



Scott Davis
Jason Rudolph 著

胡键 译

Grails 入门指南

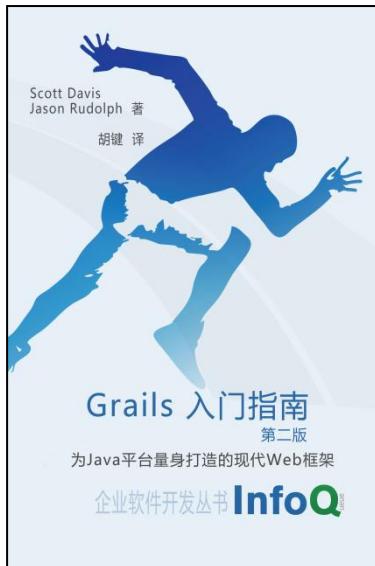
第二版

为Java平台量身打造的现代Web框架

企业软件开发丛书 **InfoQ**
ueue

免费在线版本

(非印刷免费在线版)



了解本书更多信息请登录[本书的官方网站](#)

InfoQ 中文站出品



本书由 InfoQ 中文站免费发放，如果您从其他渠道获取本书，请注册 InfoQ 中文站以支持作者和出版商，并免费下载更多 InfoQ 企业软件开发系列图书。

本迷你书主页为

<http://www.infoq.com/cn/minibooks/grails-getting-started-ii>

Grails 入门指南

第 2 版

Scott Davis、Jason Rudolph 著

胡键译

译者序

本书的第一版是我的 Grails 入门书。在快速浏览完整本书之后，当时就刺激得我想把平常使用的那些框架工具给扔到九霄云外。Grails 的出现，作为当今 Java 社区最让人激动的事件之一，本书的第一版已经给出了充分的解释。

从那之后，只要有机会，我就尝试着使用Groovy和Grails来解决问题。在Grails GAE插件推出的时候，我甚至还编写了一个Groovy的Web Console：GroovyLive（<http://trygroovy.appspot.com>），它的独特之处在于，内置了一个交互式的Groovy初级教程，其形式受到了Ruby社区的TryRuby的影响。

"寻觅就此结束 (The search is over)"，这是 Grails 官方网站上的宣传语。其中固然有些噱头的成分，但并非夸大其词。MVC、ORM、拦截器、验证、事务、标签库、URL Mapping、日志、i18n 等这些日常开发中最常用到的特性，在使用 Grails 时都可以拿来就用，不必经过大量繁琐的配置和准备。还有测试，Grails 提供了大量的基础设施来简化测试代码的编写，这无疑为贯彻 TDD 提供了有力的支持。

要是仅限于以上特点，那 Grails 的能耐也就不过如此了。在当今的时代，一个框架如果不支持插件，它的生命力必然有限，最终会很快地被新秀取代。插件架构，这是 Grails 的一大特点，也正是这一特点使得 Grails 得以借助整个社区的力量来不断扩充完善，长久发展。目前，Grails 的插件个数已经超过 400，这对 Grails 的使用者来讲显然是一笔巨大的财富。

本书第一版在 InfoQ 上发布距今已经过了快 3 年的时间，这期间 Grails 自身已经发生了翻天覆地的变化。想想看，当时书中所用到的版本还是 Grails 0.3.1，而此时 Grails 1.3 RC2 已经发布。按照 InfoQ 中文站的发布流程，估计 Grails 1.3 正式版都已经可以下载了。如此巨大的差异当然值得专门写一本书，于是，有了第二版。在这一版中，使用的 Grails 版本是 1.2。

出于对 Grails 的热爱，在 InfoQ 英文站上看到这一版的预告后，我便迫不及待地向中文站负责文章和迷你书发布的刘申同学"预约"，同时在正式版出来的第一时间就在我们编辑的邮件组中"夺过"本书的翻译权。在此，还要感谢各位编辑的割爱。

关于第二版的内容和优点，这里就不必多言了，基本延续了第一版的风格。这里只是提醒那些对Grails感兴趣、有意尝试的同学，多做多练才是用好它的王道。当然，还不要忘了查阅参考文档。从Grails的邮件列表来看，很多问题都可以归结为对文档的阅读理解不

够。这里，我也顺便做一个小广告。我在GroovyQ (<http://www.groovyq.net>) 上发布了Grails 1.2 的文档速读系列 (<http://www.groovyq.net/node/152>)，相当于参考文档的缩写，有兴趣的读者不妨前往一读。

最后，感谢晁晓娟对本书的审校，同时也感谢我的家人对我工作的支持。如果您发现了译文中的错误和纰漏，请直接给我发邮件 (jianhgreat@hotmail.com) 或到InfoQ中文站进行交流。如果您对Groovy有兴趣，欢迎访问GroovyQ，这个专注于Groovy社区动态，分享Groovy开发经验的站点。

胡键

2010-5-1

Scott的致谢

(第 2 版)

Jason Rudolph 在本书的第一版完成了一项杰出的工作。完全道出了我学习 Grails 的历程。这本书的快节奏--从不在任何主题上做过多的停留，但却涵盖了创建一个完整可运行应用所需了解的全部内容--同那些又臭又长一般有 1000 来页的技术书籍相比，无疑有一种让人如释重负般的舒爽。

随着我和他这么多年彼此交情日深--一直以来我们都在同一会议上演讲，时不时还小酌几杯--我总是不厌其烦地问他同一个问题：“你打算什么时候出《Grails 入门指南》的第二版？伙计，你必须完成它……”我成为这一版合著者的事充分说明了 Jason 的业务头脑--永远不要对一个渴望并且给你买啤酒的人说“不”。多谢，Jason，感谢你让我能把你的宝贝带到 Grails 的摩登时代。

同时也要感谢 Graeme Rocher、Jeff Brown，以及核心 Grails 开发团队的其他成员。在如此短的时间内，从开源项目到成立如 G2One 这样的公司，再到被 SpringSource 收购（接着它又被 VMWare 收购了），其价值一定是难以置信--象 codehaus.org 上的邮件列表流量第一或是在一个月内达到 90,000 的下载量都不足以体现。你们的努力不只是搞出了一个 Web 框架，而且还建立了一个社区。干得漂亮，兄弟们。

说到搭档，那些来自巡回会议的哥们不断地给予我启迪和增强我的信心。Neal Ford、Stuart Halloway、Venkat Subramaniam、David Geary、Andy Glover、David Bock、Brian Goetz、David Hussman、Ted Neward、Michael Nygard、Mark Richards、Brian Sam-Bodden、Nate Shutta、Ken Sipe 和 Brian Sletten--感谢深沉的夜晚、机场的同志友情，以及每个星期天的墨西哥式自助餐。在一个疯狂的世界里，唯有疯子才是正常人。（奥斯卡·王尔德）

感谢 Floyd Marinescu、Alexandru Popescu、John-Paul Grabowski，以及 InfoQ 团队的其他成员，你们让本书的第二版成为了现实。

最后，按惯例，感谢长期以来一直容忍我的妻子 Kim。她是这样的人，一边会问“你是不是该写另一本书了？”，一边又会因为来年让它霸占了维持一个正常家庭生活的一切而非常后悔。同样也是她，还说出了这样的话：“你是不是该把 MacBook Pro 扔到一边，陪陪 Christopher 和 Libby 了？”她似乎总是知道在合适的时候说合适的话。

Jason的致谢

(第 1 版)

首先我要感谢 Graeme Rocher (Grails 项目领导者 , 《The Definitive Guide to Grails》的作者) 。感谢您审阅了这本书 , 并在路上启发我深入 Grails 的内部工作原理。您细心审阅了本书 , 并提出了卓越的见解及宝贵建议令各个实例更加 "Groovy" 。

我还要感谢 Venkat Subramaniam (《Practices of an Agile Developer》的合著者) , 在审阅此书过程中强调学习的体验及如何最好地把 Grails 介绍给读者。此外 , 您具有深刻见解的序言 , 能帮助开发人员愉快接受一个可带来许多好处并将敏捷发扬光大的框架。

谢谢 Steve Rollins 勤勉地在这本书中倾注心血 , —— 解决残留的问题 , 哪怕这意味着在几个星期里 , 正常工作时间外默默无闻的付出。您关注细节而不知疲倦 , 成果都清楚地反映在这本书上了。

此外 , 我还要感谢 Jared Richardson (《Ship it! A Practical Guide to Successful Software Projects》的合著者) , 不仅感谢他审阅了本书 , 还要感谢他最早鼓励我写这本书。非常感谢在我写书过程中您给予的鼓励 , 以及给最后书稿的可贵观点。

感谢 Floyd Marinescu (InfoQ.com 的共同创始人之一 , 《EJB Design Patterns》的作者) , 以及为出版此书作出努力的整个 InfoQ 团队 , 感谢你们在出版过程中的热情支持。

最要感谢的是永远耐心、一直鼓励我的妻子 Michelle , 感谢你一路对我的支持。只有你 (在一切事情上都) 愿意承担远远超过你本应该承担的责任 , 我才有时间来做这件事情。你不但支持我完成了这本书 , 而且你的创造力、美感和在内容编辑上的建议 , 使本书比它原本的样子更好。

目录

译者序.....	III
SCOTT的致谢（第二版）	V
JASON的致谢（第一版）	VI
1 简介	1
例程学习	1
RACETRACK应用	1
2 安装GRAILS	3
安装JDK.....	3
安装GRAILS.....	3
安装数据库.....	3
3 创建GRAILS应用程序.....	4
创建RACETRACK应用.....	4
GRAILS应用目录结构	6
领域类.....	8
使用脚手架创建控制器和视图.....	11
4 验证	17
自定义字段顺序.....	17
增加验证.....	19
改变错误消息.....	22
创建自定义验证.....	24
测试验证.....	27
5 关系	33
创建“一对多”关系.....	33
创建“多对多”关系.....	36
启动时初始化数据.....	40
6 数据库	45
GORM	45
DATASOURCE.GROOVY	45
切换到外部数据库.....	47
7 控制器	52

比较CREATE-CONTROLLER和GENERATE-CONTROLLER.....	52
理解URL和控制器	53
从请求到控制器再到视图.....	54
GSP速览	55
了解控制器ACTION的其余内容	56
展示不匹配ACTION名字的视图	57
8 GROOVY服务器页面	59
理解GSP	59
理解SITEMESH	62
理解局部模板.....	63
理解自定义标签库.....	67
自定义缺省模板.....	69
9 安全	75
实现用户认证.....	75
对控制器进行单元测试.....	81
创建口令编解码器 (CODEC)	88
创建认证标签库.....	91
利用BEFOREINTERCEPTOR.....	92
利用过滤器 (FILTER)	95
安全插件.....	96
10 插件、服务及部署.....	97
理解插件.....	97
安装SEARCHABLE插件	97
探索SEARCHABLE插件	101
理解服务.....	102
增加搜索框.....	103
使用URL映射 (URLMAPPING) 改变主页	108
产品部署检查清单.....	110
总结	113
作者简介	115



欲知布丁味，必待亲口尝。一叶落而知天下秋。

- Miguel de Cervantes Saavedra

1 简介

Grails 是一个注重成效的开源 Web 应用框架。它使用了大多数 Java 开发者已经正在使用的最佳技术——最著名的当属 Spring 和 Hibernate——但是，Grails 并非只是它们的简单堆砌。

从输入 `grails create-app` 的那一刻，你就可以看出这不同于你平时的 Java 开发项目。每一种事物在 Grails 中都有其相应位置的这一事实——每一个新增的组件已经有一个相应的位置正等着它——让 Grails 有一种奇怪而又熟悉的感觉，哪怕你是第一次用它。只是在事后，你才意识到，把时间都主要花在了解决业务问题而非软件问题上。

例程学习

本书通过例子来介绍 Grails。你会看到从头构建一个 Grails 应用会有多快，同时了解如何对其进行自定义以满足不同需求。

要理解这些内容，你得需要点面向对象编程和 MVC Web 应用开发的基础知识。虽然阅读本书并不要求熟悉 Java，但如果熟悉的话，你肯定能从中受益。

你还会看到这些例子大量用到了 Groovy。但本书并没有打算教授 Groovy，凡是有点编程背景的人都应该能够看懂它们。

RaceTrack 应用

贯穿本书，你会在构建一个叫 RaceTrack 的小型 Web 应用的过程中探索到 Grails 开发的各个方面。在美国东南部有一个地区性的赛跑俱乐部，他们如今仍然还在使用纸张来跟踪俱乐部参与的比赛和每次赛跑注册的选手。他们准备跨入数字时代，我们将帮助他们迅速地完成这一跨越。

一开始，他们想要一个俱乐部工作人员可以管理赛跑和注册信息的应用程序。他们向我们保证不会需要任何花哨的东西——别忘了，他们现在还在用纸张进行管理——因此，要是有一个应用程序可以提供对这些数据进行内部管理的界面，他们会很高兴。

开发 RaceTrack 应用让你有机会亲自动手对 Grails 进行全面探索。你将创建 Web 用户界面、管理数据表之间的关系，应用验证逻辑，以及开发自定义查询。随着应用功能的扩展，你还会探索自定义标签库、Java 集成、安全、页面布局，以及动态方法的威力。在结束之前，你还会了解单元测试和部署。

基本上，你构建应用需要的信息和源代码都包含在本书的字里行间。你可以下载完整的应用，但我还是强烈建议你在阅读本书的过程中亲手构建一个。相信我说的，由于 Groovy 的强大和简洁，这个应用你完全可以轻而易举地搞定。

可以从源代码库：<http://github.com/scottdavis99/gswg-v2/> 检出 Racetrack 应用的源代码。
Git 的 安 装 地 址 : <http://git-scm.com> 。 安 装 Git 之 后 , 输 入 : git
clone <http://github.com/scottdavis99/gswg-v2.git>。

2 安装Grails

安装JDK

要让本书的例子正常运行，你至少需要安装JDK 1.5。花点时间下载并安装它，地址：<http://java.sun.com/javase/downloads>。完事之后，把 JAVA_HOME 环境变量指向你的 JDK 安装路径。

输入 `java -version` 验证安装是否正确。

安装Grails

接下来，下载最新的Grails版本，地址：<http://grails.org/Download>。（本书使用的是Grails 1.2。）。

如果你能从无到有地安装上 JDK，那么没理由不能安装好 Grails：

1. 解压Grails（提示：确保所有路径名中间没有空格。我的设置是：在 Unix/Linux/Mac OS X 机器用 /opt/grails；在 Windows 机器上用 c:\opt\grails。）
2. 创建 GRAILS_HOME 环境变量
3. 把 GRAILS_HOME/bin 加到 PATH 中。**(WINDOWS下注意\)**

输入 `grails`，确认 Grails 已经安装成功并可使用（想了解更多关于安装 Grails 的信息，请访问：<http://grails.org/Installation>。）

安装数据库

Grails 附带了一个内嵌式的 HSQLDB，它是一个纯 Java 的关系数据库。HSQLDB 非常适合用于快速开发应用演示，但你终归会在某个时候升级到一个功能齐全的数据库。既然 Grails 对象关系映射（GORM）API 是建立在 Hibernate 之上一层薄薄的 Groovy 门面（façade），因而任何拥有 JDBC 驱动程序及 Hibernate 方言的数据库，它都支持。

我们将在本书的后面使用 MySQL，但要是你已经安装配置了其他数据库，使用它们也没有关系。你可以下载一份免费的 MySQL 社区版，地址：<http://dev.mysql.com/downloads/>。

3 创建Grails应用程序

上一章，你已经安装好了 Grails。在这一章，你将创建你的第一个 Grails 应用程序。

你会看到 Grails 的脚手架是怎样在短时间内帮你组织应用结构并使之运行起来——从设置基础的目录结构到创建具备基本增/读/改/删（CRUD）功能的 Web 页面。整个过程中，你将学到如何改变 Grails 运行的端口，了解 Grails 应用的基础组成部分（领域类、控制器和视图）、指定字段的缺省值，以及其他许多内容。

创建RaceTrack应用

既然 Grails 已经安装完毕，那就为你的 Web 应用创建一个目录吧。

```
$ mkdir web  
$ cd web
```

所有现代 Java IDE 都提供了对 Groovy 和 Grails 的支持：IntelliJ、NetBeans 和 Eclipse。像 TextMate、vi 和 Emacs 这类的文本编辑器也都有针对 Groovy 和 Grails 的插件。为了避免陷入以上任何流行应用的细节，我们在此将坚持使用命令行界面。

要创建你的第一个 Grails 应用的结构，请输入 grails create-app racetrack。

```
$ grails create-app racetrack  
Welcome to Grails 1.2 - http://grails.org/  
Licensed under Apache Standard License 2.0  
Grails home is set to: /opt/grails  
...  
Created Grails Application at /web/racetrack
```

不出意外，你应该会看到 Grails 在为你的新应用创建基础目录结构过程中的一系列输出。

尽管尚未创建任何领域类或 Web 页面，但这并不妨碍我们启动应用来对其进行一次快速而全面的检查。进入 racetrack 目录，输入 grails run-app。

```
$ cd racetrack  
$ grails run-app  
...  
Base Directory: /Users/sdavis/web/racetrack  
Running script /opt/grails/scripts/RunApp.groovy  
Environment set to development  
[mkdir] Created dir: /Users/sdavis/.grails/1.2/projects/racetrack/classes  
[groovyc] Compiling 6 source files to /Users/sdavis/.grails/1.2/projects/racetrack/classes  
...
```

Running Grails application..

Server running. Browse to <http://localhost:8080/racetrack>

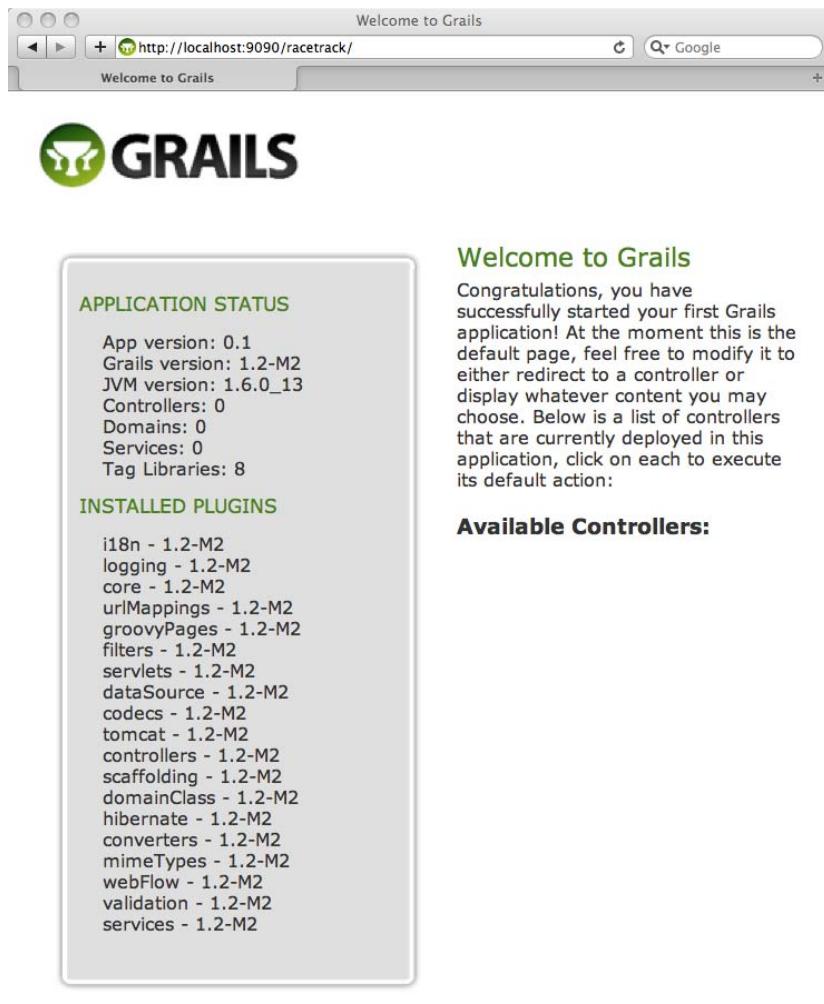
一切顺利的话，你应该可以在浏览器中访问 <http://localhost:8080/racetrack/> 并看到欢迎页面。反之，如果已经有一个服务运行于 8080 端口，那你很可能就会看到问候你的如下错误消息：

```
Server failed to start: LifecycleException: Protocol handler initialization failed:  
java.net.BindException: Address already in use<null>:8080
```

由于已经有一个 Tomcat 实例运行在系统的 8080 端口，为了避免以后发生任何端口冲突，我打算把运行端口改为 9090。你可以在\$GRAILS_HOME/scripts/_GrailsSettings.groovy 中永久性地改变这个值。你也可以在每次运行时通过在命令行中覆盖这个值来进行设置：

```
$ grails -Dserver.port=9090 run-app
```

现在，Grails 既然运行在了一个不会发生冲突的端口上，我应该可以看到那个欢迎页面了。



The screenshot shows a web browser window with the following details:

- Address Bar:** Shows the URL <http://localhost:9090/racetrack/>.
- Page Title:** "Welcome to Grails".
- Content Area:**
 - GRAILS Logo:** A green circular icon with a white stylized 'G' shape.
 - Welcome to Grails:** A heading followed by a paragraph of text: "Congratulations, you have successfully started your first Grails application! At the moment this is the default page, feel free to modify it to either redirect to a controller or display whatever content you may choose. Below is a list of controllers that are currently deployed in this application, click on each to execute its default action:"
 - APPLICATION STATUS:** A table-like section with the following data:

App version:	0.1
Grails version:	1.2-M2
JVM version:	1.6.0_13
Controllers:	0
Domains:	0
Services:	0
Tag Libraries:	8
 - INSTALLED PLUGINS:** A list of installed plugins and their versions:
 - i18n - 1.2-M2
 - logging - 1.2-M2
 - core - 1.2-M2
 - urlMappings - 1.2-M2
 - groovyPages - 1.2-M2
 - filters - 1.2-M2
 - servlets - 1.2-M2
 - dataSource - 1.2-M2
 - codecs - 1.2-M2
 - tomcat - 1.2-M2
 - controllers - 1.2-M2
 - scaffolding - 1.2-M2
 - domainClass - 1.2-M2
 - hibernate - 1.2-M2
 - converters - 1.2-M2
 - mimeTypes - 1.2-M2
 - webFlow - 1.2-M2
 - validation - 1.2-M2
 - services - 1.2-M2
 - Available Controllers:** A section listing controllers: "racetrack".

在动手实际创建 RaceTrack 应用之前，你还应该更多地了解一些内务管理方面的细节。再仔细看看 grails run-app 的输出，它充斥着对你的主（Home）目录中.grails/1.2/projects/racetrack 子目录的引用。这是 Grails 保存所有临时文件的位置。看到目录名前的那个点了没有？这能让它在像 Unix 这样的操作系统中隐藏起来。（这些 Grails 开发者还真是藏了一手。）

在本书后面的章节，你将学到使用 grails war 把所有东西打成一个整洁的包，这是为了将你的应用可以部署到 Tomcat、JBoss 以及其他任何标准 Java 企业版（JEE）应用服务器中去。在此之前，该目录（注：.grails/1.2/projects/racetrack）便是你运行代码以及可以找到所产生代码的位置。一条便捷的 grails clean 命令即可删除这个目录，或者你也可以象对待其他任何目录那样直接删除它。你可以定期这样做，不必有什么后顾之忧——每次输入 grails run-app，它都会被重新创建，使用最新的编译代码重新产生。

Grails 应用目录结构

现在，你已经对 Grails 背后的机制有了更好的理解，让我们走近瞧瞧 Grails 应用的组成部分。

Grails 非常强调**惯例优于配置**（convention over configuration）。这意味着 Grails 并不是靠**配置**（configuration）文件来把应用各部分组织在一起，相反，它靠的是**惯例**（convention）。所有领域类都存放在 domain 目录。控制器保存在 controllers 目录，视图则待在 views 目录，等等诸如此类。由于每一种东西都已经有了一个预定义的存储位置，你完全避免了配置的书写，哪怕是一行。

图 3-1 描述了 Grails 应用的组织结构。

racetrack	
grails-app	
conf	配置文件（如数据源、URL 映射、遗留的 Spring 和 Hibernate 配置文件）
controllers	控制器（MVC 中的“C”）
domain	领域类（MVC 中的模型或“M”。该目录中的每个文件在数据库中都有对应的表。）
i18n	资源包（用于错误消息和页面标识的国际化）
services	服务类（用于可能跨多个领域类的业务逻辑）

	辑)
taglib	自定义标签库 (用于构建 GSP 页面的可重用元素)
utils	自定义脚本 (存放像编解码器 (Codec) 这样的其他工具)
views	Groovy 服务器页面 (GSP) (MVC 中的 “V”)
lib	JAR (存放 JDBC 驱动包和第三方类库)
scripts	Gant 脚本 (存放项目特定的脚本)
src	
groovy	通用 Groovy 源文件 (存放其他那些没有惯例位置的文件)
java	Java 源文件 (用于存放遗留的 Java 代码) 。 (该目录中的文件将被编译并包含到 WAR 文件中。)
test	
integration	依赖其他组件 (如数据库) 的测试脚本
unit	测试隔离组件的测试脚本
web-app	
css	层叠样式表 (CSS)
images	图像文件 (JPG、GIF、PNG 等)
js	JavaScript 文件 (包括像 Prototype 、 script.aculo.us 、 YUI 等这样的第三方库。)
META-INF	典型的 JEE 目录 , 用于存放 manifest 文件
WEB-INF	典型的 JEE 目录 , 用于存放 web.xml 和其他配置文件

图 3-1 : 应用的目录结构

领域类

领域类是 Grails 应用的生命血液。简单的讲，它们定义了你打算跟踪的“东西”。

Grails 接受这些简单的类，并利用它们完成许多工作。相应的数据库表会自动为每个领域类创建。控制器和视图会从关联的领域类中派生出名字。领域类还是存放验证规则、定义“一对多”关系，以及包含其他许多信息的地方。

当然，典型的最终用户可能也会按领域类来描述应用。“瞧瞧，我们需要跟踪比赛（Race）、注册信息（Registration）、还有……”

观察一下该跑步俱乐部用来记录每次比赛的纸质表格：

Southeastern Regional Running Club								
Race Name:	TURKEY TROT	Distance:	5K					
Date:	Nov 22, 2007	Time:	9:00 AM					
Location:	DUCK, NC							
Maximum # Runners:	350	Cost:	\$20.00					
Registered Runners:								
Date	Name	Address	E-Mail	Gender	DoB			
JUNE 1, 2007	JANE DOE	1234 ROCKY ROAD SOMWHERE, NC 12345	jane@doe.com	F	FEB 1, 1975			
JUNE 3, 2007	JOHN DOE	567 SUNDAY DRIVE ELSE WHERE, NC 12345	john@doe.com	M	JAN 1, 1974			

图 3-2：目前以纸张为基础的流程中所用表格

你能看出比赛和注册信息吗？你能找出用于创建这两个领域类所需的字段吗？老练的软件开发者还可能在这两个实体之间建立“一对多”关系——一次比赛有多人注册。

让我们来创建这两个领域类。回到命令行提示符，输入 grails create-domain-class Race。（如果之前的应用还在运行，请先按 Control-C 停止它）。

```
$ grails create-domain-class Race
```

...

