

Grails 入门指南

Getting Started with Grails



【美】Jason Rudolph 著

陈俊 林仪明 彭青 吴仕橹 译

InfoQ 企业软件开发丛书

免费在线版本

(非印刷免费在线版)

InfoQ 中文站出品

InfoQ 中文站

本书由 InfoQ 中文站免费提供下载，如果您从其它渠道获取本书，请在 InfoQ 中文站上进行注册，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。

本书主页为
<http://www.infoq.com/cn/minibooks/grails>

Grails 入门指南

【美】Jason Rudolph
陈俊 林仪明 彭青 吴仕橹

著 译

© 2006-2007 C4Media Inc.

保留所有权利

C4Media 是企业软件开发社区 InfoQ.com 的出版商。

本书是 InfoQ 企业软件开发系列丛书之一。

关于这本书或者其它 InfoQ 的书籍，如果你希望了解相关信息或进行订购，请联系 books@c4media.com。

未经出版商预先的书面许可，不得以如电子、机械、影印、录音、扫描或其它任何形式及手段对本书的任何部分进行翻印或储存于可重复使用的系统中。

被公司用来唯一标识产品的名称常常被称作商标。在 C4Media Inc. 了解相关声明的所有产品实例中，对应的产品名称都会以首字母大写或者全部大写的形式出现。如果读者需要获得商标和注册更完整的相关信息，请与相应的公司联系。

欢迎共同参与 InfoQ 中文站的内容建设工作，包括原创内容投稿及翻译等，详情请垂询 china-editorial@infoq.com。

英文版责任编辑: Floyd Marinescu

英文版封面设计: Gene Steffanson

英文版排版: Laura Brown

中文版责任编辑: 赖翥翔

中文版翻译: 陈俊 林仪明 彭青 吴仕橹

中文版审校: 郭晓刚 赖翥翔

10 9 8 7 6 5 4 3 2 1

译者序

随着 Ruby on Rails 这动态脚本语言的日渐盛行，快速开发的理念逐渐深入民心。但由于新兴的 Ruby on Rails 缺乏像 Java 那样成熟稳健的虚拟机，以及对企业级服务（如分布式事务、消息传递等）的成熟支持，让不少谨慎的企业和开发者观而止步，而 Grails 的出现正好弥补了这方面的缺陷。

Grails 构建于 Groovy 之上，与 Java 无缝结合，充分利用了 Java 丰富的第三方开源库。Grails 的内核就是基于 Spring、Hibernate 和 SiteMesh 这些成熟而完善的框架组合之上的。它可被部署到任何主流的 JavaEE 应用服务器（如 WebSphere、WebLogic 等）之上，在为你的应用开发增添强大的动态语言优势的同时，又能直接访问你业务所依赖的那些企业服务。

作者以其风趣轻松的文笔贯通全书，由搭建一个原始的网络应用实例起步，通过介绍各种紧贴实际的应用需求，逐步建立起功能完备的应用，向你展现 Grails 的各种特征，让你领略“不重复开发”和“规约重于配置”原则的优势，引领你体验 Grails 开发简便快捷的乐趣，使你在这愉快的学习旅途中爱上 Grails。

本书中例子的所有源代码均可在 InfoQ 网站中下载。如果你是刚接触 Grails，该书会是一本不错的入门指南，通过本书按图索骥，你可以很快熟悉并领略到 Grails 的方方面面。如果你已熟悉 Grails，则可以将该书作为一本参考手册，在开发中进行借鉴。如果你已有 Ruby 的开发经验，则能从中体会到 Grails 不单单是 Rails 的克隆品，还有了许多 Ruby 中并不存在的特性和概念。

本书翻译工作由 4 名译者共同完成：陈俊、林仪明、彭青和吴仕橹。全书由 InfoQ 的编辑赖翥翔和郭晓刚统稿、润色和审校。译文

中疏漏和错误在所难免，如果您在阅读中发现任何疑问，或对本书有任何建议，欢迎到InfoQ中文站（<http://www.infoq.com/cn/>）交流。

译者
2007年10月

目录

1. 简介.....	1
通过例子来学习	2
RaceTrack应用	2
2. 准备上路.....	5
安装JDK.....	5
安装Grails	5
安装数据库	5
3. 你好, Grails!	7
创建你的第一个Grails程序	7
里面有什么东西呢?	8
建立你的领域	9
取得控制	13
我的数据哪去了?	17
建造更好的脚手架	22
理解URL和控制器	29
4. 提升用户体验.....	35
自定义错误消息	35
添加警告信息	39
实现确认信息	40
移除数据ID	43
格式化数据	44
5. 动态.....	53
动态查询器	53
构建自己的查询条件	58
6. 并不仅限于内部网络的应用	65
除增删查改功能外	66
实现用户认证	74
界面美化: 布局与CSS	86
7. 测试.....	91
单元测试	91

功能测试	95
8. 终点线	97
日志	97
部署	99
9. 深入应用的技巧	103
自行定义数据表	103
处理遗留数据表	105
ORM问题与解决	105
升级Grails	106
10. 总结	109
关于作者	111
资源	113

致谢

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, 并在一路上启发我深入 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的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, 一一解决残留的问题, 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。非常感谢在我写书的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, 使这本书比它原本的样子更好。

序言

我们当中热衷于程序开发的人会发现敏捷开发大大提高了我们成功的机会。但什么是敏捷呢？它的核心是，先开发相对小的一部分功能，然后从客户那里得到快速的反馈。客户会告诉我们方向是否是正确，帮助我们让开发紧贴他们的需要，同时共同确保我们创建的程序将会为他们增加商业价值。为什么我们需要敏捷呢？作为一个产业，长久以来我们一直看着其它开发方法论在“交付真正满足客户需要的系统”这个问题上跌倒。要想最终交付客户想要的系统，我们必须采取更小更快的步伐，频繁地与客户一起达成阶段性目标，以交付客户能够操作和评审的有用功能。

两个关键点影响了我们敏捷开发的能力。

首先，是每个相关人的态度。团队里每个人以及整个团队的态度，对项目的方方面面都有重大的影响。这些态度决定了我们有多乐意去与其他程序员及客户紧密合作，以获取他们的反馈，并通过这些反馈来改进我们的工作。

第二，是开发语言、框架、以及我们用来开发的工具。我们所选择的技术必须能够让我们快速创建代码并快速应对客户的反馈作出变更。

一些编程语言和框架需要开发人员在开发出有用的可执行的代码前就付出大量的精力。你可能不得不对付 XML 配置文件，实现一大堆接口，在应用程序的不同层上重复执行某些操作。在这个环境中，要想看到任何修改的成果都要先经历过度的付出。一旦完成了最终的代码，哪怕用户在恳求，只要一想到要引入变更你就忍不住推搪。你会很快下意识地抵触需求变更，让客户满意突然间没有避免变更那么重要了。

幸运的是，我们已经看到了近年来出现的一些编程语言和框架支持一种更敏捷的开发方法。它们允许你在短短几分钟内开发出可运行可演示的代码，而不是以天算或以周算，同时又保证了代码的质量。当你从客户那拿到宝贵的反馈时，这些编程语言和框架倾向于使变更更加容易，而不是妨碍变更。

那到底是怎么实现的呢？首先，这些框架遵循不重复开发（Don't Repeat Yourself, DRY）原则。因为程序的每项功能都只存在一处实现，你不需要修改多层代码来仅仅完成一项变更。相反，你在一个地方作出修改，大部分情况下，工作就已经完成了。第二，它们遵循一种“规约取代配置”的策略，所以你不必花费时间去编写那些显而易见的东西；相反，你可以直接编写真正有用的代码。第三，它们是轻量级的；你可以更改一段代码然后立即看到更改的结果，而不用在外面重编译或重新做缓慢繁重的部署工作。

那听起来很吸引人，但你可能会问：“那我们对其他语言和框架如 Java、Hibernate 和 Spring 等等的投入不是白费了？”所以 Grails 出现了。因为 Grails 用的是（运行在 JVM 上的）Groovy 语言，所以它允许你随意整合 Java 代码，因此你仍然可以有效利用当前的投资。当你研究 Grails 的时候，你会发现它遵循的原则既好又实用——它发挥了规约取代配置的力量，崇尚不重复开发原则，并且是完全轻量级的——这使它成为一个强健的敏捷框架。但别只信我说的，跟着 Jason 帮你创建好的例子，边读边写代码，亲身体验 Grails 是如何提升你的敏捷程度的。接着，找个合伙人，为你的应用选个方向，尝试去实现它吧！

Venkat Subramaniam
2006 年 11 月

言简而意赅，保持直白；说话的目的不是为了卖弄，而是为了让人能真正明白其意。

- *William Penn*

以身作则并不是影响他人的主要因素，它是“唯一”要素。

- *Albert Schweitzer*

简介

Grails 是一个开源网络应用框架，它的宗旨是实用。

曾有位大学教授向我宣扬他的观点：“最好的程序员，是那些能在早上 10 点钟前离开办公室去参加高尔夫球课程的人。为什么呢？因为他们都是专注的、高效的、高产的。”事实上他也试着让这观点深入到他的学生心里。当然，主张在咖啡变冷之前就完成一天的工作太夸张了，但能在喝第二杯咖啡之前完成一天的工作量是不是很令人向往的事情呢？

Ruby on Rails 包含了一种强大的编程语言，同时又是一个坚持己见，提倡用通情达理的默认值来替代复杂的配置的框架，它开创了两者的创造性结合。开发人员称赞 Rails 框架的革命性，因为它大大提高了生产率（的确如此）。但很多机构仍未准备好脱离 Java 这个安全地带。所以，如果我们能够达到类似的生产率，而又同时拥有一个更以 Java 为核心的解决方案，何乐而不为呢？Grails 使这个想法成为了事实。因为 Grails 应用程序是用 Groovy（一种动态类型的脚本语言，即将成为 Java 标准¹）编写的，所以 Grails 应用程序能跟原有 Java 代码的友好结合，令谨慎的企业更加放心。

Grails 信奉“规约重于配置（Convention over Configuration）”，这种技术同时改善了注意力和生产效率，“规约（Convention）”支配了每一个组件在程序中的归属、命名，以及如何与其它组件相互作用。简而言之，开发人员能确切知道某一组件应放在何处，该如何命名，即便是新组员也会清楚该组件的确切所在。这都是规约使然。

Groovy 为创建 Grails 应用程序提供了高效的表达方式和完全面

向对象的语言。Groovy 允许开发人员以简单易懂的风格简洁自然地传达想法。同时，Java 开发人员会赏识它跟 Java 相似的语法，以及跟 Java 的无缝结合，因为他们能够在 Groovy 里调用任何 Java 类，反之亦然。

Grails 的基础，是建立在已被公认的技术上的。如：Hibernate，一个事实上的软件产业的标准，为 Grails 提供“对象—关系映射”（ORM）的基础；Spring 框架为 Grails 提供“模型—视图—控制器”（MVC）的核心，和强大的“依赖注入”功能；SiteMesh 给 Grails 带来了灵活高效的界面布局管理。当然，别忘了 Java，由于 Groovy 跟 Java 的完美结合，使 Grails 应用程序不仅可以直接使用众多的 Java 类库，而且可以使用 J2EE 应用程序服务器提供的企业级服务（分布式事务、通讯等等），这对很多应用程序来说是必不可少的。有了这些牢固的立足点，Grails 将会为 Java 以及企业级平台上的快速网络应用开发作出证明。

通过例子来学习

本书以示例来介绍 Grails。我们将会看到如何快速地创建 Grails 应用程序，以及如何定制应用来适合不同的需求。我们会探究我们碰到的每一个概念的本质，对于那些想要深入研究某些主题的读者，我们会在最后给出相应的注释来提供补充信息。

为了更好地跟上进度，你需要有面向对象编程和开发MVC网络应用的基础知识，并且如果你熟悉Java的话那就更容易上手了（虽然你也可以在完全不懂Java的情况下学会Grails）。我们的例子主要使用Groovy。虽然本书并不是以讲解Groovy为目的，但有一定编程基础的人都应该可以看懂例子跟上本书进度。如果你觉得有必要探究某个Groovy用法的细节，可以到Groovy网站查阅更多的信息²。

RaceTrack 应用

在学习本书的过程中，我们会通过创建一个叫 *RaceTrack* 的网络应用来探究 Grails 开发的各个方面。有一个在比较落后的地区的赛跑俱乐部，他们还在用纸面的流程来跟踪俱乐部参与的比赛以及每场比赛注册的会员。他们现在已准备好跨入到数字时代了。作为起点，他们想要一个允许俱乐部职员管理比赛和注册信息的程序。

他们对我们说不需要太绚丽的界面——不奇怪，他们现在还在用纸张纪录。所以，他们会很高兴有这么一个可以管理比赛数据的内部程序。我们会从一个企业内部网络程序开始，这个程序将满足他们的需要。不过我们听到了一些暗示，如果一切进展顺利的话，他们想让赛跑员自己来注册赛跑，所以这就需要我们把程序也暴露给外部用户使用。

开发 *RaceTrack* 程序给了我们一个全面的实际演练 Grails 的机会。我们会创建一个网络用户界面，管理数据库表间的关系，应用验证逻辑和开发自定义查询。随着我们继续扩展程序功能，我们将会探究自定义标签库、Java 整合、安全性、页面布局以及动态方法的威力。在我们结束前，我们还将探究一些背后的支持技术，包括单元测试、日志和部署。

大体上，创建程序所需的信息以及代码片段都包含在本书的文本里。在我们结束时会碰到个例外，不过我们到时会再介绍。别担心，所有例子的源代码你都可以下载得到³。代码包中包含了每一章结束时程序源代码的完整快照。

2

InfoQ 中文站.NET 社区

.NET 和微软的其它企业软件开发解决方案

<http://www.infoq.com/cn/dotnet/>

准备上路

安装 JDK

Grails 只要求 JDK 1.4，但在以后几个章节里面的一些例子，我们会用到 JDK 5 的一些新特性。所以，你需要用到 JDK 5 来运行那些例子（我们会在介绍时确切地指出哪些例子要依靠 JDK 5）。

所以，当我们谨记Grails能很好的支持JDK 1.4 的同时，你不妨去下载并安装JDK 5 (<http://java.sun.com/javase/downloads/>)，以便顺利学习全部例子⁴。然后，设置你的JAVA_HOME环境变量，把它指向JDK 5 的安装路径。

安装 Grails

接着，从<http://grails.org/Download>⁵ 下载最新的Grails稳定版本（本书用Grails 0.3.1——写书的时候的稳定版本）。然后，按照快速安装步骤 (<http://grails.org/Installation>) 安装好Grails，并配置必要的环境变量⁶。

安装数据库

Grails 周到地附带了一个内嵌的 HSQLDB 数据库，所以每个 Grails 程序都默认有个内存数据库可用。这个可选的数据库对于快速演示很有用，也方便你跟客户并排坐在一起快速试验各种功能。当然长期运行的话，我们还是需要转移到更传统的磁盘数据库（这样即使偶尔重启程序也不会丢失数据）。

Grails支持大部分的数据库（包括Oracle、DB2、PostgreSQL等等），本书中我们会用 MySQL 5.0 来做例子。你可以到<http://dev.mysql.com/downloads/mysql/5.0.html> 下载 MySQL

Community版本，按照说明安装好，我们就准备就绪了⁷。

InfoQ 中文站 Ruby 社区

面向 Web 和企业开发的 Ruby，主要关注

Ruby on Rails

<http://www.infoq.com/cn/ruby/>

3

你好，Grails！

创建你的第一个 Grails 程序

现在已经安装好了 Grails，我们还要为我们的 Grails 程序创建一个目录。你可以任意命名并且放在你喜欢的任何位置。

```
jason> mkdir grails_apps  
jason> cd grails_apps  
grails_apps>
```

接着，在我们刚刚创建的目录里生成我们的项目结构。输入 `grails create-app`，当提示输入程序名称时，输入 `racetrack`。

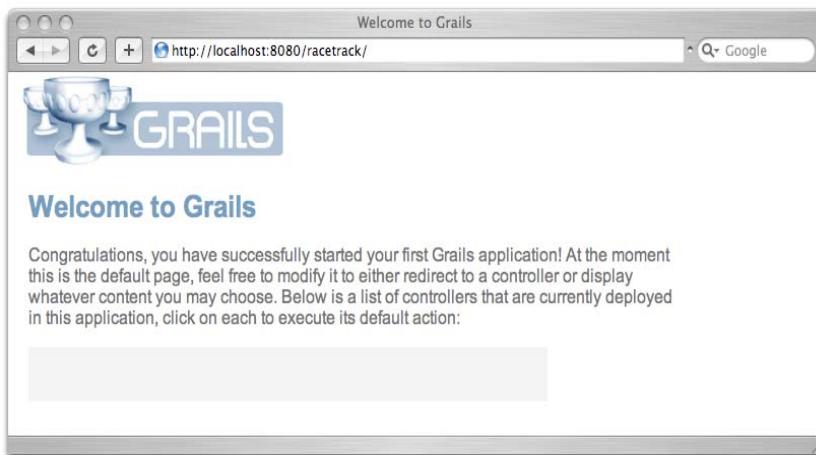
```
grails_apps> grails create-app  
...  
create-app:  
[input] Enter application name:  
racetrack  
...  
BUILD SUCCESSFUL  
Total time: 4 seconds
```

启动我们的程序来检验一下环境是否配置无误。进入到我们新创建的程序目录里，然后输入 `grails run-app` 来启动程序。

```
grails_apps> cd racetrack  
racetrack> grails run-app  
...  
run-app:watch-context:
```

程序正等待着我们的请求呢，打开你的浏览器访问

<http://localhost:8080/racetrack/>，你会看到这个友好的信息欢迎你来到Grails的世界。



里面有什么东西呢？

一切顺利，但在我们更进一步之前，我们先靠近一些看看我们的新程序里都有些什么东西。

Grails对快速程序开发的支持主要来自于它对“规约重于配置”这个理念的贯彻（这是Ruby on Rails所宣扬的“固执软件（Opinionated Software）”模式的一个主要部分）。Grails项目结构（下图 3-1）仰仗“规约”，为程序的各个部分建立了合理的组织结构⁸。

racetrack	
+ grails-app	
+ conf	配置设置，包括开发、测试和产品的数据源
+ controllers	控制器
+ domain	领域类
+ i18n	国际化资源绑定信息
+ services	Service 类（在 Grails 里等同于本地 Session Bean）
+ taglib	标签库
+ views	视图模板（还有一个所有控制器共用的而准备的子目录）

+ layouts	布局模板 (所有控制器都可以使用)
+ grails-tests	单元测试
+ hibernate	可选的 Hibernate 配置文件
+ lib	应用程序中需要的自定义库
+ spring	可选的 Spring 配置文件
+ src	
+ groovy	Groovy 源文件 (除了控制器、领域类、或者是 service 类之外的文件)
+ java	Java 源文件
+ web-app	
+ css	样式表文件
+ images	图像文件
+ js	JavaScript 文件和第三方库 (例如 Prototype、Yahoo 等)
+ WEB-INF	与部署相关的配置文件
+ index.jsp	应用程序的首页

图 3-1: 程序目录结构

建立你的领域

Grails 把领域类 (Domain Class) 看作是程序的中心, 并且是程序最重要的组件。我们将通过领域类来驱动程序中的一切 (如果你使用过 Ruby on Rails, 你会发现这跟 Rails 的做法不同, Rails 的做法是从底层的数据库定义中引申出领域模型【Domain Model】)。

为了更好地理解我们需要获取的数据, 我们首先检查一下赛跑俱乐部现有的运作过程。一开始, 他们提供了目前用来跟踪赛跑信息的表格。

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	

从这个表格可以看出，我们基本上需要对两种数据进行处理：赛跑与注册（这两个实体间是一对多关系）。这个信息将是我们领域模型的基本，而表格里的各种数据元素将会是我们领域类的属性。

那么，让我们来创建代表这些元素的领域类。回到命令行窗口，输入 `grails create-domain-class`，在提示输入领域类名称的时候，输入 `Race`（如果你早前启动的程序还在运行，按 `Control-C` 来停止程序，回到命令提示）。

```
racetrack> grails create-domain-class
...
create-domain-class:
  [input] Enter domain class name:
Race
...
  [echo] Domain class created: grails-
app/domain/Race.groovy
...
  [echo] Created test suite: grails-
tests/RaceTests.groovy

BUILD SUCCESSFUL
Total time: 5 seconds
```

我们会发现 Grails 不仅为我们创建了领域类，而且还创建了对应的单元测试类，之后我们会用到它。

接着，重复这个步骤来创建 `Registration` 领域类。

现在，让我们看看自动创建的领域类。用你喜欢的编辑器打开 `racetrack/grails-app/domain/Race.groovy`。

```
class Race {  
}
```

诚然，现在很难说这些给我们留下深刻的印象，但稍忍几分钟吧，一切都会逐步变好的。

从我们早前跟客户的讨论以及对文件表格的分析中，我们知道每个赛跑都有哪些属性，那么让我们把那些属性加到领域类里面吧。

```

class Race {
    String name
    Date startTime
    String city
    String state
    Float distance
    Float cost
    Integer maxRunners = 100000

    statichasMany = [registrations:Registration]
}

```

好了, 上面所做的都相当的直观。一场赛跑包括名字、开始日期/时间等等。每场赛跑缺省最多只能有 100,000 个赛跑运动员。这再简单不过了, 并且所有的数据类型看起来都相当眼熟, 但这“hasMany”是什么意思呢?

