|  |
| --- |
| Play 1.2.4 |
| Play 技术宝典 |
| 学习第一手资料收集 赵浩霖 |

2012-4-24

目录

[**MVC应用程序模型** - 7 -](#_Toc322991653)

[**app/controllers** - 8 -](#_Toc322991654)

[**app/models** - 8 -](#_Toc322991655)

[**app/views** - 8 -](#_Toc322991656)

[**请求生命周期** - 8 -](#_Toc322991657)

[**标准应用程序布局layout** - 9 -](#_Toc322991658)

[**app目录** - 9 -](#_Toc322991659)

[**public目录** - 10 -](#_Toc322991660)

[**conf目录** - 10 -](#_Toc322991661)

[**lib目录** - 11 -](#_Toc322991662)

[**开发生命周期** - 11 -](#_Toc322991663)

[**连接到java调试器** - 12 -](#_Toc322991664)

[**类增强Enhancement** - 13 -](#_Toc322991665)

[**02.HTTP路由** - 13 -](#_Toc322991666)

[**关于REST** - 14 -](#_Toc322991667)

[**routes文件语法** - 14 -](#_Toc322991668)

[**HTTP方法** - 15 -](#_Toc322991669)

[**URI范示 Pattern** - 15 -](#_Toc322991670)

[**Java调用定义** - 17 -](#_Toc322991671)

[**把404当作action来用** - 17 -](#_Toc322991672)

[**指派静态参数** - 17 -](#_Toc322991673)

[**变量和脚本** - 18 -](#_Toc322991674)

[**路由优先级** - 18 -](#_Toc322991675)

[**服务器静态资源** - 18 -](#_Toc322991676)

[**staticDir: mapping** - 18 -](#_Toc322991677)

[**staticFile: mapping** - 19 -](#_Toc322991678)

[**URL 编码** - 19 -](#_Toc322991679)

[**反转路由：用于生成某些URL** - 19 -](#_Toc322991680)

[**设置内容风格(CSS)** - 20 -](#_Toc322991681)

[**HTTP 内容协商 negotiation** - 21 -](#_Toc322991682)

[**从http headers开始设置内容类型** - 21 -](#_Toc322991683)

[**定制格式** - 22 -](#_Toc322991684)

[**03.控制器** - 23 -](#_Toc322991685)

[**控制器概览** - 23 -](#_Toc322991686)

[**获取http参数** - 24 -](#_Toc322991687)

[**使用params map** - 25 -](#_Toc322991688)

[**还可以从action方法签名实现转换** - 25 -](#_Toc322991689)

[**高级HTTP Java绑定** - 26 -](#_Toc322991690)

[**简单类型** - 26 -](#_Toc322991691)

[**Date类型** - 26 -](#_Toc322991692)

[**Calendar日历** - 27 -](#_Toc322991693)

[**File** - 27 -](#_Toc322991694)

[**支持类型的数组或集合** - 28 -](#_Toc322991695)

[**POJO对象绑定** - 29 -](#_Toc322991696)

[**JPA 对象绑定** - 30 -](#_Toc322991697)

[**定制绑定** - 30 -](#_Toc322991698)

[**@play.data.binding.As** - 30 -](#_Toc322991699)

[**@play.data.binding.NoBinding** - 31 -](#_Toc322991700)

[**play.data.binding.TypeBinder** - 31 -](#_Toc322991701)

[**@play.data.binding.Global** - 32 -](#_Toc322991702)

[**结果类型** - 32 -](#_Toc322991703)

[**返回一些文本类型的内容** - 33 -](#_Toc322991704)

[**返回一个JSON字符串** - 33 -](#_Toc322991705)

[**返回一个XML字符串** - 34 -](#_Toc322991706)

[**返回二进制内容** - 34 -](#_Toc322991707)

[**作为附件下载文件** - 34 -](#_Toc322991708)

[**执行一个模板** - 35 -](#_Toc322991709)

[**跳转到其他URL** - 36 -](#_Toc322991710)

[**Action链** - 36 -](#_Toc322991711)

[**定制web编码** - 37 -](#_Toc322991712)

[**拦截器** - 38 -](#_Toc322991713)

[**@Before** - 38 -](#_Toc322991714)

[**@After** - 39 -](#_Toc322991715)

[**@Catch** - 40 -](#_Toc322991716)

[**@Finally** - 41 -](#_Toc322991717)

[**控制器继承** - 42 -](#_Toc322991718)

[**使用@With注释添加更多的拦截器** - 42 -](#_Toc322991719)

[Because Java does not allow multiple inheritance, it can be very limiting to rely on the Controller hierarchy to apply interceptors. But you can define some interceptors in a totally different class, and link them with any controller using the @With annotation.由于java不允许多继承，通过控制器继承特点来应用拦截器就受到极大的限制。但是我们可以在一个完全不同的类里定义一些拦截器，然后在任何控制器里使用@With注释来链接他们。 - 42 -](#_Toc322991720)

[**Session和Flash作用域** - 42 -](#_Toc322991721)

[**04.模板引擎** - 43 -](#_Toc322991722)

[**模板语法** - 43 -](#_Toc322991723)

[**Expressions: ${…}** - 44 -](#_Toc322991724)

[**Template decorators : #{extends /} and #{doLayout /}** - 44 -](#_Toc322991725)

[**Tags: #{tagName /}** - 45 -](#_Toc322991726)

[**Actions: @{…} or @@{…}** - 46 -](#_Toc322991727)

[**Messages: &{…}** - 46 -](#_Toc322991728)

[**Comment: \*{…}\*** - 46 -](#_Toc322991729)

[**Scripts: %{…}%** - 46 -](#_Toc322991730)

[**Template inheritance继承** - 47 -](#_Toc322991731)

[**定制模板标签** - 48 -](#_Toc322991732)

[**检索tag参数** - 48 -](#_Toc322991733)

[**调用标签体** - 48 -](#_Toc322991734)

[**格式化特定标签** - 49 -](#_Toc322991735)

[**定制java标签** - 49 -](#_Toc322991736)

[**标签命名空间** - 50 -](#_Toc322991737)

[**在模板里的Java对象扩展** - 51 -](#_Toc322991738)

[**创建定制扩展** - 52 -](#_Toc322991739)

[**模板里可以使用的保留对象** - 52 -](#_Toc322991740)

[**05.用play验证http数据** - 53 -](#_Toc322991741)

[**在play里验证如何进行的？** - 53 -](#_Toc322991742)

[**验证的错误消息** - 54 -](#_Toc322991743)

[**Localised validation messages 局部验证消息** - 55 -](#_Toc322991744)

[**验证消息参数** - 55 -](#_Toc322991745)

[**定制局部验证消息** - 56 -](#_Toc322991746)

[**定制teral（非局部）验证消息** - 57 -](#_Toc322991747)

[**在模板里显示验证错误消息** - 57 -](#_Toc322991748)

[**验证注释** - 60 -](#_Toc322991749)

[**验证复杂对象** - 60 -](#_Toc322991750)

[**内建验证** - 61 -](#_Toc322991751)

[**使用@CheckWith定制验证** - 61 -](#_Toc322991752)

[**定制注释** - 62 -](#_Toc322991753)

[**06.域对象模型** - 64 -](#_Toc322991754)

[**属性模仿** - 65 -](#_Toc322991755)

[**设置数据库来持久化模型对象** - 68 -](#_Toc322991756)

[**用hibernate持久化对象模型** - 69 -](#_Toc322991757)

[**保持模型stateless** - 70 -](#_Toc322991758)

[07.JPA持久化 - 70 -](#_Toc322991759)

[启动JPA实体管理器 - 70 -](#_Toc322991760)

[获取JPA实体管理器 - 70 -](#_Toc322991761)

[事务管理 - 71 -](#_Toc322991762)

[play.db.jpa.Model支持类 - 71 -](#_Toc322991763)

[为GenreicModel定制id映射 - 72 -](#_Toc322991764)

[Finding对象 - 72 -](#_Toc322991765)

[Find by ID - 72 -](#_Toc322991766)

[Find all - 73 -](#_Toc322991767)

[使用简单查询进行查找 - 73 -](#_Toc322991768)

[使用JPQL 查询进行查找 - 74 -](#_Toc322991769)

[Counting统计对象 - 74 -](#_Toc322991770)

[用play.db.jpa.Blob存储上传文件 - 74 -](#_Toc322991771)

[强制保存 - 75 -](#_Toc322991772)

[更多公共类型generic typing问题 - 77 -](#_Toc322991773)

[**08.Play.libs库包** - 78 -](#_Toc322991774)

[**用XPath解析XML** - 78 -](#_Toc322991775)

[**Web Service client** - 79 -](#_Toc322991776)

[**Functional programming with Java功能扩展？** - 79 -](#_Toc322991777)

[**Option<T>, Some<T> and None<T>** - 80 -](#_Toc322991778)

[**Tuple<A, B>** - 80 -](#_Toc322991779)

[**Pattern Matching模式匹配** - 81 -](#_Toc322991780)

[**Promises** - 81 -](#_Toc322991781)

[**OAuth** - 82 -](#_Toc322991782)

[**OAuth 1.0** - 82 -](#_Toc322991783)

[**OAuth 2.0** - 83 -](#_Toc322991784)

[**OpenID** - 84 -](#_Toc322991785)

[**09.异步Jobs** - 86 -](#_Toc322991786)