Created Domain Class for Race

Created Tests for Race

注意：为了简单起见，你可以无视关于未指定包名的警告。或者，你可以输入 grails create-domain-class com.acme.Race 继续，使用包名 com.acme。

你会发现 Grails 同时创建了一个领域类和一个测试类。我们会在本书的后面用到这些测试类。至于现在，让我们把焦点放在领域类身上。

使用文本编辑器打开 racetrack/grails-app/domain/Race.groovy。

```
class Race {
    static constraints = {}
}
```

嗯.....是不是没几行代码？让我们把从那张纸上搜刮到的属性加上：

```
class Race {
  String name
  Date startDate
  String city
  String state
  BigDecimal distance
  BigDecimal cost
  Integer maxRunners = 100000
  static constraints = {}
}
```

每个字段都有名字和 Java 数据类型。我们给 `maxRunners` 字段设置了一个缺省值：100,000。在后续章节，你将学到如何设置验证规则，其中有条规则可以为给定字段指定合法值的范围。但是眼下，这些已经足以让我们开个头了（现在不要去操心 `static constraints` 那行代码——你会在下一章知道它的作用。）

快速提示：Groovy 会自动把 10.5 这样的小数值自动装箱（Autobox）成 `java.math.BigDecimal` 而非你想当然的 `java.lang.Float` 或 `java.lang.Double` 类型。为什么要这样？举个大多数 Java 开发者都没有意识到的一个惨痛例子：写一个简短的 Java 应用，它循环 10 次，每次都加上 0.1。你最终会得到 0.99999 或 1.000001，这要看你是把和存成 `Double` 还是 `Float`。使用 `BigDecimal`，你将会每次都得到 1.0，和预期的一模一样。这就是那些酷小孩们说的“最小惊讶法则（The Principle of Least Surprise）。”

让我们创建另一个领域类。在命令行提示符下输入 `grails create-domain-class Registration`。给 `racetrack/grails-app/domain/Registration.groovy` 增加一些字段：

```
class Registration {
  String name
  Date dateOfBirth
  String gender
  String address
  String city
  String state
  String zipcode
  String email
  Date dateCreated //注意：这是一个特殊的名字

  static constraints = {}
}
```

除了最后一个字段，对其余的应该没什么好惊奇的。如果你定义了一个名为 `dateCreated` 的日期字段，Grails 将自动在第一次向数据库保存实例的时候填上这个值。要是你创建了

另一个名为 lastUpdated 的日期字段，Grails 将在每次把更新后的记录存回数据库的时候填充这个日期。

这个惯例可以轻易地通过配置关闭——只需简单地在你的类中增加 static mapping 代码块：

```
class Registration {  
    // ...  
  
    Date dateCreated  
    Date lastUpdated  
  
    static mapping = {  
        autoTimestamp false  
    }  
}
```

另一方面，倘若想让 Grails 完成一些更复杂的任务，而不是简单地填充一两个字段的时戳，你可以介入领域类的生命周期事件，方法是创建几个直观的命名闭包：

```
class Registration {  
    // ...  
    def beforeInsert = {  
        // your code goes here  
    }  
    def beforeUpdate = {  
        // your code goes here  
    }  
    def beforeDelete = {  
        // your code goes here  
    }  
    def onLoad = {  
        // your code goes here  
    }  
}
```

在技术上，static mapping 代码块可以用来把类名映射成另一个表名，把字段名映射成另一个列名，但是你还可以用它做一些其他的有趣事情。例如，要是想返回的 Race 列表按某一顺序排列，可以在 Race.groovy 中加入下列代码：

```
class Race {  
    static mapping = {  
        sort "startDate"  
    }  
    // ...  
}
```

关于 static mapping 代码块更多的信息，请参见：<http://grails.org/GORM+-+Mapping+DSL>。

但是，在学会走之前，先别忙着学跑。（对不起，丝毫没有双关语的意思。）你甚至还未曾看到这些简单类的实际使用。让我们现在马上创建一个控制器和一些视图，这样就能在浏览器中试着用用它们了。

使用脚手架创建控制器和视图

控制器和视图最终形成了 Grails 应用中的“3 要素（ Big Three ）”。毫无疑问，大多数 Web 开发者都非常熟悉模型/视图/控制器（ MVC ）模式。通过强行彻底地分离出在这三个元素各自的关注点，你努力的回报则是获得了最大限度的灵活性和重用。

我们前面定义的两个领域类——Race 和 Registration——跟数据的“哑”占位符没什么区别（这么说有一点过于简化了，但就目前来讲，它的确就是这样）。领域类并不关心它的存储方式和显示给最终用户的最终形式。用最简单的话来讲，控制器负责把数据从数据库中取出，创建新模型，最后把它们交给视图用于显示。

我们将在以后更深入地谈及如何自定义控制器的行为和视图的外观。至于现在，我们只让 Grails 处理把它们两个自动产生的必要细节。正如你将看到的，只需一行代码就能立刻得到缺省行为。

在命令行下，输入 grails create-controller Race。

```
$ grails create-controller Race  
...  
Created Controller for Race  
[mkdir] Created dir: /web/racetrack/grails-app/views/race  
Created Tests for Race
```

仔细看，除了创建 grails-app/controllers/RaceController.groovy，Grails 还创建了相应的测试（对于大多数命令，结果都是这样：create-service、create-tag-lib 等。）

还请留意，Grails 为 Race 类创建了一个空的视图目录。接下来，在你输入 grails generate-views 想自定义 Groovy 服务器页面（ GSP ）的时候，你就知道到哪儿去找这些页面了。

在文本编辑器中看看你的新控制器：

```
class RaceController {  
    def index = {}  
}
```

它就和最初的领域类一样，没多少内容，是不是？领域类和控制器之间有一个极大的区别。尽管领域类负责包容数据，但却是控制器指出了最终用户在 Web 浏览器中最后看到的 URL。此处空的 index 闭包就像其他 Web 框架中的 index.jsp 或 index.html 文件一样——它是从 Web 传入的 Race 请求的缺省目标。

要是愿意，你可以加入类似如下的代码：

```
class RaceController {
    def index = {
        render "Hello World"
    }
}
```

这将用所有软件开发者都耳熟能详的通用消息问候最终用户。当他们在浏览器中访问 <http://localhost:9090/racetrack/race> 时候，将看到“Hello World”。

你在这里定义的任何闭包都会被暴露成一个 URL。例如，你可能会通过显示“Bar”，给 <http://localhost:9090/racetrack/race/foo> 提供一个预期的响应：

```
class RaceController {
    def index = {
        render "Hello World"
    }

    def foo = {
        render "Bar"
    }
}
```

但是这让我们显得有点儿无聊，不是吗？让我们给这个练习加点有真正业务价值的内容。

先不要给 RaceController 加入自己的代码，而是把它改成下面的形式：

```
class RaceController {
    def scaffold = Race
}
```

这到底是在干什么？其实，正是这段代码提供了对 Race 类进行增/查/改/删（CRUD）的全部功能。当 Grails 看到控制器中的 scaffold 属性，它会**动态地**产生针对指定领域类的控制器逻辑和必要的视图。所有这些只有一行代码而已！

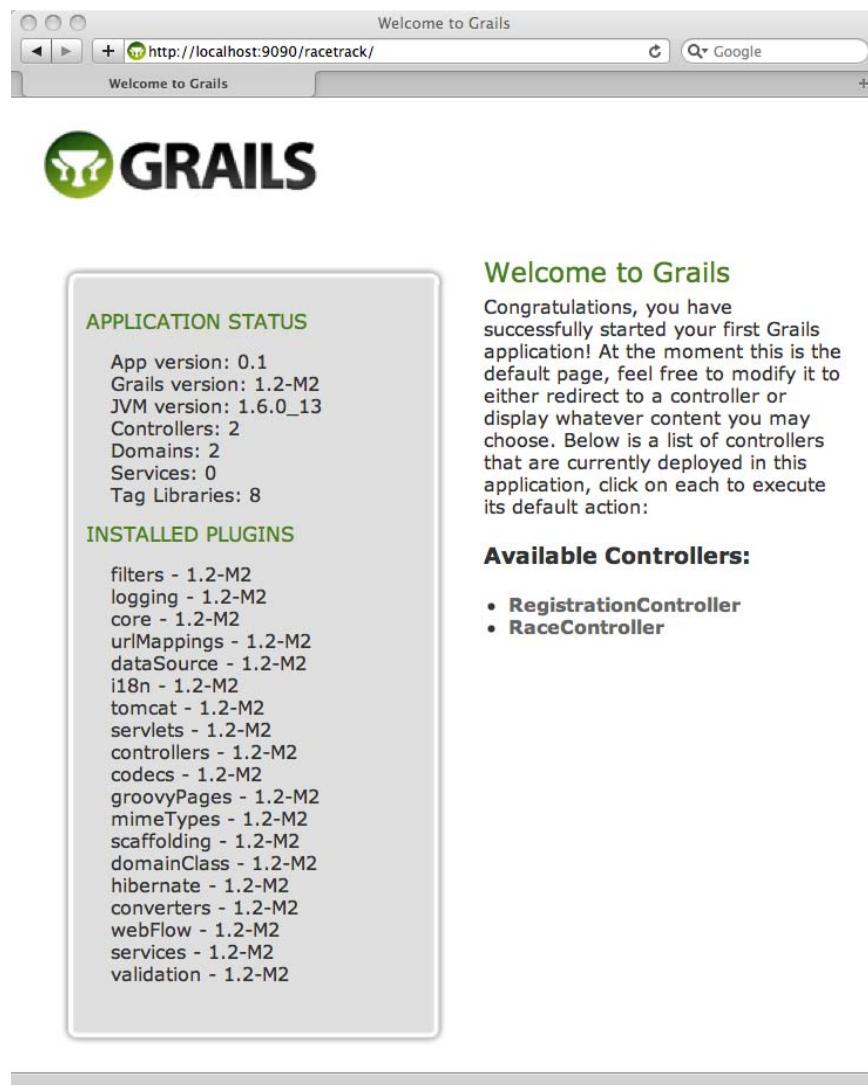
你可能还想看看原始 Scaffold 命令的更通用形式：

```
class RegistrationController {
    def scaffold = true
}
```

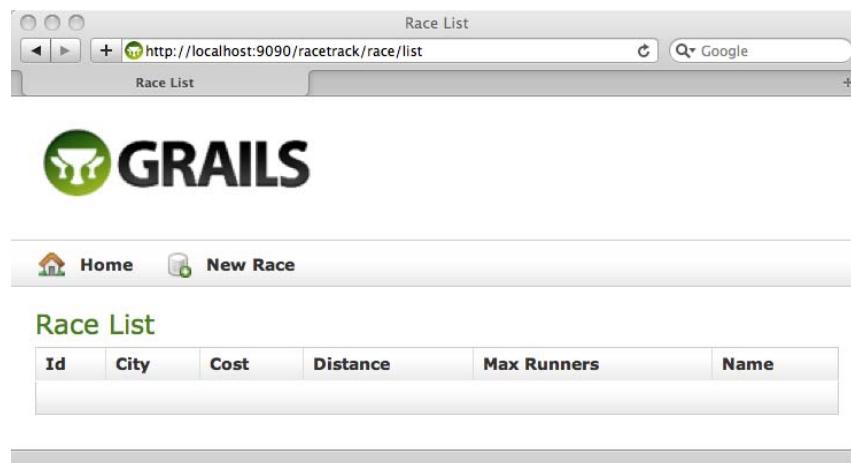
在你使用 true 而不是领域类名的时候，“惯例优于配置”再一次发挥了作用。Grails 会查看控制器的名字，找出使用哪个领域类来产生脚手架。

在 RaceController 和 RegistrationController 都就绪之后，让我们看看是否所有这些脚手架都实际符合我们的预期。仅仅指定领域类的字段和在控制器中加入一行代码，就真的能得到一个完整的 CRUD 应用？输入 grails run-app，然后访问 <http://localhost:9090/racetrack> 查个究竟吧。

你首先应该查看的是每个你创建的新控制器的链接。（只要看看“可用控制器”列表即可。）

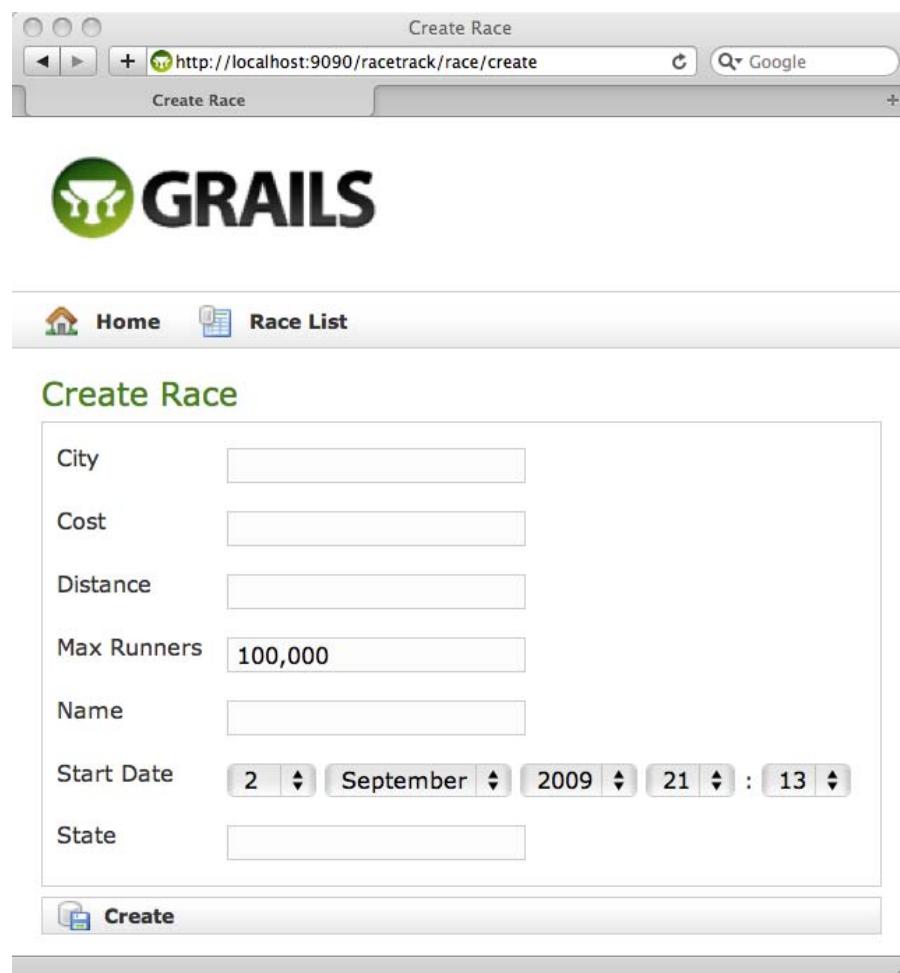


点击 RaceController 链接，可以看到现有比赛（Race）的列表。



Grails 显示了这个类的前 6 个字段，包括它自己创建的用于存放主键的 id。是否留意到了像 maxRunners 这样的驼峰字段名是如何针对最终用户进行美化的？

现在点击“创建比赛（ New Race ）”。



我们会在这个表单内看到该类的所有字段。瞧，maxRunners 的缺省值预先用我们在领域类中指定的值产生出来了。

继续摆弄你这个全新的 Web 应用吧——不要着急，我会在这儿等着你的。确认编辑和删除比赛（Race）的功能跟预想的一样，在你做这些的时候再创建一些新的注册信息（Registration）。

只是不要对你输入的任何信息都太过执着。一旦按下 Control-C，所有这些就都灰飞烟灭了。下次再输入 grails run-app，又会有一个全新的（没错，是空的）RaceTrack 应用等着你。

这到底是错误，还是特色？狡猾的问题：两者都不是；而是惯例！你是在开发（Development）模式下运行 Grails，这隐式意味着你想在每次运行之间都清除所有内容。除了该模式之外，还有测试（Test）和产品（Production）模式，每个都有自己的行为模式。输入 grails prod run-app，你的数据将会有个稍长的生命周期。（不必担心——grails war 会创建一个运行在产品模式下的 WAR 文件）你将在数据库一章中学到如何改变这种行为。

哦，字段的顺序（或者说根本无序）让你烦心了？我们将在下一章改变它。

如何可以让我们从 HSQLDB 切换到其他数据库？如何创建自定义验证？如何定义领域类之间的“一对多”关系？

所有这些都是好问题。我担保你得问题还不止这么点。但是，再次提醒，不要操之过急。你要做的是跑完全程，而现在，你才刚刚跨出起跑线。要做的事情多的是，但不要忘了你已经做过的。

在命令行中输入 grails stats：

```
$ grails stats
Welcome to Grails 1.2 - http://grails.org/
Licensed under Apache Standard License 2.0
Grails home is set to: /opt/grails

Base Directory: /web/racetrack
Running script /opt/grails/scripts/Stats.groovy
Environment set to development

+-----+-----+-----+
| Name           | Files | LOC   |
+-----+-----+-----+
| Controllers    |     2 |     6 |
| Domain Classes |     2 |    22 |
| Unit Tests     |     4 |    44 |
+-----+-----+-----+
| Totals         |     8 |    72 |
+-----+-----+-----+
```

不计单元测试（顺便说一句，要不了多久我们就会去写它了）你已经写了 28 行代码，分
布在 2 个控制器和 2 个领域类中。得到的回报是，你有了一个完整可以上线运行的 CRUD
应用。我得说你从你的投资上已经得到了相当不错的回报。同不同意？

4 验证

前一章，你已经了解了 Grails 应用的基本组成：领域类、控制器和视图。我们将在本章深入了解领域类。

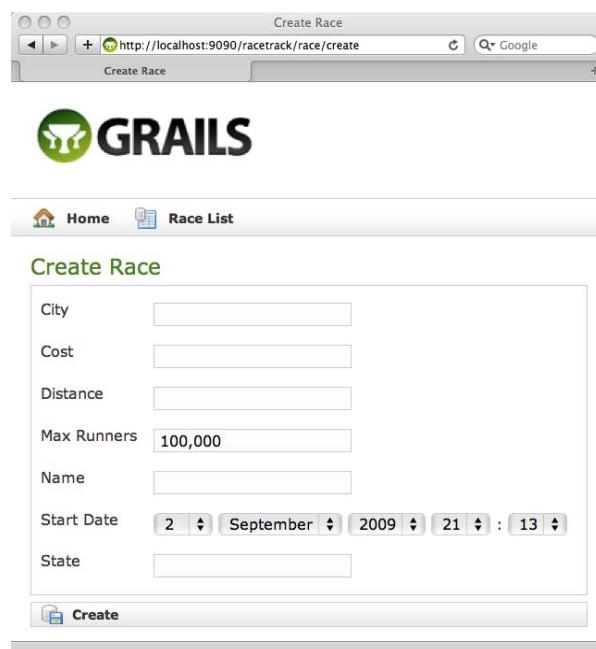
你会学到如何指定字段的顺序。你将探索 Grails 直接提供的验证选项，并创建自己的自定义验证。当然，在完成一个自定义验证的创建之后，你还将学到如何测试它。最后，你将看到如何改变你的应用给验证错误提供的错误消息。

自定义字段顺序

让我们回顾一下 Race 类。在文本编辑器内打开它。

```
class Race {
    String name
    Date startDate
    String city
    String state
    BigDecimal distance
    BigDecimal cost
    Integer maxRunners = 100000
    static constraints = {}
}
```

你是以特定顺序加入这些字段的——name、startDate、city……当看到 Grails 并没有按同样顺序显示这些字段的时候，你是否感到吃惊？

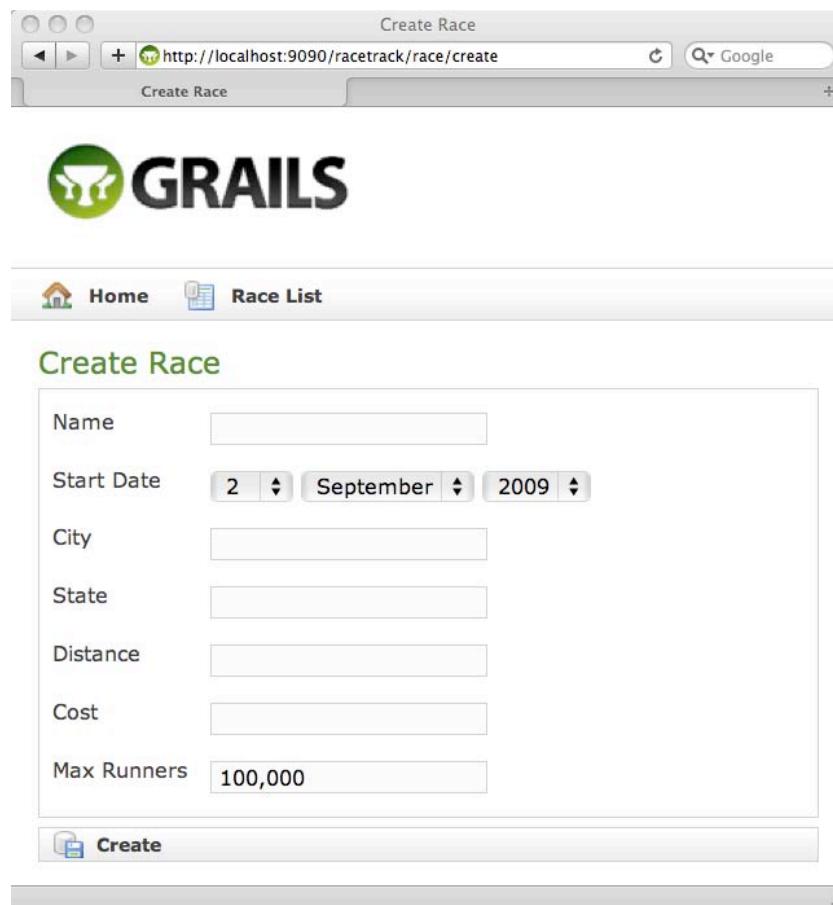


这完全是 Grails 底层的 Java 在作怪——在 Java 里并没有一个命令说，“给我按顺序显示那个类的字段。” Getter/Setter 范式 (Paradigm) 让 Java 类在概念上更接近 HashMap , 而不是 LinkedList。

所幸，Grails 可以很容易的说，“嗨，在你给我产生 Web 页面主框架的时候，请务必让这些字段按照这样一个顺序出现。”你给 Grails 的这个暗示是以 static constraints 代码块的形式出现的。

```
class Race {  
    static constraints = {  
        name()  
        startDate()  
        city()  
        state()  
        distance()  
        cost()  
        maxRunners()  
    }  
  
    String name  
    Date startDate  
    String city  
    String state  
    BigDecimal distance  
    BigDecimal cost  
    Integer maxRunners = 100000  
}
```

请输入 grails run-app , 然后访问 <http://localhost:9090/racetrack/race/create> , 验证这些字段是否是按正确顺序显示了。



The screenshot shows a web application interface for creating a race. At the top, there's a browser window with the title "Create Race" and the URL "http://localhost:9090/racetrack/race/create". Below the browser is the Grails application's header, featuring the "GRAILS" logo. A navigation bar includes links for "Home" and "Race List". The main content area is titled "Create Race" and contains several input fields: "Name" (empty), "Start Date" (set to "2 September 2009"), "City" (empty), "State" (empty), "Distance" (empty), "Cost" (empty), and "Max Runners" (set to "100,000"). At the bottom of the form is a "Create" button.

此刻，我猜你在想：“为什么我非得把所有事情都做两次？这难道不是违反了‘不要重复你自己（DRY）’这条原则吗？”。如果 static constraints 代码块就干了这么点工作，我完全同意你的观点。但事实上，字段顺序不仅仅只是自娱自乐这么简单。你做这些事情的真正原因是验证（或者说约束）这些字段。

增加验证

同比赛的组织者谈谈，很多约束就会从这次不经意的闲谈中冒出来：

- “比赛的名字不能超过 50 个字符，否则它就没法印在 T 恤和横幅上了。”
- “我们是地区性俱乐部。我们只参加乔治亚州、维吉尼亚州，以及卡罗莱纳州北部和南部的比赛。”
- “每场比赛的收费我们从不超过\$100，而且不少比赛还是免费的。”

身为一名经验丰富的程序员，你可能会提出一些最终用户也许从来没有考虑过的荒谬但必要的约束：

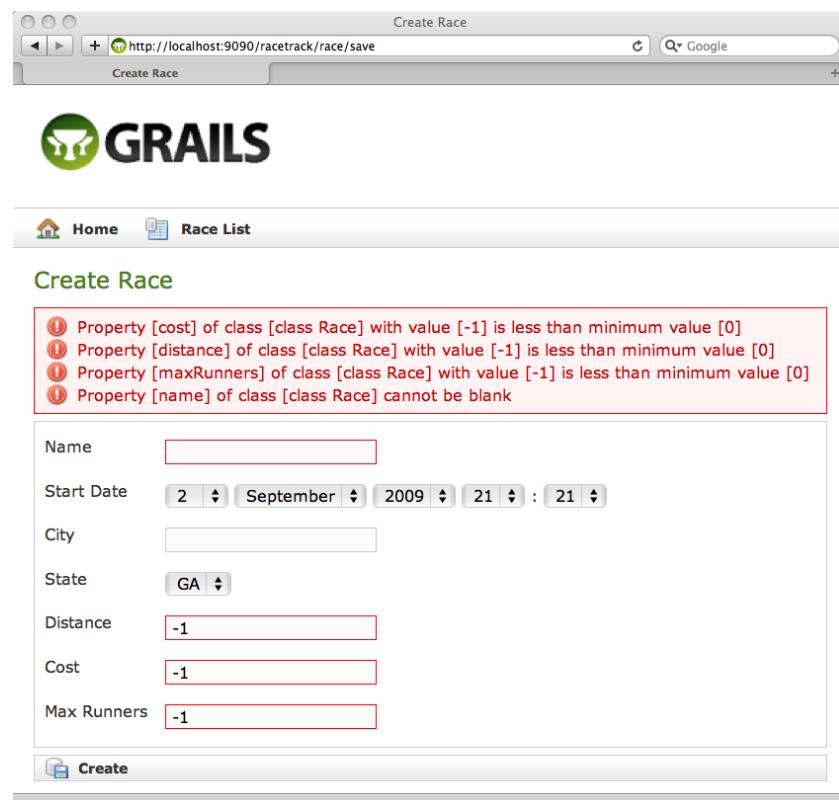
- “每个比赛都必须有名字。”
- “不存在距离为负数的比赛。”

掌握这些信息之后，让我们来给 Race 类增加一些有意义的约束：

```
class Race {
    static constraints = {
        name(blank:false, maxSize:50)
        startDate()
        city()
        state(inList:["GA", "NC", "SC", "VA"])
        distance(min:0.0)
        cost(min:0.0, max:100.0)
        maxRunners(min:0, max:100000)
    }

    String name
    Date startDate
    String city
    String state
    BigDecimal distance
    BigDecimal cost
    Integer maxRunners
}
```

现在，好戏开场了。如果试图违反自己的验证规则，你不会变得火冒三丈——你是要成为一名尽职、认真的软件工程师。输入一些你知道会触发验证错误的值（比方说，-1）。



哈，一片红色的海洋！在看到错误消息（预期的错误消息）的时候，只有开发人员会感到开心。

看起来验证起作用了。我们设置的某些约束——maxSize、inList——让使用者不可能提供伪造的值（假设他老老实实地待在浏览器划定的限制之内，不求助于任何形式的黑客手段）。此时，你不可能给名字字段输入超过 50 个字符或选择不属于这 4 个有效州的其他州。但是，你可以让字段空着，也可以在文本框中输入不正确的值。我们的验证规则保证了这些错误——不管是有意还是无意——不会污染数据库。

图 4-1 列出了 Grails 所有可用的验证选项。

约束	用法	描述
blank , nullable	blank:true nullable:true	blank 用于捕获 Web 层的空白； nullable 用户捕获数据库层的空白。
creditCard	creditCard:true	该约束保证了信用卡号码的内部一致性，基于 Apache Commons CreditCardValidator。
display	display:false	隐藏字段在 create.gsp 和 edit.gsp 中的显示。注意：一定要与 blank:true 和 nullable:true 同时使用，或者在控制器中产生这个值，否则缺省的验证将失败。
email	email:true	确保值匹配基本的模式：“user@somewhere.com”。
password	password:true	把视图由<input type="text">改为<input type="password">
inList	inList:["A", "B", "C"]	创建一个包含可能值的组合框
matches	matches:"[a-zA-Z]+"	允许你提供自己的正则表达式
min, max	min:0, max:100 min:0.0, max:100.0	用于数字并实现了 java.lang.Comparable 的类
minSize,	minSize:0,	用来限制字符串和数组的长度

maxSize, size	maxSize:100, size:0..100	
notEqual	notEqual:"Foo"	顾名思义，值不能和约束相等
range	range:0..10	创建一个组合框，其中每个值都在这个范围之内
scale	scale:2	设置小数点的位数
unique	unique:true	确保值并没有存在于数据库表中（这非常适合用于创建一个新的登录用户）
url	url:true	确保值匹配基本模式：“http://www.somewhere.com”
validator	validator: {return(it%2)==0}	允许你创建自己的自定义验证闭包

图 4-1 : Grails 的验证选项

改变错误消息

比赛的组织者看到我们为他们理清了他们的数据输入非常高兴，但其中有人说道，“听起来我象是挨了机器人的骂。”。所幸，跟一开始设置验证的活一样，改变缺省的错误消息也很容易。

Grails 把所有的错误消息都保存在 grails-app/i18n 目录下的 messages.properties 文件里。没错，这就是 Java 开发者习惯用来开发国际化应用的资源包（Resource Bundle）。如果查看 i18n 目录，你会发现标准的 Grails 错误消息已经被翻译成了德语、西班牙语、法语、意大利语、日语以及其他几个国家的语言。

```
default.blank.message=Property [{0}] of class [{1}] cannot be blank
```

再到 Web 浏览器中仔细看看这条错误消息，看起来 {0} 似乎是一个占位符，用于显示字段的名字。类名则保存在 {1}。

在更复杂的消息中，还有几个其他的占位符。例如，查看一下违反范围（Range）约束所对应的错误消息：

```
default.invalid.range.message=Property [{0}] of class [{1}] with value [{2}] does not fall
within the valid range from [{3}] to [{4}]
```

用户输入的值保存在 {2}。范围的起始值和结束值保存在 {3} 和 {4} 中。

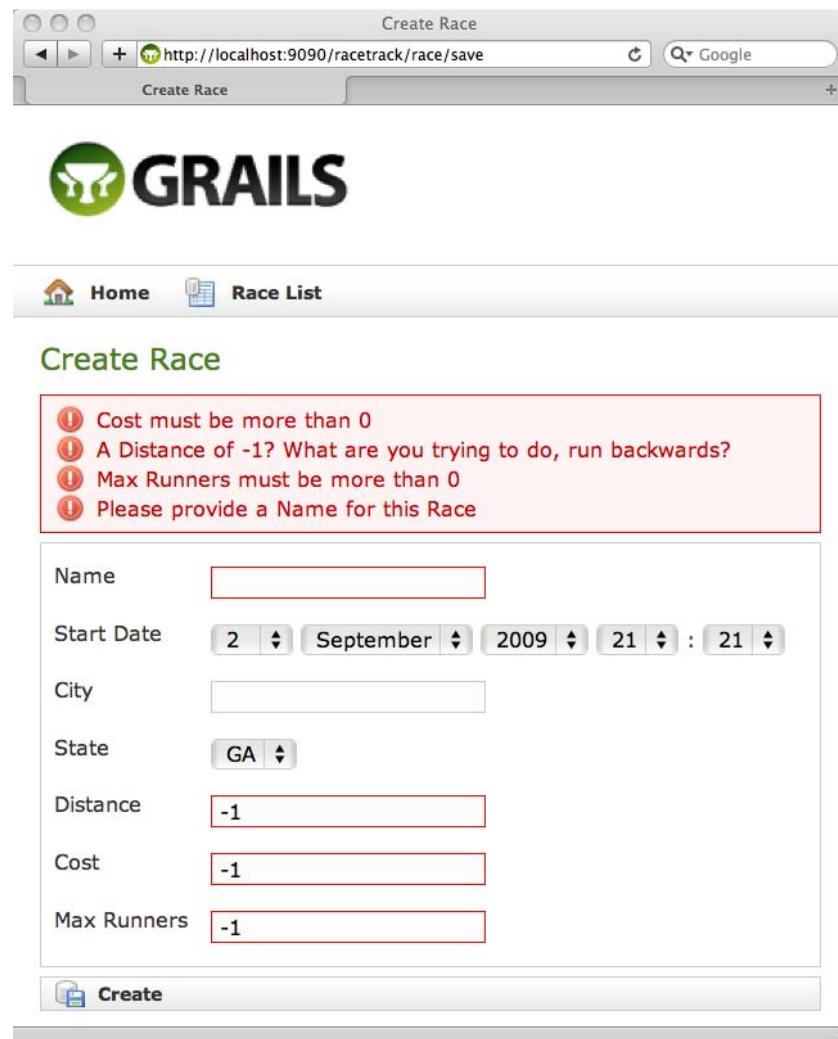
与其改变缺省的错误消息，我们为何不为每个类、每个字段设置一些自定义的错误消息呢？把以下内容加到 messages.properties 中。（注意，每条消息必须占一行。）

```

race.name.blank=Please provide a Name for this Race
race.name.maxSize.exceeded=Sorry, but a Race Name can't be more than {3} letters
race.distance.min.notmet=A Distance of {2}? What are you trying to do, run backwards?
race.maxRunners.min.notmet=Max Runners must be more than {3}
race.maxRunners.max.exceeded= Max Runners must be less than {3}
race.cost.min.notmet=Cost must be more than {3}
race.cost.max.exceeded=Cost must be less than {3}

```

试着再次在 Web 浏览器中保存你伪造的值。这次，应该不会再被“一个机器人责难”了：



The screenshot shows a web browser window with the title "Create Race". The address bar shows the URL "http://localhost:9090/racetrack/race/save". Below the address bar is a toolbar with a "Create Race" button. The main content area has a "GRAILS" logo at the top. Below it is a navigation bar with "Home" and "Race List" links. The main form is titled "Create Race". It contains several input fields: "Name" (empty), "Start Date" (set to September 21, 2009, 21:21), "City" (empty), "State" (set to GA), "Distance" (set to -1), "Cost" (set to -1), and "Max Runners" (set to -1). A red box highlights the error messages: "Cost must be more than 0", "A Distance of -1? What are you trying to do, run backwards?", "Max Runners must be more than 0", and "Please provide a Name for this Race". At the bottom of the form is a "Create" button.

