.8 应用约束

URL 映射同样支持 Grails 统一 验证规约 机制， 它允许你更进一步"约束"一个 URL 是怎么

被匹配的。例如，如果我们回到早前的 blog 示例代码，这个映射当前看上去会像这样：

```
static mappings = {
   "/$blog/$year?/$month?/$day?/$id?"(controller:"blog", action:"show")
}
```

允许 URLs 像这样:

```
/graemerocher/2007/01/10/my_funky_blog_entry
```

不过，它也允许这样:

```
/graemerocher/not_a_year/not_a_month/not_a_day/my_funky_blog_entry
```

当它强迫你在控制器代码中做一些聪明的语法分析时会有问题。幸运的是，URL 映射能进一步的约束验证 URL 标记:

```
"/$blog/$year?/$month?/$day?/$id?" {
    controller = "blog"
    action = "show"
    constraints {
        year(matches:/d{4}/)
        month(matches:/d{2}/)
        day(matches:/d{2}/)
    }
}
```

在这种情况下，约束能确保 year, month 和 day 参数匹配一个具体有效的模式，从而在稍后来减轻你的负担 .

6.5 Web 流(Flow)

概述

Grails 基于 Spring Web Flow 项目来支持创建 Web 流(Flow)。一个 Web 流(Flow)就是一个会话，它跨越多个请求并保持着流(Flow)作用域的状态。 一个 Web 流(Flow)也定义了开始和结束状态。 .

Web 流(Flow)无需 HTTP session,但作为替代，它将状态存储在序列化表单中，然后通过 Grails 来回传递的 request 参数中的执行流中的 key 进行还原。这相比其他使用 HttpSession 来保存状态的应用来说更具有可扩展性，尤其是在内存和集群方面.

Web 流(Flow)本质是高级的状态机，它管理着一个状态到下个状态"流"的执行。因为为你管理着状态，你就勿需担心用户在进入多步骤流(Flow)的操作(action) ，因为 Web 流(Flow)已经帮你管理了，因此 Web 流(Flow)在处理象网上购物、宾馆预定及任何多页面的工作流的应用具有出乎意料的简单.

创建流

创建一个流(Flow)只需简单的创建一个普通的 Grails 控制器(controller)，然后添加一个以规约 Flow 结尾的操作。例如:

```
class BookController {
  def index = {
    redirect(action:"shoppingCart")
  }
  def shoppingCartFlow = {
      …
  }
}
```

注意，当重定向或引用流(Flow)时，可以把它当做一个操作(action)而省略掉流(Flow)前缀。换句话说，上面流的操作(action)名为 shoppingCart.

6.5.1 开始与结束状态

如上所述，一个流(Flow)定义了开始和结束状态。一个开始状态是当用户第一次开始一个会话（或流(Flow)）。Grails 的开始流(Flow)是第一个带有代码块的方法调用。例如:

```
class BookController {
  …
  def shoppingCartFlow = {
    showCart {
      on("checkout").to "enterPersonalDetails"
      on("continueShopping").to "displayCatalogue"
    }
    …
    displayCatalogue {
      redirect(controller:"catalogue", action:"show")
    }
    displayInvoice()
  }
```

```
}
```

这里，showCart 节点是这个流的开始状态。因为这个 showCart 状态并没有定义一个操作(action)或重定向,只被视为是一个视图状态。 通过规约，指向 grails-app/views/book/shoppingCart/showCart.gsp 视图 .

注意，这不像正规的控制器(controller)操作(action)，这个视图被存储于与其流名字匹配的 grails-app/views/book/shoppingCart 目录中 .

shoppingCart 流(Flow)也可能拥有两个结束状态。第一个是 displayCatalogue，执行外部重定向到另一个控制器(controller)和操作(action)，从而结束流(Flow)。第二个是 displayInvoice 是一个最终状态，因为它根本没有任何事件，只是简单的渲染一个名为 grails-app/views/book/shoppingCart/displayInvoice.gsp 的视图，并在同一时间终止流(Flow).

一旦一个流(Flow)结束，它只能从开始状态重新开始，对于 showCart 不会来自任何其他状态.

6.5.2 操作(Action)状态和视图状态

视图状态

视图状态没有定义操作(action)或 redirect。下面是一个视图状态示例:

```
enterPersonalDetails {
  on("submit").to "enterShipping"
  on("return").to "showCart"
}
```

它默认查找一个名为 grails-app/views/book/shoppingCart/enterPersonalDetails.gsp 的视图。注意，enterPersonalDetails 定义了两个事件：submit 和 return。视图负责触发(triggering)这些事件。假如你想让视图用于渲染，使用 render 方法来完成:

```
enterPersonalDetails {
  render(view:"enterDetailsView")
  on("submit").to "enterShipping"
  on("return").to "showCart"
}
```

现在，它将查找 grails-app/views/book/shoppingCart/enterDetailsView.gsp。假如使

用共享视图，视图参数以/ 开头:

```
enterPersonalDetails {
  render(view:"/shared/enterDetailsView")
  on("submit").to "enterShipping"
  on("return").to "showCart"
}
```

现在，它将查找 grails-app/views/shared/enterDetailsView.gsp

操作(Action)状态

操作(Action)状态只执行代码但不渲染任何视图。操作(Action)的结果被用于控制流(Flow)的切换。为了创建一个操作操作(Action)状态，你需要定义一个被用于执行的操作。 这通过调用 action 方法实现并传递它的一个代码块来执行:

```
listBooks {
  action {
        [ bookList:Book.list() ]
  }
  on("success").to "showCatalogue"
  on(Exception).to "handleError"
}
```

正如你看到的，一个操作看上去非常类似于一个控制器(controller)操作(action)，实际上，假如你需要可以重用控制器(controller)操作(action)。假如这个操作没有错误成功返回，success 事件将被触发。在这里，返回一个 map,它被视为"model"看待，并自动放置于流(flow)作用域.

此外，在上面的示例中也使用了下面的异常处理程序来处理错误:

```
on(Exception).to "handleError"
```

这使当流(Flow)切换到状态出现异常的情况下调用 handleError.

你可以编写与流(flow)请求上下文相互作用更复杂的操作(action):

```
processPurchaseOrder  {
    action {
        def a =  flow.address
        def p = flow.person
        def pd = flow.paymentDetails
        def cartItems = flow.cartItems
```

```
        flow.clear()
        def o = new Order(person:p, shippingAddress:a, paymentDetails:pd)
        o.invoiceNumber = new Random().nextInt(9999999)
        cartItems.each { o.addToItems(it) }
        o.save()
        [order:o]
    }
    on("error").to "confirmPurchase"
    on(Exception).to "confirmPurchase"
    on("success").to "displayInvoice"
}
```

这是一个更复杂的操作(action),用于收集所有来自流(flow)作用域信息，并创建一个 Order 对象。然后，把 Order 作为模型返回。这里值得注意的重要事情是与请求上下文和 "流(flow)作用域"的相互作用.

切换操作

另一种形式的操作(action)被称之为*切换*操作(action)。一旦一个 event 被触发，切换操作优先于状态切换被直接执行。普通的切换操作如下：

```
enterPersonalDetails {
  on("submit") {
     log.trace "Going to enter shipping"
  }.to "enterShipping"
  on("return").to "showCart"
}
```

注意，我们是怎样传递一个代码块给 submit 事件，它只是简单的记录这个切换。切换状态对于数据绑定与验证是非常有用的，将在后面部分涵盖.

6.5.3 流(Flow)执行事件

为了执行流流从一个状态到下一个状态的 *切换* ，你需要一些方法来触发一个 *event* ，指出流流下一步该做什么。事件的触发可以来自于任何视图状态和操作状态.

来自于一个视图状态的触发事件

正如之前所讨论的，在早前代码列表内流的开始状态可能处理两个事件。一个 checkout 和一个 continueShopping 事件:

```
def shoppingCartFlow = {
    showCart {
        on("checkout").to "enterPersonalDetails"
        on("continueShopping").to "displayCatalogue"
    }
    ...
}
```

因为 showCart 事件是一个视图状态,它会渲染 grails-app/book/shoppingCart/showCart.gsp 视图. 在视图内部,你需要拥有一个用于触发流(Flow)执行的组件.在一个表单中，这可使用 submitButton 标签:

```
<g:form action="shoppingCart">
    <g:submitButton name="continueShopping" value="Continue Shopping"></g:submitButton>
    <g:submitButton name="checkout" value="Checkout"></g:submitButton>
</g:form>
```

这个表格必须提交返回 shoppingCart 流流。每个 submitButton 标签的 name 属性标示哪个事件将被触发。假如，你没有表格，你同样可以用 link 标签来触发一个事件，如下:

```
<g:link action="shoppingCart" event="checkout" />
```

来自于一个操作(Action)的触发事件

为了触发来自于一个操作(action)的一个事件，你需要调用一个方法。例如，这里内置的 error()和 success()方法。下面的示例在切换操作中验证失败后触发 error()事件:

```
enterPersonalDetails {
  on("submit") {
       def p = new Person(params)
       flow.person = p
       if(!p.validate())return error()
  }.to "enterShipping"
  on("return").to "showCart"
}
```

在这种情况下，因为错误，切换操作将使流回到 enterPersonalDetails 状态.

有了一种操作状态，你也能触发事件来重定向流:

```
shippingNeeded {
  action {
```

```
        if(params.shippingRequired) yes()
        else no()
    }
    on("yes").to "enterShipping"
    on("no").to "enterPayment"
}
```

6.5.4 流(Flow)的作用域

作用域基础

在以前的示例中，你可能会注意到我们在“流作用域（flow scope）” 中已经使用了一个特殊的流(flow)来存储对象，在 Grails 中共有 5 种不同的作用域可供你使用：

- request - 仅在当前的请求中存储对象
- flash - 仅在当前和下一请求中存储对象
- flow - 在工作流中存储对象，当流到达结束状态，移出这些对象
- conversation - 在会谈（conversation）中存储对象，包括根工作流和其下的子工作流
- session - 在用户会话（session）中存储对象

Grails 的 service 类可以自动的定位 web flow 的作用域，详细请参考 Services .

此外从一个 action 中返回的模型映射（model map）将会自动设置成 flow 范围，比如在一个转换（transition）的操作中，你可以象下面这样使用流(flow)作用域：

```
enterPersonalDetails {
    on("submit") {
        [person:new Person(params)]
    }.to "enterShipping"
    on("return").to "showCart"
}
```

要知道每一个状态总是创建一个新的请求，因此保存在 request 作用域中的对象在其随后的视图状态中不再有效，要想在状态之间传递对象，需要使用除了 request 之外的其他作用域。此外还有注意，Web 流(Flow)将：

1. 在状态转换的时候，会将对象从 flash 作用域移动到 request 作用域;
2. 在渲染以前，将会合并 flow 和 conversation 作用域的对象到视图模型中（因此你不

需要在视图中引用这些对象的时候，再包含一个作用域前缀了）.

流(Flow)的作用域和序列化

当你将对象放到 flash, flow 或 conversation 作用域中的时候，要确保对象已经实现了 java.io.Serializable 接口，否则将会报错。 这在 domain 类尤为显著，因为领域类通常在视图中渲染的时候被放到相应的作用域中，比如下面的领域类示例：

```
class Book {
    String title
}
```

为了能够让 Book 类的实例可以放到流(flow)作用域中，你需要修改如下:

```
class Book implements Serializable {
    String title
}
```

这也会影响到领域类中的关联和闭包，看下面示例:

```
class Book implements Serializable {
    String title
    Author author
}
```

此处如果 Author 关联没有实现 Serializable，你同样也会得到一个错误。此外在 GORM events 中使用的闭包比如 onLoad, onSave 等也会受到影响，下例的领域类如果放到 flow 作用域中，将会产生一个错误:

```
class Book implements Serializable {
    String title
    def onLoad = {
        println "I'm loading"
    }
}
```

这是因为 onLoad 事件中的代码块必能被序列化，要想避免这种错误，需要将所有的事件声明为 transient :

```
class Book implements Serializable {
    String title
    transient onLoad = {
```

```
            println "I'm loading"
        }
}
```

6.5.5 数据绑定和验证

在 开始和结束状态 部分， 开始状态的第一个示例触发一个切换到 enterPersonalDetails 状态。这个状态渲染一个视图，并等待用户键入请求信息：

```
enterPersonalDetails {
  on("submit").to "enterShipping"
  on("return").to "showCart"
}
```

一个视图包含一个带有两个提交按钮的表格，每个都触发提交事件或返回事件:

```
<g:form action="shoppingCart">
    <!-- Other fields -->
    <g:submitButton name="submit" value="Continue"></g:submitButton>
    <g:submitButton name="return" value="Back"></g:submitButton>
</g:form>
```

然而，怎么样捕捉被表格提交的信息？为了捕捉表格信息我们可以使用流切换操作:

```
enterPersonalDetails {
  on("submit") {
       flow.person = new Person(params)
       !flow.person.validate() ? error() : success()
  }.to "enterShipping"
  on("return").to "showCart"
}
```

注意，我们是怎样执行来自请求参数的绑定，把 Person 实体放置于流(flow)作用域中。同样有趣的是，我们执行 验证，并在验证失败是调用 error()方法 .这个流(flow)的动机即停止切换并返回 enterPersonalDetails 视图,因此,有效的项通过 user 进入,否则,切换继续并转到 enterShipping state.

就像正规操作(action),流(flow)操作(action)也支持 命令对象概念,通过定义闭包的第一个参数：

```
enterPersonalDetails {
```

```
    on("submit") { PersonDetailsCommand cmd ->
        flow.personDetails = cmd
        !flow.personDetails.validate() ? error() : success()
    }.to "enterShipping"
    on("return").to "showCart"
}
```

6.5.6 子流程和会话

Grails 的 Web Flow 集成同样支持子流（subflows）。一个子流在一个流中就像一个流。拿下面 search 流作为示例:

```
def searchFlow = {
        displaySearchForm {
            on("submit").to "executeSearch"
        }
        executeSearch {
            action {
                [results:searchService.executeSearch(params.q)]
            }
            on("success").to "displayResults"
            on("error").to "displaySearchForm"
        }
        displayResults {
            on("searchDeeper").to "extendedSearch"
            on("searchAgain").to "displaySearchForm"
        }
        extendedSearch {
            subflow(extendedSearchFlow)    // <--- extended search subflow
            on("moreResults").to "displayMoreResults"
            on("noResults").to "displayNoMoreResults"
        }
        displayMoreResults()
        displayNoMoreResults()
}
```

它在 extendedSearch 状态中引用了一个子流。子流完全是另一个流：

```
def extendedSearchFlow = {
    startExtendedSearch {
        on("findMore").to "searchMore"
```

```
            on("searchAgain").to "noResults"
        }
        searchMore {
            action {
                def results = searchService.deepSearch(ctx.conversation.query)
                if(!results)return error()
                conversation.extendedResults = results
            }
            on("success").to "moreResults"
            on("error").to "noResults"
        }
        moreResults()
        noResults()
}
```

注意，它是怎样把 extendedResults 放置于会话范围的。这个范围不同于流范围，因为它允许你横跨整个会话而不只是这个流。同样注意结束状态（每个子流的 moreResults 或 noResults 在主流中触发事件：

```
extendedSearch {
        subflow(extendedSearchFlow)   // <--- extended search subflow
        on("moreResults").to "displayMoreResults"
        on("noResults").to "displayNoMoreResults"
}
```

6.6 过滤器

尽管 Grails 支持良好的细粒度控制器(controller)，但只对少数控制器(controller)的应用时非常有用，当管理大型应用时就会变得很困难。另一方面，过滤器能横跨一群控制器(controller),一个 URI 空间或一个具体的操作(action)。过滤器对插件更容易并能保证彻底的分离主要控制器(controller)逻辑，有利于所有像安全，日志等等这样的横切关注点 .