从业务上看, 赛跑跟注册是一对多的关系, 我们的 `Race` 领域类包括 `hasMany` 这个属性来支持这种关系。属性 `hasMany` 告诉 Grails 一个 `Race` 跟某个类有着一对多的关系。为了指明这个关联类, 我们把一个由“属性名-类名”对组成的 `Map` 赋值给 `hasMany` 属性。这样一来, 我们就声明了 `registrations` 属性是 `Registration` 对象的一个集合。

我们并没有明确地定义 `registrations` 这个属性 (这正是 Grails 程序简洁而强表现力的本性的一个体现)。在 `hasMany` 这个 `Map` 里面, 我们告诉 Grails 一个 `Race` 对象的 `registrations` 属性应该存储 `Registration` 对象的集合。既然 Grails 知道了我们希望 `Race` 对象拥有这个属性, 那么我们还有必要再次单独为它声明一个属性吗? Grails 信奉 DRY (不重复开发) 原则⁹, 这原则让我们尽量从不必要的冗余中解脱出来。

既然我们是在一个 `Map` 中声明这些信息, 要是限制我们只能定义一组一对多关系, 就太没意义了。例如, 如果我们想要记录某场赛跑的赞助商, 我们只需往这个 `Map` 中加入另外一个集合元素。

```

statichasMany = [ registrations : Registration,
                  sponsors : Sponsor ]

```

当然, 你不可能只碰到一对多这种关系类型。Grails 也支持一

对一、多对一、多对多的关系¹⁰。

现在看看关系另一边的 `Registration`，用你的编辑器打开 `racetrack/grails-app/domain/Registration.groovy`，让我们来定义它的属性。

```
class Registration {
    Race race
    String name
    Date dateOfBirth
    String gender = 'F'
    String postalAddress
    String emailAddress
    Date createdAt = new Date()
    static belongsTo = Race
    static optionals = ["postalAddress"]
}
```

大部分的属性看起来也都相当平常：名字、出生日期等等。只有几个地方是需要我们留意的。

用户告诉我们所有的赛跑属性都是必要的，同样，大多数的注册属性也都是必要的。然而，用户觉得互联网应该是趋势，所以他们决定有一个电邮地址就足够了，而邮寄地址则作为一个可选属性。你会发现我们没有在代码里面声明哪些属性是必须的，那是因为 Grails 默认地假定所有属性都是必须的。如果某一属性不是必须的，那么我们需要声明它是可选的。为了做到这点，我们要定义 `optionals` 属性，顾名思义这是领域类中可选属性的一个列表。如果我们想要定义另一个可选属性，例如“性别”，把它加入列表里面就可以了。

```
static optionals = ["postalAddress", "gender"]
```

在 `Race` 这个领域类里面，我们声明了一个 `Race` 对象拥有零个或者多个 `Registration` 对象。你会发现我们并没有明确地说一个 `Registration` 对象是否可以单独存在（即在没有一个相应的 `Race` 对象的情况下）。我们当然知道不能在不存在的比赛上注册，所以我们如何告诉 Grails 注册是依附在赛跑上的呢？既然我们知道所有的属性默认上都是必须的，那么除非 `race` 属性有一个非空值，否则无法保存一个 `Registration` 对象。所以，事实上我

们的确已经声明了 `Registration` 对象不可能单独存在。

`belongsTo` 属性标志了赛跑跟注册之间的一对多关系的所有方。因为我们声明了一个注册属于一个赛跑（由注册的 `race` 属性确定），Grails 将会保证删除一个注册不会删除对应的赛跑，但删除一个赛跑会删除所有相关的注册。

熟悉的世界

现在看过了领域模型，我们可以看出 Grails 是多么地信奉面向对象编程。我们的建模手法将与 Java、Ruby 或者其它面向对象语言类似。事实上，随着我们的进一步学习，你会发现你所喜欢的 OO 和 MVC 概念将会继续在 Grails 里面为你服务。

听上去不错，实际又如何？

在我们开始讲述控制器之前，注意一下我们在这些类中没提到过任何“对象一关系映射”（Object Relational Mapping, ORM）或持久层之类的东西。这些领域类似乎没有扩展任何提供持久化功能的类，它们也没有实现任何标志这些类需要持久化的接口，它们也没有关联任何提供持久化服务的类。而我们也没编辑任何配置文件。是什么告诉 Grails 这些类需要持久化支持呢？

是“规约”让我们省去了这些步骤。通过规约，Grails 自动地把放在 `racetrack/grails-app/domain` 里面的类识别为领域类，并且它知道领域类需要持久化。简单地遵循这个规约，Grails 帮我们做了所有的持久化工作，最终呢，它使我们有更多的时间去解决更高难度且更有意思的问题。

在背后，“Grails 对象一关系映射”（GORM）担负起必要的持久化功能。当在 Grails 程序里用到持久化逻辑时，GORM 就挑起了所有重担（在底层，GORM 现时依赖于 Hibernate，但在即将到来的版本中还会支持 JPA）。

取得控制

万事俱备，只欠东风。到目前为止我们所做的都是些没法偷懒

的事情（不管你的框架多么的智能，你始终需要说明你要处理哪些数据）。然而，下一步我们要为用户界面建立一个良好而有效的出发点，那就不再是体力活了。那么，是时候加上控制器了。

确保你是在项目的根目录下（在我们的例子中是 racetrack）。接着，输入 `grails create-controller`，并告诉 Grails 为 Race 这个领域类创建一个控制器。

```
racetrack> grails create-controller
...
create-controller:
  [input] Enter controller name:
Race
...
  [echo] Created controller:
grails-app/controllers/RaceController.groovy
...
BUILD SUCCESSFUL
Total time: 3 seconds
```

现在，让我们来打开新建的控制器，并看看要从哪里开始。找到 `racetrack/grailsapp/controllers/RaceController.groovy`，打开看一下。

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

嗯。我们最终目标就是通过用户界面来管理赛跑数据。所以，我们需要往控制器里面加入列出、添加、修改和删除赛跑记录的功能。让我们加入必要的代码来满足这些需求。那需要多少行代码呢？

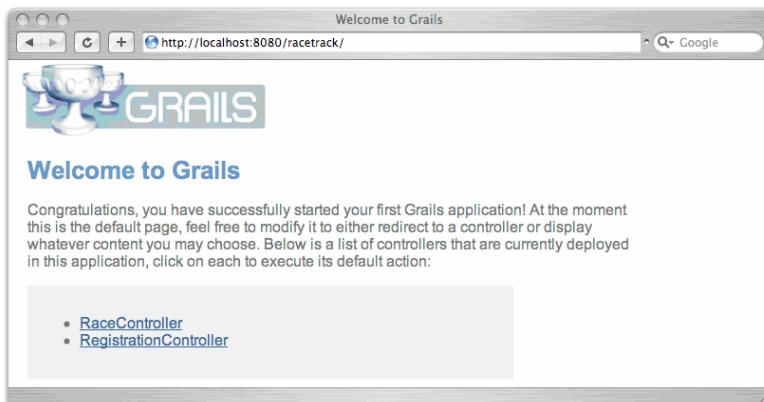
```
class RaceController {
  def scaffold = Race
}
```

那是什么啊？！事实上，那就是我们想要得 CRUD（添加-读取-修改-删除）功能。当 Grails 在控制器中碰到 `scaffold` 属性时，它会动态地为特定的领域类生成控制逻辑以及必要的视图，而所有的这些都来自于那一行代码！

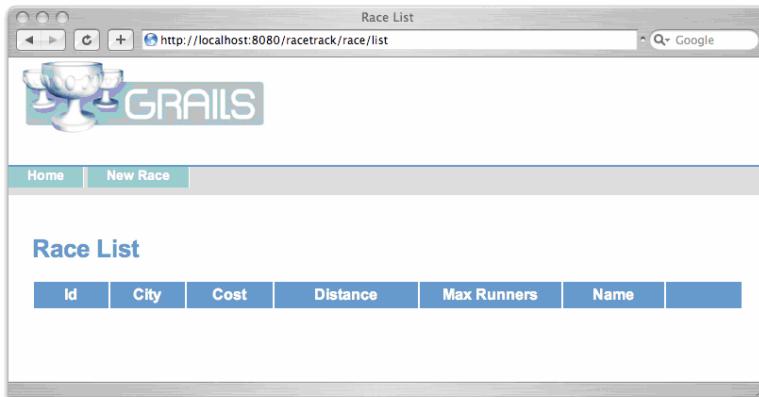
别光听我说, 让我们来对注册数据重复应用这个步骤, 接着我们就准备好运行程序看看效果了。跟着前面的步骤来创建 `RegistrationController` 类。

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

现在, 是时候看一下我们的成果了。在你的程序目录下 (`racetrack`), 输入 `grails run-app`。当你看到程序成功启动的时候, 打开浏览器并访问 <http://localhost:8080/racetrack>。



好了。我们已有两个控制器。就像欢迎页中所说, 我们最终将用定制的欢迎页来取代这个页面。不过现在呢, 让我们先进入 `RaceController`。



我们所期待的大部分的字段都在这里了，但由于没有数据，所以还没有什么意思。点击新建比赛来创建一些数据，然后花些时间探索一下目前的这个程序。当你加入一些新数据之后，回过头去试试编辑现有的赛跑，删除赛跑，并且试一下管理注册部分的类似特性。在操作时，记下你想要改进的地方和需要改进的地方。同时，看看那些已经满足了你的需要的东西，它们都是不费吹灰之力就得来的！

你可能已经发现有几个地方显然还需要改进, 但大部分的程序只需稍微化化妆就可以了。

下面是我觉得必须具备的功能, 至于那些锦上添花的功能当然迟点我们也会谈到。

- 验证——我们从没告诉 Grails 一场比赛的距离必须大于零米, 比赛的名字必须不超过五十个字符, 或者“松鼠”不是一个有效的性别。一句话, 没有验证。但我们很快就能加上去。
- 字段/数据列的排序——当我们查看比赛数据的时候, 为什么城市排在最前面而州则在最后面, 中间又夹杂着其它字段? 因为我们未曾告诉 Grails 该怎么排。其实, 解决这个问题的方法很简单, 我们很快就会看到。
- 记录标识符——当我们编辑单独一条注册信息时, 看一下在 Race 字段中显示的是什么。你能记住哪个比赛是“Race: 1”, 哪个是“Race: 2”吗? 编辑一个比赛并查看它的注册信息时也存在同样的问题。我们应该让这个信息更清晰一点。

就这些了。如果能够解决这几个问题, 我们就会有一个功能完整的网络应用程序来管理我们的赛跑跟注册数据。在此之后, 剩下的就只是锦上添花了。(当然“添花”对我们继续完成这个应用也是大有裨益的。)

我的数据哪去了?

是不是有件事情还没做? 我们从未告诉 Grails 我们安装的

MySQL 数据库的任何信息。那个内存数据库有时真的很好用，特别是初次实验的时候。但一旦我们重启程序，就会丢失我们刚刚录入的所有数据。这对目前来讲没什么问题，但当我们继续深入的时候，我们绝对不会想要浪费时间在重新输入测试数据上。所以，既然我们挺满意最初的领域模型，就让 Grails 用 MySQL 数据库来支持我们的程序吧。

小提示：如果我们只是单单想避免再次录入测试数据，我们可以考虑用 Grails 的 Bootstrap 类来帮忙，这样内存数据库机还可以再撑一阵子。Bootstrap 类给执行各种初始化工作提供了一个方便的机制，包括填充数据库表在内。然而，Bootstrap 类每次都会用同样的数据来重新初始化程序，这样我们重启程序时将会丢失所有修改过的数据。所以我们最好还是连上一个传统数据库。Bootstrap 类我们放到第 6 章再讲。

如果你查看 `racetrack/grails-app/conf/`，会发现我们的程序包含了每种常用环境（开发、测试以及产品）的配置文件。我们会为每个环境分别创建独立的数据库。

- 开发环境——我们将定义一个叫 `racetrack_dev` 的数据库来支持开发过程，而相应的配置会放在 `racetrack/grailsapp/conf/DevelopmentDataSource.groovy`。
- 调试环境——我们的单元测试和功能测试将依赖于 `racetrack_test` 数据库，我们会在 `racetrack/grailsapp/conf/TestDataSource.groovy` 里面引用它。
- 产品环境——最后，我们会创建 `racetrack_prod` 数据库。在 `racetrack/grailsapp/conf/ProductionDataSource.groovy` 的帮助下，我们将用这个环境来让全世界享用我们的程序并一夜间成为亿万富翁！

让我们开始吧……

回到命令行窗口，按照下面的步骤创建必要的数据库并赋以适当的访问权限（我选择了用我的个人 ID 来访问测试和开发环境，

用另外一个高机密的 ID 和密码访问产品环境。你自己觉得怎样合适就怎样做。)

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

现在, 我们已经准备好了把这些数据库告知 Grails。用你的编辑器打开 `DevelopmentDataSource.groovy`, 按照下面的配置修改它。你可以在 `racetrack/grails-app/conf/` 里找到这个文件。(注意: 你需要把用户名和密码替换成你自己的 MySQL 帐户的用户名和密码。)

```
class DevelopmentDataSource {
    boolean pooling = true
    String dbCreate = "update"
    String url = "jdbc:mysql://localhost/racetrack_dev"
    String driverClassName = "com.mysql.jdbc.Driver"
    String username = "jason"
    String password = ""
}
```

注意, 我们把 `dbCreate` 属性的值改成了“`update`”。当数据库定义和领域类不匹配时, 这个值指示 Grails 在运行期改变你的数据库定义, 让它可以跟你的领域类同步。因为我们现在还没有任何数据库表, Grails 会在下次重启程序时为我们的领域类创建相应的数据库表。让 Grails 帮我们处理数据库定义是快速程序开发的又

一次体现，这可以提高我们的生产效率。

当然，有些情况下我们要面对已存在的数据库，或者我们只是想自己来管理数据库。Grails 也支持这种方式，你只要很简单地删除 `dbCreate` 这个属性，Grails 就不会去动数据库定义了。

当我们的开发进行到相应的阶段时，我们会再回过头来配置测试与生产平台的数据源。

我们差不多已经能使程序跟数据库连接起来了。我们已经告诉 Grails 我们要用来连接数据库的驱动的类名。现在，我们只需要提供相应的驱动就行。从<http://www.mysql.com/products/connector/j/> 下载 MySQL 的 Java 驱动。我建议用最新的稳定版本，本书写作时的稳定版是 5.0.4¹¹。

打开下载的 `zip` 文件并解压出 `mysql-connector-java-5.0.4-bin.jar` 文件，放到你的 Grails 应用的 `lib` 目录里——我们例子中是 `racetrack/lib`。（请注意，JAR 包的确切名字随你下载的驱动版本而不同）。

现在，可以启动程序了，让我们看看修改后更持久稳固的程序的运行情况（如果你的程序从先前一直运行到现在，输入 `Control-C` 来停止程序并回到命令行提示）。

```
racetrack> grails run-app
...
run-app:watch-context:
```

你可以尝试再体验一下程序，但界面上应该没有变化。现在我们感兴趣的是后台部分。让我们去看看开发环境数据库，我们会看到对应着每个领域类都有一张数据库表。

```
racetrack> mysql
Welcome to the MySQL monitor.  Commands end with ; or
\g.
...
mysql> use racetrack_dev
...
Database changed
```

```

mysql> show tables;
+-----+
| Tables_in_racetrack_dev |
+-----+
| race
| registration
+-----+
2 rows in set (0.00 sec)

mysql> describe race;
+-----+-----+-----+-----+-----+-----+
| Field | Type | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| id | bigint(20) | NO | PRI | NULL | auto_increment |
| version | bigint(20) | NO | | | |
| distance | float | NO | | | |
| max_runners | int(11) | NO | | | |
| start_date_time | datetime | NO | | | |
| state | varchar(255) | NO | | | |
| cost | float | NO | | | |
| name | varchar(255) | NO | | | |
| city | varchar(255) | NO | | | |
+-----+-----+-----+-----+-----+-----+
9 rows in set (0.00 sec)

mysql> describe registration;
+-----+-----+-----+-----+-----+-----+
| Field | Type | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| id | bigint(20) | NO | PRI | NULL | auto_increment |
| version | bigint(20) | NO | | | |
| gender | varchar(255) | NO | | | |
| date_of_birth | datetime | NO | | | |
| postal_address | varchar(255) | YES | | NULL |
| email_address | varchar(255) | NO | | | |
| created_at | datetime | NO | | | |
| race_id | bigint(20) | YES | MUL | NULL | |
| name | varchar(255) | NO | | | |
+-----+-----+-----+-----+-----+-----+
9 rows in set (0.00 sec)

mysql>

```

我们来讨论一下 Grails 为我们生成的数据库定义。这又一次体现了规约在 Grails 中扮演的重要角色。注意：

- 数据库表名跟领域类名完全匹配。
- 字段名跟类的属性名匹配。
- 对于多个单词组成的属性名字，字段名中用下划线来分开单词（例如，属性 `maxRunners` 跟字段 `max_runners` 对应）。

- `id` 字段是用来做每张表的主键，它被定义为自增 (`auto_increment`) 字段，从而允许 Grails 去依赖数据库为新纪录自动产生唯一的 ID。
- 对于一对多关系，子表中将会包含一个字段用来保存父表的 ID。该字段是以父领域类名命名的（在这个例子中是 `Race`），并且根据父表声明了一个外键约束。在这个例子里，结果是产生了 `race_id` 字段。

你也许会想，为什么这里会有 `id` 和 `version` 字段呢？我们在领域类里没看到相对应的属性啊。因为每个 Grails 领域类都需要这些属性，所以，在编译期 GORM 把它们注入到类里面。我们不用显式地声明它们（如果你想知道，其实这个 `version` 属性是 Hibernate 用来支持事务管理和乐观锁的。它的存在是很重要的，但除此之外，我们可以完全忽略这个属性）。

作为题外话，如果一个属性永远不会超过 50 个字符，却为它定义一个 `VARCHAR(255)` 的字段，又或者为一个只需要日期而不需要时间的属性定义一个 `DATETIME` 字段，许多数据库管理员都会有所疑虑。所以，有些人会不可避免地想要自己创建并管理数据库表（而不想让 Grails 来做）。如果你打算这样做，只要确保你的数据库定义仍然符合上面所说的规约，那么 Grails 会自动地知道怎样把你的领域类跟数据库对应上。如果你不得不背离这些规约，也不是不行，但你需要提供一些配置信息来告诉 Grails 怎样去做正确的映射。

这样我们就大体了解了 Grails 中用到的主要的数据库“规约”。我们将会在第九章更详细地讨论这些主题。与此同时，既然我们已经有一个可以持久保存数据的数据库，那么我们回到之前提过需要改进的程序部分吧。

建造更好的脚手架

首先，我们已经提过我们需要验证功能并把字段的顺序调整得更合适一些。Grails 可以用一个步骤就同时满足我们的这两个需要。

声明约束属性

Grails的约束属性（constraints）声明了我们对领域类属性的期望值。我们可以用约束属性来声明一个特定属性不能超过某个长度；限制一个列表的元素数目；要求一个保存邮件地址的属性必须遵循有效的E-mail地址语法结构等等¹²。我们用Groovy闭包¹³来在领域类里定义这些约束。闭包所用的Builder语法¹⁴简单而清晰。

我们从 `Race` 领域类开始吧，在 `Race.groovy` 里添加如下所示的闭包。

```
class Race {
    //...
    static constraints = {
        name(maxLength:50,blank:false)
        startTime(min:new Date())
        city(maxLength:30,blank:false)
        state(inList:['GA', 'NC', 'SC',
        'VA'],blank:false)
        distance(min:3.1f,max:100f)
        cost(min:0f,max:999.99f)
    }
}
```

这个结构看起来就是这么的简洁直观，简直不需要多作解释。例如，我们可以直观地看到 `name` 属性最长为 50 个字符并且不为空。

也许你会想：“我们不是说过`name`属性是必须的吗？”是的。当将一个领域对象写进数据库时，如果有必填属性是为空的，Grails将抛出一个异常。不过我们倾向于用一个更优雅的方法来处理类似这样的问题，而且我们也想在进入持久层之前就确保对象是有效的。约束属性满足了这个需求。当调用领域对象的`validate`或`save`方法时，Grails会用定义好的约束属性来检验对象并报告所有的问题¹⁵。如果对象满足不了约束属性的条件，Grails就不会把它写到数据库。

让我们依次解释上面用到的其它约束属性的意义。

- `race` 不能从过去的日期开始。
- `city` 属性最长不能超过 30 个字符，并且不为空。

- `state` 必须是列表里四个州之一（因为是个区域性的俱乐部），且不为空。
- 距离必须在 3.1 英里（5 千米）和 100 英里（160.9 千米）之间。
- 费用必须是在\$0 和\$999.99 之间。

等一等，`startDateTime` 属性的约束条件好像不是很正确。我们的业务规则表明（也符合常识），一个新比赛不可能从过去日期开始，但照它现在看来，它的最小日期/时间会在这个类初始化的时候赋值。这意味着，在程序运行了几天之后，只要开始时间/日期是在我们启动程序时间之后，那用户就真的能够创建从过去开始的比赛了。这显然不是我们想要的功能。我们需要的是一个可以比较输入值跟现在日期/时间的约束。很幸运，Grails 允许我们用自定义检验器来满足我们程序的特殊需要¹⁶。那么，让我们用下面的声明来取代之前的`startDateTime` 约束。

```
startDateTime validator: {return (it > new Date())})
```

自定义检验器以闭包的形式定义。这个特殊的检验器比较`startDateTime`（由 `it` 变量代表）跟当前日期/时间的值，并返回一个布尔值。如果输入值是有效的，检验器（显然）不会返回任何错误。另一方面，如果闭包返回 `false`，约束就会产生一个检验错误并阻止这个对象写进数据库。

调整用户界面上的字段顺序

虽然不是很直接，但其实我们也已经完成了对用户界面上字段顺序的调整。字段会按照在约束属性闭包里定义的顺序出现。如果你有某个属性不需要约束属性，但仍然想指定它在用户界面中出现的位置，你仍可把这个属性包含在约束属性闭包里。例如，如果我们想把 `maxRunners` 属性作为第一个字段，我们可以以空规则集的形式把它放在闭包的最前面。

```
static constraints = {
  maxRunners()
  //...
}
```

既然我们清楚了约束属性是如何工作的，那么我们快点为

Registration 领域类加上约束属性吧。如下所示往 `Registration.groovy` 加入闭包。

```
class Registration {
    //...
    static constraints = {
        name(maxLength:50,blank:false)
        dateOfBirth(nullable:false)
        gender(inList:["M", "F"])
        postalAddress(maxLength:255)
        emailAddress(maxLength:50,email:true)
        race(nullable:false)
    }
}
```

下次当我们重启程序后, 就会看到验证和字段排序的问题已经解决了。

有意义的记录标识符

当查看单个比赛的细节时, 前面提到过我们希望用一个比“`Registration: 1`”和“`Registration: 4`”更有意义的方式来表示注册信息, 比赛信息也一样 (要是所有的问题都像这个一样简单, 那么我们每天早上 9:30 前就可以在高尔夫球场打球了——早了整整半个小时[译注: 见第 1 章开头])。解决方法极其简单, 只要为每个领域类重写默认的 `toString` 方法就行了。我们现在给 `Race.groovy` 加上这个方法。

```
class Race {
    //...
    String toString() {"${this.name} : ${this.city},\n" +
        " ${this.state}" }
}
```

