Grails 开源框架 - 使用指南

作者：Graeme Rocher, Marc Palmer

版本：1.0

目 录

1. 简介

如今的 Java Web 开发对于需求来说已经变得过于复杂。当今众多 Java 领域的 Web 开发框架不仅使用复杂，而且并没有很好的遵循 Don’t Repeat Yourself（DRY）原则。

像 Rails，Django 和 TurboGears 这样的动态框架在 Web 开发领域开辟了一条新的道路，Grails 基于这些概念之上，采用动态方法减小了 Java 平台上进行 Web 开发的复杂度,不过与那些框架不同的 是,Grails 是构建在 Spring 和 Hibernate 等 Java 已有的技术之上的。

Grails 是一个 full-stack 框架，它借助于核心技术与相关的插件（plug-in）来解决 Web 开发中方方面面的问题，其中包括：

- 易于使用的基于 Hibernate 的 对象-关系映射(ORM)层
- 称为 Groovy Server Pages (GSP)的表现层技术
- 基于 Spring MVC 的控制器层
- 构建于 Gant 上的命令行脚本运行环境
- 内置 Jetty 服务器，不用重新启动服务器就可以进行重新加载
- 利用内置的 Spring 容器实现依赖注入
- 基于 Spring 的 MessageSource 核心概念，提供了对国际化（i18n）的支持
- 基于 Spring 事务抽象概念，实现事务服务层

借助于功能强大的 Groovy 动态语言和领域特定语言（Domain Specific Language，DSL），以上那些特性变得非常易用。

这篇文档会向你介绍如何使用 Grails 框 架来搭建 Web 应用程序。

2. 起步

2.1 下载并安装 Grails

首先需要下载 Grails 的发行包并进行安装，执行步骤如下：

- 下载 Grails 二进制发行包并解压到指定的文件目录下。
- 在环境变量中添加 GRAILS_HOME,值为上一步解压的文件目录。
  - Unix/Linux 系统上运行 export GRAILS_HOME=/path/to/grails。
  - Windows 系统上右击“我的电脑”/“属性”/“高级”/“环境变量”，点击新建。
- 将解压目录下的 bin 目录路径添加到 path 中
  - Unix/Linux 系统上运行 export PATH="$PATH:$GRAILS_HOME/bin。
  - Windows 系统上右击“我的电脑”/“属性”/“高级”/“环境变量”，修改 path 的值。

如果环境变量设置无误，此时可以打开终端（window 下为命令提示符，Unix/Linux 下为 Shell），输入 grails，如果屏幕上显示如下提示则说明安装成功。

```
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 创建一个 Grails 应用

在创建应用程序之前，先熟 悉一下 grails 命令的使用（grails 中的命令都在终端中输入，请参考上面的讲解）。

```
grails command name
```

现在我们为了创建一个 Grails 应用， 需要输入的命令是 create-app

```
grails create-app helloworld
```

这样就在当前目录下创 建了一个名为 helloworld（即我们的应用程序名）的文件夹，在这个文件夹中包含了我们这个项目的整个文件目录，可以使用如下命令进入这个目录中查 看：

```
cd helloworld
```

2.3 Hello World 示例

为了完成这个经典的 Hello World 示例，我们需要运行 create-controller 命令

```
grails create-controller hello
```

运行该命令后 会在 grails-app/controller 目录下创建一个名为 HelloController.groovy 的 控制器（更多关于控制器的内容请参考控制器一节）

控制器主要用来完成对 Web 请求的处理，我们稍 微修改一下控制器的内容，使它能够在页面上输出"Hello World!"的字样，代码如下:

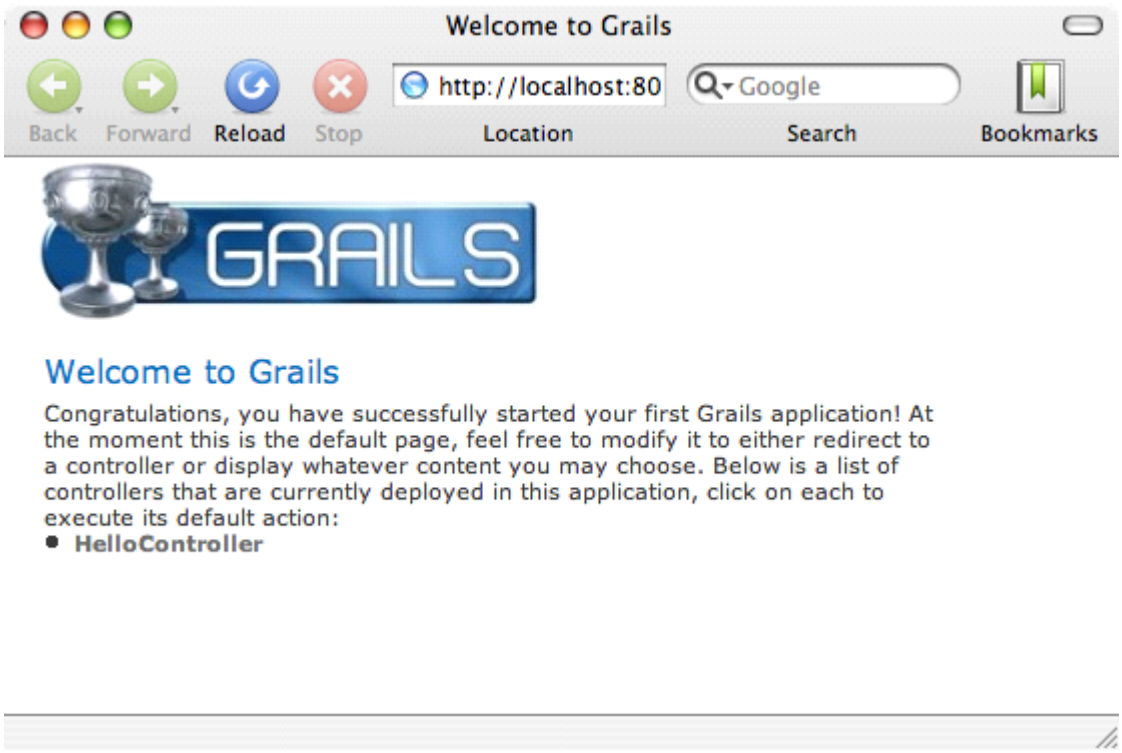
```
class HelloController {
        def world = {
                render "Hello World!"
        }
}
```

现在控制器已经完成 了，接下来要使用 run-app 来启动内置的 jetty 服务器运行刚刚创建的 helloworld 程序

```
grails run-app
```

运行后会在 8080 端口（默认，可以使用 -Dserver.port 来指定端口）启动服务器，然后在浏览器中输入 http://localhost:8080/helloworld 来启动应用程序.

运行结果如下图所示：



这个 Grails 简介页面是由 web-app/index.gsp 来显示的。从上图可以看见刚才创建的控制器，点 击链接之后会在浏览器中显示 Hello World!的字样。

2.4 使用 IDE

IntelliJ IDEA

至今为止最成熟、最全面的 Groovy&Grails 开发集成工具就是 IntelliJ IDEA 7.0 和 它的 JetGroovy 插件。对于大型项目，Grails 团队优先推荐使用 IntelliJ。

TextMate

由于 Grails 目标集中 于如何更简洁，所以我们可以使用一些更简单的编辑器进行 Groovy&Grails 的开发，例如在 Mac 上可以使用 TextMate ，下载地址 Texmate bundles SVN

Eclipse

Eclipse 的 Groovy 插件 也可以支持语法高亮，代码自动完成等特性。

在 Grails 的 Wiki 上有更多关于 Eclipse 插件的谈论。

Grails 在创建应用时会自动创建 Eclipse 的工程文件.project 和 classpath，这样如果要在 Eclipse 中导入一个 Grails 工程只需要在 Eclipse 中右击 Package

Explorer，选择 Import，然后选择 Existing project into Workspace，最后指定 Grails 的项目目录位置。

点击 OK，Finish 后会自动将 Grails 工程导入到 Eclipse 中。 并自动创建合适的运行配置（Run Configuration），这样在 Eclipse 中直接点击 Run 就可以运行 Grails 的项目。

2.5 规约配置

Grails 中的配置遵循“规约优于配置”的原则，即通过文 件的名称和位置来替代显式的配置，因此需要熟悉以下几个目录结构的用途。

此处仅仅为一个分类，具体的请参考相 关章节：

- grails-app - Groovy 源文件的顶级目录
  - conf - 配置文件目录,详细参考配置.
  - controllers - 控制器目录（MVC 模型中的 C），详细参考控制器
  - domain - 领域模型目录（MVC 模型中的 M），详细参考 GORM
  - i18n - 国际化目录，用来支持 i18n，详细参考国际化
  - services - 服务目录，详细参考服务层
  - taglib - 标签库目录，详细参考标签库
  - views - 视图 GSP 目录（MVC 中的 V），详细参考 GSP
- scripts - Gant 脚本目录,详细参考 Gant 脚本
- src - 源文件目录
  - groovy - 其他的 Groovy 源文件目录
  - java - 其他的 Java 源文件目录
- test - 单元和集成测试目录,详细参考测试

2.6 运行 Grails 应用

可以运行 run-app 命令来启动 Grails 内置的 Jetty 服务器,默认启动端口为 8080
grails run-app

如果 8080 端口已经被占用，可以通过 server.port 参数指定其他端口

grails -Dserver.port=8090 run-app

更多关于 run-app 的内容可以参照参考指南。

2.7 测试 Grails 应用

create-* 命令会在 test/integration 目录下自动创建相应的测试文件，可以在这些测试文件中编写测试用例来进行单元测试和集成测试，关于测试的内容可以参考测试章节。运行这些测试可以使用 test-app 命令。

```
grails test-app
```

Grails 会自动创建 Ant 的 build.xml 文件，也可以通过 Ant 来运行测试(其实运行的是 Grails 的 test-app)。

```
ant test
```

这样当 Grails 应用作为持续集成平台，例如 CruiseControl 的一部分时就会十分方便

2.8 部署 Grails 应用

Grails 应用程序以 Web 应用 归档（.WAR）文件的形式进行部署，并且 Grails 还提供了 war 命令来执行生成归档文件。

```
grails war
```

这条命令会生成当前应用的 war 文件，可以按照服务器容器的不同进行相应的配置.

一定不要使用 run-app 命令来部署 Grails，因为此命令会在运行时自动加载，这样会对服务器的性能和可扩 展性有严重影响。

当部署时应当通 JVM 的-server 参数来为服务器 分配足够的内存空间，推荐的 VM 参数是：

```
-server -Xmx512M
```

2.9 所支持的 Java EE 容器

Grails 可以支持的相当多 的 Web 容器，包括:

- 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

虽然在一些服务器上运行还存在 Bug，但是大部分情况下都可以工作的很好。在 Grails 的 Wiki 上可以找到关于已知部署问题的 清单。

2.10 创建工件

除了以上介绍的命令，Grails 还有一系列用来创建其 他工件类型的命令，例如可以使用 create-controller,create-domain-class 等等。
为了方便可以使用 IDE 或者其他你喜欢的编辑器来创建
比如说，为了创建一个应用的基础，你至少需要一个领域模型，命令如下：

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

这样会在 grails-app/domain/Book.groovy 中创建一个领域 类，其内容如下：

```
class Book {
}
```

更多的 create-*命令可以在参考指南的命令行中 被探索。

2.11 生成 Grails 应用

在创建完 Grails 应用后通 常会使用“脚手架”来生成整个应用程序的骨架。这是通过使用 generate-*命令来完成的， 例如使用 generate-all 命令来根据领域模型生成控制器及其相应视图。由于之前我们创建了一个 Book.groovy 的领域模型，因此在这里运行如下命令。

```
grails generate-all Book
```

更多关于脚手架的内容请参考用户指南的后边章节。

3. 配置

也许在这里谈论配置对于一个遵循“规约优于配置”的框架来说，会让人感到比较奇怪，但是实际上我们这里所说的配置是两个不同的概念，请不要混淆。

实际上 Grails 的默认配置已经足 以我们进行开发，并且它内置了容器和内存模式的 HSQL 数据库，这样我们几乎连数据库都不用配置了。

不 过，在将来你肯定是想要配置一个真正的数据库的，下面的章节将介绍如何实现。

3.1 基本配置

Grails 提供了一个 grails-app/conf/Config.groovy 配 置文件，用来完成一般通常的配置。这个文件使用非常类似于 Java 属性（propertiese）文件、不过是纯 Groovy 的 ConfigSlurper ，这样就可以重用（re-use）变量以及指定合适的 Java 类型。

例如，可以在文件中加上自己定义的配置 信息。

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

这样在应用程序中有两种方式可以访问这些配置信息。最常用的一种方式是通过 GrailsApplication 对象，它在控制器和标签库中都可以作为变量来使用。

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

3.1.1 内置选项

Grails 同样提供了如下配置选项：

- grails.config.locations - 属性 (properties) 文件的位置或者 Grails 将同主配置合并的额外配置文件。
- grails.enable.native2ascii-如果不需要对 Grails 的 i18n 配置文件使用默认的 native2ascii 约定，可以将该选项设为 false。
- grails.views.default.codec - 设置 GSP 缺省的字符编码，可以是：'none', 'html', 或者 'base64' (缺省为'none'). 为了减少 XSS 攻击的风险，建议设置成'html'.
- grails.views.gsp.encoding - GSP 源文件的字符编码（缺省是'utf-8'）
- grails.war.destFile - 设置 war 生成 WAR 文件的位置
- grails.mime.file.extensions - 是否使用文件的扩展名表示内容协商中的媒体类型(mime type)
- grails.mime.types - 内容协商所支持的媒体类型
- grails.serverURL - 一个指向服务器 URL 的绝对连接，包括服务器名称比如 grails.serverURL="http://my.yourportal.com". 详细请看 createLink.

3.1.2 日志

日 志基础

Grails 通过自身的配置机制来配置 Log4j 日志系统。要使用日志记录功能，需要修改位于 grails-app/conf 目录下的 Config.groovy 文 件。在这个文件中可

以指定不同环境，例如 development,test 和 production 环境下的日志方式。Grails 会自动处理该文件并在 web-app/WEB-INF/classes 目 录下生成相应的 log4j.properties 文件。

例如，在 Grails 中可以按照如下方式配置 Log4j：

```
log4j {
    appender.stdout = "org.apache.log4j.ConsoleAppender"
        appender.'stdout.layout'="org.apache.log4j.PatternLayout"
    rootLogger="error,stdout"
    logger {
        grails="info,stdouti"
        org {
            grails.spring="info,stdout"
            codehaus.groovy.grails.web="info,stdout"
            codehaus.groovy.grails.commons="info,stdout"
            ...
        }
}
```

如果想使用 Log4j 文件的标准风格进行配置，可以使用 Groovy 中的多行字符 串，例如：

```
log4j = '''
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
# ...remaining configuration
'''
```

其他一些有用的日志记录器包括：

- org.codehaus.groovy.grails.commons - 用来记录核心工件的信息，例如类的加载等等。
- org.codehaus.groovy.grails.web - 用来记录 Grails web 请求处理
- org.codehaus.groovy.grails.web.mapping - 用来调试 URL 映射
- org.codehaus.groovy.grails.plugins - 用来记录插件的活动状况
- org.springframework - 用来记录 Spring 的活动
- org.hibernate - 用来记录 Hibernate 的活动

完整的信息跟踪

当异常发生 的时候，可能由于 Java 和 Groovy 内部的错误而产生一大堆的无用信息。Grails 对这些不需要的信息进行了过滤，保证对非 Grails/Groovy 核心类包的信息跟踪。

通常这些信息都会通过StackTrace日志记录 器记录到stacktrace.log文件中。不过通过修改Config.groovy可以改变其记录方式，例如如果要将跟踪信息直接输出到标准输出 （standard out），需要将如下配置：

StackTrace="error,errors"

修改为：

StackTrace="error,stdout"

也可以通过设置grails.full.stacktrace虚拟机属性为true来 禁用对跟踪信息的过滤，例如

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

按照规约的日志记录

所有的应用程序工件都 动态地增加了log属性，其中包括领域模型（Domain Class），控制器以及标签库等等。使用方法如下所示：

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

在Grails中使用grails.app.<artefactType>.ClassName的 命名规约来命名日志记录器。例如