6.6.1 应用过滤器

为了创建一个过滤器,可在 grails-app/conf 下创建一个以规约 Filters 结尾的类。在这个类中，定义一个名为 filters 的代码块，它包含了过滤器的定义：

```
class ExampleFilters {
  def filters = {
```

```
        // your filters here
    }
}
```

每个在 filters 块中定义的过滤器(Filters)拥有一个名字和一个作用域。名字是方法的名字，作用域使用命名参数来定义。例如，假如你需要定义一个应用于所有控制器(controller)和操作(action)的过滤器(Filters)可以使用通配符：

```
sampleFilter(controller:'*', action:'*') {
  // interceptor definitions
}
```

过滤器的作用域可以是下面之一:

- 具有通配符的一个控制器(controller)和/或操作(action)名字对
- 具有 Ant 路径匹配语法的一个 URI

过滤器的一些示例包括:

- 所有控制器(controller)和操作(action)

```
all(controller:'*', action:'*') {
}
```

- 只适合 BookController

```
justBook(controller:'book', action:'*') {
}
```

- 适合一个 URI 空间

```
someURIs(uri:'/book/**') {
}
```

- 适合所有的 URIs

```
allURIs(uri:'/**') {
}
```

另外，这个次序决定了你所定义的过滤器的执行次序.

6.6.2 过滤器(Filters)类型

在过滤器的主体内，你可以定义下列过滤器(Filters)的拦截器类型之一:

- before - 操作（Action）之前执行. 返回 false 来指示后续的控制器(controller)和操作(action)不会被执行
- after - 操作（Action）之后执行. 获取第一参数作为视图模型
- afterView - 视图渲染之后执行

例如，为实现普通身份验证，可以定义如下过滤器(Filters):

```
class SecurityFilters {
  def filters = {
    loginCheck(controller:'*', action:'*') {
      before = {
        if(!session.user && !actionName.equals('login')) {
          redirect(action:'login')
          return false
        }
      }
    }
  }
}
```

这里的 loginCheck 过滤器(Filters)使用一个 before 拦截器来执行代码块，检查是否一个用户在 session 内，假如不是，重定向到 login 操作(action)。注意，如何返回 false 确保操作(action)本身不被执行 .

6.6.3 变量与作用域

过滤器支持所有在 控制器(controllers) 和 标签库 中可用的属性，附加 application context :

- request - HttpServletRequest 对象
- response - HttpServletResponse 对象
- session - HttpSession 对象
- servletContext - ServletContext 对象
- flash - flash 对象
- params - 请求参数对象
- actionName - 被分配的 action 名
- controllerName - 被分配的 controller 名
- grailsApplication - 当前运行的 Grails 应用程序

- applicationContext - ApplicationContext 对象

不过，过滤器只支持用于控制器(controller)和标签库方法的子集 。这些包括:

- redirect - 重定向到其他的控制器(controller)和操作(action)
- render - 渲染自定义响应

6.7 Ajax

Ajax 代表异步 Javascript 与 XML,它是转向富 web 应用程序的驱动力. 这些类型的应用程序，通常更适合于像 Ruby 和 Groovy 语言所写的敏捷，动态框架,Grails 通过它的 Ajax 标签库提供支持构建 Ajax 应用程序. 它们完整的列表可以参看标签库参考.

6.7.1 用 Prototype 实现 Ajax

Grails 默认装载 Prototype 库,但通过 Plug-in 系统,可以提供对 Dojo, Yahoo UI 和 Google Web Toolkit 等其他框架的支持.

这部分涵盖 Grails 对 Prototype 的支持。你需要在页面的<head>标签内添加这样一行就可以开始了：

```
<g:javascript library="prototype" />
```

这里使用 javascript 标签自动插入 Prototype 正确位置的引用。假如你同样需要 Scriptaculous ，你可以如下这样做为替换：

```
<g:javascript library="scriptaculous" />
```

6.7.1.1 远程链接

远程内容可以通过多种方式加载，最常使用的方法是通过 remoteLink 标签。 这个标签允许创建的 HTML 锚标记执行一个异步请求，并在一个元素中随意设置响应。用这个简单方式创建的远程链接就像这样：

```
<g:remoteLink action="delete" id="1">Delete Book</g:remoteLink>
```

上面的连接发送一个异步请求给当前 id 为 1 的控制器的 delete 操作 .

6.7.1.2 内容更新

这真是太棒了，但通常你想提供一些事情发生的反馈信息给用户:

```
def delete = {
    def b = Book.get( params.id )
    b.delete()
    render "Book ${b.id} was deleted"
}
```

GSP 代码:

```
<div id="message"></div>
<g:remoteLink action="delete" id="1" update="message">Delete Book</g:remoteLink>
```

上面的示例将调用这个操作并设置 message div 的响应内容为"Book 1 was deleted"。这通过标签上的 update 属性来完成，它同样可以获取一个 map 来指出在失败时什么被更新：

```
<div id="message"></div>
<div id="error"></div>
<g:remoteLink action="delete" id="1"
        update="[success:'message',failure:'error']">Delete Book</g:remoteLink>
```

这里，error div 在请求失败时被更新.

6.7.1.3 远程表单提交

,一个 HTML form 也可以异步被提交通过以下两种方式之一。第一个，使用 formRemote 标签，它和 remoteLink 标签有类似的属性：

```
<g:formRemote url="[controller:'book',action:'delete']" update="[success:'message',failure:'error']">
    <input type="hidden" name="id" value="1" />
    <input type="submit" value="Delete Book!" />
</g:formRemote >
```

或者作为选择可以使用 submitToRemote 来创建一个提交按钮。它允许一些按钮远程提交而一些不依赖操作：

```
<form action="delete">
    <input type="hidden" name="id" value="1" />
    <g:submitToRemote action="delete" update="[success:'message',failure:'error']" />
</form>
```

6.7.1.4 Ajax 事件

某些事件的发生会调用特定的 javascript。所有以"on"开头的事件，在适当的时候允许你反馈信息给用户,或采取其他行为:

```
<g:remoteLink action="show"
        id="1"
        update="success"
        onLoading="showProgress()"
        onComplete="hideProgress()">Show Book 1</g:remoteLink>
```

上述代码将执行"showProgress()"函数来显示一个进度条或者其他适当的展示，其他的事件还包括：

- onSuccess - 成功时调用的 javascript 函数
- onFailure - 失败时调用的 javascript 函数
- on_ERROR_CODE - 处理指定的错误代码时调用的 javascript 函数 (例如 on404="alert('not found!')")
- onUninitialized - 一个 ajax 引擎初始化失败时调用的 javascript 函数
- onLoading - 当远程函数加载响应时调用的 javascript 函数
- onLoaded - 当远程函数加载完响应时调用的 javascript 函数
- onComplete - 当远程函数完成（包括任何更新）时调用的 javascript 函数

假如你需要引用 XmlHttpRequest 对象，你可以使用隐式的 event 参数 e 获取它：

```
<g:javascript>
  function fireMe(e) {
        alert("XmlHttpRequest = " + e)
  }
}
</g:javascript>
<g:remoteLink action="example"
        update="success"
        onSuccess="fireMe(e)">Ajax Link</g:remoteLink>
```

6.7.2 用 Dojo 实现 Ajax

Grails 把 Dojo 作为一种外部插件来支持 Grails 的特性。在终端窗口，进入你项目的根目录键入下列命令来安装插件：

```
grails install-plugin dojo
```

将下载 Dojo 最新的支持版本，并安装到你的 Grails 项目中。完成上面的步骤后，你可以在你页面的顶部添加下列引用:

```
<g:javascript library="dojo" />
```

现在，所有像 remoteLink, formRemote 和 submitToRemote 标签都可以和 Dojo 进行远程处理工作 .

6.7.3 用 GWT 实现 Ajax

Grails 同样支持 Google Web Toolkit 特性，插件的全面 文档 可以在 Grails wiki 中找到 .

6.7.4 服务端的 Ajax

虽然 Ajax 特性 X 为 XML，但通常可以分解成许多不同方式执行 Ajax:

- 内容为中心的 Ajax - 只不过是使用远程调用的 HTML 结果来更新页面
- 数据为中心的 Ajax - 实际上是发送一个来自于服务器端的 XML 或 JSON，通过编程更新页面
- 脚本为中心的 Ajax - 服务器端发送的 Javascript 流在运行中被赋值

在 Ajax 部分中的更多的示例涵盖了内容为中心的 Ajax 在什么地方更新页面，但同样你可能使用数据为中心的 Ajax 或脚本为中心的 Ajax。这份指南涵盖了不同风格的 Ajax .

内容为中心的 Ajax

作为概括，内容为中心的 Ajax 涉及从服务器端发送一些 HTML 返回和通过使用 render 方法来渲染模板：

```
def showBook = {
    def b = Book.get(params.id)
    render(template:"bookTemplate", model:[book:b])
}
```

在客户端调用这个会涉及到 remoteLink 标签的使用 :

```
<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>
```

数据为中心的 Ajax 与 JSON

数据为中心的 Ajax 通常涉及到客户端响应的赋值和编程化更新。Grails 中的 JSON 响应，通常使用 Grails 的 JSON marshaling 能力：

```
import grails.converters.*
def showBook = {
    def b = Book.get(params.id)



  render b as JSON
}
```

然后，在客户端使用一个 Ajax 事件处理解析这个进入的 JSON 请求:

```
<g:javascript>
function updateBook(e) {
    var book = eval("("+e.responseText+")") // evaluate the JSON
    $("book"+book.id+"_title").innerHTML = book.title
}
<g:javascript>
<g:remoteLink action="test" update="foo" onSuccess="updateBook(e)">Update
Book</g:remoteLink>
<g:set var="bookId">book${book.id}</g:set>
<div id="${bookId}">
    <div id="${bookId}_title">The Stand</div>
</div>
```

数据为中心的 Ajax 与 XML

在服务器端使用 XML 同样普遍:

```
import grails.converters.*
def showBook = {
    def b = Book.get(params.id)
```

```
    render b as XML
}
```

不过，因为涉及到 DOM，客户变得更复杂:

```
<g:javascript>
function updateBook(e) {
        var xml = e.responseXML
        var id = xml.getElementsByTagName("book").getAttribute("id")
        $("book"+id+"_title")=xml.getElementsByTagName("title")[0].textContent
}
<g:javascript>
<g:remoteLink action="test" update="foo" onSuccess="updateBook(e)">Update
Book</g:remoteLink>
<g:set var="bookId">book${book.id}</g:set>
<div id="${bookId}">
        <div id="${bookId}_title">The Stand</div>
</div>
```

脚本为中心的 Ajax 与 JavaScript

脚本为中心的 Ajax 涉及实际返回的 Javascript 在客户端被赋值。这样的示例见下表:

```
def showBook = {
        def b = Book.get(params.id)
    response.contentType = "text/javascript"
    String title = b.title.encodeAsJavascript()
    render "$('book${b.id}_title')='${title}'"
}
```

要记住的重要事情是，设置 contentType 为 text/javascript。如果在客户端使用 Prototype，由于设置了 contentType，返回的 Javascript 将自动被赋值.

很明显，在这种情况下，它是关键性的，你有一个一致的 client-sideAPI，因此，你不想客户端的改变破坏服务器端。这就是 Rails 有些像 RJS 的理由之一。虽然，Grails 当前没有像 RJS 的一个特性，但动态 Dynamic JavaScript Plug-in 插件提供了类似的能力.

6.8 内容协商

Grails 已经内置支持内容协商通过使用任意 HTTP Accept 报头，一种明确格式请求参数或 URI 映射的扩展.

配置 Mime 类型

在你开始处理内容协商之前，你必须告诉 Grails 希望支持什么样的内容类型。 默认情况下，grails-app/conf/Config.groovy 内使用 grails.mime.types 设置来配置若干不同的内容类型：

```
grails.mime.types = [ xml: ['text/xml', 'application/xml'],
                      text: 'text-plain',
                      js: 'text/javascript',
                      rss: 'application/rss+xml',
                      atom: 'application/atom+xml',
                      css: 'text/css',
                      cvs: 'text/csv',
                      all: '*/*',
                      json: 'text/json',
                      html: ['text/html','application/xhtml+xml']
                    ]
```

上面的小块配置，允许 Grails 检查把包含 'text/xml' 或 'application/xml' 媒体类型的一个请求的格式只当做 'xml'看待，你可以添加你自己的类型通过简单的添加条目到 map 中.

内容协商使用 Accept 报头

每个进入的 HTTP 请求都有个指定的 Accept 报头，它定义了什么样的媒体类型(或 mime 类型)客户端能"接受"。在老式浏览器中通常是：

```
*/*
```

这意味着任何事物.不过在新生浏览器中,所有东西一起像这样发送更有用(一个 FirefoxAccept 报头示例)：

```
text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5
```

Grails 解析这个进入的格式，并添加一个 property 给 request 对象，用于描叙首选的请求格式。对于上述示例下列的断言会通过：

```
assert 'html' == request.format
```

为什么？这个 text/html 媒体类型拥有最高"质量"等级 0.9，因此，具有最高优先权。如前所述，假如你有一老式浏览器结果会稍微不同：

```
assert 'all' == request.format
```

在这种情况下，'all'可能的格式会被客户端接受。为了处理来自控制器(Controllers)不同类型的请求，你可以使用 withFormat 方法，它的行为被当作 switch 表达式：

```
import grails.converters.*
class BookController {
    def books
    def list = {
        this.books = Book.list()
        withFormat {
            html bookList:books
            js { render "alert('hello')" }
            xml { render books as XML }
        }
    }
}
```

当 Grails 只执行 html()调用并且首选的格式是 html 时会发生什么。它只是让 Grails 寻找每个名为 grails-app/views/books/list.html.gsp 或 grails-app/views/books/list.gsp 的视图。如果格式是 xml，那么，闭包会被调用，XML 响应会被渲染 .

我们怎样处理'all'格式?只需在 withFormat 代码块中简单指定 content-types，以便,无论你想要的哪个都会被首先执行。因此，在上面示例中的"all" 将触发 html 处理 .

> 当使用 withFormat 时确保它在控制器(controller)操作(action)中最后一个被调用，因为 withFormat 方法的返回值用来决定操作(action)下一步做什么.

内容协商与格式化请求参数

如果请求头的内容跟你的不一致，通过指定一个 format 的请求参数覆盖这个格式：

```
/book/list?format=xml
```

你同样可以在 URL Mappings 定义中定义这个参数：

```
"/book/list"(controller:"book", action:"list") {
    format = "xml"
}
```

内容协商与 URI 扩展

Grails 同样可以通过 URI 扩展支持内容协商。例如，给定下列 URI:

/book/list.xml

Grails 将剔除扩展并映射到/book/list 作为替代，同时，基于这个扩展把内容格式化为 xml。这个行为是默认允许的，那么，假如你希望关闭它， 你必须把 grails-app/conf/Config.groovy 下的 grails.mime.file.extensions 属性设置为 false：

```
grails.mime.file.extensions = false
```

测试内容协商

为了在一个综合测试中测试内容协商(参见 测试部分)你可以操作每个进入的请求包头：

```
void testJavascriptOutput() {
        def controller = new TestController()
        controller.request.addHeader "Accept", "text/javascript, text/html, application/xml, text/xml, */*"
        controller.testAction()
        assertEquals "alert('hello')", controller.response.contentAsString
}
```

或者你可以设置格式化参数来实现类似的效果:

```
void testJavascriptOutput() {
        def controller = new TestController()
        controller.params.format = 'js'
        controller.testAction()
        assertEquals "alert('hello')", controller.response.contentAsString
}
```

7. 验证

Grails 的验证功能基于 Spring's Validator API 和数据绑定功能。不过，Grails 利用这些特性，通过它的"constraints(约束)"机制， 提供了一个统一的定义验证约束方式。

Grails 中的 Constraints(约束)是用声明式指定效验规则的方式。常用于 domain 类，不过 URL Mappings 和 Command 对象同样支持 Constraints(约束)。

7.1 声明 Constraints(约束)

在一个 domain 类中，constraints(约束) 是通过给 constraints 属性赋值代码块的形式来定义的:

```
class User {
   String login
   String password
   String email
   Integer age
   static constraints = {
        ...
   }
}
```

然后，通过与属性名匹配的方法调用,并结合命名参数来指定 constraints(约束)

```
class User {
   ...
   static constraints = {
      login(size:5..15, blank:false, unique:true)
      password(size:5..15, blank:false)
      email(email:true, blank:false)
      age(min:18, nullable:false)
   }
}
```

在这个示例中，我们声明 login 属性必须在 5-15 个字符长度之间，不能为空，并且必须是唯一的。我们还可以为 password,email 和 age 属性运用其他的约束.