如果你是Groovy新手, 你很快就会喜欢上Groovy的GString¹⁷。GString允许你在字符串里嵌入一个表达式并在运行时求值。这的确不是什么革命性的做法, 但比起Java字符串串联或者`StringBuffer`, 它是一个很受欢迎的选择。上面的方法会返回一个类似“`Turkey Trot : Duck, NC.`”这样的值, 这毫无疑问比“`Race: 1`”要清晰多了。

够简单吧。让我们同样收拾一下 `Registration.groovy` 里

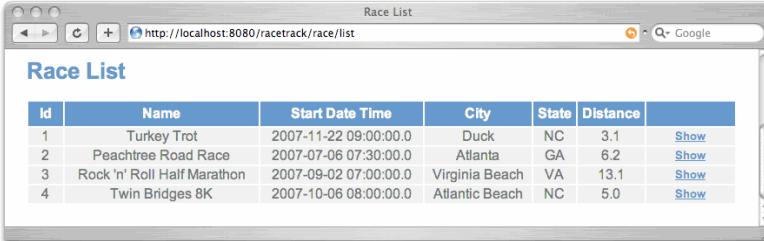
面的东西。要识别出注册的人，用名字和 E-mail 地址应该就够了。

```
class Registration {
    ...
    String
    toString(){"${this.name}:${this.emailAddress}"}
}
```

结果出来了

准备好好看看我们的劳动成果了吗？在命令行里输入 `grails run-app`。等到看到程序启动成功，然后打开浏览器访问 <http://localhost:8080/racetrack>。

我们首先看一下赛跑列表，观察一下字段排序的巨大改进。

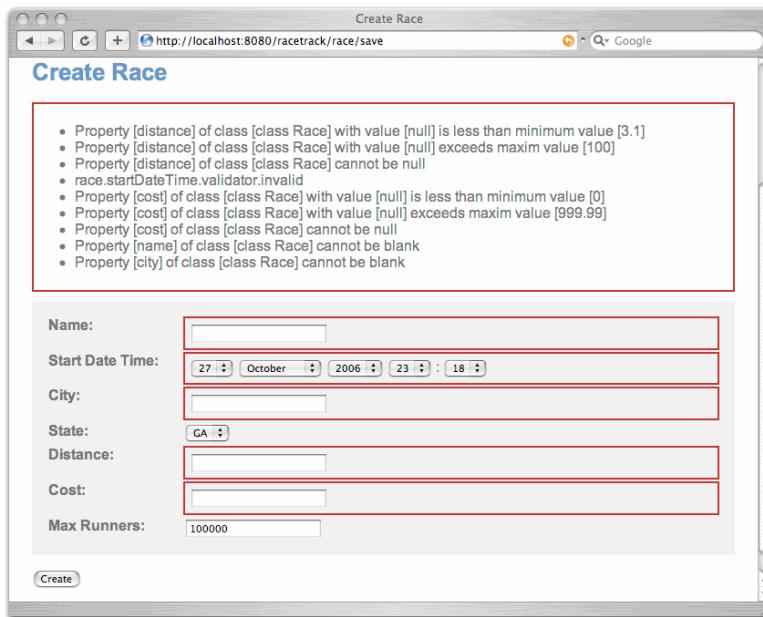


Race List						
Id	Name	Start Date Time	City	State	Distance	
1	Turkey Trot	2007-11-22 09:00:00.0	Duck	NC	3.1	Show
2	Peachtree Road Race	2007-07-06 07:30:00.0	Atlanta	GA	6.2	Show
3	Rock 'n' Roll Half Marathon	2007-09-02 07:00:00.0	Virginia Beach	VA	13.1	Show
4	Twin Bridges 8K	2007-10-06 08:00:00.0	Atlantic Beach	NC	5.0	Show

你有没有注意到列表中没有显示每场赛跑中允许的最多人数？那是因为默认的列表视图脚手架最多只显示 6 个属性。如果你有定义约束属性，它将显示 `id` 属性和约束属性闭包中列出的前 5 个属性。Grails 不得不加上一点限制，目的是让产生的表格大小合理，以防止讨厌的水平滚动条。

我们还应该注意到这个脚手架只是为了给程序提供一个很好的起点，我们可以把它定制到令我们满意为止。所以，如果你想要其它字段，完全可以把它们加进来。但暂时先把这件事放到一边吧，我们很快就要给用户界面来个大变身。

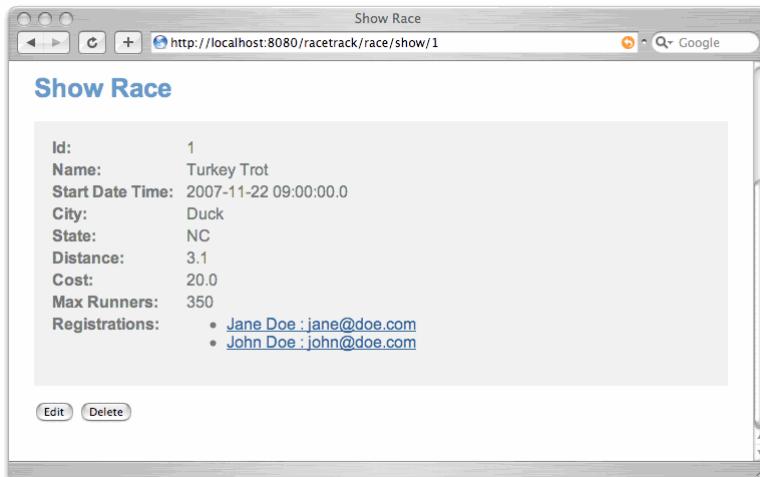
我们试下检验功能吧，怎么样？尝试毁坏一个程序总是件有趣的事情，试试看吧。



当我们试图创建一个无效的赛跑时, 我们不仅仅得到错误信息, 还得到一些相当不错的视觉上的提示来标志出有问题的字段。很自然地我们会想让错误信息的语言更加业务一点, 更少技术色彩, 这个我们马上就会处理到。而且, 我们很高兴地看到这个系统在防止数据输入的纰漏上表现不错。

我们的约束属性还让这个脚手架产生了其它的一些改进。现在“州”字段成了一个选择框 (原来的是文本框)。还有, 对于那些有最大输入长度约束的属性, 现在表格里相应的输入框都会限制输入的文本长度 (例如名字、城市等等)。所以, 我们现在同时拥有客户端的验证(通过 HTML 表格元素)和服务器端的验证 (通过领域类)。

最后, 我们检验一下记录标识符的改进。



好多了！我们现在可以清楚地看到 Jane Doe 和 John Doe 已经注册了这次赛跑。这可比之前显示在这里的神秘数字有用多了。

我们看看代码吧！

到目前为止，我们都在依赖声明式的脚手架，做法是直接在控制器里引进 `def scaffold = Race` 和 `def scaffold = Registration`。这个方法到目前为止都工作得很好，但我们现在已经准备好开始定制这个脚手架了。

我们可以加入新的 Action 到控制器里，就像现在存在的 Action 一样。因为声明式的脚手架动态地产生控制器逻辑，所以我们所定义的任何新 Action 将比默认的 Action 优先被处理（如果我们的新 Action 的跟默认 Action 同名）。然而，在接下来几章里我们只想对现有的控制器逻辑做小小的修改，因此为了提高编程效率，我们可以先生成脚手架的代码，然后定制它来满足我们的需求。现在我们需要去接触真正的代码了。那么我们就把脚手架逻辑实际生成出来吧。

我们想用包含展开了的脚手架代码的控制器来覆盖现有的控制器（以及它们仅有的三行代码）。重要的事情先来，我们先删除现

有的控制器类 `RaceController.groovy`、`RegistrationController.groovy`。

接着, 确保你在项目的根目录下 (在我们的例子中是 `racetrack`)。然后, 输入 `grails generate-all`。当被要求输入领域类名时, 输入 `Race`。

```
racetrack> grails generate-all
...
input-domain-class:
  [input] Enter domain class name:
Race
...
BUILD SUCCESSFUL
Total time: 11 seconds
```

重复这个过程, 同样为 `Registration` 领域类生成控制器和视图。现在, 我们得到了完整代码了, 并且可以随意去定制它。

理解 URL 和控制器

既然我们能够查看代码了, 那么我们看看 Grails 对于一个特定的 URL 是如何确定调用哪个控制器和 Action 的。(提示: 它并没有使用到任何一个 XML 配置文件!)

URL 规约

想想我们前面用到的 URL。

```
http://localhost:8080/racetrack/race/
http://localhost:8080/racetrack/race/create
http://localhost:8080/racetrack/registration/show/2
```

在 Grails 规约里, URL 的每个部分都扮演着重要的角色。这种遵循规约的路线 (谢天谢地) 把我们从编写 URL、控制器和视图的外部绑定配置文件中解放了出来。

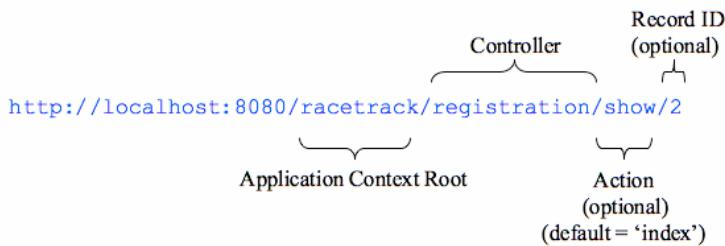


图 3-3: URL 结构

第一部分是程序上下文根目录，这部分当然是代表着在服务器上注册的应用程序的名字（默认情况下，Grails 用程序名称作为上下文根目录，并且在我们调用 `grails run-app` 时会自动在服务器上注册应用程序）。

下个部分表示的是处理请求的控制器。假如 URL 中的这个部分写的是“`registration`”，那么 Grails 将把请求发送到一个叫“`RegistrationController`”的控制器（如果没有这个控制器，我们会看到一个 HTTP 404 错误）。

Grails 接着寻找代表要调用的 Action 的部分，这个 Action 来自上一部分指定的控制器。如果 URL 中包含 Action 部分，那么 Grails 就会调用控制器中的相应 Action（如果该 Action 不存在，我们会得到一个错误提示）。如果 URL 中不包含 Action 部分，Grails 则会调用控制器中的默认 Action。除非在控制器里另有指定，不然 Grails 会查找一个叫“`index`”的默认 Action¹⁸。

最后，Grails 查找代表着一条特定记录 ID 的部分。如果 URL 中有这个部分，Grails 就在请求里增加一个“`id`”参数，并赋值。在上面的例子中，“`id`”参数将被赋值为“`2`”。

从请求到控制器再到视图

接下来我们看看 `RaceController.groovy` 中的一些代码示例，并且一步一步解释当 Grails 收到一个 <http://localhost:8080/racetrack/race> 请求时会发生什么事情。

首先, 因为没声明任何 Action, 所以 Grails 把请求发送到 index Action (提示: 所有 Action 都是以 Groovy 闭包的方式实现的)。

```
def index = {
    redirect(action:list,params:params)
}
```

index Action 的默认实现是把请求重定向到 list Action。 redirect 方法是 Grails 控制器里众多动态方法之一¹⁹。

```
def list = {
    if (!params.max) params.max = 10
    [ raceList: Race.list( params ) ]
}
```

list Action 的实现也相当简洁。第一行语句用到了内嵌的分页功能, 默认情况下, 每页最多显示 10 条记录。第二行语句是真正要注意的地方。它声明并返回一个只包含一个键 (raceList) 的 Map, 这个键的值是数据库中所有 Race 对象的集合。(现在先记住 list 方法是领域类的众多动态方法之一²⁰, 我们会很快开始详细探索这些方法。)

控制器写完了。不过当这个 Action 返回的时候, 是哪个部分负责去渲染我们在浏览器中看到的实际视图呢? 好了, 其实是规约 (再一次) 令 list 方法变得如此简洁。因为我们是从 RaceController 的 list Action 返回, 所以 Grails 知道应该使用放在 racetrack/grailsapp/views/race/list.gsp 的视图模板。

Grails 用 GSP (Groovy 服务器页面) 来给视图提供模板。如果你用过 JSP (JavaServer Pages), 你会认出相似的语法。如名字所暗示的, GSP 跟 JSP 最大的不同是用 Groovy 取代 Java 来编写脚本。你将会看到, 为了让我们直奔重点, Groovy 厚道地减少了我们需要编写的代码量。

现在, 让我们来探索一下 list.gsp。看看下面的模板片段, 我们可以看到它用 (由 list 方法返回的) raceList 变量来渲染

响应页面。

```
<g:each in="${raceList}">
  <tr>
    <td>${it.id}</td>
    <td>${it.name}</td>
    <td>${it.startTime}</td>
    <td>${it.city}</td>
    <td>${it.state}</td>
    <td>${it.distance}</td>
    <td class="actionButtons">
      <span class="actionButton">
        <g:link action="show" id="${it.id}">
          Show
        </g:link>
      </span>
    </td>
  </tr>
</g:each>
```

在这个例子中，`<g:each>`标签循环访问这个集合，并对集合中的每个对象都用标签体来渲染一次。在每个循环中，标签体通过`it` 变量访问集合的当前项。例如，`it.name` 返回当前 `Race` 对象的 `name` 属性的值。

当然，你不可能永远都让 Action 跟模板一一对应。所以，当没有同样名称的视图模板来跟 Action 配对时（或者你只是不想用那个模板），Grails 允许你明确地指定想要使用的视图模板。`save` 方法（在 `RaceController.groovy` 里）包含了这样的一个例子。

```
def save = {
  def race = new Race()
  race.properties = params
  if(race.save()) {
    redirect(action:show,id:race.id)
  }
  else {
    render(view:'create',model:[race:race])
  }
}
```

在上面的代码中，如果调用 `race.save()` 返回 `false`，这个 Action 指示 Grails 用 `race` 对象来填充表格，并渲染 `create` 模板（`racetrack/grailsapp/views/race/create.gsp`）。

控制器的生命周期

在我们结束这个主题之前, 应该稍微说明一下 Grails 控制器的生命周期。Grails 为每次请求都创建一个新的控制器实例, 而每个控制器实例只在处理请求期间短暂存活。所以如果有必要, 你可以放心地在控制器里增加实例变量, 而不用操心多用户线程问题。不过最好记住别在你以前用的传统 Java MVC 框架中尝试这种做法。

提升用户体验

到目前为止，开发出来的应用程序已经能满足我们的需要，不过还可以再打磨一下，让它对用户更友好。并且，既然我们希望用户在使用程序的时候能有享受的感觉，就让我们花一点时间在上面锦上添花。

- 错误信息——我们前面已经注意到，验证的错误信息还需要改进一下。我们应该定制一下这些信息，使之少一些技术词汇，对终端用户更加通俗易懂一点。
- 警告和确认信息——你曾试着删除过数据吗？你会因为程序删除数据而没向你要求任何确认而感到惊讶吗？绝大多数用户都希望能够在删除数据前得到警告。在完成添加、更新、删除数据的操作之后，我们也要显示确认信息。
- 记录 ID——虽然数据库和程序严重依赖 ID，但 ID 对最终用户来说没什么意义。最终用户只希望看到业务数据（比如竞赛名称、注册时间等），我们也不应该让他们操心记录 ID 之类的底层实现细节。
- 数据格式——真正专业的应用程序应该用恰当的格式来显示数据。这方面我们还需要做一些工作。例如，在注册页面上生日字段的日期显然不需要精确到小时，完全没必要。各种数字和货币值的格式，改进一下也会好看很多。

自定义错误消息

每个约束验证都有一系列预先定义的错误代码。到现在为止，我们看到的都是这些错误信息的默认文字。在我们开始自定义错误信息之前，最好先翻一下Grails参考文档，找到每个约束对应的错误代码¹²。

首先，让我们来看下忘了填比赛名称的时候出现的错误信息吧。

Create Race

http://localhost:8080/racetrack/race/save

Create Race

- Property [name] of class [class Race] cannot be blank

Name:

Start Date Time: July 2007 07 : 30

City: Atlanta

State: GA

Distance: 6.2

Cost: 28.0

Max Runners: 55000

Create

我们需要再看一遍领域类，找到我们给比赛的名称属性指定的约束。

```
static constraints = {
    name(maxLength:50,blank:false)
    //...
}
```

检查空白值的约束属性的名称是“blank”。我们现在可以在 Grails 参考文档中找到这个约束。

Grails : Validation Reference

file://usr/local/grails/grails-docs/Validation%20Reference.html

Validation Constraint Reference

Name	Description	Example	Error Codes
blank	set to false if a string value cannot be blank	login(blank:false)	className.propertyName.blank
email	set to true if a string value is an email address	contactEmail(email:true)	className.propertyName.email.invalid

文档告诉我们，这个约束生成的错误代码按照下面的格式命名。

```
className.propertyName.blank
```

所以，当我们自定义错误消息的时候，需要在 `racetrack/grailsapp/i18n/messages.properties` 里面添加如下形式的条目

```
race.name.blank=Please enter a name for this race
```

添加完这个条目后，让我们重新启动应用来看看结果。

在一些约束属性上面，Grails 还提供了运行时参数，让你插入到消息里面。你可以查看默认的错误消息（在 `messages.properties` 里）来帮助你确定每个参数的值。例如，`max` 约束的错误消息里面有一个参数，它定义了允许的最大值。这些信息将极大地提高我们的应用程序的可用性，当然我们不需要在 `message.properties` 里面重复定义最大值，因为约束自身已经定义过了（在 `Race.groovy` 里面）。从这个约束的默认消

息（定义为 `default.invalid.max.length.message`）里面可以看到，消息的第四个参数是显示最大值的。（因为参数数组索引是以 0 开始，所以我们访问第四个参数时索引值为 3。）

```
Property [{0}] of class [{1}] with value [{2}]
exceeds the maximum length of [{3}]
```

所以，我们可以在 `message.properties` 里面定义下面的条目，来指定当赛跑的长度超过允许的最大值的时候，出现的错误文字。

```
race.distance.max.exceeded=Please enter a valid
distance no more than {3} miles
```

现在，用户找到出错的地方所需要的全部信息，都在我们的错误消息里面了。

The screenshot shows a 'Create Race' form on a web browser. The browser title is 'Create Race' and the URL is 'http://localhost:8080/racetrack/race/save'. The form fields are: Name (Peachtree Road Race), Start Date Time (4 July 2007 07:30), City (Atlanta), State (GA), Distance (1000.0), Cost (28.0), and Max Runners (55000). A validation error message 'Please enter a valid distance no more than 100 miles' is displayed in a red-bordered box above the Distance field.

我们现在已经知道怎么自定义错误消息了，只需要对剩下的约束属性重复上述过程（当然，你最好一次过把要改的消息都改完，除非你真的很喜欢不停地重启服务器）。

添加警告信息

毫无疑问，用户希望在执行任何具有破坏性的操作之前，程序都能够提出警告。删除数据当然是一种破坏性行为，尤其是当删除一场比赛同时也会删除所有相关的报名人员的情况下。所以，让我们添加一点逻辑，确保在删除任何数据之前，都让用户再次确认。

因为在显示比赛和编辑比赛页面上用户都能删除比赛，所以我们希望把两个页面上用到的警告逻辑都放到同一个地方。我们采用 JavaScript 对话框来显示警告信息，所以，让我们先创建一个文件（`racetrack/web-app/js/racetrack.js`）来存放通用的 JavaScript 函数。把下面的代码加进 `racetrack.js` 文件中。

```
function warnBeforeRaceDelete() {
    return confirm('Are you sure you want to delete
this race?')
}
```

现在我们需要在显示比赛和编辑比赛页面上调用此函数。打开显示比赛页面的模版文件（`racetrack/grails-app/views/race/show.gsp`），然后添加下面的脚本标签，这样就可以在页面中调用 `racetrack.js` 中定义的函数。

```
<head>
<meta http-equiv="Content-Type" content="text/html;
charset=UTF-8" />
<g:javascript library="racetrack" />
<meta name="layout" content="main" />
<title>Show Race</title>
</head>
```

然后，找到删除按钮，并给 `onclick` 属性加上下面的代码，让它调用我们刚才写的 JavaScript 函数。如果用户选择删除比赛，那么函数返回 `true`，浏览器就会提交这个表单。否则函数返回 `false`，浏览器将不会做任何操作。

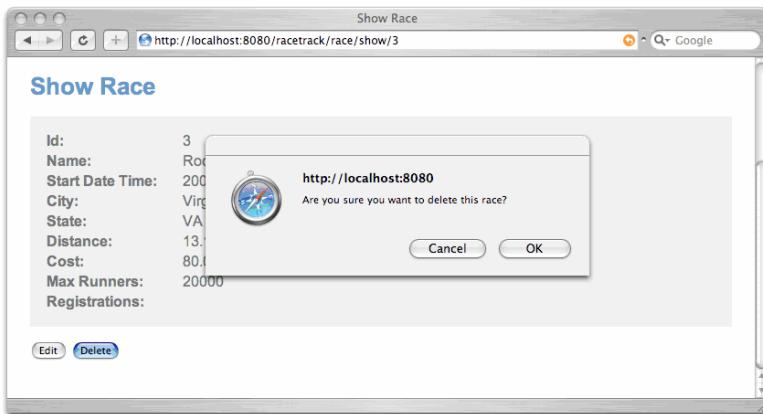
```
<g:form controller="race">
    <input type="hidden" name="id" value="${race?.id}" />
    <span class="button">
        <g:actionSubmit value="Edit" />
    </span>
```

```

<span class="button">
    <g:actionSubmit value="Delete"
        onclick="return
        warnBeforeRaceDelete();"
    />
</span>
</g:form>

```

现在，让我们试着在显示比赛页面上删除比赛。



这正是我们希望看到的。非常简单，但是也是非常重要的。继续修改一下编辑比赛页面，让它也调用这个函数。然后，我们再为删除注册也提供一个类似的 JavaScript 函数来发出警告，并把它加进显示注册和编辑注册页面。