```
# Set level for all application artefacts
log4j.logger.grails.app="info, stdout"

# Set for a specific controller

log4j.logger.grails.app.controller.YourController="debug, stdout"

# Set for a specific domain class

log4j.logger.grails.app.domain.Book="debug, stdout"

# Set for a specific taglib

log4j.logger.grails.app.tagLib.FancyAjax="debug, stdout"

# Set for all taglibs

log4j.logger.grails.app.tagLib="info, stdout"
```

其中的工件名（即<artefactType>）也是按照规约来命名的，一些常见的如下所示:

- bootstrap – 用于启动时的加载类
- dataSource – 用于数据源
- tagLib – 用于标签库
- service – 用于服务类
- controller – 用于控制器
- domain – 用于领域模型类

3.2 环境

各个 环境的配置

Grails 支持针对不同的环境进行不同的配置，这样不管是 Config.groovy 文 件还是 grails-app/conf 目录下的 DataSource.groovy 文件，都可以借 助于 ConfigSlurper 来随时改变所处的环境。例如，默认 Grails 中的 DataSource 定义如下：

```
dataSource {
    pooling = false
    driverClassName = "org.hsqldb.jdbcDriver"
    username = "sa"
    password = ""
}
environments {
    development {
        dataSource {
            dbCreate = "create-drop" // one of 'create',
'createeate-drop','update'
            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，在文件的最开始提供了一个通用的配置，然后在 environments 代 码块中针对不同的环境下的 DataSource，配置了对应的 dbCreate 和 url 属 性。在 Config.groovy 中也可以使用这样的语法。

打包以及切换环境

grails 的命令行中可以指定运行时的环境，形式如下：

```
grails [environment] [command name]
```

其中的 environment 参数分别用 dev,prod,test 指 代开发环境（development），产品环境（production）以及测试环境（test），例如我们想在测试环境中创建 war 文件，可以执行如 下命令：

```
grails test war
```

如果自定义了其他的环境，可以在命令行中通过 grails.env 参数指定环境，例如:

```
grails -Dgrails.env=UAT run-app
```

在程序中检测环境

在 Gant 脚本或者启动 类中，可以通过 GrailsUtil 类来判断运行的环境，例如：

```
import grails.util.GrailsUtil

...

switch(GrailsUtil.environment) {
        case "development":
            configureForDevelopment()
        break
        case "production":
            configureForProduction()
        break
```

}

3.3 数据源

由于 Grails 是基于 Java 之上的，所以需要读者有一定的关于 JDBC 的知识。

首先，如果要使用 HSQL 之外其它的数据库，需要有相应的 JDBC Driver（驱动），例如对于 MySQL 需要有 Connector/J

这些数据库驱动一般都是 JAR 的形式 发布，只需要把这些 JAR 文件放到工程下的 lib 目录下即可。

在放置完 JDBC 驱动后，我们需要熟悉以下位于 grails-app/conf/下的 Grails 数据库配置文件 DataSoruce.groovy。 初始状态下该文件会包括如下选项:

- driverClassName - JDBC 驱动（driver）的类名
- username - 用来创建 JDBC 连接的用户名
- password - 用来创建 JDBC 连接的密码
- url - 数据库的 JDBC URL
- dbCreate - 是否根据域模型类自动创建数据库
- pooling - 是否使用数据连接池(默认设置为 true)
- logSql - 启动 sql 日志记录

一个典型的 MySQL 的配置如下：

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

3.3.1 数据源和环境

之前的示例前提是假设我们在 development,production 和 test 等环境下的配置一样。

不过 Grails 中的数据源是可以根据不同环境来进行不同的配置，例如：

```
dataSource {
```

```
        // common settings here
}
environments {
  production {
     dataSource {
          url = "jdbc:mysql://liveip.com/liveDb"
     }
  }
}
```

3.3.2 JNDI 数据源

由于 J2EE 容器一般都支持根据 Java Naming and Directory Interface (JNDI) 来查找数据源。

所以 Grails 也提供了根 据 JNDI 来配置数据源的方式,示例如下:

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

虽然 J2EE 容器间定义 JNDI 名称格式差别很大,但是你定义 DataSource 的 方式依旧是相同的。

3.3.3 自动数据库移植

DataSource.groovy 文 件中的 dbCreate 属性十分重要，因为它可以用来指定是否自动根据 GORM 类来创建数据库表。其可选值为：

- create-drop - 当 Grails 运行时删除并重新建立数据库
- create - 如果数据库不存在则创建数据库，存在则不作任何修改
- update - 如果数据库不存在则创建数据库，存在则进行修改

开发环境（development）下 dbCreate 属性默认设置为 create- drop:

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

这样在 Grails 应用程序启动的时候会删除掉原来的数据库并重新建立，在 production 环境中通常需要修改该值。

虽然 Grails 暂时不支持 Rails 方式的移植，不过可以通过安装 LiquiBase 和 DbMigrate 两 个插件来实现，在控制台中可以通过 grails list-plugins 命令来查看这两个插件。

3.4 外部配置

对于大多数情况来说,使用位于 grails-app/conf 目 录下的 DataSource.groovy 文件中的默认配置已经够用了。不过有些情况下需要在工程项目以外来保存这些配置。例 如在打包 WAR 时，有时候需要将配置文件放置于工程之外来避免由于配置的修改导致 WAR 包的重新打包。

为了支 持这种外部配置，需要在 Config.groovy 文件中首先指定 grails.config.locations 选 项，以便告诉 Grails 应该到何处去加载外部配置文件。

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

"file:${userHome}/.grails/${appName}-config.properties",

"file:${userHome}/.grails/${appName}-config.groovy"]
```

在上例中，我们加载了不同 classpath 下以及 USER_HOME 下的配置文件，包括 Java 属性文件和 ConfigSlurper 配置文件。

最终可以通过 GrailsApplication 对象的 config 属性来获得这些配置文件中的值。

Grails 还支持 Spring 中的属性占位（property place holders）和属性重载（property override）配置，关于这些方面的详细内容请参考 Grails 和 Spring 章节。

3.5 定义版本

版本定义基础

Grails 内置支持应用程序的版本定义。当使用 create-app 创建一个应用程序的时候，该应用程序的版本即被设置为 0.1。 版本是存储在项目文件的根目录下的 application.properties 元文件中。

要改变应用程序的版本可以使用 set-version 命令，例如；

```
grails set-version 0.2
```

Grials 在很多的命令中都会使用到版本，例如 war 命令就会在生成的 WAR 文件最后加上应用程序的版本。

运行时判断版本

由于 Grails 支持应用程序元数据（metadata），所以可 以通过 GrailsApplication 类来得到当前的版本信息。例如在控制其中可以使用内置的 grailsApplication 变量。

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

如果想查看 Grails 的版本，可以有两种方式：

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

或者 GrailsUtil 类：

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

4. 命令行

Grails 的命令行是基于 Gant--用 Groovy 对 Apache Ant 进行的简单封装之上的。

不过 Grails 通过使 用规约以及 grails 命令进一步对其进行了扩展。当输入:

```
grails [command name]
```

时，Grails 会在以下目录中搜索相应的 Gant 脚本并执行：

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

Grails 会自动将命令名称从小写形式转换成驼峰形式（camel case）来查找相应脚本文件，例如如果输入:

```
grails run-app
```

那么 Grails 会去搜索以下脚本文件：

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

如果有多个匹配文件，Grails 会让你选择其中的一个来执行。当 Gant 脚本执行的时 候，“default”目标（Target）也会一同执行。

要想获得有效命令的清单，可以输入:

```
grails help
```

这 样会列出 Grails 中可以使用命令清单，如：

```
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

...
```

关于单个命令的更多详细信息请参考左边菜单的命令行指南。

4.1 创建 Gant 脚本

通过在项目的根目录下运行 create-script 命令可以创建 Gant 脚本。例如：

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

这条命令会在 scripts 目录下创建一个名为 CompileSources.groovy 的 脚本文件。Gant 脚本本身类似于 Groovy 代码，但是它可以支持目标（Target）以及它们之间的依赖关系。

```
target(default:"The default target is the one that gets executed by
Grails") {
        depends(clean, compile)
}
```

```
target(clean:"Clean out things") {
        Ant.delete(dir:"output")
}
target(compile:"Compile some sources") {
        Ant.mkdir(dir:"mkdir")
        Ant.javac(srcdir:"src/java", destdir:"output")
}
```

从如上脚本中可以看出，其中有一个默认可以访问 Apache Ant API 的 Ant 变量。

如上 default 目 标中所使用的一样，你也可以使用 depends 方法来指定其依赖的其他目标。

4.2 可复用的 Grails 脚本

Grails 的命令行中还 可以重用许多有用的脚本代码(查看所有命令的入门指南请参考命令行指南)，其中最主要的有 compile, package 和 bootstrap 脚本。

以 bootstrap 脚本为例，以下代码可以在启动时调用 Spring 中的 ApplicationContext 实例来访问数据源：

```
Ant.property(environment:"env")
grailsHome = Ant.antProject.properties."env.GRAILS_HOME"

includeTargets << new File ( "${grailsHome}/scripts/Bootstrap.groovy" )



target ('default': "Load the Grails interactive shell") {

        depends( configureProxy, packageApp, classpath, loadApp,
configureApp )

        Connection c

        try {

                // do something with connection

                c = appCtx.getBean('dataSource').getConnection()

        }

        finally {
```

```
                c?.close()

        }

}
```

4.3 脚本中的事件

Grails 提供了调用脚本事件的能力，这些事件是在 Grails 目标（target）和插件脚本执行时所触发的。

此机制宗旨是简单和自由，因此无法得到所有可能 触发事件的列表。这样当没有核心目标脚本触发事件的时候，就有可能调用插件脚本所触发的事件。

定义事件句柄

事件句柄是在 scripts/目录下（插 件脚本）或者 USER_HOME 目录下的.grails/scripts/目录下的 Events.groovy 脚 本中定义的。当一个事件触发的时候会调用所有的事件脚本，所以可以使用 10 个插件脚本或者使用一个用户自定义脚本来处理事件。

在 Events.groovy 中是通过以“event”开头命名的代码块来定义事件的。如下例所示，请将下例放入/scripts 目录。

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

eventStatusUpdate = { msg ->

   println msg

}

eventStatusFinal = { msg ->

   println msg

}
```

可以看到其中定义了三个句柄 eventCreatedArtefact,eventStatusUpdate 和 eventStatusFinal。 Grails 也提供其他一些标准的事件，可以在命令行参考指南中找到。例如，compile 命令会触发如下事件。

- CompileStart - 当编译开始时触发，指定编译的类型??源文件或者测试

- CompileEnd - 当编译结束时触发，指定编译的类型??源文件或者测试

触发事件

如下例所示，要触 发 Init.groovy 脚本的事件只需引入它并且调用 event()闭包。

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


event("StatusFinal", ["Super duper plugin action complete!"])
```

常用事件

下表列出了一些常用的事件：

事 件	参数	描述
StatusUpdate	message	传 入的字符串表明当前脚本的状态/进度
StatusError	message	传 入的字符串表明当前脚本的错误信息
StatusFinal	message	传 入的字符串表明最后脚本的状态信息，比如当一个目标/任务（target）完成时，即便是脚本环境中并并存在目标，照样可以显示状态信息
CreatedArtefact	artefactType,artefactName	在 一个 create-xxxx 脚本完成并且创建了一个工件的时候调用
CreatedFile	fileName	在 一个工程的源文件被创建的时候调用，但不包括被 Grails 管理的那些常量文件
Exiting	returnCode	在 脚本环境即将完全终止的时候被调用
PluginInstalled	pluginName	在 一个插件被安装以后被调用
CompileStart	kind	在 编译开始的时候调用，传入的参数是编译的源文件或者测试文件的种类
CompileEnd	kind	在 编译结束的时候调用，传入的参数是编译的源文件或者测试文件的种类
DocStart	kind	在 生成文档即将开始的时候调用 ~~包括 javadoc 或者 groovydoc~~

DocEnd	kind	在生成文档结束的时候调用包括 javadoc 或者 groovydoc
SetClasspath	rootLoader	在 classpath 初始化其间调用，这样插件就可以使用 rootLoader.addURL(...)来设置 classpath 参数。注意！此设置 classpath 参数是在事件脚本加载之后，因此你不能用此 classpath 来加载所需的类，即便是你在事件脚本中已经导入，但是如果你是通过名字来 加载类，那是可以的。
PackagingEnd	none	在 打包完完毕的时候调用（在 Jetty 服务器启动之前，但在 web.xml 生成之后） at the end of packaging (which is called prior to the Jetty server being started and after web.xml is generated)
ConfigureJetty	Jetty Server object	在 Jetty web 服务器配置初始化之后调用

4.4 Ant 和 Maven

Ant 集成

当通过 create-app 命令创建了一个 Grails 应用时，Grails 会自动创建一个 Apache Ant 的 build.xml 文 件，其中包括以下目标：

- clean - 清除 Grails 应用
- war - 创建 WAR 文件
- test - 运行单元测试
- deploy - 缺省是空，但是可以被用来实现自动发布。

这几个目标都可以通过 Ant 来运行，如:

```
ant war
```

这个 build.xml 文 件可以调用 Grails 中的命令，这样就可以将 Grails 应用与持续集成服务器，如 CruiseControl 和 Hudson 集成起来。

Maven 集成

Grails 暂时不支持 Maven ，但是已经有了一个名为 Maven Tools for Grails 的项目，它可以对当前的 Grails 应用创建一个 POM，这样就可以将 Grails 集成到 Maven 的生命周期中。

要获得更多信息请访问 Maven Tools for Grails 项目网站

5. 对象关系映射(GORM)

领域类是任何商业应用的核心， 它们保存这这些商业过程的状态并且实现相应的行为，它们还通过一对一或者一对多的关系相互联系在一起。

GORM 是 Grails 的对象关系映射（ORM）的实现，实际上它使用的是 Hibernate3（非常流行和灵活的开源 ORM 解决方案），但因为有 Groovy 的 动态特性支持，因此 GORM 既支持动态类型也支持静态类型，再加上 Grails 的规约，现在创建 Grails 的领域类只需要更少的配置就可以了。

你也可以用 Java 类写 Grails 的领域类。集成 Hibernate 的相关章节介绍了如何使用 Java 来写 Grails 领域类，但又不失动态持久方法的优势。以下是 GORM 实践的预览：

```
def book = Book.findByTitle("Groovy in Action")

book

  .addToAuthors(name:"Dierk Koenig")

  .addToAuthors(name:"Guillaume LaForge")

  .save()
```

5.1 快速指南

用 create-domain-class 命令创建一个 domain 类:

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

这将在 grails-app/domain/Person.groovy 中创建下面的类：

```
class Person {
}
```

如果你在数据源中将 dbCreate 属性设置为"update", "create" 或 "create-drop"，Grails 会自动生成或修改数据库中的表。

你可以通过添加属性来自定义类：

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

一旦你有了一个 domain 类，你可以通过 shell 或 console 中操纵它：