现有约束的完整参考可以在参考指南中找到

7.2 验证约束

验证基础

你可以在任何实体中调用 validate 方法验证 domain:

```
def user =  new User(params)
if(user.validate()) {
   // do something with user
}
else {
   user.errors.allErrors.each {
      println it
   }
```

```
}
```

domain 的 errors 属性是一个 Spring Errors 接口实例. Errors 提供用于导航验证错误以及取回原始值的方法。

验证阶段

Grails 中本质上有 2 个验证阶段，第一个阶段是 data binding ，当你把请求参数绑定到实体上发生，例如:

```
def user = new User(params)
```

这时，因为类型转换(如 String 转换为 Dates),在 errors 属性可能已经出现错误。 你可以检查它们并通过使用 ErrorsAPI 获得原始输入值:

```
if(user.hasErrors()) {
        if(user.errors.hasFieldErrors("login")) {
                println user.errors.getFieldError("login").rejectedValue
        }
}
```

验证的第 2 阶段发生在当你调用 validate 或 save 时。这时，Grails 将会验证你在 constraints 定义的约束值。比如，默认的持久方法 save 会在执行之前调用 validate 。因此，允许你像下面这样编码:

```
if(user.save()) {
    return user
}
else {
    user.errors.allErrors.each {
        println it
    }
}
```

7.3 客户端验证

显示错误

通常，当你得到一个验证错误后，你会重定向回页面渲染这些错误。这时，你就需要一些渲染错误的方法。 Grails 提供了一组丰富的标签，处理错误渲染。 如果只是想简单的渲染错

误列表，可以使用 renderErrors:

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

假如，你需要更多的控制，可以使用 hasErrors 和 eachError:

```
<g:hasErrors bean="${user}">
  <ul>
   <g:eachError var="err" bean="${user}">
      <li>${err}</li>
   </g:eachError>
  </ul>
</g:hasErrors>
```

高亮错误

当一个字段存在错误的输入时，一个红色块和一些提示符，对于高亮错误非常有用。这时通过把 hasErrors 当做方法调用来做到。 比如:

```
<div class='value ${hasErrors(bean:user,field:'login','errors')}'>
   <input type="text" name="login" value="${fieldValue(bean:user,field:'login')}"/>
</div>
```

上面的代码做了什么？它会检查 user 的 login 字段是否存在任何错误，如果存在，就给 div 添加一个 errors CSS class，这样就可以让你使用 CSS 来高亮 div.

取回输入值

任何错误实际上是 Spring 中 FieldError 类的实体，它会在内部保存原始输入值。通过 fieldValue 标签获取错误对象的原始输入值:

```
<input type="text" name="login" value="${fieldValue(bean:user,field:'login')}"/>
```

这段代码会查看，在 User bean 中是否存在一个 FieldError，如果是，就获取 login 字段的原始输入值。

7.4 验证与国际化

Grails 中另一个关于 errors 值得注意的重要事情是：错误消息的显示，无需任何的硬编码。Spring 中的 FieldError 类使用 Grails 的 i18n 支持，基本上解决了来自消息绑定的消息。

规约与 Message 编码

编码它们自己通过规约来规定，例如，考虑早前看到约束:

```
package com.mycompany.myapp
class User {
    ...


    static constraints = {
        login(size:5..15, blank:false, unique:true)
        password(size:5..15, blank:false)
        email(email:true, blank:false)
        age(min:18, nullable:false)
    }
}
```

如果 blank 约束不合法， Grails 将在 form 中通过规约查找消息编码:

```
[Class Name].[Property Name].[Constraint Code]
```

在这种情况下， blank 约束就会是 user.login.blank 因此，你需要在 grails-app/i18n/messages.properties 文件中包含下面这样的消息:

```
user.login.blank=Your login name must be specified!
```

它会查找带 package 或不带 package 的类名，带有 package 的将会优先。作为示例，com.mycompany.myapp.User.login.blank 将先于 user.login.blank 使用。 当你 domain 类的消息编码与插件产生冲突时，可以这样使用。

每个规约的编码参考可以参考参考指南 constraints refer to the reference guide for each constraint.

显示消息

当你使用 message 标签时， renderErrors 标签将自动处理查找消息。 不过，假如你想获得更多的渲染控制，你需要自己编写代码:

```
<g:hasErrors bean="${user}">
  <ul>
   <g:eachError var="err" bean="${user}">
      <li><g:message error="${err}" /></li>
   </g:eachError>
```

```
  </ul>
</g:hasErrors>
```

这个示例中， eachError 标签主体内，我们使用了 message 标签的 error 参数来读取给定的错误 。

7.5 验证非 Domain 与命令行对象

Domain 类与 command objects(命令行对象)默认支持验证。其他类也可以在类中定义静态 constraints 属性获得验证(如上所述)，然后把它们告诉框架。 当应用程序在框架中注册验证类 是非常重要的。简单定义 constraints 属性是不够的。

Validateable 注解

任何定义了静态 constraints 属性和标有 @Validateable 接口的类可以在框架中被验证。考虑下面示例:

```
// src/groovy/com/mycompany/myapp/User.groovy
package com.mycompany.myapp
import org.codehaus.groovy.grails.validation.Validateable


@Validateable
class User {
    ...


    static constraints = {
        login(size:5..15, blank:false, unique:true)
        password(size:5..15, blank:false)
        email(email:true, blank:false)
        age(min:18, nullable:false)
    }
}
```

默认情况下，框架会搜索所有带有@Validateable 注解的类。 你可以指定框架只搜索某个 packages，通过给 Config.groovy 中的 grails.validateable.packages 属性赋值一列字符串。

```
// grails-app/conf/Config.groovy
```

...

```
grails.validateable.packages = ['com.mycompany.dto', 'com.mycompany.util']
```

...

假如 grails.validateable.packages 属性被设置，框架只会在这些 packages 中搜索 (和它们的子 packages) 标有@Validateable 的类.

注册 Validateable 类

假如一个类没有被标为@Validateable，它仍然可能通过框架验证。 那就是必须在类中定义静态 constraints 属性 (如上所述) ，然后，通过在 Config.groovy 中为 grails.validateable.classes 属性设置值来告诉框架。

```
// grails-app/conf/Config.groovy
...

grails.validateable.classes = [com.mycompany.myapp.User, com.mycompany.dto.Account]

...
```

8. Service 层

除了 Web 层 之外， Grails 还定义了 service 层的概念。Grails 团队不赞成在 controllers 中嵌入核心的应用程序逻辑，因为这样并没有提升重用和清楚的关注点分离。

Grails 中的 Services 在应用程序中被视为放置多数逻辑的地方 。从 controllers 脱离，负责处理通过重定向的请求流等等。

创建 Service

你可以在终端窗口的项目根目录下运行 create-service 创建 Service:

```
grails create-service simple
```

上面的示例将在 grails-app/services/SimpleService.groovy 位置创建一个 Service。service 的名字按规约以 Service 结尾。除此之外，service 就是个普通的 Groovy 类:

```
class SimpleService {
}
```

8.1 声明式事务处理

Services 一般涉及协调 domain 类之间的逻辑，，因此常常涉及大范围的持久化操作。因为 services 性质，它们常常需要事物状态。你可以使用 withTransaction 方法来编程事物，不过，这是重复性的，没有充分利用 Spring 强大的潜在事物抽象

Services 允许启用事物，本质上是以声明的方式来声明 service 中的所有方法必须用于事物。默认情况下，所有 services 的事物都是可用的——禁用它，只需设置 transactional 属性为 false:

```
class CountryService {
    static transactional = false
}
```

你也可以默认设置这个属性为 true 在以后改变它，或者清楚的表明这个服务是有意地用于事物。

> 警告: 依赖注入 是 **唯一** 声明事物工作的方式。你不能使用 new 操作符，像这样 new BookService()获取事物服务

其结果是,所有的方法都被包含在事物中，当方法体中抛出异常时，自动回滚。事物的传播级别被默认设置为 PROPAGATION_REQUIRED.

8.2 服务作用域

默认情况下，存取服务方法是非同步的，所以无法阻止同步执行这些函数。事实上，因为服务是单例的，可以被同时使用，你必须非常小心服务中存储状态。或者采用容易（和更好的）途径并不在 y service 中存储状态。

你可以通过把 service 放置于特定的作用域来改变这样的行为:

- prototype -一个新的 service 每次被注入到其他类时创建
- request - 一个新的 service 在每次请求时创建

- flash - 一个新的 service 只在当前或下个请求时创建
- flow - 在 web flows 中， service 将存在于 flow 的作用域
- conversation - 在 web flows 中， service 将存在于会话的作用域。根 flow 和它的子 flows
- session - 一个 service 被创建用于 session 的作用域
- singleton (默认) - 只有一个实例的 service，任何时候都存在

  假如你的 service 为 flash, flow 或 conversation 作用域，它需要实现 java.io.Serializable 并只用于 Web Flow 上下文

为了启用一个作用域，在你的类中添加一个静态 scope 属性，其值为上面所述的作用域之一:

```
static scope = "flow"
```

8.3 依赖注入与服务

依赖注入基础

Grails 服务的一个重要方面是，有能力利用 Spring 框架的依赖注入能力。 Grails 支持 "依赖注入通过规约". 换句话说，你可以使用一个属性名表示的一个服务的类名，自动把他们注入到 controllers, tag libraries,等等。

作为示例，给定的服务名为 BookService, 如果你像下面这样在 controller 中放置一个名为 bookService 的属性:

```
class BookController {
  def bookService
  ...
}
```

在这种情况下，Spring 容器将自动注入一个基于它自己配置作用域的服务实体。所有的依赖注入是通过名字的; Grails 不支持类型注入。你也可以像下面这样指定类型:

```
class AuthorService {
      BookService bookService
}
```

不过, 存在副作用，即在开发模式下 BookService 的改变会在加载时抛出一个错误。

依赖注入与服务

你可以使用相同的技术在一个服务中注入另一个服务。如果说，你的 AuthorService 需要一个 BookService, 可以像下面这样声明 AuthorService:

```
class AuthorService {
    def bookService
}
```

依赖注入与 Domain 类

你甚至可以在 domain 类中注入服务，这可以帮助开发出各种丰富的 domain:

```
class Book {
    ...
    def bookService
    def buyBook() {
        bookService.buyBook(this)
    }
}
```

8.4 Using Services from Java

服务的强大在于它包含了可重用的逻辑，你可以使用来自其他类的服务，包括 Java 类。这里有一些方法让你重用来自 Java 的服务。简单的方法是把你的服务移动到 grails-app/services 目录下的一个包里。这是关键步骤，因为你不可能在 Java 中导入一个默认 package 。作为示例， BookService 就是因为上面的原因，在下面 Java 中不能使用:

```
class BookService {
    void buyBook(Book book) {
        // logic
    }
}
```

不过, 把这个类放入一个 package 中便可修复，, 把这个类移动到 grails-app/services/bookstore 子目录，然后，修改 package 声明:

```
package bookstore
class BookService {
    void buyBook(Book book) {
        // logic
```

```
    }
}
```

package 的替代是，定义个需要服务实现的接口:

```
package bookstore;
interface BookStore {
    void buyBook(Book book);
}
```

然后，服务:

```
class BookService implements bookstore.BookStore {
    void buyBook(Book b) {
        // logic
    }
}
```

后一种方法更熟悉, 在 Java 端，只需要接口的引用，而不需要实现类。无论哪种方式，这个练习的目的是，在编译时，让 Java 能够静态解决类（或接口）的使用。现在，这样便可在 src/java 包内创建一个 Java 类，并提供了一个 setter，在 Spring 中使用 bean 的类型和它的名字:

```
package bookstore;
// note: this is Java class
public class BookConsumer {
    private BookStore store;
    public void setBookStore(BookStore storeInstance) {
        this.store = storeInstance;
    }
    ...
}
```

这样一来，你可以在 grails-app/conf/spring/resources.xml 中把这个 Java 当做 Spring bean 来配置 (更多详情查看 Grails and Spring):

```
<bean id="bookConsumer" class="bookstore.BookConsumer">
    <property name="bookStore" ref="bookService" />
</bean>
```

9. 测试

自动化测试被看成是 Grails 中一个重要部分，以 Groovy Tests 为基础执行测试。因此，Grails 提供了许多方法，使不管是简单的单元测试，还是高难度的方法测试都能更容易执行。这个章节详细描述了 Grails 给出的各种不同的测试方法。

你要明白的第一件事是，所有 create-*命令，实际上 Grails 最后都会自动帮它们创建集成好的全部测试实例。比如你运行下方的 create-controller 命令：

```
grails create-controller simple
```

Grails 不仅在 grails-app/controllers/目录下创建了 SimpleController.groovy，而且在 test/integration/目录下创建了对它的集成测试实例 SimpleControllerTests.groovy。，然而 Grails 不会在这个测试实例里自动生成逻辑代码，这部分需要你自己写。

当你完成这部分逻辑代码，就可以使用 test-app 执行所有测试实例：

```
grails test-app
```

上面的这个命令将输出如下内容：

```
-------------------------------------------------------
Running Unit Tests...
Running test FooTests...FAILURE
Unit Tests Completed in 464ms ...
-------------------------------------------------------
Tests failed: 0 errors, 1 failures
```

同时运行结果放在 test/reports 目录下。你也可以指定名字单独运行一个测试，不需要测试后缀参数：

```
grails test-app SimpleController
```

除此之外，你可以以空格隔开同时运行多个实例：

```
grails test-app SimpleController BookController
```

9.1 单元测试

单元测试是对单元块代码的测试。换句话说你在分别测试各个方法或代码段时，不需要考虑它们外层周围代码结构。在 Grails 框架中，你要特别注意单元测试和集成测试之间的一个不

同点，因为在单元测试中，Grails 在集成测试和测试运行时，不会注入任何被调用的动态方法。

这样做是有意义的，假如你考虑到，在 Grails 中各个数据库注入的各自方法（通过使用 GORM），和潜在使用的 Servlet 引擎（通过控制器）。例如，你在 BookController 调用如下的一个服务应用：

```
class MyService {
   def otherService
     String createSomething() {
            def stringId = otherService.newIdentifier()
            def item = new Item(code: stringId, name: "Bangle")
            item.save()
            return stringId
      }
   int countItems(String name) {
            def items = Item.findAllByName(name)
            return items.size()
      }
}
```

正如你看到的，这个应用调用了 GORM，那么你用在单元测试中怎样处理如上这段代码呢？答案在 Grails 测试支持类中可以找到。

测试框架

Grails 测试插件最核心部分是 grails.test.GrailsUnitTestCase 类。它是 GroovyTestCase 子类，为 Grails 应用和组件提供测试工具。这个类为模拟特殊类型提供了若干方法，并且提供了按 Groovy 的 MockFor 和 StubFor 方式模拟的支持。

正常来说你在看之前所示的 MyService 例子和它对另外一个应用服务的依赖，以及例子中使用到的动态域类方法会有一点痛苦。你可以在这个例子中使用元类编程和“map as object”规则，但是很快你会发现使用这些方法会变得很糟糕，那我们要怎么用 GrailsUnitTestCase 写它的测试呢？

```
import grails.test.GrailsUnitTestCase
class MyServiceTests extends GrailsUnitTestCase {
      void testCreateSomething() {
            // Mock the domain class.
            def testInstances = []
            mockDomain(Item, testInstances)
```

```
        // Mock the "other" service.
            String testId = "NH-12347686"
            def otherControl = mockFor(OtherService)
            otherControl.demand.newIdentifier(1..1) {-> return testId }
           //      Initialise the service and test the target method.
            def testService = new MyService()
            testService.otherService = otherControl.createMock()
          def retval = testService.createSomething()
            // Check that the method returns the identifier returned by the
            // mock "other" service and also that a new Item instance has
            // been saved.
            assertEquals testId, retval
            assertEquals 1, testInstances
            assertTrue testInstances[0] instanceof Item
    }
  void testCountItems() {
            // Mock the domain class, this time providing a list of test
            // Item instances that can be searched.
            def testInstances = [ new Item(code: "NH-4273997", name: "Laptop"),
                                  new Item(code: "EC-4395734", name: "Lamp"),
                                  new Item(code: "TF-4927324", name: "Laptop") ]
            mockDomain(Item, testInstances)
        // Initialise the service and test the target method. def testService = new MyService()
          assertEquals 2, testService.countItems("Laptop")
            assertEquals 1, testService.countItems("Lamp")
            assertEquals 0, testService.countItems("Chair")
    }
}
```