实现确认信息

确认信息只是一个相当简单的特性，但如果应用程序没有确认信息的话，用户会忍不住怀疑程序是不是真的已经完成了操作。虽然说用户能在程序里看得到自己刚才操作的结果：当用户更新了一个比赛，程序就会进入显示比赛页面并显示更新后的数据。但是，还是需要用户自己去检查数据来确认更改。一个简单的确认信息就能让用户大大提升对程序的信心。

也许你会注意到，程序已经在我们删除数据后给出确认信息

了。我们可以参考这个已经存在的逻辑，为增加和更新记录添加确认信息。

在创建和更新比赛之后，程序将会进入显示比赛页面。如果你查看这个页面的模版的话（`racetrack/grails-app/views/race/show.gsp`），你会发现里面已经包含了显示确认信息（或者其它任何我们认为合适的消息）的逻辑。

```
<g:if test="${flash.message}">
    <div class="message">${flash.message}</div>
</g:if>
```

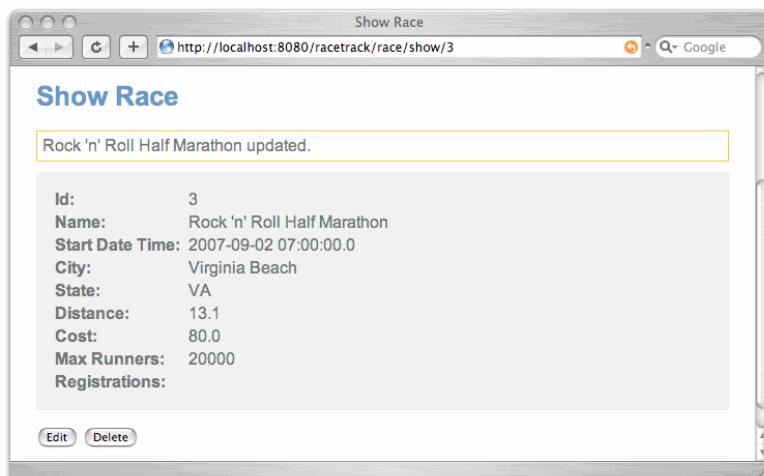
先看一下控制器中提供确认信息的逻辑，我们接下来会详细讨论上面的几行代码。打开 `RaceController.groovy` 找到 `update` Action。添加如下所示的代码，把我们成功保存比赛后要显示的确认消息赋值给 `flash.message`（在闭包的最后，你可以看到默认的脚手架已用了 `flash.message` 来显示另一条消息）。

```
def update = {
    def race = Race.get( params.id )
    if(race) {
        race.properties = params
        if(race.save()) {
            flash.message = "${params.name} updated."
            redirect(action:show,id:race.id)
        }
        else {
            render(view:'edit',model:[race:race])
        }
    }
    else {
        flash.message = "Race not found with id
                        ${params.id}"
        redirect(action:edit,id:params.id)
    }
}
```

`flash.message` 从哪里来的？Grails 有一个叫“Flash”的作用域。像控制器和模版使用的 `Request` 和 `Session` 作用域一样，“Flash”作用域也可以被程序访问。和 `Request`、`Session` 里面的参数一样，`flash` 对象也是一个单纯的键值对组成的 `Map`。Flash 作用域的独特之处是它的生命周期。储存在 Flash 作用域里的对象，在当前请求和下一次请求期间都可以被使用。在上面的例子里，我

们重定向到了显示比赛页面。重定向会产生新的 Request，如果我们把消息保存在 Request 里面，当打开显示比赛页面的时候，就再也访问不到刚才保存的消息了。但如果把对象保存在 Flash 作用域里，即使在重定向后我们仍然可以获得之前的消息。并且，在第二次请求结束的时候，Grails 会自动清除 Flash 作用域里面的对象，并不需要手动清除。

让我们试着更新一个比赛，当我们看到了确认信息后，是不是安心了不少？



看起来不错。现在让我们在 `save` Action 上添加代码，让它在创建一个新的比赛后也显示类似的信息。

```
def save = {
    def race = new Race()
    race.properties = params
    if(race.save()) {
        flash.message = "${params.name} saved."
        redirect(action:show,id:race.id)
    }
    else {
        render(view:'create',model:[race:race])
    }
}
```

运行程序看一下刚刚加上的新消息，这样我们就把比赛部分的确认信息添加完了。让我们再花一点时间给注册部分也添加类似的信息。然后我们就可以开始下一个任务了。

移除数据 ID

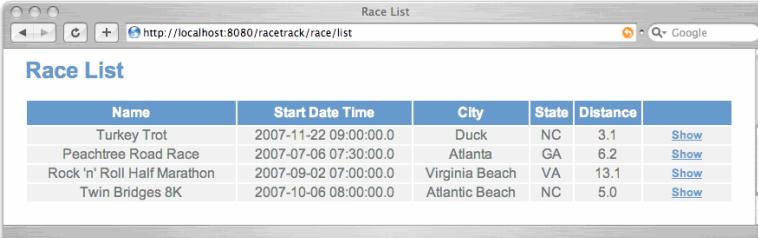
有时候，去除程序里的某些特性会让程序变得更好。数据 ID 就属于这种情况。我们的终端用户是不会去关心它的，这个理由就足以让我们将数据 ID 从用户界面上删除了。让我们从比赛列表页面开始。在 `racetrack/grails-app/views/race/list.gsp` 中找到如下代码片段，并删除 ID 列。

```

<tr>
    <th>Id</th>
    <th>Name</th>
    <th>Start Date Time</th>
    <th>City</th>
    <th>State</th>
    <th>Distance</th>
    <th></th>
</tr>
<g:each in="${raceList}">
    <tr>
        <td>${it.id}</td>
        <td>${it.name}</td>
        <td>${it.startTime}</td>
        <td>${it.city}</td>
        <td>${it.state}</td>
        <td>${it.distance}</td>
        <td class="actionButtons">
            <span class="actionButton">
                <g:link action="show" id="${it.id}">
                    Show
                </g:link>
            </span>
        </td>
    </tr>
</g:each>

```

现在，当用户查看比赛列表页面的时候，就只会看到业务数据。



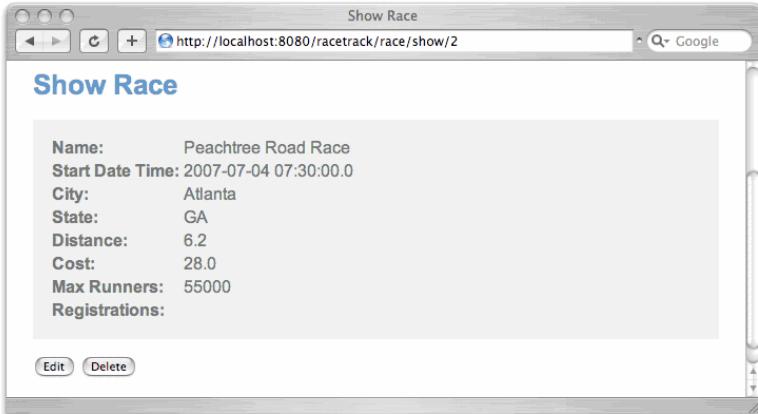
The screenshot shows a web browser window titled 'Race List' with the URL <http://localhost:8080/racetrack/race/list>. The page displays a table of race information with the following data:

Name	Start Date Time	City	State	Distance	
Turkey Trot	2007-11-22 09:00:00.0	Duck	NC	3.1	Show
Peachtree Road Race	2007-07-06 07:30:00.0	Atlanta	GA	6.2	Show
Rock 'n' Roll Half Marathon	2007-09-02 07:00:00.0	Virginia Beach	VA	13.1	Show
Twin Bridges 8K	2007-10-06 08:00:00.0	Atlantic Beach	NC	5.0	Show

数据 ID 还出现在显示比赛页面，编辑比赛页面，以及类似的用户注册管理页面中。我们需要将这些页面里的 ID 都找出来加以删除。

格式化数据

合适的数据格式毫无疑问会改善一个应用。不用特意去找，在显示比赛页面就有难看的时间/日期，不正确的货币精度，缺少货币符号，等等好几个需要改进的地方。



The screenshot shows a web browser window titled 'Show Race' with the URL <http://localhost:8080/racetrack/race/show/2>. The page displays the following race details:

Name: Peachtree Road Race
 Start Date Time: 2007-07-04 07:30:00.0
 City: Atlanta
 State: GA
 Distance: 6.2
 Cost: 28.0
 Max Runners: 55000
 Registrations:

At the bottom of the page are two buttons: 'Edit' and 'Delete'.

除了为表格单元格设置样式外，程序采用每个属性的 `toString` 方法来渲染它们的输出。

```

<tr class="prop">
  <td valign="top" class="name">Start
  Date/Time:</td>
  <td valign="top" class="value">

```

```

    ${race.startTime}
  </td>
</tr>

```

除了依赖 `toString` 方法外，我们需要一个途径给每个时间和数字定义我们希望的格式。我们可以将格式化逻辑写死在视图模版中，但是这样写无疑很难看，我们最终会得到一堆重复的代码。在这种情况下，更好的选择是实现一个自定义标签库。幸运的是，在 Grails 里面添加新的标签是件很轻松的事情。

自定义标签库

让我们开始格式化时间/日期。如果我们把标签看作是一组工具方法，我们可以问问自己，应该提供什么参数，获得什么返回值。在这里，我们希望提供一个 `Date` 对象和一个表明格式的字符串作为参数。（我们可以使用 `java.text.SimpleDateFormat` 里面定义的格式化语法。）接着，我们希望标签能够按照给定格式输出日期。当我们实现了这个新标签（叫做 `formatDate` 应该比较贴切），我们就可以用自定义标签来输出格式化后的日期了。

```

<tr class="prop">
  <td valign="top" class="name">Start
  Date/Time:</td>
  <td valign="top" class="value">
    <g:formatDate date="${race.startTime}"
                  format="yyyy-MMM-dd HH:mm"/>
  </td>
</tr>

```

现在格式化日期标签的基本 API 已经有了，让我们再看看格式化数字的需求。（我们首先确定标签的 API，然后再去讨论标签库的实现。）定义这个标签的方式与 `formatDate` 标签相似。我们有一个数字和想要的格式（这里我们将用 `java.text.DecimalFormat` 里面定义的格式语法），并且我们希望输出给定格式的数字。听起来够直白。那么在显示比赛页面上的各种数值，采用我们的新标签后会是什么样子呢？

```

<tr class="prop">
  <td valign="top" class="name">Distance:</td>
  <td valign="top" class="value">
    <g:formatNumber number="${race.distance}"
                  format="###0.0 mi"/>
  </td>
</tr>

```

```

        </td>
    </tr>
<tr class="prop">
    <td valign="top" class="name">Cost:</td>
    <td valign="top" class="value">
        <g:formatNumber number="\${race.cost}" format="\$##0.00"/>
    </td>
</tr>
<tr class="prop">
    <td valign="top" class="name">Max Runners:</td>
    <td valign="top" class="value">
        <g:formatNumber number="\${race.maxRunners}" format="###,##0"/>
    </td>
</tr>

```

定义好了 API，现在我们准备开始对这个标签库进行编码。Grails 标签库是用 Groovy 类（理所当然）写的。每个标签被定义成一个闭包，这个闭包将接受一个 Map 作为运行时参数，标签属性就放在里面。

让我们为自定义标签定义一个新的 Groovy 类。我们将创建一个叫 `RaceTrackTagLib.groovy` 的文件，放在 `racetrack/grails-app/taglib` 目录下。

```

class RaceTrackTagLib {
    /**
     * Outputs the given <code>Date</code> object
     * in the specified format. If the
     * <code>date</code> is not given, then the
     * current date/time is used. If the
     * <code>format</code> option is not given, then
     * the date is output using the default format.
     *
     * e.g.:
     * <g:formatDate date="\${myDate}" format="yyyy-MM-dd HH:mm"/>
     *
     * @see java.text.SimpleDateFormat
     */
    def formatDate = { attrs ->
        def date = attrs.get('date')

        if (!date) {
            date = new Date()
        }
    }
}

```

```

def format = attrs.get('format')
if (!format) {
    format = "yyyy-MM-dd HH:mm:ss z"
}

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

/**
 * Outputs the given number in the specified
 * format. If no <code>number</code> is given,
 * then zero is used. If the <code>format</code>
 * option is not given, then the number is output
 * using the default format.
 *
 * e.g.:
 * <g:formatNumber number="${myNumber}"
 *                  format="###,##0" />
 *
 * @see java.text.DecimalFormat
 */
def formatNumber = { attrs ->
    def number = attrs.get('number')

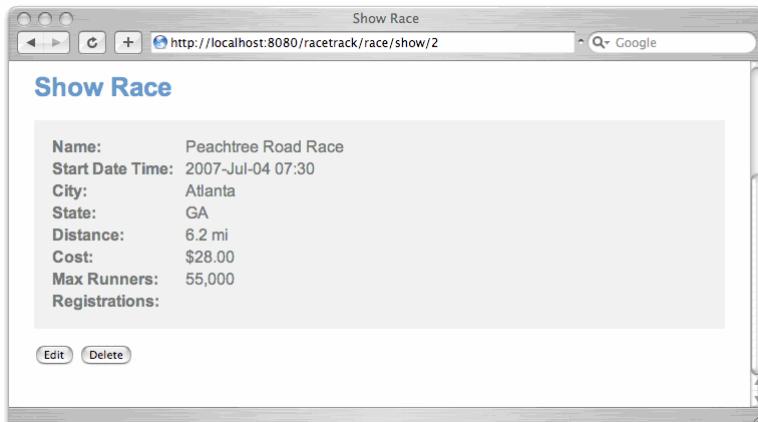
    if (!number) {
        number = new Double(0)
    }

    def format = attrs.get('format')
    if (!format) {
        format = "0"
    }

    out << new java.text.DecimalFormat(format)
        .format((Double)number)
}
}

```

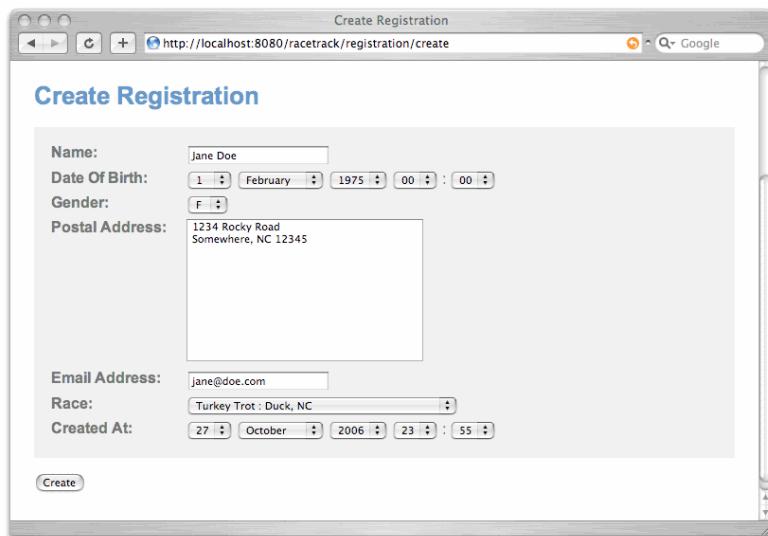
最后，我们需要注册 TLD (Tag Library Descriptor)。等等！去掉这一步，Grails 并不需要任何 TLD。TLD 代表了配置，而 Grails 选择了规约。还记得我们把标签库放在 `racetrack/grails-app/taglib` 吗？这就是当 Grails 在显示模版中遇到一个标签的时候，它会去查找的地方。Grails 会找到每个标签的闭包并执行它们，就这么简单。我们可以测试一下了，让我们看看正确格式化后的数据。



有进步吧？当然，其它页面也要分享一下这些改进。比赛列表页面也要格式化得好看一点，创建比赛和编辑比赛页面中距离字段应该包含长度单位，成本字段应该有货币符号。当你完成上述改进之后，我们就可以开始下一个任务了。

标准标签库

我们总不能老是自定义标签库。幸运的是，Grails提供了一套健全的标准标签库²¹。我们要调整一下报名比赛的相关页面中标签的用法。



以创建注册页面为例，这个页面里用了 `datePicker` 标签来显示出生日期和报名时间这两个字段。报名时间字段表示报名的时间戳，一般来说我们并不需要用户输入，实际上，我们根本不打算让用户编辑这个字段；否则我们可能会得到一个错误的报名时间。另外我们上面提到过，参赛者的出生日期并不需要精确到小时。

回到编辑器，打开创建注册页面的模版（`racetrack/grails-app/views/registration/create.gsp`）。现在，找到模版中渲染出生日期字段的部分。如果看了Grails的文档中关于`datePicker`标签的说明²²，我们就知道，它支持一个可选的精度参数。这就是我们所需要的。让我们在出生日期字段的标签里添加参数，让它只显示年月日。

```
<g:datePicker name='dateOfBirth'
               value='${registration?.dateOfBirth}'
               precision='day'>
</g:datePicker>
```

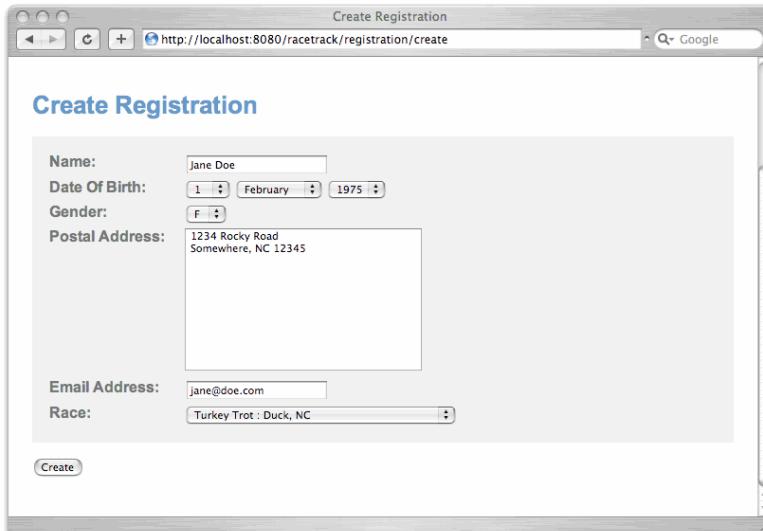
真简单。好了，开始我们的下一个改进。保证我们获得正确的

注册时间非常简单，将报名时间字段从页面中删除就足够了。

找到 `Registration` 类，将 `createdAt` 属性初始化为当前时间。这样，当 Grails 根据提交的表单创建一个新的 `Registration` 对象的时候，该对象将会具有正确的创建时间（即当前时间）。

```
class Registration {  
    //...  
    Date createdAt = new Date()  
    //...  
}
```

现在，经过我们的微调，页面上的数据更加符合需要了。



我们还应该在其它与注册相关的页面中也找一下类似的字段。调整所有页面中出生日期字段的精度。在编辑注册页面中，我们只要简单地按照上面的方式调整 `DatePicker` 标签的精度。在注册列表和显示注册页面中，我们可以用新创建的 `formatDate` 标签，让它只显示年月日。

现在只剩下报名时间字段要做些处理了。我们并不希望任何人修改此字段的值。这个时间在报名被创建的时候就已经确定了，历史不应被篡改。现在编辑注册页面上还允许用户修改报名时间字段，我们应该把这个字段作为只读字段来显示。花点时间调整一下这个小问题，然后我们准备进入下一个迭代。

InfoQ 中文站 Agile 社区
敏捷软件开发和项目管理
<http://www.infoq.com/cn/agile/>

5

动态

好了，我们已经和用户一起完成了最初的几个迭代。我们收到了反馈，到现在为止他们对所见所闻都感到满意。他们对我们这么快就把应用构建起来，并连续不断地让他们看到逐步的改进而感到高兴。在完成了前面看到的基本 CRUD 功能后，我们接着得到的需求是“我需要得到__月的所有比赛日程”或者“列出所有在__地方举行的__米赛跑”。这些都是很合理的需求。Grails 也会继续帮助我们快速地完成这些新报表，让客户惊喜。

还记得我们讲到自动生成的控制器代码的时候，曾经简短地提到过的动态方法吗？我们曾经用 `list` 方法来获取所有的比赛并显示在比赛列表页面上，它是一个所有的领域类都共有的动态方法。我们并不需要在领域类中写任何额外的代码来获得这个功能。它是 Grails 领域类的一个标配。当我们开始实现上面的新的查询需求时，我们会更深入地去看一下 Grails 领域类提供的其它动态方法。

动态查询器

首先，假设我们需要找到在一个城市举行的所有比赛。我们将在比赛列表页面上添加一个新的菜单项，用来进入新的查询页面。将下面这段添加进 `racetrack/grails-app/views/race/list.gsp` 页面中的菜单部分。

```
<span class="menuButton">
    <g:link action="search">Search for Races</g:link>
</span>
```

点击这个菜单项将调用控制器里面一个叫 `search` 的新 Action。我们暂时只需要它显示搜索框页面。所以，我们在

RaceController.groovy 里面添加一个空的闭包，来接受这个请求。

```
def search = {  
}
```

这个空的闭包告诉 Grails，这个控制器里面有一个叫做 search 的 Action。当 Grails 接收到对这个 Action 的请求，它会直接在 racetrack/grails-app/views/race/ 目录下查找具有相同名字的模版（即 search.gsp），然后渲染模版的内容。

由于我们只想为比赛建立一个简单的带输入框的表单，create.gsp 模版是一个不错的参考。完成之后，我们新建立的 search.gsp 模版应该看起来像这样：

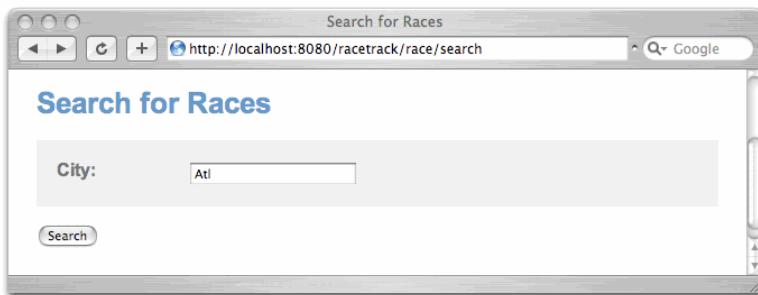
```
<html>  
  <head>  
    <meta http-equiv="Content-Type"  
          content="text/html; charset=UTF-8" />  
    <meta name="layout" content="main" />  
    <title>Search for Races</title>  
  </head>  
  <body>  
    <div class="body">  
      <h1>Search for Races</h1>  
      <g:if test="${flash.message}">  
        <div class="message">  
          ${flash.message}  
        </div>  
      </g:if>  
      <g:form action="search" method="post">  
        <div class="dialog">  
          <table>  
            <tr class='prop'>  
              <td valign='top' class='name'>  
                <label for='city'>City:</label>  
              </td>  
              <td valign='top' class='value'>  
                <input type="text" maxlength='30'  
                      name='city'>  
              </input>  
            </td>  
          </tr>  
        </table>  
      </div>  
      <div class="buttons">  
        <input type="submit" value="Search" />
```

```

        class="formbutton">
    </input>
</div>
</g:form>
</div>
</body>
</html>