图 4-2 列出了所有可用的验证错误消息。

约束	每个类、每个字段的消息
blank, nullable	ClassName.propertyName.blank ClassName.propertyName.nullable

creditCard	ClassName.propertyName.creditCard.invalid
display	N/A
email	ClassName.propertyName.email.invalid
password	ClassName.propertyName.password.invalid
inList	ClassName.propertyName.not.inList
matches	ClassName.propertyName.matches.invalid
min, max	ClassName.propertyName.min.notmet ClassName.propertyName.max.exceeded
minSize, maxSize, size	ClassName.propertyName.minSize.notmet ClassName.propertyName.maxSize.exceeded ClassName.propertyName.size.toosmall ClassName.propertyName.size.toobig
notEqual	ClassName.propertyName.notEqual
range	ClassName.propertyName.range.toosmall ClassName.propertyName.toobig
scale	N/A
unique	ClassName.propertyName.unique
url	ClassName.propertyName.url.invalid
validator	ClassName.propertyName.validator.invalid

图 4-2 : Grails 的验证错误消息

创建自定义验证

这次是一个有趣的挑战。比赛组织者每年 1 月都会被粗心的家伙随手写上去年的时间而不是今年的时间给惹毛。“你有没有办法保证让我们不会把比赛安排在过去的时间？这种问题把我们的帐簿弄得乱七八糟，让我们的会计成天冲我们喊叫。”

就你目前已经掌握的 Grails 验证知识，你可能会忍不住在 startDate 字段上使用 min 约束：

```
class Race {
  static constraints = {
    // ...
    // 注意：这并不会按你预想的那样工作
    startDate(min: new Date())
  }
}
```

```
// ...  
}  
  
Date startDate  
// ...  
}
```

虽然在理论上这是个好主意，但是它并不能按你预想的方式去工作。在这个例子中，`new Date()`的值是每次在重启服务器的时候设置，而不是在每次计算验证的时候。结果，这同我在 `distance`、`cost` 和 `maxRunners` 字段上使用如 `max` 和 `min` 硬编码一个固定的值没有多大区别。

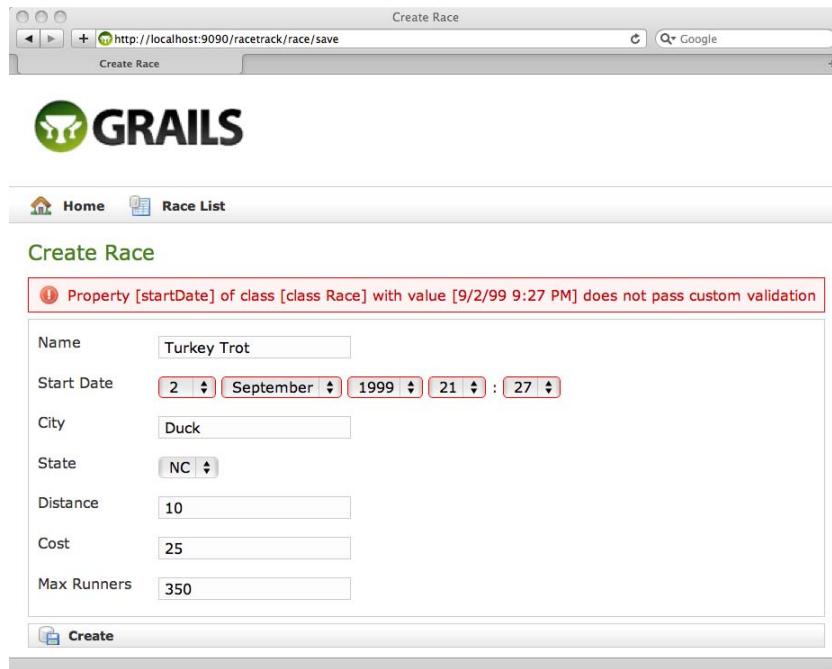
按照这样的方式思考：你要验证的是一个表达式的结果，而不是一个固定的值。要想达到这个目的，你需要创建一个自定义验证。以下代码将防止比赛组织者把比赛时间安排在过去：

```
class Race {  
    static constraints = {  
        // ...  
        startDate(validator: {return (it > new Date())})  
        // ...  
    }  
  
    Date startDate  
    // ...  
}
```

所有的自定义验证器都要返回一个布尔值。这个闭包可以视其返回 `true` 或 `false` 的需要随便使用多少行代码，只要它回答这个问题：“这是否有效？”

验证器闭包每次在其调用的时候被计算。这意味着 `new Date()` 每次都会是一个新值。在这里，`it` 并没有明确给出，它代表了用户试图提交的日期。

试着给比赛的开始日期输入一个过去的时间：



Create Race

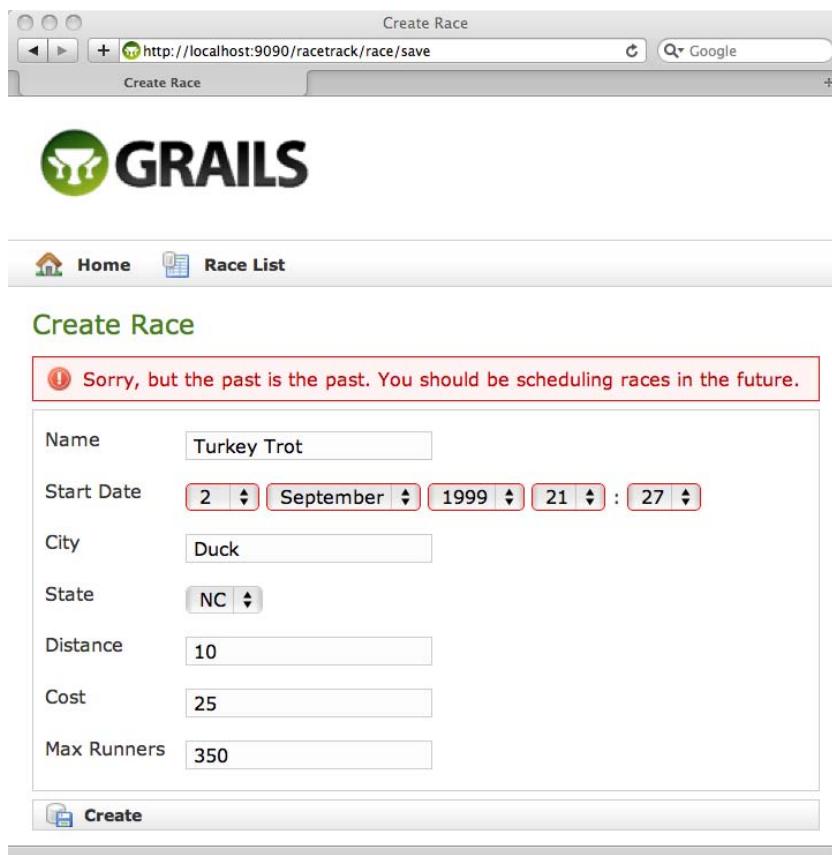
Property [startDate] of class [class Race] with value [9/2/99 9:27 PM] does not pass custom validation

Name	Turkey Trot
Start Date	2 September 1999 21:27
City	Duck
State	NC
Distance	10
Cost	25
Max Runners	350

一旦确信验证器生效，那就赶快修改一下 messages.properties (别忘了，所有这些都需要写在一行内。)

```
race.startDate.validator.invalid=Sorry, but the past is the past. You should be scheduling races in the future.
```

一个更糟的日期和一条更友好的错误消息：



Create Race

Sorry, but the past is the past. You should be scheduling races in the future.

Name	Turkey Trot
Start Date	2 September 1999 21:27
City	Duck
State	NC
Distance	10
Cost	25
Max Runners	350

测试验证

从我们编写 RaceTrack 应用开始，我就一直在叫嚣要写些测试。这个时候终于来了。

为什么是现在？就因为我们终于增加了一些值得测试的自定义功能。在“比赛不能发生在过去”故事之前的任何功能都只是简单地利用了 Grails 的现有功能。测试这些开箱即用的行为没多大价值——我们就把测试 Grails 内核的任务留给 Grails 开发团队吧。我们将专注于测试我们自己的代码。

到目前为止，我们做的测试都是“有样学样（monkey-see, monkey-do）”——输入错误的值，然后查看屏幕上的结果，这没有问题，但是，很难把你做的这些称为是严格的。当然，你应该用鼠标在运行的 Web 应用中点点这儿、点点那儿以完整地检查自己的工作，但作为程序员，你的工作并没有就此结束。老练的开发者知道最好避免跌入“但是它在我这儿是可以运行的”这一陷阱。

用代码写一个简短的测试可以立即给我们带来两个好处：你自动化了点击过程，这样就不需要每一次改变代码时手动完成这些动作，你已经写了点可执行文档。以后的开发者（或者甚至是两个月后的你）只要快速扫一下测试，就能知道你的意图。

回忆一下，在输入 grails create-domain-class Race 的时候，Grails 为你相应创建了一个单元测试。看看 test/unit 目录下的 RaceTests.groovy：

```
import grails.test.*

class RaceTests extends GrailsUnitTestCase {
    protected void setUp() {
        super.setUp()
    }

    protected void tearDown() {
        super.tearDown()
    }

    void testSomething() {
    }
}
```

那个空的测试架子正等着你用它完成一些聪明的事。例如，给 Race 增加一个助手方法，用来把距离从公里转换成英里。

```
class Race {
    // ...
    BigDecimal distance
```

```
BigDecimal inMiles(){
    return distance * 0.6214
}
```

现在，把 RaceTests.groovy 中不起眼的 testSomething() 方法改造得更有用一点：

```
import grails.test.*

class RaceTests extends GrailsUnitTestCase {
    // ...
    void testInMiles() {
        def race = new Race(distance:5.0)
        assertEquals 3.107, race.inMiles()
    }
}
```

testInMiles() 的第一行实例化了一个新的 Race，并将 distance 初始化成 5.0。第二行则验证 5 公里等于 3.107 英里。要是验证失败，测试也将失败。

如果熟悉 Java 测试框架 JUnit，你将很高兴地知道 GrailsUnitTestCase 扩展了 GroovyTestCase，该类进而又扩展了 JUnit 的 TestCase。这意味着所有你已经熟悉和热爱的断言——assertTrue、assertFalse、assertNull、assertNotNull 等——在这儿也同样可以使用（在安全章节里我们将谈到不同类型的 TestCase。）

运行测试，在命令行中输入 grails test-app。

```
$ grails test-app
...
Environment set to test

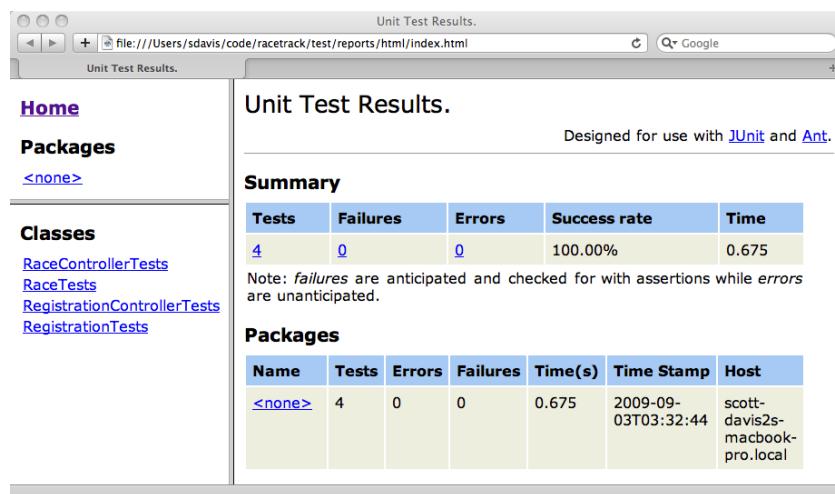
-----
Running 4 Unit Tests...
Running test RaceControllerTests...PASSED
Running test RaceTests...PASSED
Running test RegistrationControllerTests...PASSED
Running test RegistrationTests...PASSED
Unit Tests Completed in 463ms ...
```

No tests found in test/integration to execute ...

Tests PASSED - view reports in target/test-reports

命令行输出包含了大量信息。让我们逐一检查。

- 首先让你注意到的事情是环境被设置成测试（Test）而不是开发（Development）。（在我们讨论数据源的时候，你会了解更多关于环境的事情。）
- 接着，你可以看到有 4 个测试，包括你新增加的 testInMiles() 测试，每个都运行并且通过了。
- 集成测试没有被找到，但是它们是被期望出现——我们还没有写。待会儿，我们将给自定义验证写一个集成测试。
- 最后，在 target/test-reports/html 目录产生了一个新的 JUnit 报告。在浏览器中打开它：



Tests	Failures	Errors	Success rate	Time
4	0	0	100.00%	0.675

Note: failures are anticipated and checked for with assertions while errors are unanticipated.

Name	Tests	Errors	Failures	Time(s)	Time Stamp	Host
<none>	4	0	0	0.675	2009-03T03:32:44	scott-davis2s-macbook-pro.local

只要有测试失败了，你都可以深入这个类和各个测试探个究竟。

如果想专注某个类，你会非常乐意知道，可以通过输入 grails test-app Race 来只运行单个测试脚本。（在使用 test-app 命令运行测试时，只需要把末尾的“Tests”后缀去掉就可以了。）

祝贺你！你刚刚完成了你的首个 Grails 单元测试。但是，我们不是还没有开始着手测试我们的自定义验证吗？我们现在就完成它，在此之前你还需要了解一个更重要的概念——单元测试和集成测试的区别。

Grails 为你创建单元测试是因为它们可能是最容易运行的测试。单元测试并不要求 Web 服务器运行起来，数据库被初始化或者其它条件。单元测试就意味着独立地测试单个代码单元。

相反，集成测试是需要所有东西全都跑起来运行的。既然我们的自定义验证可能依赖数据库查找，Web 服务调用或者其它什么的，Grails 就把它当成了一类完全不同的测试。

在 test/integration 目录下创建文件 RaceIntegrationTests.groovy。

```
class RaceIntegrationTests extends GroovyTestCase {

    void testRaceDatesBeforeToday() {
        def lastWeek = new Date() - 7
        def race = new Race(startDate:lastWeek)

        assertFalse "Validation should not succeed", race.validate()
        assertTrue "There should be errors", race.hasErrors()
    }
}
```

注意，不用实际把类保存到数据库就可以通过调用 race.validate() 来测试你的验证。如果所有验证都通过，它将返回 true。另一种查看是否存在问题的方法是在调用 race.validate() 之后调用 race.hasErrors()。

你可能会忍不住让你的测试就写成现在这个样子，但是要注意，你并没有查找特定的错误。在有多个验证要进行测试时，你这样很可能会被麻痹，有一种安全的感觉。这里有一种更为细致的测试，深入并且断言实际的验证错误确实如我们所料地被触发了。

```
class RaceIntegrationTests extends GroovyTestCase {
    void testRaceDatesBeforeToday() {
        def lastWeek = new Date() - 7
        def race = new Race(startDate:lastWeek)
        assertFalse "Validation should not succeed", race.validate()
        //在验证失败后，它应该有错误
        assertTrue "There should be errors", race.hasErrors()

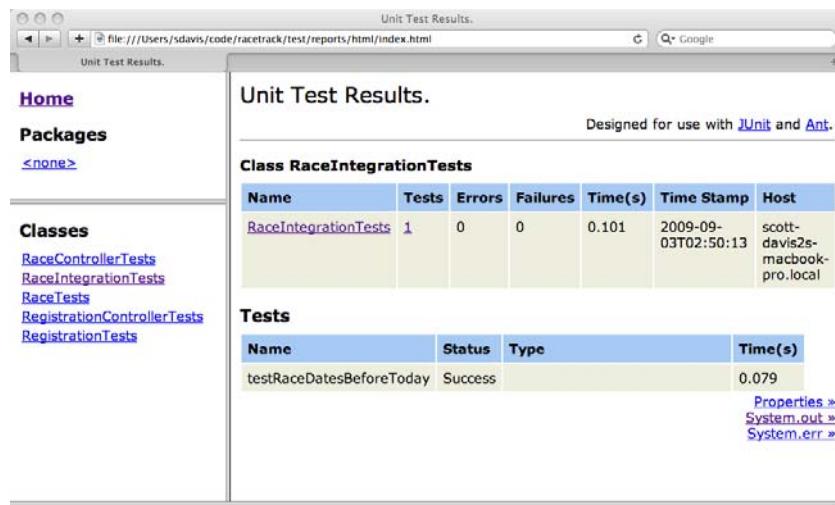
        println "\nErrors:"
        println race.errors ?: "no errors found"

        def badField = race.errors.getFieldError('startDate')
        println "\nBadField:"
        println badField ?: "startDate wasn't a bad field"
        assertNotNull "Expecting to find an error on the startDate field", badField

        def code = badField?.codes.find {
            it == 'race.startDate.validator.invalid'
        }
        println "\nCode:"
        println code ?: "the custom validator for startDate wasn't found"
        assertNotNull "startDate field should be the culprit", code
    }
}
```

输入 grails test-app 来看看这个测试的实际运行情况。

当你在 HTML 报表中下钻到 RaceIntegrationTests 时，看看右下角的一组链接。

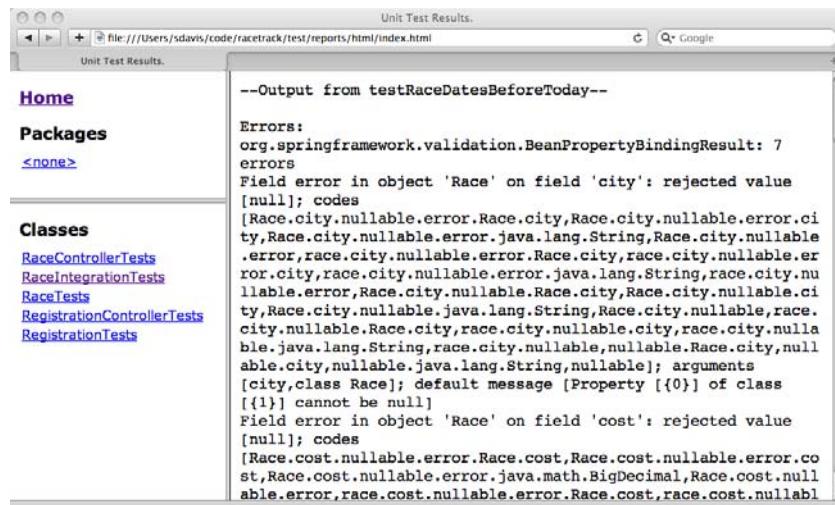


Name	Tests	Errors	Failures	Time(s)	Time Stamp	Host
RaceIntegrationTests	1	0	0	0.101	2009-09-03T02:50:13	scott-davis2s-macbook-pro.local

Name	Status	Type	Time(s)
testRaceDatesBeforeToday	Success		0.079

[Properties »](#)
[System.out »](#)
[System.err »](#)

要想查看所有 println 命令的输出，请点击 System.out 链接。



```
--Output from testRaceDatesBeforeToday--

Errors:
org.springframework.validation.BeanPropertyBindingResult: 7
errors
Field error in object 'Race' on field 'city': rejected value
[null]; codes
[Race.city.nullable.error.Race.city,Race.city.nullable.error.ci
ty,Race.city.nullable.error.java.lang.String,Race.city.nullable
.error,race.city.nullable.error.Race.city,race.city.nullable.er
ror.city,race.city.nullable.error.java.lang.String,race.city.nu
llable.error,Race.city.nullable.Race.city,Race.city.nullable.ci
ty,Race.city.nullable.java.lang.String,Race.city.nullable,race.
city.nullable.Race.city,race.city.nullable.city,race.city.nulla
ble.java.lang.String,race.city.nullable,nullable.Race.city,null
able.city,nullable.java.lang.String,nullable]; arguments
[city,class Race]; default message [Property [{0}] of class
[{1}]] cannot be null]
Field error in object 'Race' on field 'cost': rejected value
[null]; codes
[Race.cost.nullable.error.Race.cost,Race.cost.nullable.error.co
st,Race.cost.nullable.error.java.math.BigDecimal,Race.cost.null
able.error,race.cost.nullable.error.Race.cost,race.cost.nullabl
```

噢，7 个验证错误？没错——name 是空的，cost 小于 0 等等。尽管我们测试的第二个版本更琐碎，但也更精确。

好了，让我们再次输入 grails stats，看看目前的成果：

\$ grails stats

Name	Files	LOC
Controllers	2 6	
Domain Classes	2 33	
Unit Tests	4 46	
Integration Tests	1 19	
Totals	9 104	

+-----+-----+-----+

我们刚刚完成了超过 100 行的代码，但是已经完成了相当一部分的功能。我们现在已经有了一组页面，可以按正确的顺序显示字段。我们已经有了合适的验证，可以净化用户的输入，让无效数据无法进入数据库。而且，我们甚至还恰到好处地设置了一些有趣的测试，既包括单元测试，也包括集成测试。

5 关系

你在前一章完成了验证的探索。在本章，我们将把焦点转移到领域类之间的关系。

我们将在 Race 和 Registration 类之间建立起一个“一对多”关系（1:M）。最后，我们将在启动时初始化一些数据，免得不得不一遍又一遍地重复输入它们。

创建“一对多”关系

在给 RaceTrack 写用户故事的时候，我们首先识别的东西里有一个是比赛和注册信息之间的关系。若是连一个注册信息都没有，那比赛就没多少意义了，我说的你同意吗？

再看看这张纸质表格：

Southeastern Regional Running Club						
Race Name:	TURKEY TROT		Distance:	5K		
Date:	Nov 22, 2007		Time:	9:00 AM		
Location:	DUCK, NC					
Maximum # Runners:	350		Cost:	\$20.00		
Registered Runners:						
Date	Name	Address	E-Mail	Gender	DoB	
JUNE 1, 2007	JANE DOE	1234 ROCKY ROAD SOMWHERE, NC 12345	jane@doe.com	F	FEB 1, 1995	
JUNE 3, 2007	JOHN DOE	567 SUNDAY DRIVE ELSEWHERE, NC 12345	john@doe.com	M	JAN 1, 1974	

图 5-1：目前以纸张为基础的流程中所用表格

用 Grails 创建这个“一对多”关系再容易不过了——简直就是说大白话。

在文本编辑器中打开 grails-app/domain/Race.groovy，加入一行代码：

```
class Race {
    // ...
    statichasMany = [registrations:Registration]
}
```

这行代码创建了一个名为 registrations 的 新字段，类型是 java.util.Set。要是 Race 有多个关系，你可以逐一把它们加进来，使用逗号分隔：

```
class Race {
    // ...
    statichasMany = [registrations:Registration, locations:Location, sponsors:Company]
}
```

你的眼前是不是浮现出了一种模式？在定义约束、初始化构造函数中的值等活动中，你都用的是同一种“名字:值”列表。这里，名字是字段名，值是数据类型。就 `sponsors` 和 `Company` 这个例子而言，你是否看出你能够轻易地使名字不同于类名且更具描述性？

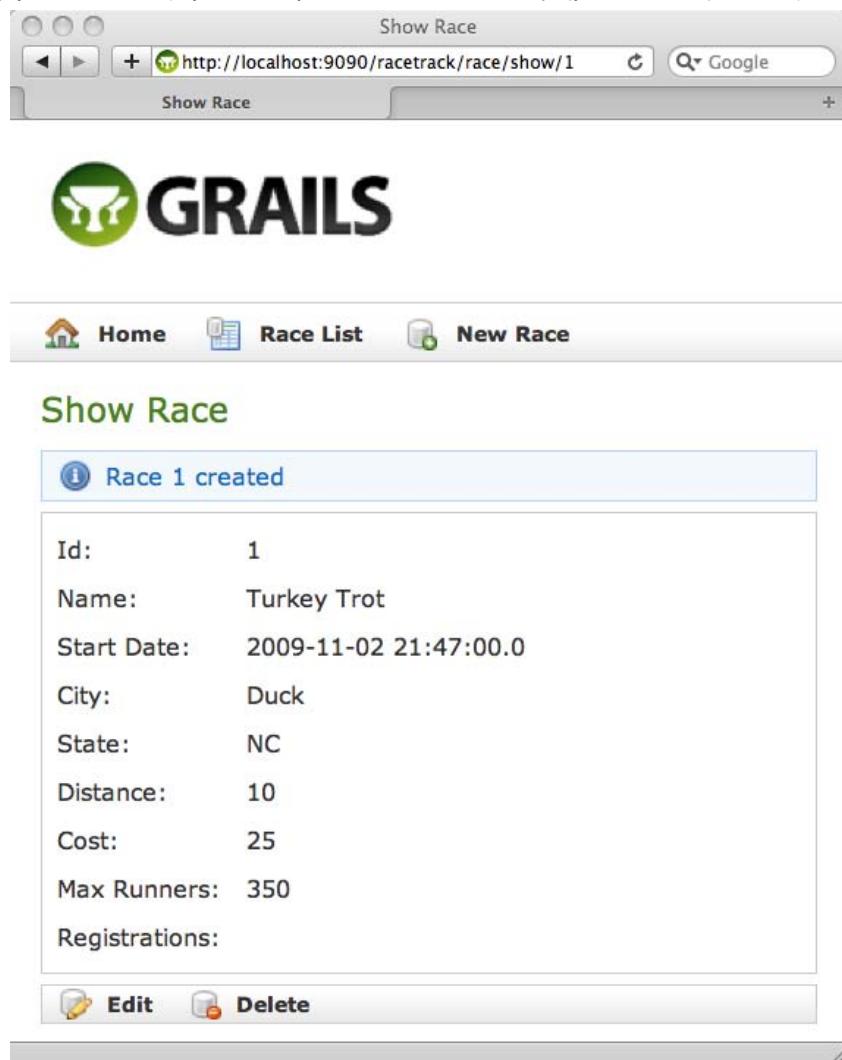
要是就此止步，你就已经得到了一个单向的“一对多”关系。换句话说，比赛（Race）非常清楚 它的注册信息（Registration），而注册信息（Registration）并不清楚它属于哪个比赛（Race）。

如果打算使关系变成双向的，你只需要在“一对多”中的“多”侧加一行代码。

```
class Registration {
    // ...
    static belongsTo = [race:Race]
}
```

这不仅形成了闭环；它还强制了级联更新和删除。

在我们这两行异样的代码就位之后，让我们看看它的实际运行情况。创建一个新的比赛：



The screenshot shows a web browser window with the following details:

- Address Bar:** Shows the URL `http://localhost:9090/racetrack/race/show/1`.
- Page Title:** "Show Race".
- Header:** "GRAILS" logo.
- Navigation:** "Home", "Race List", "New Race".
- Content:**
 - A message box: "Race 1 created".
 - A table of race details:
 - Id:** 1
 - Name:** Turkey Trot
 - Start Date:** 2009-11-02 21:47:00.0
 - City:** Duck
 - State:** NC
 - Distance:** 10
 - Cost:** 25
 - Max Runners:** 350
 - Registrations:** (empty)
 - Action Buttons:** "Edit" and "Delete".

看到这个新的 Registrations 字段了吗？在你第一次创建这个比赛的时候，它并没有出现，因为那个时候根本就不存在关系的这个“一”侧。但要是保存了比赛，你就可以开始增加这个关系的“多”侧了。



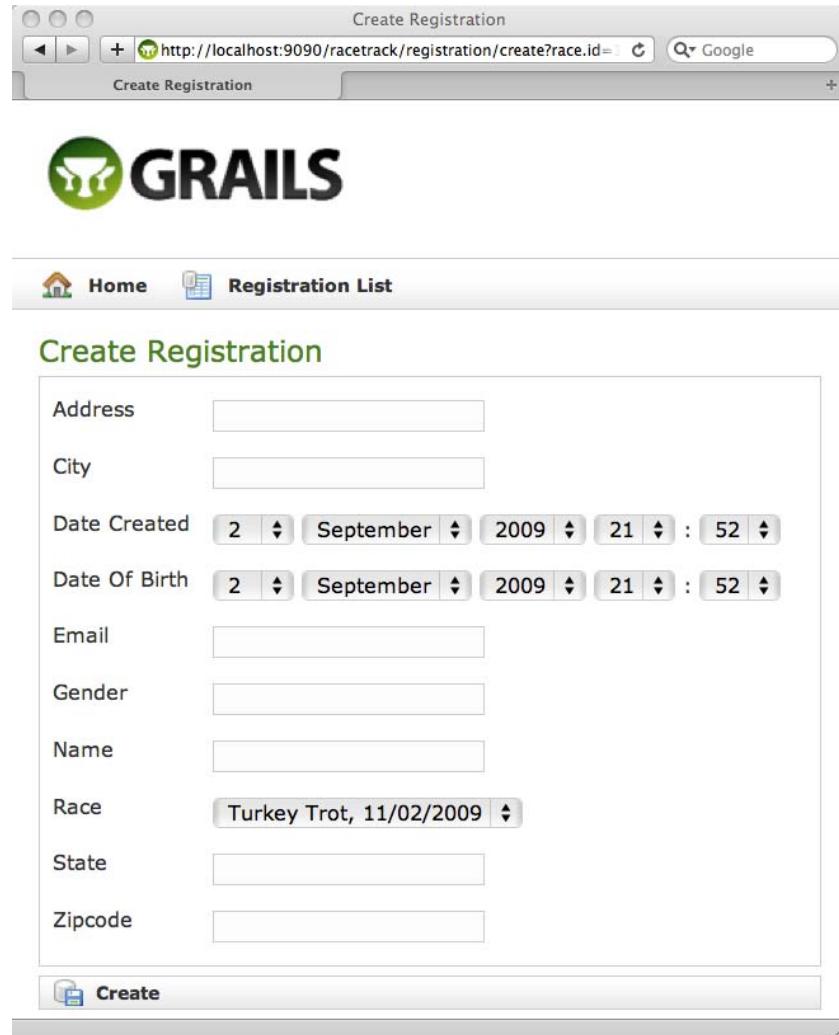
注意到那个指出该注册信息所关联比赛的组合框了吗？在验证一章中，我们用过 `inList` 来创建一个任意的值列表。这次，借助 `hasMany` 和 `belongsTo`，这个列表是从数据库中产生的。最终用户要做全部的就是记住其他表的主键。不用多说，这当然要换成更自动化的解决方案，不是吗？

等等，该怎么做？好吧，接下来你要做的就是对 `Race.groovy` 做一次小小的改动。Grails 会在组合框中使用类名和主键，除非类中有一个 `toString()` 方法（没错，Java 开发者们，也就是你们认为的那个相同的 `toString()` 方法。）

```
class Race {
    // ...
    static hasMany = [registrations:Registration]
```

```
String toString(){
    return "${name}, ${startDate.format('MM/dd/yyyy')}"
}
```

`toString()`方法并不需要返回一个唯一值——那是主键的任务——但是，在我们的这个例子中，比赛的名字和开始日期已经非常完美地使得组合框中的这些项不仅唯一，而且可读性很强。



这样，凭借 `hasMany` 声明、`belongsTo` 声明和 `toString()`方法，我们已经设法创建了一个“一对多”关系。但是，你猜怎么着？我认为我们在开始那一轮漏掉了一个类和一个关系。我认为有一个“多对多”关系就在我们眼前。

创建“多对多”关系

可以理解，比赛组织者关心比赛和注册信息——那是他们的工作。但是，选手们（Runner）关心的又是什么呢？更确切地说，如果我们遇到了一名跑了多次比赛的选手该

怎么办？一次比赛有多名选手，但是一名选手也可以参加多次比赛。看起来我们遇到了一个经典的“多对多”关系（这就是那些酷小孩们说的突发设计（**emergent design**））。

假设，作为一名选手，你报名参加一年的 12 次比赛，在一遍又一遍地为每次比赛都重复输入你的地址、性别和电子邮件地址之后，你肯定会感到相当厌烦。为何我们不把这些信息只捕获一次，然后每次在注册参加新比赛的时候重复利用呢？

你已经知道创建新领域类的方法了。在命令行中输入 grails create-domain-class Runner。把一些合适的字段由 Registration 移动到 Runner。完成之后，这两个类应该看起来是下面这个样子：

```
class Runner {  
    static constraints = {  
        firstName(blank:false)  
        lastName(blank:false)  
        dateOfBirth()  
        gender(inList:["M", "F"])  
        address()  
        city()  
        state()  
        zipcode()  
        email(email:true)  
    }  
  
    statichasMany = [registrations:Registration]  
  
    String firstName  
    String lastName  
    Date dateOfBirth  
    String gender  
    String address  
    String city  
    String state  
    String zipcode  
    String email  
  
    String toString(){  
        "${lastName}, ${firstName} (${email})"  
    }  
}  
  
class Registration {  
    static constraints = {  
        race()  
        runner()  
    }
```

```
paid()
dateCreated()
}

static belongsTo = [race:Race, runner:Runner]

Boolean paid
Date dateCreated
}
```

瞧，当我们把那些并不真正与 Registration 相关的全部字段抽取出之后，我们是不是发现了另一个需求浮现出来？我们忘了检查选手有没有给比赛付费。（纸质表格上并没有对应它的复选框，因为你不可能不交钱就报名。）

“多对多”关系是软件开发中最常被人误解的一种关系。如果我们一开始就分析选手和比赛，我们可能很早就发现它们共同拥有一个 M:M 关系。但是有经验的软件开发者并不会就此止步——优秀的开发者会像古生物学家一样继续挖掘其他信息，直到他们发现隐藏的第 3 个类（在我们的例子中是 Registration）。换句话讲，M:M 关系实际只是与你尚未发现的第 3 个类关联的 2 个 1:M 关系。

让我们在 Web 浏览器中看看这些新类。但在再次输入 grails run-app 之前别忘了创建一个管理选手的新控制器——grails create-controller Runner。

```
class RunnerController {
    def scaffold = true
}
```

修改这个新的控制器，以便它可以把所有事情都自动搞定。现在，你应该可以创建新的比赛.....