上面代码出现了一些新的方法，但是一旦对它们进一步解释，你应该很快懂得要使用这些方法是多么容易。首先看 testCreateSomething()测试方法里调用的 mockDomain()方法，这是 GrailsUnitTestCase 类提供的其中一个方法：

```
def testInstances = []
mockDomain(Item, testInstances)
```

这个方法可以给给定的类添加所有共同域的方法（实例和静态），这样任何使用它的代码段都可以把它当作一个真正全面的 domain 类。举个例子，一旦 Item 类被模拟了，我们就可以在实例它的时候放心得调用 save()；那么这时，如果我们调用一个被模拟的 domain 类的这个方法，要怎么做？很简单，在 testInstances 数组列表里添加新的实例，这个数组被当

成参数传进 mockDomain()方法。

下面我们将重点讲解 mockFor 方法：

```
def otherControl = mockFor(OtherService)
otherControl.demand.newIdentifier(1..1) {-> return testId }
```

这段代码功能与 Groovy 中的 MockFor 类和 StubFor 类非常接近，你可以用这个方法模拟任何类。事实上，上述 demand 语法跟 MockFor 和 StubFor 使用的语法一样，所以你在用它时应该不会觉得有差别，当然你要需要频繁注入一个 mock 实例作为关联，但是你可以简单得调用上述的 mock 控制对象的 createMock()方法，很容易实现。对那些熟悉 EasyMock 用法的人，它们知道这是 otherControl 强调了 mockFor()返回的对象角色，它是一个控制对象而非 mock 对象。

testCreateSomething()方法中剩余部分应该很熟悉了，特别是你现在已经知道了 save()模拟方法是往 testInstances 数组里添加实例。我们能确定 newIdentifier()模拟方法被调用，因为它返回的值对 createSomething()方法返回结果产生直接的影响。那假如情况不是这样呢？我们怎么知道它是否被调用？在 MockFor 类和 StubFor 类中，use()方法最后会做这个检查，但是 testCreateSomething()方法中没有这个方法，然而你可以调用控制对象的 verify()方法。在这个例子中，otherControl 对象可以实现。这个方法会执行检查，在 newIdentifier()应该被调用但没有被调用的情况下抛出诊断结果。

最后，这个例子中的 testCountItems()向我们展示了 mockDomain()方法的另外一个特性：

```
def testInstances = [ new Item(code: "NH-4273997", name: "Laptop"),
                      new Item(code: "EC-4395734", name: "Lamp"),
                      new Item(code: "TF-4927324", name: "Laptop") ]
mockDomain(Item, testInstances)
```

通常手工模拟动态遍历器比较烦人，而且你经常不得不为每次执行设置不同的数组；除了这个之外，假如你决定使用一个不同的遍历器，你就不得不更新测试实例去测试新的方法。感谢 mockDomain()方法为一组域实例的动态遍历器提供了一个轻量级的执行实现，把测试数据简单得作为这个方法的第二个参数，模拟遍历器就会工作了。

GrailsUnitTestCase - 模拟方法

你已经看过了一些介绍 GrailsUnitTestCase 中 mock..()方法的例子。在这部分我们将详细地介绍所有 GrailsUnitTestCase 中提供的方法，首先以通用的 mockFor()开始。在开始之

前，有一个很重要的说明先说一下，使用这些方法可以保证对所给的类做出的任何改变都不会让其他测试实例受影响。这里有个普遍出现且严重的问题，当你尝试通过 meta-class 编程方法对它自身进行模拟，但是只要你对每个想模拟的类使用任何一个 mock..()方法，这个问题就会消失了。

```
mockFor(class, loose = false)
```

万能的 mockFor 方法允许你对你一个类设置 strict 或 loose 请求。

这个方法很容易使用，默认情况下它会创建一个 strict 模式的 mock 控制对象，它的方法调用顺序非常重要，你可以使用这个对象详细定义各种需求：

```
def strictControl = mockFor(MyService)
strictControl.demand.someMethod(0..2) { String arg1, int arg2 -> ... }
strictControl.demand.static.aStaticMethod {-> ... }
```

注意你可以在 demand 后简单得使用 static 属性，就可以 mock 静态方法，然后定义你想 mock 的方法名字，一个可选的 range 范围作为它的参数。这个范围决定这个方法会被调用了多少次，所以假如这个方法的执行次数超过了这个范围，偏小或偏大，一个诊断异常就会被抛出。假如这个范围没有定义，默认的是使用“1..1”范围，比如上面定义的那个方法就只能被调用一次。

demand 的最后一部分是 closure，它代表了这个 mock 方法的实现部分。closure 的参数列表应该与被 mock 方法的数量和类型相匹配，但是同时你可以随意在 closure 主体里添加你想要的代码。

像之前提到的，假如你想生成一个你正在模拟类的能用 mock 实例，你就需要调用 mockControl.createMock()。事实上，你可以调用这个方法生成你想要的任何数量的 mock 实例。一旦执行了 test 方法，你就可以调用 mockControl.verify()方法检查你想要执行的方法执行了没。

最后，如下这个调用：

```
def looseControl = mockFor(MyService, true)
```

将生成一个含有 loose 特性的 mock 控制对象，比如方法调用的顺序不重要。

mockDomain(class, testInstances =)

这个方法选一个类作为它的参数，让所有 domain 类的非静态方法和静态方法的 mock 实现都可以在这个类调用到。

使用测试插件模拟 domain 类是其中的一个优势。手工模拟无论如何都是很麻烦的，所以 mockDomain()方法帮你减轻这个负担是多么美妙。

实际上，mockDomain()方法提供了 domain 类的轻量级实现，database 只是存储在内存里的一组 domain 实例。所有的 mock 方法，save()，get()，findBy*()等都可以按你的期望在这组实例里运行。除了这些功能之外，save()和 validate()模拟方法会执行真正的检查确认，包括对唯一的限制条件支持，它们会对相应的 domain 实例产生一个错误对象。

这里没什么其他要说了，除了插件不支持标准查询语句和 HQL 查询语句模拟。假如你想使用其中的一个，你可以简单得手工 mock 相应的方法，比如用 mockFor()方法，或用真实的数据测试一个集成实例。

mockForConstraintsTests(class, testInstances =)

这个方法可以对 domain 类和 command 对象进行非常详细地模拟设置，它允许你确认各种约束是否按你想要的方式执行。

你测试 domain 约束了？如果没有，为什么没有？如果你的回答是它们不需要测试，请你三思。你的各种约束包含逻辑部分，这部分逻辑很容易产生 bug，而这类 bug 很容易被捕捉到，特别的是 save()允许失败也不会抛出异常。而如果你的回答是太难或太烦，现在这已经不再是借口了，可以用 mockForConstraintsTests()解决这个问题。

这个方法就像 mockDomain()方法的简化版本，简单得对所给的 domain 类添加一个 validate()方法。你所要做的就是 mock 这个类，创建带有属性值的实例，然后调用 validate()方法。你可以查看 domain 实例的 errors 属性判断这个确认方法是否失败。所以假如所有我们正在做的是模拟 validate()方法，那么可选的测试实例数组参数呢？这就是我们为什么可以测试唯一约束的原因，你很快就可以看见了。

那么假设我们拥有如下的一个简单 domain 类：

```
class Book {
   String title
   String author
   static constraints = {
            title(blank: false, unique: true)
            author(blank: false, minSize: 5)
      }
}
```

不要担心这些约束是否合理，它们在这仅仅是示范作用。为了测试这些约束，我们可以按下

面方法来做：

```
class BookTests extends GrailsUnitTestCase {
    void testConstraints() {
        def existingBook = new Book(title: "Misery", author: "Stephen King")
        mockForConstraintsTests(Book, [ existingBook ])
          // Validation should fail if both properties are null.
              def book = new Book()
              assertFalse book.validate()
              assertEquals "nullable", book.errors["title"]
              assertEquals "nullable", book.errors["author"]
           // So let's demonstrate the unique and minSize constraints.
              book = new Book(title: "Misery", author: "JK")
              assertFalse book.validate()
              assertEquals "unique",      book.errors["title"]
              assertEquals "minSize", book.errors["author"]
            // Validation should pass!
              book = new Book(title: "The Shining", author: "Stephen King")
              assertTrue book.validate()
        }
}
```

你可以在没有进一步解释的情况下，阅读上面这些代码，思考它们正在做什么事情。我们会解释的唯一一件事是 errors 属性使用的方式。第一，它返回了真实的 Spring Errors 实例，所以你可以得到你通常期望的所有属性和方法。第二，这个特殊的 Errors 对象也可以用如上 map/property 方式使用。简单地读取你感兴趣的属性名字，map/property 接口会返回被确认的约束名字。注意它是约束的名字，不是你所期望的信息内容。

这是测试约束讲解部分。我们要讲的最后一件事是用这种方式测试约束会捕捉一个共同的错误：typos in the "constraints" property。正常情况下这是目前最难捕捉的一个 bug，还没有一个约束单元测试可以直接简单得发现这个问题。

mockLogging(class, enableDebug = false)

这个方法可以给一个类增加一个 mock 的 log 属性，任何传递给 mock 的 logger 的信息都会输出到控制台的。

mockController(class)

此方法可以为指定类添加 mock 版本的动态控制器属性和方法，通常它和

ControllerUnitTestCase 一起连用。

mockTagLib(class)

此方法可以为指定类添加 mock 版本的动态 tablib 属性和方法，通常它和 TagLibUnitTestCase 一起连用。

9.2 集成测试

集成测试与单元测试不同的是在测试实例内你拥有使用 Grails 环境的全部权限。Grails 将使用一个内存内的 HSQLDB 数据库作为集成测试，清理每个测试之间的数据库的数据。

测试控制器

测试控制器之前你首先要了解 Spring Mock Library。

实质上，Grails 自动用 MockHttpServletRequest，MockHttpServletResponse，和 MockHttpSession 配置每个测试实例，你可以使用它们执行你的测试用例。比如你可以考虑如下 controller：

```
class FooController {
       def text = {
          render "bar"
       }
   def someRedirect = {
             redirect(action:"bar")
       }
}
```

它的测试用例如下：

```
class FooControllerTests extends GroovyTestCase {
  void testText() {
             def fc = new FooController()
             fc.text()
             assertEquals "bar", fc.response.contentAsString
       }
   void testSomeRedirect() {
          def fc = new FooController()
             fc.someRedirect()
             assertEquals "/foo/bar", fc.response.redirectedUrl
```

```
    }
}
```

在上面的实例中，返回对象是一个 MockHttpServletResponse 实例，你可以使用这个实例获取写进返回对象的 contentAsString 值，或是跳转的 URL。这些 Servlet API 的模拟版本全部都很更改，不像模拟之前那样子，因此你可以对请求对象设置属性，比如 contextPath 等。

Grails 在集成测试期间调用 actions 不会自动执行 interceptors，你要单独测试拦截器，必要的话通过 functional testing 测试。

用应用测试控制器

假如你的控制器引用了一个应用服务，你必须在测试实例里显示初始化这个应用。

举个使用应用的控制器例子：

```
class FilmStarsController {
   def popularityService
   def update = {
      // do something with popularityService
   }
}
```

相应的测试实例：

```
class FilmStarsTests extends GroovyTestCase {
   def popularityService
   public void testInjectedServiceInController () {
      def fsc = new FilmStarsController()
      fsc.popularityService = popularityService
      fsc.update()
   }
}
```

测试控制器 command 对象

使用 command 对象，你可以给请求对象 request 提供参数，当你调用没有带参数的 action 处理对象时，它会自动为你做 command 对象工作。

举个带有 command 对象的控制器例子：

```
class AuthenticationController {
```

```
    def signup = { SignupForm form ->
        ...
    }
}
```

你可以如下对它进行测试：

```
def controller = new AuthenticationController()
controller.params.login = "marcpalmer"
controller.params.password = "secret"
controller.params.passwordConfirm = "secret"
controller.signup()
```

Grails 把 signup()的调用自动当作对处理对象的调用，利用模拟请求参数生成 command 对象。在控制器测试期间，params 通过 Grails 的模拟请求对象是可更改的。

测试控制器和 render 方法

render 方法允许你在一个 action 主体内的任何一个地方显示一个定制的视图。例如，考虑如下的例子：

```
def save = {
        def book = Book(params)
        if(book.save()) {
                // handle
        }
        else {
                render(view:"create", model:[book:book])
        }
}
```

上面举的这个例子中，处理对象用返回值作这个模型的结果是不可行的，相反结果保存在控制对象的 modelAndView 属性当中。modelAndView 属性是 Spring MVC ModelAndView 类的一个实例，你可以用它测试一个 action 处理后的结果：

```
def bookController = new BookController()
bookController.save()
def model = bookController.modelAndView.model.book
```

模拟生成请求数据

如果你测试一个 action 请求处理对象需要类似 REST web 应用的请求参数，你可以使用 Spring MockHttpServletRequest 对象实现。例如，考虑如下这个 action，它执行一个进来请求的数据邦定：

```
def create = {
        [book: new Book(params['book']) ]
}
```

假如你想把 book 参数模拟成一个 XML 请求对象，你可以按如下方法做：

```
void testCreateWithXML() {
        def controller = new BookController()
        controller.request.contentType = 'text/xml'
        controller.request.contents = '''<?xml version="1.0" encoding="ISO-8859-1"?>
        <book>
                <title>The Stand</title>
                ...
        </book>
        '''.getBytes() // note we need the bytes
  def model = controller.create()
        assert model.book
        assertEquals "The Stand", model.book.title
}
```

同样你可以通过 JSON 对象达到这个目的：

```
void testCreateWithJSON() {
        def controller = new BookController()
        controller.request.contentType = "text/json"
        controller.request.content = '{"id":1,"class":"Book","title":"The Stand"}'.getBytes()
       def model = controller.create()
        assert model.book
        assertEquals "The Stand", model.book.title
}
```

> 使用 JSON，也不要忘记对 class 属性指定名字，绑定的目标类型。在 XML 里，在 book 节点内这些设置隐含的，但是使用 JSON 你需要这个属性作为 JSON 包的一部分。

更多关于 REST web 应用的信息，可以参考 REST 章节。

测试 Web Flows

测试 Web Flows 需要一个特殊的测试工具 grails.test.WebFlowTestCase，它继承 Spring Web Flow 的 AbstractFlowExecutionTests 类。Testing Web Flows requires a special test harness called grails.test.WebFlowTestCase which sub classes Spring Web Flow's AbstractFlowExecutionTests class.