[**引导程序任务Bootstrap jobs** - 87 -](#_Toc322991787)

[**预定义任务Scheduled jobs** - 87 -](#_Toc322991788)

[**触发任务job** - 88 -](#_Toc322991789)

[**停止应用程序** - 89 -](#_Toc322991790)

[**10.在HTTP下进行异步编程** - 89 -](#_Toc322991791)

[**暂停http请求** - 89 -](#_Toc322991792)

[**Continuations** - 90 -](#_Toc322991793)

[**回调Callbacks** - 91 -](#_Toc322991794)

[**HTTP response流 streaming** - 92 -](#_Toc322991795)

[**使用WebSockets** - 92 -](#_Toc322991796)

[**11.在play框架里使用Ajax** - 94 -](#_Toc322991797)

[**通过jsAction标签使用jQuery** - 95 -](#_Toc322991798)

[**12.** **Internationalization国际化支持** - 96 -](#_Toc322991799)

[**仅使用 UTF-8!** - 96 -](#_Toc322991800)

[**国际化你的信息** - 96 -](#_Toc322991801)

[**通过应用程序定义支持的语言** - 96 -](#_Toc322991802)

[**依照你的区域定义日期格式** - 97 -](#_Toc322991803)

[**找回区域信息** - 97 -](#_Toc322991804)

[**Message arguments** - 97 -](#_Toc322991805)

[**模板输出** - 98 -](#_Toc322991806)

[**多参数** - 98 -](#_Toc322991807)

[**立即数Argument indices** - 98 -](#_Toc322991808)

[**13.使用cache** - 99 -](#_Toc322991809)

[**The cache API** - 99 -](#_Toc322991810)

[**不要把Session当成缓存!** - 101 -](#_Toc322991811)

[**配置mcached** - 101 -](#_Toc322991812)

[**14.发送e-mail** - 102 -](#_Toc322991813)

[**Mail 和MVC 集成** - 102 -](#_Toc322991814)

[**text/html e-mail** - 103 -](#_Toc322991815)

[**text/plain e-mail** - 104 -](#_Toc322991816)

[**text/html e-mail with text/plain alternative** - 104 -](#_Toc322991817)

[**在应用程序里链接到邮件** - 104 -](#_Toc322991818)

[**SMTP配置** - 105 -](#_Toc322991819)

[**使用 Gmail** - 105 -](#_Toc322991820)

[**15.测试应用程序** - 105 -](#_Toc322991821)

[**书写测试程序** - 105 -](#_Toc322991822)

[**单元测试** - 106 -](#_Toc322991823)

[**功能性测试** - 106 -](#_Toc322991824)

[**Selenium test用例测试** - 107 -](#_Toc322991825)

[**Fixtures固定值** - 108 -](#_Toc322991826)

[**运行测试** - 110 -](#_Toc322991827)

[**陆续集成,并自动运行测试** - 111 -](#_Toc322991828)

[**16.安全指南** - 112 -](#_Toc322991829)

[**Sessions** - 112 -](#_Toc322991830)

[**守住你的安全…安全** - 112 -](#_Toc322991831)

[**不要存储关键性的数据** - 112 -](#_Toc322991832)

[**跨站点脚本攻击** - 112 -](#_Toc322991833)

[**SQL注入** - 113 -](#_Toc322991834)

[**跨站点请求伪造** - 114 -](#_Toc322991835)

[**17.Play模块和模块仓库** - 115 -](#_Toc322991836)

[**什么是模块?** - 115 -](#_Toc322991837)

[**如何从一个应用程序里加载模块** - 115 -](#_Toc322991838)

[**从模块加载默认的routes** - 115 -](#_Toc322991839)

[**为模块增加文档说明** - 115 -](#_Toc322991840)

[**使用模块仓库** - 116 -](#_Toc322991841)

[**贡献新模块到模块仓库里** - 117 -](#_Toc322991842)

[**先决条件** - 117 -](#_Toc322991843)

[**模块注册** - 117 -](#_Toc322991844)

[**发布你的模块** - 118 -](#_Toc322991845)

[**18.依赖管理** - 118 -](#_Toc322991846)

[**依赖格式** - 119 -](#_Toc322991847)

[**动态版本** - 119 -](#_Toc322991848)

[**dependencies.yml** - 119 -](#_Toc322991849)

[**‘play dependencies’命令** - 120 -](#_Toc322991850)

[**透明依赖** - 121 -](#_Toc322991851)

[**保持lib/和modules/目录同步** - 122 -](#_Toc322991852)

[**冲突判定Conflict resolution** - 123 -](#_Toc322991853)

[**增加要的仓库** - 124 -](#_Toc322991854)

[**Maven仓库** - 125 -](#_Toc322991855)

[**本地仓库** - 125 -](#_Toc322991856)

[**定制ivy设置（Apache ivy:项目依赖管理工具）** - 126 -](#_Toc322991857)

[**清除Ivy缓存** - 127 -](#_Toc322991858)

[**19.管理数据库变化Evolution** - 128 -](#_Toc322991859)

[**Evolutions脚本** - 128 -](#_Toc322991860)

[**同步同时发生的改变** - 130 -](#_Toc322991861)

[**数据不一致状态** - 133 -](#_Toc322991862)

[**Evolutions 命令** - 136 -](#_Toc322991863)

[**20.日志配置** - 139 -](#_Toc322991864)

[**对应用程序进行日志** - 139 -](#_Toc322991865)

[**配置日志级别** - 140 -](#_Toc322991866)

[**生产配置** - 140 -](#_Toc322991867)

[**21.管理多环境下的application.conf** - 140 -](#_Toc322991868)

[**框架id(framework ID)** - 141 -](#_Toc322991869)

[**从命令行设置框架id** - 142 -](#_Toc322991870)

[**22.生产部署** - 142 -](#_Toc322991871)

[**application.conf** - 142 -](#_Toc322991872)

[**设置框架为prod模式：** - 142 -](#_Toc322991873)

[**定义一个真实的数据库：** - 143 -](#_Toc322991874)

[**禁止JPA的自动结构更新：** - 143 -](#_Toc322991875)

[**定义一个安全的secret key:** - 143 -](#_Toc322991876)

[**日志配置** - 143 -](#_Toc322991877)

[**前端http服务器（Front-end HTTP server）** - 144 -](#_Toc322991878)

[**部署到lighttpd服务器的设置** - 144 -](#_Toc322991879)

[**部署到Apache服务器的设置** - 145 -](#_Toc322991880)

[**Apache作为前端代理服务器，可以允许透明更新你的应用程序** - 145 -](#_Toc322991881)

[**高级代理设置** - 146 -](#_Toc322991882)

[**HTTPS配置** - 147 -](#_Toc322991883)

[**不依赖Python进行部署** - 148 -](#_Toc322991884)

[**23.部署选择** - 148 -](#_Toc322991885)

[**独立Play应用程序** - 149 -](#_Toc322991886)

[**Java EE应用服务器** - 149 -](#_Toc322991887)

[**支持应用服务器** - 149 -](#_Toc322991888)

[**部署** - 150 -](#_Toc322991889)

[**数据源** - 150 -](#_Toc322991890)

[**定制web.xml** - 151 -](#_Toc322991891)

[**基于云的主机Cloud-based hosting** - 151 -](#_Toc322991892)

[**AWS Elastic Beanstalk** - 152 -](#_Toc322991893)

[**CloudBees** - 152 -](#_Toc322991894)

[**Cloud Foundry** - 152 -](#_Toc322991895)

[**Google App Engine (GAE)** - 152 -](#_Toc322991896)

[**Heroku** - 152 -](#_Toc322991897)

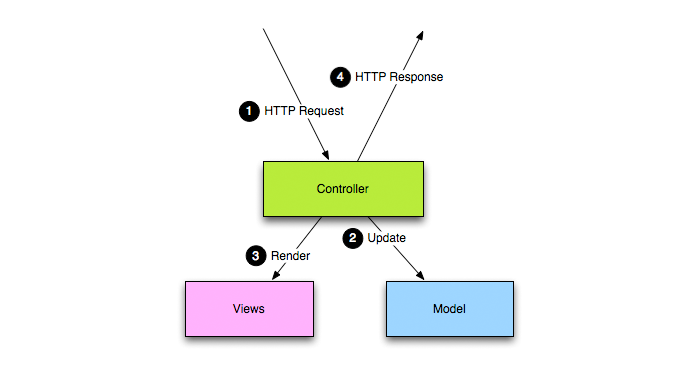
[**playapps.net** - 153 -](#_Toc322991898)

# 01.Play框架最主要的概念

**MVC应用程序模型**

Play框架完全遵循MVC模式,MVC模式把应用程序分成几个独立的层：presentation表现层和model模型层，表现层进一步分成view呈现层和controller控制层。

* **Model**层是应用程序操作的特定领域展现的信息。领域逻辑把“领域的意思”添加到原始数据里。许多应用程序使用持久存储机制，比如数据库来存储数据。MVC不特别指明数据访问层，因为mvc模型能够理解底层操作，或数据访问层已经被模型进行了封装。
* **View** 层把model层渲染到一个适当的交互窗体，如典型的用户接口。 一个model可以有多个views，以实现不同的作用。在一个web应用程序里，view通用渲染成web格式的内容，比如HTML, XML或JSON。然而，在某些情况下，view也可表示成二进制窗体，比如一个动态渲染的图表。
* **Controller** 负责响应事件 (通常是用户actions)，并对这些事件进行处理，也可能是调用model的修改方法。在一个Web应用程序里，事件特指http请求：一个专门用于监听http请求的控制器，它从’事件’里提取相关数据，比如查询字符串参数，请求headers… 并把修改结果更新到底层model对象。