```

现在，新的搜索页面准备就绪。



你也许注意到了，在 `search.gsp` 中我们让表单把所有的请求 `post` 到 `search Action`。

```
<g:form action="search" method="post" >
```

当前，我们的 `search Action` 是空的，只是单纯地转到搜索表单页面。现在我们需要添加一点逻辑来实现搜索。

```

def search = {
    if (request.method == 'POST') {
        render(view:'list', model:[ raceList:
            Race.findAllByCityLike('%' + params.city +
            '%') ])
    }
}

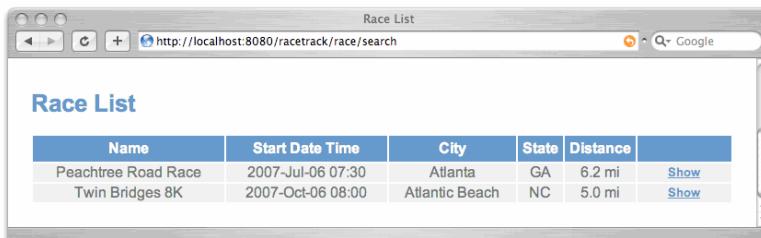
```

因为我们使用同一个 `Action` 来显示搜索表单页面和输出搜索结果，我们需要一个途径来确定收到请求的时候到底应该执行哪个功能。我们知道，查询参数将会通过 `post` 请求得到，那么当遇到其它类型的请求——一般会是 `get` 请求——我们就显示搜索表单页面。

当我们遇到一个 `post` 请求的时候，这段代码又做了些什么呢？好的，这里又一次体现了 Grails 用简洁的命令完成重大功能的能力。对于所有的 `post` 请求，`search` Action 将会……

- 通过 `params.city` 从 Request 中获得搜索的条件。
- 用 `findAllByCityLike` 方法——许多动态查询器方法中的一个——找出相应城市举行的所有比赛，得到一个 `Race` 对象列表。
- 将结果放入一个叫 `raceList` 的对象中。
- 渲染页面，将 `raceList` 设置为 `list` 视图模版中的 `model`。（因为我们只打算显示满足条件的一个比赛列表，所以可以直接用已经存在的比赛列表模版——`racetrack/grails-app/views/race/list.gsp`）

按下搜索之后，果然，我们得到了匹配的结果。



The screenshot shows a web browser window with the title 'Race List'. The address bar displays the URL `http://localhost:8080/racetrack/race/search`. The main content area is a table titled 'Race List' with the following data:

Name	Start Date Time	City	State	Distance	Action
Peachtree Road Race	2007-Jul-06 07:30	Atlanta	GA	6.2 mi	Show
Twin Bridges 8K	2007-Oct-06 08:00	Atlantic Beach	NC	5.0 mi	Show

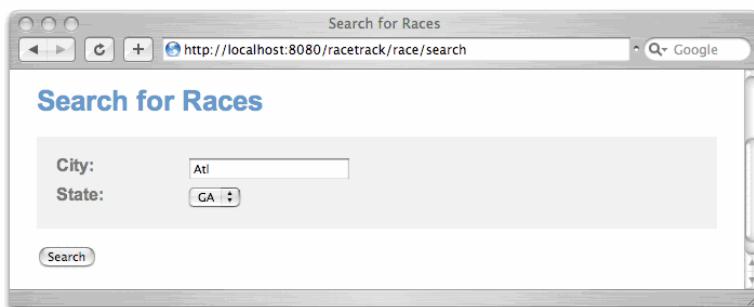
如果我们能够按城市搜索比赛，为什么不能按州来搜索呢？我们很容易在搜索页面上添加一个新的输入框。

```

<tr class='prop'>
    <td valign='top' class='name'>
        <label for='state'>State:</label>
    </td>
    <td valign='top' class='value'>
        <g:select name='state'
            from='${[""] + new
Race().constraints.state.inList}'>
            </g:select>
    </td>
</tr>

```

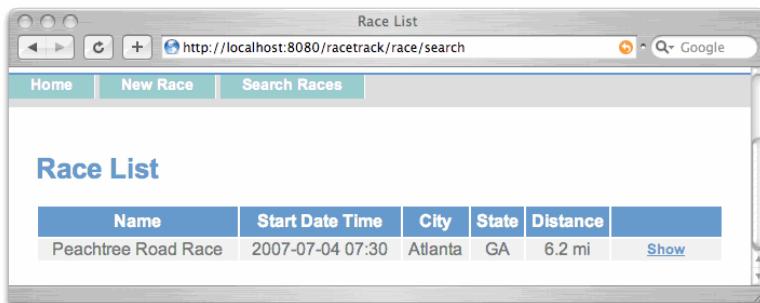
城市用简单的输入框就可以了，而州名应该用一个下拉选择框更加合适。要构建一个选择框，我们需要知道有哪些选项可供选择。我们当然不希望把州的列表硬编码到页面上。幸运的是，我们可以从领域类中获得有效州名的列表——领域类应该是最佳信息来源。另外，因为用户可能有时候不想按州搜索，所以我们把一个空白的选项放在列表的第一位。



上面所做的非常简单，那么控制器里面又应该怎么修改呢？其实，我们只需要做一点点改动。我们这次不用 `findAllByCityNameLike` 方法，改为调用另外一个动态查询器 `findAllByCityLikeAndStateLike`。

```
def search = {
    if (request.method == 'POST') {
        render( view:'list',
            model:[raceList:
                Race.findAllByCityLikeAndStateLike
                ('%' + params.city + '%',
                 '%' + params.state + '%')
            ]
        )
    }
}
```

任何你能想到的查询组合，Grails 都能够自动提供动态查询器！试试看，上面的搜索得到的正好就是我们预期的查询结果。



构建自己的查询条件

慢慢地，我们很可能会发现需求越来越复杂了。（这样很好，能让我们继续专注于工作）。那么，我们怎样才能处理更多具有不同数据类型、不同条件操作符的查询参数？举个例子，假设我想要找到所有距离至少 5 英里，并且时间在 7 月 1 日和 9 月 15 日之间的所有比赛。这样的查询已经超出了我们上面看到的动态查询器能力之外了。但是不用担心，这并不意味着你的程序必须增加额外的复杂性。Grails 提供了非常灵活的 Hibernate Criteria Builder²³，既能满足这样的需求，又不用去触动到我们现在已有的任何功能。

小提示：如果觉得你的程序并不需要这种类型的查询，你可以略过下面的内容直接去阅读第六章。当你有需要的时候再随时回来继续阅读这一节。

介绍 Hibernate Criteria Builder

在我们添加新的查询参数之前，先让我们花点时间理解一下 Hibernate Criteria Builder 的结构。为此，让我们用 Hibernate Criteria Builder 来代替现在 search Action 所用的动态查询器。

```
def search = {
    if (request.method == 'POST') {
        def criteria = Race.createCriteria()

        def results = criteria {
            and {
                like('city', '%' + params.city + '%')
                like('state', '%' + params.state + '%')
            }
        }
    }
}
```

```

        }
    }

    render(view:'list', model:[ raceList: results ])
}
}

```

即使你以前从来没有用过 Hibernate 或者 Grails，你也能够迅速地理解我们所做的要点。Hibernate Criteria Builder 是 Groovy Builder 的一种，在 Groovy Builder 里面，每个部分都被称为一个“节点”。and 节点的意思是，只要满足它的子节点定义的一个或多个查询条件，记录就会被匹配。在这里，它有两个子节点。

第一个子节点将会返回在 `city` 属性中满足采用 `like` 操作，大小写完全匹配的所有结果。

```
like('city', '%' + params.city + '%')
```

类似地，第二个子节点返回在 `state` 属性中满足采用 `like` 操作，大小写完全匹配的所有结果。

```
like('state', '%' + params.state + '%')
```

这些节点合在一起，等价于我们之前使用的动态查询器。然而，现在我们能够非常方便地扩展这个 `Builder`，来支持更加高级的查询需求。

如果你觉得 `Builder` 的结构看起来有些怪异，那是因为 Hibernate Criteria 采用了波兰式语法（Polish Notation），你要把操作符放到操作数的前面²⁴。虽然和 Java 的关系运算符有些不同，但当你写过几次这样的查询后就会觉得相对直观了。

改进视图

现在，让我们来修改搜索页面，让它能够获得我们新需求中的输入。让我们编辑 `search.gsp`，添加了下面的几个输入框。

```

<tr class='prop'>
    <td valign='top' class='name'>
        <label for='date'>Date:</label>
    </td>

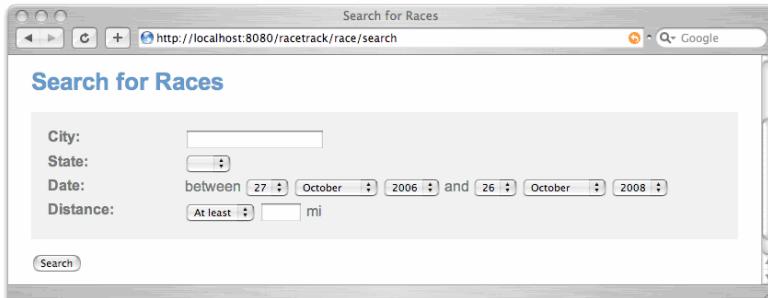
```

```

<td valign='top' class='value'>
    between
    <g: datePicker name='minDate' precision='day' />
    and
    <g: datePicker name='maxDate' precision='day'
    value='${new Date().plus(365*2)}' />
</td>
</tr>
<tr class='prop'>
    <td valign='top' class='name'>
        <label for='distance'>Distance:</label>
    </td>
    <td valign='top' class='value'>
        <select name='distanceOperator' >
            <option value='AT_LEAST' >At least</option>
            <option value='EXACTLY' >Exactly</option>
            <option value='AT_MOST' >At most</option>
        </select>
        <input type='text' name='distance' size='5'>
        </input> mi
    </td>
</tr>

```

来一次快速的完整性检查，以确认我们仍在正确的方向上。



使用 Java 类

你也许注意到了，现在我们获得的查询参数并不直接映射到我们的领域类上。比如，查询参数中现在包含了一个最早日期和最晚日期，但是我们的领域类中只包含了一个日期——比赛开始的日期/时间。同样地，现在赛跑距离还跟一个查询运算符（即至少、正好或者至多）联系起来了，但是在比赛的领域模型中只有赛跑的距离。简而言之，我们缺少一个恰当的映射来把领域模型跟查询参数

一一对应起来。在这种情况下，我选择定义一个类来代表查询本身。让我们添加一个 `RaceQuery` 类来实现这个需求。

这样的一个类应该放在我们项目结构里的什么地方呢？它本质上并不是一个领域类，因为我们不需要持久化任何一个查询对象。它也不是一个控制器，当然，它更不是一个视图。这就是 `racetrack/src` 目录出场的时候了，我们可以在这个目录里添加 Grails 程序里所用到的 Groovy 或者 Java 类。

距离的运算符有三种值——至少、正好和至多——刚好适合用枚举类型来实现。如果我们希望使用枚举类型的话，我们需要在 Java 里实现 `RaceQuery` 类（Groovy 以后会支持枚举类型，不过我们要等到 Groovy 2.0²⁵）。

Java 5 相关提示：如果你正在使用 Java 5 之前的版本，就要改用另一种机制（例如 `java.lang.String` 对象）来获得距离的运算符的值。

闲话少说，让我们来实现这个新类 `RaceQuery.java`，并且将它放在 `racetrack/src/java` 目录下。

```
import java.util.Date;

public class RaceQuery {
    public enum DistanceOperator {AT_LEAST, EXACTLY,
    AT_MOST};

    private Date minDate;
    private Date maxDate;
    private String city;
    private String state;
    private Float distance;
    private DistanceOperator distanceOperator;

    public String getCity() {
        return city;
    }

    public void setCity(String city) {
        this.city = city;
    }

    // remaining getters and setters not shown
```

```
// ...
}
```

现在我们有了一个用来捕获查询参数的类，接下来我们准备在 Builder 中使用它。

扩展 Criteria Builder

让我们更新一下 Builder，让它用上我们新的查询对象和由该对象提供的新的查询参数（用这段代码替换掉 RaceController.groovy 里面 searchAction 现在的代码）。

```
def search = {
    if (request.method == 'POST') {
        RaceQuery query = new RaceQuery()
        bindData(query, params)

        def criteria = Race.createCriteria()

        def results = criteria {
            and {
                like('city', '%' + query.city + '%')
                like('state', '%' + query.state + '%')
                if (query.distance) {
                    switch (query.distanceOperator) {
                        case
                    RaceQuery.DistanceOperator.AT_LEAST:
                        ge('distance', query.distance)
                        break
                    case
                    RaceQuery.DistanceOperator.EXACTLY:
                        eq('distance', query.distance)
                        break
                    case
                    RaceQuery.DistanceOperator.AT_MOST:
                        le('distance', query.distance)
                        break
                    default:
                        log.error "Found unexpected
value for distance" + " operator -
${query.distanceOperator}"
                    }
                }
            }
        }
        // Add 1 day (24 hours) to the max date.
        // (If user selects a max date of Jan
        // 1st, the date object will hold Jan
        // 1st 00:00, but the user will want any
        // events occurring thru Jan 1st 23:59.)
    }
}
```

```

        between('startDateTime', query.minDate,
query.maxDate + 1)
    } //and
}
render(view:'list', model:[ raceList: results ])
} //if
}
}

```

代码已经看到了，现在让我们逐一解释为了让这个 Action 使用 `RaceQuery` 对象和新的参数而做的每一处修改。

- 首先，我们添加代码来实例化一个新的 `RaceQuery` 对象，并将请求参数绑定到这个 `RaceQuery` 对象中。Grails 为所有的控制器提供了一个动态的 `bindData` 方法。这个方法会查看请求参数和目标对象（在这里是 `RaceQuery` 对象），然后把请求里相应的参数（按照名称来判断）装载到目标对象中去，并且根据需要适当地转换类型。

```
RaceQuery query = new RaceQuery()
bindData(query, params)
```

- 然后，我们在 `Builder` 中添加了距离相关参数的逻辑。`switch` 语句判断 `distanceOperator` 属性，并添加适当的节点到 `Builder` 中。比如如果一个查询要找的是至少 5 英里距离的赛跑，`Builder` 将会包含一个 `ge` 节点，表示赛跑的距离大于或者等于给定的距离。

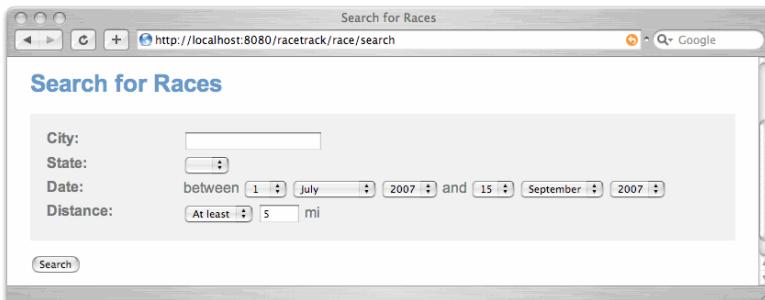
```
ge('distance', query.distance)
```

- 最后，我们在 `Builder` 中加入日期相关参数的节点。`between` 节点会匹配出比赛开始时间在给定的最早和最迟日期之间的所有比赛（包括这两个日期）。

```
between('startDateTime', query.minDate, query.maxDate
+ 1)
```

现在，让我们来看看运行的情况。首先，我们需要重启应用，以便让新的 `RaceQuery` 类起作用。然后就可以试一下了。重新查找距离至少 5 英里，并且日期在 7 月 1 日到 9 月 15 日之间的所有

比赛。



Search for Races

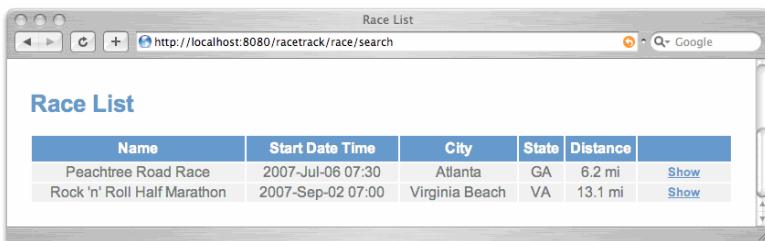
City:

State:

Date: between July 2007 and September 2007

Distance: At least 5 mi

当我们点击搜索之后，将会看到如下的内容。



Race List

Name	Start Date Time	City	State	Distance	
Peachtree Road Race	2007-Jul-06 07:30	Atlanta	GA	6.2 mi	Show
Rock 'n' Roll Half Marathon	2007-Sep-02 07:00	Virginia Beach	VA	13.1 mi	Show

6

InfoQ 中文站 Architecture 社区
设计、技术趋势，以及架构师感兴趣的话题
<http://www.infoq.com/cn/architecture/>

并不仅限于内部网络的应用

到现时为止，我们已经把所有管理比赛和注册的功能都整合在一起了，但我们真的要我们内部小组的人来为每个选手填写注册信息吗？难道我们不能让这些选手自己来进行注册吗？这样当然最好了。但把程序开放给外部用户前，需要另外考虑到一些问题。

- 公众与管理员的差别——管理员（内部用户）应有权限访问至此为止我们所建立的全部功能。我们想允许公众（非管理员）查找赛跑信息并自己去注册比赛（假如这场比赛的注册人数还没满），而他们的权限最多也就只能访问到这一步了。我们需要保护程序的管理功能，只允许内部用户使用。
- 认证——为了能让不同的用户访问不同级别的功能，我们需要一种方法去区分管理员和非管理员，这当然需要用到一个认证机制来允许管理员提交他们身份的凭证。
- 外观——至今我们所用到的那些 Grails 模板对于内部应用来说已经相当不错了（对于管理功能来说也是这样），但我们想给程序的公共部分提供更加个性化的外观。

在了解客户的需求后，我们草拟了程序公共部分的一个基本流程。

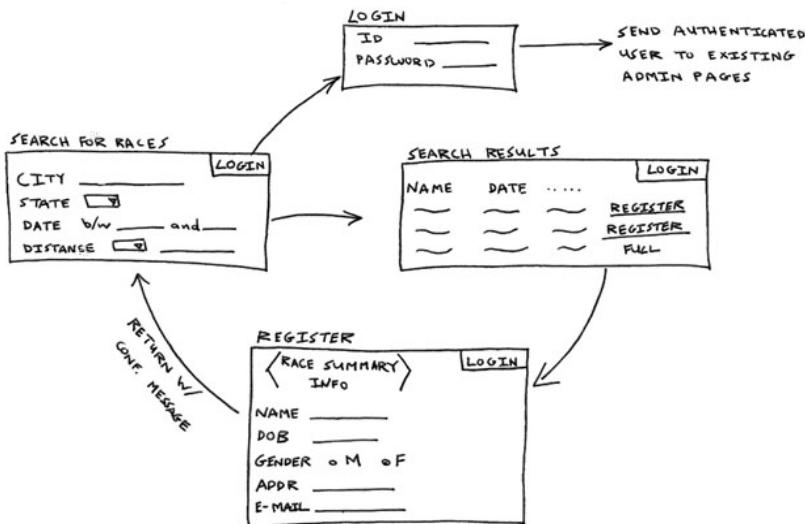


图 6-1：公共页面的用户界面流程

除增删查改功能外

修改入口页面

当用户访问根URL (<http://localhost:8080/racetrack>) 时，我们想用比赛查询页面作为新的默认入口页面（我们不希望让公众看到当前只是简单地提示用户去选择控制器的入口页面）。所有到根URL的请求都是由racetrack/webapp/index.jsp处理的。当你用编辑器打开这个文件，就会看到用于生成当前入口页面的内容。用以下这行来替换当前的内容。

```
<% response.sendRedirect(request.getContextPath() +"  
/race/search"); %>
```

如你所见，现在index.jsp将会重定向到比赛查询页面 <http://localhost:8080/racetrack/race/search>。

添加对公众友好的视图

接着我们需要为公众增加一些页面，让他们通过这些页面去查

询符合条件的比赛并进行注册。因为现有的页面各自包含着不同的管理功能，所以我们不能简单地利用它们。例如，当前显示比赛列表的页面也同样允许用户去创建一场新的比赛，但我们只想管理员拥有这样的访问权限，公众只可以访问比赛的只读视图而没有权限去选择新增、更新、或者删除任何记录。

我们也许可以让当前的比赛列表页面足够聪明地去区分出管理员和非管理员，但这个假设的基础是我们的管理员视图和公众视图总是很相似。这可能不是一个稳妥的假设，所以还是让我们动手添加一个新模板，racetrack/grails-app/views/race/searchresults.gsp，来给这些数据提供一个公众视图。

```

<html>
<head>
<meta http-equiv="Content-Type"
      content="text/html; charset=UTF-8" />
<meta name="layout" content="main" />
<title>Races Meeting Your Criteria</title>
</head>
<body>
<div class="body">
    <h1>Search Results</h1>
    <h2>
        <g:if test="${raceList?.size() == 1}">
            1 Race Meets Your Criteria:
        </g:if>
        <g:else>
            ${raceList?.size()} Races Meet Your
            Criteria:
        </g:else>
    </h2>
    <g:if test="${flash.message}">
        <div class="message">${flash.message}</div>
    </g:if>
    <div class="dialog">
        <table>
            <tr>
                <th>Name</th>
                <th nowrap>Start Date</th>
                <th nowrap>Start Time</th>
                <th>City</th>
                <th>State</th>
                <th>Distance</th>
                <th>Cost</th>
                <th></th>
            <tr>