> WebFlowTestCase 子类必须是集成测试实例 Subclasses of WebFlowTestCase **must** be integration tests

例如在下面的这个小 flow 情况下：

```
class ExampleController {
    def exampleFlow = {
        start {
            on("go") {
                flow.hello = "world"
            }.to "next"
        }
        next {
            on("back").to "start"
            on("go").to "end"
        }
        end()
    }
}
```

接着你需要让测试工具知道使用什么样的 flow 定义。通过重载 getFlow 抽象方法可以实现：

```
class ExampleFlowTests extends grails.test.WebFlowTestCase {
    def getFlow() { new ExampleController().exampleFlow }
    ...
}
```

假如你需要指定一个 flow 标识，你可以通过重载 getFlowId 方法实现，同时默认情况下是一个测试实例：

```
class ExampleFlowTests extends grails.test.WebFlowTestCase {
    String getFlowId() { "example" }
    ...
}
```

一旦这在你的测试实例里实现了，你需要用 startFlow 方法开始启动这个 flow，这个方法会返回 ViewSelection 对象：

```
void testExampleFlow() {
        def viewSelection = startFlow()
   assertEquals "start", viewSelection.viewName
        …
}
```

如上所示，你可以通过使用 ViewSelection 对象的 viewName 属性，检查你是否是正确的。触发事件你需要使用 signalEvent 方法：

```
void testExampleFlow() {
        …
        viewSelection = signalEvent("go")
        assertEquals "next", viewSelection.viewName
        assertEquals "world", viewSelection.model.hello
}
```

这里我们可以给 flow 发送信号执行 go 事件，这导致了到 next 状态的转变。在上面的这个例子中转变的结果把一个 hello 变量放进 flow 范围。我们可以检查如上 ViewSelection 的 model 属性测试这个变量的值。

测试标签库

其实测试标签库是一件很容易的事，因为当一个标签被当作一个方法执行时，它会返回一个字符串值。所以例如你拥有如下的一个标签库：

```
class FooTagLib {
   def bar = { attrs, body ->
         out << "<p>Hello World!</p>"
   }
   def bodyTag = { attrs, body ->
      out << "<${attrs.name}>"
         out << body()
      out << "</${attrs.name}>"
   }
}
```

相应的测试如下：

```
class FooTagLibTests extends GroovyTestCase {
```

```
  void testBarTag() {
        assertEquals "<p>Hello World!</p>", new FooTagLib().bar(null,null)
   }
   void testBodyTag() {
        assertEquals "<p>Hello World!</p>", new FooTagLib().bodyTag(name:"p") {
           "Hello World!"
      }
   }
}
```

注意在第二个例子的 testBodyTag 中，我们传递了返回标签主体内容的代码块作为内容，把标签主体内容作为字符串比较方便。

使用 GroovyPagesTestCase 测试标签库

除了上述简单的标签库测试方法之外，你也可以使用 grails.test.GroovyPagesTestCase 类测试标签库。

GroovyPagesTestCase 类是常见 GroovyTestCase 的子类，它为 GSP 显示输出提供实用方法。

> GroovyPagesTestCase 类只能在集成测试中使用。

举个时间格式化标签库的例子，如下：

```
class FormatTagLib {
      def dateFormat = { attrs, body ->
            out << new java.text.SimpleDateFormat(attrs.format) << attrs.date
      }
}
```

可以按如下方法进行测试：

```
class FormatTagLibTests extends GroovyPagesTestCase {
      void testDateFormat() {
            def template = '<g:dateFormat format="dd-MM-yyyy" date="${myDate}" />'
        def testDate = ... // create the date
            assertOutputEquals( '01-01-2008', template, [myDate:testDate] )
      }
}
```

你也可以使用 GroovyPagesTestCase 的 applyTemplate 方法获取 GSP 的输出结果：

```
class FormatTagLibTests extends GroovyPagesTestCase {
        void testDateFormat() {
                def template = '<g:dateFormat format="dd-MM-yyyy" date="${myDate}" />'
          def testDate = ... // create the date
                def result = applyTemplate( template, [myDate:testDate] )
         assertEquals '01-01-2008', result
        }
}
```

测试 Domain 类

用 GORM API 测试 domain 类是一件很简单的事情，然而你还要注意一些事项。第一，假如你在测试查询语句，你将经常需要 flush 以便保证正确的状态持久保存到数据库。比如下面的一个例子：

```
void testQuery() {
        def books = [ new Book(title:"The Stand"), new Book(title:"The Shining")]
        books*.save()
 assertEquals 2, Book.list().size()
}
```

这个测试实际上会失败，因为调用 save 方法的时候，save 方法不会真的持久保存 book 实例。调用 save 方法仅仅是向 Hibernate 暗示在将来的某个时候这些实例应该会被保存。假如你希望立即提交这些改变，你需要 flush 它们：

```
void testQuery() {
        def books = [ new Book(title:"The Stand"), new Book(title:"The Shining")]
        books*.save(flush:true)
 assertEquals 2, Book.list().size()
}
```

在这个案例中我们传递了 flush 的 true 值作为参数，更新将马上被保存，因此对此后的查询语句也有效。

9.3 功能测试

功能测试是测试正在运行的应用，经常自动化较难实现。Grails 没有发布任何功能测试开箱即用支持，但是通过插件实现了对 Canoo WebTest 的支持。

首先按如下的命令按照 Web Test：

grails install-plugin webtest

参考 reference on the wiki ，它里面了解释怎么使用 Web Test 和 Grails。 Show details

10. 国际化

通过潜在支持 Spring MVC 国际化标准，Grails 支持国际化标准(i18n)开箱即用。在 Grails 中，你能够根据用户属地定制该地区语言的文字。引用 Java 语言中的 Locale 文档定义：

> Locale 对象描述了特定的地理、政治和文化地区。需要 Locale 来执行其任务的操作称为本地化操作，它使用 Locale 为用户量身定制信息。例如，显示一个数值就是本地化操作，应该根据用户本国家、地区或文化的风俗/传统来格式化该数值。

一个 Locale 对象由 language code 和 country code 组成。比如，en_US 是美国英语的代码，而 en_GB 是英国英语的代码。

10.1 理解消息绑定

现在你知道了本地化，为了在 Grails 中使用它们，你不得不创建你想显示的不同语言信息资源。Grails 中的信息资源以简单的 java 属性文件格式放置在 grails-app/i18n 目录下。

每个资源束根据规则，以 messages 名字开始和 locale 结束。Grails 在 grails-app/i18n 下发布了一串不同语言范围内的内置信息，例如：

messages.properties
messages_de.properties
messages_es.properties
etc.

默认情况 Grails 会在 messages.properties 文件中检索信息，除非用户已经指定了一个自定义本地化文件。通过创建一个新的以 locale 标签结尾的属性文件，你可以创建你感兴趣的属于自己的信息资源。比如属于英式英语范畴的 messages_en_GB.properties。

10.2 修改本地化

默认情况用户地区从传进来的 Accept-Language 头部得知。然而 通过简单得给 Grails 传进 lang 参数作为请求参数，用户就可以更改地区了：

/book/list?lang=es

Grails 会自动更改用户地区，并把这个值存放在 cookie 里，随后的各种请求会有个新的头部。

10.3 读取信息

视图中读取信息

你 通常最需要信息的地方是在视图内。要在视图内读取信息，使用 message 标签就可以了，如下：

```
<g:message code="my.localized.content" />
```

只要你在带有合适 locale 后缀的 messages.properties 文件有个 key 键，比如下面这种格式，Grails 就会找到相对应的信息：

```
my.localized.content=Hola, Me llamo John. Hoy es domingo.
```

注 意有时候你需要向相应的信息传递参数。参考下面这个 message 标签：

```
<g:message code="my.localized.content" args="${ ['Juan', 'lunes'] }" />
```

还有可能在信息中使用定位参数：

```
my.localized.content=Hola, Me llamo {0}. Hoy es {1}.
```

控制器和标签库中读取信息

因为你可以在 controllers 中像方法一样使用标签，所以在 controllers 中读取信息也很经常，如下：

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

tag libraries 中使用的方法一样，但是注意如果你的标签库使用了不同的 namespace，你需要使用 g.前缀：

```
def myTag = { attrs, body ->
 def msg = g.message(code:"my.localized.content", args:['Juan', 'lunes'])
}
```

10.4 脚手架和 i18n

Grails 没有发布可以生成控制器和视图的 i18n 特性脚手架模板。然而 i18n 模板插件可以提供 i18n 特性脚手架模板，这些模板与默认的脚手架模板一样，除了它们为标签，按钮等定义信息时使用 message 标签外。

首先用下面的这个命令安装 i18n 模板：

```
grails install-plugin i18n-templates
```

参考 reference on the wiki，它里面了解释怎么使用 i18n 模板。

11. 安全性

Grails 差不多和 Java Servlets 一样可靠。然而由于 JVM 运行代码的特性，Java servlets 对一般的缓冲区溢出和恶意 URL 使用是极为安全和免疫的。

Web 安全问题通常由于开发人员的无知过错造成的，Grails 提供了一些帮助，可以避免常出现的错误，使安全应用更加容易编写。

Grails 可以自动做什么

Grails 拥有一些默认的内置安全机制

1. 所有通过 GORM 域对象访问标准数据库可以自动避免 SQL 语句以防止 SQL 注入攻击。

2. 默认 scaffolding 模板 HTML 文件当打开时所有数据域不显示。
3. 所有 Grails 的链接创建标签(link, form, createLink, createLinkTo 等)都使用适当的转义机制以防止代码注入。
4. Grails 提供 codecs，运行你在显示 HTML，JavaScript 和 URLs 时，转义数据以避免在数据里注入攻击。

11.1 防止攻击

SQL 注入

Hibernate 是实现 GORM 域类的基础技术，当提交数据库时会自动转义数据，所以这个没什么问题。然而编写使用未检查的请求参数的脏动态 HQL 代码，仍然会有问题可能存在。

比如如下的这种做法就很容易受 HQL 注入攻击：

```
def vulnerable = {
    def books = Book.find("from Book as b where b.title ='" + params.title + "'")
}
```

千万别这样做。假如你想传递参数，用命名参数和定位参数代替：

```
def safe = {
    def books = Book.find("from Book as b where b.title =?", [params.title])
}
```

钓鱼式攻击

这是一个公关关系问题，涉及到避免你的品牌化过程和与顾客设定的沟通手段遭到黑客攻击。顾客需要知道怎么确认收到的 emails 是真的。

XSS-跨站脚本攻击

你的应用要尽可能多得检验进来的请求是从你的应用里发出的，而不是其它网站。标签和页面流系统能做到这点，Grails 对 Spring Web Flow 的支持也默认包含了这个安全特性。

确保所有呈现到视图的数据值都被转义过也是非常重要的。例如当呈现 HTML 文件或 XHTML 文件时，你必须对每个对象调用 encodeAsHTML，以便保证用户不会向其他人读取的数据和标签恶意注入 JavaScript 代码或其他 HTML 代码。Grails 为此目的提供了若干个动态编码方法，因此假如你的输出转义格式没有现成的，你可以很容易得编写自己的编码器。

你也必须避免使用请求参数和数据域来决定用户转向的下一个链接。假如你使用一个 successURL 参数，在你成功登入之后，用来指示用户的转向；这时攻击者可以通过你的网站模拟登入程序，然后一旦登入就把用户转向到他们的网站，这样就潜在允许 JS 代码使用该网站的登入帐号。

HTML/URL 注入

HTML 和 URL 注入提供有害的数据，之后被用来在页面生成一个链接，点击它不会产生期望的行为，可能会转向另外一个网站或更改请求参数。

Grails 提供的 codecs 可以很容易得处理 HTML/URL 注入，Grails 提供的标签库在适用的地方全都使用 encodeAsURL。如果你自己创建能生成链接的标签，你在做的时候要小心。

拒绝服务 DoS

负载均衡器和其他应用在这里可能会起到用处，但是还存在其他问题，比如过度查询，攻击者创建一个链接设置结果集的最大值，导致一个查询超过服务器的最大内存限制或拖慢系统运行。解决办法是在请求参数传进动态遍历器或其他 GORM 查询方法之前，给这些请求参数“消毒”：

```
def safeMax = Math.max(params.max?.toInteger(), 100) // never let more than 100 results be returned
return Book.list(max:safeMax)
```

可推测 ID 号

许多应用把 URL 的最后一部分当作从 GORM 或者其他地方获取的某个对象的 id。特别是当发生在 GORM 中时，这些 id 号是很容易猜测的，因为这些 id 号通常是一串数字。

因此你必须假定请求用户在请求返回时用请求 id 号可以看见相对应的对象。

不这样做是隐藏式安全，这样做毫无疑问是非法的，像有 letmein 的默认密码等等这些情况。

你必须假设每个未受保护 URL 都可以公共访问。

11.2 编码和解码对象

Grails 支持动态编码/解码方法概念。Grails 捆绑了一些标准的编解码器，Grails 也为开发人员提供了一个贡献自己编解码器的简单机制，这些编解码器在运行时可以被识别。

编解码器类

一个 Grails 编解码器是个包含一个编码闭包，一个解码闭包或两者皆有。当一个 Grails 应用启用了，Grails 框架会动态从 grails-app/utils/目录加载编解码器。

Grails 框架将在 grails-app/utils/目录下查找以 Codec 结尾命名的类名。例如 Grails 捆绑的其中一个标准编解码器就是 HTMLCodec。

假如一个编解码器包含一个 encode 属性，该属性被赋予一个代码块，Grails 会创建一个动态的 encode 方法，并把该方法添加到 Object 类，方法名表示了定义 encode 闭包的编解码器。例如，HTMLCodec 类定义了一个编码器代码块，因此 Grails 会把该闭包与名为 encodeAsHTML 的 Object 类相关联。

HTMLCodec 类和 URLCodec 类也定义了解码块，所以 Grails 会把这些闭包与 decodeHTML 和 decodeURL 相关联的。动态编解码器能在 Grails 应用的任何一个地方执行。例如，考虑一下这种情况，一个报告文件含有一个叫 description 的属性，该属性包含了需要被转义显示在 HTML 文档的特殊字符。GSP 文档里，一种处理方法就是用如下的动态编码器编码 description 属性：

```
${report.description.encodeAsHTML()}
```

执行解码使用 value.decodeHTML()语句。

标准的编解码器

HTMLCodec

该编解码器执行 HTML 转义过程和反转义过程，所以你提供的数值在没有创建任何 HTML 标签或破坏页面布局下可以被安全得显示出来。例如，给个"Don't you know that 2 > 1?"字符串，你就不能在 HTML 页面中安全得显示出来，因为大于符号>看起来像要关闭一个标签，特别是你在某个属性内显示这个字符串，情况会更糟糕，像输入框的 value 属性。

使用例子如下：

```
<input name="comment.message" value="${comment.message.encodeAsHTML()}"/>
```

> 注意 HTML 编码不会重新编码单引号或双引号，你必须对属性值只用两个重复引号避免含有引号的正文毁坏你的页面。

URLCodec

当在生成跳转链接，形体处理(form actions)链接，或者任何时候需要数据生成链接时，URL 编码是必需的。URL 编码可以阻止非法字符串进入链接改变它跳转的目的地，例如"Apple & Blackberry"不能作为 get 请求中的一个参数，因为&符号为破坏参数解析过程。

使用例子如下：

```
<a href="/mycontroller/find?searchKey=${lastSearch.encodeAsURL()}">Repeat last search</a>
```

Base64Codec

执行 Base64 编码/解码函数，使用例子如下：

```
Your registration code is: ${user.registrationCode.encodeAsBase64()}
```

JavaScriptCodec

JavaScriptCodec 会转义字符串成为合法的 JavaScript 字符串，使用例子如下：

```
Element.update('${elementId}', '${render(template: "/common/message").encodeAsJavaScript()}')
```

HexCodec

HexCodec 会把字节数组或数字数列编码为小写十六进制字符串，可以把十六进制字符串编码为字节数组，使用例子如下：

```
Selected colour: #${[255,127,255].encodeAsHex()}
```

MD5Codec

MD5Codec 使用 MD5 算法摘要字节数组，数字数列或默认系统编码的字符串字节数组，得到一格小写十六进制字符串，使用例子如下：

```
Your API Key: ${user.uniqueID.encodeAsMD5()}
```

MD5BytesCodec

MD5BytesCodec 使用 MD5 算法摘要字节数组，数字数列或默认系统编码的字符串字节数组，得到一个字节数组，使用例子如下：

```
byte[] passwordHash = params.password.encodeAsMD5Bytes()
```

SHA1Codec

SHA1Codec 使用 SHA1 算法摘要字节数组，数字数列或默认系统编码的字符串字节数组，得到一格小写十六进制字符串，使用例子如下：

```
Your API Key: ${user.uniqueID.encodeAsSHA1()}
```

SHA1BytesCodec

SHA1BytesCodec 使用 SHA1 算法摘要字节数组，数字数列或默认系统编码的字符串字节数组，得到一个字节数组，使用例子如下：

```
byte[] passwordHash = params.password.encodeAsSHA1Bytes()
```

SHA256Codec

SHA256Codec 使用 SHA256 算法摘要字节数组，数字数列或默认系统编码的字符串字节数组，得到一格小写十六进制字符串，使用例子如下：

Your API Key: ${user.uniqueID.encodeAsSHA256()}

SHA256BytesCodec

SHA256BytesCodec 使用 SHA1 算法摘要字节数组，数字数列或默认系统编码的字符串字节数组，得到一个字节数组，使用例子如下：

```
byte[] passwordHash = params.password.encodeAsSHA256Bytes()
```