在Play应用程序里，这三个层被分别定义到app目录下的三个java包里。

**app/controllers**

控制器就是一个java类，其中的每个public/static方法都是一个Action。一个action就是一个java入口点，当接收到一个http请求时，这个action就会被调用。控制器类里的java代码并不真正的面向对象的。Action方法从http请求中提取相关的数据，读取或更新model对象，并向http请求者返回一个封装好的response结果。

**app/models**

领域模型对象层（以下简称model）是一系列完全使用java面向对象语言特征的java类，它包含了数据结构和数据操作。无论何时，model对象都需要存储到持久化存储设备里，一般情况下他们或许还包含有一些jpa注释和sql语句。

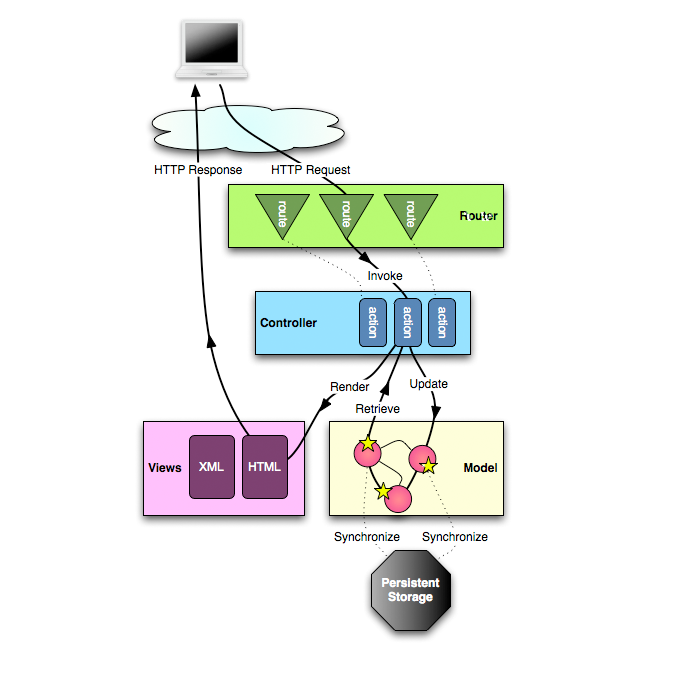
**app/views**

通过使用play提供的高效模板系统，可以生成大多数应用程序的views。控制器从model层获得某些感兴趣的数据，然后把这些数据应用到一个模板，并且通过模板把这些数据装饰得非常漂亮。这个包由HTML, XML, JSON或其他特定用于动态生成model展现的模板文件组成。

**请求生命周期**

Play框架是一个完全的stateless框架，而且只面向request/response。所有的http请求都遵循以下过程：

1. 一个http请求被框架接收。
2. Router组件试着找到能够接收这个请求的确切路由，相应的action方法随后被调用。
3. 应用程序代码被执行。
4. 如果需要生成一个复杂的view，那么一个模板文件将会被渲染。
5. acton方法的结果（http 响应代码、内容）随后被作为http response发出。



**标准应用程序布局layout**

应用程序布局标准化是保持开发简单化的保证。

**app目录**

这个目录包含了所有可执行的代码：java源代码和view模板。

**我的.class文件去哪里了？**   
  
请不要寻找编译过的java类。框架只在运行时才编译java源代码，而且只会把编译后类文件缓存到tmp目录下。在play框架里，可执行文件是.java源文件，而不是编译后的类。

在app目录下有三个标准包，分别对应mvc架构的三个层，你也可以添加你自己的包，如utils包。

另外，views包下，还可以有以下子包：

* tags, 主应用程序的标签包，比如可重用的模板片段。
* 每个控制器的views文件夹–按照约定，每个控制器相关的模板都要存储到他们自己的子包里。

**public目录**

存储在public目录的资源都是些可以直接被web服务器向外发布的静态资产。

这个目录共分为三个标准的子目录：分别存储images, CSS和JavaScript文件。 你应用试着像这个标准一样组织你的静态资产，以保持所有play应用程序的一致性。

默认情况下，/public目录被映射到/public URL路径，你也可以自行修改映射到其他目录，甚至为你的静态资产使用多个目录。

**conf目录**

conf目录包含了所有应用程序的配置文件。

这里有两个必须的配置文件，application.conf和routes:

* application.conf,应用程序最主要的配置文件。它包含了标准的配置参数。
* routes, 路由定义文件。

如果你需要增加一些特定配置选项，直接在application.conf中进行添加比较易于管理，文件的配置选项可在程序代码中通过Play.configuration.get("propertyName")方法读取。 当你创建一个新的应用程序时，play new命令将从$PLAY\_HOME/resources/application-skel/conf目录复制一个默认的配置文件。

如果其他库需要一个指定的配置文件，那么请把将该指定的配置文件放到conf目录（**这个目录是play指定的java 类路径 ClassPath**）,并在application.conf中用@include进行指定。

**注意，这是一个尚处在试验阶段的特性，目前或许还不能正常工作。尤其是占位符和框架id还不能正确处理这个特性。**

比如，如果你在conf/mime-types.conf文件里定义了一个附加的MIME类型：

# Web fonts

mimetype.eot = application/vnd.ms-fontobject

mimetype.otf = application/octet-stream

mimetype.ttf = application/octet-stream

mimetype.woff = application/x-font-woff

通过在application.conf文件里添加下面这条代码就可以把这些配置包含进来：

@include.mime = mime-types.conf

**lib目录**

这个目录包含了所有应用程序需要的标准java库，他们会自动添加到java classpath里。

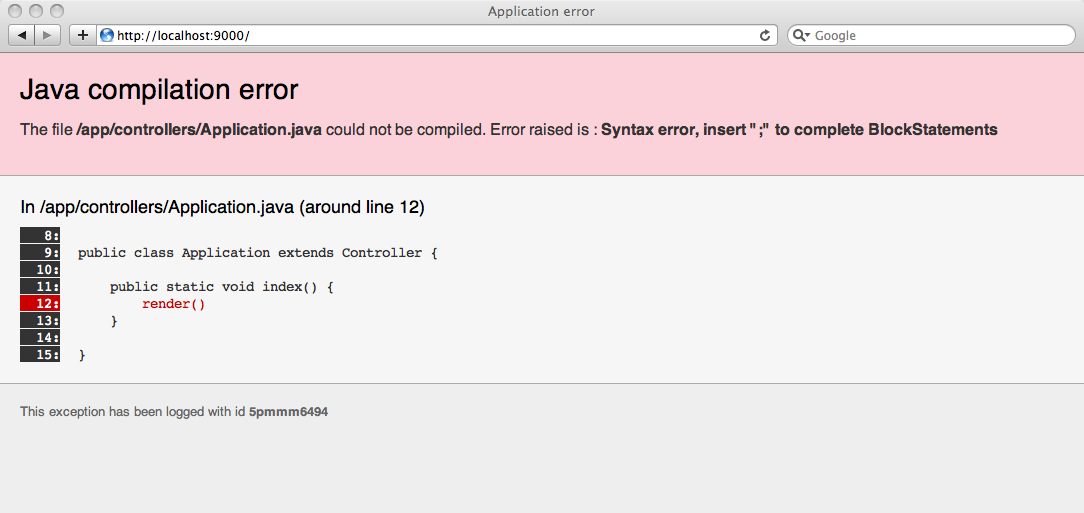
**开发生命周期**

使用play进行开发，没有编码、打包或部署阶段。然而，play通过两种环境来执行这些过程：DEV模式用于开发阶段，PROD模式用于部署阶段。

关于**DEV/PROD模式**  
  
应用程序可以运行于DEV或PROD模式。使用[application.mode configuration](http://www.playframework.org/documentation/1.2.4/configuration#application.mode)可以进行切换。当运行于DEV模式时，play将检查文件修改，并在必要的情况下重新加载程序。  
  
PROD模式是一个十分高效的生产环境：java源代码和模板将被编译一次，并为所有的用户进行缓存。

java源代码将在运行时进行编译和加载。当应用程序在运行时，如果一个Java源文件被修改了，那么这个源文件将会在JVM里被重新编译并进行热交换。

如果有编译错误发生，精确的错误发生点将会显示在浏览器里（仅限DEV模式）。

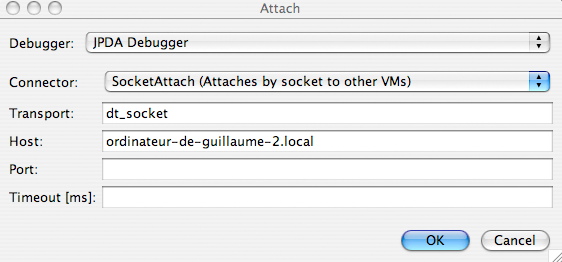


模板文件也是热编译、热加载的。

**连接到java调试器**

当在DEV模式运行应用程序时，你可以通过8000端口连接到一个java调试器。

比如，使用NetBeans调试器：



**类增强Enhancement**

Play plug-in (比如play.PlayPlugin的子类) 可以包含‘enhancers’，以便于在运行时修改应用程序库，以增加功能。这就是play一些神奇的地方。