The screenshot shows a web browser window with the URL <http://localhost:9090/racetrack/race/show/1>. The page title is "Show Race". At the top, there is a navigation bar with links for "Home", "Race List", and "New Race". Below the navigation bar, the main content area displays the details of a race named "Turkey Trot" with ID 1. The race details are as follows:

Id:	1
Name:	Turkey Trot
Start Date:	2009-11-02 21:47:00.0
City:	Duck
State:	NC
Distance:	10
Cost:	25
Max Runners:	350
Registrations:	(empty)

At the bottom of the content area, there are "Edit" and "Delete" buttons.

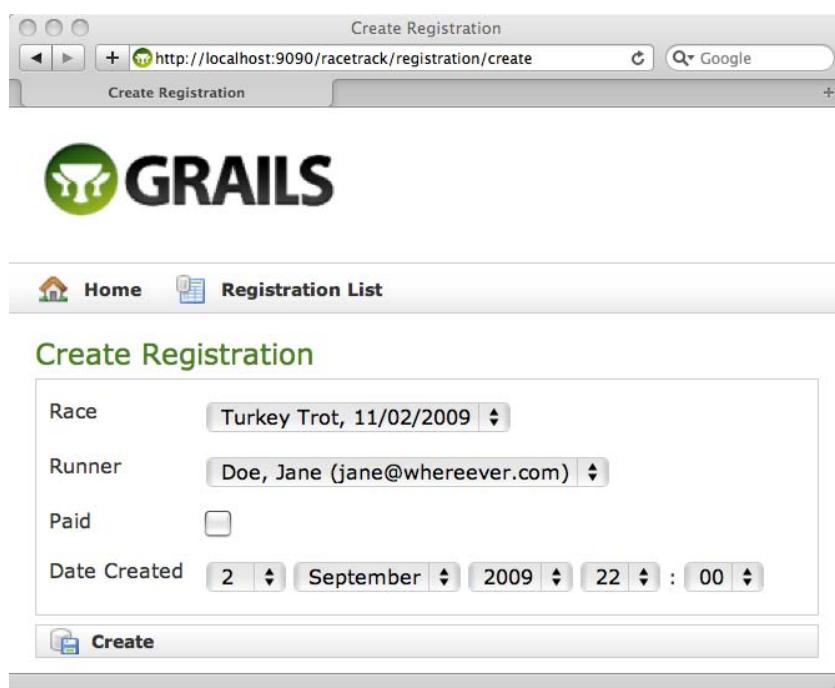
.....创建新的选手.....

The screenshot shows a web browser window with the URL <http://localhost:9090/racetrack/runner/show/1>. The page title is "Show Runner". At the top, there is a navigation bar with links for "Home", "Runner List", and "New Runner". Below the navigation bar, the main content area displays the details of a runner named "Jane Doe" with ID 1. The runner details are as follows:

Id:	1
First Name:	Jane
Last Name:	Doe
Date Of Birth:	1980-01-01 21:58:00.0
Gender:	F
Address:	123 Main St
City:	Goose
State:	NC
Zipcode:	12345
Email:	jane@whereever.com
Registrations:	(empty)

At the bottom of the content area, there are "Edit" and "Delete" buttons.

.....然后是让选手报名参加比赛：



启动时初始化数据

每次重启 Grails，都要重新输入数据，你是否对此感到厌烦？记住，这是一个特性，而非错误。我们处于开发（Development）模式，这意味着数据库表每次会在启动 Grails 时被创建，同时每次停止 Grails 时被删除。在数据库一章中你将学到如何调整这个特性，但在那之前，这里你会找到一个工具，不论运行在哪种模式之下，它都会对你有帮助。

检查一下 grails-app/conf 目录，里面有一个文件的名字叫 BootStrap.groovy。

```
class BootStrap {
    def init = { servletContext ->
    }

    def destroy = {}
}
```

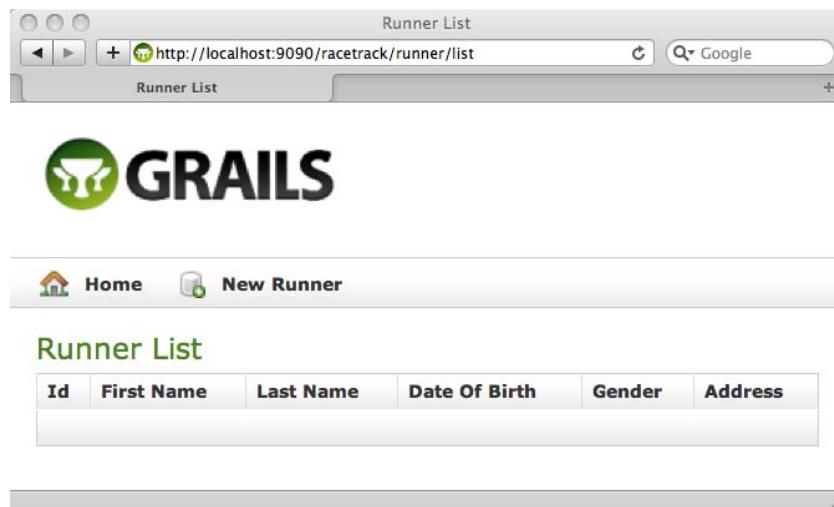
你可能已经猜到，init 闭包会每次在 Grails 启动时被调用；destroy 闭包会每次在 Grails 停止时被调用。

还记得我们在单元测试中实例化过一个类吗？在这里我们将使用同样的技巧，只是这次我们还要把这个类保存到数据库当中。

```
class BootStrap {
    def init = { servletContext ->
        def jane = new Runner(firstName:"Jane", lastName:"Doe")
        jane.save()
    }
}
```

```
def destroy = {}
```

启动Grails，访问 <http://localhost:9090/racetrack/runner>：



嗯，Jane 并没有把自己保存到数据库中，想一想怎样才能知道出了什么问题？

```
class BootStrap {
    def init = { servletContext ->
        def jane = new Runner(firstName:"Jane", lastName:"Doe")
        jane.save()
        if(jane.hasErrors()){
            println jane.errors
        }
    }

    def destroy = {}
}
```

当然！要是它在测试中能行，这里也应该没有什么意外，对不对？再次启动 Grails。

```
$ grails run-app
...
Environment set to development
Running Grails application..

org.springframework.validation.BeanPropertyBindingResult: 7 errors
Field error in object 'Runner' on field 'address': rejected value [null];
...

```

哦，那些约束真是麻烦，不是吗？让我们这次创建一个满足约束的选手：

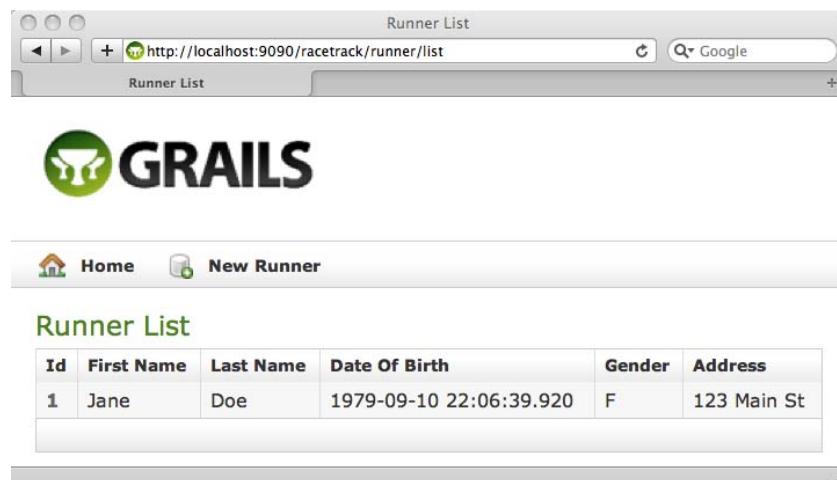
```
class BootStrap {
```

```

def init = { servletContext ->
    def jane = new Runner(
        firstName:"Jane",
        lastName:"Doe",
        dateOfBirth:(new Date() - 365*30),
        gender:"F",
        address:"123 Main St",
        city:"Goose",
        state:"NC",
        zipcode:"12345",
        email:"jane@whereever.com"
    )
    jane.save()
    if(jane.hasErrors()){
        println jane.errors
    }
}
def destroy = { }
}

```

再一次输入grails run-app , 这次你应该能够在 <http://localhost:9090/racetrack/runner>看到我们的好朋友Jane :



Id	First Name	Last Name	Date Of Birth	Gender	Address
1	Jane	Doe	1979-09-10 22:06:39.920	F	123 Main St

既然已经知道这对选手行得通，那让我们也增加一场比赛和一个注册信息。哦，还有件事：我会增加一些逻辑，以确保这些测试数据（绝非冒犯，Jane）只出现在开发模式里。

```

import grails.util.GrailsUtil

class BootStrap {
    def init = { servletContext ->
        switch(GrailsUtil.environment){
            case "development":
                def jane = new Runner(

```

```
firstName:"Jane",
lastName:"Doe",
dateOfBirth:(new Date() - 365*30),
gender:"F",
address:"123 Main St",
city:"Goose",
state:"NC",
zipcode:"12345",
email:"jane@whereever.com"
)
jane.save()
if(jane.hasErrors()){
    println jane.errors
}

def trot = new Race(
    name:"Turkey Trot",
    startDate:(new Date() + 90),
    city:"Duck",
    state:"NC",
    distance:5.0,
    cost:20.0,
    maxRunners:350
)
trot.save()
if(trot.hasErrors()){
    println trot.errors
}

def reg = new Registration(
    paid:false,
    runner:jane,
    race:trot
)
reg.save()
if(reg.hasErrors()){
    println reg.errors
}

break

case "production" : break
}
}
def destroy = { }
```

改完 `Bootstrap.groovy` 文件后，再次输入 `grails run-app` 以确保从今往后你可以专心输入代码，而不是重复地输入样本数据。

在本章结束之时，输入 `grails stats` 看看我们目前的成果：

```
$ grails stats
```

Name	Files	LOC
Controllers	3	9
Domain Classes	3	62
Unit Tests	6	68
Integration Tests	1	19
Totals	13	158

程序慢慢成型了，不是吗？我们的代码刚刚超过了 150 行。

现在，你知道了如何使用 `hasMany`、`belongsTo` 和 `toString()` 创建简单的 1:M 关系。你学到了 M:M 关系实际只是一个你尚未发现的拥有 2 个 1:M 关系的第 3 个隐藏类。而且最后，你花了少量时间在 `BootStrap.groovy` 上，免得让你因为每次重新启动 Grails 时需要重复输入数据而把键盘敲坏了。

6 数据库

前一章，我们利用 `Bootstrap.groovy` 来保证每次 Grails 重启都有样本数据出现。在这一章里，我们将探索运行于开发模式之下的 Grails 得健忘症的根本原因。我们还会让 Grails 能够与其他数据库进行对话，而不仅限于 HSQLDB。

GORM

不管愿不愿意，我们都已经创建了领域类，从来没有操心过这些数据的保存和位置。之所以我们能够如此享福，这都得感谢 GORM。Grails 对象-关系映射（Grails Object-Relational Mapping）API 得以让我们放心地以对象方式去思考问题——而不至于陷入到关系数据库相关的 SQL 当中。

GORM 是建立在 Hibernate 之上的一层轻量级的门面模式（我猜“Gibernate”不如 GORM 一样好发音）。这意味着，Hibernate 支持的任何数据库，Grails（借助 GORM）也能够支持。这还意味着，所有你已经熟悉的 Hibernate 的技巧同样也能在新的旅程中发挥作用——如果你有 HBM（Hibernate 映射文件）或带有 EJB3 注解的 POJO，你都可以拿来就用，无需改变。在本书中，我们将聚焦于 Grails 开发的新领域。

DataSource.groovy

要想更好地理解 GORM 的缺省设置，请在文本编辑器中打开 `grails-app/conf/DataSource.groovy`。

```

dataSource {
    pooled = true
    driverClassName = "org.hsqldb.jdbcDriver"
    username = "sa"
    password = ""
}
hibernate {
    cache.use_second_level_cache = true
    cache.use_query_cache = true
    cache.provider_class = 'com.opensymphony.oscache.hibernate.OSCacheProvider'
}
// environment specific settings
environments {
    development {
        dataSource {
            // one of 'create', 'create-drop','update'
            dbCreate = "create-drop"
            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"
    }
}
```

让我们对这个文件逐段进行探索。

第一段（被标记为 `dataSource`）包含了所有环境（environments）都共享的基本数据库设置。在这个例子里，开发（development）、测试（test）和产品（production）都共享一个公共的数据库驱动、用户名和口令（我们等会儿会详细地谈到环境）。如果你的产品（production）数据库要求的证书设置同开发（development）数据库不一样，你可以把这些值放到合适的代码段里。任何在环境（environment）段中的值都将覆盖这些“全局”设定。

顾名思义，`hibernate` 代码段是你可以调整 `Hibernate` 缺省设置的地方。除非你非常精通 `Hibernate` 的性能调优，否则没有必要改变这些设置。

最后一段（被标记为 `environments`）是一个会发生有趣行为的地点。首先，还记得那个每次在你重启 Grails 时删除数据的“特性”吗？这之所以会发生，都是因为 `dbCreate` 被设置成 `create-drop`。这个名字已经相当好地说明了自己的意思——每次重启 Grails，数据库表就要被创建；每次停止 Grails，这些表都会被删除。

注意，在测试（test）和产品（production）中，`dbCreate` 都被设置成了 `update`。在这个模式下，`Hibernate` 会保留这些表，使用 SQL `ALTER TABLE` 来使这些表与领域类保持同步。（第 3 个选项——`create`——会让这些表保持不变，但在停止 Grails 时删除数据。）

你可能会忍不住想在开发（development）环境中切换到 `update` 模式。尽管这在技术上不会引起任何问题，但请记住，在开发过程中，数据库模式很可能处于极端不稳定的状态——尤其是在刚开始的阶段。我发现，让它保持在 `create-drop` 模式，使用 `Bootstrap.groovy` 是一个让所有东西都保持同步的更优策略。

要是你认为让 GORM 管理数据库模式会让你的 DBA 浑身冒冷汗，大可不必担心。把这个值注释掉就可以告诉 GORM 不要做任何事情。这时，你就需要自己负责保证领域类和数据库表的同步，但有时这可能是最好的策略。如果你在一个遗留、共享的数据库上开发 Grails 应用，这就是一个完美的解决方案（致 Hibernate 专家：dbCreate 变量会直接映射到 hibernate.hbm2ddl.auto 值。）

另一个设置——url——是 JDBC 连接字符串。注意，产品（production）模式使用的是基于文件的 HSQLDB，而测试（test）和开发（development）模式则都使用的是基于内存的 HSQLDB。

切换到外部数据库

HSQLDB 是一个非常不错的数据库，特别适合快速搭建可执行应用，但是在实际投入到产品使用之前，你很可能想要切换到一个外部数据库。要完成这个工作，你需要做 3 件简单的事情：

1. 创建数据库，包括用户名和密码
2. 把 JDBC 驱动器复制到 grails-app/lib 目录
3. 调整 grails-app/conf/DataSource.groovy 中的设置

首先，我将向你演示如何在 MySQL 里创建一个新的数据库（至于其他数据库，请参考它们附带的手册。）。要是还没有安装好 MySQL，现在就该去做了。我会等你把它安装完。

好了，要创建新数据库，你需要以 root 用户登录。在整个开发过程中，你或许能只靠此时给开发（development）模式创建的数据库来度过，虽然你会觉得野心有点大，但是请尽管放心地也给测试（test）和产品（production）模式分别创建数据库好了。

```
$ mysql -u root -p
Enter password: *****
Welcome to the MySQL monitor. Commands end with ; or \g.
...
mysql> create database racetrack_dev;
...
mysql> grant all on racetrack_dev.* to grails@localhost identified by 'server';
...
mysql> exit
```

上面倒数第二条命令可能值得花点功夫解释一下。你把 racetrack_dev 数据库中所有对象的所有权限（如创建表、删除表、更改表等）都赋予了一个名叫 grails 的用户。这个用户只能从本机（localhost）登录——在数据库和 Grails 不在一台机器上的时候，你可以把 localhost 改为一个 IP 地址或另一个主机名。最后，在这个例子里，该用户的密码是 server。

为了验证 MySQL 终端上的所有设置都配置正确了，使用你新创建的凭证重新登录 MySQL。

```
$ mysql --user=grails -p --database=racetrack_dev  
Welcome to the MySQL monitor.  
  
Mysql> show tables;  
Empty set (0.00 sec)
```

行了，这样你就已经确认可以登录 MySQL，而且数据库是空的。

下一步就是把 MySQL 驱动器复制到 grails-app/lib 目录。你可以从 <http://dev.mysql.com/downloads/connector/j/> 下载最新的 MySQL Java 驱动器。解压该文件，查找 JAR 文件。（它的名字应该类似 mysql-connector-java-5.1.5-bin.jar。）

最后，把 grails-app/conf/DataSource.groovy 调整为你的新设置（注意：文件中的每个 URL 都应该是单独一行。）。

```
dataSource {  
    pooled = true  
    driverClassName = "com.mysql.jdbc.Driver"  
    username = "grails"  
    password = "server"  
}  
hibernate {  
    cache.use_second_level_cache = true  
    cache.use_query_cache = true  
    cache.provider_class =  
        'com.opensymphony.oscache.hibernate.OSCacheProvider'  
}  
// environment specific settings  
environments {  
    development {  
        dataSource {  
            // one of 'create', 'create-drop','update'  
            dbCreate = "create-drop"  
            // NOTE: the JDBC connection string should be  
            //      all on the same line.  
            url = "jdbc:mysql://localhost:3306/racetrack_dev?  
                  autoreconnect=true"  
        }  
    }  
    test {  
        dataSource {  
            dbCreate = "update"  
            url = "jdbc:mysql://localhost:3306/racetrack_dev?  
                  autoreconnect=true"  
        }  
    }  
}
```

```
    }
}

production {
    dataSource {
        dbCreate = "update"
        url = "jdbc:mysql://localhost:3306/racetrack_dev?"
            autoreconnect=true"
    }
}
```

看到没有，我将测试（test）和产品（production）模式都设置成了相同的数据库？那只是因为我的一时懒惰——对于实际的产品应用，我不推荐这么做。如果为每个模式都单独创建了 MySQL 数据库（而且你应该这样做！），你就可以在合适的时机做合适的改动。

现在到了关键时刻：我们真的把所有配置都设对了，可以让 Grails 来使用 MySQL 了吗？想确认这一点只有一个法子。输入 grails run-app，检查控制台上是否有错误输出。倘若一切正常，下一个要检查的地方就是 MySQL。

```
$ mysql --user=grails -p --database=racetrack_dev
Welcome to the MySQL monitor.
```

```
mysql> show tables;
+-----+
| Tables_in_racetrack_dev |
+-----+
| race           |
| registration   |
| runner         |
+-----+
3 rows in set (0.01 sec)
```

```
mysql> desc registration;
+-----+-----+-----+-----+
| Field      | Type      | Null | Key |
+-----+-----+-----+-----+
| id          | bigint(20) | NO   | PRI  |
| version     | bigint(20) | NO   |       |
| date_created | datetime  | NO   |       |
| paid         | bit(1)    | NO   |       |
| race_id     | bigint(20) | NO   | MUL  |
| runner_id   | bigint(20) | NO   | MUL  |
+-----+-----+-----+-----+
6 rows in set (0.02 sec)
```

```
mysql> desc race;
```

```
+-----+-----+-----+
| Field   | Type      | Null | Key |
+-----+-----+-----+
| id      | bigint(20) | NO   | PRI  |
| version | bigint(20) | NO   |       |
| city    | varchar(255)| NO   |       |
| cost    | decimal(5,2) | NO   |       |
| distance | decimal(19,2) | NO   |       |
| max_runners | int(11) | NO   |       |
| name    | varchar(50) | NO   |       |
| start_date | datetime | NO   |       |
| state   | varchar(2)  | NO   |       |
+-----+-----+-----+
9 rows in set (0.01 sec)
```

mysql> desc runner;

```
+-----+-----+-----+
| Field   | Type      | Null | Key |
+-----+-----+-----+
| id      | bigint(20) | NO   | PRI  |
| version | bigint(20) | NO   |       |
| address | varchar(255)| NO   |       |
| city    | varchar(255)| NO   |       |
| date_of_birth | datetime | NO   |       |
| email   | varchar(255)| NO   |       |
| first_name | varchar(255)| NO   |       |
| gender   | varchar(1)  | NO   |       |
| last_name | varchar(255)| NO   |       |
| state   | varchar(255)| NO   |       |
| zipcode | varchar(255)| NO   |       |
+-----+-----+-----+
11 rows in set (0.01 sec)
```

Grails 目前似乎能成功地跟 MySQL 进行对话。这一点也不困难，对不对？

看到了吗，grails-app/domain 目录中的每个类都有相应的表。每个属性都有对应的列。驼峰字段名（如 dateOfBirth）被转换成了包含下划线的小写名字（如 date_of_birth）。

GORM 还给每个类添加了两个额外的字段——id 和 version。毫无疑问，id 是主键。正是该列让 registration 表中的外键可以发挥作用。

版本（version）字段保留一个简单的整数，每次更新记录时该值都会增加。用 Hibernate 的行话来讲，这被称为乐观锁（**optimistic locking**）。两个使用者如果试图同时编辑该记录，第一次更新会增加版本（version）字段的值。当第二个使用者想保存该记录时，GORM 可以轻易地发现记录已经过时，从而警告这名使用者。

在这一章里，你终于知道了 Grails 为什么在每次重启之间看起来会忘记你的数据。要想修正这一行为，只需要简单的把 grails-app/conf/DataSource.groovy 中的 dbCreate 的值从 create-drop 改为 update。（如果是在使用 HSQLDB，你还需要把开发环境由内存数据库改为文件数据库。完成这一任务的一种方法是，把产品的 url 复制到开发中。）你还学到了如何给 Grails 指定外部数据库。

现在，我们已经解决完了数据库配置的难题，让我们把焦点放在下一个主要的组件：控制器。

7 控制器

在这一章中，我们将让缺省的控制器脚手架代码显现原形，这样我们就能对它进行细微的调整。这将帮助我们更好地了解控制器里的闭包（或 Action）和最终用户看到的 URL 之间的关系。这也让我们可以了解闭包与其对应的 Groovy 服务器页面（GSP）之间的关系。

比较create-controller 和generate-controller

到现在为止，我们都一直依赖控制器中神奇的 def scaffold = true 语句来提供完整的 CRUD 界面。这条语句不仅给控制器提供了行为，而且也给视图产生了 GSP。对于下一个要创建的类，让我们揭露其中的奥秘，这样便可以更好地理解背后的运作机制。

我们要创建一个新的 User领域类（在接下来的 安全一章里，我们会 用到它）。输入 grails create-domain-class User并添加以下代码：

```
class User {
    String login
    String password
    String role = "user"

    static constraints = {
        login(blank:false, nullable:false, unique:true)
        password(blank:false, password:true)
        role(inList:["admin", "user"])
    }

    String toString(){
        login
    }
}
```

在正常情况下，这时我会让你输入grails create-controller User。还记得吧，create-*命令会创建一个类的雏形。但这次我们的做法不一样，请输入grails generate-all User。generate-all命令会产生一个完整实现的控制器和一组对应的GSP视图（你也可以输入generate-views 只产生GSP，或输入generate-controller只创建控制器。）

与之前看到的控制器只有一行不同，你会看到 100 行左右的 Groovy 代码（为了简洁起见，这里的代码进行了修改）：

```
class UserController {
    static allowedMethods = [save: "POST", update: "POST", delete: "POST"]
    def index = {
```

```

    redirect(action: "list", params: params)  }
def list = {
    params.max = Math.min(params.max ? params.int('max') : 10, 100)
    [userInstanceList: User.list(params), userInstanceTotal: User.count()]
}
def create = {
    def userInstance = new User()
    userInstance.properties = params
    return [userInstance: userInstance]
}
def save = { // ... }
def show = { // ... }
def edit = { // ... }
def update = { // ... }
def delete = { // ... }
}
  
```

现在，所有的实现细节都一览无遗，那就让我们看看所有这些东西是怎么组合起来的吧。

理解URL和控制器

想想我们在使用应用进行浏览时所看到的 URL 类型：

<http://localhost:9090/racetrack/registration>
<http://localhost:9090/racetrack/registration/create>
<http://localhost:9090/racetrack/registration/show/2>

URL 的每个部分在 Grails 的惯例中都扮演着重要的角色。而且，正是因为遵循这个惯例，才使得我们免于在外部配置文件中自行把 URL、控制器和视图组装起来的痛苦，感谢上帝！

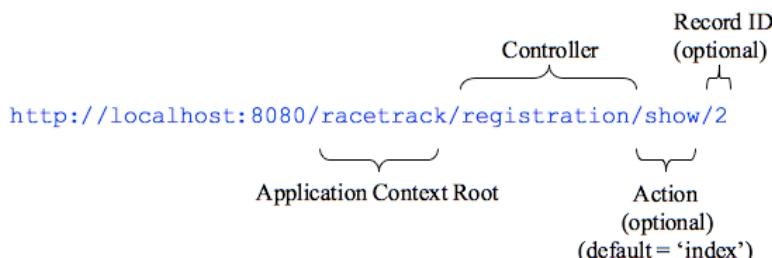


图 3-1：URL 的结构

URL 的第一部分是应用名或应用上下文的根。毫不奇怪，这就是我们在输入 `grails create-app racetrack` 时创建的那个目录的名字。可是，应用名和目录名并不需要一定匹配。打开 RaceTrack 应用根目录下的 `application.properties`，你会看到你可以把 `app.name` 设置成任何你想要的名字。这个新值将出现在 URL 的第一部分。

```
app.version=0.1
app.name=racetrack
app.servlet.version=2.4
app.grails.version=1.2
plugins.hibernate=1.2
plugins.tomcat=1.2
```

URL 接下来的部分是控制器的名字。注意， RegistrationController 的 Controller 后缀被去掉了，而且第一个字母换成了小写。

在控制器名字之后，你会看到 Action 的名字，它是可选的。这个名字直接对应于控制器中的闭包。假设有一个闭包名为 list，那么 URL 就是 registration/list。倘若省略 Action 名，那么缺省的 index 闭包就会被调用。

再仔细检查产生的 UserController，会发现 index 闭包只是简单地完成到 list Action 的重定向。第二个参数——params——是一个 HashMap，包含查询字符串的名字/值对。如果在 URL 中输入类似 <http://localhost:9090/racetrack/registration/list?max=3&sort=paid>，在控制器中，你可以分别使用 params.max 和 params.sort 来访问查询字符串中的值。

URL 的最后一部分，也是可选的，该位置一般是主键出现的地方。当你请求 edit/2、show/2 或 delete/2 时，你就是在要求编辑、显示和删除主键等于 2 的这条记录。由于处理记录 ID 实在是太常见了，为了方便起见，Grails 自动把这个值赋给 params.id。

让我们看看能否跟踪一个典型 Web 请求的整个过程。

从请求到控制器再到视图

下面的代码摘自刚刚创建的 UserController。

```
class UserController {
    def index = {
        redirect(action: "list", params: params)
    }
    def list = {
        params.max = Math.min(params.max ? params.int('max') : 10, 100)
        [userInstanceList: User.list(params), userInstanceTotal: User.count()]
    }
}
```

在 Web 浏览器中输入 <http://localhost:9090/racetrack/user>。你最终会看到一个空的用户列表。那么，你是怎样到达这个界面的呢？

首先，由于没有指定 Action，Grails 会把请求路由到 index Action。index Action 依次把请求重定向到 list Action，把查询字符串也一道传了过去。在 redirect 方法中用到的“名字:值”结构在整个控制器内都可以使用。其实，它就是一个简单“键/值”对的 HashMap。

三个 R : Action 闭包一般以 3 种方式结束，每种方式名称的第一个字母都是 R。（我喜欢把这称为“控制器的三个 R”）

第一个 R——redirect——把请求由一个 Action 传给另一个。第二个 R——render——把任意文本传回给浏览器（回想一下前面那个简单的“Hello World”例子）。render 命令不仅限于展示简单文本，你会在后面看到它的更多功能。它还可以展示 XML、JSON、HTML 等众多格式。最后一个 R 是 return。在某些情况下，return 是显式给出的。其他时候，如你在这个例子的 list 中所见，return 是隐含的。