```
grails console
```

这将会为你载入 一个可以输入 Groovy 命令的交互式图形界面。

5.1.1 基本的 CRUD

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

创建

用 Groovy 的 new 操作符创建一个 domain 类的实例，设置它的属性,然后调用 save:

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

save 方法将会使用底层的 Hibernate ORM 将你的实例持久化到数据库。

读取

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

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

这里使用 get 方法，它会根据数据库标识符从数据库中取回一个 Person 对 象。

更新

要更新一个实例，只 要设置一下属性，然后也是简单的调用一下 save：

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

删除

使用 delete 来删除一个实例:

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

5.2 在 GORM 中进行领域建模

当构建一个 Grials 应用的时候，你必须事先考虑需要解决的问题域 （problem domain）。比如，如果你要构建一个 Amazon 书店，你就要考虑书，作者，顾客和出版商等等。

这些在 GORM 中是作为 Groovy 类来建模的，因此 Book 类要有 title,release date,ISBN number 等属性。在下面的几节将演示在 GORM 中如何为领域建模。

要创建一个领域类，运行 create-domain-class:

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

生成了 grails-app/domain/Book.groovy:

```
class Book {
}
```

如果你想使用包，你可以把 Book.groovy 类移入 domain 目录下的一个子目录，并根据 Groovy(也是 Java)的包规则添加一个合适的 package 声 明。

上面的类将会在数据库中自动映射一个 book (跟类同名)的表。映射规则可以通过 ORM 的 DSL 来自定义。

现在，你有了一个领域类，你可以把 它的属性定义为 Java 语言的类型.比如:

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

每个属性映射到数据库中的一个列,列名的规则命名规则是全部小写并用下划线分隔。比如 releaseDate 映射为列 release_date 上。 SQL 类型是根据 Java 类型来自动检测的，你也可以用约束 或者 ORM DSL 自定义映射类型。

5.2.1 GORM 中的关联

关联定义了 domain 类之间如何 互相交互。除非在两端都显式的指定，否则一个关联只存在于定义它的那端。

5.2.1.1 一对一

一对一的关联是最简单的关联。它只要定义一个属性的类型为另一个 domain 类就可以了。看下面的例子:

示例 A

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

这样我们就有了一个从 Face 到 Nose 的 单向关联。要把这个关联变为双向的,只要在另一端定义一下:

示例 B

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

这样就是双向关联了。但是，这种情况下，关联的双方并不能级联更新。

看下面的变化:

示例 C

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

在这里，我们使用 belongsTo 表示 Nose "属于" Face。其结果就是我们创建并保存它时，数据库可以_级联_更新/插入 Nose:

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

上面这个例子会将 face 和 nose 都保存。注意反向的操作并不可行，它会导致一个对于临时 Face 对 象的错误:

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

belongsTo 的另一个重要作用是，如果你删除了一个 Face 实例，那么相关的 Nose 也会被删除:

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

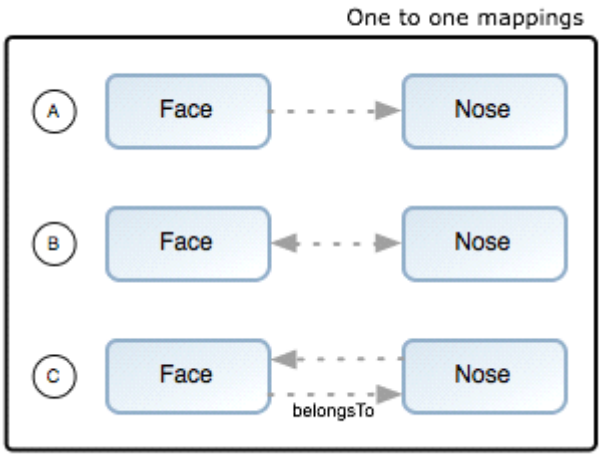
没 有 belongsTo 的话，将不会级联删除，你会得到一个外键约束的错误，除非你手工去删除 Nose:

```
// 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()
```

你可以保持前面那种单向关联的关系，并允许级联保存/ 更新:

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

注意，在这种情况下，因为我们没有在 belongsTo 声 明中使用映射语法明确地声明这个关联,Grails 将认为它是一个单向关联。下面是对这三个例子的总结:



5.2.1.2 一对多

一对多的关系是指，当一个类(比如 Author) 拥有另一个类(Book)的多个实例这种情况。在 Grails 中，你可以使用 hasMany 来定义这种关联：

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

    String name

}

class Book {

        String title

}
```

这样我们有了一个一对多的单向关联。Grails 在数据库级别将默认使用 外键映射来映射这种关联。

ORM DSL 允许使用连接表作为代替来映射单向关联。

Grails 将会根据 hasMany 设置为 domain 类自动注入一个类型为 java.util.Set 的属性。这个可以被用来遍历集合:

```
def a = Author.get(1)

a.books.each {

        println it.title

}
```

Grails 使用的默认抓取策略是"lazy",这意味这集合将会被延迟初始化。如果你不小心的话,这可能导致 n+1 问题 .

如果你需要"eager"抓取，你可以使用 ORM DSL 或者指定立即抓取作为查询的一部分。

默认的级联行为是级联保存和更新，但不级联删除，除非你也指定了 belongsTo ：

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

    String name

}

class Book {

        static belongsTo = [author:Author]

        String title

}
```

如果你有两个相同类型的属性，他们都是一对多关系的多方，你必须用 mappedBy 来 指定他们分别映射的是哪个集合:

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

如果你有多个映射到不同属性的集合，也需要这样:

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

```
        Airport destinationAirport
}
```

5.2.1.3 多对多

Grails 支持多对多关联，这种关联需要在关联的两方都定义 hasMany，并在关联的被拥有方定 义 belongsTo:

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

Grials 在数据库层使用连接表来映射多对多关联。关联的拥有方，在这里是 Author, 负责持久化这个关联，并且它是唯一可以级联保存对方的一方。

比如下面的代码可以工作，并会级联保存:

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

但是下面的代码只保存 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()
```

这正是我们期望的行为，跟 Hibernate 中一样，多对多关联 中只有一方可以管理关联。

Grails 的 脚手架 特性当前还不支持多对多关联，因此你必须自己写代码来管理关联。

5.2.2 GORM 的组合

跟关联一样,Grails 支持组合的概念。在这种情况下,要映射到一个独立的表的类,可以嵌入到当前表。比如:

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

映射的结果看起来像下面这样:

Person Table

id	home_address _number	home_address _code	work_address _number	work_address _code
1	47	343432	67	43545

如果你在 grails-app/domain 目录下一个单独的 Groovy 文件中定义了 Address 类，你还是会得到一个 address 表。如果你不想这样，你可以使用 Groovy 可以在一个文件中定义多个类的特性，在 grails-app/domain/Person.groovy 文件中在 Person 类的下面定义 Address 类。

5.2.3 GORM 的继承

GORM 支持从抽象基类和从具体实体类继承。比如:

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

在上面的例子中我们有个叫 Content 的父类，然后不同的子类有更多它们各自特定的行为。

继承映射的思考

在数据库级别 Grails 默认使用每个类分层结构一个表(uses table-per-hierarchy)的映射策略，有一个辨别标识(discriminator)列叫 class ，因此父类 Content 和它的子类(BlogEntry, Book 等等.)共享同一个表。

每个类分层结构一个表(uses table-per-hierarchy)有一个副作用，就是你继承映射中的属性不能有非空约束。一个选择是使用每个子类一个表(table-per-subclass)映射，它可以通过 ORM DSL 设置。

但是，过度的使用继承和每个子类一个表 (table-per-subclass)的映射策略，会导致过度使用连接查询，使得查询性能很差。通常我们建议保持简单，只有在你确实需要继承的时候才 用它。

多态查询

使用继承的好 处是你获得了多态查询的能力，比如在超类 Content 上使用 list 方法将会返回 Content 的所有子类:

```
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 集合、列表和映射

对象集合(Set)

当你在 GROM 中定义一个关联的时候,默认地使用 java.util.Set ,它是一个无序并不能包含重复对象的集合.换句话说,当你写了下面代码:

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

GORM 注入的 books 属性是一个 java.util.Set. 这里有一个问题,当访问这个集合时是无序的,这可能不是你想要的.为了获得自定义的顺序你可以将这个集合设置为 SortedSet:

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

```
}
```

在这里使用了 java.util.SortedSet 实现,这就意味着你的 Book 类必须实现 java.lang.Comparable 接口:

```
class Book implements Comparable {
   String title
   Date releaseDate = new Date()

   int compareTo(obj) {

       releaseDate.compareTo(obj.releaseDate)

   }

}
```

上面类的结果是 Author 类的 books 集合中所有的 Book 实例将按 他们的发布日期排序.

对象列表

如 果你只是想能够简单的保持对象按照他们被加进去的顺序排序，并能像数组一样按照索引来引用他们,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 // my 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

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()
```

Hibernate 的一个主要的不同在于当你调用 save 时它不需要马上执行任何 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)
```

需要注意的是,这时包括保存之前所有等待执行的 SQL 语句都会同步到数据库中. 你也可以捕获任何抛出的异常,这通常 在包含了乐观锁的高并发情况下非常有用.

```
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()
```

delete 方法也允许通过 flush 参数来控制 flushing.

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

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 方法来执行批量的 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 ,那么就能级联操作,你将需要手工保存每个对象.

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不能工作了.

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.

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 fetchMode = [flights:"eager"]
}
```

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

5.3.4 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。

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:"asc", order:"title")
```

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 动态查找器

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][Suffix]*[Boolean Operator]*[Property][Suffix]

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:

- 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 - 等价于 SQL like 表达式
- Ilike -类似于 Like, 但不是大小写敏感
- NotEqual - 不等于
- Between - 介于两个值之间 (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 )
```

同样的 isNull 和 isNotNull 不需要参数:

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

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)
```

显然这种情况下方法名会变得相当长，这时候你应该考虑使用 条件查询.

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:"asc",
                                         order:"title"])
```

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)
       }
}
```

The above code will find all the Account instances that have performed transactions within the last 10 days. 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

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 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)
        }
```

立即抓取的方式查询

在 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:

```
import org.hibernate.FetchMode as FM
// ......

def criteria = Task.createCriteria()
```

```
def tasks = criteria.list{
      eq("assignee.id", task.assignee.id)
      fetchMode('assignee', FM.EAGER)
      fetchMode('project', FM.EAGER)
      order('priority', 'asc')
}
```

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

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%'",
                   [max:10, offset:20, sort:"asc", order:"title"])
```

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 supports the registration of events as closures that get fired when certain events occurs such as deletes, inserts and updates. To add an event simply register the relevant closure with your domain class. 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。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

触发当一个对象保存到数据库之前

```
class Person {
   Date dateCreated

   def beforeInsert = {

       dateCreated = new Date()

   }

}
```

beforeUpdate 事件

更新前事件

```
class Person {
   Date dateCreated
   Date lastUpdated

   def beforeInsert = {

       dateCreated = new Date()

   }

   def beforeUpdate = {

       lastUpdated = new Date()

   }

}
```

beforeDelete 事件

触发但一个对象被删除

```
class Person {
   String name
   Date dateCreated
   Date lastUpdated
```

```
    def beforeDelete = {

        new ActivityTrace(eventName:"Person Deleted",data:name).save()

    }

}
```

onLoad 事件

触发当一个对象从数据库取出

```
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（本例子中是 firstName），接下来使用命名的 column 来 指定字段名称的映射

列类型

GORM supports configuration of Hibernate types via the DSL using the type attribute. This includes specifing user types that subclass the Hibernate org.hibernate.types.UserType class. As an example 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.types.UserType 的子类。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 列映射到数据库的 SQL TEXT 或者 CLOB 类型

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 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

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，是不是需要一个非严格-读写缓存可能是适宜的
- 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 件中配置 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。此外还有许多不同的生成器可以配置，具体参考 Hibernate 在这个主题上的相关文档。

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

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（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

5.6 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。

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 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，我们可以通过约束来完成，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）的话，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

如果定义了约束 max、min 或者 range，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
```

6. Web 层

6.1 控制器

控 制器是属于请求范围的,用于处理请求，创建或者准备响应。换句话说，每次 request 会创建一个新的控制器实体。控制器能产生响应或者委托给一个视图。创建一个控制器， 简单 创建一个类名以 Controller 结尾的类，并放于 grails-app/controllers 目 录中。

默认的 URL 映射设置能确保你的控制器名字的第一部分被映射到一个 URI 上，并且控制器中的每个操作定义被映射到控制器 命名 URI 中的 URI。

6.1.1 理解控制器和操作

创建一个控制器

控制器可以使用 create-controller 目标创建.作为示例，可以在 Grails 根目录下尝试运行下列命 令。

```
grails create-controller book
```

这条命令将导致在 grails-app/controllers/BookController.groovy 位 置上创建一个控制器。

```
class BookController { … }
```

BookController 默认映射到/book URI 上（相对于您的应用程序根目录）

create-controller 命令只不过是个方便的工具，你还可以使用你喜欢的文本编辑器或者 IDE 更容易的创建控制 器。

创建操作（Actions）

一个控制器可以拥有多个属性，每个属性都被分配一个代码块.每个这样的属性将被映射到 URI 上：

```
class BookController {
    def list = {

        // do controller logic

        // create model

        return model

    }

}
```

默认情况下这个例子映射到/book/list URI 上，因为属性被命名为 list。

默认的操作

一个控制器具有默认 URI 的概念即映射到控制器的根 URI。默认情况下缺省 URI 在这里的是/book。 默认的 URI 通过以下规则来支配：

- 如果只有一个操作存在，一个控制器的默认 URI 将映射到该操作。
- 假如你定义了一个 index 操作，当没有操作被指定在 URI/book 上时，这个操作将处 理请求。
- 除此之外，你可以明确的设置为 defaultAction 属性：

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

6.1.2 控制器和作用域

h4.有效作用域

作用域本质上就像 hash 对象，允许你存储变量。下列为控制器有效作用域：

- servletContext - 也被叫做应用范围，这个范围允许你横跨整个 web 应用程序共享状态. 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 - 可变的进入请求参数 map（map 为 java.util.Map 类型参数）(CGI)。
- flash - 见下文.

访问作用域

作用域的访问可 以通过使用上面变量名与 Groovy 的 array 索引操作符的结合来访问，甚至可以使用 Servlet API 提供的类，例如 HttpServletRequest ：HttpServletRequest:

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

    }

}
```

你甚至可以使用.操作符来存取作用域内部值，这样使语法更加简洁清楚：

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

    }

}
```

这是统一存取不同作用域的方式之一.

使用 Flash 作用域

Grails 支持 flash 作用域的概念，它临时存贮只在这次请求和下次请求中使用的属性，随后属性值将被清除。这在重 定向之前直接设置消息是非常有用的。

```
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 模型和视图

返回模型

模型本质上是个 map 类型，当视图被渲染时使用。map 中的 keys 转变成变量名可以让视图访问。这里有一对方式来 返回模型，第一种方式是明确返回 map 实体：

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

假如没有明确的模型被返回，控制器的属性将被当做模型来使用，因此允许你的代码写 成下面这样:

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

这可能是由于事实上控制器是原型（prototype）范围。换句话说，每次请求，一个新的控制器会被创建，否则上面的代码将不是线程安全的。

在上面示例中的 books 和 authors 属 性将在视图中被使用。

一个更加高级的方法是返回 Spring 的 ModelAndView 类。

```
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 ])

}
```

选择视图

在先前的两个示例里没有代码指定哪个 视图:guide:GSP 去渲染。那么 Grails 怎么知道哪个视图被选取了？答案在于规约。观察这个操作：

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

Grails 将自动在 grails-app/views/book/show.gsp 位置寻找一个视图(事实上 Grails 将首先尝试寻找 JSP 页面, 因为 Grails 可以等同的用于 JSP).