内建的play.CorePlugin使用enhancers（play.classloading.enhancers包）来动态添加代码到你的应用程序里：

* ContinuationEnhancer –为控制器类添加continuations支持
* ControllersEnhancer – 让控制器的action方法实现线程安全，还可以为方法调用添加HTTP跳转功能
* LocalvariablesNamesEnhancer – 跟踪控制器里的本地变量名称
* MailerEnhancer – 设置play.mvc.Mailer子类
* PropertiesEnhancer - 把所有的应用程序类转换到可用的JavaBeans(其中的属性是基于域的)
* SigEnhancer – 为每个类的签名计算一个唯一的哈希值，以便自动加载

另外，play.db.jpa.JPAPlugin增强了play.db.jpa.JPABase的子类，提供了更方便的jpa查询方法。这个一般用于play.db.jpa.Model的子类，作为应用程序的model类。如果这些类作为play.db.jpa.GenericModel的子类，会存在一定问题。

要想添加自己的java增强特性，use a subclass of play.classloading.enhancers.Enhancer in your plug-in’s enhance(ApplicationClass) method。

**02.HTTP路由**

router组件负责将http请求交给对应的Action处理（一个static/public的控制器方法）。

一个http请求在mvc框架里被当作一个mvc事件看待。这个事件包含了两个主要信息：

* 包含在查询字符串里的请求路径 (比如/clients/1542, /photos/list)。
* HTTP方法(GET, POST, PUT, DELETE)

**关于REST**

Representational state transfer (表述性状态转移REST)是一种应用于分布式超媒体系统（如www）的软件架构设计风格（注意：不是标准）。

REST规定了几个关键的设计原则：

* 应用程序功能都被当作资源看待
* 每个资源都有唯一的可寻址URI地址
* 所有的资源共享同一个接口来传送客户端和资源间的状态
* 其表现形式为每个超链接都不带有参数

如果你正在使用http，那么这些接口都是通过一系列可用的http方法进行定义。协议用于访问属于下列状态的资源：

* 客户端服务器Client-server
* 无状态的Stateless
* 可缓存的Cacheable
* 可分层的Layered

如果某个应用程序遵循主要的REST设计原则，那么这个应用程序就是RESTful的。play框架很容易创建RESTful风格的应用程序：

* Play router把所有的URI和HTTP方法都进行解析，并把请求路由一一对应到相应的java调用。基于正则表达式的URL范示使这个操作过程更灵活。
* 协议是stateless的，也就是说你不能在服务器上为两个连接的请求存储任何状态。
* Play把http当作关键特性进行考虑，因此play框架提供了让你完全访问http信息的能力。

**routes文件语法**

Router使用conf/routes文件作为配置文件。此文件列出了所有应用程序所需要的路由。每个路由都由一个HTTP方法+ URI 范示来表示一个Java调用。

示例：

GET /clients/{id} Clients.show

每个route开始于一个http方法，接着是一个URL范示，最后一个元素是java调用定义。

你可以使用#为路由添加一个注释

# Display a client

GET /clients/{id} Clients.show

**HTTP方法**

HTTP方法可以是以下几个http协议支持的任何方法：

* **GET**
* **POST**
* **PUT**
* **DELETE**
* **HEAD**

这些方法也支持WS（web service）作为action方法来指明一个[WebSocket](http://www.playframework.org/documentation/1.2.4/asynchronous#UsingWebSockets) 请求：

如果使用\*作为http方法，则route将匹配所有的http方法请求，如：

\* /clients/{id} Clients.show

路由将接受下面两个请求：

GET /clients/1541

PUT /clients/1212

**URI范示 Pattern**

URI范示定义了路由的请求路径。请求路径的某些部分可以是动态的。任何动态的部分都必须使用大括号进行界定{…}，如：

/clients/all

将精确匹配:

/clients/all

但…

/clients/{id}

可以匹配下面的请求:

/clients/12121

/clients/toto

一个URI范示可以有多个动态部分:

/clients/{id}/accounts/{accountId}

对于动态部分的默认匹配策略使用的是正则表达式 /[^/]+/，因此，我们可以为动态部分定义自己的正则表达式。

下面的正则表达式只能匹配数字值作为id:

/clients/{<[0-9]+>id}

下面的表达式只允许4至10个小写字母作为id：

/clients/{<[a-z]{4,10}>id}

在这里，可以使用任何可用的正则表达式。

**注意！**   
  
凡是被命名了的动态部分，控制器随后可以从http参数map里得到动态部分的值。

默认情况下，play会把URL后的反斜线/当作不同的值，比如：

GET /clients Clients.index

此路由只会匹配**/clients** URL路径，不会匹配**/clients/**。如果你打算匹配这两个URL路径，那么你需要在路径后增加一个/?作为结束标志，比如：

GET /clients/? Clients.index

注意：

在这里，除了反斜线外，URI范示不能含有任何可选部分。

**Java调用定义**

route定义的最后一部分就是java调用，通过使用action方法的完整名称来进行定义。action方法必须是控制类的public static void方法，而且这个控制器类必须定义于controllers包里，且是play.mvc.Controller的子类。

如果控制器类不在controllers包下，则需要在控制器类名称前添加java包名称。默认的controllers包可以直接使用，因此在此包下的控制器不需要单独指定包名。

GET /admin admin.Dashboard.index

**把404当作action来用**

你可以直接使用400作为路由action，用于标记那些应用程序必须忽略的URL路径，如忽略网站ico请求:

# Ignore favicon requests

GET /favicon.ico 404

**指派静态参数**

在某种情况下，你可能会重复使用一个存在的基于某些参数值的路由，如下：

public static void page(String id) {

Page page = Page.findById(id);

render(page);

}

相应的route:

GET /pages/{id} Application.page

现在，我打算为一个id为’hame’的页面定义一个URL别名，我可以定义一个具有静态参数的其他路由:

GET /home Application.page(id:'home')

GET /pages/{id} Application.page

当页面的id值为‘home’时，第一个路由与第二路由造价。但是，既然第一个具有更高优先级，那么这个路由将用作Application.page方法处理id为‘home’时的默认路由。

**变量和脚本**

在路由里，也可使用${…} 语法来在路由中使用变量，使用%{…}语法来在路由中使用脚本，和你在模板中使用变量和脚本的情况是一样的，比如：

%{ context = play.configuration.getProperty('context', '') }%

# Home page

GET ${context} Secure.login

GET ${context}/ Secure.login

另外一个例子就是在CRUD模块里的路由文件使用crud.types标签来循环调用所有的model types，以为每种类型生成控制器路由定义。

**路由优先级**

许多路由定义可能会匹配同一个请求，如果产生了冲突,那么只采用第一个路由（遵照下面的声明顺序）。

比如:

GET /clients/all Clients.listAll

GET /clients/{id} Clients.show

照此定义，则下面的URL:

/clients/all

将调用第一个路由指定的Clients.listAll方法(那怕第二个路由也同样匹配这个URL)。

**服务器静态资源**

**staticDir: mapping**

使用特定的action **staticDir，可以用来**定位每个你打算当作静态资源容器进行发布的文件夹。

比如：

GET /public/ staticDir:public

当提供的请求含有/public/\* 路径时，Play将把应用程序/public目录下的文件发布出去。

其优先权和标准的路由优先权相同。

**staticFile: mapping**

也可直接映射一个URL路径到一个静态文件。

# Serve index.html static file for home requests

GET /home staticFile:/public/html/index.html

**URL 编码**

由于不可能对URL解码或重新编码 (比如你并能确定URL里的斜线是否就是真正的斜线或%2F), URL应该明确进行编码。play默认使用UTF-8进行编码，但你也可以使用ISO-8859-1, UTF-16BE, UTF-16LE, UTF-16当成默认的WebEncoding配置参数。详见<http://download.oracle.com/javase/1.4.2/docs/api/java/nio/charset/Charset.html>。

比如:

映射/stéphane时可以使用如下方法：

GET /st%C3%A9phane Application.stephane

**反转路由：用于生成某些URL**

Router可以用于在一个java调用里生成一个URL。因此，你就可以把所有的URI范示集中到一个配置文件里，这样，在重构应用程序时，你就更有把握了。

比如，下面的路由定义：

GET /clients/{id} Clients.show

在代码里，同样可以生成URL，以便调用Clients.show:

map.put("id", 1541);

String url = Router.reverse("Clients.show", map).url;

//结果为 GET /clients/1541

URL生成器已经被集成到许多框架的组件里，你根本就不需要使用Router.reverse进行操作。

比如，如果你添加的参数并没有包含到URI范示里，那么这些参数将加到查询字符串里：

map.put("id", 1541);

map.put("display", "full");

//结果为：GET /clients/1541?display=full

String url = Router.reverse("Clients.show", map).url;

优先权顺序再次用于查找最特别的路由，以生成URL。

**设置内容风格(CSS)**

Play选择依照request.format的值来为http response选择合适的[media type](http://en.wikipedia.org/wiki/Internet_media_type)。 这个值通过文件扩展名确定哪个视图模板将会被使用，同时设置response的Content-type的值为Play的mime-types.properties文件映射格式确定的媒体类型。

play请求的默认格式为html。因此，index()控制器方法的默认模板就是index.html文件。如果指定了一个不同的格式，就需要选择对应格式的模板。

你可以在调用render方法前以编程的方式设定格式。比如，为了使用媒体类型为text/css的CSS，你可这样处理：

request.format = "css";