定制编解码器 Custom Codecs

许多应用可能定制属于自己的编解码器，Grails 在装载标准编解码器时把它们一起装载。定制编解码器类必须在 grails-app/utils/目录下定义，而且类名必须以 Codec 结尾。定制编解码器可能含有一个静态 encode 块，一个静态 decode 块或两者皆有。这些编解码代码块需要一个单一参数，当作动态方法操作对象，如下：

```
class PigLatinCodec {
  static encode = { str ->
    // convert the string to piglatin and return the result
  }
}
```

在适当的地方，一个应用可以使用上方定义的编解码器做如下的工作：

```
${lastName.encodeAsPigLatin()}
```

11.3 认证

尽管现在认证没有默认机制，实际上有上千种方法可以执行认证。然而，用 interceptors 或 filters 实施一个简单的认证机制是没意义的。

过滤器运行你对所有的控制器或 URI 空间应用认证。比如你可以在 grails-app/conf/SecurityFilters.groovy 类中创建一组新过滤器如下：

```
class SecurityFilters {
   def filters = {
      loginCheck(controller:'*', action:'*') {
         before = {
           if(!session.user && actionName != "login") {
              redirect(controller:"user",action:"login")
              return false
```

```
            }
        }
    }
  }
}
```

在请求处理执行之前，上述类中的 loginCheck 过滤器将拦截该执行动作。假如 session 里没有一个用户而且请求被执行，该 login 请求处理不是自己转向到自己。

Login 请求处理也是很小的：

```
def login = {
    if(request.get) render(view:"login")
    else {
        def u = User.findByLogin(params.login)
        if(u) {
            if(u.password == params.password) {
                session.user = u
                redirect(action:"home")
            }
            else {
                render(view:"login", model:[message:"Password incorrect"])
            }
        }
        else {
            render(view:"login", model:[message:"User not found"])
        }
    }
}
```

11.4 安全插件

如果你需要比简单认证更高级的功能，诸如授权(authorization)，角色(roles)等，那么你可能要考虑使用一个可用的安全插件。

11.4.1 Acegi

Acegi 插件是建立在 Spring Acegi 项目上，该项目为建立各种认证和授权架构提供了一个灵活，易扩展的框架。

Acegi 插件需要你在 URI 和角色之间制定个详细的映射，为规范人，权威专家和请求 maps 提供一个默认的领域模型 domain model。点击 documentation on the wiki，查看更多信息。

11.4.2 JSecurity

JSecurity 是另外一个面向 Java POJO 的安全框架，它也可以提供一个规范领域，用户，角色和权限的默认领域模型。使用 JSecurity，你必须让每个你想保护的 controller 类继承一个 controller 基类，然后提供一个建立角色的 accessControl 代码块。例子如下：

```
class ExampleController extends JsecAuthBase {
    static accessControl = {
        // All actions require the 'Observer' role.
        role(name: 'Observer')
        // The 'edit' action requires the 'Administrator' role.
        role(name: 'Administrator', action: 'edit')
        // Alternatively, several actions can be specified.
        role(name: 'Administrator', only: [ 'create', 'edit', 'save', 'update' ])
    }
    ...
}
```

更多关于 JSecurity 的信息，参考 JSecurity Quick Start。

12. 插件

Grails 提供了许多扩展点来满足你的扩展，包括从命令行接口到运行时配置引擎。以下章节详细说明了该如何着手来做这些扩展。

12.1 创建和安装插件

创建插件

创建一个 Grails 插件，只需要运行如下命令即可：

```
grails create-plugin [PLUGIN NAME]
```

根据你输入的名字将产生一插件工程。比如你输入 grails create-plugin example. 系统将

创建一个名为 example 的插件工程.

除了插件的根目录有一个所谓的“插件描述”的 Groovy 文件外，其他的跟一般的 Grails 工程结构完全一样.

将插件作为一个常规的 Grails 工程是有好处的，比如你可以马上用以下命令来测试你的插件:

```
grails run-app
```

> 由于你创建插件默认是没有 URL 映射的,因此控制器并不会马上有效.如果你的插件需要控制器，那要创建 grails-app/conf/MyUrlMappings.groovy 文件,并且在起始位置增加缺省的映射 "/$controller/$action?/$id?"().

插件描述文件本身需要符合以 GrailsPlugin 结尾的惯例并且将位于插件工程的根目录中。比如:

```
class ExampleGrailsPlugin {
  def version = 0.1
  ...
}
```

所有插件的根目录下边都必须有此类并且还要有效，此类中定义了插件的版本和其他各式各样的可选的插件扩展点的钩子（hooks）--即插件预留的可以扩展的接口.

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

以 Quartz Grails plugin 为例：

```
class QuartzGrailsPlugin {
   def version = "0.1"
   def author = "Sergey Nebolsin"
   def authorEmail = "nebolsin@gmail.com"
   def title = "This plugin adds Quartz job scheduling features to Grails application."
   def description = '''
Quartz plugin allows your Grails application to schedule jobs to be
```

```
executed using a specified interval or cron expression. The underlying
system uses the Quartz Enterprise Job Scheduler configured via Spring,
but is made simpler by the coding by convention paradigm.
'''
    def documentation = "http://grails.org/Quartz+plugin"
   ...
}
```

插件的安装和发布

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. 以先前的 example 插件为例，这个文件名是 grails-example-0.1.zip. package-plugin 命令还将生成 plugin.xml f 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。

产生了可以发布的插件文件以后（zip 文件），进入到你自己的 Grails 工程的根目录，输入:

```
grails install-plugin /path/to/plugin/grails-example-0.1.zip
```

如果你的插件放在远程的 Http 服务器上，你也可以这样:

```
grails install-plugin http://myserver.com/plugins/grails-example-0.1.zip
```

注意被排除的组件

尽管 create-plugin 命令为您创建某些文件，以便插件能做为 Grails 应用运行，但是当打包插件的时候不是所有的文件都会在含在里面. 以下是通过 package-plugin 创建时，不包含的文件和目录：

- grails-app/conf/DataSource.groovy
- grails-app/conf/UrlMappings.groovy
- grails-app/conf/DataSource.groovy
- build.xml
- Everything within /web-app/WEB-INF

如果你希望创建包含 WEB-INF 目录的组建，那么建议你使用 _Install.groovy 脚本文件 (covered later)，这个脚本文件之后会解释；当安装一个插件提供这些组件时，这个脚本文件会被执行。 此外，除了用 UrlMappings.groovy 之外，也允许你使用包括 UrlMappings

名字来定义不同的名称，例如 FooUrlMappings.groovy

12.2 插件仓库

在 Grails 插件的存储仓库（Repository）发布插件

更好的发布插件的方式是将其发布到 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。

> 如果你创建了一个 Grails 插件，你可以访问 创建插件，这里详细说明了如何在容器中发布你的插件。

当你有访问 Grails 插件仓库的权限时，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.

配置附加库

默认情况下，您使用的 list-plugins, install-plugin and release-plugin 命令都指向 http://plugins.grails.org。

然而, 要配置多个插件仓库，您可以使用 grails-app/conf/BuildSettings.groovy 文件:

```
grails.plugin.repos.discovery.myRepository="http://svn.codehaus.org/grails/trunk/grails-test-plugin-repo"
grails.plugin.repos.distribution.myRepository="https://svn.codehaus.org/grails/trunk/grails-test-plugin-repo"
```

Repositories are split into those used for discovery over HTTP and those used for

distribution, typically over HTTPS. 如果你想在多个项目中使用相同的设置，你可以把这些配置到 USER_HOME/.grails/settings.groovy。

一旦使用了 list-plugins, install-plugin and plugin-info 命令将会自动处理最新配置的插件库。如果你只想把插件库中的插件列表列出来，你可以使用别名：

```
grails list-plugins -repository=myRepository
```

此外，如果你想和配置好的插件包一起发布插件，你可以用 release-plugin 命令：

```
grails release-plugin -repository=myRepository
```

12.3 理解插件的结构

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 plugins 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.

12.4 提供基础的工件

增加新的脚本

在插件的 scripts 目录下可以增加新的 Gant 相关的脚本:

```
+ MyPlugin.groovy
  + scripts     <-- additional scripts here
  + grails-app
      + controllers
      + services
      + etc.
   + lib
```

增加新的控制器，标签库或者服务

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
```

Providing Views, Templates and View resolution

提供控制器的插件也会提供默认的视图。通过插件模块化您的应用是个很好的途径。Grails 视图处理机制的工作原理是首先查看应用中被安装的视图，如果失败将视图查找插件中的视图。

比如有一个 AmazonGrailsPlugin 插件提供一个叫 BookController 的控制器，如果执行了 list 将会首先查找 grails-app/views/book/list.gsp 这个视图，如果失败，将会在插件里查找相同名称的视图。

但是，如果视图使用了模板，同时插件也提供了这个视图，那么必须使用以下的语法：

```
<g:render template="fooTemplate" contextPath="${pluginContextPath}"/>
```

注意 pluginContextPath 变量做为 contextPath 属性值的用法。如果没有指定这个属性，Grails 将在应用中的模板中查找。

Excluded Artefacts

默认的，when packaging a plug-in，当打包一个插件时，Grails 的插件包中将不包含以下文件：

- grails-app/conf/DataSource.groovy
- grails-app/conf/UrlMappings.groovy
- Everything under web-app/WEB-INF

如果你的插件需要 web-app/WEB-INF 目录下的文件，那么建议你修改插件的 scripts/_Install.groovy Gant 脚本文件把项目的目标目录安装到插件包中。

此外, UrlMappings.groovy 文件默认不会避免命名冲突，你可以使用在默认名字前加增加**前缀**。比如叫做 grails-app/conf/BlogUrlMappings.groovy。

12.5 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

完整的索引请参考 javadoc API.

12.6 参与构建事件

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"
```

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。更多详细信息请参考 Hooking into Events.

12.7 运行时配置中的钩子 Hooking into Runtime Configuration

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。

跟 Grails 的 Spring 配置进行交互

首先你可以使用 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:

```
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)
      }
}
```

这个插件建立起了 Grails messageSource bean 和一对其他 beans 以管理 Locale 解释和更改。它使用 Spring Bean Builder 语法。

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.

考虑如下来自 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。

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
       }
}
```

12.8 运行时添加动态方法

基础知识

Grails 插件允许你在运行时注册 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，你可以增加方法，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.each { controllerClass ->
            controllerClass.metaClass.myNewMethod = {-> println "hello world" }
        }
  }
}
```

此处我们通过隐含的 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，你已经知道要处理的类的类型了，那你只需要在此类的 metaClass 属性上增加一个方法即可，代码如下:

```
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()
  }
}
```

此例中，我们直接在 java.lang.String 的 metaClass 上增加一个新的 swapCase 方法.

跟 ApplicationContext 交互

doWithDynamicMethods 闭包的参数是 Spring 的 ApplicationContext 实例，这点非常有用，因为这允许你和该应用上下文实例中的对象进行交互。比如你打算实现一个跟 Hibernate 交互的方法，那你可以联合着 HibernateTemplate 来使用 SessionFactory 例，代码如下:

```
import org.springframework.orm.hibernate3.HibernateTemplate
class ExampleHibernatePlugin {
   def doWithDynamicMethods = { applicationContext ->
       application.domainClasses.each { domainClass ->
          domainClass.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，你可以在运行时实现更强大的动态

构造器，此构造器使用 applicationContext 来装配你的对象及其依赖：

```
class MyConstructorPlugin {
   def doWithDynamicMethods = { applicationContext ->
       application.domainClasses.each { domainClass ->
           domainClass.metaClass.constructor = {->
              return applicationContext.getBean(domainClass.name)
           }
       }
   }
}
```

这里我们实际做的是通过查找 Spring 的原型 beans（prototyped beans）来替代缺省的构造器。

12.9 参与自动重载

监控资源的改变

通常来讲，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 对象定义了一些有益的属性:

- event.source - The source of the event which is either the reloaded class or a Spring Resource
- event.ctx - The Spring ApplicationContext instance
- event.plugin - The plugin object that manages the resource (Usually this)
- event.application - The GrailsApplication instance

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.

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，除非你也重新加载 controllers，否则你将加载过的 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，你可以指定将要受到“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。你也可以使用一个通配符查看所有加载的插件：

```
def observe = ["*"]
```

Logging plugin 不仅如此，当应用运行时它都能添加 log 属性到 *任何* 插件库。

12.10 理解插件加载的顺序

Controlling Plug-in Dependencies

插件经常依赖于其他已经存在的插件，并且也能调整这种依赖. 为了做到这点，一个插件可以定义两个属性，首先是 dependsOn.让我们看看 Grails Hibernate 插件的代码片段：

```
class HibernateGrailsPlugin {
    def version = 1.0
    def dependsOn = [dataSource:1.0,
                     domainClass:1.0,
                     i18n:1.0,
                     core: 1.0]
}
```

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

根本上讲，这些被依赖的插件将先被加载，接着才是 Hibernate 插件，如果这些被依赖的插件没有加载，那么 Hibernate 也不会加载。

dependsOn 属性也支持一个小型的表达语言指定版本范围。以下是一些简单的语法例子：

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

当使用*通配符的时候，它表示"任何"版本。 The expression syntax also excludes any suffixes such as -BETA, -ALPHA etc. so for example the expression "1.0 > 1.1" would match any of the following versions:

- 1.1
- 1.0
- 1.0.1

- 1.0.3-SNAPSHOT
- 1.1-BETA2

Controlling Load Order

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
 }
```

这里，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 interface 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。

13. Web 服务

Web 服务就是让你的 web 应用提供一套 web API，通常用 SOAP 或 REST 来实现。 .

13.1 REST

REST 就本身而言不是一种技术，而是一种架构模式。is not really a technology in itself, but more an architectural pattern. REST 非常简单，以普通 XML 或 JSON 作为通信机制，结合可以表现底层系统状态的 URL 形式和 HTTP 方法如 GET, PUT, POST 和 DELETE.

每一个 HTTP 方法映射到一个 action，如用 GET 方法获取数据，用 PUT 方法创建数据，用 POST 更新数据等等。在这个意义上 REST 非常适合 CRUD.

URL 形式

要用 Grails 实现 REST,第一步就是提供 REST 形式的 URL 映射 URL 映射:

```
static mappings = {
  "/product/$id?"(resource:"product")
}
```

这就将 URI /product 映射到 ProductController. 在 controller 内部每个 HTTP 方法,GET,PUT,POST 和 DELETE 都映射到一个 action 上，如下表所示：

方法	Action
GET	show
PUT	update
POST	save
DELETE	delete

可以通过 URL 映射机制修改 HTTP 方法和 URL 的映射关系:

```
"/product/$id"(controller:"product"){
   action = [GET:"show", PUT:"update", DELETE:"delete", POST:"save"]
}
```

但是在这个例子中，Grails 并不像前面使用过的 resource 参数那样自动提供 XML 或 JSON 序列化，除非提供在 URL 映射中提供 parseRequest 参数:

```
"/product/$id"(controller:"product", parseRequest:true){
   action = [GET:"show", PUT:"update", DELETE:"delete", POST:"save"]
}
```

XML 序列化 - 读取

controller 可通过 Grails 提供的 XML 序列化机制 来实现 GET 方法:

```
import grails.converters.*
class ProductController {
    def show = {
        if(params.id && Product.exists(params.id)) {
            def p = Product.findByName(params.id)
            render p as XML
        }
        else {
            def all = Product.list()
```

```
                render all as XML
            }
        }
        ..
}
```

这里，如果参数中指定 id，通过 id 搜索 Product 如果指定 id 的 Product 存在，则返回该 Product，否则返回所有 Product. 这样，如果访问 /products 我们会得到所有的 Product，如果访问/product/MacBook，我们只获取到一个 MacBook 记录.