假如你希望渲染另一个视图，那么 render 方法可以帮助你：

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

在这种情况下 Grails 将尝试渲染 grails-app/views/book/display.gsp 位 置上的视图。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。很便利，但是假如你拥有一些共享的视图用来存取，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)。因为使用了高度灵活的 "render"方法。

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

上面的代码打印出"Hello World!" 响应，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")
```

6.1.4 重定向和链

重定向

所有的控制器的操作都可以使用 redirect 方法来重定向。

```
class OverviewController {
    def login = {}

    def find = {

        if(!session.user)

           redirect(action:login)

        .....

    }
```

}

在 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：

- 在同一控制器类中另外的的闭包（closure）：

```
// 同一个类中调用的 login 操作在
redirect(action: login)
```

- 控制器的名字和操作：

```
// 同样重定向到 homeController 中的 index 操作
redirect(controller:'home',action:'index')
```

- 一个应用程序上下文资源关联路径 URI：

```
// 重定向到一个明确的 URI
redirect(uri:"/login.html")
```

- 或者完整的 URL：

```
// 重定向到一个 URL
redirect(url:"http://grails.org")
```

使用 params 作为方法的参数,参数可以被随意的从一个操作传递到下一个：

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

这些参数通过 params 动态属性变得可用，它同样存取请求参数。如果一个参数跟请求参数同名，那么请求参数将被覆 盖，控制器的参数将被优先使用。

由于这个 params 对象同样是一个 map，你可以使用它传递当前的请求参数从一个操作传递到下个操作:

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

链（Chaining）

操作同样可以作为一个链。 链（Chaining）允许模型从一个操作传递到下个操作之间被保存。在下面的操作中调用操作 first 操 作，例如：

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

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

返回的模型结果：

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

通过 chainModelmap，该模型可在随后的连接中的控制器的操作中被存取。这个动态属性只存在于操作：

```
class ChainController {

    def nextInChain = {

        def model = chainModel.myModel

        .....

    }

}
```

同 redirect 方法一样，你也可以给 chain 方法传递参数：

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

6.1.5 控制器拦截器

通常，它有利于拦截基于每个 request,session 或者 application 状态的数据处理。 这个可以通过操作的拦截器来完成。当前有两种类型的拦截器:before 和 after。
如果你的拦截很可能用于一个以上的控制器，你几乎肯定会写一个更好的 Filter，这样可以在一个横切（cross cutting）管理中应用于多个控制器或者 URIs 而不需要改变每个控制器的逻辑

Before 拦截器

这个 beforeInterceptor 拦截器在操作执行之前被处理。如果它返回 false 那么被拦截的操作将不会被执行。在一个控制器内，拦截器可以被定义在所有的操作内，如下：

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

上面在控制器定义主体内被声明。它将在所有操作之前被执行。它将在所有操作之前被处理并且不干扰数据处理。一个通常使用的情形是为了验证：

```
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。一个方法作为一个操作使用，以至于不被暴露于外界（即它是私有的）。beforeInterceptor 被 用于'除了'login'操作之外的所有操作，并被告知执行'auth'方法。'auth'使用 Groovy 的方法指示器语法来引用。在方法中，方法本身 会检查一个用户是否存在于 session 中，否则，它将导航到 login 操作并返回 false,说明被拦截的操作不会被执行。

After 拦截器

使用 afterInterceptor 属性来定义一个拦截器，它会在一个操作之后被执行：

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

这个 after 拦截器获取结果模型作为参数，因此可以执行传送模型操作或者响应。

一个 after 拦截器也可以事先来渲染通过修改 Spring MVC 的 ModelAndView 对象。在这种情况下，上面的例子变成

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

这里允许通过当前的操作改变基于返回模型的视图。注意，modelAndView 可 能为 null 假如被拦截的操作调用 redirect 或 render。

拦 截条件

Rails 用户将会十分熟悉 authentication 示例和当执行拦截的时候怎样使用 'except'条件(在 Gails 中拦截被叫做'filters',这个术语与 Java 中的 servlet filter 存在冲突）：

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

执行除指定操作外的所有操作拦截。操作列表同样可以按下列这样被定义：

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

其他支持的条件是'only'，只执行指定操作拦截：

```
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。它们通常通过表单提交来实现，并总是 String 类型，可是，Groovy 或者 Java 对象的属性很可能不是。

Grails 使用 Spring 的基本数据绑定能力 来完成数据绑定。

绑定 Request 数据到模型上

这里有 2 个方法来绑定请求参数到一个领域类的属性上。第一，涉及到使用领域类的隐式构造器

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

数据绑定发生在 new Book(params)代码 内。通过传递 params 对象到领域类的构造器，Grails 自动识别你正在试图绑定来自请求中的参数。因此，假如 我们有一个像这样进入的请求：

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

那么，title 和 author 请求参数将自动被设置 到领域类上。假如，你需要执行数据绑定到一个已存在的实例上，那么，你可以使用 properties 属性：

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

这与使用隐式构造函数是完全一样的作用

数据绑定和关联

假如你有一个 one-to-one 或 many-to-one 关联，你同样可以使用 Grails 的数据绑定能力来更新这些关联，例如，假如你有这样一个进入的 请求：

```
/book/save?author.id=20
```

Grails 将自动检测请求参数上的 .id 后缀，并在进行数据绑定的时候查找给定的 id 的 Author 实体，像这样：

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

数据绑定多个领域类

有可能绑定数据到领域 类来自 params 对象。

举例来说，你有个进入的请求：

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

你将注意到和上面请求不同的是每个参数带有个像 author. 或 book.这 样的前缀，这是用来分隔哪个参数属于哪种类型的。Grails 的 params 对象就像一个多维散列表，你可以索引

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

注意我们是怎么样使用 book.title 的第一圆点前面的前缀参数来隔离唯一的参数 来绑定。我们同样可以这样来使用 Author 领域类：

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

数据绑定和类型转换错误

有时，当执行数据 绑定时它可能不会把一个特殊的 String 转换到一个特殊的目标类型。你得到一个什么样的类型转换错误。Grails 将保存类型转换错误到一个领域类的 errors 属性中。借这个例子：

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

这里，我们有个领域类 Book，它使用 Java 具体类 型 java.net.URL 来表示 URLs。现在，假设我们有个像这样的进入参数：

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

在这种情况下，它不可能绑定这个 string 类型 a-bad-url 到 publisherURL 属 性上，你可以像这样检查：

```
def b = new Book(params)

if(b.hasErrors()) {

   println "The value
${b.errors.getFieldError('publisherURL').rejectedValue} is not a valid
URL!"

}
```

机关我们尚未隐藏错误代码(更多参考见 验证部分)，对于类型转换错误，你将可以使用 grails-app/i18n /messages.properties file 来定义错误信息。你可以像这样来使用普通的消息处理：

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

或者更具体的：

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

数据绑定和安全关系

当批量更新来自请求参 数的属性时，你必须小心，不能允许客户端绑定恶意的数据到领域类，最后持久到数据库中。

这个问题可以 通过两种方式来避免，一种是使用命令对象。另一种方式是本文所述的，使用灵活的 bindData 方法。

这个 bindData 方 法有着同样的数据绑定能力，但是是对任何对象：

```
def sc = new SaveCommand()
bindData(sc, params)
```

然而，这个 bindData 方 法允许排除某些不想更新的参数：

```
def sc = new SaveCommand()
bindData(sc, params, ['myReadOnlyProp'])
```

6.1.7 XML 和 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 是隐式的通过 render 方法。

render 方 法可以传递一个代码块来执行标记生成器产生 XML

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

这段代码的结果将会像这样：

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

注意，当你使用标记生成器时，必须小心避免命名冲突。例如，这段代码将产生一个错误：

```
def list = {
        def books = Book.list()  // naming conflict here
        render(contentType:"text/xml") {
                books {
```

```
                        for(b in results) {
                                book(title:b.title)
                        }
                }
        }
}
```

原因是，这里的一个本地变量 books 企图作为方法被 调用。

使用 render 方法输出 JSON

render 方法可以同样被用于输出 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，结果就会是大致相同的：

```
[
        {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)领域类为 XML 通过特定的转换器。

首先，导入 grails.converters 类包到你的控制器（Controllers）中：

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

现在，你可以使用下列高度易读的语法来自动转换领域类成 XML：

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

输出结果看上去会像下列这样：

```
<?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 部分

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
```

输出结果看上去会像下列这样：

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

再次作为一种替代，你可以使用 encodeAsJSON 达 到相同的效果

6.1.8 上传文件

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>
```

有那么一些的方式来处理文件上传。第一种方法是直接与 Spring 的 MultipartFile 实例合作：

```
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')
    }
}
```

这显然是得心应手，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.

文件上传通过数据绑定

文件上传同样可以通 过数据绑定来完成。例如，假定你有个 Image 领域类按照下面的示例：

```
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 来更新一个领域类情形时是非常有用的。

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，就象你在领域类用法一样。

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: {

                        loginService.canLogin(username, password)

                })

        }

}
```

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

6.2 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")
```

一个 GSP 是典型的混合标签和 GSP 指令来帮助视图渲染。

虽然，它可能在把 Grrails 逻辑嵌入到你的 GSP 中，这样做将会覆盖这个文档，这个做法是强烈反对的。混合标签和代码是个 bad 事情，很多 GSP 没有包括代码也没有必要这样做。

一个 GSP 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 变量和作用域

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 标签已经足够：

```
<% 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不被转义的，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 变量和作用域

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 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 Book 集合的实体。不过，假设每个 Book 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 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')}" />
```

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 custom namespace，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 视图和模板

和视图一样，Grails 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。

既然这样，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

```
<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]" />
```

在控制器和标签库中的模板

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 = {
        de b = Book.get(params.id)
        render(template:"bookTemplate", model:[book:b])
}
```

在控制器中的 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 = {
        de b = Book.get(params.id)
        String content = g.render(template:"bookTemplate",
model:[book:b])
        render content
}
```

注意， g. namespace 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

早前的示例也证明了 pageProperty 标签能被用去检查和返回目标页面的方向。

启 动布局

这里有少许方法来启动启动一个布局。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--></menu>
                 <div class="body">
                                        This is my content!
                 </div>
            </div>
      </body>
</html>
```

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>
```

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 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。

6.3.2 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>
```

6.3.3 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 德尔变量名，随 后，我们可能在标签主体类引用这个变量。

6.3.4 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。

6.4 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做：

```
/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。

动态控制器和操作名

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，控制器名，操作名和 id 名，隐式的从嵌入在 URL 中的 controller, action 和 id 中获得。

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
}
dao
```

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 BookControllerURL 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"(controllers:"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"(controllers:"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"(controllers:"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"(controllers:"image")
}
```

这样的话，它将储存路径，从 params 对象获得命名参数里的通配符：

```
def name = params.name
println name // prints "logo.jpg" or "other/logo.jpg"
```

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 统一验证约束机制，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")
}
```

这寻去 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 Flow

概述

Grails 构建在 Spring Web Flow 项目之上来支持创建 web flow。一个 web flow 是一个会话，它横跨多个请求，并把状态保留在 flow 范围内。一个 web flow 同样定义了开始和结束状态。

Web Flow 不需要一个 HTTP session,作为另外一种选择，他们将状态存储在一种连续的形态中，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 flow 就是一个基础而高级的状态机，它管理这从一个状态到下一状态的“流”。既然是状态管理，你就勿需再关心用户在一个多步骤的流程中已经进入那个操作 了，因为 Web Flow 已经帮你管理了，因此 Web 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

创建一个流只需简单的创建一个常规 的 Grails 控制器，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，当重定向或引用流时，可以把它看做一个操作而省略掉 Flow 前 缀。换句话说，上面流的操作名为 shoppingCart.

6.5.1 开始和结束状态

如前所述，一个流定义了开始和结束状 态。一个开始状态是当一个用户第一次启动一个会话（或流）时开始。Grails 的开始流是第一个需要一个块的方法调用。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 状态没有定义一个操作或重定向，它被认为是一个视图状态。通过规约，引用 grails-app/views/book/shoppingCart/showCart.gsp 视 图。

注意，这不像正规的控制器的操作，这个视图被存储在和它流的名字匹配的目录 grails-app/views/book/shoppingCart 中。

shoppingCart 流同样可能拥有两个结束状态。第一个是 displayCatalogue， 执行一种外部重定向到另一个控制器和行动，从而结束流。第二个是 displayInvoice 是一个最终状态，因为它已经没有 根本没有任何事件了，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。

一旦一个流结束，它可能只是从开始状态重新开始，在这种情况下的 showCart 不 会来自任何其他状态。

6.5.2 操作状态和视图状态

视图状态

一种视图状态没有定义 action 或 redirect。 因此，下面是一个视图状态的示例：

```
enterPersonalDetails {
    on("submit").to "enterShipping"
    on("return").to "showCart"
}
```

它将默认查找一个名为 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。假如你想改变视图的渲染，你可以使用 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。假如你想使用共享的视图，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

操作状态

一个操作状态是执行代码却不渲染 任何视图的这样一个状态：。操作结果被用于支配流的切换。创建一个操作状态，你需要定义一个被执行的操作：

```
listBooks {
   action {
          [ bookList:Book.list() ]
   }
   on("success").to "showCatalogue"
   on(Exception).to "handleError"
}
```

正如你所看到的，一个操作看上去非常类似于一个控制器的操作，实际上，假如你需要 你可以重用控制器操作的。假如这个操作没有错误成功返回，success 事件将被触发。这样的话，我们返回一个 map,它被到 做"model"看待，并自动放置于 flow 范围.

另外，在上面的示例中同样使用了一个异常处 理在这行上处理错误：

```
on(Exception).to "handleError"
```

当错误发生的情况下，它将使流切换到一个静态调用 handleError.

你可以写更复杂的操作与流请求上下文相互作用：

```
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"

}
```

这里，一个更复杂的操作收集所有的来自流范围积累的数据，并创建一个 Order 对 象。然后，把 Order 作为模型返回。在这里值得注意的重要事情是与请求上下文和 "flow scope"的相互作用。

切换操作

其他格式的操作被称之为切换操作。 切换操作,一旦一个事件被触发，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， 只是简单的记录这个切换。切换状态对于数据绑定和验证是非常有用的，将在后面部分涵盖。

6.5.3 流执行事件

为了执行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 事件，指出流下一步该做什么。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"
    }
    ...
}
```

一旦这个 showCart 事件是一个视图状态，它将渲 染 grails-app/book/shoppingCart/showCart.gsp 视图.在这个视图内你需要有触发流执 行的组件。

```
<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。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" />
```

来自于一个操作的触发事件

为了触发来自于 一个操作的一个事件，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 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 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 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）

流的作用域和序列化

当你将 对象放到 flash, flow 或者 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 classes 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 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 事件中使用的闭包比如 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 flow 范围中。同样有趣的是，我们执行 验证，并在验证失败是调用 error()方法。

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

虽 然，Grails 支持良好的细粒度拦截器，它们只是对少数控制器有用，当管理大型应用时就会变得很困难。另一方面，过滤器能横跨整组控制器,一个 URI 空间或者一种具体的操作。过滤器对插件更容易，维护彻底的分离你主要的控制器逻辑，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 块中定义的过滤器拥有一个 name 和一个 scope。name 是方法的名字，scope 使用的命名参数被定义。例如，假如你需要定义一个引用于所有控制器和操作的过滤器：

```
sampleFilter(controller:'*', action:'*') {
  // interceptor definitions
}
```

过滤器的 scope 可能是下列事件之一：

- 可选择的通配符配对一个控制器和/或操作名字
- 一个 URI, 使用 Ant 路径匹配语法

过滤器的一些示例包括：

- 所有的控制器和操作

```
all(controller:'*', action:'*') {

}
```

- 只对 BookController

```
justBook(controller:'book', action:'*') {

}
```

- 适用于 URI 空间

```
someURIs(uri:'/book/**') {

}
```

- 适用于所有的 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 过滤器的类型

在过滤器的主体内，你可以定义下列过滤器的 拦截器类型之一：

- before - 操作（Action）之前执行. 返回 false 指出将要的过滤器，并且操作（Action）将不被执行
- after - 操作（Action）之后执行. 获取第一参数作为视图模型
- afterView -视图显然之后执行

例如，完成普通的身份验证使用情形，你可以定义一个过滤器，如下：

```
class SecurityFilters {
   def filters = {
       loginCheck(controller:'*', action:'*') {
           before = {
              if(!session.user && !actionName.equals('login')) {
                  redirect(action:'login')
                  return false
               }
           }
```

```
        }
    }
}
```

这里的 loginCheck 过滤器使用一个 before 拦截器来执行一个代码块，检查是否一个用户 在 session 内，假如不是，重定向到 login 操作。注意，怎么样返回 false 确保操作本身不被执行.