然而，更清晰的方法就是在rourtes文件里使用URL来指定格式。你可以通过为控制器方法指定格式来特定的路由添加格式，比如：下面的路由将处理来自/index.xml的请求，Application.index()方法将设置格式为xml，并使用index.xml模板进行渲染。

GET /index.xml Application.index(format:'xml')

类似的还有:

GET /stylesheets/dynamic\_css css.SiteCSS(format:'css')

在下面的路由里，Play也可直接从URL里提取格式：

GET /index.{format} Application.index

在这个路由里，/index.xml请求将设置格式为xml，同时使用XML模板进行渲染； /index.txt请求将使用原始的text模板进行渲染。

Play也可使用http内容协商自动设置格式。

**HTTP** **内容协商 negotiation**

play和其他RESTful架构一样，直接使用http提供的功能，而不是试着隐藏http或是在其之上放置一个抽象层。http的内容协商（[Content negotiation](http://en.wikipedia.org/wiki/Content_negotiation)）特性允许http服务器依照http客户端请求的媒体类型为同一个URL提供不同的媒体类型[media types](http://en.wikipedia.org/wiki/Internet_media_type)。客户端特定的可接受的内容类型是通过Accept header确定的，比如需要一个xml response就定义为：

Accept: application/xml

客户端可以指定多个媒体类型，并且可以指定（\*/\*）来表示可接受任意媒体类型：

Accept: application/xml, image/png, \*/\*

传统的Web浏览器总是在其Accept header里包含了通配符：这样，他们就可以接收任意媒体类型，当然包括play提供的默认html类型。内容协商更多是用于定制的客户端，比如一个要求返回JSON的Ajax请求，或一个e-book阅读器需要的PDF或EPUB版本的文档。

**从http headers开始设置内容类型**

如果Accept header包含了text/html或application/xhtml以及作为\*/\*通配符的结果时, Play选择其默认格式html。如果通配符值为空时，默认格式将不被选择。

Play对html, txt, json和 xml媒体格式提供了内建支持。比如，下面定义了一个用于渲染某些数据的控制器方法：

public static void index() {

final String name = "Peter Hilton";

final String organisation = "Lunatech Research";

final String url = "http://www.lunatech-research.com/";

render(name, organisation, url);

}

在一个浏览器里，如果请求的URL映射到这个方法时，那么play将渲染index.html模板，因为浏览器发送的Accept header里包含了text/html值。

通过设置请求格式为xml，Play响应的结果Accept header类型为：text/xml，同时使用index.xml模块进行渲染，比如：

<?xml version="1.0"?>

<contact>

<name>${name}</name>

<organisation>${organisation}</organisation>

<url>${url}</url>

</contact>

Accept header内建的格式对应的格式以及对应的模板文件见下表（以index()控制器方法为例）：

|  |  |  |  |
| --- | --- | --- | --- |
| **Accept header** | **Format** | **Template file name** | **Mapping** |
| null | null | index.html | Default template extension for null format |
| image/png | null | index.html | 媒体类型没有映射到格式 |
| \*/\*, image/png | html | index.html | 默认媒体类型映射到html格式 |
| text/html | html | index.html | Built-in format |
| application/xhtml | html | index.html | Built-in format |
| text/xml | xml | index.xml | Built-in format |
| application/xml | xml | index.xml | Built-in format |
| text/plain | txt | index.txt | Built-in format |
| text/javascript | json | index.json | Built-in format |
| application/json, \*/\* | json | index.json | Built-in format, default media type ignored |

**定制格式**

通过检查请求header和设置相应用的格式，可以为自己的自定义类型添加内容协商，因此只需在http请求选择相应的媒体类型时设置这个自定义格式即可。比如，为了在控制器里实现带有text/x-vcard的[vCard](http://en.wikipedia.org/wiki/Vcard)功能，需要所有请求处理之前对定制格式进行检查：

@Before

static void setFormat() {

if (request.headers.get("accept").value().equals("text/x-vcard")) {

request.format = "vcf";

}

}

现在，一个Accept: text/x-vcard header请求将会渲染index.vcf模板，比如：

BEGIN:VCARD

VERSION:3.0

N:${name}

FN:${name}

ORG:${organisation}

URL:${url}

END:VCARD

**继续讨论**  
  
当Router明确了哪个java调用将用于处理已经接收的http请求，play就会调用这个java调用。见[Controllers](http://www.playframework.org/documentation/1.2.4/controllers) 节，以了解控制器是如何工作的。

**03.控制器(http传输层和模型对象之间的粘合剂)**

在Play框架中，商业逻辑在domain model层里进行管理，Web客户端不能直接调用这些代码，domain对象的功能作为URI资源暴露出来。

客户端使用HTTP协议提供的统一API来暗中操作这些底层的商业逻辑实现资源的维护。然而，这些domain对象到资源的映射并不是双向注入的：它可以表示成不同级别的粒度，一些资源可能是虚拟的，某些资源的别名或许已经定义了…

这正是控制器层发挥的作用：它在模型对象和传输层事件之间提供了粘合代码。和模型层一样，控制器也是纯java书写的代码，这样控制器层就很容易访问和修改model对象。和http接口类似，控制器是一个面向Request/Response的程序。

控制器层减少了http和模型层之间的阻抗。

**注意！**  
  
不同的策略具有不同的架构模型。一些协议可以让你直接访问模型对象。典型代表就是EJB和Corba协议，这种架构风格使用的是RPC(远程过程调用)，这样的通信风格和web架构很难兼容。  
SOAP协议则试着通过web访问model对象。这是另外一个rpc类型的协议，这种情况，soap使用http作为传输协议，这不是一种应用程序协议。

由于web规则并不是完全面向对象的，所以在这些协议下，针对不同的语言，需要不同的http适配器。

**控制器概览**

一个controller就是一个java类，位于controller包下，是play.mvc.Controller的一个子类。

示例：

package controllers;

import models.Client;

import play.mvc.Controller;

public class Clients extends Controller {

public static void show(Long id) {

Client client = Client.findById(id);

render(client);

}

public static void delete(Long id) {

Client client = Client.findById(id);

client.delete();

}

}

控制器中的每个public、static方法叫做Action（动作）。action动作方法签名总是如下：

public static void action\_name(params...);

在action方法签名里可以定义参数，这些参数值会被框架自动从相应的http参数里找到。

通常情况下，一个action方法不包括return字段，方法退出是通过调用result方法完成的，在上面的示例里，render()就是一个result方法，用于执行和显示一个模板。

**获取http参数**

一个HTTP请求包含有数据。这些数据可以从如下渠道提取：

* URI路径：比如/clients/1541请求, 1541就是URI范示的动态部分
* 请求字符串：/clients?id=1541
* 请求体：如果是通过html窗体发送的请求，请求体就包含有以x-www-urlform-encoded方式编码的窗体数据

所有这些情况下，play都会自动进行数据提取，并存入同一个Map<String, String[]> ，这里面包含有所有的HTTP参数。key就是参数名name，具体从以下方式确定：

* URI范示的动态部分名称（和在routes文件里指定的一样）
* 查询字符串里的name-value pair中的name
* x-www-urlform-encoded体的内容

**使用params map**

params对象是一个可用于任何控制器类的变量（在play.mvc.Controller超类中定义的），这个对象包含了所有从当前http请求找到的参数。

比如:

public static void show() {

String id = params.get("id");

String[] names = params.getAll("names");

}

你也可以让Play帮助完成类型转换:

public static void show() {

Long id = params.get("id", Long.class);

}

请等一等，你有更好的方式完成类型转换，如下：

**还可以从action方法签名实现转换**

你可以直接从action方法签名里取回http参数。但Java参数的名称必须和http参数的名称相同：

比如下面的请求:

/clients?id=1451

一个action方法可以通过在其方法签名里声明一个id参数来取回id参数值：

public static void show(String id) {

System.out.println(id);

}

还可以使用其他java类型，比如String。在这种情况下，框架将试着预测参数值的正确类型：

public static void show(Long id) {

System.out.println(id);

}

如果某个参数具有多个值，则可以声明一个数组参数：

public static void show(Long[] id) {

for(String anId : id) {

System.out.println(anid);

}

}

或是一个集合类型：

public static void show(List<Long> id) {

for(String anId : id) {

System.out.println(anid);

}

}

**例外情况！**  
  
如果在action方法参数里找不到http参数对应的参数，则相应的方法参数将默认设置为默认值（对象类型为null，数字类型为0）。如果能够在action方法参数里找到对应参数，但框架不能预测需要的java类型， play将在validation error集合里增加一个错误，并使用默认值。

**高级HTTP Java绑定**

**简单类型**

所有本地的和通用的java类型都是自动进行绑定的：

int, long, boolean, char, byte, float, double, Integer, Long, Boolean, Char, String, Byte, Float, Double

注意：如果http请求的参数丢失或自动类型转换失败，对象类型将置null，简单类型将设置为其默认值，类型预测失败还要在validation.error集合中增加一个错误。