```

```

        </tr>
        <g:each in="${raceList}">
        <tr>
            <td nowrap>
                <span class="lineItemValue">
                    ${it.name}
                </span>
            </td>
            <td nowrap>
                <span class="lineItemValue">
                    <g:formatDate
                        date="${it.startTime}"
                        format="EEE, MMM d, yyyy"/>
                </span>
            </td>
            <td nowrap>
                <span class="lineItemValue">
                    <g:formatDate
                        date="${it.startTime}"
                        format="h:mm a z"/>,
                </span>
            </td>
            <td nowrap>
                <span class="lineItemValue">
                    ${it.city}
                </span>
            </td>
            <td nowrap>
                <span class="lineItemValue">
                    ${it.state}
                </span>
            </td>
            <td nowrap class="numericData">
                <span class="lineItemValue">
                    <g:formatNumber
                        number="${it.distance}"
                        format="#0.0 mi"/>
                </span>
            </td>
            <td nowrap class="numericData">
                <span class="lineItemValue">
                    <g:formatNumber number="${it.cost}"
                        format="\$0.00"/>
                </span>
            </td>
            <td class="actionButtons">
                <g:if test="${it.registrations?.size() <
                    it.maxRunners}">
                    <span class="actionButton">
                        <g:link controller="registration"
                            action="register"
                            id="${it.id}">

```

```

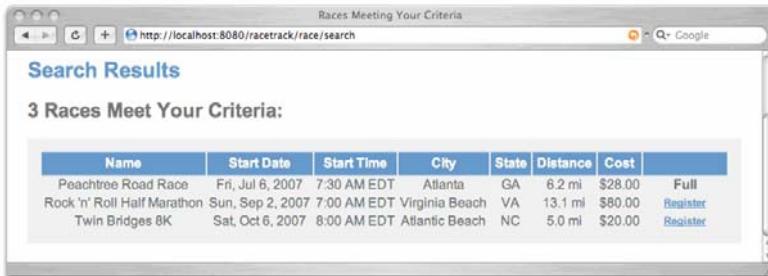
        Register
        </g:link>
        </span>
    </g:if>
    <g:else>
        <strong>Full<strong>
    </g:else>
    </td>
</tr>
<g:each>
</table>
</div>
</div>
</body>
</html>

```

现在我们需要去修改 `RaceController.groovy`，让它用新模板替换掉 `list.gsp` 模板。在 `search` Action 中，只需简单地把视图的名字由 `list` 改为 `searchresults` 即可。

```
render(view: 'searchresults', model:[ raceList: results ])
```

现在我们已经定义了一个新的视图，并且也已经更新了控制器，所以我们准备试一下刚刚所做的修改。



现在我们完成了给公众使用的合理的只读视图。顺便地，最右边的列现在也标明了用户是否仍然能进行相应比赛的注册。

当讲到注册比赛时或许你已经注意到，我们让注册链接（在 `searchresults.gsp` 中）把请求发送到 `RegistrationController` 的 `register` Action。

```
<g:link controller="registration"
    action="register" id="${it.id}">Register
</g:link>
```

我们现在还没有 `register` Action，所以让我们在 `RegistrationController.groovy` 中定义一个。

```
def register = {
    def registration = new Registration()
    registration.properties = params

    if (request.method == 'GET') {
        def race = Race.get(params.id)
        return ['registration':registration, 'race':race]
    }
    else {
        if(registration.save()) {
            flash.message =
                "Successfully registered for
                ${registration.race.name}"
            redirect(controller:'race',action:'search')
        }
        else {
            def race = Race.get(params['race.id'])
            return
                ['registration':registration, 'race':race]
        }
    }
}
```

与我们在 `RaceController.groovy` 里处理 `search` Action 的方式相似，`register` Action 会根据请求的类型来判断如何对请求进行处理。

对于 `get` 类型的请求来说，`Action` 会简单地显示当前比赛的注册页面。模板会获取相应的 `Registration` 对象和 `Race` 对象来生成页面。

对于 `post` 类型的请求来说，`Action` 会尝试去持久化 `Registration` 对象（用请求的参数构造出来）。如果成功地通过检验并保存了注册信息，`Action` 会把用户重新引导回查询页面，同时显示友好的确认信息。否则，`Action` 会重新显示注册输入页面并提醒用户哪些问题导致了注册失败。

如我们所知，退出 register Action 后，按照规约 Grails 会在 racetrack/grailsapp/views/registration/register.gsp 这个位置查找相应的模板。因为我们实际上等于是新建注册，所以这个新模板的大部分结构都可以照抄 create.gsp。由于我们想让非管理用户使用这个页面，因此我们需要相应地定制这个模板。

```

<html>
<head>
    <meta http-equiv="Content-Type"
          content="text/html; charset=UTF-8" />
    <meta name="layout" content="main" />
    <title>Register</title>
</head>
<body>
<div class="body">
    <h1>Register for ${race.name}</h1>
    <em>Start Date:
        <g:formatDate date="${race.startDate}" 
                      format="EEE, MMM d, YYYY"/>
        <br />
    </em>
    <em>Start Time:
        <g:formatDate date="${race.startTime}" 
                      format="h:mm a z"/>
        <br />
    </em>
    <g:if test="${flash.message}">
        <div class="message">${flash.message}</div>
    </g:if>
    <g:hasErrors bean="${registration}">
        <div class="errors">
            <g:renderErrors bean="${registration}" 
                           as="list" />
        </div>
    </g:hasErrors>
    <g:form action="register" method="post">
        <g:hiddenField name="race.id"
                       value="${race.id}" />
        <div class="dialog">
            <table>
                <tr class='prop'>
                    <td valign='top' class='name'>
                        <label for='name'>Name:</label>
                    </td>
                    <td valign='top' 
                        class='value'
                        ${hasErrors(bean:registration,
                                    field:'name', 'errors')}>

```

```

<input type="text" maxlength='50'
       name='name'
       value='${registration?.name}'>
</input>
</td>
</tr>
<tr class='prop'>
<td valign='top' class='name'>
    <label for='dateOfBirth'>
        Date Of Birth:
    </label>
</td>
<td valign='top'
     class='value'
     ${hasErrors(bean:registration,
                field:'dateOfBirth','errors')}>
    <g:DatePicker name='dateOfBirth'
                  value='${registration?.dateOfBirth}'
                  precision='day'>
    </g:DatePicker>
</td>
</tr>
<tr class='prop'>
<td valign='top' class='name'>
    <label for='gender'>Gender:</label>
</td>
<td valign='top'
     class='value'
     ${hasErrors(bean:registration,
                field:'gender','errors')}>
    <g:radio name='gender' value='M'
checked='${registration?.gender?.equals("M")}'>
    </g:radio>
    Male
    <g:radio name='gender' value='F'
checked='${registration?.gender?.equals("F")}'>
    </g:radio>
    Female
</td>
</tr>
<tr class='prop'>
<td valign='top' class='name'>
    <label for='postalAddress'>
        Postal Address:
    </label>
</td>
<td valign='top'
     class='value'
     ${hasErrors(bean:registration,
                field:'postalAddress','errors')}>
    <textarea rows='3' cols='30'
name='postalAddress'>

```

```
        ${registration?.postalAddress}
    </textarea>
    </td>
</tr>
<tr class='prop'>
    <td valign='top' class='name'>
        <label for='emailAddress'>Email
Address:</label>
    </td>
    <td valign='top'
        class='value'
        ${hasErrors(bean:registration,
            field:'emailAddress','errors')}>
        <input type="text" maxlength='50'
            name='emailAddress'
            value='${registration?.emailAddress}'>
    </td>
</tr>
</table>
</div>
<div class="buttons">
    <input type="submit" value="Register"
        class="formbutton">
    </input>
    <input type="button" value="Cancel"
        onClick="history.back()"
        class="formbutton">
    </input>
</div>
</g:form>
</div>
</body>
</html>
```

现在当用户注册比赛时，注册页面会包含比赛名称、日期和时间，但会隐藏另外的增删查改这些内部用户才能进行操作的管理功能。

The screenshot shows a web browser window with the title 'Register' at the top. The URL in the address bar is <http://localhost:8080/racetrack/registration/register/3>. The main content is a form titled 'Register for Rock 'n' Roll Half Marathon'. It includes fields for Name (Jane Doe), Date Of Birth (1 February 1975), Gender (Male), Postal Address (1234 Rocky Road, Somewhere, NC 12345), Email Address (jane@doe.com), and two buttons at the bottom: 'Register' and 'Cancel'.

假如用户输入的信息有效，程序就会显示确认注册信息，那用户就可以束紧鞋带开始准备训练了。

The screenshot shows a web browser window with the title 'Search for Races' at the top. The URL in the address bar is <http://localhost:8080/racetrack/race/search>. The main content is a search form with a success message: 'Successfully registered for Rock 'n' Roll Half Marathon'. The search form includes fields for City, State, Date (between October 27, 2006 and October 26, 2008), Distance (At least [] mi), and a 'Search' button.

实现用户认证

现在我们已经拥有了想给公众使用的所有页面，但是仅仅相信

他们会自觉远离管理区域当然是不够的。是的，我们需要在适当的地方设置些安全措施来限制那些对非公共功能的访问。为此，我们首先需要一个办法来让管理用户向应用程序证明他们本人的身份。一旦通过验证，他们就能够访问系统中受保护的区域。

管理用户帐户

既然我们需要维护已授权用户以及他们的凭证信息，那为什么不定义一个新的 Grails 领域类来维护这些数据呢？用我们先前演示的创建 `Race` 类和 `Registration` 类的方法，我们来新增一个带有以下属性和约束属性的 `User` 领域类。

```
class User {
    String userId
    String password

    static constraints = {
        userId(length:6..8,unique:true)
        password(length:6..8)
    }
}
```

小提示：作为一个安全可靠的应用产品，我们不应该使用明文密码。我们应该选择一个单向加密算法（如SHA²⁶）以便得到密码的散列值。这里我们将不作详细的讨论，但你可以看看Java的 `MessageDigest`²⁷类，以获取更多关于如何在Java（和Grails）程序中使用安全的单向散列功能的信息。

既然我们已经有了适当的领域类，那么可以用 `grails generate-all` 命令生成界面的脚手架以管理用户数据。

在我们把管理功能封闭起来之前，首先需要为程序添加至少一位管理员。虽然我们可以通过用户界面去添加第一个管理员，但有另一种方法更为便利。

Grails提供了一种Bootstrap机制，这种机制能够执行任何你想在程序启动时执行的初始化任务²⁸。按照规约，Grails在 `racetrack/grails-app/conf` 目录里查找名字类似于 `FooBootstrap.groovy`、`BarBootstrap.groovy` 等的类。Grails期望这些类包含一个初始化闭包(`init`)在系统启动时被调

用，和一个销毁闭包（`destroy`）在应用关闭时调用。（作为一般原则，你不应该依赖于销毁闭包去执行任何至关重要的工作，因为如果某些原因导致系统非正常终结，那销毁闭包就不一定会被运行到）。

Grails 默认提供了一个空的 `Bootstrap` 类给我们看是否合用。我们
们 就 用 这 个 类
(`racetrack/grailsapp/conf/ApplicationBootStrap.groovy`) 来添加第一个管理员。

```
class ApplicationBootStrap {
    def init = { servletContext ->
        final String BACKUP_ADMIN = 'adminjoe'
        if (!User.findById(BACKUP_ADMIN)) {
            new User(userId:BACKUP_ADMIN,
                     password:'password').save()
        }
    }

    def destroy = {
    }
}
```

因为每次程序启动时都会运行这个闭包，所以我们可以确信在程序启动的时候总是能有至少一个管理员。这样我们就不至于没法登录（当然，我们不希望存在重复的用户实体，所以我们的代码会在添加用户前检查用户是否已存在）。

拦截与认证

如果我们希望管理员能登录到系统中，显然我们需要给他们一个登录的地方。让我们添加一个新模板 (`racetrack/grails-app/views/user/login.gsp`) 来作为认证页面。

```
<html>
<head>
<meta http-equiv="Content-Type"
      content="text/html; charset=UTF-8" />
<meta name="layout" content="main" />
<title>Log in</title>
</head>
<body>
<div class="body">
    <h1>Please log in</h1>
```

```

<g:if test="${flash.message}">
    <div class="message">${flash.message}</div>
</g:if>
<g:hasErrors bean="${race}">
    <div class="errors">
        <g:renderErrors bean="${user}" as="list" />
    </div>
</g:hasErrors>
<g:form controller="user" method="post" >
    <div class="dialog">
        <table>
            <tr class='prop'>
                <td valign='top' class='name'>
                    <label for='userId'>User ID:</label>
                </td>
                <td valign='top' class='value'>
                    <input type="text" maxlength='8'
                           name='userId'
                           value='${user?.userId}'>
                </input>
            </td>
        </tr>
        <tr class='prop'>
            <td valign='top' class='name'>
                <label for='password'>
                    Password:
                </label>
            </td>
            <td valign='top' class='value'>
                <input type="password" maxlength='8'
                       name='password'
                       value='${user?.password}'>
            </input>
        </td>
    </tr>
</table>
</div>

<div class="buttons">
    <span class="button">
        <g:actionSubmit value="Log in" />
    </span>
</div>
</g:form>
</div>
</body>
</html>

```

接下来，我们需要一些 Action 去负责渲染登录页面和处理登录页面发送过来的请求。前面我们已经试过在一个 Action 中处理不同

类型的请求，所以在这我们也会继续使用这种模式。让我们来添加下列 Action 到 UserController 类中。

```
def login = {
    if (request.method == "GET") {
        session.userId = null
        def user = new User()
    }
    else {
        def user =
User.findByIdAndPassword(params.userId,
                           params.password)
        if (user) {
            session.userId = user.userId
            redirect(controller:'race')
        }
        else {
            flash['message'] =
'Please enter a valid user ID and
password'
        }
    }
}
```

我们的程序会使用一种简单（但有效）的方法去识别一个已登录的用户。当一个用户成功登录后，我们会把用户的 ID 保存在 Session 中（在一个叫 `userId` 的参数里）。当用户登出时，就把用户 ID 从 Session 中移除。所以在任何时候，程序都能通过查找在 Session 中的用户 ID 来判定一个用户是否已经登录。

当 `login` Action 接收到一个 `get` 类型的请求时，这表明用户正在尝试登录。万一用户因为某些原因已经登录，或许用户正在尝试以另一个的身份登录，所以我们会首先确定 `session.userId` 是否为 `null`，然后才会去呈现登录页面。

当用户提交他的用户 ID 和密码时，`login` Action 会接收到一个 `post` 类型的请求，而我们需要验证用户的输入信息。我们通过 ID 和密码来查找用户，如果这个用户是存在的话，我们会把用户 ID 加入到 Session 中，并把用户转移到比赛列表页面。如果通过 ID 和密码找不到用户，我们则把用户返回到登录页面，并且善意地请他们停止尝试入侵我们的应用程序。

在结束 `UserController` 类之前，我们需要给用户一个途径去登出程序。让我们来添加一个 `logout` Action 来移除 Session 中的用户 ID，把用户返回到应用程序的公共部分，并给出友好的登出确认信息。

```
def logout = {
    session.userId = null
    flash['message'] = 'Successfully logged out'
    redirect(controller:'race', action:'search')
}
```

添加拦截器

现在我们需要找一种方法来拦截任何进入应用管理区域的访问，同时迫使用户登录后才能继续访问这些区域。简而言之，除了不久之前实现的一些公用模块外，我们要限制对所有其它功能的访问。Grails的Action拦截器为我们实现这样的功能提供了便利的机制²⁹。

Action 拦截器会截断正常的 Action 执行流程，并允许我们在 Action 执行之前或之后执行一些其它功能。出于安全性方面的考虑，在一个用户进行管理活动之前，我们首先要确定该用户是否管理员。因此，在执行任何与管理相关的 Action 之前，我们需要一个 `beforeInterceptor` 去调用我们的认证模块。

假设我们定义了一个名叫 `auth` 的方法，用来判定用户是否已经登录程序。在 `RaceController` 类中，我们想在除了 `search` Action 之外的所有 Action 调用之前都执行这个方法（因为我们想允许公众访问 `search` Action）。我们只需简单地作如下的声明即可实现该功能。

```
class RaceController {
    def beforeInterceptor = [action:this.&auth,
                           except:['search']]
    //...
}
```

这个声明告诉 Grails，“除了 `search` 以外的所有 Action，在把控制权交到 Action 之前都需要调用 `auth` 方法”。当然，我们仍需要实现 `auth` 方法，但让我们晚一点再去对付它。首先，我们为

其它两个控制器添加必要的拦截器。

在 `RegistrationController.groovy` 中，公众只有权限访问 `register` Action。用户在访问任何其它 Action 前都必须进行验证。

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

    //...
}
```

在 `UserController` 类中，我们想限制除了 `login` 和 `logout` Action 之外的所有访问。

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

    //...
}
```

我们也应该注意到，前面的拦截器都是用 `except` 来指定我们不想限制的 Action。除了这种方法，Grail 也允许我们用 `only` 来排除掉其它我们不想限制的 Action。例如……

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

引入 `BaseController`

现在我们所有的控制器都依赖于还没实现的 `auth` 方法，但我们显然不打算在每个控制器中都重复实现它。因此，是时候为我们的控制器引入一个抽象父类了，我们可以在这个父类中定义 `auth` 方法。我们叫它 `BaseController`，把它放到 `racetrack/grails-app/controllers/BaseController.groovy`。

```
abstract class BaseController {
    def auth() {
```

```

        if(!session.userId) {
            redirect(controller:'user',action:'login')
            return false
        }
    }
}

```

在整个应用的范围内，我们知道能够通过查找 Session 里的用户 ID 来判断已登录用户。auth 方法依赖于这个规则来决定把请求引导到哪里。

如果用户 ID 不在 Session 内，那么我们就知道用户还没有登录。我们会把用户重定向到登录页面并且返回 `false`。回想一下这个方法，它是在控制权转移到请求的 Action 之前在拦截器里被调用的。通过返回 `false`，我们告知 Grails 我们已经覆盖了 Action 执行的正常流程，所以 Grails 不应该继续执行原先请求的 Action 了。

另一方面，如果在 Session 里找到了用户 ID，那么我们就知道用户已登录，我们会简单地退出 auth 方法。Grails 会继续接着执行被拦截了的 Action。

既然已经有了 `BaseController` 类，那我们接着更新现有的控制器去继承它的功能。

```

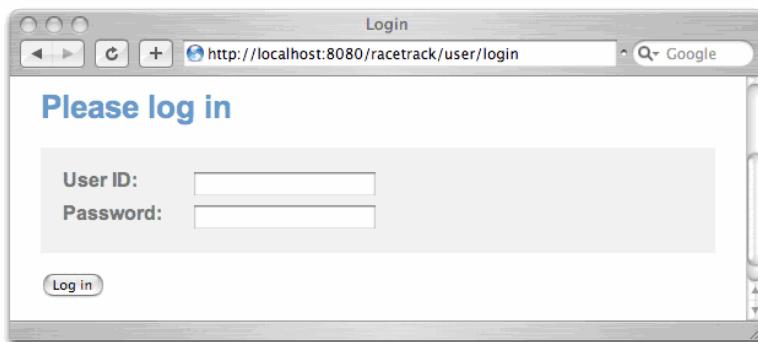
class RaceController extends BaseController {
//...
}

class RegistrationController extends BaseController {
//...
}

class UserController extends BaseController {
//...
}

```

哇！这次变动有点大，不先做一点中间测试可能有些不稳妥，那我们就动手吧。启动程序并试着偷偷进入系统中我们刚刚加上保护的区域。我们知道只有管理员才能够新增比赛，所以我们要留意一下新的安全措施能否把一般公众拒之门外。当我们尝试访问 <http://localhost:8080/racetrack/race/create> 时，会看到什么？



太好了。当一个未认证的用户尝试去访问网站的管理区域，程序会让用户先去登录。这正是我们想要的功能。

在我们继续之前，再花几分钟来检验一下我们的新安全系统。保证我们在没登录的情况下仍可访问应用的公共区域。检查我们能否成功登出。确保输入错误密码时不能进入。

完成了吗？好的，安全模块应该没问题了，但现在的用户体验会不会有点怪呢？你有没有发觉在你登录后程序总会把你带到比赛列表页面，不管你之前尝试访问的是哪个页面？总体上的导航性又如何呢？现在并没有一种友好的方式可以让登录用户进入到比赛管理、用户管理等区域。其实这方面我们应该能够做得更好一些。

改进我们程序的记忆功能

当一个用户还没登录就尝试去访问程序的受限区域，我们就会把他引导到登录页面。这种一直以来的做法考虑得还不够周到。我们完全忘记了用户正在尝试访问的是哪个页面。事实上我们应该设立一个用户请求的快照，一旦用户登录成功，才能够继续原先的请求。

在 `auth` 方法里面（在 `BaseController.groovy`），添加以下的新代码，把请求先保存起来，直到我们准备好去执行它为止。我们首先来建立一个存放控制器名和 Action 名的 Map，然后我们把请求中的所有的参数都加入到 Map 中。在把用户转到登录页面

之前，我们会把 Map 放在 Session 里保管好。

```
def auth() {
    if (!session.userId) {
        def originalRequestParams =
            [controller:controllerName,
             action:actionName]

        originalRequestParams.putAll(params)

        session.originalRequestParams =
            originalRequestParams

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

另外一个需要注意的是 UserController 类里的 login Action。用户一旦登录后，我们就取回原先请求中的信息并继续执行它（作为防范措施，如果因某些原因而导致不能找回原先请求的话，那我们就把用户转到比赛列表页面）。