好了，我们现在已经从 index Action 移动到了 list Action。list 的第一行代码会在查询字符串中没有出现 max 参数的情况下设置它。这个值用于分页。（一旦得到超过 10 个的 Runner、Race 或 Registration，Grails 就会创建指向另一页的导航链接。要想强制分页标准小于 10，可以传入 max 参数，如 runner/list?max=2。）

list 的最后一行是 return 语句。凭什么这么说？好吧，任何 Groovy 方法的最后一行都是一条隐式的 return 语句。在这个例子里，list Action 就是给 GSP 视图返回一个 HashMap 值（等下会详细介绍它）。

HashMap 中的第一项是用户（User）列表，大小受 max 参数的限制；第二项是用户（User）的总数，因为分页的缘故，该值会反复地被用到。

但是，这些静态方法是从哪儿冒出来的？仔细翻翻 User.groovy 文件，我没有看到哪个地方定义了静态的 list() 或 count() 方法。它们都是由 GORM 在运行时动态添加到 User 类上的（这被那些酷小孩们称为是元编程（metaprogramming））。浏览剩余的 UserController 源代码，你会发现有大量的方法被添加到了 User 类上—— save()、delete()、get() 等等。

那么，我们现在知道了 list Action 的最后一行是 return 语句。但还是有一个问题：我们怎么就从一个包含一组 User 的 HashMap 跑到了一个显示它们的 HTML 表格了呢？换句话说，这个 return 语句下一步会把我们带到哪里？

惯例再次发挥了作用，让 list Action 可以如此简洁。由于我们是在 UserController 中，而且由 list Action 返回，Grails 知道去使用位于 grails-app/views/user/list.gsp 的视图模板。

GSP 速览

我们会在下一章花更多时间深入探索 GSP。这里的目的是为了让我们可以了解整个请求生命周期的概况。在文本编辑器中打开 grails-app/views/user/list.gsp。下面是 body 部分的代码：

```
<g:each in="${userInstanceList}" status="i" var="userInstance">
  <tr class="${(i % 2) == 0 ? 'odd' : 'even'}">
```

```

<td>
  <g:link action="show" id="${userInstance.id}">
    ${fieldValue(bean:userInstance, field:'id')}
  </g:link>
</td>
<td>${fieldValue(bean:userInstance, field:'login')}</td>
<td>${fieldValue(bean:userInstance, field:'password')}</td>
<td>${fieldValue(bean:userInstance, field:'role')}</td>
</tr>
</g:each>
  
```

看到 userInstanceList 了吗？这个名字跟 list Action 返回的名字相同。不论控制器返回什么，GSP 都能直接使用。

在 list.gsp 的尾部，你可以看到实际的分页代码：

```

<div class="paginateButtons">
  <g:paginate total="${userInstanceTotal}" />
</div>
  
```

在自定义的 g:paginate 标签里使用了控制器返回的 userInstanceTotal 值。

了解控制器Action的其余内容

既然你已对一个完整的请求生命周期了然于胸，那么大可放心地自行浏览控制器 Action 的其余内容。不论哪种情况，都是从 URL 中抽取出 Action，传给合适的闭包，最后传递给 GSP。

假如 URL 是 user/show/2，那首先在 UserController 中要看的就是 show 闭包。

```

def show = {
  def userInstance = User.get(params.id)
  if (!userInstance) {
    flash.message = "${message(code:'default.not.found.message',
      args: [message(code: 'user.label',
        default: 'User'), params.id])}"
    redirect(action: "list")
  } else {
    [userInstance: userInstance]
  }
}
  
```

GORM 会尝试从数据库里获得主键值为“2”（保存在 params.id 中）的用户（User）。如果找不到，就在 flash 范围（类似 request 范围，存在时间稍长，在 redirect 之后仍然可以发挥作用）中放置一条消息。反之，若是找到了，则返回一个包含该用户（User）的 HashMap。

Grails 中的范围：Grails 中有 4 种范围：request、flash、session 和 application。每种范围的生命跨度都不相同。

- request 范围里的变量，仅出现于响应中，然后就马上丢弃。
- flash 范围里的变量（通常用于 Grails 中的错误消息）可以在单次重定向之后仍存在。
- session 范围用于那些需要在多个请求中使用，但仅限于当前用户的值。
- application 范围里的变量是“全局的”——它们的生命周期跟 Grails 运行的时间一样长，在所有用户中共享。

这个 HashMap 的目的地是哪儿？还用问，当然是 grails-app/views/user/show.gsp。整个 show.gsp 里，你可以随处看到 userInstance 的使用：

```
<tr class="prop">
  <td valign="top" class="name">
    <g:message code="user.id.label" default="Id" />
  </td>
  <td valign="top" class="value">
    ${fieldValue(bean: userInstance, field: "id")}
  </td>
</tr>
<tr class="prop">
  <td valign="top" class="name">
    <g:message code="user.login.label" default="Login" />
  </td>
  <td valign="top" class="value">
    ${fieldValue(bean: userInstance, field: "login")}
  </td>
</tr>
```

如何，简单吧？show URL 导出 show Action，接着导出 show.gsp 视图。“惯例优于配置”最大限度地发挥了其作用！

展示不匹配 Action 名字的视图

虽然传统根深蒂固，但有时你得打破规矩。总有些时候你会想使用与 Action 名字不匹配的视图。

例如，在创建新用户（User）的时候，表单被提交到了 save Action。检查一下 create.gsp 中表单的 action 属性，确认我说的没错。

```
<g:form action="save" method="post" >
  <!-- ... -->
</g:form>
```

但是，grails-app/views/user 目录中并没有相应的 save.gsp。save 该如何应付这种情况？

```
def save = {  
    def userInstance = new User(params)  
    if (userInstance.save(flush: true)) {  
        flash.message = "${message(code: 'default.created.message',  
            args: [message(code: 'user.label',  
                default: 'User'), userInstance.id])}"  
        redirect(action: "show", id: userInstance.id)  
    } else {  
        render(view: "create", model: [userInstance: userInstance])  
    }  
}
```

假如用户（User）成功保存，save Action 只是简单地重定向到 show Action。但若是在保存用户（User）过程中遇到问题（很可能是因为验证问题），save Action 则直接显示 create.gsp 视图。

在这一章，我们深入了解了控制器。我们生成了控制器，而不是单单地借助脚手架，从而揭示了其内部的工作机制。这让我们得以知道控制器中的每个闭包（或 Action）是 URL 的一部分。我们还学到了 Action 可以把值的 HashMap 返回给同名的 GSP，重定向到另一个 Action，或显示任意字符串或任意名字的 GSP。

在下一章，我们将深入了解 GSP。

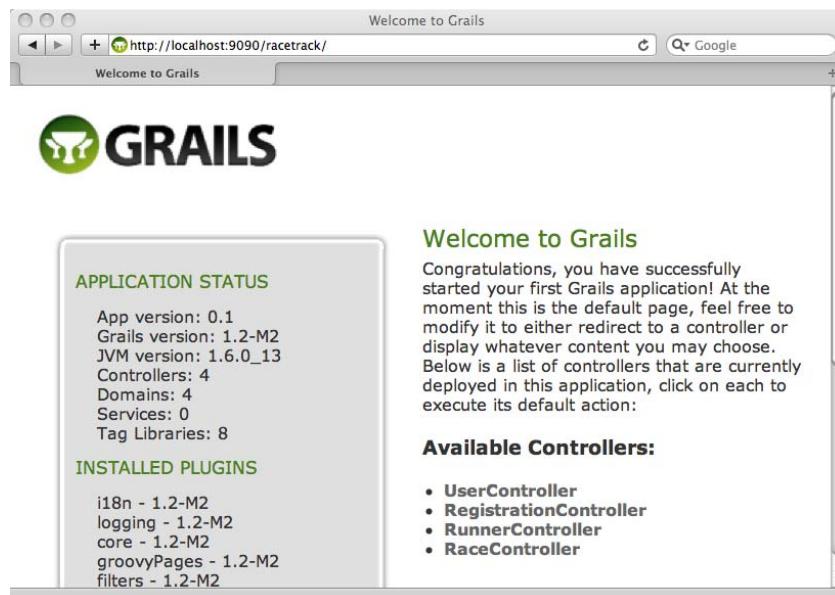
8 Groovy服务器页面

在这一章，我们将学到产生 Grails 应用视图的细节。你会看到 SiteMesh 如何与你的 GSP 结合使用。你将创建局部模板、自定义标签库等组件。

我们会在这一章快速前进。如果你已经熟悉HTML和基本的Web开发，那么跟上我们就没有任何问题。假如你需要提高你的客户端 Web 技能，请访问：<http://htmlcodetutorial.com>。

理解GSP

首先，输入 grails run-app，然后在浏览器中访问应用首页。



唔——我没有在 URL 上看到控制器，因此我们肯定是直接访问了 GSP。既然控制器里的 index 是缺省的 Action，我想知道是不是有个地方存在 index.gsp 页面？千真万确，grails-app/views 目录，那儿就有一个 index.gsp 文件。在文本编辑器中打开它。

```
<html>
  <head>
    <title>Welcome to Grails</title>
    <meta name="layout" content="main" />
    <!-- CSS styling snipped -->
  </head>
  <body>
    <div id="nav">
      <div class="homePagePanel">
        <div class="panelTop"></div>
```

```
<div class="panelBody">
    <h1>Application Status</h1>
    <ul>
        <li>App version: <g:meta name="app.version"></g:meta></li>
        <li>Grails version: <g:meta name="app.grails.version"></g:meta></li>
        <li>JVM version: ${System.getProperty('java.version')}</li>
        <li>Controllers: ${grailsApplication.controllerClasses.size()}</li>
        <li>Domains: ${grailsApplication.domainClasses.size()}</li>
        <li>Services: ${grailsApplication.serviceClasses.size()}</li>
        <li>Tag Libraries: ${grailsApplication.tagLibClasses.size()}</li>
    </ul>
    <h1>Installed Plugins</h1>
    <ul>
        <g:set var="pluginManager"
               value="${applicationContext.getBean('pluginManager')}">
        </g:set>
        <g:each var="plugin" in="${pluginManager.allPlugins}">
            <li>${plugin.name} - ${plugin.version}</li>
        </g:each>
    </ul>
</div>
<div class="panelBtm"></div>
</div>
</div>
<div id="pageBody">
    <h1>Welcome to Grails</h1>
    <p>Congratulations, you have successfully started your first Grails application! At the moment this is the default page, feel free to modify it to either redirect to a controller or display whatever content you may choose. Below is a list of controllers that are currently deployed in this application, click on each to execute its default action:</p>

    <div id="controllerList" class="dialog">
        <h2>Available Controllers:</h2>
        <ul>
            <g:each var="c" in="${grailsApplication.controllerClasses}">
                <li class="controller">
                    <g:link controller="${c.logicalPropertyName}">
                        ${c.fullName}
                    </g:link>
                </li>
            </g:each>
        </ul>
    </div>
</div>
</body>
</html>
```

它看上去是一个相当基本的 HTML 文件。然而，里面有几个你可能不认识的`<g:>`元素。它们是 Grails 标签。整个一章里，你不仅会学到很多缺省的 Grails 标签，而且最后还会自己动手创建一些。但这里学到的重要一课是：GSP 就是简单普通的 HTML + Grails 标签。

有一个常用的 Grails 标签你会反复地看到，它就是`<g:each>`。看看它在这里（以及前一章的 `list.gsp` 代码）的用法可能会让你很好地了解它的作用——遍历集合中的每一项。在这个例子里，它显示我们 Grails 应用中的每个控制器。

你还会看到`<g:link>`的频繁使用。它创建——让我们先等它创建完——一个指向具名控制器的链接（这一点也不难，对不对？）。查看 Web 浏览器中该页的源代码，你会看到一组包含正确格式的 URL 的超链接：

```

<ul>
  <li class="controller">
    <a href="/racetrack/runner">RunnerController</a>
  </li>

  <li class="controller">
    <a href="/racetrack/race">RaceController</a>
  </li>

  <li class="controller">
    <a href="/racetrack/registration">RegistrationController</a>
  </li>

  <li class="controller">
    <a href="/racetrack/user/index">UserController</a>
  </li>
</ul>

```

假如要创建指向某个 Action（从上一章你已经学到，`index.gsp` 中的这些链接会把使用者带到缺省的 index Action）的链接，只需增加一个 `action` 属性：

```

<g:link controller="user" action="create">
  New User
</g:link>

```

Grails 标签非常多，无法在这里一一介绍。我觉得熟悉它们最好的法子就是在产生的 GSP 里实际地观察它们。例如，在每个 `list.gsp` 文件的前面，你都会看到这个：

```

<g:if test="${flash.message}">
  <div class="message">${flash.message}</div>
</g:if>你可能会猜到，这段代码是检查flash范围中是否保存有message的值。如果有，就显示<g:if>标签包围的<div>。

```

想要了解所有可以使用的 Grails 标签，可以查阅这个优秀的联机文档：<http://grails.org/Documentation>。

理解 SiteMesh

index.gsp 里有一件事情比较蹊跷——我看不到 Grails 标志（Logo）的来源，你行吗？如果从 Web 浏览器中查看源代码，我能看到这个链接，但在 index.gsp 里却怎么也找不到它。你觉得这是怎么回事？

Grails 用到了一个流行的模板库，叫做 SiteMesh。从字面意思看，相当容易理解，SiteMesh 就是把两个 GSP 给拼接到一起。它是一种 Grails 把公共行为分解到一个可重用部分中的一种方法。Grails 标志、共享的 CSS 文件等都保存在一个单独的文件里，该文件再与你已经看到的 index.gsp 文件进行组合。

发现<head>部分中的<meta>标签了吗？

```
<head>
  <title>Welcome to Grails</title>
  <meta name="layout" content="main" />
</head>
```

这小提示（有人可能会大叫这也太不起眼了！）告诉你有一个名为 main.gsp 的模板在 grails-app/views/layouts 目录下。在文本编辑器里打开它。

```
<html>
  <head>
    <title><g:layoutTitle default="Grails" /></title>
    <link rel="stylesheet" href="${resource(dir:'css',file:'main.css')}" />
    <link rel="shortcut icon" href="${resource(dir:'images',file:'favicon.ico')}"
          type="image/x-icon" />
    <g:layoutHead />
    <g:javascript library="application" />
  </head>
  <body>
    <div id="spinner" class="spinner" style="display:none;">
      
    </div>
    <div id="grailsLogo" class="logo">
      <a href="http://grails.org">
        
      </a>
    </div>
    <g:layoutBody />
  </body>
</html>
```

根据<g:layoutHead>和<g:layoutBody>标签，这似乎表明，该文件要与我们前面看到的 index.gsp 文件结合起来使用。其实，事情就是这样。该文件的<head>部分会和其他 GSP 文件的<head>部分结合在一起，它们俩的<body>部分也是这样。

为了验证这一理论，把显示 Grails 标志的<div>注释掉，保存文件，然后在浏览器中刷新视图。

```
<!-- bye bye, logo
<div id="grailsLogo" class="logo">
  <a href="http://grails.org">
    
  </a>
</div>
-->
```

那个标志消失了吧？好了，我们现在了解了一些内幕。（顺便说一句，你注意到没有，resource 标签会创建指向目录中文件的链接，而 link 则创建指向控制器中 Action 的链接？还发现了吗，你可以使用<g:>语法或\${}语法调用标签库？）

接下来，让我们给应用创建一个自定义的首部。我们可以开始着手给 SiteMesh 模板直接增加代码，但是我想向你介绍另一种用 GSP 分解出公共制品的方法。

SiteMesh 非常适于组合两个完整的 GSP，但要是你想以更小规模完成同样的事情该怎么办？假如你只是想拥有部分的 GSP 代码，局部模板就是这个问题的答案。

理解局部模板

要在 Grails 里创建局部模板，只需在 GSP 文件名前加下划线即可。（在浏览 views 目录中文件的时候，文件名前的下划线从视觉上给人们提供了一个快速的提示：这些文件不是完整、格式良好的视图。）例如，在 grails-app/views/layouts 里创建名字为_header.gsp 的文件，把下列代码加到文件中：

```
<div id="header">
  <p><a class="header-main" href="${resource(dir:'')}>RaceTrack</a></p>
  <p class="header-sub">When's your next race?</p>
</div>
```

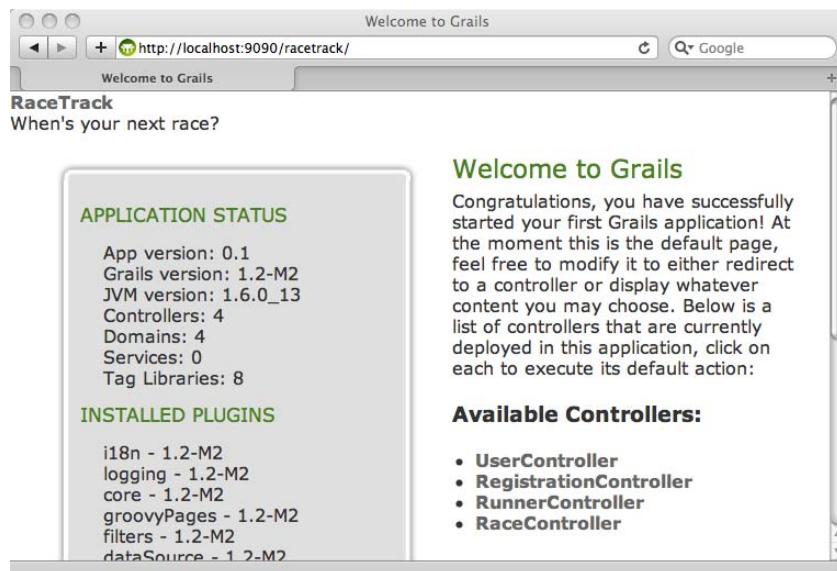
现在，让我们实际使用它。回到 main.gsp，然后在 main.gsp 内部使用<g:render>标签显示局部模板：

```
<body>
  <!-- snip -->
  <g:render template="/layouts/header" />
  <g:layoutBody />
```

```
</body>
```

在这里的上下文中，根目录是 `views` 目录。既然我们把局部模板放到了 `grails-app/views/layouts` 里面，模板所在路径就是`/layouts/`。而且，你没有看错，在使用`<g:render>`标签时，前导下划线和后缀`.gsp` 都去掉了。（第一次看到这个时候，你可能会有些混乱，但是很快你就会把它抛到脑后。）

再次刷新浏览器。你的屏幕应该看上去类似下面的样子：



现在，首部的局部模板已经完成，让我们加点 CSS 来格式化页面，让它变得好看点。把下列代码加到 `web-app/css/main.css` 的末尾。

```
/* RaceTrack Customization */
#header {
    background: #48802c;
    padding: 2em 1em 2em 1em;
    margin-bottom: 1em;
}

a.header-main:link,a.header-main:visited {
    color: #fff;
    font-size: 3em;
    font-weight: bold;
}

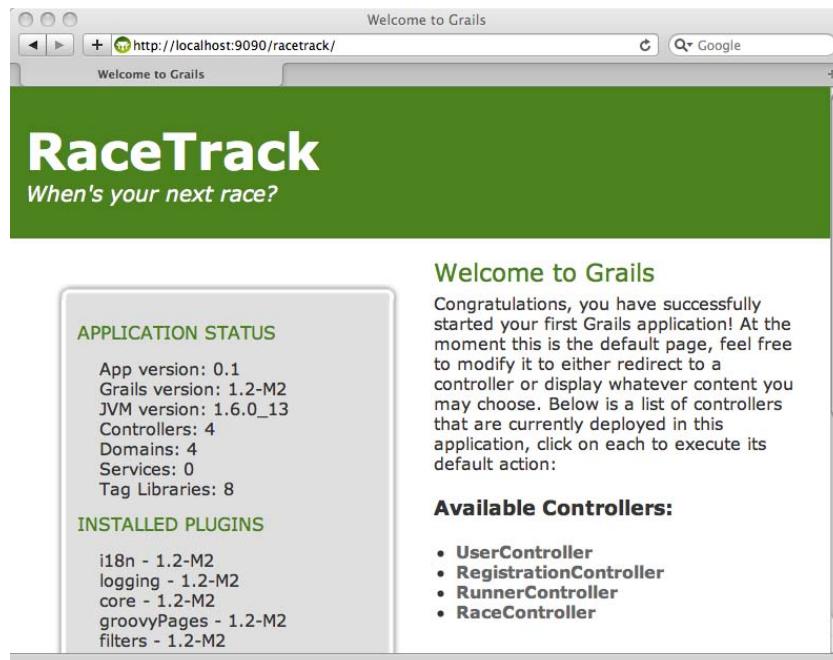
.header-sub {
    color: #fff;
    font-size: 1.25em;
    font-style: italic;
}
```

快速提示：以#开头的css条目指向的是ID（如header）。以.（点）开头的css条目指向的是类（如 header-main 和 header-sub）。要想提高你的css技巧，参见：<http://w3.org/Style/Examples/011/firstcss>。

有没有发现，在 grails-app/views/layouts/main.gsp 中的 SiteMesh 模板有一行包含了这个 CSS 文件？

```
<head>
    <link rel="stylesheet" href="${resource(dir:'css', file:'main.css')}" />
    <g:layoutHead />
</head>
```

再次刷新浏览器看看结果：



哦——真的开始看得有些顺眼了。现在，让我们暂停，复习目前所学的知识。你现在已经知道 Grails 利用 SiteMesh 通过<g:layoutHead>和<g:layoutBody>插入点把公共、共享的元素和各个 GSP 合并起来，局部模板让你完成更小规模的同类事情。你可以使用<g:render>标签包含来自局部模板的小段 GSP 代码。明白了？

既然你已经知道如何给首部创建局部模板，让我们给尾部也创建一个。创建 grails-app/views/layouts/_footer.gsp，它包含以下代码：

```
<div id="footer">
    <hr />
    &copy; 2009 Racetrack, Inc.
</div>
```

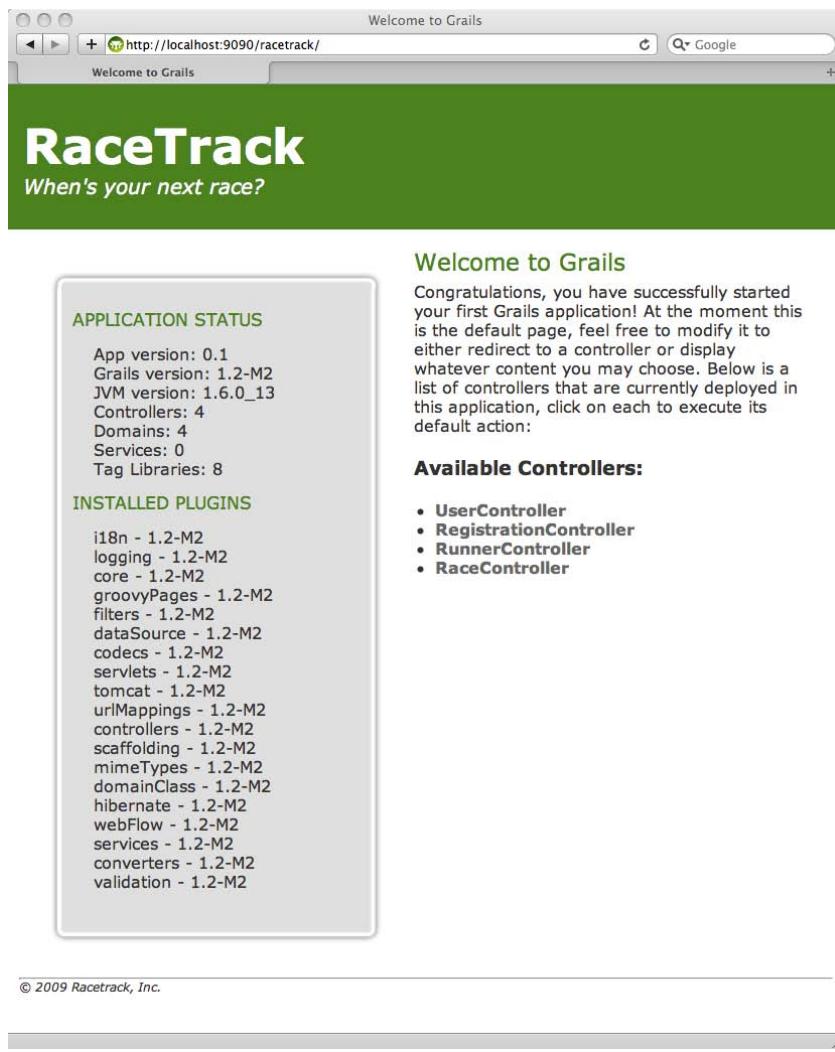
然后，把模板加到 main.gsp 中：

```
<body>
  <g:render template="/layouts/header" />
  <g:layoutBody />
  <g:render template="/layouts/footer" />
</body>
```

最后，给 main.css 加点 CSS：

```
#footer {
  font-size: 0.75em;
  font-style: italic;
  padding: 2em 1em 2em 1em;
  margin-bottom: 1em;
  margin-top: 1em;
  clear: both;
}
```

刷新浏览器查看结果：



The screenshot shows a web browser window with the title "Welcome to Grails". The address bar displays "http://localhost:9090/racetrack/". The main content area has a green header with the text "RaceTrack" and "When's your next race?". Below the header is a "Welcome to Grails" message and a "Available Controllers:" section listing "UserController", "RegistrationController", "RunnerController", and "RaceController". On the left, there is a sidebar titled "APPLICATION STATUS" showing app version 0.1, Grails version 1.2-M2, JVM version 1.6.0_13, controllers 4, domains 4, services 0, and tag libraries 8. Another section titled "INSTALLED PLUGINS" lists various plugins and their versions.

一切都看上去不错，但还是有些东西让人烦心。我不喜欢年份像现在这个样子硬编码到尾部模板中。要是有比局部模板规模更小的东西岂不是更妙？创建自己的自定义标签就是你要寻找的方案。

理解自定义标签库

既然 Grails 已经提供了大量<g:>标签，那么你也应该不会对可以创建自己的标签感到惊讶。可能让你感到意外的是创建它非常容易。

在命令行中输入 grails create-tag-lib Footer。就像 create-controller 和 create-domain-class 一样，它会在 grails-app/taglib 目录下创建一个 FooterTagLib.groovy 空样板。它也会创建相应的测试类。在文本编辑器里打开 FooterTagLib.groovy。

```
class FooterTagLib {  
}
```

暂时还看不出有多少内容，是不是？现在，增加以下代码：

```
class FooterTagLib {  
  
    def thisYear = {  
        out << new Date().format("yyyy")  
    }  
  
}
```

这就是创建一个新的<g:thisYear />标签的全部工作，仅仅只是把当前年份传给了它。Date 上的 format()方法与能在 java.text.SimpleDateFormat 中找到的那个方法是同一个方法。Groovy 利用元编程（metaprogram）把这个方法加到了 java.util.Date 上以方便使用，这跟 GORM 利用元编程把 list()、save()和 delete()加到你的领域类上的做法很象。

测试一下吧，编辑 grails-app/views/layouts/_footer.gsp 使用你的新标签：

```
<div id="footer">  
    <hr />  
    &copy; <g:thisYear /> Racetrack, Inc.  
</div>
```

输入 grails run-app，然后刷新浏览器，验证一下所有都没改变。（这些工作完全没有意义？好吧，等到 12 月 31 号晚上 11 点 59 分，然后持续 60 秒飞快地不断点击刷新。要是那天晚上你也可以安心睡大觉，知道自己不必在新年晚会的宿醉中不得不编辑你的 Grails 应用。）

创建一个简单的自定义标签是件快速而轻松的活儿，但要是想组装一个复杂的标签库，该怎么办？例如，假设你想创建一个<g:copyright>标签，它把版权日期输出成一个范围

(如，1999 – 2009) 而不是一个单独的年份。你可能会明确地给出起始年份，然后让标签动态地使用当前年份作为结束年份。最终的标签可能会像这个样子：

```
<div id="footer">
    <hr />
    <g:copyright startYear="1999">Racetrack, Inc.</g:copyright>
</div>
```

要创建这个更高级的自定义标签，给 FooterTagLib.groovy 增加一个如下的版权 (copyright) 闭包：

```
class FooterTagLib {

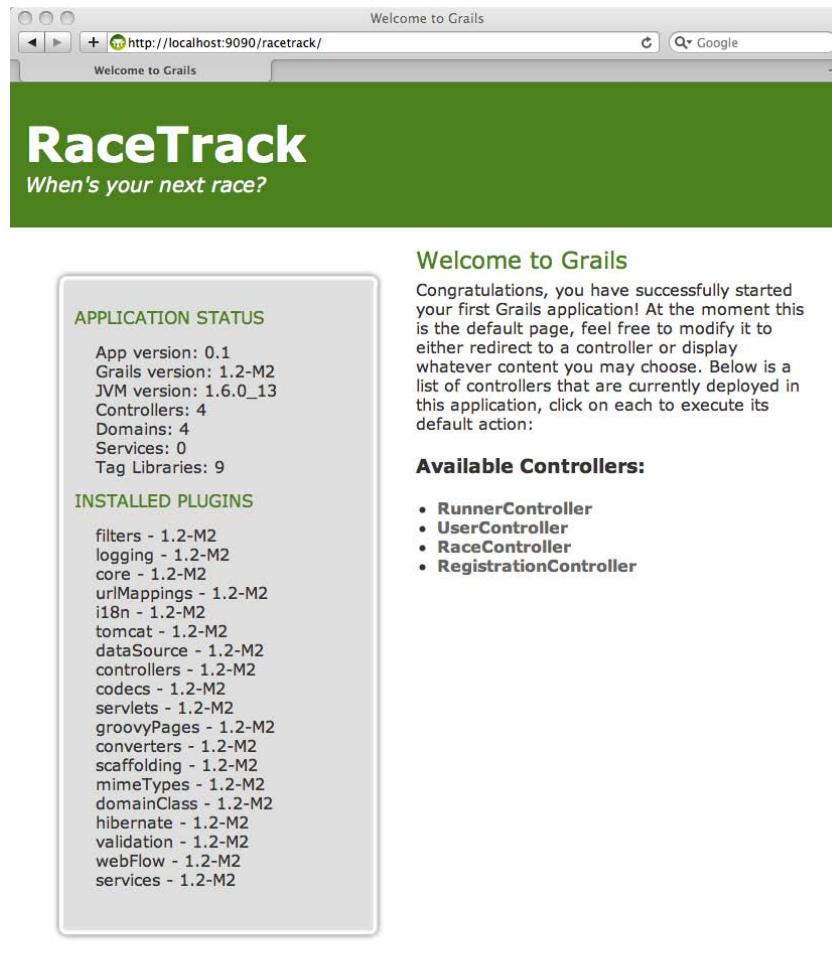
    def thisYear = {
        out << new Date().format("yyyy")
    }

    def copyright = {attrs, body->
        out << "&copy; " + attrs.startYear + " - "
        out << thisYear() + " " + body()
    }

}
```

看到闭包声明的那两个参数没有？ attrs 属性是一个 HashMap，类似控制器中的 params 和 flash， attrs 让你可以访问<g:copyright>标签的 startYear 参数。另一方面，body 标签是作为闭包传入的，你需要象方法一样去调用它。这就是你为什么会看到带括号的 body() 而不是简单的 body 的原因。不知你注意没有，你也可以象调用其它标签中的方法一样调用一个标签（如 thisYear()）？