**Date类型**

如果日期的字符串输出匹配下面的范示，，日期对象将进行自动绑定：

* yyyy-MM-dd’T’hh:mm:ss’Z' // ISO8601 + 时区
* yyyy-MM-dd’T’hh:mm:ss" // ISO8601
* yyyy-MM-dd
* yyyyMMdd’T’hhmmss
* yyyyMMddhhmmss
* dd'/‘MM’/'yyyy
* dd-MM-yyyy
* ddMMyyyy
* MMddyy
* MM-dd-yy
* MM'/‘dd’/'yy

使用@As注释，你可以指定日期格式：

archives?from=21/12/1980

public static void articlesSince(@As("dd/MM/yyyy") Date from) {

List<Article> articles = Article.findBy("date >= ?", from);

render(articles);

}

你也可以依照对应的语言调整日期格式：

public static void articlesSince(@As(lang={"fr,de","\*"},

value={"dd-MM-yyyy","MM-dd-yyyy"}) Date from) {

List<Article> articles = Article.findBy("date >= ?", from);

render(articles);

}

在这个示例里，我们假定法语和德语的日期格式为dd-MM-yyyy，其他语言的日期格式为MM-dd-yyyy。请注意lang和value可以用逗号进行分隔。最为重要的是lang的数字型参数匹配的是value的数字型参数。

如果没有指定@As注释，那么play将依照访问者的时区使用默认的日期格式。[date.format configuration](http://www.playframework.org/documentation/1.2.4/configuration#date.format)可以指定默认的日期格式。

**Calendar日历**

日历可以精确与日期进行绑定，除非play依照你的时区来选择Calendar对象，还可以使用@Bind注释。

**File**

在play里，实现文件上传非常简单。首先使用一个multipart/form-data编码格式的请求来传送文件到服务器，然后使用ava.io.File类型来取回上传的文件对象：

public static void create(String comment, File attachment) {

String s3Key = S3.post(attachment);

Document doc = new Document(comment, s3Key);

doc.save();

show(doc.id);

}

上传到服务器的文件名称和原始文件名称一致。首先，上传的文件会暂时保存在服务器的tmp临时目录，当请求结束时，该文件将被删除。因此，你必须把这个文件复制到安全的目录。

上传文件的MIME类型，通常情况下在http请求的Content-type header里已经 指定。然而，当从一个Web浏览器上传文件时，一些不常见的文件可能不会为其指定MIME类型。在这种情况，你可以手工使用play.libs.MimeTypes类映射文件的扩展名到某个MIME类型。

String mimeType = MimeTypes.getContentType(attachment.getName());

play.libs.MimeTypes类将查找$PLAY\_HOME/framework/src/play/libs/mime-types.properties里设定的文件扩展名来得到MIME类型。

使用[Custom MIME types configuration](http://www.playframework.org/documentation/1.2.4/configuration#mimetype)配置，你也可添加你自己的MIME类型。

**支持类型的数组或集合**

所有支持类型都可以当作对象的集合或数组取回：

public static void show(Long[] id) {

…

}

或:

public static void show(List<Long> id) {

…

}

或:

public static void show(Set<Long> id) {

…

}

Play也可以处理特殊的Map<String, String>绑定，比如：

public static void show(Map<String, String> client) {

…

}

下面这个查询字符串:

?client.name=John&client.phone=111-1111&client.phone=222-2222

将绑定client变量到一个带有两个元素的map。第一个元素为name:John(key/value)，第二个元素为phone: 111-1111, 222-2222(同一个key,两个值)。

**POJO对象绑定**

使用同样的命名转换规则，Play也可自动对任何model类进行绑定。

public static void create(Client client ) {

client.save();

show(client);

}

使用上面这个create方法，创建一个client的查询字符串可以是下面这个样式：

?client.name=Zenexity&client.email=contact@zenexity.fr

Play将创建一个Client实例，同时把从http参数里取得的值赋值给对象中与http参数名同名的属性。不能明确的参数将被安全忽略，类型不匹配的也会被安全忽略。

参数绑定是通过递归来实现的，也就是说你可以采用如下的查询字符串来编辑一个完整的对象图（object graphs）：

?client.name=Zenexity

&client.address.street=64+rue+taitbout

&client.address.zip=75009

&client.address.country=France

使用数组标记（即[]）来引用对象的id，可以更新一列模型对象。比如，假设Client模型有一列Customer模型声明作为List Customer customers。为了更新这列Customers对旬，你需要提供如下查询字符串：

?client.customers[0].id=123

&client.customers[1].id=456

&client.customers[2].id=789

**JPA 对象绑定**

使用http到java的绑定，可以自动绑定一个JPA对象。

比如，在http参数里提供了user.id字段，当play找到id字段里，play将从数据库加载匹配的实例进行编辑，随后会自动把其他通过http请求提供的参数应用到实例里，因此在这里可以直接使用save()方法，如下：

public static void save(User user) {

user.save(); // ok with 1.0.1

}

和在POJO映射里采取的方式相同，你可以使用JPA绑定来编辑完整的对象图（object graphs）,唯一区别是你必须为每个打算编辑的子对象提供id:

user.id = 1

&user.name=morten

&user.address.id=34

&user.address.street=MyStreet

**定制绑定**

绑定系统现在支持更多的定制化。

**@play.data.binding.As**

首先是@play.data.binding.As注释，这个注释使上下方相关的配置进行绑定成为可能。比如，你可以使用这个注释来明确指定日期必须使用DateBinder进行格式化:

public static void update(@As("dd/MM/yyyy") Date updatedAt) {

…

}

@As注释也提供了国际化支持，也就是说你可以为每个时区提供一个特定的注释：

public static void update(

@As(

lang={"fr,de","en","\*"},

value={"dd/MM/yyyy","dd-MM-yyyy","MM-dd-yy"}

)

Date updatedAt

) {

…

}

@As可以和所有支持它的绑定一起工作，包含你自己的绑定，比如使用ListBinder:

public static void update(@As(",") List<String> items) {

…

}

这个绑定简单使用逗号分隔List里的String。

**@play.data.binding.NoBinding**

@play.data.binding.NoBinding注释允许你标记一个非绑定字段，以解决潜在的安全问题。比如：

public class User extends Model {

@NoBinding("profile") public boolean isAdmin;

@As("dd, MM yyyy") Date birthDate;

public String name;

}

public static void editProfile(@As("profile") User user) {

…

}

在这种情况下， isAdmin字段绝不会被editProfile action绑定, 即使某个恶意用户在伪造的窗体post里写入了user.isAdmin=true代码。

**play.data.binding.TypeBinder**

**@As**注释也允许你定制一个绑定。一个定制绑定必须是TypeBinder接口的子类，比如：

public class MyCustomStringBinder implements TypeBinder<String> {

public Object bind(String name, Annotation[] anns, String value,

Class clazz) {

return "!!" + value + "!!";

}

}

这样，你就可以在任何action里使用这个绑定了，比如：

public static void anyAction(@As(binder=**MyCustomStringBinder.class**)

String name) {

…

}

**@play.data.binding.Global**

作为选择，你可以定义一个全局性的定制绑定，这样的绑定将应用于相应的类型。比如，为java.awt.Point类定制了一个绑定，如下：

@Global

public class PointBinder implements TypeBinder<Point> {

public Object bind(String name, Annotation[] anns, String value,

Class class) {

String[] values = value.split(",");

return new Point(

Integer.parseInt(values[0]),

Integer.parseInt(values[1])

);

}

}

正如你所看到的，一个全局性绑定就是一个使用**@play.data.binding.Global**进行注释的传统绑定。通过这种方式，一个外部模块可以将其定制的绑定应用到一个项目里，这就为定义一个可重用的绑定扩展提供了方法。

**结果类型**

一个action方法必须生成一个http response响应，最简便的方法就是放出一个结果Result对象。当一个Result对象被放出时，常规的执行流程被中断，方法退出。比如:

public static void show(Long id) {

Client client = Client.findById(id);

render(client);

System.out.println("这个消息永远不会显示~! ");

}

render(…)方法放出一个Result对象，并停止方法执行。

**返回一些文本类型的内容**

renderText(…)方法放出一个简单的Result事件，只在底层的http Response里写入了一些简单的文本，比如：

public static void countUnreadMessages() {

Integer unreadMessages = MessagesBox.countUnreadMessages();

**renderText**(unreadMessages);

}

你也可使用java标准的格式化语法来格式化输出文本消息:

public static void countUnreadMessages() {

Integer unreadMessages = MessagesBox.countUnreadMessages();

renderText("There are **%s** unread messages", **unreadMessages**);

}

**返回一个JSON字符串**

Play使用renderJSON(…)方法来返回一个JSON字符串。这个方法的作用是设置响应内容为application/json，同时返回一个JSON字符串。

你可以指定你自己的JSON字符串，或把这个字符串传递到一个通过GsonBuilder进行串行化的对象里，比如：

public static void countUnreadMessages() {

Integer unreadMessages = MessagesBox.countUnreadMessages();

renderJSON("{\"messages\": " + unreadMessages +"}");

}

当然，如果面对一个更加复杂的对象结构，你可能会希望使用GsonBuilder来创建这个JSON字符串。

public static void getUnreadMessages() {

List<Message> unreadMessages = MessagesBox.unreadMessages();

renderJSON(unreadMessages);