6.6.3 过滤器的功能

过滤器支持所有在控制器 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 - request parameters 对象
- actionName - 正被发送的操作名
- controllerName - 正被发送的控制器名
- grailsApplication - 当前运行的 Grails 应用
- applicationContext - ApplicationContext 对象

然而，过滤器只支持一个到控制器和标签库可用的方法的子集 。包括：

- redirect - 重定向到其他的控制器和操作
- render - 渲染定制响应

6.7 Ajax

Grails 所写的动态框架，通过它的标签库提供对构建 Ajax 应用的支持。完整的标签列表，查看标签库参考资料。

6.7.1 用 Prototype 实现 Ajax

Grails 默认装载 Prototype 库，但是通过 Plug-in 系统提供像 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。在开始之 间，你必须在你页面的<head>标签内添加这样一行：

```
<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 异步链接

远程内容可以使用多种方法载入，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，并随意的在一个元素中设置响应。如下 简单方法创建一个远程连接：

```
<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 更新内容

这很重要，但是通常你可能想提供一些事情发 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 异步表单提交

一个 HTML form 同样可以以两种方式之一被异步提交。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，你也可以使用 submitToRemote 来创建一个提交按钮。它允许一些按钮远程提交，同时，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

假 如，确定事件发生，特定的 javascript 可能被调用。所有的事件以"on"开头，在适当的地方反馈给用户，或者采用其他的操作：

```
<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现，其他的事件还包括：

- onSuccess - 成功时被调用 javascript 函数
- onFailure - 失败时被调用 javascript 函数
- on_ERROR_CODE - 处理指定的错误代码时调用 javascript 函数 (例如 on404="alert('not found!')")
- onUninitialized - 一个 ajax 引擎初始化失败时调用 javascript 函数
- onLoading - 当远程函数加载响应时调用 javascript 函数
- onLoaded - 当远程函数加载完响应时调用 javascript 函数
- onComplete - 当远程函数完成（包括任何更新）时调用 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，从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 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 流在传送过程中被赋值

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])

}
```

在客户端调用这个涉及使用 remoteLink 标签:

```
<g:remoteLink action="showBook" id="${book.id}"
update="book${book.id}">Update Book</g:remoteLink>
<div id="book${book.id}">
   <!--existing book mark-up -->
</div>
```

数据为中心的 Ajax 与 JSON

数据为中心 的 Ajax 通常涉及在客户端上的响应赋值和通过编程更新。Grails 中的 JSON 响应，你将使用 Grails 典型的 JSON marshaling 能力：

```
import grails.converters.*

def showBook = {

        def b = Book.get(params.id)

    render b as JSON

}
```

随后，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 同样简单：

```
import grails.converters.*

def showBook = {

        def b = Book.get(params.id)

    render b as XML

}
```

然而，因为涉及到 DOM，客户变得更加复杂：

```
<g:javascript>
function updateBook(e) {
        var xml = e.responseXML
        var id = xml.getElementsByTagName("book").getAttribute("id")
        $("book"+id+"_title")=xml.getElementsByTagName("title")[0].te
xtContent
}
<g:javascript>
<g:remoteLink action="test" update="foo"
onSuccess="updateBook(e)">Update Book</g:remoteLink>
<g:set var="bookId">book${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,在客户端得到赋值。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。 假如你在客户端使用 Prototype，由于这个 contentType 设置，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，因为，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 JavaScript 插件提供了类似的功能。

6.8 内容协商

Grails 使用任何一个 HTTP Accept 报头来内置支持内容协商，一种明确格式化请求参数或扩展的一种 URI 映射 。

配 置 Mime 类型

在你开始处理内容协商之前，你必须告诉 Grails 你希望支持什么样的内容类型。默 认情况下，Grails 在 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 请求都有个特别的 Accept 报头，它定义了客户端能“接受”什么样的媒体类型(或 mime 类型)。这个在老式的浏览器中是比较典型的：

```
*/*
```

简单的表示任何事 物。然而，在较新的浏览器中，更多有用的东西像这样被全部一起发送(一个 Firefox Accept 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，来描叙更为理想的请求格式。对于上述的例子下列的断言将通过

```
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。如果如前所述，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'可能的格式将被客户端接受。为了处理来自控制器不同类型的请求，你可以使用 withFormat 方法，它的行为被当作 switch 语句类型:

```
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 }

                }

        }

}
```

这里发生了什么，假如优先格式是 html，随 后 Grails 将只执行 html()调用。这个做了什么，它使 Grails 寻找一个视图调用每个 grails-app/views/books/list.html.gsp 或

grails-app/views/books/list.gsp。假如这个格式是 xml， 随后，闭包将会被调用，一个 XML 响应被渲染。

假如这个 accept 报头是'all'，就像在老式浏 览器里，随后，格式的选择将基于 withFormat 方法之内的调用秩序。在这种情况下，早前示例的 html 方法将首先被调 用。

当使用 withFormat 时确保它是控制器操作的最后一个被调用，当 withFormat 方 法的返回值被操作使用支配下一步做什么。

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直，那么你可以覆盖在 request 参数中的特定 format 来 解决：

```
/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 扩展来支持内容协商。例如，给出下列 URI：

```
/book/list.xml
```

Grails 将刮去扩展，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

为了在一个集成测试中测试内 容协商(参见 测试部分)你能任一个操纵引入的请求报头：

```
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

}
```

或者你可以设置这个格式化参数来完成一个简单效果：

```
void testJavascriptOutput() {
        def controller = new TestController()
        controller.params.format = 'js'

        controller.testAction()

        assertEquals "alert('hello')",
controller.response.contentAsString

}
```

7. 验证

Grails 的验证功能是建立在 Spring's Validator API 和数据绑定之上的，但 Grails 在此特性和约束机制的基础上又 提供了统一的定义验证约束的方法。

Grails 中的约束是采用声明性验证规则的方式进行的，大多情况下这些规 则都是应用在领域类上，不过 URL 映射 and 命令对象同样也支持约束。

7.1 声明约束

在一个领域类中，约束 constraints 被定义在 constraints 属性的代码块中：

```
class User {
    String login
    String password
    String email
    Integer age

    static constraints = {

          ...

    }

}
```

然后可以通过方法来调用相匹配的属性名，而属性的约束是通过命名参数的组合来满足特定的限制的：

```
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)
    }
}
```

在上述示例中，我们声明的 login 属性其长 度是 5-10 个字符，而且不能为空，且必须唯一，我们同样也在 password, email 和 age 属性上应用了约束。

完整有效的约束参考请浏览参考指南

7.2 验证约束

验证基础

要验证一个领域类，你可以在任何实例上调用其 validate 方法：

```
def user =  new User(params)
if(user.validate()) {
    // do something with user
}
else {
    user.errors.allErrors.each {
        println it
```

```
    }

}
```

领域类的 errors 属性是 Spring 的 Errors 接口实例，Errors 接口提供了访问这些验证错误的方法，并且还可以取得原 始的值。

验证阶段

在 Grails 中，有两个基本的验证阶段，第一个是数据绑定，发生在将你的请求参数绑定到一个类实例的时候：

```
def user = new User(params)
```

在这个时候如果你通过 errors 捕获到错误，可能是因为类型转换（比如将字符串转化 为日期类型），你还可以通过 Errors 来检测和获取到输入的原始值，比如：

```
if(user.hasErrors()) {
        if(user.hasFieldError("login")) {
                println user.getFieldError("login").rejectedValue
        }
}
```

验证的第二阶段发生在你调用 validate 或者 save 的时候，Grails 将通过你定义的 constraints 来验证这些值的边界。比如，缺省情况下，在执行 save 操作以前，将会先调用 validate 方法，因此你可以写 如下类似代码：

```
if(user.save()) {
    return user
}
else {
    user.errors.allErrors.each {
        println it
    }
}
```

7.3 客户端验证

显 示错误

通常情况下，如果你得到一个验证错误，你希望重定向到要渲染的视图，在那里，你需要一些方法 来呈现这些错误。Grails 通过很多丰富的标签来处理这些错误，比如你只想简单的将这些错误显示为列表（list），你可以用 renderErrors

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

如果你想更详细地显示，你可以用 hasErrors 和 eachError

```
<g:hasErrors bean="${user}">
  <ul>
   <g:eachError var="err" bean="${user}">
       <li>${err}</li>
   </g:eachError>
  </ul>
</g:hasErrors>
```

高亮显示错误

当一个字段输入不正确的时候，经常是用红色方框或者类似的东东来高亮显示，这也可以通过调用 hasErrors 来完成，比如：

```
<div class='value ${hasErrors(bean:user,field:'login','errors')}'>
   <input type="text" name="login"
value="${fieldValue(bean:user,field:'login')}"/>
</div>
```

这段代码所做的是检查 user bean 的 login 字段是 否存在错误，如果发现错误，就在 div 上的增加一个 errors 的 CSS 类，这样你就可以使用 CSS 的规则来高亮显示 div 了。

获取输入值

每一个 error 实际上是 Spring 的 FieldError 类的一个实例，它包含着输入的原始值，当你希望使用出错的对象来恢复原来值的时候，这是很有用 的，你可以通过 fieldValue 标签来实现：

```
<input type="text" name="login"
value="${fieldValue(bean:user,field:'login')}"/>
```

此代码将先检查 User bean 上是否存在一个 FieldError， 如果存在它将为 login 字段获取到原始输入值。

7.4 验证和国际化

Grails 中关于错误的另一重要的事情是不要在代 码中到处硬编码，在 Grails i18n 所支持的消息绑定（message bundles）的协助下，Spring 的 FieldError 类基本可以解决硬编码信息的问题。

约 束和信息代码

约束代码本身就是规约（convention）支配的，让我们看看早些时候的约束的示例：

```
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)

    }

}
```

通常如果违反了 blank 约束，Grails 将以如下形式寻找信息编码：

```
[Class Name].[Property Name].[Constraint Code]
```

在 blank 约束中，以上代码应该是 user.login.blank， 因此在你的 grails-app/i18n/messages.properties 文件中需要如下信息：

```
user.login.blank=Your login name must be specified!
```

关于什么编码匹配什么约束的参考请参照每个约束的参考指南（本指南的 constraint 部分--译者注）。

显示信息

renderErrors 标签将使用 message 标签来为你寻找信息，但是如果你希望自己更多控制显示，那你需要自己做如下的事情：

```
<g:hasErrors bean="${user}">
  <ul>
   <g:eachError var="err" bean="${user}">
       <li><g:message error="${err}" /></li>
   </g:eachError>
  </ul>
</g:hasErrors>
```

在此示例中，我们在 eachError 标签内又使用了 message 标签，通过 error 参数我们可以读取错误的信息。

8. 服务层

和 web 层一样，Grails 中也有 service 层的概念。Grails 团队不鼓励在 controller 中嵌入 核心应用逻辑，因为这样不利于代码的重用，也影响清晰的分层。

Grials 中,应用的主要逻辑都放在的 service 层，controller 负责处理请求流程。

创建一个 Service

要创建一个 Grails service,你只要进入命令行模式，在项目的根目录下，执行 create-service 命令:

```
grails create-service simple
```

这样就会创建一个 service，这个 service 位于 grails-app/services/SimpleService.groovy .除了名字按照 Grails 的约定以 Service 结尾以外,这个文件就是一个普通的 Groovy 类:

```
class SimpleService {
}
```

8.1 声明式事务

Services 通常会包含这样的逻辑--需要多个 domain 类之间相互配合。因此它常常会出现这样的情况:涉及到的持久化包括大量的数据库操作。这些问题使得 service 中经常都需要对方法进行事务管理。当然你可以用 withTransaction 方法来管理事务，但是这样很繁琐，也不能充分利用 Spring 的强大的事务抽象能力。

Grails 中可以对 service 进行事务划分，它声明 service 中所有方法都是事务型的。缺省所有的 service 都进行了事务划分。要禁用这个配置，只需要设置 transactional 属性为 false:

```
class CountryService {
    static transactional = false
}
```

你也可以设置这个属性为 true，以防止将来这个默认 值改变后对你的应用造成影响，或者是明确声明 service 是事务型的。

警告：依赖注入 是使声明式事务工作的**唯一** 途径。如果你自己用 new 操作符，比如 new BookService()，将不能得到一个事务型 的 service.

这样的结果是所有的方法被包装在一个事务中，在方法中有异常抛出时，将会自动回滚。事务传播级别默认是 PROPAGATION_REQUIRED.

8.2 服务的作用域

缺省情况下，service 中的方法不是同步的,没 有什么能阻止并发执行 service 中的方法。实际上，因为 service 是单例并且可能会并发执行，你要在 service 中保存状态时需要慎重考虑一 下，最好是不要在 service 中存储任何状态.

你可以通过将 service 放入一个特定的范围来改变这个行 为.目前支持的范围有:

- prototype - 当它每次被注入另外一个类时，创建一个新的 service
- request - 每个请求创建一个新的 service
- flash - 为当前和下一个请求创建一个 service
- flow - 为一个 web flow 创建一个 service
- conversation -在 web flow 中，为一个会话创建一个 service. 比如一个根流程和它的子流程.
- session - 为一个用户会话创建一个 service
- singleton (默认) - 始终只有一个 service 实例存在

如果你的 service 是 flash, flow 或者 conversation 范围的，它需要实现 java.io.Serializable 接口，并且只能用于 Web Flow 上下文.

要使用上面的某个范围，只要在你的类中增加一个 static scope 属性:

```
static scope = "flow"
```

8.3 依赖注入和服务

基 本的依赖注入

Grails service 的一个重要方面是它从 Spring Framework 获得的依赖注入能力。Grails 提供了"基于约定的依赖注入".也就是说，你可以使用 service 的类名作为属性名，自动将它们注入到 controller,tag 库等地方。

例如,有一个叫做 BookService 的 service,如果你在一个 controller 里定义了一个 bookService 属性:

```
class BookController {
   def bookService
```

```
    …
}
```

这时 Spring 容器将会根据这个 service 的 scope 配置，自动注入一个 service 的实例。所有的依赖注入都根据名字完成;Grails 不支持根据类型注入.你也可以指定类型，如下:

```
class AuthorService {
        BookService bookService
}
```

但是这样做会产生一个副作用，如果你在开发模式下改变了 BookService ，重新加载时会抛出一个错误.

依赖注入和 service

你可以用同样的方法在一个 service 中注入另一个 service.比如你有一个 AuthorService， 它需要使用 BookService,那么只要这样:

```
class AuthorService {
        def bookService
}
```

依赖注入和 Domain 类

你甚至可以将 service 注入到 domain 类中,这在开发富 domain 模型时可能会有用:

```
class Book {
        …
        def bookService
        def buyBook() {
                bookService.buyBook(this)
        }
}
```

8.4 使用 Java 的服务

Grails service 封装了可重用的逻辑,你可以在包括 Java 类的其他类中使用他们，这是它另外一个强大地方。有多种方法重用用 Java 编写的 service,最简单是把你的 service 移进 grails-app/services 目录下的一个包里,因为 Java 中不能从缺省包(如果没有声明包,就使用缺省包)中导入一个类。所以像下面用 Java 编写的 BookService 就 不能被导入从而无法重用:

```
class BookService {
```

```
        void buyBook(Book book) {
                // logic
        }
}
```

但是，这个问题是很容易解决的。你只要将这个类放进一个包里就可以了，也就是将这 个类移进一个子目录，比如 grails-app/services/bookstore ，然后修改包声明:

```
package bookstore
class BookService {
        void buyBook(Book book) {
                // logic
        }
}
```

另外一个替代方案是，在包里面建立一个 service 要实现的接口:

```
package bookstore;
interface BookStore {
        void buyBook(Book book);
}
```

然后 service 实现这个接口:

```
class BookService implements bookstore.BookStore {
        void buyBook(Book b) {
                // logic
        }
}
```

后面这种方法更清楚，因为在 Java 里只引用了接口，与实现类无关。无论哪种方 法，这个任务的目标是要使得 Java 在编译时能够静态地找到它要用的类(或接口)。现在这个任务完成了,你可以在 src/java 包里面创建一个 Java 类，然后增加一个 setter 方法，以便在 Spring 中能够装配这个 bean:

```
package bookstore;
// note: this is Java class
public class BookConsumer {
        private BookStore store;

        public void setBookStore(BookStore storeInstance) {

                this.store = storeInstance;
```

```
        }

        ...