XML 序列化 - 更新

为支持 PUT 和 POST 你可以使用 params 对象。Grails 中 params 对象具有读取 XML 数据包的能力。如下面的 XML 数据包：

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<product>
      <name>MacBook</name>
      <vendor id="12">
            <name>Apple</name>
    </vender>
</product>
```

你可以通过在 数据绑定章节描述过的同样的方法，通过 params 对象来读取 XML 数据:

```
def save = {
      def p = new Product(params['product'])
   if(p.save()) {
            render p as XML
      }
      else {
            render p.errors
      }
}
```

在这个例子中，通过提取 params 对象中的 'product'对应的值，我们可以通过 Product 的构建器自动创建和绑定 XML 数据 。 注意这一行:

```
def p = new Product(params['product'])
```

这里我们不需要修改任何代码就可以以处理 XML 数据请求的方法处理表单提交。同样的方

法也可以用来处理 JSON 请求.

> 如果需要对不同的客户端（REST，HTML 等）提供不同的响应，你可以使用 content negotation

The Product object is then saved and rendered as XML, otherwise an error message is produced using Grails' validation capabilities in the form:

```
<error>
  <message>The property 'title' of class 'Person' must be specified</message>
</error>
```

13.2 SOAP

Grails 通过 XFire 插件来支持 SOAP。XFire 插件使用流行的 XFire SOAP 协议栈，它允许你通过特定的 expose 属性将 Grails 的 services 作为 SOAP 服务提供:

```
class BookService {
  static expose=['xfire']
  Book[] getBooks(){
    Book.list() as Book[]
  }
}
```

WSDL 文件可通过: http://127.0.0.1:8080/your_grails_app/services/book?wsdl 获取

更多信息参考 XFire 插件的 wiki 文档 。

13.3 RSS 和 Atom

Grails 没有直接提供对 RSS 和 Atom 的支持. You could construct RSS or ATOM feeds with the render method's XML capability. 可以通过 Grails Feeds 插件来构建 RSS 和 Atom。改插件使用流行的 ROME 库. 下面是简单使用这个插件的例子:

```
def feed = {
    render(feedType:"rss", feedVersion:"2.0") {
        title = "My test feed"
        link = "http://your.test.server/yourController/feed"
        Article.list().each() {
            entry(it.title) {
                link = "http://your.test.server/article/${it.id}"
```

```
            it.content // return the content
        }
    }
  }
}
```

14. Grails 和 Spring

这一节适合于高级用户， Spring 框架，和想通过 插件开发来配置 Grails 的开发人员。

14.1 Grails 内部实现

实际上 Grails 是变相的 Spring MVC 应用. Spring MVC 是 Spring 框架内置的 MVC web 开发框架.虽然从易用性来说 Spring MVC 比不上 Struts 这样的框架,但它的设计和架构都非常优秀，正适合在其基础之上构建另一个像 Grails 这样的框架。

Grails 在以下方面利用了 Spring MVC:

- 基本控制器逻辑 - Grails 继承 Spring 的 DispatcherServlet 并使用它作为代理将请求转发给 Grails 的控制器
- 数据绑定和校验 - Grails 的校验 和 数据绑定正是建立在 Spring 的数据绑定和校验之上
- 运行时配置 - Grails 的整个"约定优先配置"机制全部用 Spring 来实现 ApplicationContext
- 事务处理 - Grails GORM 使用 Spring 的事务处理

也就是说 Grails 内嵌 Spring 并在框架的各个环节上使用 Spring.

Grails ApplicationContext

Spring 开发人员经常热衷于想知道 Grails 中的 ApplicationContext 实例是怎么创建的.基本过程如下：

- Grails 通过 web-app/WEB-INF/applicationContext.xml 创建一个父 ApplicationContext 对象。 这个 ApplicationContext 对象设置 GrailsApplication 对象 and GrailsPluginManager 对象.
- 使用这个 ApplicationContext 作为父对象 Grails 通过“约定优先”分析 GrailsApplication 对象构建一个子 ApplicationContext 对象，此对象作为 web 应

用的 ApplicationContext

配置 Spring Beans

大部分 Grails 的配置都是在运行时进行. 每个 插件 都可以配置在上面创建的 ApplicationContext 对象中注册过的 Spring bean. For a reference as to which beans are configured refer to the reference guide which describes each of the Grails plug-ins and which beans they configure.

14.2 配置其他 Bean

使用 XML

Beans 可用过 grails-app/conf/spring/resources.xml 来配置. 这个文件是一个标准的 Spring 配置文件，在 Spring Spring 参考文档中对如何配置 Spring Beans 有详细描述。下面是一个简单的例子:

```
<bean id="myBean" class="my.company.MyBeanImpl"></bean>
```

配置完毕后, myBean 就可以在 Grails 控制器，标签库，服务等很多地方引用：

```
class ExampleController {
    def myBean
}
```

引用现有的 Spring bean

在 resources.xml 中声明的 bean 也可以通过约定来引用 Grails 类 . 比如， 如果你想在你的 bean 中引用 BookService 这样一个 service，你可以用如下的代码：

```
<bean id="myBean" class="my.company.MyBeanImpl">
    <property name="bookService" ref="bookService" />
</bean>
```

这个 bean 本身需要一个 public setter 方法，在 Groovy 中这样定义:

```
package my.company
class MyBeanImpl {
    BookService bookService
}
```

或在 Java 中:

```
package my.company;
class MyBeanImpl {
	private BookService bookService;
	public void setBookService(BookService theBookService) {
		this.bookService = theBookService;
	}
}
```

既然大部分 Grails 配置都是在运行时通过约定机制来完成，大部分 bean 并不需要声明, 但仍然可以在 Spring 应用中进行引用. 如你需要引用一个 Grails DataSource 你可以这样:

```
<bean id="myBean" class="my.company.MyBeanImpl">
	<property name="bookService" ref="bookService" />
	<property name="dataSource" ref="dataSource" />
</bean>
```

或者你需要引用 Hibernate SessionFactory:

```
<bean id="myBean" class="my.company.MyBeanImpl">
	<property name="bookService" ref="bookService" />
	<property name="sessionFactory" ref="sessionFactory" />
</bean>
```

所有提供的 bean 既说明可参考插件开发文档.

使用 Spring DSL

如果你想使用 Grails 提供的 Spring DSL ，你必须创建 grails-app/conf/spring/resources.groovy 文件，定义一个 beans 属性块:

```
beans = {
	// 定义的 beans
}
```

同样在的配置可以应用于 XML 例子:

```
beans = {
	myBean(my.company.MyBeanImpl) {
		bookService = ref("bookService")
	}
}
```

这样做最大的好处是你能够在 bean 的定义中混合各种逻辑，如基于 environment:

```
import grails.util.*
beans = {
    switch(GrailsUtil.environment) {
        case "production":
            myBean(my.company.MyBeanImpl) {
                bookService = ref("bookService")
            }
      break
        case "development":
            myBean(my.company.mock.MockImpl) {
                bookService = ref("bookService")
            }
        break
    }
}
```

14.3 运行时 Spring 与 Beans DSL

Grails 提供 BeanBuilder 的目的是提供一种简化的方法来关联使用 Spring 的各中依赖关系.

这是因为 Spring 的常规配置方法（通过 XML）在本质上是静态的，除了通过程序方式来动态产生 XML 配置文件外，很难在运行时修改和添加程序配置。而且这种方法非常繁琐，也容易出错. Grails 的 BeanBuilder 改变了这一点，它可以让你在运行时通过系统属性和环境属性来动态改变程序逻辑.

这使得程序代码动态适配它的环境，避免不必要的重复代码（如在 Spring 中为测试环境，开发环境和生产环境做不同的配置)

BeanBuilder 类

Grails 提供了 grails.spring.BeanBuilder 类使用动态 Groovy 来创建 bean 的声明. 基本点如下:

```
import org.apache.commons.dbcp.BasicDataSource
import org.codehaus.groovy.grails.orm.hibernate.ConfigurableLocalSessionFactoryBean;
import org.springframework.context.ApplicationContext;
def bb = new grails.spring.BeanBuilder()
bb.beans {
   dataSource(BasicDataSource) {
```

```
        driverClassName = "org.hsqldb.jdbcDriver"
        url = "jdbc:hsqldb:mem:grailsDB"
        username = "sa"
        password = ""
    }
    sessionFactory(ConfigurableLocalSessionFactoryBean) {
        dataSource = dataSource
        hibernateProperties = [ "hibernate.hbm2ddl.auto":"create-drop",
                                "hibernate.show_sql":true  ]
    }
}
ApplicationContext appContext = bb.createApplicationContext()
```

> 在 插件 和 grails-app/conf/spring/resources.groovy 文件中你不需要创建一个 BeanBuilder 实例， 它在 doWithSpring 和 beans 块中都隐式存在.

上面这个例子说明了如果使用 BeanBuilder 类来配置某个特定的 Hibernate 数据源。

实际上，每个方法调用(dataSource 和 sessionFactory 调用) 都映射到 Spring 中的 bean 的名字. 方法的第一个参数是 bean 的 class 名字, 最后一个参数是一个块（block）. 在块内部可以用标准的 Groovy 语法设置 bean 的属性。

通过 bean 的名字自动查找 bean 的引用. 通过上面的 sessionFactory bean 解析 dataSource 可以看点这一点。

也可以通过 builder 设置一些与 bean 管理相关的特殊的 bean 属性，如：

```
sessionFactory(ConfigurableLocalSessionFactoryBean) { bean ->
    bean.autowire = 'byName'       // Autowiring behaviour. The other option is 'byType'. [autowire]
    bean.initMethod = 'init'       // Sets the initialisation method to 'init'. [init-method]
    bean.destroyMethod = 'destroy' // Sets the destruction method to 'destroy'. [destroy-method]
    bean.scope = 'request'         // Sets the scope of the bean. [scope]
    dataSource = dataSource
    hibernateProperties = [ "hibernate.hbm2ddl.auto":"create-drop",
                            "hibernate.show_sql":true  ]
}
```

括号中的字符串对应于 Spring XML 定义中相应的 bean 属性名。

在 Spring MVC 中使用 BeanBuilder

如果想在 Spring MVC 中使用 BeanBuilder，你必须确保 grails-spring-<version>.jar 包

含在 classpath 中. 还要在/WEB-INF/web.xml 文件中做如下设置：

```
<context-param>
   <param-name>contextConfigLocation</param-name>
   <param-value>/WEB-INF/applicationContext.groovy</param-value>
</context-param>
<context-param>
   <param-name>contextClass</param-name>
   <param-
value>org.codehaus.groovy.grails.commons.spring.GrailsWebApplicationContext</param-value>
</context-param>
```

然后在创建 /WEB-INF/applicationContext.groovy 文件并配置如下:

```
beans {
      dataSource(org.apache.commons.dbcp.BasicDataSource) {
      driverClassName = "org.hsqldb.jdbcDriver"
      url = "jdbc:hsqldb:mem:grailsDB"
      username = "sa"
      password = ""
   }
}
```

从文件系统中加载 bean 定义

你可以使用 BeanBuilder 并使用下面的语法 来加载在外部 Groovy 脚本中定义的 bean：

```
def bb = new BeanBuilder()
bb.loadBeans("classpath:*SpringBeans.groovy")
def applicationContext = bb.createApplicationContext()
```

这里 BeanBuilder 将加载在 classpath 中以 SpringBeans.groovy 结尾的 Groovy 文件并将它们解析成 bean 的定义.这里是一个范例脚本文件：

```
beans {
   dataSource(BasicDataSource) {
      driverClassName = "org.hsqldb.jdbcDriver"
      url = "jdbc:hsqldb:mem:grailsDB"
      username = "sa"
      password = ""
   }
   sessionFactory(ConfigurableLocalSessionFactoryBean) {
```

```
        dataSource = dataSource
        hibernateProperties = [ "hibernate.hbm2ddl.auto":"create-drop",
                                "hibernate.show_sql":true  ]
    }
}
```

绑定变量

如果从脚本中加载 bean，可以通过创建 Groovy Binding 对象来实现绑定:

```
def binding = new Binding()
binding.foo = "bar"
def bb = new BeanBuilder()
bb.binding = binding
bb.loadBeans("classpath:*SpringBeans.groovy")
def ctx = bb.createApplicationContext()
```

14.4 BeanBuilder DSL

使用构建器参数

可以通过在 bean 的 class 和最后一个 closure 之间定义的方法来定义构建器参数:

```
bb.beans {
    exampleBean(MyExampleBean, "firstArgument", 2) {
        someProperty = [1,2,3]
    }
}
```

配置 BeanDefinition (使用工厂方法)

传给 closure 的第一个参数是一个 bean 配置对象引用,你可以使用它来配置工厂方法，调用 AbstractBeanDefinition 的方法:

```
bb.beans {
    exampleBean(MyExampleBean) { bean ->
        bean.factoryMethod = "getInstance"
        bean.singleton = false
        someProperty = [1,2,3]
    }
```

```
}
```

你也可以通过 bean 定义方法的返回值来配置 bean:

```
bb.beans {
   def example = exampleBean(MyExampleBean) {
      someProperty = [1,2,3]
   }
   example.factoryMethod = "getInstance"
}
```

使用工厂 bean（Factory beans）

Spring 提供了工厂 bean 的概念，即 bean 不是从 class 创建，而是由这些工厂创建 defines the concept of factory beans and often a bean is created not from a class, but from one of these factories. 在这种情况下 bean 没有 class，你必须将工厂 bean 的名字传给定义的 bean:

```
bb.beans {
   myFactory(ExampleFactoryBean) {
      someProperty = [1,2,3]
   }
   myBean(myFactory) {
      name = "blah"
   }
}
```

注意：上面的例子中我们传递的是 myFactory bean 而不是一个 clas. 另一个常见的需求是提供调用工厂 bean 的工厂方法名，可以用下面的 Groovy 语法做到这一点：

```
bb.beans {
   myFactory(ExampleFactoryBean) {
      someProperty = [1,2,3]
   }
   myBean(myFactory:"getInstance") {
      name = "blah"
   }
}
```

这里 ExampleFactoryBean 的 getInstance 会被调用来创建 myBean bean.

运行时创建 bean 的引用

有时只有在运行是才知道需要创建的 bean 的名字. 在这种情况情况下你可以使用字符串替换来实现动态调用:

```
def beanName = "example"
bb.beans {
   "${beanName}Bean"(MyExampleBean) {
      someProperty = [1,2,3]
   }
}
```

在这个例子中，使用早先定义的 beanName 变量来调用 bean.

另外, 可使用 ref 来动态引用在运行时才知道的 bean 的名字，如下面的代码：

```
def beanName = "example"
bb.beans {
   "${beanName}Bean"(MyExampleBean) {
      someProperty = [1,2,3]
   }
   anotherBean(AnotherBean) {
      example = ref("${beanName}Bean")
   }
}
```

这里 AnotherBean 属性通过运行时对 exampleBean 的引用来设置 . 也可以通过 ref 来引用在父 ApplicationContext 定义的 bean， ApplicationContext 在 BeanBuilder 的构建器中提供:

```
ApplicationContext parent = ...//
der bb = new BeanBuilder(parent)
bb.beans {
   anotherBean(AnotherBean) {
      example = ref("${beanName}Bean", true)
   }
}
```

这里第二个参数 true 指定了在父 ApplicationContext 中查找 bean 的引用.

使用匿名内部 bean

你可以通过将属性块付给 bean 的一个属性来使用匿名内部 bean，这个属性块提供一个

bean 的类型参数:

```
bb.beans {
    marge(Person.class) {
        name = "marge"
        husband = { Person p ->
            name = "homer"
            age = 45
            props = [overweight:true, height:"1.8m"]
        }
        children = [bart, lisa]
    }
    bart(Person) {
        name = "Bart"
        age = 11
    }
    lisa(Person) {
        name = "Lisa"
        age = 9
    }
}
```

在上面的例子中我们将 marge bean 的 husband 属性 赋值一个属性块（参数类型是 Person）的方式创建一个内部 bean 引用. 如果你有一个工厂 bean 你也可以忽略类型参数，直接使用传进进来的 bean 的定义：

```
bb.beans {
    personFactory(PersonFactory.class)
    marge(Person.class) {
        name = "marge"
        husband = { bean ->
            bean.factoryBean = "personFactory"
            bean.factoryMethod = "newInstance"
            name = "homer"
            age = 45
            props = [overweight:true, height:"1.8m"]
        }
        children = [bart, lisa]
    }
}
```

抽象 bean 和父子 bean 定义

要创建一个抽象 bean，定义一个没有 class 的 bean：