刷新浏览器查看结果：

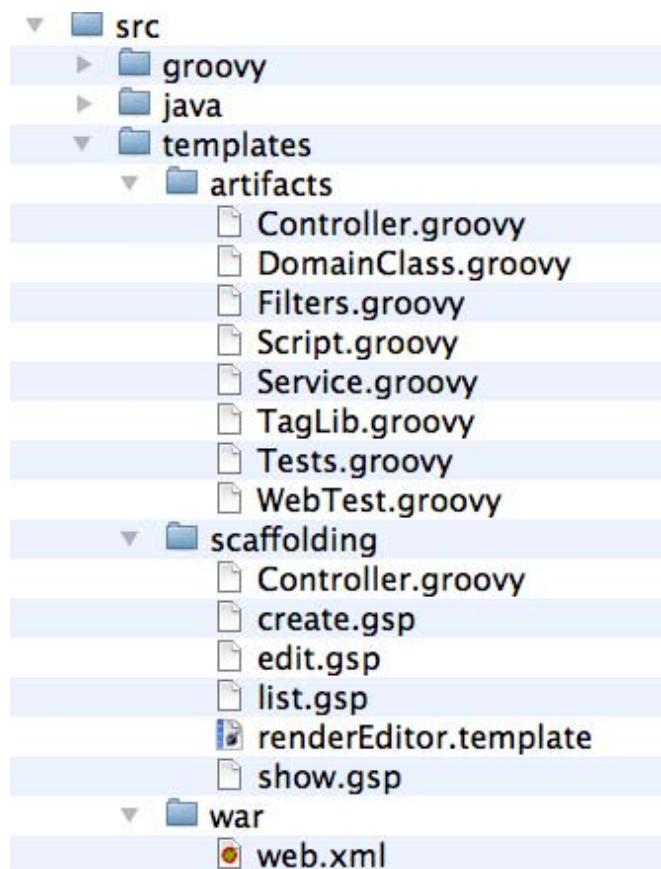


现在，我们非常善于在自动产生的视图范围内自定义内容了。但是，怎样自定义视图脚手架内的内容呢？我们可以自定义 Grails 本身吗？

自定义缺省模板

如果你也能自定义 `def scaffold = true`，它的魔法才真正显出威力。幸运的是，我们可以要求 Grails 暴露它的内部工作，这样我们就可以对其进行深度定制，直达其核心。这次的情况不同在于，我们并不是对以单个领域类为基础的一次性代码进行定制，我将向你展示如何一次性自定义所有的视图脚手架。

要想真正得到 Grails 的内部工作零件，输入 `grails install-templates`。这个命令会告诉 Grails 为所有视图、控制器、领域类、标签库等能够定制的组件创建模板代码。检查 `src/templates`，你可以了解所有这些文件的样板代码和细节。



好了，你能对这些样板文件做什么？比方说，你想让所有的控制器缺省都有脚手架代码。既然已经安装了模板，你就可以简单地把 `src/templates/artifacts/Controller.groovy` 修改成：

```
@artifact.package@class @artifact.name@ {
    def scaffold = true
}
```

每一个你通过输入 `grails create-controller` 创建的控制器都将使用这个模板作为它的源头。

作为另一个例子，假设你想给所有的领域类都产生 `toString()` 方法。你可以在 `src/templates/artifacts/DomainClass.groovy` 中做如下调整：

```
@artifact.package@class @artifact.name@ {

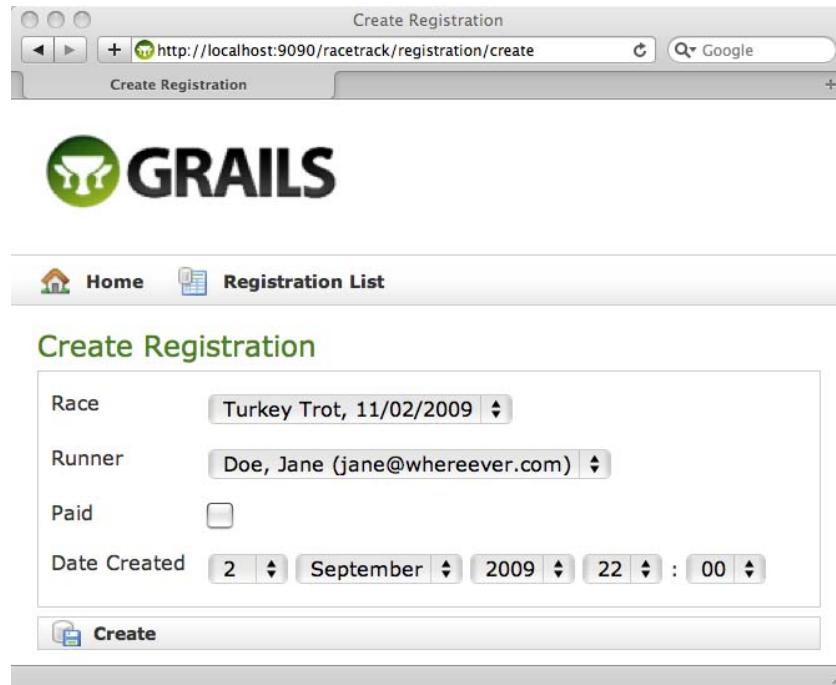
    static constraints = {}

    String toString(){
        // TODO: make me interesting
    }
}
```

如你所见，在开发早期自定义这些制品可以省去你在每次创建一个新的领域类、控制器等时不必要的代码拷贝粘贴。

虽然 artifacts 目录提供了针对于 grails create-*命令的模板，但 scaffolding 目录才是那些有趣 GSP 代码的源头。你能为这些 GSP 脚手架文件做哪些有益的事情呢？

首先，在自动产生出视图代码的时候，你是否发现那些神奇的 dateCreated 和 lastUpdated 字段也一并产生了？（在此提醒，我们曾给 Registration 类增加了 dateCreated 字段。记得吗，具有这些名字的字段，其值是由 Grails 自动在幕后产生的。）

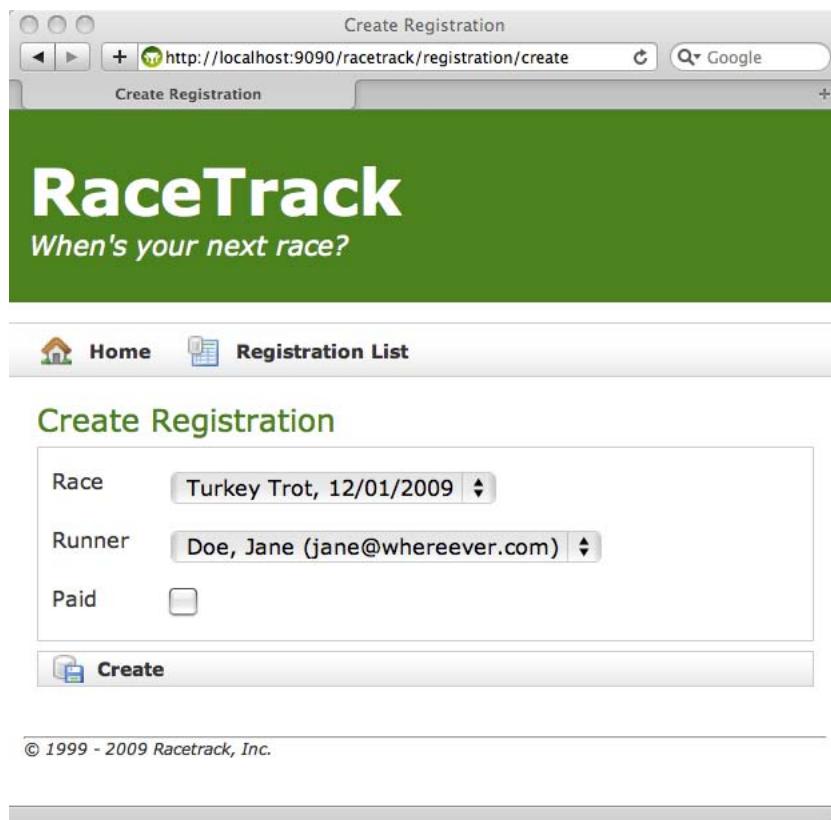


这太让人不爽了，是不是？毕竟，这些时间戳不可能真的让使用者进行任何调整。要是能把它们排除在视图脚手架之外岂非妙事？所幸，现在你已经安装了模板，你当然就可自己动手把事情搞定了。

在 src/templates/scaffolding/create.gsp 和 edit.gsp 中查找 excludedProps 列表。顾名思义，任何出现在这个列表中的领域类字段（或属性），都将排除在视图脚手架之外。把 dateCreated 和 lastUpdated 加到这个列表中：

```
<div class="dialog">
  <table>
    <tbody>
      <%
        excludedProps = Event.allEvents.toList()
        excludedProps << 'version'
        excludedProps << 'id'
        excludedProps << 'dateCreated'
        excludedProps << 'lastUpdated'
        props = domainClass.properties.findAll {
          !excludedProps.contains(it.name)
        }
      %>
```

再次启动Grails，访问 <http://localhost:9090/racetrack/registration/create>，瞧，时间戳不见了。



作为另一个例子——它会把我们带到下一章要讨论的授权和认证中——假设，你想对那些没有以管理员身份登录的用户隐藏列表视图中的导航条。你是否还记得，在这一章的前面，你看到的那个`<g:if>`检查，它只在有消息的时候才显示 `flash.message`？你可以在这里使用相同的技术。

在文本编辑器内打开 `src/templates/scaffolding/list.gsp`。搜索 `nav` div，它就在 `body` 标签后面。如果你把它包在一个简单的`<g:if>`检查中，你就可以把它隐藏在一个地方，而不是每次都重复手工产生和编辑每个 `list.gsp`。

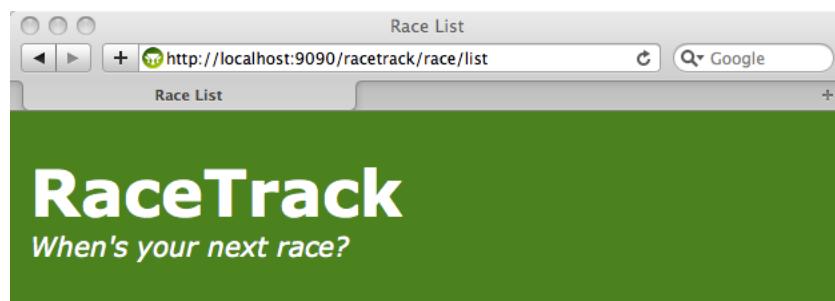
```
<g:if test="\${session?.user?.admin}">
    <div class="nav">
        <span class="menuButton">
            <a class="home" href="\${createLink(uri: '/')}">Home</a>
        </span>
        <span class="menuButton">
            <g:link class="create" action="create">
                <g:message code="default.new.label" args="[entityName]" />
            </g:link>
        </span>
    </div>
</g:if>
```

先不要去担心我们还没有创建任何用户（User），更不消说去创建那个在登录成功之后把他们放进 session 范围内的登录基础设施。而且，也不要操心你的用户（User）还没有 isAdmin()方法（这似乎有什么预示？的确如此。我们会把这些内容全部放在下一章再谈。）。这里的重点是，你已经成功地向非管理员使用者隐藏了导航条。搞清楚了？

快速提示：这里更多的是实现细节。有时在处理这些 GSP 模板的时候，你可能会期望 \${} 里的代码在模板产生的时候就执行（比方说，在输入 grails generate-all 或 grails generate-views 的时候）。另一些时候，你可能会期望 \${} 中的代码在该视图在运行应用请求的时候执行。这里的区别是产生时（**generation-time**）执行和运行时（**run-time**）执行。

要是仔细阅读了 list.gsp 模板，你会实际看到这两种情形。在 `<g:if test="\${session?.user?.admin}">` 里，美元符前的反斜杠标志告诉 Grails 在运行时执行这些代码。在其他地方，你会看到类似 `var="\${propertyName}"` 这样的代码。在这种情况下，propertyName 将在产生模板时被解析。

再次重启 Grails，访问 Race、Runner 或 Registration 的 list 视图脚手架。导航条应该在视图上被隐藏了。



Race List

Id	Name	Start Date	City	State	Distance
1	Turkey Trot	2009-12-01 23:46:24.0	Duck	NC	5

© 1999 - 2009 Racetrack, Inc.

当然，如果访问 <http://localhost:9090/racetrack/user/list>，你还是会看到导航条。你并没有让 User 的视图动态自动产生——不久前，你输入的是 grails generate-all User。假如重新产生这个视图，覆盖当前视图，你的新模板就会被采用。这个故事的意义在于，要在开发早期安装这些模板，在你产生大量代码之前定制它们。

既然确信它适用于 `list.gsp`，你可能也会想把这个改变应用到 `create.gsp`、`show.gsp` 和 `edit.gsp`。

在这一章，我们了解了很多方面的内容。你知道了 GSP 其实就是简单的 HTML 文件，里面不过散布了一些 Grails 标签（如`<g:if>`）。你学到了 SiteMesh 可以让你定义每个页面上包含的公共内容（比方，Grails 标志），通过 `grails-app/views/layouts/main.gsp` 和各个 GSP 模板进行组装。至于更细粒度的重用，你可以创建局部模板，创建以下划线开始的文件名，并在其他模板内部使用`<g:render>`标签来显示它们。对于粒度最细的重用，你学到了如何创建自定义标签库。最后，你安装了 Grails 模板，这样你就可以定制 Grails 中脚手架代码的几乎各个方面。

9 安全

在本章，我们会学习认证（你是谁？）和授权（你能做什么？）的基础。我们会动手做一个自定义的编解码器（Codec），它会在口令被保存到数据库当中的时候对它们加密。我们还将探索拦截器和过滤器等组件。

实现用户认证

检查一下你先前创建的用户（User）领域类。它包含了我们用来设置基本保护所需的全部信息。登录名（login）和口令（password）字段可用来认证（authenticate）用户——在他们做一些诸如改变个人信息这样的操作时，迫使他们表明自己的身份。角色（role）字段稍后会被授权（authorization）使用——判断他们是否被允许完成他们想要进行的操作。

```
class User {
    String login
    String password
    String role = "user"

    static constraints = {
        login(blank:false, nullable:false, unique:true)
        password(blank:false, password:true)
        role(inList:["admin", "user"])
    }

    String toString(){
        login
    }
}
```

还记得在 Groovy 服务器页面一章里我们给 list.gsp 增加了检查 admin 角色的功能吗？趁现在何不增加一个方便的方法，名字就叫 isAdmin()？正如你所见，user.admin 要比 user.role == admin 清晰得多。

```
class User {
    String login
    String password
    String role = "user"

    static constraints = {
        login(blank:false, nullable:false, unique:true)
        password(blank:false, password:true)
        role(inList:["admin", "user"])
    }
}
```

```

}

static transients = ['admin']

boolean isAdmin(){
  return role == "admin"
}

String toString(){
  login
}
}

```

如果你增加的访问器或修改器方法 (get、 set 或象本例中的 is) 没有一个与之对应的实际字段，有一点需要注意：它会哄骗 GORM 去到数据库的表中找这个字段。（嗨，人无完人！）

要想让 GORM 保持诚实，那就创建一个 static transients 列表，然后把 admin 加进去。transients 列表特别指明了那些不应该保存回数据库中的字段。即便严格地讲这儿并不存在 admin 字段，把它加到表中，也可以让 GORM 不要在我们背后搞鬼。

添加完管理员判断之后，接下来就要在 UserController 中增加两个方法，好让用户可以登录和注销。

```

class UserController {

  def login = {}
  def logout = {}
  def authenticate = {}

  // ...
}

```

让我们先处理登录。根据我们目前掌握的 Grails 生命周期的知识，访问 <http://localhost:9090/racetrack/user/login> 将在首先经过 UserController 中的 login Action 之后，最终显示 grails-app/views/user/login.gsp。（logout 和 authenticate 闭包将在本章的后面补充完整，这里让 login 空着完全没问题。假如该闭包不干任何事，它将简单地按预期显示该 GSP。）

我们并不打算从头开始创建 login GSP，而是借用现有的 create.gsp 文件，然后做些调整。把 create.gsp 复制到 login.gsp，按以下内容修改：

```

<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>

```

```
<meta name="layout" content="main" />
<title>Login</title>
</head>
<body>
<div class="body">
    <h1>Login</h1>
    <g:if test="${flash.message}">
        <div class="message">${flash.message}</div>
    </g:if>

    <g:form action="authenticate" method="post" >
        <div class="dialog">
            <table>
                <tbody>
                    <tr class="prop">
                        <td valign="top" class="name">
                            <label for="login">Login:</label>
                        </td>
                        <td valign="top">
                            <input type="text" id="login" name="login"/>
                        </td>
                    </tr>
                    <tr class="prop">
                        <td valign="top" class="name">
                            <label for="password">Password:</label>
                        </td>
                        <td valign="top">
                            <input type="password" id="password" name="password"/>
                        </td>
                    </tr>
                </tbody>
            </table>
        </div>
        <div class="buttons">
            <span class="button">
                <input type="submit" value="Login" />
            </span>
        </div>
    </g:form>
</div>
</body>
</html>
```

别担心——它完全不像第一眼看上去那样复杂。首先，把<title>和<h1>的值由 Create User 修改成 Login。接着，把 form action 改为 authenticate（它是 UserController 里的 Action，你马上就要去实现它）。你可以根据 login 和 password 简化<td>和<input>元素，

移走所有处理校验错误的代码——就简单的登录窗体而言，它们永远都派不上用场。注意，你也可以一并移除 role 字段。最后，把提交按钮的显示名改为登录（ Login ）。

用户填写表单，然后点击登录按钮，这时该表单将提交给 `authenticate` Action。把下列代码加到 `UserController` 中：

```
def authenticate = {
    def user = User.findByLoginAndPassword(params.login, params.password)
    if(user){
        session.user = user
        flash.message = "Hello ${user.login}!"
        redirect(controller:"race", action:"list")
    }else{
        flash.message = "Sorry, ${params.login}. Please try again."
        redirect(action:"login")
    }
}
```

我可以让你猜 3 次 `User.findByLoginAndPassword()` 的作用。要是你猜：“唔，它可能是通过 `login` 和 `password` 来查找 `user`”，那就奖励自己一下。这又一次展示了 GORM 元编程（metaprogramming）的魔法。你可以根据单个字段进行 `findBy`，也可以使用 `and` 或 `or` 针对两个字段。这些字段名特定于要查询的类，因此 `Race.findByNameAndCity()` 也是一个有效的查询。

假如你想返回一个结果列表，试试用 `findAllBy` 代替 `findBy`，如 `Race.findAllByState()`。

(关于 GORM 提供的动态查找器方法的更多内容 , 请参见 :<http://grails.org/DomainClass+Dynamic+Methods>。)

如果找到了 User记录，它就会被加到 session中，一条友好消息会随即加到flash范围，而且登录用户会被重定向到RaceController的list Action。反之，用户将被重定向回login Action。

要想进行测试，可以在 grails-app/conf/BootStrap.groovy 中创建一些样本用户。

```
import grails.util.GrailsUtil

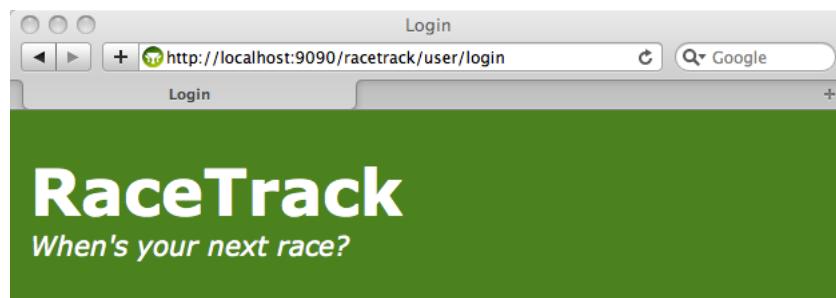
class BootStrap {
    def init = { servletContext ->
        switch(GrailsUtil.environment){
            case "development":
                def admin = new User(login:"admin",
                        password:"wordpass",
                        role:"admin")
```

```
admin.save()
if(admin.hasErrors()){
    println admin.errors
}

def jdoe = new User(login:"jdoe", password:"password",role:"user")
jdoe.save()
if(jdoe.hasErrors()){
    println jdoe.errors
}

// ...
}
```

随着login表单就位，authenticate Action也已准备就绪等待运行，同时还增加了2个测试用户。那就让我们把这个新的认证系统转起来吧。启动并访问：<http://localhost:9090/racetrack/user/login>。

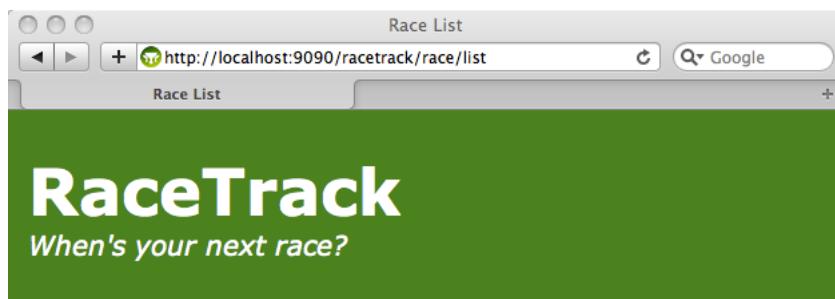


Login

Login:	<input type="text"/>
Password:	<input type="password"/>
<input type="button" value="Login"/>	

© 1999 - 2009 Racetrack, Inc.

到目前为止还不错。首先试试正常情况——输入 jdoe 和 password。你有没有到达比赛列表页面？



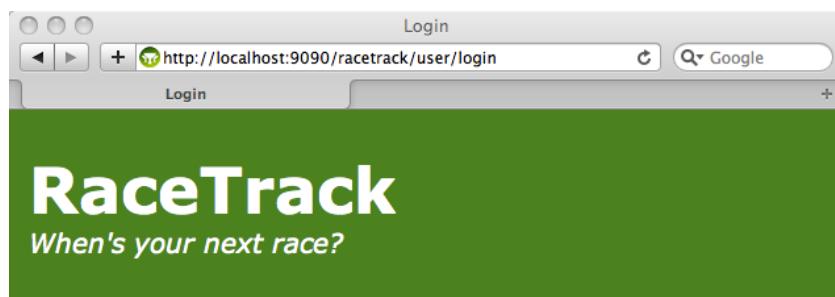
Race List

 Hello jdoe!

ID	Name	Start Date	City	State	Distance
1	Turkey Trot	2009-12-02 00:06:25.0	Duck	NC	5

© 1999 - 2009 Racetrack, Inc.

很好！现在试着使用错误口令登录。



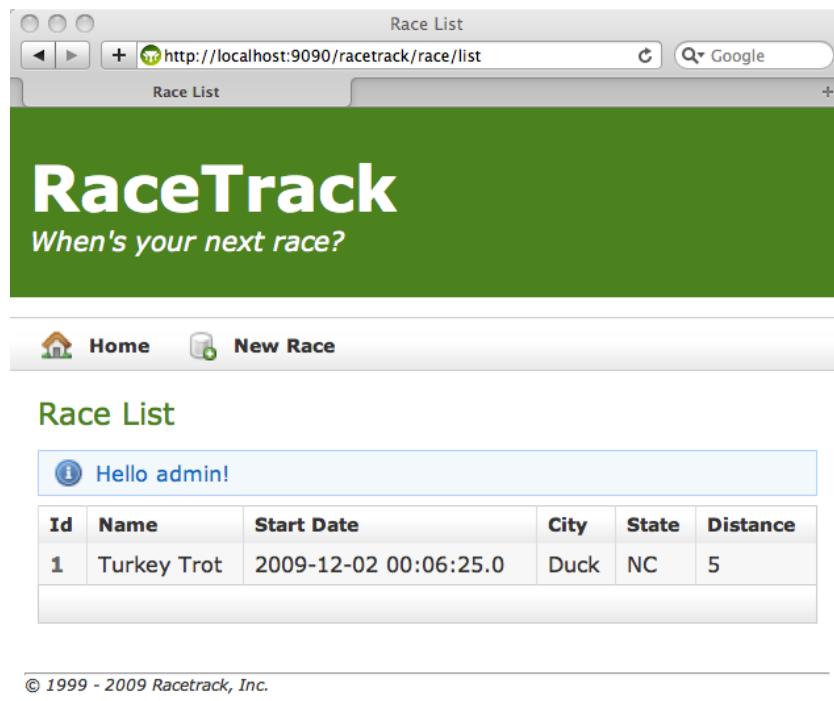
Login

 Sorry, jdoe. Please try again.

Login:	<input type="text"/>
Password:	<input type="password"/>
<input type="button" value="Login"/>	

© 1999 - 2009 Racetrack, Inc.

不错，现在看我们能不能 3 连胜。输入口令 wordpass，作为管理员登录。



Id	Name	Start Date	City	State	Distance
1	Turkey Trot	2009-12-02 00:06:25.0	Duck	NC	5

看到导航条出现没有？完美！

作为本节的结束，在 UserController 中输入以下代码处理注销：

```
def logout = {
    flash.message = "Goodbye ${session.user.login}"
    session.user = null
    redirect(action:"login")
}
```

要是愿意，你现在就可以测试这个 Action，但我想等下就创建一个标签库，它将完全利用 login 和 logout Actions。

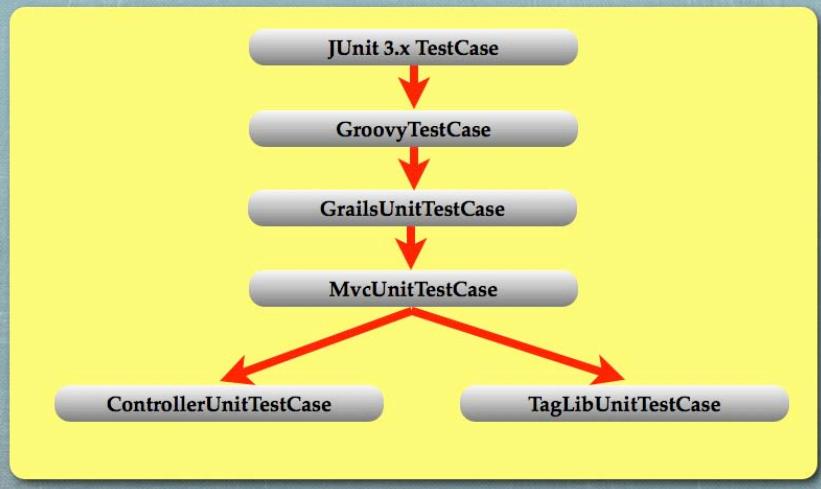
既然确信所有功能都可以工作了，如果有两个单元测试就位，你会不会感觉更好点？当然会.....