}
```

完成了这些，你就可以在 grails-app/conf/spring/resources.xml(关 于这个的更多信息请参考 Grails and Spring 一章)中将这个 Java 类为一个 Spring bean 了:

```
<bean id="bookConsumer" class="bookstore.BookConsumer">
        <property name="bookStore" ref="bookService" />
</bean>
```

9. 测试

自动化测试是 Grails 的关键部 分之一，它是用 Groovy Tests 实现的。因此，Grials 提供了从低级的单元测试到高级的功能测试的多种方法来使测试更简单。这章详细解释 Grails 提供的各种 测试的不同功能。

首先需要知道的是所有的 create-*命令结束时 Grails 都会为你创建集成测试。比如说你运行 create-controller 命令:

```
grails create-controller simple
```

Grails 不仅为你在 grails-app/controllers/SimpleController.groovy 创 建了一个 controller,而且在 test/integration/SimpleControllerTests.groovy 生 成了一个集成测试类.Grails 不会为你填充测试中的逻辑！这个留给你做.

一旦你完成了这个，就可 以用 test-app 命令执行所有的测试:

```
grails test-app
```

执行上面的 命令将会得到下面这样的输出：

```
-------------------------------------------------------
Running Unit Tests…
Running test FooTests...FAILURE
Unit Tests Completed in 464ms …
-------------------------------------------------------

Tests failed: 0 errors, 1 failures
```

同时将会在 test/reports 目录下生成测试报告。你也可以按照指定的名字(不含 Tests 后缀)运行某个测试:

```
grails test-app SimpleController
```

另外，把一组测试名用空格分开，你就可以运行一组测试:

```
grails test-app SimpleController BookController
```

9.1 单元测试

单元测试是"单元" 级的测试。也就是说，你可以测试单独的方法或者代码块而不用考虑周边基础设施.在 Grails 中，你需要确切地理解单元测试和集成测试的差 别，Grails 在集成测试和运行时生成的动态方法，在单元测试中都不可用。

因为这个原因，Grails 提供 了一些工具来模拟这些方法，比如 Groovy Mock 或者 ExpandoMetaClass.

例如,在 BookController 里你有一个这样的 action：

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

params 对象和 get 方法都是运行时 Grails 提供的，在单元测试中不能用，但你可以用 ExpandoMetaClass 模 拟它们:

```
void testShow() {
        // mock the static get method
        Book.metaClass.static.get = { Long id ->
                assert id == 10
                new Book(id:id,title:"The Stand")
    }
    // mock the params object
        BookController.metaClass.getParams = {-> [id:10] }
        def controller = new BookController()
        def model = controller.show()
        assert model
        assert model.book
        assertEquals 10, model.book.id
        assertEquals "The Stand", model.book.title
}
```

注意上面我们是怎样自定义了一个 get 方法的实现，让 它返回一个模拟的实例，甚至在这个实现中可以使用断言。还要注意我们怎样用 map 模拟一个 params 对象的实例。

9.2 集成测试

集成测试与单元测试 的不同在于你在测试中可以完全访问 Grails 的环境。Grails 用一个 in-memory HSQLDB 数据库做集成测试,并在每次测试时都清空数据库中所有数据.

测试 Controller

要测试 controller 你首先要理解 Spring Mock 库

实际上，Grails 为会每个测试自动配置 MockHttpServletRequest, MockHttpServletResponse, 和 MockHttpSession。你可以用它们运行你的测试。例如下面的 controller：

```
class FooController {

        def text = {

            render "bar"

        }

        def someRedirect = {

                redirect(action:"bar")

        }

}
```

相应地它的测试将会是:

```
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

        }

}
```

在上面例子里，response 是 MockHttpServletResponse 的一个实例，我们可以用它为这个例子实现 contentAsString (当写入到这个 response 时)或者 URL 重定向.这些 Servlet API 的模拟版本不像真实版本，在它们中所有东西都是可变的，所以你可以设置 request 上的属性，比如 contextPath 等 等。

在集成测试时，Grails 在执行 action 时 **不 会** 自动调用 interceptors。你需要在孤立环境中测试拦截器，如果需要可以参考 functional testing 。

测 试有 Service 的 Controller

如果你的控制器引用了一个 service,你必须在测试 中明确地初始化这个 service。

下面是一个用了 service 的 controller:

```
class FilmStarsController {
    def popularityService

    def update = {

        // do something with popularityService

    }

}
```

它的测试应该是:

```
class FilmStarsTests extends GroovyTestCase {
    def popularityService

    public void testInjectedServiceInController () {

        def fsc = new FilmStarsController()

        fsc.popularityService = popularityService
```

```
        fsc.update()

    }

}
```

测试 controller 命令对象

要使用 command 对象，你只需将参数放入 request,然后不带参数执行 action 方法，它为你自动装配 command 对象.

下面是一个使用 command 对象的 controller:

```
class AuthenticationController {
  def signup = { SignupForm form -> .. }
}
```

你可以这样测试它:

```
def controller = new AuthenticationController()
controller.params.login = "marcpalmer"
controller.params.password = "secret"
controller.params.passwordConfirm = "secret"
controller.signup()
```

Grails 会自动把你对 signup()的 调用看作 作为一个 action 调用,并将模拟的 request 参数装配为 command 对象。在 controller 测试过程中，Grails 提供的模拟 request 的 params 是可变的.

测试 controller 和 render 方法

render 方法允许你在 action 内部任何地方渲染自定义视图。下面看一个示例:

```
def save = {
        def book = Book(params)
        if(book.save()) {
                // handle
        }
        else {
                render(view:"create", model:[book:book])
        }
}
```

在上面的例子中，action 结果中的 model 不是一个返回值，而是保存在 controller 的 modelAndView 属性中。 modelAndView 属性是 Spring MVC 的 ModelAndView 类的一个实例，你可以用它测试 action 的执行结果:

```
def bookController = new BookController()
bookController.save()
def model = bookController.modelAndView.model.book
```

模拟 request 数据

如果你正在测试一个 需要 request 数据的 action,比如 REST web service,那么你可以用 Spring 中的 MockHttpServletRequest 对象。例如下面这个 action 它从一个 request 中获取数据进行绑定:

```
def create = {
        [book: new Book(params['book']) ]
}
```

如果你想把'book'参数模拟为一个 XML 请求，你可以这么做:

```
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

}
```

同样的也可以得到一个 JSON 请求:

```
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
}
```

用JSON的话别忘了用 class 指定 要绑定的目标类的名字.在XML中这个隐含地用<book>节 点的名字，但是在JSON中你需要把这个属性作为JSON数据包属性的一部分.

关于REST web service主题的更多信息请参考REST这章.

测试web flow

测试Web Flows要用grails.test.WebFlowTestCase作为 基类,这个类是Spring Web Flow中AbstractFlowExecutionTests类的一个子类.

WebFlowTestCase 类的子类必须是集成测试.

比如下面是一个简单流程：

```
class ExampleController {
        def exampleFlow = {
                start {
                        on("go") {
                                flow.hello = "world"
                        }.to "next"
                }
                next {
                        on("back").to "start"
                        on("go").to "end"
                }
                end()
        }
}
```

You need to tell the test harness what to use for the "flow definition". This is done via overriding the abstract getFlow 你需要重写抽象方法getFlow,指定要用的流程定义。

```
class ExampleFlowTests extends grails.test.WebFlowTestCase {
        def getFlow() { new ExampleController().exampleFlow }
        ...
}
```

如果你需要指定流程id,你可以重写getFlowId方法，否则缺省是 test:

```
class ExampleFlowTests extends grails.test.WebFlowTestCase {
        String getFlowId() { "example" }
        …
}
```

一旦上面工作都完成了，你要用 startFlow 方法 开始流程，这个方法返回一个 ViewSelection 对象:

```
void testExampleFlow() {
        def viewSelection = startFlow()

        assertEquals "start", viewSelection.viewName

        …

}
```

正如上面所演示的，你可以用 ViewSelection 对象的 viewName 检查你是否在正确的状态。你还需要用 signalEvent 方法触发事 件:

```
void testExampleFlow() {
        …
        viewSelection = signalEvent("go")
        assertEquals "next", viewSelection.viewName
        assertEquals "world", viewSelection.model.hello
}
```

这里我们给流程一个执行事件"go"的信号，这会切换到"next"状态.在这个 例子里，这个切换动作将 hello 变量放入 flow 范围。我们可以通过检查上面 ViewSelection 的 model 属 性来测试这个变量的值。

测试标签库

测试标签库实际上非常简单，因为当标签被当作方法调用的时候，它的结果作为字符串返回。所以如果我们有一个这样的标 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}>"
    }
}
```

测试是这样的:

```
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!"
        }
     }
}
```

注意第二个例子，testBodyTag，我 们传一个块进去，它返回这个标签的 body。这里会方便地将 body 作为一个字符串.

用 GroovyPagesTestCase 测试标签库

除了像上面这样简单地测试标签库以外，你也可以 用 grails.test.GroovyPagesTestCase 来测试标签库.

GroovyPagesTestCase 是 GroovyTestCase 类 的一个子类，它提供了一些测试 GSP 输出的工具方法。

GroovyPagesTestCase 只能用于集成测试.

下面是一个日期格式化标签的例子:

```
class FormatTagLib {
```

```
        def dateFormat = { attrs, body ->
                out << new java.text.SimpleDateFormat(attrs.format) <<
attrs.date
        }
}
```

可以用下面的方法很容易地测试这个标签:

```
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] )

        }

}
```

你也可以用 GroovyPagesTestCase 类的 applyTemplate 方法获得 GSP 的结果:

```
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

测试 domain 类通常只是简单的测试 GORM API 的使用,但是有一些事情必须知道。首先,如果你测试查询,你经常需要用"flush"来确保将正 确的状态持久化到数据库中。例如下面的例子:

```
void testQuery() {
        def books = [ new Book(title:"The Stand"), new Book(title:"The
Shining")]
        books*.save()

        assertEquals 2, Book.list().size()

}
```

这个测试将会失败，因为 save 调用在被调用后并没有真正持久化 Book 实例。save 调 用只是告诉 Hibernate 在以后某些时候要将这些实例持久化。如果你希望马上提交更改的话你需要"flush"他们:

```
void testQuery() {
        def books = [ new Book(title:"The Stand"), new Book(title:"The
Shining")]
        books*.save(flush:true)

        assertEquals 2, Book.list().size()

}
```

这样，因为我们传了一个值为 true 的 flush 参 数，更新操作立即执行，所以随后的查询是正确的。

9.3 功能测试

的支持。

用下面 的命令开始安装 Web Test:

```
grails install-plugin webtest
```

然后请参考 reference on the wiki ，这里解释了怎样使用 Web Test 和 Grails.

10. 国际化

Grails 通过 Spring MVC 所提供的功能支持国际化（i18n）。在 Grails 中,你可以根据用户的 Locale 自定义任意视图中的文本。Java 类中 Locale 的 javadoc 引用如下：
一个 Locale 对象描述了一个特定地理的、政治的和文化的语言环境。一个要求 Locale 执行的任务操作是 语言环境敏感的 ，它使用 Locale 来为用户裁剪信息。例如，显示一个数字是语言环境敏感的操作--数字应根据用户的国家、地区或文化的风俗/传统来格式化。

Locale 是由语言代码 和 国家代码组成。例如，"en_US"是美国英语的语言码，而"en_GB"代表英国英语。

10.1 理解信息绑定

既然你已经有了 Locales 的概念，为了在 Grails 中利用它们，你必须要创建你希望展现的不同语言的信息绑定。Grails 的信息绑定是一些简单的 Java 属性文件,并且位于 grails-app/i18n 目 录下。

每一个绑定的名字习惯上都是以 messages 开始，并且以 locale 结束。Grails 的 grails-app/i18n 目录下自带了很多不同语言的信息绑定。比如：

```
messages.properties
messages_de.properties
messages_es.properties
etc.
```

缺省情况下，Grails 将在 messages.properties 查 找信息，除非用户指定了自己的 locale。你也可以创建你自己的信息绑定，只需要简单地创建一个属性文件，此文件要以你感兴趣的 locale 结尾。比如 messages_en_GB.properties 代表的是英国英语。

10.2 改变 Locales

缺省情况下,用户的 locale 是 通过 HTTP 信息头(header)中的 Accept-Language 来检测的。但是你也可以简单地在 Grails 的请求 参数中传入 lang 参数来切换用户认可的 locale，比如：

```
/book/list?lang=es
```

Grails 将自动地切换到用户的 locale，并且将其保存在 cookie 中，因此接下来的请求将使用新的信息 头。

10.3 读取信息

在视图中读取信息

你最需要信息的地方是在视图中，要在视图中读取信息，只需要使用 message 标签：

```
<g:message code="my.localized.content" />
```

只要在你的 messages.properties （当然要有适当的 locale 后缀）中有相应的关键字（如以下所示），Grails 将会在其中查找消息：

```
my.localized.content=Hola, Me llamo John. Hoy es domingo.
```

注意，有时候你可能需要传入一些参数给消息，这在 message 标签中也是可以做到 的，比如：

```
<g:message code="my.localized.content" args="${ ['Juan', 'lunes'] }" />
```

接下来就可以在消息中使用这些位置参数了：

```
my.localized.content=Hola, Me llamo {0}. Hoy es {1}.
```

在控制器和标签库中读取信息

既然你可以在 控制器中象调用方法一样调用标签，那也可以很容易的在控制器内读取信息：

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

同理，你也可以在标签库中使用，但是要注意：如果你的标签库有不同的命名空间，你需要加一个前缀 g.：

```
def myTag = { attrs, body ->
        def msg = g.message(code:"my.localized.content", args:['Juan',
'lunes'])
}
```

11. 安全

Grails 的安全性既不强于也不弱于 Java Servlets 的安全性。Java servlets (也即 Grails)对常见的缓存溢出和的非法 URL 攻击基本上完全免疫，这得益于 Java Virtual Machine 对 Servlet 代码在安全性方面的支持。

Web 的安全性问题通常是由于开发 者的幼稚或过失造成的，grails 在避免通常的错误方面有少许帮助，从而更容易编写安全的 Web 应用。

Grails 自动完成了什么

Grails 默认的内置安全机制。

1. 所有通过 GORM 领域对象的数据库访问将自动执行 SQL 转义来防止 SQL 注入攻击
2. 默认的 scaffoldingHTML 模板在显示的时候将转义所有的数据字段

3. Grails 所有的连接创建标签(link, form, createLink, createLinkTo 等等)使用适当的转义机制防止代码注入
4. Grails 提供 codecs 允许你在渲染 HTML，JavaScript 和 URLs 的时候进行简单的数据转义以防止注入攻击

11.1 预防攻击

SQL 注入

GORM 领域类的底层基于 Hibernate 技术，在提交数据到数据库时 Hibernate 会自动执行数据转义，因此 SQL 注入攻击不是什么问题。然而，使用未检查的请求参数仍然有可能编写出会被 SQL 注入攻击的动态 hql 代码。例如，下面的代 码很容易受到 HQL 注入攻击：