```
def login = {
    if (request.method == 'GET') {
        //...
    }
    else {
        def user =
        User.findByUserIdAndPassword(params.userId,
                                     params.password)
        if (user) {
            session.userId = user.userId

            def redirectParams =
                session.originalRequestParams ?
                session.originalRequestParams :
                [controller:'race']
            redirect(redirectParams)
        }
        //...
    }
}
```

花点时间重启程序来看看这些改进。现在当用户尝试去访问受保护资源（如<http://localhost:8080/racetrack/race/create>），只要登录成功，我们就会转送他到原先请求的资源。

请问我能看看菜单吗？

我们所提供的功能是很简单的，所以他们可以不需要任何菜单。他们只是查询一下比赛，然后从列表中选取比赛来注册。然而，我们的管理员就有好几种选择了。假如我们想做一个所有页面都能共用的菜单的话，我们要如何避免为每个模板都添加各自的菜单呢？Grails 允许我们定义子模板，子模板只渲染页面的一部分，我们可以把它嵌入到其它模板视图中。因为我们可以随心所欲地重用这些子模板，所以用一个子模板来实现我们的新菜单应该没问题。

我们靠什么来区分子模板和标准模板？你猜对了，是规约。假设我们把菜单子模板命名为 `adminmenubar`，那么按照规约，我们需要在名叫 `_adminmenubar.gsp` 的文件里定义这个模板。在决定这个文件应该放在哪里之前，我们首先需要考虑有哪些模板要包含这个子模板。如果我们只是想在跟比赛相关的视图中使用这个子模板的话，我们就应该把子模板放到 `racetrack/grails-app/views/race` 里。然而，我们想在所有视图里都用到它，所以我们需要把这个子模板放到视图根目录 `racetrack/grailsapp/views` 下。

在子模板里，我们会根据用户是否已经登录来显示相应的菜单选项。

```

<g:if test="${!session.userId}">
    <span class="menuButton">
        <g:link controller="user" action="login">
            Log in
        </g:link>
    </span>
</g:if>
<g:else>
    <span class="menuButton">
        <g:link controller="race" action="list">
            Manage Races & Registrations
        </g:link>
    </span>
    <span class="menuButton">
        <g:link controller="user" action="list">
            Manage Administrators
        </g:link>
    </span>
</g:else>

```

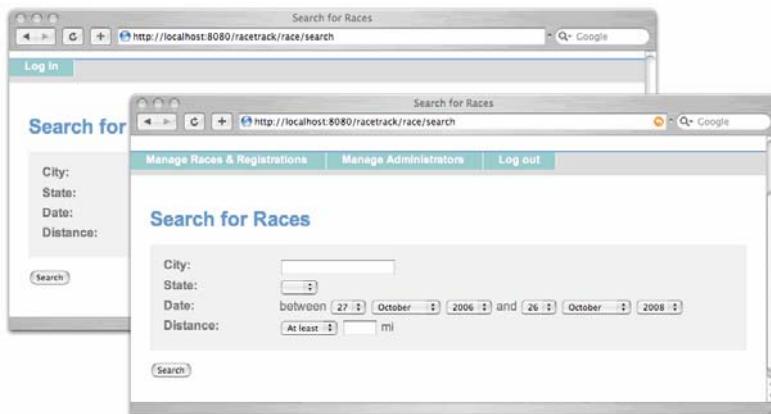
```
<span class="menuButton">
  <g:link controller="user" action="logout">
    Log out
  </g:link>
</span>
</g:else>
```

现在，为了在适当的页面里添加这个菜单，我们只需简单地在页面中引用这个子模板。我们从程序的入口比赛查询页面开始。添加如下的内容到 `racetrack/grails-app/views/race/search.gsp`。`<g:render>` 标签会处理子模板的内容，并把相应地结果包含到当前页面中。

```
<html>
  ...
  <body>
    <div class="nav">
      <g:render template="/adminmenubar" />
    </div>
  ...
</body>
</html>
```

值得注意的是，用 `<g:render>` 标签引用子模板的时候我们并没有写出前面的下划线。因为这个标签是专门用来渲染子模板的，所以在查找 GSP 文件时它会自动补上下划线。

同样需要注意的是，我们在模板名字前加上了一个正斜杠。正斜杠是让 Grails 去根视图目录下查找模板。如果没有斜杠的话，Grails 就会在引用者所在目录下找子模板。



现在我们在程序的其它页面里也加上菜单。我们只定义了菜单一次，其它页面都只是引用它（使用`<g:render>`标签），这样我们就把对菜单的修改全都控制在了一个地方。当我们修改子模板时，所有包含该子模板的页面也都同样会自动反映出我们所做的修改。

你也许会问我们干嘛一定要去更新所有与子模板关联的页面那么麻烦呢？这问题问得很好。尽管我们已经不用在每个模板中分别加入生成菜单的代码细节，但这仍不是一个完美的解决方案。实际上，如果我们是白手起家构建程序（或者我们想在自动生成的脚手架视图上做大幅度的修改），那么我们只需把菜单放置在主布局`racetrack/grails-app/views/layouts/main.gsp`里。主布局里已经包含了一些我们现在在每个页面上都看得到的公用组件（例如 Grails 标志、样式等），所以我们也能轻易地把菜单也包含到主布局里。这种方式我们将会下一节中详细介绍。

界面美化：布局与 CSS

当话题开始转移到谈论程序的外观时，这通常来说都是个好现象。这表示我们已经做好了开发者的本分，同时网站的功能也已足够完善，客户开始去注意美工了。所以，让我们比 Grails 的默认外观更进一步，给终端用户一个更为用户化的界面。

界面的美化包括了三个主要步骤。首先，我们会为网站的公共部分定制一个外观样式表。然后，我们会为这个新外观添加一些合适的图片到项目中。最后，我们会为公共页面单独定义一个布局模板。你准备好了吗？启程吧。

首先，让我们把新的样式表添加到程序中。Grails会在 `racetrack/web-app/css` 中查找样式表，所以我们需要把新样式表 `public.css` 放在这个目录里（为了简洁，样式表的内容就不写在这书里了，但是完整的样式表，以及所有本书中例子的完整源代码，你都可以下载到³）。

我们的新样式表用到 4 幅图片，`banner.jpg`、`bg.gif`、`formbg.gif` 和 `transrace.png`。所以下一步我们需要把这些图片拷贝到 `racetrack/web-app/images` 目录中。

```
racetrack> ls -1 web-app/images/
banner.jpg
bg.gif
formbg.gif
grails_logo.jpg
spinner.gif
transrace.png
tree
```

既然用到的文件都已经各就各位，那接着让我们花点时间来讨论一下布局。你有没有注意到在这个程序中我们曾做过的模板都没有提及过样式表？你又有没有注意到所有页面的模板都没有引用过 Grails 的标志，我们却在每个页面上都看到它？如果模板里都没有包含这些信息，那为什么我们又能够清晰地在浏览器里看到页面上的 Grails 标志和样式表的效果呢？答案就在于包含在每个模板中的一个标签里。

```
...
<head>
  <meta http-equiv="Content-Type"
        content="text/html; charset=UTF-8" />
  <meta name="layout" content="main" />
  <title>Some Title</title>
</head>
...
```

这个标签告诉Grails在渲染模板时使用一个叫 `main` 的布局。于

是Grails就会在应用程序的全局布局目录`racetrack/grails-app/views/`中查找布局模板`main.gsp`。我们通过布局来定义应用于整个网站（或者一部分页面）的总体结构形式，然后在打算使用该布局的模板里引用它（既可明确地用`meta`标签来指明，也可按照规约来隐含地确定³⁰）。模板只需要关注它自身的内容细节（例如，一个比赛的列表），让布局去提供页面的结构设计（例如标志、公共的菜单和链接等）和一些供多模板共用的项目（如样式表，公共的JavaScript函数等）。

采用布局不仅符合我们的不重复开发原则，而且布局也通过提供一站式的服务出色地重新调整了页面的结构。举个例子说，假设目前在我们应用程序中菜单是水平排列在页面的顶端。我们后来又决定菜单应该垂直放置在页面左边。如果如何展现菜单的逻辑是定义在布局里（而不是分别嵌入到各模板中），那么我们只要改变这个布局就会马上看到所有使用了该布局的页面也相应地发生改变。

现在我们的应用程序中所有的页面都在用 Grails 的默认布局模板。让我们来为网站的公共部分定义一个新的自定义布局。新增一个 布 局 模 板 `racetrack/grails-app/views/layouts/public.gsp`，并且在其中添加如下内容。

```

<html>
<head>
    <title>
        <g:layoutTitle default="Racetrack" />
    </title>
    <link rel="stylesheet"
        href="\${createLinkTo(dir:'css',file:'public.css')}">
        </link>
        <g:layoutHead />
</head>
<body>
    <table class="contentArea">
        <tr>
            <td>
                <div class="logo"></div>
                <g:layoutBody />
            </td>
        </tr>
    </table>
</body>
</html>

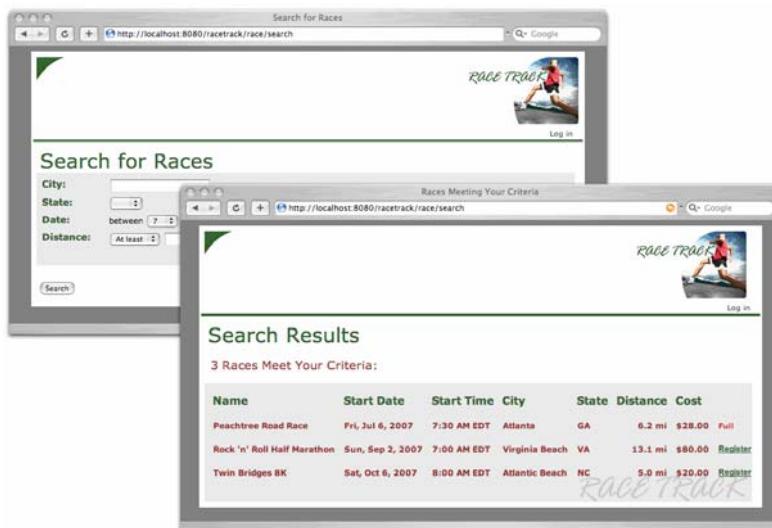
```

我们可以看到这个布局引用到了新的样式表。同时，仔细看看 `<g:layoutTitle>`、`<g:layoutHead>` 和 `<g:layoutBody>` 标签。这些标签的作用是作为实际页面内容的占位符。在运行时，这些标签会从使用布局的模板中获取相应的内容。举例说，当我们让 `search.gsp` 改用这个布局时，在运行时 Grails 会渲染 `search.gsp` 里的 `<body>` 标签的内容来代替在布局模板里的 `<g:layoutBody>` 标签（这些标签的工作方式与 SiteMesh 的 `decorator` 标签非常相似。实际上在背后，Grails 就是通过委托 SiteMesh 来完成这个工作的）。

既然现在我们已经有了新布局，那最后一步就是去更新公共页面，替换掉默认的布局。在 `search.gsp`、`searchresults.gsp` 和 `register.gsp` 中，找到定义布局的标签，把它改为引用 `public.gsp`。

```
<meta name="layout" content="public" />
```

做完这一步，我们就准备揭晓程序的新形象了。



这样我们就完成了，朋友们。我们开发了一个令客户满意的应

用程序，他们也准备让我们把产品上线了！（在我们继续之前，你或许会注意到，其实我们有机会进一步实践“不重复”的原则。如果我们把三个公共页面中的 `adminmenubar` 去除，把它移到布局模板里面，那我们就又进一步减少了页面里的重复内容，这样就更加完美了。你能自己完成这个工作吧？）

测试

如果要快速开发应用的话就没时间去做测试，对不？错！你使用一个系统是因为它开发得快吗？当然不是了。是因为系统能正常工作而且可以提高我们的工作效率，我们才被它吸引的。让那些企图强迫我们使用错漏百出的软件的可怜灵魂们天天被查税天天牙痛吧。让我们也鄙视一下自己过了这么久才讲到测试（好的，你只是一个无辜的旁观者，而我会接受这次谴责^_^）。

单元测试

Grails 的内置支持给单元测试提供了便利。如果这样我们还不去做测试的话就有点内疚了。看一下 `racetrack/grails-tests`，Grails 已为每一个领域模型生成了一个相应的测试类。但当我们认真看一下其中一个类时羞愧感便油然而生，因为到现在为止我们的测试覆盖率还是 0%。

```
class RaceTests extends GroovyTestCase {  
  
    void testSomething() {  
    }  
}
```

现在该是时候出手去改变这种状况了（如果你是写Groovy单元测试³¹方面的新手的话，进来接受款待吧。但注意了，一旦你开始了这次尝试，也许就永远不想回头了）。首先决定我们需要测试什么。Race领域类的所有单元测试都放在RaceTests类里面（我们马上就会讨论到控制器的测试该放哪里）。虽然Race类很直接明了，但它毕竟不单是包含了简单的属性。回想一下，我们除了为这个类添加了几个约束属性，还自定义了一个约束属性。

```

static constraints = {
    name(maxLength:50,blank:false)
    startTime.validator: {return (it > new
Date())})
    city(maxLength:30,blank:false)
    state(inList:["GA", "NC", "SC", "VA"],blank:false)
    distance(min:3.1f,max:100f)
    cost(min:0f,max:999.99f)
}

```

我觉得为这些约束属性添加测试用例是有必要的。即使你用看的就已经知道配置是正确的，但实际上写一些单元测试用例来验证那些烦人的边界情形还是很有意义的。例如，考虑上面自定义的那个 `startTime` 约束属性。如果因某些原因导致日期为空值时会有什么情况发生呢？（`null` 大于 `new Date()` 吗？）我们当然不想让任何比赛的开始日期/时间为 `null` 值，所以我们应该检查一下那个约束属性是否会为我们捕获空值。

```

void testSomething() {
    def race = new Race()
    race.startTime = null

    assertFalse(race.validate())

    def fieldError =
    race.errors.getFieldError('startTime')
    def validationError = fieldError.codes.find {
        it == 'race.startTime.validator.invalid'
    }
    assertNotNull(validationError)
}

```

在这个测试用例中，`Race` 对象的 `startTime` 属性为空值，并断言对象会验证失败，然后检查抛出的错误是否包含了预期的 `startTime` 属性的错误代码。接着，运行所有的测试用例，我们只需简单地在命令行窗口输入 `grails test-app`，就可以看到结果了。

```

racetrack> grails test-app
...
[echo] Running tests for environment: test
...
[java] OK (3 tests)
...

```

```
BUILD SUCCESSFUL
Total time: 11 seconds
```

喔！看到程序能够恰当地防止 `startDateTime` 为空值，我们可以松口气了。但是这个自定义检验器在其它可能发生的场景下又会如何呢？让我们把那些测试用例也包含进来吧，这就能再提高点我们的自信心了。（如果你疑惑为什么 Grails 报告了 3 个完全成功的测试，注意一下我们还有两个另人难堪的领域类的空壳测试类。里面的空白测试用例，不奇怪，当然是通过的。）

让我们添加一个测试来确保当 `startDateTime` 属性为过去的日期时会得到验证错误。并且，与确保错误数据会给我们带来验证错误同样重要的是，一个完全正确的 `Race` 对象不应该给我们带来任何错误。如果你想给其它约束属性也加上一点额外的保障，你也可以为它们编写测试用例。有了第一个测试用例的示范，我们现在能快速地为各种场景添加测试用例了。

```
class RaceTests extends GroovyTestCase {

    void
    testStartTimeCustomConstraintWithNullValue() {
        def race = getValidRace()
        race.startTime = null
        assertValidationErrors(race, 'startTime',
            'race.startTime.validator.invalid')
    }

    void
    testStartTimeCustomConstraintWithPastValue() {
        def race = getValidRace()
        race.startTime = new Date(0)
        assertValidationErrors(race, 'startTime',
            'race.startTime.validator.invalid')
    }

    void testNameMaxConstraint() {
        def race = getValidRace()
        race.name = 'It may very well take longer to' +
            ' typeout the name of this race' +
            ' than to just go run it.'
        assertValidationErrors(race, 'name',
            'race.name.maxLength.exceeded')
    }

    //...
}
```

```

private Race getValidRace() {
    def race = new Race()
    race.name = 'Fast 5K'

    // 1 day in the future
    race.startDate = new Date().plus(1)
    race.city = 'Somewhere'
    race.state = 'NC'
    race.distance = 3.1
    race.cost = 20.00
    race.maxRunners = 1000

    // Make sure that we have indeed constructed a
    // valid Race object
    assertTrue(race.validate())

    return race
}

private assertValidationErrors(race, fieldName,
                               errorMessage) {
    assertFalse(race.validate())
    def fieldError =
    race.errors.getFieldError(fieldName)
    def validationError = fieldError.codes.find {
        it == errorMessage
    }
    assertNotNull(validationError)
}
}

```

在得到令人满意的测试覆盖率之后，就是时候检查一下所有约束属性是否真的都符合我们的期望，无懈可击了。

```

racetrack> grails test-app
...
[echo] Running tests for environment: test
...
[java] OK (19 tests)
...
BUILD SUCCESSFUL
Total time: 12 seconds

```

当然除了约束属性，其它功能也可以加上相应的测试用例。如果你有些棘手的关联关系，或者是其它的持久化方面的问题，那么测试用例也应该覆盖这些区域。

在测试任何与数据库相关功能之前，首先应该把测试数据源配置为你的测试数据库。Grails允许你用任何数据源来运行测试³²。

不过，你通常都希望有一个专门的数据库能让测试能够随意更改里面的数据，而不至于影响到开发或者生产所用的数据库。因此，我们需要照下面修改改 `racetrack/grailsapp/conf/TestDataSource.groovy`（注意把用户名和密码改成你的MySQL帐户的相应值）。

```
class TestDataSource {
    boolean pooling = true
    String dbCreate = "update"
    String url =
"jdbc:mysql://localhost/racetrack_test"
    String driverClassName = "com.mysql.jdbc.Driver"
    String username = "jason"
    String password = ""
}
```

也许你已经注意到了 `grails test-app` 命令的输出结果，Grails 默认使用了测试数据源。如果你想在运行测试时使用其它数据源，只需简单地在命令里加上环境名就可以了（即 `dev`、`test` 或 `prod`）。

```
racetrack> grails dev test-app
...
[echo] Running tests for environment: development
...
[java] OK (19 tests)
...
BUILD SUCCESSFUL
Total time: 10 seconds
```

功能测试

对于那些追求程序健壮性的人来说，Grails也同样支持功能测试。Canoo WebTest为功能测试提供了一个框架，你可以通过测试用例去检查程序作出的各种响应，这些响应与终端用户的体验是一样的。测试用例可以导航到不同的URL上，点击按钮，顺着链接爬行，改变页面内容等等。如果你有兴趣把功能测试加到程序里（或者仅仅想看看它们是如何工作的），在线文档里包括了一份指南，它会教你整个功能测试的流程，教你如何定制所需的测试，如何运行测试，以及如何查看测试报告结果³³。

终点线

到目前为止，本轮的开发已经完成。程序已经在运转着，客户对他们所看到的也挺喜欢；然而，客户并不打算在开发环境上运行系统，我们还有一些工作需要完成。让我们把应用程序部署到生产环境，那样就算完工了。

日志

一旦应用程序上了生产环境，我们希望可以通过一定手段来观察系统活动的方方面面。我们可能想要收集关于用户如何使用这个系统的统计数据，记录相关的数据以帮助可能的调试等等。Grails 内置了日志系统——采用 log4j——来帮助我们完成这些需求。

每个Grails控制器都包含一个动态的log方法，该方法提供了标准的 log4j 功能³⁴。每个应用系统也都包含一个 log4j.properties 文件，用于定制日志输出。

记录任何未授权用户的登录企图无疑是一个好主意。现在就让我们添加一些日志操作来记录这些行为。在 UserController 类中，login 方法用于验证输入的用户 ID 和密码组合是否存在于授权用户的列表之中。一旦发现了无效的用户 ID 和密码组合，我们就记录一个包含该入侵企图细节信息的警告。

```
def login = {
    if (request.method == "GET") {
        session.userId = null
        def user = new User()
    }
    else {
        def user =
User.findByIdAndPassword(params.userId,
```

```
params.password)
    if (user) {
        session.userId = user.userId
        def redirectParams =
            session.originalRequestParams ?
                session.originalRequestParams :
                [controller:'race']
        redirect(redirectParams)
    }
    else {
        log.warn "Shields up! Somebody's trying " +
            "to hack through our rock-solid " +
            "DEFCON 1 security -- " +
            "User ID - $params.userId, " +
            "Password - $params.password"

        flash['message'] = 'Please enter a valid ' +
            'user ID and password'
    }
}
```

Grails默认只记录错误信息，而且所有的日志都输出到控制台上。但在生产环境当中，我们显然希望把这些日志输出到一个文件甚至是发送电子邮件给管理员。Log4j支持各种各样的输出方式³⁵，不过对我们来说，把记录输出到文件系统就够了。

就像为不同环境提供不同数据源一样，Grails 同样支持为不同环境分别配置日志操作。我们将为生产环境做一些定制化的配置。打开 `racetrack/web-app/WEB-INF/log4j.production.properties`，并加入如下配置：

```
log4j.appenders.access.access=org.apache.log4j.FileAppender
log4j.appenders.access.access.file=access.log
log4j.appenders.access.access.layout=org.apache.log4j.PatternLayout
log4j.appenders.access.access.layout.ConversionPattern=%d %p
  %x [%c] %m%n

log4j.logger.UserController=warn,access
log4j.additivity.UserController=false
```

配置中首先定义了输出方式是文件，输出文件名是 `access.log`（在应用程序的根目录下，但对于生产环境，我们可以指定一个专门放置日志的文件目录）；同时为

UserController 类记录下所有的警告级别（或以上）的事件。

好了，现在我们可以检验下我们的设置，输入 `grails prod run-app`（用生产环境的配置运行应用程序），然后访问 <http://localhost:8080/racetrack/user/login>，输入无效的用户名和密码，你将在 `racetrack/access.log` 看到如下输出信息：

```
2006-10-28 16:25:29,239 WARN [UserController] Shields
up! Somebody's trying to hack through our rock-solid
DEFCON 1 security -- User ID - hacker, Password -
letmein
```

好了，当一些心怀不轨的朋友试图闯入我们的应用系统，我们将很容易得到他们的行为记录。

部署

现在，让我们把项目部署到一个生产环境服务器上，我们将分三步完成。

首先，我们要让生产环境的应用程序知道如何访问相应的数据库。更新新 `racetrack/grailsapp/conf/ProductionDataSource.groovy` 中的配置信息（注意把用户名和密码换成你自己的 MySQL 账户的用户名和密码）。

```
class ProductionDataSource {
    boolean pooling = true
    String dbCreate = "update"
    String url =
"jdbc:mysql://localhost/racetrack_prod"
    String driverClassName = "com.mysql.jdbc.Driver"
    String username = "prod"
    String password = "wahoowa"
}
```

一旦为应用系统配置好了生产环境的数据库，我们就可以将它打成 WAR 包。我们只需输入 `grails war`，Grails 会完成剩下的工作。