对控制器进行单元测试

还记得你给 Race 类写的单元测试和集成测试吗？一个使用的是 GrailsTestCase，另一个使用的是 GroovyTestCase。它们两个都扩展了 JUnit 3.x 的 TestCase。仔细瞧瞧给控制器提供的测试用例，你应该可以看到另一种类型的 TestCase——ControllerUnitTestCase。

这里有一个在 Grails 1.1 里引入的继承层次：

Grails 1.1.x Test Classes



那么，这种测试用例（TestCase）类的寒武纪大爆发（译注：寒武纪大爆发，也称为寒武纪生命大爆发，是指在距今5.3亿年前的寒武纪时期，短短200万年间，生命进化出现飞跃式发展的情形。几乎所有动物的“门”都在这一时期出现了。因出现大量的较高等生物以及物种多样性，于是，这一情形被形象地称为生命大爆发。这也是显生宙的开始。摘自维基百科，参考链

接：<http://zh.wikipedia.org/wiki/%E5%AF%92%E6%AD%A6%E7%BA%AA%E5%A4%A7%E7%88%86%E5%8F%91>) 意义何在 ? 答案很简单 : 模拟 (Mocking) 。集成测试运行成本很高——它们需要 Web 服务器运行起来 , 所有的元编程都就位 , 数据库初始化等。

`GrailsUnitTestCase`及其子孙都提供了相应的模拟服务，复制了集成测试中可用的那些服务，但又不会招致运行实际服务所需的开销。

GrailsUnitTestCase 提供了 3 个便捷的方法： mockDomain()、 mockForConstraintsTests() 和 mockLogging()。通常情况下，要书写基于这些功能的断言，你必须得创建一个集成测试。要是愿意，你仍然可以这么做。但要知道，单元测试运行起来要比集成测试快得多，模拟出这种行为是非常有意义的。

例如，假设你想测试用户（User）类上的验证。

```
import grails.test.*  
  
class UserTests extends GrailsUnitTestCase {  
  
    void testSimpleConstraints() {  
        def user = new User(login:"someone",  
                            password:"blah",  
                            role:"SuperUser")
```

```
// 啊呀，role应该是'admin'或'user'。  
// 验证会接受这个值吗？  
assertFalse user.validate()  
}  
}
```

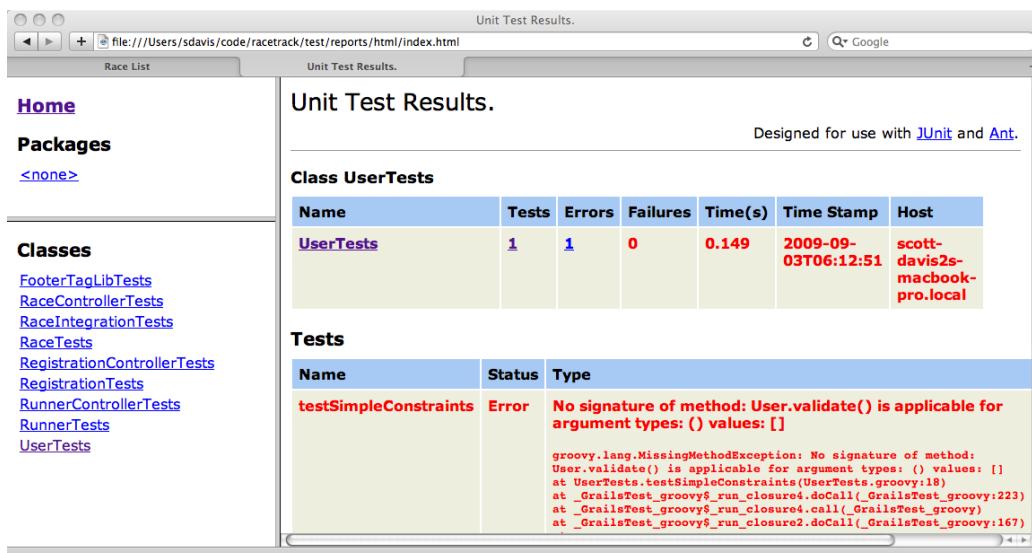
要是在不模拟出这些约束的情况下运行这些单元测试，你会以一次有趣的失败而收场。

```
$ grails test-app  
Environment set to test  
  
Starting unit tests ...  
Running tests of type 'unit'  
  
-----  
Running 8 unit tests...  
Running test FooterTagLibTests...PASSED  
Running test RaceControllerTests...PASSED  
Running test RaceTests...PASSED  
Running test RegistrationControllerTests...PASSED  
Running test RegistrationTests...PASSED  
Running test RunnerControllerTests...PASSED  
Running test RunnerTests...PASSED  
Running test UserTests...  
    testSimpleConstraints...FAILED  
Tests Completed in 734ms ...  
  
-----
```

```
Tests passed: 7  
Tests failed: 1  
  
-----
```

```
Starting integration tests ...  
...  
  
[junitreport] Processing  
Tests FAILED - view reports in target/test-reports.
```

检查 test/reports 目录里的 HTML 报告。



Name	Tests	Errors	Failures	Time(s)	Time Stamp	Host
UserTests	1	1	0	0.149	2009-09-03T06:12:51	scott-davis2s-macbook-pro.local

Name	Status	Type
testSimpleConstraints	Error	No signature of method: User.validate() is applicable for argument types: () values: [] groovy.lang.MissingMethodException: No signature of method: User.validate() is applicable for argument types: () values: [] at UserTests.testSimpleConstraints(UserTests.groovy:18) at _GrailsTest_groovy\$run_closure4.doCall(_GrailsTest_groovy:223) at _GrailsTest_groovy\$run_closure4.call(_GrailsTest_groovy) at _GrailsTest_groovy\$run_closure2.doCall(_GrailsTest_groovy:167)

这里的错误是“缺少方法签名：User.validate()。”这显然说明了 validate()方法是通过元编程加到领域类中的，在集成测试中可用，但不是单元测试。这样，你就面临两种选择：要么把这个测试由 unit 文件夹移动到 integration 文件夹，要么模拟出这些约束。

记住，集成错误绝对没有任何过错。如果我们在一个处于活动状态、运行着的 Grails 应用中运行这个测试，你根本无需进行任何模拟——所有东西的行为跟产品环境中的一模一样。我之所以鼓吹模拟出这个功能的原因在于：速度，速度，还是速度。一个让人感到不快的事实是，要是你的测试要花很长时间运行，你很可能就会不愿意运行它们。GrailsTestCase 的模拟能力就是为了让你的测试过程提速，消除另一个（yet another）站不住脚的你可能用来逃避测试代码的借口。

在这样的精神鼓舞下，给测试代码增加 2 行，模拟出约束检查：

```
import grails.test.*

class UserTests extends GrailsTestCase {
    void testSimpleConstraints() {
        mockForConstraintsTests(User)
        def user = new User(login:"someone",
                           password:"blah",
                           role:"SuperUser")
        // 啊呀，role应该是'admin'或'user'。
        // 验证会接受这个值吗？
        assertFalse user.validate()
        assertEquals "inList", user.errors["role"]
    }
}
```

mockForConstraintsTest()告诉单元测试把计算约束的方法元编程到 User 类上。最后一行断言，inList 对于 role 字段的验证会失败。

要运行所有测试——包括单元测试和集成测试——输入 grails test-app。如果只想运行单元测试，输入 grails test-app -unit。如果只想运行集成测试，输入 test-app -integration。如果只想运行针对 User 的单元测试，输入 type grails test-app User -unit。

```
$ grails test-app User -unit
```

```
Starting unit tests ...
Running tests of type 'unit'
```

```
-----
Running 1 unit test...
Running test UserTests...PASSED
Tests Completed in 466ms ...
```

```
-----
Tests passed: 1
Tests failed: 0
```

好了，既然你已经理解了 mockForConstraintsTests()，让我们再看看 mockDomain()。它在模拟之路上走得更远——它允许你模拟出整个数据库层。为了测试 login 字段上的 unique 约束，你需要一个数据库（不论真假）。我们将使用一个模拟出的数据库来达到我们的目的。

把下列代码加到 UserTests :

```
void testUniqueConstraint(){
    def jdoe = new User(login:"jdoe")
    def admin = new User(login:"admin")
    mockDomain(User, [jdoe, admin])

    def badUser = new User(login:"jdoe")
    badUser.save()
    assertEquals 2, User.count()
    assertEquals "unique", badUser.errors["login"]

    def goodUser = new User(login:"good", password:"password", role:"user")
    goodUser.save()
    assertEquals 3, User.count()
    assertNotNull User.findByLoginAndPassword("good", "password")
}
```

mockDomain()方法创建了包含 2 个用户的模拟表。试图保存包含有与现有用户相冲突的 login 字段的用户将会失败，原因是违反了 unique 约束。但是保存 goodUser 则没有任何问题。（你有没有注意到 mockDomain() 同时也给你提供了全部 GORM 动态查找器的好处？）

最后一个 GrailsUnitTestCase 的方法——`mockLogging()`——会给类注入一个日志对象并把输出写到 `System.out`，而非依赖 Log4J 是否配置正确。

我们不是想开始测试 UserController 中新的 authenticate Action 吗？这就是 ControllerUnitTestCase 出来发挥作用的地方。如果 `UserControllerTests.groovy` 还不存在，那就在 `test/unit` 目录中创建它：

```
import grails.test.*

class UserControllerTests extends ControllerUnitTestCase {
    void testSomething() { }
}
```

ControllerUnitTestCase 具有 GrailsUnitTestCase 的全部功能，而且更多。

例如，你可以验证重定向按预想的发生了。你知道UserController中的 index Action应该重定向到list Action：

```
class UserController {
    def index = {
        redirect action:"list", params:params
    }
}
```

这里就是验证它的测试：

```
import grails.test.*

class UserControllerTests extends ControllerUnitTestCase {
    void testIndex() {
        controller.index()
        assertEquals "list", controller.redirectArgs["action"]
    }
}
```

UserController 实例会以名字为 controller 的变量传给每个测试方法。在这个简单测试里，你调用 `index()` Action，然后断言 list Action 是 redirect HashMap 中键值为 action 的值。

输入 `grails test-app UserController -unit` 运行这个测试。

这里有个稍微棘手点的例子。记得在类似 <http://localhost:9090/racetrack/user/show/2> 的 URL 中，数字 2 在查询字符串中代表 `params.id`。

```
class UserController{
    def show = {
        def userInstance = User.get(params.id)
        if (!userInstance) {
            flash.message="${message(code:'default.not.found.message',
```

```
        args: [message(code: 'user.label', default: 'User'), params.id])}"  
        redirect(action: "list")  
    } else {  
        [userInstance: userInstance]  
    }  
}
```

要想测试这个 Action，你需要在调用这个 Action 前正确产生 params HashMap。

```
class UserControllerTests extends ControllerUnitTestCase {  
  
    void testShow(){  
        def jdoe = new User(login:"jdoe")  
        def suziq = new User(login:"suziq")  
        mockDomain(User, [jdoe, suziq])  
  
        controller.params.id = 2  
        def map = controller.show()  
        assertEquals "suziq", map.userInstance.login  
    }  
}
```

在清楚如何把正确的值填入 params 之后，你现在就可以给控制器写一个针对 authenticate Action 的测试了：

```
void testAuthenticate(){  
    def jdoe = new User(login:"jdoe", password:"password")  
    mockDomain(User, [jdoe])  
  
    controller.params.login = "jdoe"  
    controller.params.password = "password"  
    controller.authenticate()  
    assertNotNull controller.session.user  
    assertEquals "jdoe", controller.session.user.login  
  
    controller.params.password = "foo"  
    controller.authenticate()  
    assertTrue controller.flash.message.startsWith( "Sorry, jdoe")  
}
```

至于得到的 session 和 flash HashMap，你可以像传入的 params HashMap 一样去对待。

这里是 ControllerUnitTestCase 中模拟出的元素的完整列表：

- controller.request
 - controller.response
 - controller.session

- controller.flash
- controller.redirectArgs
- controller.renderArgs
- controller.template
- controller.modelAndView

好了，现在你已经知道了如何测试你的 authenticate Action，让我们回过头去保护这个应用。下一个改进：去掉保存在数据库中的口令明文。

创建口令编解码器（Codec）

编解码器（Codec）（编码器-解码器的缩写）是转换字符串的一种方法。Grails 提供了大量便捷的编解码器，而且已经元编程到了所有字符串实例上。

例如：

- "<p>Hello</p>".encodeAsHTML() 返回 <p&gtHello</p&gt
- "You & Me".encodeAsURL() 返回 You%26+Me
- "ABC123".encodeAsBase64() 返回 QUJDMTIz

相应的还有 decodeHTML()、decodeURL() 和 decodeBase64() 方法完成以上转换的逆操作。

要想创建自己的编解码器，只需在 grails-app/utils 目录创建一个以 Codec 结尾的文件即可。例如，假设你想将空格转换成下划线。要完成这一工作，创建 UnderscoreCodec.groovy，然后增加以下代码：

```
class UnderscoreCodec {
    static encode = {target->
        target.replaceAll(" ", "_")
    }

    static decode = {target->
        target.replaceAll("_", " ")
    }
}
```

之后，你就可以在整个应用中调用 "Hello World".encodeAsUnderscore() 和 "Hello_World".decodeUnderscore() 了。

那么编解码器如何帮助我们完成对口令的加密呢？创建 grails-app/utils/SHACodec.groovy，然后增加下列代码：

```
import java.security.MessageDigest
```

```
class SHACodec{
    static encode = {target->
        MessageDigest md = MessageDigest.getInstance('SHA')
        md.update(target.getBytes('UTF-8'))
        return new String(md.digest()).encodeAsBase64()
    }
}
```

正如你等下会看到的，我们真正需要做的就只是编码口令。当口令明文传入时，我们只需对其进行哈希操作，然后将该结果与保存在数据库中的哈希值进行比较即可。缺少 decode 闭包意味着，数据库中的已编码口令没有被解码成其对应明文的危险——不论是有意还是无意。

应用这个新创建的编解码器有 2 个地方。首先，给 User.groovy 增加 beforeInsert 闭包：

```
class User {
    String login
    String password
    String role = "user"

    static constraints = {
        login(blank:false, nullable:false, unique:true)
        password(blank:false, password:true)
        role(inList:["admin", "user"])
    }

    static transients = ['admin']

    boolean isAdmin(){
        return role == "admin"
    }

    def beforeInsert = {
        password = password.encodeAsSHA()
    }

    String toString(){
        login
    }
}
```

这确保口令总是会被加密，无论你是通过 BootStrap.groovy，还是 UserController 里的 create Action 或是其他什么方式保存用户。启动 Grails，然后验证数据库中的口令已经被执行了哈希操作。

```
mysql> select * from user;
```

id	login	password	role
1	admin	KTkJTzWjut8qiQdougNPpesW6V4=	admin
2	jdoe	W6ph5Mm5Pz8GgiULbPgzG37mj9g=	user

接下来，修改 authenticate Action，对已哈希的口令进行检查，而非口令明文。

```
class UserController {

    def authenticate = {
        def user = User.findByLoginAndPassword(
            params.login, params.password.encodeAsSHA())
        if(user){
            session.user = user
            flash.message = "Hello ${user.login}!"
            redirect(controller:"race", action:"list")
        }else{
            flash.message = "Sorry, ${params.login}. Please try again."
            redirect(action:"login")
        }
    }
}
```

口令明文在我们的应用消失了，嗯？放心地在你运行的 Grails 应用中去测试这个功能。我会在这等着。

当然，你现在还得修改你的 test/unit/UserControllerTests.groovy 文件。毕竟，我们不愿意把失败的测试丢得满地都是，把你并不完美的应用搞得乱七八糟，是不是？记住，这是单元测试，编解码器不会在单元测试中被元编程到字符串上去。

要升级单元测试，你需要完成 3 件事：导入 Grails 编解码器所在包，在 setUp() 方法中手工把编解码器元编程到字符串上，在 testAuthenticate() 中把模拟出的口令修改成经过 SHA 编码的。

```
import grails.test.*
import org.codehaus.groovy.grails.plugins.codecs.*

class UserControllerTests extends ControllerUnitTestCase {
    protected void setUp() {
        super.setUp()
        String.metaClass.encodeAsBase64 = {->
            Base64Codec.encode(delegate)
        }
    }
}
```

```

String.metaClass.encodeAsSHA = {->
    SHACodec.encode(delegate)
}
}

// ...
void testAuthenticate(){
    def jdoe = new User(login:"jdoe", password:"password".encodeAsSHA())
    mockDomain(User, [jdoe])

    controller.params.login = "jdoe"
    controller.params.password = "password"
    controller.authenticate()
    assertNotNull controller.session.user
    assertEquals "jdoe", controller.session.user.login

    controller.params.password = "foo"
    controller.authenticate()
    assertTrue controller.flash.message.startsWith("Sorry, jdoe")
}
}

```

输入 grails test-app , 确保所有的测试现在都通过了。

接下来 , 让我们在首部提供一个链接 , 简化用户的登录和注销。

创建认证标签库

大多数公共网站都在首部的右上角有一个不醒目的登录链接。我们可以利用在 Groovy 服务器页面一章中学到的标签库技巧完成同样的事情。

输入 grails create-tag-lib Login , 然后将下列代码加到 grails-app/taglib/LoginTagLib.groovy 中 :

```

class LoginTagLib {
    def loginControl = {
        if(request.getSession(false) && session.user){
            out << "Hello ${session.user.login} "
            out << """[${link(action:"logout", controller:"user")}""]"""
        } else {
            out << """[${link(action:"login", controller:"user")}""]"""
        }
    }
}

```

下一步就是把该标签库加到 grails-app/views/layout/_header.gsp :

```

<div id="header">
    <p>

```

```

<a class="header-main" href="#">  

    RaceTrack  

</a>  

</p>  

<p class="header-sub">When's your next race?</p>  

<div id="loginHeader">  

    <g:loginControl />  

</div>  

</div>

```

再给 web-app/css/main.css 里加点 CSS，让它变得好看点：

```

#loginHeader {  

    float: right;  

    color: #fff;  

}  

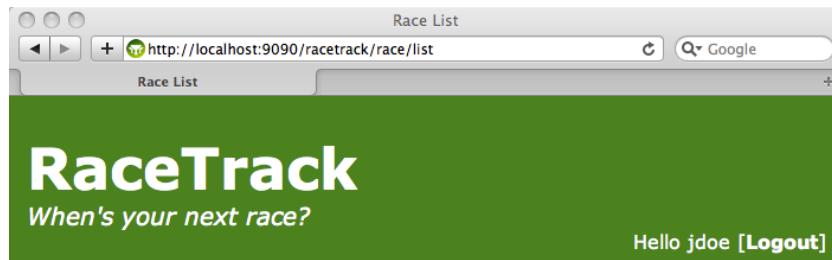
#loginHeader a{  

    color: #fff;  

}

```

再次重启 Grails，看看它现在的样子：



Race List

Race List						
i Hello jdoe!						
Id	Name	Start Date	City	State	Distance	
1	Turkey Trot	2009-12-02 00:40:45.0	Duck	NC	5	

© 1999 - 2009 Racetrack, Inc.

容易吧？你还可以走得更远，增加你自己的<g:loggedIn>标签、<g:isAdmin>标签，等等。但我认为你要抓住事情的要点。

接下来，让我们对 UserController 进行控制，只允许管理员创建新用户。

利用beforeInterceptor

我们完全可以让 UserController 中的每个 Action 都充斥着千篇一律的 if(session.user.admin) 判断，但是有一种更简单的方法可以确保让这个功能只能由管理员使用。我们可以设置

beforeInterceptor 代码块——顾名思义——它会在执行每个 Action 之前被调用（Grails 同时还提供了 afterInterceptor）。

把下列代码加到 UserController 中：

```
class UserController {

    def beforeInterceptor = [action:this.&debug]

    def debug(){
        println "DEBUG: ${actionUri} called."
        println "DEBUG: ${params}"
    }

    def logout = {
        flash.message = "Goodbye ${session.user.login}"
        session.user = null
        redirect(action:"login")
    }

    // ...
}
```

注意 debug 和 logout 之间有一个细微但却重要的区别。形如 logout 这样的闭包会在 URL 中暴露给最终用户。但像 debug()这样的方法是私有的，只能被类中的其他闭包或方法调用。（你可以通过末尾的括号来判断 debug()是一个方法。）

beforeInterceptor 中的&符号是方法指针，它是由 Groovy 提供的。这大体意味着，debug()将会在调用每个 Action 之前被调用。

启动Grails并访问 <http://localhost:9090/racetrack/user>。你应该会在启动Grails的控制台上看到以下输出：

```
DEBUG: /user/index called.
DEBUG: [controller:user]
DEBUG: /user/list called.
DEBUG: [action:list, controller:user]
```

正如预期的，隐式的 index 的调用被重定向到了 list，导致了 debug()方法的两次调用。

现在，我们让 beforeInterceptor 指向稍微有用点的方法。把下列代码加到 UserController：

```
class UserController {

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

    def auth() {
        if(!session.user) {
```

```

        redirect(controller:"user", action:"login")
        return false
    }

    if(!session.user.admin){
        flash.message = "Tsk tsk-admins only"
        redirect(controller:"race", action:"list")
        return false
    }

}

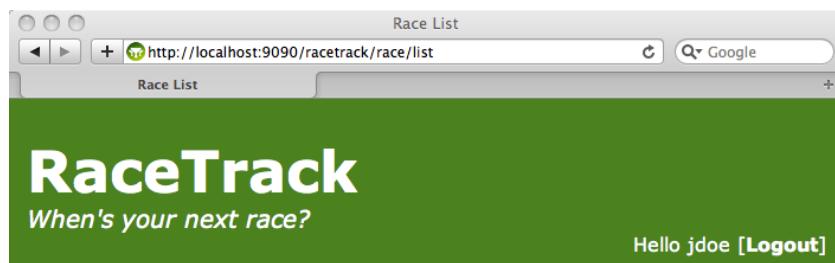
// ...
}

```

瞧，这次我们把 `beforeInterceptor` 指到了 `auth()` 方法，它首先检查你是否已登录，然后检查你是否为管理员（因为我们只想让管理员能够管理用户数据）。

`except` 列表允许对 `login`、`logout` 和 `authenticate` Action 不进行控制（即，这个过滤器会应用到这个控制器中的所有 Action，除了这 3 个 Actions）。（除了 `except`，你也可以使用 `only` 声明这个过滤器要应用的 Action 列表。）

启动Grails，然后访问 <http://localhost:9090/racetrack/user>。验证在第一次访问时，你会被重定向到登录页面。接下来，使用`jdoe/password`登录。因为该用户没有管理员权限，你应该被重定向，并得到口吻温和的责备：



Race List

InfoQ

Race List						
Tsk tsk -- admins only						
Id	Name	Start Date	City	State	Distance	
1	Turkey Trot	2009-12-02 00:45:39.0	Duck	NC	5	

© 1999 - 2009 Racetrack, Inc.

最后，使用 `admin/wordpass`，验证作为管理员，你可以随心所欲地使用 `UserController` 中的任何 Action。

我们在本章要做的最后一件事情是，从与单个控制器打交道更进一步，把安全应用到整个所有控制器。要完成这一点，我们需要创建过滤器。

利用过滤器（Filter）

`beforeInterceptor` 可以让你以每个控制器为基础拦截请求。过滤器可以让你以整个应用程序范围为基础做同样的事情。在命令行中输入 `grails create-filters Admin`。然后在文本编辑器里打开 `grails-app/conf/AdminFilters.groovy`。

```
class AdminFilters {

    def filters = {
        all(controller:'*', action:'*') {
            before = { }
            after = { }
            afterView = { }
        }
    }
}
```

`filters` 闭包内定义了一个缺省的过滤器，名字是 `all`。如你所见，它将拦截对任意控制器、任意 Action 的请求。在 `all` 内部，你可以创建 `before`（在 Action 执行前被调用，类似 `beforeInterceptor`）、`after`（在 Action 执行后但视图显示之前被调用）或 `afterView`（在视图显示之后被调用）。

你可以在 `filters` 闭包内创建任意多的条目。在这个例子中，为了更好地说明它的用途，我把 `all` 过滤器更名为 `adminOnly`——任何可能更新记录的调用都被限制为登录的管理员才能操作。我还删除了没有用到的 `after` 和 `afterView` 闭包。这里是最终的结果：

```
class AdminFilters {
    def filters = {
        adminOnly(controller:'*', action:"(create|edit|update|delete|save)") {
            before = {
                if(!session?.user?.admin){
                    flash.message = "Sorry, admin only"
                    redirect(controller:"race", action:"list")
                    return false
                }
            }
        }
    }
}
```

一个经常需要留意的地方是你为过滤器定义多个 Action 的方式。在 beforeInterceptor 中，你传入的是 Action 列表。在过滤器中，它们使用正则表达式来指定。除了这个区别，你可以在过滤器中完成所有你能在拦截器中做的事情。

安全插件

我们到现在还没有讨论插件。（别着急——我们将在下一章深入这一主题），但是Grails让新增一个全新的功能块变得异常简单，只要输入grails install-plugin。（你可以输入grails list-plugins或访问 <http://grails.org/plugins>了解各种可用的插件。）

我在这第一章的目标是向你展示可以用来保护应用的核心机制——拦截器、过滤器等。象RaceTrack这样的简单程序，我们这种闭门造车的方式就够了。但若是你的安全需求更复杂，或者你想与外部系统进行交互——如LDAP（<http://grails.org/plugin/ldap>）或（<http://grails.org/plugin/openid>）——你可能会觉得安装一个已经存在的安全插件是更好的方法。

有很多流行的安全插件可供你使用。我推荐你去访问 <http://grails.org/plugins>，然后找到最适合你需求的那个。不同的网站需求不同，因此安全根本就没有一劳永逸的解决方案。

例如，假设你要运营一个面向大众的网站，期望提供通过电子邮件启用“自注册”功能，你可能会觉得Authentication插件（<http://grails.org/plugin/authentication>）很有意思。它对普遍的“你能把我忘记的口令用邮件发给我吗？”这一需求提供了开箱即用的支持，而且还有不少其他功能。

对于更传统、基于内联网、封闭的安全应用，JSecurity插件（<http://grails.org/plugin/jsecurity>）同时具备了简单和功能强大的特性。它由Grails开发者核心团队编写和支持，这样，你可以确信，从长期看，它不会处于无人看管的状态。

要是你已经具备了Spring Security框架（以前被称为ACEGI）的经验，这里也有一个插件适合你（<http://grails.org/plugin/acegi>）。

不论选择哪个插件，你大可放心，它们都使用了你在本章中学到的相同技术。它们都提供了用于认证的用户帐号、用于授权的某种形式的角色，以及用于规范用户行为的标签库、拦截器和过滤器，避免他们访问不该访问的功能。

10 插件、服务及部署

在这一章里，我们将探索 Grails 的插件生态系统。同时，你将学到另一个核心 Grails 技术：服务。最后，你将创建一个 WAR 文件，它可以部署到 Tomcat、JBoss、GlassFish 或其他 JEE 应用服务器上。

理解插件

Java 开发者习惯于通过把 JAR 放到 CLASSPATH 中来混入新的功能。在本书的前面章节，你并不需要写一个自己的 MySQL JDBC 驱动器才能和数据库交互——你只需把合适的 JAR 放到 lib 目录下就行了。

Grails 插件就像是打了激素的 JAR。它们不仅可以包含一或多个 JAR，而且还可以包含 GSP、控制器、标签库、编解码器等组件。从本质上讲，Grails 插件就是迷你 Grails 应用，专门用于和其他 Grails 应用搭配使用。

在上一章的最后，我们接触到了一些在 <http://grails.org/Plugins> 上可以找到的安全插件。但是，插件并不仅限于安全的使用。有些插件提供了另一种表现层，如 Flex (<http://grails.org/plugin/flex>) 和 Google Web Toolkit (<http://grails.org/plugin/gwt>)。有些插件则包含了其他测试工具，如 Selenium (<http://grails.org/plugin/selenium>) 和 Cobertura (<http://grails.org/plugin/code-coverage>)。但是，不论哪种情况，插件都是由你平常用来创建自己应用的熟悉的 Grails 组件组成的。

安装**Searchable**插件

让我们给 RaceTrack 增加一些搜索功能。若是将搜索范围限制在单个类和一两个字段，利用已经掌握的 GORM 动态查找器的知识，你足以组装出一个相当优雅的搜索特性。

例如，假设你想提供一个搜索某城市比赛的表单。假如表单已经有一个名字叫 query 的文本框，下面的这个控制器就能够提供搜索结果：

```
def search = {
    def results = Race.findAllByCity(params.query)
    return [searchResults:results]
}
```

要是想根据 city 和 state 进行搜索，你还可以把查询修改为 Race.findAllByCityAndState(params.city, params.state)。

但如果你想搜索多于两个字段呢？（动态查找器只限于一两个字段。）再进一步，假如你想跨多个类（如 Race、Runner、Sponsor 等）搜索城市，怎么办？

你完全可以卷起袖子，为自己打造一个全面的搜索工具，或者你可以输入 grails install-plugin searchable，享受别人辛苦劳动的成果。鉴于懒惰是软件开发者的一种美德——我们把它称为代码重用，好让非程序员听起来顺耳一点——我们将采用后一种方式。

安装新插件时，控制台会有大量的活动。下面是实际活动的概览。

```
$ grails install-plugin searchable
...
Reading remote plugin list ...
Plugin list out-of-date, retrieving..
[get] Getting: http://svn.codehaus.org/grails/trunk/grails-plugins/.plugin-meta/plugins-list.xml
[get] To: /Users/sdavis/.grails/1.2.0/plugins-list-core.xml

Reading remote plugin list ...
[get] Getting: http://plugins.grails.org/.plugin-meta/plugins-list.xml
[get] To: /Users/sdavis/.grails/1.2.0/plugins-list-default.xml
```

一开始，你会看到 Grails 会向母舰发出两个回调，得到当前的插件列表。核心列表包含了必需的插件——没有它们 Grails 就无法运行。（在撰写本书的时候，这种插件只有一个——支持 GORM 的 Hibernate 插件。）另一个列表——缺省（default）——包含了象 searchable 这样的可选插件。

这里有一个 plugins-list-default.xml 的片段，它描述了 searchable 插件。

```
<plugin latest-release="0.5.5" name="searchable">
  <release tag="RELEASE_0_5_5" type="svn" version="0.5.5">
    <title>Adds rich search functionality to Grails domain models.
      This version is recommended for JDK 1.5+</title>
    <author>Maurice Nicholson</author>
    <authorEmail>maurice@freeshell.org</authorEmail>
    <description>Adds rich search functionality to Grails
      domain models.
      Built on Compass (http://www.compass-project.org/)
      and Lucene (http://lucene.apache.org/)
      This version is recommended for JDK 1.5+
    </description>
    <documentation>
      http://grails.org/Searchable+Plugin
    </documentation>
    <file>http://plugins.grails.org/grails-searchable/tags/RELEASE\_0\_5\_5/grails-searchable-0.5.5.zip
```

```
</file>
</release>

<!-- snip -->
</plugin>
```

如你所见，存在着关于 searchable 插件的大量精细元数据——作者、当前版本，以及（最重要的）下载地址。获得最后一个信息后，Grails 接下来的动作就毫无悬念了：

```
$ grails install-plugin searchable
... download core and default plugin lists

... continued ...
[get] Getting: http://plugins.grails.org/
  grails-searchable/tags/RELEASE_0_5_5/
  grails-searchable-0.5.5.zip
[get] To: /Users/sdavis/.grails/1.2.0/plugins/
  grails-searchable-0.5.5.zip
```

插件被下载到了你的主（Home）目录下的.grails 目录。下一次在其他项目中安装 searchable 插件时，Grails 会检查是否存在最新版本，若是没有，Grails 就从.grails 缓存中安装，而不是从互联网。

一旦插件安装完毕，下一步就是把它复制到你的项目中：

```
$ grails install-plugin searchable

... continued ...
[copy] Copying 1 file to /Users/sdavis/.grails/1.2.0/projects/racetrack/plugins
Installing plug-in searchable-0.5.5
[mkdir] Created dir: /Users/sdavis/.grails/1.2.0/projects/racetrack/plugins/searchable-
0.5.5
[unzip] Expanding: /Users/sdavis/.grails/1.2.0/plugins/grails-searchable-0.5.5.zip into
/Users/sdavis/.grails/1.2.0/projects/racetrack/plugins/searchable-0.5.5
```

注意，Grails 并没有把它复制到你的本地源代码目录。它把插件复制到了.grails 目录，编译后代码存放的位置。我们等下就会探索该目录的详细情况。至于现在，让我们完成对控制台输出的探索。

```
$ grails install-plugin searchable

... continued ...
Thanks for installing the Grails Searchable Plugin!

Documentation is available at http://grails.org/Searchable+Plugin

Help is available from user@grails.codehaus.org
```

Issues and improvements should be raised at
<http://jira.codehaus.org/browse/GRAILSPLUGINS>

If you are upgrading from a previous release, please see
<http://grails.org/Searchable+Plugin+-+Releases>

Plugin searchable-0.5.5 installed

Plug-in provides the following new scripts:

`grails install-searchable-config`

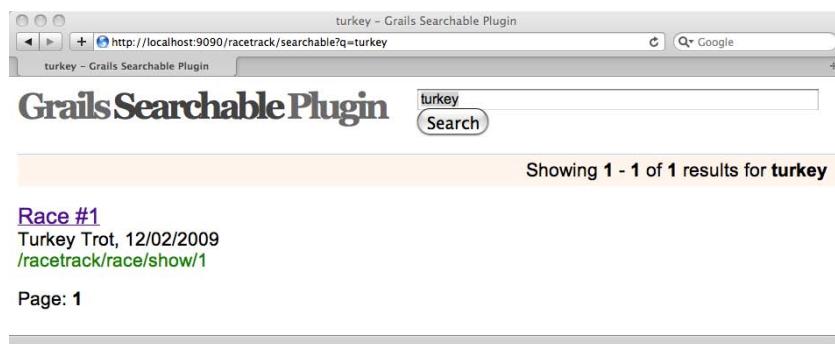
安装最终以一组对使用该插件有帮助的链接结束：详细文档的链接、提交特性请求的链接，以及寻求帮助的电子邮件地址。

你会很高兴地知道，同只需一行安装命令即可安装插件的难度相当，只需一行代码就可以让领域类变成可搜索的。

打开 grails-app/domain/Race.groovy，然后在其中输入一行代码：

```
class Race {
    static searchable = true
    // ...
}
```

现在，启动Grails，访问：<http://localhost:9090/racetrack/searchable>。确认你可以搜索保存在任意Race记录中的任意字段的任意单词。（尽管放心地添加些更多的Races，好让搜索变得更有趣一点。）