```
def vulnerable = {
        def books = Book.find("from Book as b where b.title ='" +
params.title + "'")
}
```

不要这样做。如果 你需要传入参数，请使用命名参数或位置参数作为替代：

```
def safe = {
        def books = Book.find("from Book as b where b.title =?",
[params.title])
}
```

网络钓鱼

从避免你的品牌和客户被抢走的角度来说，网络钓鱼实际上是一个公共关系问题。客户需要懂得如何识别收到的电子邮件是否合法。

XSS - 跨网站脚本注入

你的应用应该尽可能多的验证请求是来自你自己的 Web 应用而不是来自另一个网站，这一点非常重要。标签和页面流会 对此有所帮助，并且 Grails 支持的 Spring Web Flow 包括这样的默认安全措施。

同 样重要的是，确保所有的数据被渲染到页面时都能正确的转义。例如，当渲染 HTML 或者 XHTML 时，你必须在所有对象上调用 encodeAsHTML 以确保恶意注入 Javascript 数据或 HTML 标签不会被执行或显示。为 此，Grails 提供了几个 Dynamic Encoding Methods ，假如提供的这些 Dynamic Encoding Methods 无法满足你的输出数据的转义格式，你也可以很容易的编写自己的编解码器.

你还必须避免使用请求参数或 数据字段来决定用户的下一个 URL 重定向。假如，你使用一个 successURL 参数来决定一个用户成功登陆后将重定向到何 处，攻击者可以利用你自己的网站来模仿你的登录程序，之后，一旦用户登陆成功便会重定向到他们的网站，在这个网站上允许潜在的 JS 代码利用登入进来的帐 号。

HTML/URL 注入

这 是指在某处提供了非法的数据，导致之后页面上使用该数据创建链接，点击它将不会导致预期的行为，而是可能重定向到另一个网站或者改变请求参数。

HTML/URL 注入问题通过 Grails 提供的 codecs 很容易被解决掉，并且 Grails 提供的标签库全部采用了 encodeAsURL。假如创建你自己的标签来产生 URLs 同样要注意这样来使用它。

拒绝服务

负载平衡器和其他装置在这里可能更 有用，但是，也同样涉及到过度查询的问题，例如，攻击者在某处创建一个连接来请求最大结果集合，以至于一个查询造成内存溢出或使系统速度减慢。这里的解决 方法是，在把他们传递进动态查询器和其他 GORM 查询方法之前总是对请求参数做处理。

```
def safeMax = Math.max(params.max?.toInteger(), 100) // never let more
than 100 results be returned
return Book.list(max:safeMax)
```

猜测 ID

许多应用使用 URL 的最后一部分 作为对象的"id"从而通过 GORM 或者在其它手段查询数据， 。尤其在 GORM 中，id 很容易猜测因为通常情况下他们是有序的整数。

因此，你必须在响应返回给用户之前，确定请求用户有权限查看带有请求 id 的对象.

不要落入"模糊安全性"的陷阱，像有个默认的"letmein"密码等等这样的设置将不可避免的造成"模糊安全 性"。

你必须确保每个不受保护的 URL 能以这样或那样的方式被公开存取。

11.2 字符串的编码和解码

Grails 支持动态编/解码 (dynamic encode/decode)方法的概念。Grails 捆绑了一组标准的编解码器（codecs）。Grails 还提供了一个简单的机制使得开发人员可以 贡献出自己的编解码器，这些编解码器会在运行时被识别。

编解码器类

一个 Grails 编解码器类就是一个普通的类，该类可能包含一个编码闭包，一个解码闭包或者两者皆有。当 Grails 应用启动时，Grails 框架将动态加载来自 grails-app/utils/目录中的编解码器。

Grails 框架将在 grails-app/utils/中按照规约查找以 Codec 结 尾的类名。例如，Grails 标准的编解码器之一 HTMLCodec。

如 果一个编解码器包含了一个编码器 encode 属性，Grails 将创建一个动态的编码器方法，然后把该方法加到 String 类 中，该方法的名字将体现编解码器类的名字和编码器闭包的名字，例如 HTMLCodec 类定义了一个编码器 encode 属性，所以 Grails 将把编码器对应得闭包关联到 String 类中，并且使用名字 encodeAsHTML 来表示。

HTMLCodec 和 URLCodec 类同样定义了 解码器 decode，因此 Grails 也将用名字 decodeHTML 和 decodeURL 把 这些解码器关联到 String 类中。动态编解码器方法可以在 Grails 应用的任何地方调用。考虑这样一个用例，一个 report 包含一个 'description'属性，'description'中可以包含特殊字符，为了在 HTML 文档中正确的显示，这些特殊字符必须被转义，处理这种情 况的一种方法就是在 GSP 中使用动态编码方法来编码 description 属性，如下：

```
${report.description.encodeAsHTML()}
```

解码是通过 value.decodeHTML()语法执行的。

标准编解码器

HTMLCodec

这个编解码器执行 HTML 的转义和 反转义，因此数据可以在一个 HTML 页面中被安全的渲染而无需为此创建任何 HTML 标签也不会破坏页面的布局。例如，假定有字符串"Don't you know that 2 > 1?",你不可能安全的在 HTML 页面中显示它，因为，符号>将被看做是一个关闭标签，当该字符串是在一个属性中时，例如是一个输入字段的值，情况 就更加糟糕。

使用示例：

```
<input name="comment.message"
value="${comment.message.encodeAsHTML()}"/>
```

注意，HTML 编码不会再编码省略号/单引号，因此，你必须在属性值上使用双引号来避免省略号在页面上干扰你的文本。

URLCodec

当创建 URL 链接或表单动作时，必须进行 URL 编码。这可以防止 URL 中包含非法字符，例如，字符串"Apple & Blackberry" 作为 GET 请求中的参数是会有问题的，因为，字符串中的符号&会破坏参数的解析。

使用示例：

```
<a
href="/mycontroller/find?searchKey=${lastSearch.encodeAsURL()}">Repea
t last search</a>
```

Base64Codec

执行 Base64 编/解码功能，使用示例：

```
Your registration code is: ${user.registrationCode.encodeAsBase64()}
```

JavaScript 编解码器 JavaScript 编解码器将转义字符串,以便字符串可以作为合法的 JavaScript 字符串使用。使用示例：

```
Element.update('${elementId}', '${render(template:
"/common/message").encodeAsJavaScript()}')
```

定制编解码器

Web 应用可以定义自己的编解码器，Grails 会像加载标准的编解 码器一样加载他们。一个定制的编解码器类必须定义在 grails-app/utils/目录中，且类名必须以 Codec 结 尾。编解码器可能包含一个静态的编码器属性 static encode，或者一个静态的解码器属 性 static decode 或者两者皆有。实现编码器或解码器的闭包期望传入一个对象作为参 数，编码器或解码器的动态方法将在该对象上调用。例如：

```
class PigLatinCodec {
  static encode = { str ->
                                        2） 如果单词以元音开头，则加后缀
way。如“apple”变换后为“appleway”。 )
  }
}
```

当定义好上面的编解码器之后，包含该编解码器的 Web 应用就可以写下如下的代码：

```
${lastName.encodeAsPigLatin()}
```

11.3 身份验证

虽然，当前没有默 认的验证机制，因为可能有数以千计的不同方式来实现验证。然而，通过使用 interceptors 或 filters 可以轻松的实现简单的验证机制。

Filters 允许你 应用一个横跨所有控制器或者横跨一个 URI 空间的验证。例如。你可以在一个名为 grails-app/conf/SecurityFilters.groovy 类 中创建一组新的 filters：

```
class SecurityFilters {
   def filters = {
       loginCheck(controller:'*', action:'*') {
           before = {
              if(!session.user && actionName != "login") {
                  redirect(controller:"user",action:"login")
                  return false
                   }
           }

       }

   }

}
```

这里的 loginCheck 过滤器在操作 （action)被执行之前实施拦截，假如，session 中没有用户且操作 （action） 不是 login， 则重定向到 login 操 作。

login 操作也很简单：

```
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 关于安全的插件

如果你的需求超出了简单的验证功能，还需要诸如授 权，角色等等的高级功能，那么你可能需要考虑使用一个可用的安全性（security）插件.

11.4.1 Acegi

Acegi 插件构建在 Spring Acegi 之上，"Spring Acegi"为各种验证和授权方案提供了灵活，易于扩展的实现框架。

Acegi 插件要求你指定 URIs 和角色之间的映射，该插件提供了一个默认的领域模型来建模用户，授权和请求之间的映射。 查看更多详情请参考 wiki 文档

11.4.2 JSecurity

然后提供一个 accessControl 代 码块来设置角色，如下例所示：

```
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' ])

    }

    ...

}
```

关于 JSecurity 插件的更多信息请参考 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 插件 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，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
```

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 插件并且想发布在官方的存储库中，请联系"G2One 团队"http://www.g2one.com 的成员，他们将为你分配访问的权限。

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

```
grails release-plugin
```

这将自动地将改动提交到 SVN 和创建标签（svn 的 tagging），并且通过 list-plugins 命令你可以看到这些改动

12.2 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 web-app 目 录下，比如 web-app/plugins/example-1.0/js.

因此，要从正确的地方引用这些静态资源也就成为插件的责任。比如，你要在 GSP 中引用一个 JavaScript 文 件，你可以这样：

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

这样做当然可以，但是当你开发插件并且单独运行插件的时候，将产生相对链接（link）的问题。

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

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

这样在运行期间 pluginContextPath 变量将会等价于/ 或 /plugins/example，这取决于插件是单独运行还是被安装在 Grails 应用中。

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

12.3 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
```

12.4 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.

12.5 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脚本中的事件

12.6 参与运行时配置

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

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)

        }

}
```

)的解析和切换，它是通过 Spring Bean Builder 语法来完成的。

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

        }

}
```

12.7 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.metaClass.each { metaClass ->
             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 实 例，代码如下：

```
import org.springframework.orm.hibernate3.HibernateTemplate

class ExampleHibernatePlugin {

   def doWithDynamicMethods = { applicationContext ->

       application.domainClasses.metaClass.each { metaClass ->

           metaClass.static.load = { Long id->

                def sf = applicationContext.sessionFactory

                def template = new HibernateTemplate(sf)

                            template.load(delegate, id)

           }

       }

   }
```

```
}
```

另外因为 Spring 容器具有自动装配和依赖注入的能力，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。

12.8 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
- event.application - GrailsApplication实例

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

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。

12.9 理解插件加载的顺序

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, 以及 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。

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 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 服务可以让你的 Web 应用提供 web API，一般来说是通过 SOAP 或者 REST 实现的。

13.1 REST

REST 与其说是一种技术还不如说是一个构架模式。REST 是极 其简单的，使用普通的 XML 或者 JSON 来作为通信媒介，使表现底层系统的 URL 模式和 HTTP 方法（ GET, PUT, POST 和 DELETE）结合在一起。

每一种 HTTP 方法对应于一个动作（action），比如 GET 用来获 取数据，PUT 用来创建数据，POST 用来更新数据等等，从这种意义上来说，REST 跟 CRUD 正好般配。

URL 模式

用 Grails 实现 REST 的第一步是提供一个 RESTful 的 URL 映射，如下：

```
static mappings = {
   "/product/$id?"(controller:"product"){
       action = [GET:"show", PUT:"update", DELETE:"delete",
POST:"save"]
   }
}
```

此处为了给控制器提供 RESTful API，我们利用了 URL 映射（Mapping）的映射到 HTTP 方法，每个 HTTP 方法比如 GET，PUT，POST 和 DELETE 都映射到控制器的唯一动作 （action）。

XML 列集 - 读取

控制器本身就可以通过 Grails 的 XML 列集 来支持 GET 方法，代码如下：

```
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
                }
        }
        ..
}
```

此处我们所做的是：如果 params 中有 id，那我们 将根据 name 来查找 Product，并将其返回；否则我们将返回所有的 Product。这样如果 我们访问/products 将会得到所有的产品，而通过/product/MacBook 访问，我 们将得到 MacBook

XML 列集 - 更新

Grails 是通过 params 对象来支持象 PUT 和 POST 这样 的 REST 更新的，params 对象能够读取传入的 XML 信息包。一个传入的 XML 信息包可以如下所示：

```
<?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 params 对象来读取 XML 信息包，相关的技术描述请参考数据绑定一节

```
def save = {
        def p = new Product(params['product'])

        if(p.save()) {

                render p as XML

        }

        else {

                def errors = p.errors.allErrors.collect
{ g.message(error:it) }

                render(contentType:"text/xml") {

                        error {

                                for(err in errors) {

                                        message(error:err)

                                }

                        }

                }

        }

}
```

此例中，通过检索 params 对象的关键字'product'， 我们可以创建并将 XML 绑定到 Product。关注一下这行：

```
def p = new Product(params['product'])
```

看起来处理 XML 请求的代码跟处理提交表格（submits form）的数据没有什么变化，类似的技术也可以应用在 JSON 请求中。
如果你需要不同的响应来满足不同的客户端（REST, HTML 等），那你可以使用内容协商

最后 Product 对象被成功保存后被 render 为 XML，否则将产生一个被 Grails 校验的错误信息。

```
<error>
   <message>The property 'title' of class 'Person' must be
specified</message>
</error>
```

13.2 SOAP

Grails 是通过 XFire 插件来支持 SOAP 的，此插件基于非常流行的 XFire SOAP 协议栈。XFire 插件通过一个特殊的 expose 属性，可以将 Grails 的服务导出为 SOAP 服务，代码如下：

```
class BookService {

  static expose=['xfire']

  Book[] getBooks(){

    Book.list() as Book[]

  }

}
```

可以通过 ULR http://127.0.0.1:8080/your_grails_app/services/book?wsdl 来访问其 WSDL。

更多 XFire 插件的信息请参考知识库（wiki）的 XFire 插件文档。

13.3 RSS 和 Atom

Grails 并没有对 RSS 或者 Atom 提供直接支持。你可以通过控制器 render 方法的 XML 能力来构造 RSS 或者 ATOM。尽管如此，还是有一个 Feeds 插件 (基于流行的 ROME 开发库)来为 Grails 提供 RSS 和 Atom 的生成器

```
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

此章节是为 Grails 的高级 用户准备的，此类用户对 Grails 如何与 Spring Framework 集成和构建很感兴趣，另外对于考虑在运行期间(runtime)配置 Grails 的插件开发者也很有裨益。

14.1 Grails 的支柱

Grails 实际上是 Spring MVC 应用的包装，而 Spring MVC 又是建在 Spring 之上的基于 MVC 的 Web 应用框架。虽然 Spring MVC 跟 Structs 等框架在易用性方面都不尽如人意，但 Spring MVC 有着极好的设计架构，对 Grails 来说，Spring MVC 是一个可以基于其上进行开发的完美框架。

Grails 在以下领域对 Spring MVC 进行取舍：

- 基本的控制器逻辑 - Grails 继承了 Spring 的 DispatcherServlet 类并且将请求转给 Grails 的控制器。
- 数据的绑定和验证 - Grails 的验证和数据绑定都是 Spring 所提供的。
- 运行时配置 - Grails 的全部基于系统的运行时约定都是通过 Spring 的 ApplicationContext 整合起来的。
- 事务 - Grails 在 GORM 中使用 Spring 的事务管理。