```
racetrack> grails war
...
```

```

war:
  [echo] Packaging for environment 'production'
...
BUILD SUCCESSFUL
Total time: 6 seconds
racetrack> ls -l racetrack.war
-rw-r--r-- 1 jason jason 11760541 Oct 28 17:02
racetrack.war

```

注意，上面的输出显示了应用系统是为生产环境打包的。这正是当下我们期望的。但是有时我们可能想要打一个WAR包部署在开发或者测试环境上。默认情况下，grails war命令使用的是生产环境的数据库配置信息（ProductionDataSource.groovy），然而，我们也可以简单地在命令中输入目标环境（例如grails dev war），用指定环境的配置来打一个WAR包³²。

打完WAR包后，我们可以随意把应用部署到任何应用服务器上。不同的应用服务器需要不同的部署步骤。在这个例子中，我们将用免费的JBoss应用服务器³⁶做为目标服务器。对于JBoss来说，只要简单地复制WAR包到服务器的部署目录下，并启动服务器就可以了。

```

racetrack> cp racetrack.war /Applications/jboss-
4.0.5/server/default/deploy/
racetrack> /Applications/jboss-4.0.5/bin/run.sh
22:23:50,934 INFO [Server] Starting JBoss (MX
MicroKernel)...
...
22:24:13,069 INFO [TomcatDeployer] deploy,
ctxPath=/racetrack,
warUrl=.../tmp/deploy/tmp7718racetrack-exp.war/
...
22:24:35,671 INFO [Server] JBoss (MX MicroKernel)
[4.0.5] Started in 24s:726ms

```

如你所见，应用系统已经部署，启动，正等待着处理请求。打开你的浏览器来试试吧：<http://someproductionserver:8080/racetrack/>

应用服务器的怪癖

每个应用服务器都自己的怪癖，JBoss 也不例外。如果我们想正确地记录事件，就不得不越过一些限制。JBoss 认为日志配置应该属于应用服务器（而不是应用程序本身）。接下来的步骤都将顺

着这个思路走。

1. 删除 WAR 中所有和 log4j 有关的文件（我们将使用 JBoss 自身的 log4j 来代替）。
 - WEB-INF/log4j.properties
 - WEB-INF/lib/log4j-1.2.8.jar
2. 编辑 JBoss 的 log4j 配置文件（\$JBoss_HOME/server/default/conf/log4j.xml），加入我们的应用程序所需的日志配置。

```

<appender name="RACETRACK"
  class="org.jboss.logging.appenders.DailyRollingFileAppender">
  <errorHandler
    class="org.jboss.logging.util.OnlyOnceErrorHandler" />
  <param name="File"
    value="${jboss.server.log.dir}/racetrack.log" />
  <param name="Append" value="false" />

  <!-- Rollover at midnight each day -->
  <param name="DatePattern" value="''.yyyy-MM-dd" />

  <layout class="org.apache.log4j.PatternLayout">
    <!-- The default pattern: Date Priority
    [Category] Message\n -->
    <param name="ConversionPattern" value="%d %-5p
    [%c] %m%n" />
  </layout>
</appender>
<category name="UserController">
  <priority value="WARN" />
  <appender-ref ref="RACETRACK" />
</category>

```

修改完成之后，我们就可以再次使用日志文件来跟踪任何重要的事件了。

```

racetrack> cat /Applications/jboss-
4.0.5/server/default/log/racetrack.log
2006-10-28 22:40:09,174 WARN [UserController] Shields
up! Somebody's trying to hack through our rock-solid
DEFCON 1 security -- User ID - admin, Password -
letmein

```

9

InfoQ 中文站 Ruby 社区

面向 Web 和企业开发的 Ruby，主要关注

Ruby on Rails

<http://www.infoq.com/cn/ruby/>

深入应用的技巧

每个项目都是与众不同的，你的需求肯定跟眼前这个例子有些不一样的地方。下面的章节将提供一些常见的变通做法以及随之产生的问题的处理技巧。我们也会着眼于深入应用 Grails 的时候会用得上的一些常用技巧。

自行定义数据表

我们在 *RaceTrack* 例子中使用的方法能帮助我们快速搭建出应用程序，但是有时候我们想在数据库表的细节上有更多的控制能力（例如字段长度、约束属性、分区等等）。我们不一定要通过 Grails 来创建数据表。Grails 可以接受我们手工定制的数据表定义，同时只要我们遵循 Grails 的命名规约就不需要额外的配置。（如果我们真的不想遵循这些规约，在下一节中我们将讨论如何整合不符合规约要求的数据表定义）

为了让 Grails 知道我们打算自己管理数据库表，首先要从 `*DataSource` 类中删除（或者注释掉）`dbCreate` 属性，接下来我们要深入了解一下 Grails 的规约。

```
class DevelopmentDataSource {  
    boolean pooling = true  
    String dbCreate = "update"  
    String url =  
    "jdbc:mysql://localhost/racetrack_dev"  
    String driverClassName = "com.mysql.jdbc.Driver"  
    String username = "jason"  
    String password = ""  
}
```

领域类与表名

首先也是最重要的，Grails假设每个领域类都对应一张表，正如我们在*RaceTrack*应用中所见的，领域对象Race映射到表race，而领域对象Registration映射到表registration。如果你的设计刚好符合这个模式就不需要额外配置（如果你的领域模型引入了继承层次，请查阅Grails Wiki，详细了解一下“每个继承层次对应一张数据表”（table-per-hierarchy）的模型³⁷）。

属性与字段

Grails 期望领域类的属性和表的字段是一一对应的。对于名字只有一个单词的属性，属性名和字段名严格对应。例如，race 类中的 name 属性就直接对应于 race 表中的 name 字段。对于符合 Java 命名规范的由多个单词组合命名的属性，按照 Grails 规约，对应的字段名字等于用下划线分隔其中每个单词。例如，startDateTime 属性对应到字段 start_date_time。

我们在字段的类型和长度定义上有一定的灵活性。例如，我们可以映射一个 String 对象到一个 CHAR(4) 字段，也可以是 VARCHAR(255) 字段，又或者是 TEXT 字段。对于数字类型我们也有类似的灵活性。当然，作为开发人员，我们有责任定义正确的约束，确保领域对象可以正确地持久化。

最后，我们可以指定哪些字段允许为空。在这一点上，要宽要严都随我们心意；只要我们小心设计领域类的约束属性，避免应用程序试图持久化一个空属性值到一个非空字段就行了。

标识、关联及主键

GORM 给每个 Grails 领域类都提供了 id 和 version 属性，并且 Grail 希望每张表都有同样名字的两个字段。两个字段都应当被定义成整数类型，id 字段也应该是表的主键（具体的整数类型和长度由开发人员自行决定。只要保证长度能满足目标系统的数据量就行了）。

Grails 处理关联关系的方式也很直观。例如，在 *RaceTrack* 应用中，每个 `Registration` 对象都属于一个特定的 `Race` 对象。因而 Grails 期望 `registration` 表包含一个 `race_id` 字段，该字段同时作为外键指向 `race` 表。

处理遗留数据表

毫无疑问，如果你面对的是一个全新干净的程序，你可以遵循 Grails 的规约来取得最大的生产效率。毕竟，Grails 的一个核心前提是偏好规约远多于配置。不过，对于很多程序，一个全新的数据库定义几乎是一种奢望。特别是在企业应用中，我们经常面临着整合遗留系统数据库定义的难题，甚至对于定义的修改都不在选择范围内。幸运的是，Grails 并没有把我们拒之门外。即便面对遗留系统，我们依然可以从我们在 *RaceTrack* 应用中所看到的 Grails 的许多优点中受益菲浅。

因为在不符合 Grails 规约要求的数据库定义上应用 Grails，我们不得不回头进行配置。我们必须告诉 Grails 如何把领域类映射到表上，如何把属性映射到字段上，以及如何去唯一地标识每条记录。幸运的是，我们有不同的选择来完成这个映射工作。对于可以使用 Java 5 的人，你可以通过 Hibernate Annotation 提供映射信息³⁸。对于使用 JDK 1.4 的人，你可以用具有相同效果的 Hibernate XML 来提供映射信息³⁹。

ORM 问题与解决

如果你的確要对付遗留的数据库定义，或者陷入了复杂的关联关系和查询之中，你可能时不时发现程序提供的结果并不刚好符合你的期望。在此情况下，最好调查一下系统内的对象一关系映射在背后的動作是不是跟你设想的一样。你可以在数据源中设置 `logSql` 属性为 `true`，以启用 SQL 日志。

```
class DevelopmentDataSource {
    def logSql = true
    boolean pooling = true
    String dbCreate = "update" // one of 'create',
                           // 'create-drop', 'update'
    String url =
"jdbc:mysql://localhost/racetrack_dev"
```

```

String driverClassName = "com.mysql.jdbc.Driver"
String username = "jason"
String password = ""
}

```

打开 SQL 日志后，你将看到系统产生的 SQL 语句的构成。例如，当我们从“搜索比赛”页面上提交一个查询，我们将在控制台上看到如下的输出。

```

[groovy] Hibernate:
[groovy] select
[groovy]   this_.id as id1_0_,
[groovy]   this_.version as version1_0_,
[groovy]   this_.distance as distance1_0_,
[groovy]   this_.max_runners as max4_1_0_,
[groovy]   this_.start_date_time as start5_1_0_,
[groovy]   this_.state as state1_0_,
[groovy]   this_.cost as cost1_0_,
[groovy]   this_.name as name1_0_,
[groovy]   this_.city as city1_0_
[groovy]   from
[groovy]     race this_
[groovy]   where
[groovy]   (
[groovy]     this_.city like ?
[groovy]     and this_.state like ?
[groovy]     and this_.start_date_time between ? and ?
[groovy]   )

```

所有的数据库操作——创建、读取、更新以及删除——都将显示在控制台上。当你试图了解为什么某个属性没有被正确读取或者一个特定的查询丢失了它的查询参数，SQL 日志输出是你展开调试的一个好起点。因为你知道自己给了系统什么样的请求，通过观察系统把请求翻译成 SQL 语句，通常可以发现缺失的环节，从而帮助你找到问题所在。

升级 Grails

Grails 开发团队为该框架设定了一个雄心勃勃的发展蓝图⁴⁰。每隔数月 Grails 都会有一次小升级，我们也许希望随时升级我们的 Grails 应用，好用上最新版的新特性。在 Grails 里，升级的过程也一样是那么简单。

以防万一，在升级前记得备份你的应用程序。（当然，可靠的

备份并不是升级 Grails 的特殊要求。我们都在使用源代码管理系统，对吧？）

你首先需要下载和安装新版的 Grails，然后只需打开项目的根目录，输入 `grails upgrade` 命令。

```
racetrack> grails upgrade
...
BUILD SUCCESSFUL
Total time: 1 second
```

恭喜！你正运行着最新版的 Grails。现在，用那些新特性去做些伟大的事情吧！

总结

在阅读本书的过程中，我们亲身体验了如何用 Grails 在最短的时间内创建一个功能完整而又灵活的网络应用。Groovy 本身的强大表达能力让我们可以写出简练的代码，同时 Grails 使用合理的默认设置（即规约重于配置）为我们节约了许多传统框架所面临的大量的编码以及配置工作。当然，代码越少意味着维护的工作越少，同时随着需求的不断变化而需要改动程序的时候，我们要克服的遗留代码也相对更少了。

不断迭代的——甚至算是敏捷的——开发过程让我们达到今天的成功，我们也看到了Grails非常适合于这样的开发风格⁴¹。这种方式让我们在开发的过程中不断看到自身的进展，也为我们开启了一扇快速获取客户反馈的方便之门。也许我们没法在早上 10 点前就做完一天的工作去上高尔夫球课，然而或许明天兴致勃勃的客户会邀我们去打上一轮。

此外，Grails还有大量的特性我们未曾在这里展示。Grails的主页（<http://grails.org>）上有最新的新闻和更新。如果你想更深入学习Grails服务、Ajax支持（包括Prototype、Dojo以及Yahoo UI库）、定时任务、事务管理、JEE集成等等的话，你将在Grails网站找到更多的信息。如果上面恰好没有你要的信息，记得给正在不断增长的用户邮件列表发个帖子⁴²。

Web 开发并不意味着一个缓慢而笨重的开发过程，有了 Grails，在 Java 平台上实现快速网络应用开发的时代已经到来。

InfoQ 中文站 SOA 社区

关于大中型企业内面向服务架构的一切

<http://www.infoq.com/cn/soa/>

关于作者

Jason Rudolph 是 Railinc 公司的应用程序架构师，他开发的软件帮助全北美的火车提高了运转效率。最近他发布了一个全行业范围的检查报告和管理系统，被财富 500 强内的铁路企业、设备租赁公司以及联邦铁路管理部门用于操作安全管理。

Jason 专注于研究如何把合理的软件工程实践应用于解决难题和对付真正的商业挑战。Jason 曾经历几个大型的企业级 JEE 应用的构建，面对的是商业伙伴、遗留系统和网络用户，并要为高事务吞吐量提供可伸缩性的解决方案。他喜欢作为一个方案设计者，一个架构师，和一个开发者，他相信真正的令人满意的代码来自于实战，来自于把蓝图变为真正起作用的程序。

Jason的兴趣包括了动态语言、轻量级开发方法、提高开发者开发效率和寻求保持编程的快乐。正是这些兴趣让Jason成为Grails的贡献者和传道者。Jason拥有弗吉尼亚大学的计算机学位。他现在与他的夫人（她擅长于把网络应用的界面搞得美轮美奂）还有他的狗（它可以轻松超过他，但是总追不上松鼠）一起居住于北卡罗莱纳州瑞利市。你能通过<http://jasonrudolph.com>联系到Jason。

InfoQ 中文站 Architecture 社区
设计、技术趋势，以及架构师感兴趣的话题
<http://www.infoq.com/cn/architecture/>

资源

¹ <http://www.jcp.org/en/jsr/detail?id=241>

获取 Groovy 规范的官方详细资料（即 JSR 241:The Groovy Programming Language）

² <http://groovy.codehaus.org/>

在 Groovy 的网站上你可以获取所有 Groovy 官方的资源。在这里你可以获得下载和安装 Groovy、指南、API、FAQ、文档、新闻等各方面的信息。

³ <http://www.infoq.com/cn/minibooks/grails>

这个 ZIP 文件包含了本书中所有例子的完整源代码，每一章结束时的代码快照分别放在了相应的目录中。

⁴ <http://java.sun.com/javase/downloads/>

Sun 的 Java Standard Edition 下载页面提供的 JDK 版本几乎涵盖了所有平台。

⁵ <http://grails.org/Download>

Grails 下载页面提供了当前 Grails 的稳定版本和即将发布的版本的快照。

本书中的例子应该在 Grails 以后的版本中也可以使用；但在你边看书边试的时候，我还是建议你最好使用下面列出的 Grails 0.3.1 发布版。

<http://dist.codehaus.org/grails/grails-bin-0.3.1.tar.gz>

<http://dist.codehaus.org/grails/grails-bin-0.3.1.zip>

⁶ <http://grails.org/Installation>

Grails 的安装说明能助你立即上手。

⁷ <http://dev.mysql.com/downloads/mysql/5.0.html>

下载和安装你开发时需要的 MySQL Community Server 5.0。完全免费！

⁸ <http://grails.org/Quick+Start#QuickStart-CreateaGrailsproject>

Grails 快速入门指南提供了 Grails 应用的结构的正式描述。

⁹ <http://en.wikipedia.org/wiki/DRY>

在 Wikipedia 里关于 DRY 基本原则的解释。

¹⁰ <http://grails.org/GORM+-+Defining+relationships#GORM-Definingrelationships-RelationshipSummary>

GORM 文档简要列举了在 Grails 中各种不同的关系类型以及每种类型相关的关键字。

¹¹ <http://www.mysql.com/products/connector/j/>

MySQL Connector/J 提供了 Grails 应用程序（或者是任何 Java 或 Groovy 程序）操作 MySQL 数据库所需的 JDBC 驱动。

¹² <http://grails.org/Validation+Reference>

当你开始往 Grails 应用中添加约束属性，请务必把这个地址加入书签。Grails 验证参考文档列出了所有约束属性及其用法，并提供了自定义错误信息所需的资料。

¹³ <http://groovy.codehaus.org/Closures>

Groovy 网站中关于闭包的介绍部分提供了关于 Groovy 闭包的目的、语法和语义的详细解释。该页提供了几个易上手的例子，并且讲述了闭包、代码段和匿名内部类之间的区别。

¹⁴ <http://groovy.codehaus.org/GroovyMarkup>

Groovy Markup 提供了一种方便的机制，让你用 Groovy 来构建 XML、HTML、Hibernate Criteria 等。

¹⁵ <http://grails.org/Validation#Validation-ValidatingDomainClasses>

在 Grails 文档的这部分中，解释了如何定义约束属性闭包，以及如何使用 Grails 的领域类的检验和保存方法。

¹⁶ <http://grails.org/Validation+Reference#ValidationReference-validator>

Grails 检验框架参考指南除了讲述很多标准的 Grails 检验器外，也讲述了如何去定义和使用自定义检验器。

¹⁷ <http://groovy.codehaus.org/apidocs/groovy/lang/GString.html>

Groovy GString API 列举了由这个类提供的许多便捷的方法，使用这些方法的例子，还顺带提及了 GString 这个双关名字的来源。

¹⁸ <http://grails.org/Controllers#Controllers-Settingthedefaultaction>

Grails 为控制器的默认 Action 提供了两种可选的设置方式。

¹⁹

<http://grails.org/Dynamic+Methods+Reference#DynamicMethodsReference-Controllers>

Grails 控制器提供了相当多的动态方法和特性，从访问 Request 和 Session，到记录日志，到实现重定向，言不能尽。

20

<http://grails.org/Dynamic+Methods+Reference#DynamicMethodsReference-DomainClasses>

为了让 Grails 控制器工作得更好，Grails 领域类提供了动态的特性和方法来帮助你省去繁复的 ORM 工作。

²¹ <http://grails.org/GSP+Tag+Reference>

GSP 标签参考指南讲述了表单、验证、Ajax 等各个方面的 Grails 标签。每个标签都有详尽的 API 说明和使用的例子。

²² <http://grails.org/Tag+++datePicker>

datePicker 标签会呈现一个 UI 控件来提供日期/时间值的选择功能，它还带有控制精度的参数。

²³ <http://grails.org/Builders#Builders-HibernateCriteriaBuilder>

Hibernate Criteria Builder 提供了一个方便的机制来对你的领域模型进行各种查询。

²⁴ http://en.wikipedia.org/wiki/Polish_notation

真正的计算机科学怪人总是对波兰表达式念念不忘。但不要尝试用这种语法跟凡人沟通。

²⁵ <http://groovy.codehaus.org/Roadmap#Roadmap-Groovy2.0>

Groovy 路线图给我们展示了未来各 Groovy 版本的动向，包括了在 Groovy 2.0 时对枚举的支持。

²⁶ <http://en.wikipedia.org/wiki/SHA>

Wikipedia 上对 SHA (Secure Hash Algorithm) 有十分清晰的解释，也涵盖了它的各种变体，以及为什么对于大多数加密应用来说，SHA 都是个不错的选择。

27

<http://java.sun.com/j2se/1.5.0/docs/api/java/security/MessageDigest.html>

如果你准备为程序加上一把锁，并且正在考虑你行实现认证模块，那么可以看一下 Java 的 `MessageDigest` 类的 API，看看它的单向散列功能。

28

<http://grails.org/Configuration#Configuration-ConfiguringStartup>

Grails 文档包括了对 Bootstrap 功能的解释，该功能可以定义程序启动和关闭时所触发的任务。

29

<http://grails.org/Controllers#Controllers-ActionInterceptors>

Grails 的 Action 拦截器允许你在 Action 开始或结束执行的时刻注入额外的代码。

30

<http://grails.org/Views+and+Layouts#ViewsandLayouts-LayoutbyConvention>

除了用名字来引用布局（如 *RaceTrack* 例子所示），Grails 还允许你用规约的方式来决定相应视图的布局。

31

<http://groovy.codehaus.org/Unit+Testing>

Groovy 明显地简化了 Java 平台上的单元测试。Groovy 单元测试页面演示了几个 Groovy 如何帮助加快测试用例的开发的例子。

32

<http://grails.org/Configuration#Configuration-Environments>

Grails 允许你为各种命令指定所需要的环境，同时为各命令预先设定了合理的默认值。

33

<http://grails.org/Functional+Testing>

除了单元测试框架外，Grails 功能测试框架可助你在积极快速的迭代开发中更好地维护你的应用程序。

34

<http://grails.org/Controller+Dynamic+Methods#ControllerDynamicMethods-log>

看看在 Grails 控制器里如何访问和自定义日志记录功能。

35 <http://logging.apache.org/log4j/docs/documentation.html>

因为 Grails 日志记录功能使用的是 log4j，所以你能利用 log4j 极其灵活的优势度身订造你自己的日志功能。

36 <http://labs.jboss.com/portal/jbossas/download>

下载和安装 JBoss 应用服务器，享受免费的、产品级的部署方案。

37 <http://www.grails.org/GORM++Mapping+inheritance>

GORM 文档里解释了领域类的继承如何影响 ORM。

38 <http://grails.org/Hibernate+Integration#HibernateIntegration-MappingwithHibernateAnnotations>

Hibernate Annotations 让开发者即使面对非标准的数据库定义，也能使用 Grails 来进行他们的应用程序开发。

39 <http://jasonrudolph.com/blog/2006/06/20/hoisting-grails-to-your-legacy-db/>

如果你的环境暂时还不支持 Annotations（即是说你正在使用 JDK 1.4），你仍然可以通过 Hibernate XML 来映射遗留数据库定义。

40 <http://grails.org/Roadmap>

Grails 的路线图里设定了接下来每个版本的特征和改进，同时提供了每个版本大约的发布时间。

⁴¹ <http://www.amazon.com/exec/obidos/ASIN/0131111558/>

Craig Larman 的《*Agile and Iterative Development: A Manager's Guide*》，是一本关于敏捷开发的非常出色的介绍书籍。

⁴² <http://www.nabble.com/grails---user-f11861.html>

Grails 用户邮件列表提供了很多各种各样的问题的答案，你可以方便地进行搜索。你同样可以在邮件列表里提出你的问题或者提供你的建议，我们会在很短时间内回复你。