一行代码的投资回报还不坏，嗯？

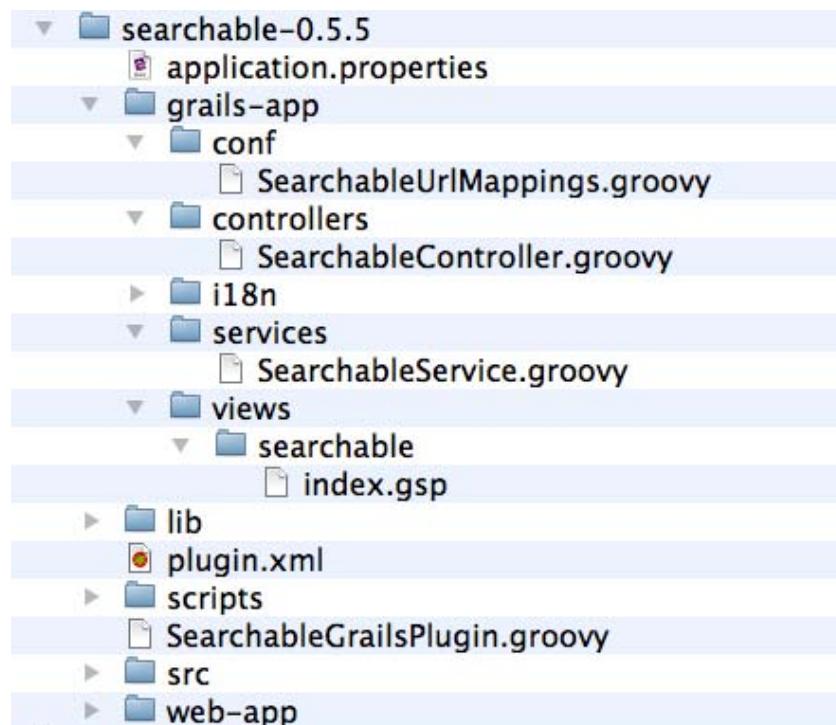
就目前的情况而言，搜索特性对任何人都是开放使用的。这意味着，在给 Runner 或 Registration 类添加 `static searchable = true` 的时候，可能最好不要包含其中的私人信息。但是，如果你想让那个特性成为管理员拥有的附加特性，你可以增加隐私数据和一点安全性来保护它。（你可以把对 Searchable 控制器的安全检查加到 grails-app/conf/AdminFilters.groovy 中，这就可以轻易地锁上这一特性。）

既然确信 searchable 插件已如所吹嘘的那样工作，接下来我们要做的就是对它进行些定制。这个外观并不符合我们的要求，我立刻看到了另一个自定义标签库出现的需要。

探索Searchable插件

就所掌握的Grails知识，若是能访问 <http://localhost:9090/racetrack/searchable>，那么就应该有一个SearchableController.groovy文件在grails-app/controllers中。但是在检查自己的本地源代码树之后，你并没有找到这个SearchableController。

在想起插件被安装到了主目录中那个神奇的 .grails 目录之后，你检查了.grails/1.2.0/projects/racetrack/plugins，应该发现这就是你一直在找的一—熟悉的 Grails 目录结构。



正如你可能会猜到的，搜索表单能够在 grails-app/views/searchable/index.gsp 里找到：

```

<g:form url='[controller: "searchable", action: "index"]'
    id="searchableForm"
    name="searchableForm"
    method="get">
    <g:textField name="q" value="${params.q}" size="50"/>
    <input type="submit" value="Search" />
</g:form>

```

它把查询提交到了 SearchableController 中的 index Action：

```

import org.compass.core.engine.SearchEngineQueryParseException

class SearchableController {

```

```

def searchableService

def index = {
    if (!params.q?.trim()) {
        return [:]
    }
    try {
        return [searchResult: searchableService.search(params.q, params)]
    } catch (SearchEngineQueryParseException ex) {
        return [parseException: true]
    }
}
// ...
}
  
```

很明显，index 闭包对 params.q 字段非常感兴趣。如果参数不存在，Action 就返回一个空的 HashMap。反之，倘若参数存在，它就返回 searchableService.search()方法调用的结果。

这里要问个问题，“什么是服务？”（我很高兴你能提问。往下看！）

理解服务

控制器通常关注于单个领域类的管理和维护。相反，服务则是应用到多个领域类的业务逻辑和行为的存放地点。

是否注意到了 SearchableService（保存在 grails-app/services）被添加到 SearchableController 中的方式？在控制器类声明下的那行 def searchableService 完成了这一戏法。这就是那些酷小孩们说的依赖注入（**dependency injection**）或者简称“DI”。Spring framework（Grails 的另一种核心技术之一）是处理它的行家里手。

其实，让我们看看 SearchableService 的源码：

```

class SearchableService {
    boolean transactional = true
    def compass
    def compassGps
    def searchableMethodFactory

    def search(Object[] args) {
        searchableMethodFactory.getMethod("search").invoke(*args)
    }

    // ...
}
  
```

你立刻就能发现另几个被注入到服务中的类：compass、compassGps 和 searchableMethodFactory。你可以在 src/java 和 src/groovy 里找到它们的实现。

但是不要急于去查看实现细节，现在让我们会回到手头的任务：给我们的应用首部添加搜索框。

增加搜索框

鉴于目前唯一可搜索的类是 Race，那我们就把 search Action 添加到 RaceController 当中。
(要是想在以后重构搜索功能，使之包含多个类，你可能也应该创建一个更通用的 SearchController。)

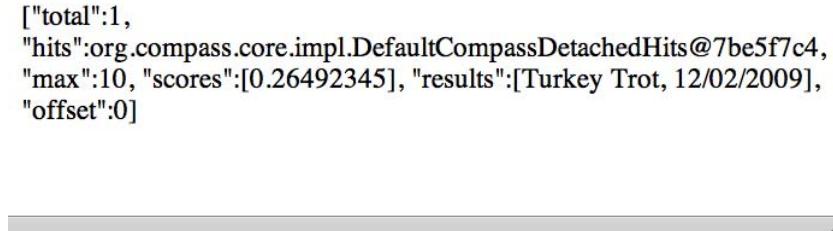
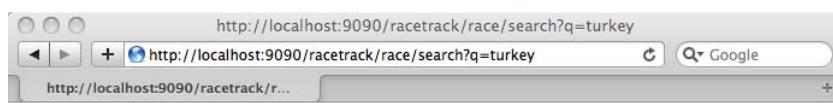
把 SearchableService 注入到 RaceController 的方式和把它注入到 SearchableController 的方式相同，但假如你是在单个类上进行搜索，有一个更简单的方法。Searchable 插件通过元编程给每个领域类都添加了一个 search()方法。这意味着搜索的首轮实现可能看上去是下面这个样子：

```
class RaceController {
    def scaffold = Race

    def search = {
        render Race.search(params.q, params)
    }
}
```

花点功夫思考一下就这么点代码究竟产生了多大效果。相当神奇，不是吗？

测试一下，输入 <http://localhost:9090/racetrack/race/search?q=turkey>。



不错，它起作用了，但是这个界面只有开发人员才会喜欢。我们要给它包装成一个稍微友好的界面。

首先，在 grails-app/views/layouts 中创建一个局部模板，名字叫_raceSearch.gsp。

```
<div id="search">
    <g:form url="[controller: "race", action: "search"]"
        id="raceSearchForm"
        name="raceSearchForm"
```

```

method="get">
<g:textField name="q" value="${params.q}" />
<input type="submit" value="Find a race" />
</g:form>
</div>

```

接下来，把它加到相同目录下的_header.gsp 模板首部：

```

<g:render template="/layouts/raceSearch" />

<div id="header">
<p>
    <a class="header-main" href="${resource(dir: '')}">RaceTrack</a>
</p>

<p class="header-sub">When's your next race?</p>
<div id="loginHeader">
    <g:loginControl />
</div>
</div>

```

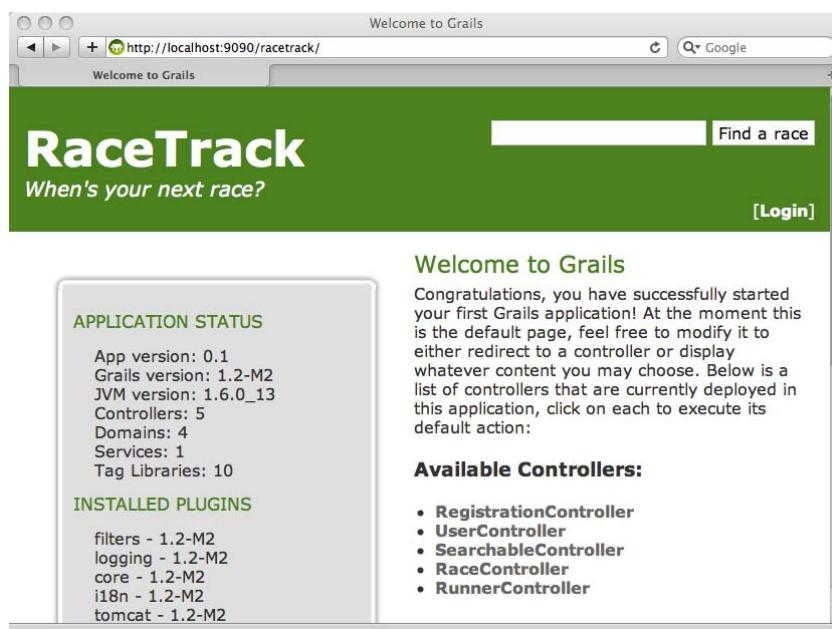
在 web-app/css/main.css 里添加点 CSS，好让搜索框位于屏幕的正确位置：

```

#search {
    float: right;
    margin: 2em 1em;
}

```

现在，访问 <http://localhost:9090/racetrack>，看看目前的工作成果：



很好，前端看起来相当不错。但是后端仍然同之前一样丑陋不堪。让我们给搜索结果创建一个结果页。给它起个名字，grails-app/views/race/list.gsp，这样，我们就可以把它同时用于常规的 list 视图和 search 结果。

```
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
        <meta name="layout" content="main" />
        <title>RaceTrack</title>
    </head>
    <body>
        <div class="body">
            <g:if test="${flash.message}">
                <div class="message">${flash.message}</div>
            </g:if>

            <g:each in="${raceInstanceList}" status="i" var="raceInstance">
                <div class="race">
                    <h2>${raceInstance.name}</h2>
                    <p class="race-details">
                        <span class="question">When?</span>
                        <span class="answer">
                            ${raceInstance.startDate.
                                format("EEEE, MMMM d, yyyy")}</span>
                        </p>
                        <p class="race-details">
                            <span class="question">Where?</span>
                            <span class="answer">
                                ${raceInstance.city}, ${raceInstance.state}</span>
                            </p>
                            <p class="race-details">
                                <span class="question">How Long?</span>
                                <span class="answer">
                                    <g:formatNumber
                                        number="${raceInstance.distance}"
                                        format="0 K" /></span>
                                </p>
                                <p class="race-details">
                                    <span class="question">How Much?</span>
                                    <span class="answer">
                                        <g:formatNumber
                                            number="${raceInstance.cost}"
                                            format="\$###,##0" /></span>
                                    </p>
                                </div>
                            </g:each>
```

```

<div class="paginateButtons">
    <g:paginate total="${raceInstanceTotal}" />
</div>
</div>
</body>
</html>

```

再给 web-app/css/main.css 添加点 CSS，改善一下它的外观：

```

.race {
    padding: 1em;
}

.race-details {
    padding-left: 2em;
}

.question {
    font-style: italic;
    font-weight: bold;
}

```

下一步，调整 RaceController.search 中的代码：

```

class RaceController {
    def scaffold = Race

    def search = {
        flash.message = "Search results for: ${params.q}"
        def resultsMap = Race.search(params.q, params)
        render(view:'list',
            model:[
                raceInstanceList:resultsMap.results,
                raceInstanceTotal:Race.countHits(params.q)
            ]
        )
    }
}

```

最后，给 grails-app/conf/BootStrap.groovy 加一些更多的 Race，让事情变得更有意思一点。

```

def burner = new Race(
    name:"Barn Burner",
    startDate:(new Date() + 120),
    city:"Cary",
    state:"NC",
    distance:10.0,
    cost:15.0,
)

```

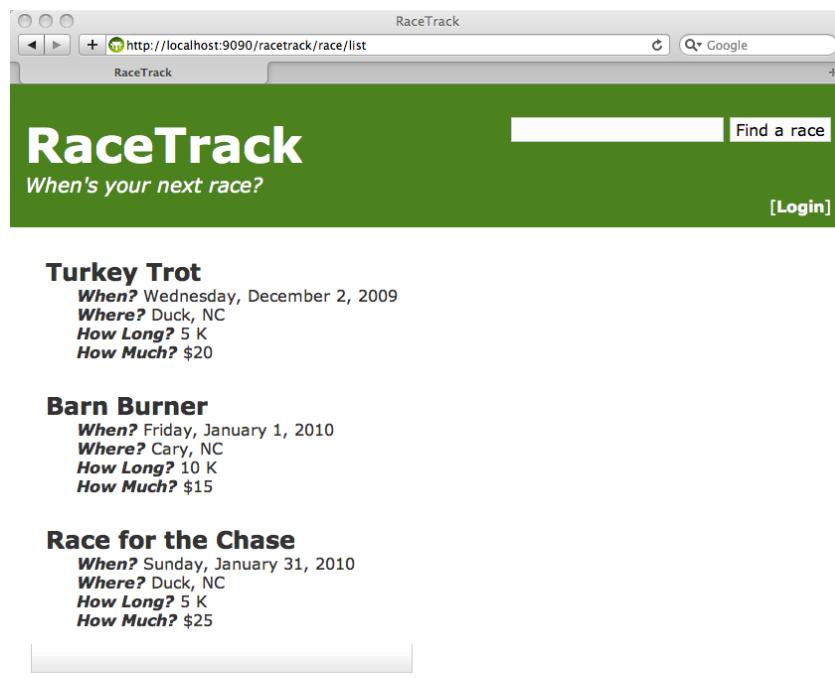
```

        maxRunners:350
    )
burner.save()
if(burner.hasErrors()){
    println burner.errors
}

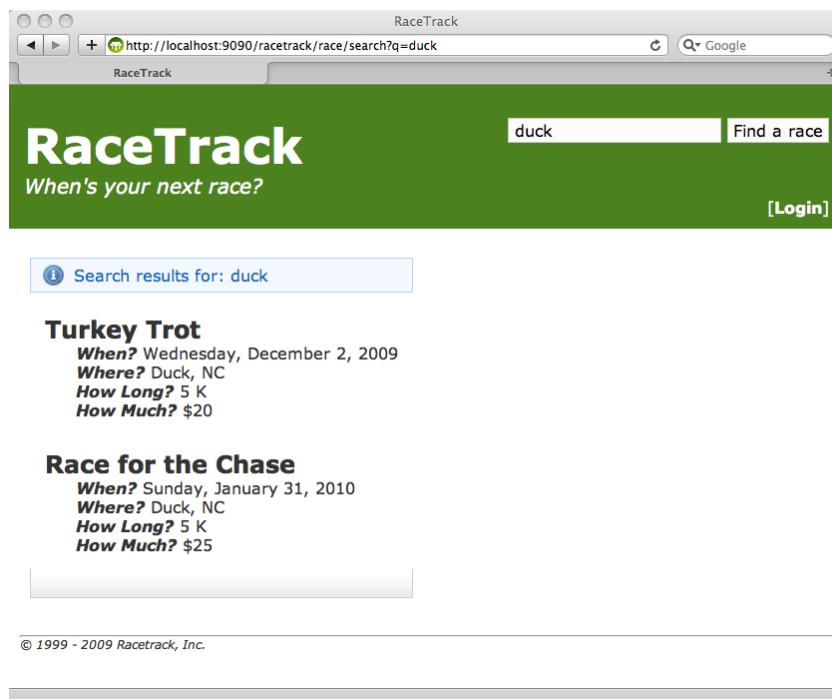
def chase = new Race(
    name:"Race for the Chase",
    startDate:(new Date() + 150),
    city:"Duck",
    state:"NC",
    distance:5.0,
    cost:25.0,
    maxRunners:350
)
chase.save()
if(chase.hasErrors()){
    println chase.errors
}

```

让我们看看最后的结果，在Web浏览器中访问：<http://localhost:9090/racetrack/race>。



并且，作为稍微正式一点的测试，那就搜一些东西，然后看看结果：



代码重用得很不错，嗯？

我们的新 list/search 页面看上去太好了，我想要把它用来作为我的新主页，而不是单调的“Grails 欢迎您” index.gsp。但是，我还想在一旁保留这个单调的页面——它作为管理页面正合适。让我们最后完成对 RaceTrack 应用的调整，然后结束这个应用。

使用URL映射（UrlMapping）改变主页

我们已经花了相当多的时间来吹捧“惯例优于配置”的各种好处了。但是，规则是用来打破的，这包括 URL 映射到底层 Grails 控制器上的方式。很幸运，grails-app/conf/UrlMappings.groovy 可以让我们心想事成。

```
class UrlMappings {
    static mappings = {
        "/$controller/$action?/$id?" {
            constraints { // apply constraints here }
        }

        "/"(view:"/index")
        "500"(view:'/error')
    }
}
```

正如你看到的，第一个映射负责把 <http://localhost:9090/racetrack/race/show/2> 映射到 params.controller、params.action 和 params.id。但是第二个映射就是我们这次的目标。它负责把 <http://localhost:9090/racetrack/> 的末端斜杠映射到 index.gsp。改动该映射，使之指向 RaceController list Action，你就会得到一个新主页：

```

class UrlMappings {
    static mappings = {
        "/$controller/$action?/$id?" {
            constraints { // apply constraints here }
        }

        "/"(controller:"race", action:"list")
        "500"(view:'/error')
    }
}
  
```

把这完成之后，让我们重新把现有的 index.gsp 页面改动到管理控制台。创建 grails-app/controllers/AdminController.groovy：

```

class AdminController{
    def beforeInterceptor = [action:this.&auth]

    def auth() {
        if(!session.user) {
            redirect(controller:"user", action:"login")
            return false
        }
        if(!session.user.admin){
            flash.message = "Tsk tsk-admins only"
            redirect(controller:"race", action:"list")
            return false
        }
    }

    def index = {}
}
  
```

看到文件末尾那个空空如也的 index Action 了吗？把 grails-app/views/index.gsp 移动到 grails-app/views/admin/index.gsp，这样你就有了一个非常不错的管理控制台。这个页面尤其不错的是，如你在整本书中所见，它会自动给每个新增的控制器加一个链接。没有什么比一个零成本的管理控制台更好的了：

```

<ul>
    <g:each var="c" in="${grailsApplication.controllerClasses}">
        <li class="controller">
            <g:link controller="${c.logicalPropertyName}">
                ${c.fullName}
            </g:link>
        </li>
    </g:each>
</ul>
  
```

你可能甚至想回到 UserController，使这个页面成为管理员认证后的缺省页面：

```
class UserController{
    // ...
    def authenticate = {
        def user = User.findByLoginAndPassword(
            params.login, params.password.encodeAsSHA())
        if(user){
            session.user = user
            flash.message = "Hello ${user.login}!"
            if(user.admin){
                redirect(controller:"admin", action:"index")
            } else{
                redirect(controller:"race", action:"list")
            }
        } else{
            flash.message = "Sorry, ${params.login}. Please try again."
            redirect(action:"login")
        }
    }
}
```

产品部署检查清单

我们准备开始产生 WAR 文件，把它部署到 Tomcat、JBoss、GlassFish 或是你选择的应用服务器。但在做之前，有几件事情需要你再次确认。

最需要确认的是 grails-app/conf/DataSource.groovy 中的数据库连接。确保让 Grails 指向了正确的数据库（最好不要与测试和开发模式用的数据库相同）。要是待部署应用服务器可以通过 JNDI 提供数据库连接，你可以把用户名、口令和 url 各变量换成一个 jndiName 变量：

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

接下来，检查 grails-app/conf/BootStrap.groovy 中任何与产品有关的设置。这个文件一般只用于开发模式，但你总是可以给你的产品应用创建管理员账户和查找（lookup）数据。

```
import grails.util.GrailsUtil

class BootStrap {
    def init = { servletContext ->
        switch(GrailsUtil.environment){
            case "development":
```

```

// ...
break

case "production":
// 可选的，在此创建产品数据
break
}
}

def destroy = { }

}

```

下一步，在 grails-app/conf/Config.groovy 中有一个 environments 配置块。你可以在这里设置服务器的 URL，以及任何其他系统范围的值。

```

// set per-environment serverURL stem for creating absolute links
environments {
  production {
    grails.serverURL = "http://www.changeme.com"
    the.word.is = "bird"
  }
}

```

假如需要在控制器或标签库里访问这些变量，你只需要调用 grailsApplication.config.grails.serverURL 或 grailsApplication.config.the.word.is 就行了。在其他情况下，你需要显式地导入 ConfigurationHolder 帮助类。

```

import org.codehaus.groovy.grails.commons.*

def grailsApplication = ConfigurationHolder
println grailsApplication.config.grails.serverURL

```

接着，在 application.properties 中修改 app.name 和 app.version 的值。这些值都要用来给 WAR 文件命名。

```

#utf-8
#Fri Jul 03 08:57:11 MDT 2009
app.version=0.1
app.servlet.version=2.4
app.grails.version=1.2.0
plugins.searchable=0.5.5
plugins.hibernate=1.2.0
plugins.tomcat=1.2.0
app.name=racetrack

```

最后，你可以在 src/templates/war/web.xml（如果这个文件不存在，输入 grails install-templates 产生它）设置任何通用的 JEE 配置。

一旦万事俱备，那么就输入 grails clean，确保不存在任何停留在开发模式的制品。

```
$ grails clean  
...  
[delete] Deleting directory  
/src/racetrack/web-app/plugins  
[delete] Deleting directory /Users/sdavis/.grails/1.2.0/projects/racetrack/classes  
[delete] Deleting directory /Users/sdavis/.grails/1.2.0/projects/racetrack/resources  
[delete] Deleting directory /Users/sdavis/.grails/1.2.0/projects/racetrack/test-classes
```

运行完毕之后，输入 grails war。最终，你应该得到一个真正的 JEE WAR 文件，可以部署到任何 Java 应用服务器中。

```
$ grails war  
...  
Done creating WAR  
/src/racetrack/racetrack-0.1.war
```

部署 WAR——它包含了 Groovy JAR、Grails 底层 JAR 和任何你加到 lib 目录的包——并不需要其他额外的服务器配置。输入 jar tvf racetrack-0.1.war，可以了解 WAR 文件里的所有内容。

总结

好啦，希望你能非常享受跟 Grails 相处的这段时间。现在，你已经掌握了 Grails 的基础内容，唯一需要提高的就是它的使用经验。正如你已经看到的，你可以在一小时内就创建并运行一个简单的应用——不是几天、周或月。

最后一次输入 grails stats。

```
$ grails stats

+-----+-----+
| Name      | Files | LOC |
+-----+-----+
| Controllers | 5 | 164 |
| Domain Classes | 4 | 83 |
| Tag Libraries | 2 | 19 |
| Unit Tests | 12 | 187 |
| Integration Tests | 1 | 19 |
+-----+-----+
| Totals      | 24 | 472 |
+-----+-----+
```

哇——不到 500 行代码，而且就这么点代码，你已经有了一个中等复杂程度的应用可供展示。它包括测试、安全等很多方面的内容。这种速度和效率应该会刺激你最终完成那种一直萦绕在你心头的应用。你应该明白我的意思——就是那种好几年不动弹“只要我有时间”的项目。

如果你在某个公司打工，哪种重复性的管理工作可以用 Grails 自动完成？可能是一个图书馆应用，它列出了哪些计算机图书是开发部拥有的，谁又把它们借走了？或者可能是一个跟踪组织内计算机硬件库存的系统——制造商、型号、序列号等。

要是你在家摆弄 Grails，何不写一个管理 DVD 或音乐专辑的应用？你可以写些什么好让你可以跟踪经常冒出来的待办事宜、减肥计划进展，或者甚至是购物清单小帮手。

倘若你属于某个社区组织（如 Java 用户组），你可以创建一个跟踪会议、讲座等信息的网站。假如你的孩子经常锻炼，你可以写个应用来跟踪练习、游戏和远足。

但不要误以为 Grails 简单就意味着它只适合编写“玩具”级别的应用。一些公司正在用 Grails 搭建自己面向公众的网站，这个数字还在持续增长，而且涉及的行业众多。从美国的连线

(<http://wired.com/reviews>) 到英国的 SkyTV (<http://sky.com/>) , 以及加拿大的 Taco Bell (<http://tacobell.ca/>) , 新的 Grails 网站正不断地在世界各地涌现。而且 , 就每一个 Grails 应用的公开范例来说 , 正有无数其他 Grails 应用在世界各地大多数公司的防火墙背后蓬勃发展。

至于进一步的学习 , 已经有许多现成的优秀资源可供参考了 :

我相信你会对 Grails 的联机文档的高质量印象深刻 : <http://grails.org/Documentation> 。

我正在为 IBM DeveloperWorks 撰写 (免费的) 系列文章 , 它们是 [精通 Grails \(Mastering Grails \)](#) (译注 : 中文版地址是 , <http://www.ibm.com/developerworks/cn/java/i-grails/>) 。

如果你喜欢 [精通 Grails \(Mastering Grails \)](#) , 你很可能会喜欢我为 IBM DeveloperWorks 撰写的另一系列 —— [实战 Groovy \(Practically Groovy \)](#) (译注 : 中文版地址是 , <http://www.ibm.com/developerworks/cn/java/j-pg/>) 。

我的《[Groovy Recipes: Greasing the Wheels of Java](#)》一书非常适合了解 Groovy 在 Web 开发之外的应用情况。

我在国际软件会议上说过 , 我想跟 Groovy 和 Grails 的高手谈天说地 , 用录像带记录整个过程。要是想观看 Graeme Rocher 以及其他 Groovy/Grails 社区人物的访谈 , 请访问 :

<http://thirstyhead.blip.tv/> 。

我在本书的开篇曾说过 : “ Grails 是一个注重成效的开源 Web 应用框架。” 褒心地希望你能同意这一点。享受 Grails 吧 !

作者简介

Scott Davis (电子邮件：scott@thirstyhead.com) 是 ThirstyHead.com 的创始人，该公司专注于 Groovy 和 Grails 方面的培训。

2006 年，Scott 发布了一个用 Grails 实现的公共网站，该站点属于首批采用该框架实现的站点之一。从那之后，Scott 积极投身于该技术的研究之中，并出版了书籍《Groovy Recipes: Greasing the Wheels of Java》，同时还在 IBM developerWorks 上连载系列文章（精通 Grails (Mastering Grails) 以及自 2009 年开始撰写的实战 Groovy (Practically Groovy) ）。Scott 撰写了大量关于 Groovy 和 Grails 何以是 Java 开发未来方向的文章。

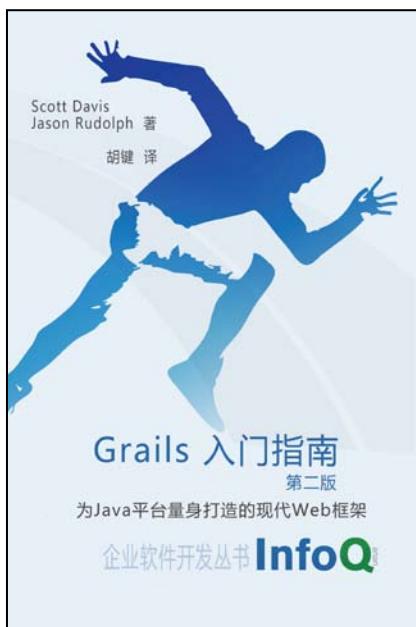
Scott 为创业公司及财富 100 强公司讲授 Groovy 和 Grails 相关的课程。他是 Groovy/Grails 经验交流会议的合伙创办人，而且定期在国际技术会议上进行巡回演讲（包括 QCon、No Fluff Just Stuff、JavaOne、OSCON 和 TheServerSide）。在 2008 年，Scott 因其演讲“Groovy, the Red Pill: How to blow the mind of a buttoned-down Java developer”当选为 JavaOne 会议的摇滚巨星。

Jason Rudolph 是 Relevance 的合伙人，该公司是领先的咨询和培训组织，致力于 Ruby、Rails、Groovy 和 Grails 方面的知识传授，以及将它们集成进企业环境之中。Jason 在开发软件解决方案方面拥有超过 10 年的经验，客户遍布国内外各种规模的公司，包括创业公司、道指 30 强和政府组织。

Jason 经常在软件会议和用户组上发表演讲，他同时还因时不时写些软件开发方面的文章而被人记起。Jason 是 Grails 早期的提交者并经常给开源社区做出贡献。最近，Jason 的身影经常出现在 Tarantula、Blue Ridge，以及其他专注于测试和改进代码质量的 Ruby 和 Rails 项目中。

Jason 拥有弗吉尼亚大学的计算机学位，目前和他的妻子（她长于搭建 Web 应用并把它们弄得光鲜亮丽）以及爱犬（它正忙活着准备它的新书----《超越松鼠指南》）生活在北卡罗莱州瑞利市。

你能通过 <http://jasonrudolph.com> 联系到 Jason。



Grails 入门指南

--- 第二版

责任编辑：霍泰稳

美术编辑：胡伟红

本迷你书主页为

<http://www.infoq.com/cn/minibooks/grails-getting-started-ii>

本书属于 InfoQ 企业软件开发丛书。

如果您打算订购 InfoQ 的图书，请联系 books@c4media.com

未经出版者预先的书面许可，不得以任何方式复制或者抄袭本书的任何部分，本书任何部分不得用于再印刷，存储于可重复使用的系统，或者以任何方式进行电子、机械、复印和录制等形式传播。

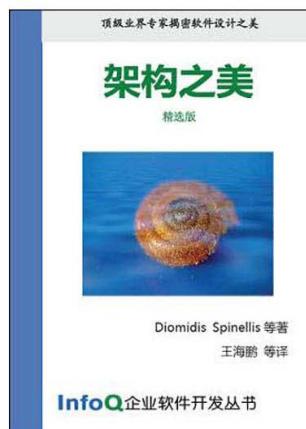
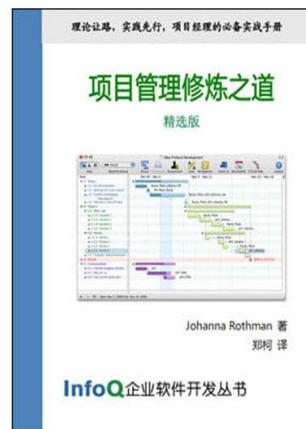
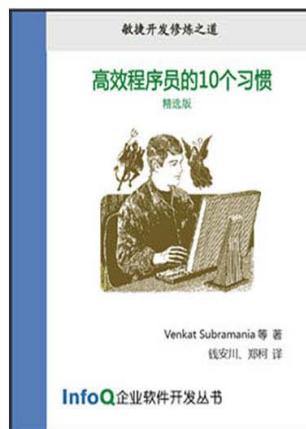
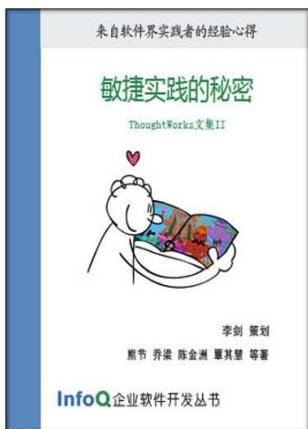
本书提到的公司产品或者使用到的商标为产品公司所有。

如果读者要了解具体的商标和注册信息，应该联系相应的公司。

欢迎共同参与 InfoQ 中文站的内容建设工作，包括原创投稿和翻译等，请联系 editors@cn.infoq.com。

InfoQ企业软件开发丛书

欢迎免费下载



商务合作: sales@cn.infoq.com

读者反馈/内容提供: editors@cn.infoq.com