换句话说，Grails 自始至终使用内嵌的 Spring 来处理

Grails 的 ApplicationContext

Spring 的开发者往往很热切地想了解 Grails 的 ApplicationContext 是如何构造的，下面就介绍一下基础知识。

- Grails 的父 ApplicationContext 是从 web-app/WEB-INF/applicationContext.xml 中构造的。

ApplicationContext 创建了 GrailsApplication 实例和 GrailsPluginManager。

- 将上述的 ApplicationContext 作为 Grails 的父 bean，不仅可以用 GrailsApplication 实例来分析 Grails 的约定，而且还可以创建子 ApplicationContext 作为 web 应用的根 ApplicationContext 。

配置的 Spring Beans

大多数的 Grails 配置是发生在运行时的。每一个插件都可以将配置好的 Spring beans 注册到 ApplicationContext 中。 一个引用需要配置哪些 beans，请参考引用指南，在那里有每个 Grails 插件的描述和哪些 beans 需要配置。

14.2 配置其他 Bean

使用 XML

可以将 Bean 配置在你 Grails 工程的 grails-app/conf/spring/resources.xml 文件中，此文件是一典型的 Spring XML 文件，具体描述详见最佳参考，作为一个入门级示例，你可以象下面示例那样进行配置：

```
<bean id="myBean" class="my.company.MyBeanImpl"></bean>
```

一旦配置好 bean，比如此处的 myBean，此 bean 将会自动装配在 Grails 应用的大多数类型中，比如控制器（controlers），标签库（tag libraris），服务（services）等等，在控制器中用法如下：

```
class ExampleController {

    def myBean

}
```

引用已经存在的 Beans

声明在 resources.xml 里的 Beans 也可以引用 Grails 的约定类。比如你需要在 bean 中引用一个 BookService 这样的服务，你只需要用一个属性名来表达相应的类名就可以了，此示例中 BookService 的名称是 bookService， 代码如下：

```
<bean id="myBean" class="my.company.MyBeanImpl">
       <property name="bookService" ref="bookService" />
</bean>
```

bean 本身当然需要一个公共的 setter 方法，在 Groovy 中可以这样定义：

```
package my.company
```

```
class MyBeanImpl {
        BookService bookService
}
```

或者在 Java 这样定义：

```
package my.company;
class MyBeanImpl {
        private BookService bookService;
        public void setBookService(BookService theBookService) {
                this.bookService = theBookService;
        }
}
```

因为 Grails 的许多配置是在运行时通过约定进行的，因此很多的 beans 不需 要到处声明定义，而且这样也不影响在你的 Spring 配置中引用它们。例如你需要引用 Grails 的 DataSource ，你可以这样做：

```
<bean id="myBean" class="my.company.MyBeanImpl">
        <property name="bookService" ref="bookService" />
        <property name="dataSource" ref="dataSource" />
</bean>
```

或者你需要 Hibernate 的 SessionFactory，下面代码将适合你：

```
<bean id="myBean" class="my.company.MyBeanImpl">
        <property name="bookService" ref="bookService" />
        <property name="sessionFactory" ref="sessionFactory" />
</bean>
```

关于有效的完整的 beans 引用，请看本参考指南的插件部分。

使 用 Spring DSL

如果你想使用 Grails 提供的 Spring DSL，那你需要创建 grails-app/conf/spring/resources.groovy 文 件并且定义一个称之为 beans 的方法，其代码块如下：

```
beans {
        // beans here
}
```

上述 XML 示例配置用 DSL 表示如下：

```
beans {
```

```
        myBean(my.company.MyBeanImpl) {
                bookService = ref("bookService")
        }
}
```

用这种方式的主要优点是你可以在 bean 的定义中混入业务逻辑，比如基于 environment 的示例如下：

```
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 通过 Beans DSL 运行 Spring

Spring 非常强大，但是基于 XML 的语法却是非常的 繁琐而且直接违反了 DRY（不要重复自己）原则，这种状况在 Spring2.0 中也没有好转。Grails 的 Bean 生成器就是为了简化那些以 Spring 为核心的 bean 的组装。

此外，Spring 的常规配置（使用 XML）是静态的因此很难在运行时 修改和配置，而通过程序创建 XML 则繁琐且容易出错。Grails 的 BeanBuilder 使得通过程序在运行时组装组件成为可能，而且还可以让你的逻辑能根据系统属性或者环境变量做相 应的调整。

这将使代码随环境而变成为可能，同时避免了不必要的重复（对于测试、开发和上线产品等不同 环境来说，就要有不同的 Spring 配置）。

BeanBuilder 类

Grails 通过 grails.spring.BeanBuilder 类和 Groovy 动态语言来定义 bean。其基本用法如下：

```
import org.apache.commons.dbcp.BasicDataSource
import
org.codehaus.groovy.grails.orm.hibernate.ConfigurableLocalSessionFact
oryBean;
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 中，你不需要创建一个 BeanBuilder 实例，因为该实例已经在各自的 doWithSpring 和 beans 代码块中创建了。

上述的示例展示了如何通过 BeanBuilder 类来配置合适的数据源。

本质上，每一个方法调用（如上例中的 dataSource 和 sessionFactory）将映射到 Srping 的 bean 名称，而且方法的第一个参数是 bean 所对应的类，而最 后一个参数是代码块。在代码块中，你可以用标准的 Groovy 语法来设置 bean 的属性。

Bean 的 引用是通过 bean 的名称来自动解析的。上面代码中的 sessionFactory 解析 dataSource 就是一个很好的例子。

某些特定的跟 bean 管理相关的属性也可以通过生成器构造，比如以下代码：

```
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]
    dataSource = dataSource
    hibernateProperties = [ "hibernate.hbm2ddl.auto":"create-drop",
                            "hibernate.show_sql":true  ]
}
```

中括号中的字符串跟 Spring 的 XML 配置中定义的 bean 属性名称是一致的。

使用构造参数

构造参数可以被定义在方法的 bean 类（第一个参数）和最后闭包（代码块）之间，如下所示：

```
bb.beans {
   exampleBean(MyExampleBean, "firstArgument", 2) {
       someProperty = [1,2,3]
   }
}
```

配置 BeanDefinition（使用工厂方法）

闭包的第一个参数是 bean 实例的一个引用，因此你可以在此配置工厂方法，并且可以调用 AbstractBeanDefinition 类的任何方法。

```
bb.beans {
   exampleBean(MyExampleBean) { bean ->
       bean.factoryMethod = "getInstance"
```

```
        bean.singleton = false
        someProperty = [1,2,3]
    }
}
```

另一种选择是，你可以利用 bean 定义方法的返回值来配置这个 bean，如下：

```
bb.beans {
    def example = exampleBean(MyExampleBean) {
        someProperty = [1,2,3]
    }
    example.factoryMethod = "getInstance"
}
```

使用工厂 bean

Spring 中定义了工厂 bean 的概念，通常一个 bean 不是直接从 class 创建的，而是通过这些工厂 bean。在这种情况下，bean 是没有类的，相反你必须传递一个工厂 bean 的名称给要创建的 bean，如下：

```
bb.beans {
    myFactory(ExampleFactoryBean) {
        someProperty = [1,2,3]
    }
    myBean(myFactory) {
         name = "blah"
    }
}
```

注意在上述的例子中，我们在 bean 的构造方法中使用 myFactory bean 的引用而不是一个类。另一个常用的方法是提供工厂方法的名字供工厂 bean 调用，这是通过 Groovy 的命名参数语法来实现的，如下：

```
bb.beans {
    myFactory(ExampleFactoryBean) {
        someProperty = [1,2,3]
    }
    myBean(myFactory:"getInstance") {
         name = "blah"
    }
}
```

此处 ExampleFactoryBean bean 的 getInstance 方法将被用来创建 myBean bean。

运行期间创建 Bean 引用

有时候，你直到运 行的时候才知道要创建的 bean 名称，此时你可以通过字符串内插（string interpolation）的方式来动态的定义一个 bean，如下：

```
def beanName = "example"
bb.beans {
   "${beanName}Bean"(MyExampleBean) {
       someProperty = [1,2,3]
   }
}
```

在这个例子中，当要定义一个 bean 的时候，先前定义的 beanName 变量将被使用。

此外，当跟其他的 beans 一起组装的时候，有时你要引用的 bean 名字直到运行时 才知道，在这种情况下，需要用到 ref 方法，如下：

```
def beanName = "example"
bb.beans {
   "${beanName}Bean"(MyExampleBean) {
       someProperty = [1,2,3]
   }
   anotherBean(AnotherBean) {
       example = ref("${beanName}Bean")
   }
}
```

在这个示例中， AnotherBean 的 example 属性就是在运行时被设置为 exampleBean。 ref 方法也可以被用来引用父 ApplicationContext 中的 beans，前提是父 ApplicationContext 要通过 BeanBuilder 构造函数传递进来，如下：

```
ApplicationContext parent = ...//
der bb = new BeanBuilder(parent)
bb.beans {
   anotherBean(AnotherBean) {
       example = ref("${beanName}Bean", true)
   }
}
```

这里的第二个参数被置为 true 用来指定要引用的 bean 将在父上下文 （parent context）中寻找。

使用匿名（内部）Beans

你可以通过使用匿名内部类的方式来设置 bean 的一个属性，此匿名代码块将此 bean 的类型作为其参数，代码如 下：

```
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
  }
}
```

在上述示例中，我们设置 bean marge 的 husband 属性为一个代码块（其实就是一个闭包--译者），在此代码块中创建了内部的 bean 应用。作为另外一种选择，如果你有一个工厂 bean， 那你就可以忽略此 bean 类型，只需要传入一个 bean 定义来设置相应的工厂，代码如下：

```
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]
  }
}
```

定义抽象 Beans 和父 Bean

创建一个抽象的 bean，只需要定义一个没有类的 bean 就可以了，如下所示：

```
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"

  }

  ...

}
```

在此我们定义了一个抽象的 bean，其 leader 被设置为 "Lancelot"，现在为了使用这个抽象 bean，只需将此抽象 bean 设置为子 bean 的父就好了，如下 bean.parent 所示：

```
bb.beans {
  ...
  quest(HolyGrailQuest)
  knights(KnightOfTheRoundTable, "Camelot") { bean ->
      bean.parent = abstractBean
      quest = quest
  }
}
```

当你使用父 bean 的时候，parent 一定要设置在其他属性之前！

如果你希望抽象 bean 带一个类参数，那你可以象下边这么做：

```
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
  }
}
```

在上述的示例中，我们创建了一个 KnightOfTheRoundTable 类型的抽象 bean，并且使用 bean 参数变量来设置其属性，随后我们定义了一个 bean：knights，此 bean 没有类参数，但其继承了父类 bean。

在绑定（Binding）/上下文（Context）中增加变量

如果你是使用脚本来加载 bean 的话，那你可以使用 Groovy 的 Binding 对象的 binding 属性的方式 来加载。代码如下：

```
def bb = new BeanBuilder("classpath:*SpringBeans.groovy")
def binding = new Binding()
binding.foo = "bar"

bb.binding = binding

def ctx = bb.createApplicationContext()
```

从文件系统中加载 Bean 定义

你也可以通 过 BeanBuilder 来加载相同 ClassPath 路径下的定义在外部的 Groovy 脚本中的 bean（当然要符合 BeanBuilder 语法），示例如下：

```
def bb = new BeanBuilder("classpath:*SpringBeans.groovy")

def applicationContext = bb.createApplicationContext()
```

此处 BeanBuilder 将加载 ClassPath 下边所有以 SpringBeans.groovy 结尾的 Groovy 文件中的 bean,示例脚本如下：

```
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  ]
        }

}
```

14.4 配置属性占位

通过扩展 Spring 的 PropertyPlaceholderConfigurer，Grails 支持属性占位符的概念，这一般结合着外置的配置来使用。

对 Spring 的配置文件 grails-app/conf/spring/resources.xml 来 说，占位符值的设置既可以是 ConfigSlurper 脚本，也可以是 Java 属性文件，如下例所示，相应的配置存放在 grails-app/conf/Config.groovy （或者一个外置的配置文件中）中：

```
database.driver="com.mysql.jdbc.Driver"
database.dbname="mysql:mydb"
```

这样你就可以在 resources.xml 使用 ${..} 语法来使用占位符了。

```
<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>
```

14.5 配置属性重载

通过扩展 Spring 的 PropertyOverrideConfigurer，Grails 支持属性重写的概念，这一般结合着外置的配置来使用。

本质上，你可以通过 ConfigSlurper 脚本中的 beans 代码块来重写一个 bean 的配置，如下所示：

```
beans {
   bookService.webServiceURL = "http://www.amazon.com"
}
```

这样的重写被应用在 Spring 的 ApplicationContext 被构造之前，其格式是：

```
[bean name].[property name] = [value]
```

当然了，你也可以提供常规 Java 属性文件的配置，只不过需要每次都要带一个前缀 beans

```
beans.bookService.webServiceURL=http://www.amazon.com
```

15. Grails 和 Hibernate

如果 GORM （Grails 对象关系映射）对你来说还不够灵活，那你依然可以使用 Hibernate 来映射你的领域 类。为此你只需在工程目录的 grails-app/conf/hibernate 中创建 hibernate.cfg.xml 文件和相应的 HBM 映射文件。
更多 Hibernate 映射信息请参考 Hibernate 官方网站的关于映射的文档

这允许你将 Grails 的领域类映射到范围更广的遗留系统而且也使你的数据库（database schema）创建更灵活。

Grails 也允许你用 Java 来实现你的领域模型 或者重用 Hibernate 映射过的领域模型，你所需要做的仅仅是将必要的 hibernate.cfg.xml 和相应的映射文件放到 grails-app/conf/hibernate 目录下。

另外一个好处是：你依然可以调用 GORM 中的所有动态持久化和查询的方法。

15.1 通过 Hibernate 注释进行映射

Grails 也支持基于 Java5 注解创建的 Hibernate 领域类。为了让 Grails 知道你正在使用注解，你需要在数据源配置中设置 configClass 属性，如下所示：

```
import
org.codehaus.groovy.grails.orm.hibernate.cfg.GrailsAnnotationConfigur
ation
dataSource {
        configClass = GrailsAnnotationConfiguration.class
   … // remaining properties
```

}

配置就是这么简单！不过因为注解需要，请确认你已经安装了 Java5。现在为了创 建一个基于注解的类，只需要在 src/java 简单的创建一个新的 Java 类就可以了，当然了，要使用 EJB3 规范的注解（更多信息请参考 Hibernate 注解文档）

```
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;

    }

}
```

一旦定义完毕，别忘了在 Hibernate 的 sessionFactory 中注册相应的类，在 Grails 中，你需要将相应的实体加到 grails-app/conf/hibernate/hibernate.cfg.xml 配置文件中，如下代码所示：

```
<!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 加载的时候，将会为这些领域类注册必要的动态方法。要了解 Hibernate 领域类还能做哪些，请 参考脚手架相关章节。

15.2 深入了解

Jason Rudolph 作为 Grails 的提交者，花时间写了许多关于 Grails 和自定义 Hibernate 映射的文章，如下所示：

- 在遗留数据库中使用 Grails - 一篇关于如何在 Grails 使用 Hibernate XML 的很精彩的文章。
- Grails 和 EJB3 领域模型 - 另一篇关于在 Grails 中使用 EJB3 风格注释的领域模型的非常好的文章。

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：

- 必要的视图
- 控制器的创建/读取/更新/删除（CRUD）操作

让脚手架生效

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，如以下代码所示：

```
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，那如果想要让脚手架对特定的领域类也有效，该怎么办呢？很简单，直接将特定的领域类赋值给 scaffold 属性就好了，如下代码所示：

```
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，只需在浏览器地址栏简单输入 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
```