}

当传递一个对象到rndoerJSON(…)方法时，如果你需要对JSON构建器进行更多控制，那么你可以把JSON串行化并把对象输入到定制的输出里。

**返回一个XML字符串**

和JSON方法一样，这里有几个可以直接从控制器渲染XML的方法。**renderXml**(…) 方法返回一个内容类型设置为text/xml的XML字符串。

在这里，你可指定自己的xml字符串，传递一个org.w3c.dom.Document对象，或传递一个将被XStream串行化的POJO对象，比如：

public static void countUnreadMessages() {

Integer unreadMessages = MessagesBox.countUnreadMessages();

renderXml("<unreadmessages>"+unreadMessages+"</unreadmessages>");

}

当然，你也可以使用org.w3c.dom.Document对象：

public static void getUnreadMessages() {

Document unreadMessages = MessagesBox.unreadMessagesXML();

renderXml(unreadMessages);

}

**返回二进制内容**

向用户返回一个存储在服务器上的二进制文件需要使用renderBinary()方法。比如你有一个User对象，并且有一个play.db.jpa.Blob的图片属性。这时，就可以使用如下语句在一个控制器方法里加载这个模型对象，并使用对应的MIME类型渲染这个对象里的图片：

public static void userPhoto(long id) {

final User user = User.findById(id);

response.setContentTypeIfNotSet(user.photo.type());

java.io.InputStream binaryData = user.photo.get();

renderBinary(binaryData);

}

**作为附件下载文件**

通过设置http header，可以指示web浏览器把一个二进制响应当作“附件”来对待，这样就可以在web浏览器把文件下载到用户的电脑。为了完成这个任务，可以把一个文件的名称传递给renderBinary方法，并作为它的参数，这时play会自动设置Content-Disposition响应header。比如，当上面示例里的User模型作为一个photoFileName属性时：

renderBinary(binaryData, user.photoFileName);

**执行一个模板**

如果生成的内容，你应该使用模板来生成response内容。

public class Clients extends Controller {

public static void index() {

render();

}

}

模板名称且根据play约定自动进行推断。默认模板路径就是控制器和Action的名称。

在上面这个示例里调用的模板路径和名称如下:

app/views/Clients/index.html

**向模板作用域添加数据**

通常情况下，模板是需要数据的。你可以使用renderArgs对象添加数据到模板作用域：

public class Clients extends Controller {

public static void show(Long id) {

Client client = Client.findById(id);

**renderArgs**.put("client", client);

render();

}

}

在模板执行期间，client变量将被定义。

比如：

<h1>Client ${client.name}</h1>

**更加简洁的形式添加数据到模板作用域**

使用render(…)方法的参数，可以把数据直接传递到模板：

public static void show(Long id) {

Client client = Client.findById(id);

render(client);

}

在这种情况下，模板可以访问的变量与java局部变量的名称完全相同。当然，还可通过这种方式传递多个变量:

public static void show(Long id) {

Client client = Client.findById(id);

render(id, client);

}

**非常重要!**  
  
你只能通过这种方式向模板传递**局部变量**。

**指定其他模板**

如果不想使用默认的模板，可以使用**renderTemplate**(…)方法指定自己的模板文件，但模板名称要作为第一个参数:

如：

public static void show(Long id) {

Client client = Client.findById(id);

renderTemplate("**Clients/showClient.html**", id, client);

}

**跳转到其他URL**

redirect(…)方法放出一个跳转事件， 用于切换生成一个http跳转响应。

public static void index() {

redirect("http://www.zenexity.fr");

}

**Action链**

play和Servlet API的forward不同，一个http请求只能调用一个action。如果你想要调用其他的action，就必须跳转浏览器到能够调用这个action的URL。通过这种方式，浏览器URL就总是和被执行的Action保持一致，因此对浏览器的back/forward/refresh管理就非常容易。

在java代码里，通过调用其他action方法，就可以实现发送一个跳转响应到任何action里，比如：

public class Clients extends Controller {

public static void show(Long id) {

Client client = Client.findById(id);

render(client);

}

public static void create(String name) {

Client client = new Client(name);

client.save();

show(client.id);

}

}

上面的示例在使用下面这两条路由的情况下:

GET /clients/{id} Clients.show

POST /clients Clients.create

* 浏览器发送一个POST到/clients URL
* 路由调用Clients控制器的create action
* action方法直接调用show action方法
* Java调用被中断，反向路由生成器创建一个带id参数的Clients.show方法调用的URL
* HTTP响应为： 302 Location:/clients/3132
* 浏览器随后发布GET /clients/3132
* …

**定制web编码**

play着重使用UTF-8编码，但某些情况下必须使用其他编码。

**为当前response定制编码**

要为当前response改变编码，可以控制器里参照下面的方法进行定制：

**response.encoding** = "ISO-8859-1";

当传递一个与服务器默认编码不同的窗体时，你应该在窗体里使用encoding/charset两次，两次都用在accept-charset属性指定上， 而且是用在一个名称叫\_charset\_的特殊隐藏窗体上。accept-charset属性会告诉浏览器在传递窗体里要使用哪种编码，form-field \_charset\_ 则告诉play需要采用哪个编码进行处理：

<form action="@{application.index}" method="POST" accept-charset="ISO-8859-1">

<input type="**hidden**" **name="\_charset\_"** value="ISO-8859-1">

</form>

**为整个应用程序定制编码**

配置[application.web\_encoding](http://www.playframework.org/documentation/1.2.4/configuration#application.web_encoding) 用于指定play与浏览器进行通信时需要采用哪种编码。

**拦截器**

在控制器里，可以定义拦截器方法。拦截器将被控制器类及其后代的所有action调用。可以利用这个特点为所有的action定义一些通用的提前处理代码：比如对用户进行验证、加载请求作用域信息等…

这些方法必须是static的，但不能是public的，并且使用有效的拦截注释：

**@Before**

用@Before注释的方法将在这个控制器的每个action被调用前执行。因此，可以利用这个注释创建一个安全检查方法：

public class Admin extends Controller {

@Before

static void checkAuthentification() {

if(**session**.get("user") == null) login();

}

public static void index() {

List<User> users = User.findAll();

render(users);

}

…

}

如果不希望@Before方法中断所有的action调用，可以指定一个需要排除的Action列表：

public class Admin extends Controller {

@Before(**unless**="login")

static void checkAuthentification() {

if(session.get("user") == null) login();

}

public static void index() {

List<User> users = User.findAll();

render(users);

}

…

}

如果希望@Before方法中断列表中的action调用，可以使用only参数：

public class Admin extends Controller {

@Before(**only**={"login","logout"})

static void doSomething() {

…

}

…

}

**unless**和**only** 参数也可用于@After, @Before和@Finally注释。

**@After**

用@After注释的方法将在每个action调用执行完后执行。

public class Admin extends Controller {

@After

static void log() {

**Logger**.info("Action executed ...");

}

public static void index() {

List<User> users = User.findAll();

render(users);

}

…

}

**@Catch**

用@Catch注释的方法将在其他action方法抛出指定异常时执行。这个被抛出的异常将作为参数传递给@Catch注释的方法。

public class Admin extends Controller {

@Catch(IllegalStateException.class)

public static void logIllegalState(Throwable throwable) {

Logger.error("Illegal state %s…", throwable);

}

public static void index() {

List<User> users = User.findAll();

if (users.size() == 0) {

**throw** new IllegalStateException("Invalid database - 0 users");

}

render(users);

}

}

和普通的java异常处理一样，你可以捕获一个超类来捕获更多的异常类型。如果不只一个捕获方法，可以指定他们的**priority**（优先级）参数，以确定执行顺序（1是最高优先级）。

public class Admin extends Controller {

@Catch(value = Throwable.class, priority = 1)

public static void logThrowable(Throwable throwable) {

// Custom error logging…

Logger.error("EXCEPTION %s", throwable);

}

@Catch(value = IllegalStateException.class, priority = 2)

public static void logIllegalState(Throwable throwable) {

Logger.error("Illegal state %s…", throwable);

}

public static void index() {

List<User> users = User.findAll();

if(users.size() == 0) {

throw new IllegalStateException("Invalid database - 0 users");

}

render(users);

}

}

**@Finally**

用@Finally注释的方法总是在这个控制器里的每个action调用执行后被执行。用@Finally注释的方法，不管action调用是否成功也会执行。

public class Admin extends Controller {

@Finally

static void log() {

Logger.info("Response contains : " + response.out);

}

public static void index() {

List<User> users = User.findAll();

render(users);

}

…

}

如果 @Finally注释的方法带有一个Throwable类型的参数，那么有异常发生时，异常将会传送到方法里面来。

public class Admin extends Controller {

@Finally

static void log(Throwable e) {

if( e == null ){

Logger.info("action call was successful");

} else{

Logger.info("action call failed", e);

}

}

public static void index() {

List<User> users = User.findAll();

render(users);

}

…

}

**控制器继承**

如果一个控制器类是其他控制器类的子类，那么拦截器也会按照继承顺序应用于相应层级的子类。

**使用@With注释添加更多的拦截器**

Because Java does not allow multiple inheritance, it can be very limiting to rely on the Controller hierarchy to apply interceptors. But you can define some interceptors in a totally different class, and link them with any controller using the @With annotation.由于java不允许多继承，通过控制器继承特点来应用拦截器就受到极大的限制。但是我们可以在一个完全不同的类里定义一些拦截器，然后在任何控制器里使用@With注释来链接他们。

比如：

public class Secure extends Controller {

@Before

static void checkAuthenticated() {

if(!session.containsKey("user")) {

unAuthorized();