```
class HolyGrailQuest {
    def start() { println "lets begin" }
}
class KnightOfTheRoundTable {
    String name
    String leader
    KnightOfTheRoundTable(String n) {
        this.name = n
    }
    HolyGrailQuest quest
    def embarkOnQuest() {
        quest.start()
    }
}
def bb = new grails.spring.BeanBuilder()
bb.beans {
    abstractBean {
        leader = "Lancelot"
    }
    ...
}
```

这里定义了一个抽象 bean，这个 bean 有一个属性 leader，属性值为 "Lancelot". 要使用抽象 bean，只要将它设为要定义的 bean 的父即可：

```
bb.beans {
    ...
    quest(HolyGrailQuest)
    knights(KnightOfTheRoundTable, "Camelot") { bean ->
        bean.parent = abstractBean
        quest = quest
    }
}
```

当使用父 bean 时，你必须在设置其他属性前设置 parent 属性!

如果你要定义一个具有 class 的抽象 bean，可以这样:

```
def bb = new grails.spring.BeanBuilder()
bb.beans {
   abstractBean(KnightOfTheRoundTable) { bean ->
      bean.'abstract' = true
      leader = "Lancelot"
   }
   quest(HolyGrailQuest)
   knights("Camelot") { bean ->
      bean.parent = abstractBean
      quest = quest
   }
}
```

上面例子中我们创建了抽象 KnightOfTheRoundTable 并将它的参数设为 abstract. 接下来我们定义了一个 knights bean，没有定义它的 class，而是继承父 bean 中定义的 class。

使用 Spring 命名空间

从 Spring 2.0 开始，通过 XML 命名空间可以更方便的使用 Spring 的各种特性. 如果使用 BeanBuilder， 你可以先声明所要使用的 Spring 命名空间:

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

然后调用与命名空间名称和属性匹配的方法:

```
context.'component-scan'( 'base-package' :"my.company.domain" )
```

通过 Spring 的命名空间可以做很多有用的事，比如查找 JNDI 资源:

```
xmlns jee:"http://www.springframework.org/schema/jee"
jee.'jndi-lookup'(id:"dataSource", 'jndi-name':"java:comp/env/myDataSource")
```

上面的例子通过查找 JNDI 创建一个 dataSourcebean 对象. 通过 Spring 命名空间，你可以在 BeanBuilder 中直接访问 Spring AOP 功能比如下面的代码:

```
class Person {
 int age;
 String name;
 void birthday() {
     ++age;
 }
}
```

```
class BirthdayCardSender {
   List peopleSentCards = []
   public void onBirthday(Person person) {
      peopleSentCards << person
   }
}
```

你可以定义一个 AOP aspect pointcut 来监测对 birthday() 方法的所有调用:

```
xmlns aop:"http://www.springframework.org/schema/aop"
fred(Person) {
 name = "Fred"
 age = 45
}
birthdayCardSenderAspect(BirthdayCardSender)
aop {
   config("proxy-target-class":true) {
      aspect( id:"sendBirthdayCard",ref:"birthdayCardSenderAspect" ) {
         after method:"onBirthday", pointcut: "execution(void ..Person.birthday()) and this(person)"
      }
   }
}
```

14.5 属性占位符配置

通过扩展的 Spring 的 PropertyPlaceholderConfigurer，Grails 支持属性占位符配置，这和 外部配置配合使用非常有用。 .

Settings defined in either ConfigSlurper scripts of Java properties files can be used as placeholder values for Spring configuration in grails-app/conf/spring/resources.xml. For example given the following entries in grails-app/conf/Config.groovy (or an externalized config):

```
database.driver="com.mysql.jdbc.Driver"
database.dbname="mysql:mydb"
```

接着在 resources.xml 中用${..}语法定义占位符：

```
<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
   <property name="driverClassName"><value>${database.driver}</value></property>
   <property name="url"><value>jdbc:${database.dbname}</value></property>
```

</bean>

14.6 属性重载

通过扩展的 Spring PropertyOverrideConfigurer，Grails 提供了对属性重载配置的支持，外部配置配合使用非常有用。 .

你可以提供一个 ConfigSlurper 脚本文件，该文件中定义了一个 beans 属性块，属性块中定义的属性值会覆盖 bean 中定义的属性值:

```
beans {
  bookService.webServiceURL = "http://www.amazon.com"
}
```

重载的属性应用在 Spring ApplicationContext 创建之前. 格式如下:

```
[bean name].[property name] = [value]
```

你也可以提供一个常规的 Java 属性文件，属性文件中的每个条目加上 beans 前缀:

```
beans.bookService.webServiceURL=http://www.amazon.com
```

15. Grails 与 Hibernate

如果 GORM (Grails Object Relational Mapping)没有你想象的那么足够灵活,作为选择,你可以使用 Hibernate 映射你的 domain 类. 要做到这点,需要在你项目的 grails-app/conf/hibernate 目录创建一个 hibernate.cfg.xml 文件并为你的 domain 类对应 HBM 映射文件 .

> 更多关于这方面的信息，请查看 Hibernate 站点的文件映射

这允许你映射 Grails domain 类适用于更广的遗留系统并更加灵活的创建数据库模式 .

Grails 也允许你在 Java 中编写 domain 类或重用以存在的 domain model,这些都通过使用 Hibernate 来映射 . 你需要做的是放置必须的 hibernate.cfg.xml 文件和对应的映射文件在 grails-app/conf/hibernate 目录中 .

另外,令人兴奋的是你仍然可以调用 GORM 中所有动态之久和查询方法！

15.1 通过 Hibernate 注解映射

Grails 也支持通过 Hibernate 的 Java 5.0 注解支持来创建 domain 类映射. 为了做到这点,你需要通过设置 DataSource 中的 configClass 属性告诉 Grails 你要使用注解配置,如下 :

```
import org.codehaus.groovy.grails.orm.hibernate.cfg.GrailsAnnotationConfiguration
dataSource {
      configClass = GrailsAnnotationConfiguration.class
  ... // remaining properties
}
```

这就是它的配置！确保你安装了 Java 5.0,因为这需要使用注解. 现在,为了创建一个注解类,我们在 src/java 中简单的创建一个新的 Java 类并使用 EJB 3.0 规范来定义注解(详情参考 Hibernate Annotations Docs):

```
package com.books;
@Entity
public class Book {
   private Long id;
   private String title;
   private String description;
   private Date date;
   @Id
   @GeneratedValue
   public Long getId() {
      return id;
   }

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

   public String getTitle() {
      return title;
   }

   public void setTitle(String title) {
      this.title = title;
   }

   public String getDescription() {
```

```
        return description;
    }

    public void setDescription(String description) {
        this.description = description;
    }
}
```

一旦完成,你需要使用 Hibernate sessionFactory 注册这个类 ,为了做到这点,你需要添加 如下的 grails-app/conf/hibernate/hibernate.cfg.xml 文件:

```
<!DOCTYPE hibernate-configuration SYSTEM
 "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <mapping package="com.books" />
        <mapping class="com.books.Book" />
    </session-factory>
</hibernate-configuration>
```

当 Grails 加载时，会注册这个类必要的动态方法 . 查看 Scaffolding 了解 Hibernate domain 类中可以做的其他事情.

15.2 进一步阅读

Grails 提交者, Jason Rudolph,花了许多时间写了许多关于通过自定义 Hibernate 使用 Grails:

- Hoisting Grails to Your Legacy DB - An excellent article about using Grails with Hibernate XML
- Grails + EJB3 Domain Models - Another great article about using Grails with EJB3-style annotated domain models

16. 脚手架

根据指定的领域类，脚手架为你自动生成一个领域相关的完整应用，包括:

- 必要的 views
- 控制器的创建/读取/更新/删除（CRUD）操作

启动脚手架

让脚手架生效的最简单方法是通过设置 scaffold 属性。 以领域类 Book 为例,你需要在其控制器中设置 scaffold 属性为 true 就可以了，代码如下:

```
class BookController {
  def scaffold = true
}
```

上述代码可以正常工作是因为控制器 BookController 命名跟领域类 Book 相一致， 如果我们想脚手架对特定的领域类使用，你可以直接将特定的领域类赋值给 scaffold 属性，代码如下:

```
def scaffold = Author
```

设置完毕后，如果你运行 grails 应用，那么那些必要的动作和视图都将在运行期间自动生成。 根据脚手架的动态机制，以下一些动作将被动态实现：

- list
- show
- edit
- delete
- create
- save
- update

即基本的 CRUD 接口将被自动生成。 为了访问以上示例生成的接口，只需要去 http://localhost:8080/app/book

如果你倾向于使用 基于 Hibernate 映射 的 Java 领域模型，你依然可以使用脚手架， 只需简单的导入必要的类，并且将此类赋值给 scaffold 属性即可。

动态脚手架

注意当使用 scaffold 属性的时候，Grails 并不是通过代码模板或者代码生成来实现脚手架功能，因此你照样可以在被脚手架过的控制器中增加自己的动作，来跟脚手架过的动作进行交互。比如，在下面的示例中， changeAuthor 可以重新定向到一个并不存在的 show 的动作:

```
class BookController {
  def scaffold = Book
```

```
  def changeAuthor = {
      def b = Book.get( params["id"] )
      b.author = Author.get( params["author.id"] )
      b.save()



      // redirect to a scaffolded action
      redirect(action:show)
  }
}
```

当然必要的时候，你也可以使用自己的动作来重写被脚手架过的动作，代码如下:

```
class BookController {
  def scaffold = Book
  // overrides scaffolded action to return both authors and books
  def list = {
       [ "books" : Book.list(), "authors": Author.list() ]
  }
}
```

所有这些就是所谓的“动态脚手架”，在这里 CRUD 接口将在运行期间动态生成。不过 Grails 同样也支持所谓的“静态”脚手架，这将在接下来的章节中讨论。

自定义生成的视图

Grails 生成的视图中，有些表单能智能地适应 验证约束. 如下面代码所示，只需要简单地重新排列生成器（builder）中约束的顺序，就可以改变其在视图中出现的顺序:

```
def constraints = {
    title()
    releaseDate()
}
```

你也可以通过使用 inList 约束来生成一个列表（list）而不是简单的文本输入框（text input）：

```
def constraints = {
    title()
    category(inList:["Fiction", "Non-fiction", "Biography"])
    releaseDate()
}
```

或者通过基于数字的 range 约束来生成列表：

```
def constraints = {
     age(range:18..65)
}
```

通过约束来限制大小（size）也可以影响生成的视图中可以输入的字符数:

```
def constraints = {
     name(size:0..30)
}
```

生成控制器和视图

以上的脚手架特性虽然很有用，但是在现实世界中有可能需要自定义逻辑和视图。Grails 允许你通过使用命令行的方式，来生成一个控制器和相关视图（跟脚手架所做的事情差不多）。为了生成控制器，只需要输入:

```
grails generate-controller Book
```

或者为了生成视图，只需输入:

```
grails generate-views Book
```

或者生成控制器和视图，只需输入:

```
grails generate-all Book
```

如果你的领域类有包名或者从 Hibernate 映射的类 来生成，那需要记住一定要用类的全名（包名+类名），如下：

```
grails generate-all com.bookstore.Book
```

定制脚手架模板

使用的 Grails 自动生成的控制器和试图模板可以自己定制安装模板通过 install-templates 这个命令.

17. 部署

Grails 可以使用很多种方式来部署,每一种都有它的缺点和优点.

"grails run-app"

现在,你已经非常属性这个方式，因为它是在部署阶段运行应用程序非常普通的方法. 内置的 Jetty 服务器被启动并加载来自开发时的应用程序源代码, 因此，允许获取应用程序文件的改变.

这种方式在产品部署时不被推荐,因为性能非常差。 检查和加载改变在服务器端是非常大的开销. 话虽如此, grails prod run-app 移除每次请求开支并允许你控制进行定期检查的频率.

设置系统属性"disable.auto.recompile" 为 true 彻底禁止常规检查 ,属性"recompile.frequency"控制着频率 .后者应该设置为你想要每次检查之间的秒数 . 默认为 3.

"grails run-war"

这非常类似于上面的选项,但 Jetty 运行依靠的是打包的 WAR 文件而不是开发时源代码. 热重载被禁止, 因此你无需在别处部署 WAR 文件而获得良好性能.

WAR 文件

当涉及到它时, 目前的 java 基本设备都要求 web 应用程序被当做 WAR 文件部署,因此,这是目前为止最常见的方式 Grails 应用程序用于生产部署 . 创建 WAR 文件只需要简单的执行 war 命令:

```
grails war
```

这里也有许多方式用于定制 WAR 文件的创建 . 例如, 你可以指定命令路径 (任何现对和绝对) ,这会指定在哪里放置文件和给定什么样的名字:

```
grails war /opt/java/tomcat-5.5.24/foobar.war
```

作为选择,你可以在 grails-app/conf/BuildConfig.groovy 添加一行来改变默认的位置和文件名：

```
grails.war.destFile = "foobar-prod.war"
```

当然,任何命令行参数都优先于这个设置.

它也可以控制在 WAR 文件包含什么样的类库，例如,如果你需要在共享文件夹中避免类库冲突.默认行为是包含所有 Grails 所需要的全部类库 ,添加的任何类库都被包含在 "lib"目录, 添加的任何类库都被包含在应用程序的"lib"目录.作为默认行为的选择,通过使用 Ant 包含模式或闭包包含 AntBuilder 语法的任一种设置 Config.groovy 的 grails.war.dependencies 和 grails.war.java5.dependencies 属性来明确指定 WAR 文件所包含的完整的类库列表,闭包的调用来自 Ant "copy"阶段，因此只有像 "fileset"的元素可以被包含,尽管每个项目都包含在模式列表中.任何闭包或模式被分配给后面的属性被包含在增加的 grails.war.dependencies 只在你运行在 JDK1.5 或以上 .

注意这些问题: 假如任何 Grails 依赖的类库丢失,应用程序肯定会失败,这里有个示例包含了标准 Grails 依赖所需的小子集:

```
def deps = [
   "hibernate3.jar",
   "groovy-all-*.jar",
   "standard-${servletVersion}.jar",
   "jstl-${servletVersion}.jar",
   "oscache-*.jar",
   "commons-logging-*.jar",
   "sitemesh-*.jar",
   "spring-*.jar",
   "log4j-*.jar",
   "ognl-*.jar",
   "commons-*.jar",
   "xstream-1.2.1.jar",
   "xpp3_min-1.1.3.4.O.jar" ]
grails.war.dependencies = {
   fileset(dir: "libs") {
      deps.each { pattern ->
         include(name: pattern)
      }
   }
}
```

这个示例只是为了说明属性的语法，假如你想在自己的应用程序中尝试使用它们,应用程序可能不会工作. 你可以在未打包的根目录的 "dependencies.txt"文件中找到 Grails 所需的依赖列表.你也可以在产生 WAR 文件的"War.groovy"脚本中找到默认的依赖 - 查看 "DEFAULT_DEPS" 和 "DEFAULT_J5_DEPS" 变量.

2 个遗留的配置选项用于 grails.war.copyToWebApp 和 grails.war.resources. 第一个允许你定制来自"web-app"目录的 WAR 文件包含什么. . 第 2 个允许你在 WAR 文件完全创建之前执行任何额外的数据处理.

```
// This closure is passed the command line arguments used to start the
// war process.
grails.war.copyToWebApp = { args ->
   fileset(dir:"web-app") {
      include(name: "js/**")
      include(name: "css/**")
      include(name: "WEB-INF/**")
   }
}
// This closure is passed the location of the staging directory that
// is zipped up to make the WAR file, and the command line arguments.
// Here we override the standard web.xml with our own.
grails.war.resources = { stagingDir, args ->
   copy(file: "grails-app/conf/custom-web.xml", tofile: "${stagingDir}/WEB-INF/web.xml")
}
```

应用程序服务器

理想情况下，你可以把通过 Grails 创建的 WAR 文件简单的放置于任何应用程序服务器并能马上工作. 不过,事情并没这么简单. Grails 站点 包含最新的 Grails 测试过的服务器列表, 连同任何其他用让 Grails WAR 文件工作的额外步骤.