}

}

}

另一个控制器:

@With(Secure.class)

public class Admin extends Controller {

…

}

**Session和Flash作用域**

为了在多个http请求间保存数据，你可以把这些数据存入Session或Flash作用域。**存储在Session里的数据在整个用户session期间是可用的，存储在Flash里的数据只在下一个请求里可用。**

明白把session和flash数据通过Cookie机制存储在每个请求客户端电脑里而不是存储在服务器上这点很重要，这就确定了数据大小不能超过最多4kb，而且只能存储字符串类型的值。

当然，play已经为cookie指派了一个安全key,因此客户端不能编辑cookie数据（否则cookie将无效）。因此play框架不打算把session用作缓存，如果你需要缓存一些与session相关的数据，你可以使用Play内建的缓存机制，并用session.getId()key来保存某个特定用户session相关的数据，比如：

public static void index() {

List messages = **Cache**.get(session.getId() + "-messages", List.class);

if(messages == null) {

// Cache miss

messages = Message.findByUser(session.get("user"));

Cache.set(session.getId() + "-messages", messages, "30mn");

}

render(messages);

}

当你关闭浏览器里，session就过期了。除非对[application.session.maxAge](http://www.playframework.org/documentation/1.2.4/configuration#application.session.maxAge)进行了配置。

play里的缓存和传统的Servlet HTTP session对象的概念完全不同。千万不要认为这些缓存了的对象会永远在缓存里。因此，play会强迫你去处理缓存丢失的情况，以防止缓存数据丢失，这样就保证了应用程序是完全无状态的。

**04.模板引擎**

Play有一个高效的模板系统，它允许动态生成html、xml、json或其他文本格式的文档。Play的模板引擎使用Groovy作为表达式语言。它的标签系统允许你创建一些可以重复使用的功能。

模板默认存储在app/views目录下。

**模板语法**

模板文件是一个文本文件，其中的一些占位符用于动态生成内容。模板的动态元素是用groovy语言写的。Groovy语法非常接近java的语法。

动态元素在模板执行期间被提取出来，最后以http response方式返回给客户端。

**Expressions: ${…}**

要创建一个动态元素，最简单的方法就是声明一个表达式。语法为${…}，表达式的最终结果将被插入在表达式使用的地方。

如:

<h1>Client ${client.name}</h1>

如果不确定client对象是否为null，则可以使用如下的groovy语法:

<h1>Client ${client?.name}</h1>

如果client不为null，则显示，否则不显示。

**Template decorators : #{extends /} and #{doLayout /}**

Decorators(装饰？母版)提供了一个清晰的解决方案来在多个不同的模板间共享同一个页面布局（或设计）。

使用[#{get}](http://www.playframework.org/documentation/1.2.4/tags#ageta) 和 [#{set}](http://www.playframework.org/documentation/1.2.4/tags#aseta)标签来在模板和decorator（母版）之间共享变量。

在decorator（母版）里嵌入一个页面就是一行代码的事：

#{extends 'simpledesign.html' /}

#{set title:'A decorated page' /}

This content will be decorated.

decorator（母版）文件simpledesign.html的代码为：

<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">

<head>

<title>#{get 'title' /}</title>

<link rel="stylesheet" type="text/css" href="@{'/public/stylesheets/main.css'}" />

</head>

<body>

<h1>#{get 'title' /}</h1>

#{doLayout /}

<div class="footer">Built with the play! framework</div>

</body>

</html>

**Tags: #{tagName /}**

一个tag就是一个可以带参数的模板碎片，如果只有一个参数，则默认的参数名称就是arg，而且arg可以省略。

例如，如下的tag用于插入一个javascript脚本:

#{script 'jquery.js' /}

tag必须是关闭的:

#{script 'jquery.js' /}

或

#{script 'jquery.js'}

#{/script}

例如，list标签用于迭代所有的集合元素，它带有两个必须的参数items和as:

<h1>Client ${client.name}</h1>

<ul>

#{list items:client.accounts, as:'account' }

<li>${account}</li>

#{/list}

</ul>

所有的动态表达式将被模板引擎转义以避免XSS安全问题（XSS:跨站脚本攻击），因此下列的title变量将被转义:

${title} --> &lt;h1&gt;Title&lt;/h1&gt;

如果不想转义，可以明确调用raw()方法:

${title.raw()} --> <h1>Title</h1>

当然，如果你想显示一个较多内容的raw html，你可以使用#{verbatim /}标签:

#{verbatim}

${title} --> <h1>Title</h1>

#{/verbatim}

**Actions: @{…} or @@{…}**

可以使用Route来反向找到URL相应的特定route配置，在模板里，可以使用@{…}语法来完成这个任务，主要用于生成URL链接。

示例:

<h1>Client ${client.name}</h1>

<p>

<a href="@{Clients.showAccounts(client.id)}">All accounts</a>

</p>

<hr />

<a href="@{Clients.index()}">Back</a>

//得到index()方法的链接

@@{…}语法与@{…}相似，不过它生成的绝对URL路径，对email很实用。

**Messages: &{…}**

&{…}语法可用于国际化支持，以显示不同语言的消息：

例如在conf/messages目录下的文件里，我们可以这样定义:

clientName=The client name is %s

在模板里使用时:

<h1>&{'clientName', client.name}</h1>

**Comment: \*{…}\***

注释部分将被模板引擎忽略：

\*{\*\*\*\* Display the user name \*\*\*\*}\*

<div class="name">

${user.name}

</div>

**Scripts: %{…}%**

脚本是一段复杂的表达式集合，在脚本内可以声明变量和定义一些语句，play使用%{…}%来插入一段脚本。

%{

fullName = client.name.toUpperCase()+' '+client.forname;

}%

<h1>Client ${fullName}</h1>

在脚本里可以使用out对象直接输出动态内容：

%{

fullName = client.name.toUpperCase()+' '+client.forname;

out.print('<h1>'+fullName+'</h1>');

}%

在脚本里也可创建一个结构，比如用于迭代：

<h1>Client ${client.name}</h1>

<ul>

%{

for(account in client.accounts) {

}%

<li>${account}</li>

%{

}

}%

</ul>

请记住，模板不是一个放置复杂代码的好地方。因此，请使用tag代替或放置在在controller或model对象里。

**Template inheritance继承**

模板可以被继承，比如模板被作为另外一个模板的一部分而被包含进去。

要继承另一个模板，请使用#{extends ‘…’}:

#{extends 'main.html' /}

<h1>Some code</h1>

main.html模板是一个标准模板，它使用doLayout标签来包含本模板的内容：

<h1>Main template</h1>

<div id="content">

#{doLayout /}

</div>

**定制模板标签**

一个tag就是一个模板文件，存储于app/views/tags目录下，模板文件名将用作tag名称。

例如，创建一个hello标签，只需在app/views/tags/目录下创建一个hello.html文件，其内容如下：

Hello from tag!

不需要任何配置，你就可以直接使用hello标签了:

#{hello /}

**检索tag参数**

tag参数默认被暴露为模板变量，变量名称带有‘\_’字符前缀。

例如:

Hello ${\_name} !

现在你就可以给name变量传递参数了:

#{hello name:'Bob' /}

如果你的tag只有一个参数，默认的参数名称是arg，而且可以忽略。

例如:

Hello ${\_arg}!

使用更简单:

#{hello 'Bob' /}

**调用标签体**

如果标签支持body，使用doBody标签，你就能在标签代码里的任何位置包含它。

比如：

Hello #{doBody /}!

然后你可使用这个名字作为标签体：

#{hello}

Bob

#{/hello}

**格式化特定标签**

你可以为不同的内容类型创建多个tag版本，play将自动选择适当的Tag。例如，play将使用app/views/tags/hello.html来处理request.format为html的内容，使用app/views/tags/hello.xml来处理格式为xml的请求。

如果内容格式不匹配， play将返回一个app/views/tags/hello.tag的处理结果。

**定制java标签**

你也可以使用java代码来定义一个定制tags。这和继承自play.templates.JavaExtensions类的JavaExtensions类相似，创建一个FastTag时，需要在类里创建一个方法，并且继承自play.templates.FastTags。每一个要当作tag来执行的方法都必须遵循如下的方法签名：

public static void \_tagName(Map<?, ?> args, Closure body, PrintWriter out, ExecutableTemplate template, int fromLine)

注意：tag名称之前必须要有下划线”\_”。

为了理解如何创建一个真正的tag，让我们看一下两个内建的tags。

比如，verbatim标签仅通过简单调用JavaExtensions的toString()方法来实现的，并且被传递给tag的body。

public static void \_verbatim(Map<?, ?> args, Closure body, PrintWriter out, ExecutableTemplate template, int fromLine) {

out.println(JavaExtensions.toString(body));

}

tag的body一定是在一对tag之间的（一开一关），即

<verbatim>My verbatim</verbatim>

Body的值应该是

My verbatim

第二个示例是option标签，要复杂一点，因为这个标签依赖于其父标签来实现相关功能。

public static void \_option(Map<?, ?> args, Closure body, PrintWriter out,

ExecutableTemplate template, int fromLine) {

Object value = args.get("arg");

Object selection = TagContext.parent("select").data.get("selected");

boolean selected = selection != null && value != null

&& selection.equals(value);

out.print("<option value=\"" + (value == null ? "" : value) + "\" "

+ (selected ? "selected=\"selected\"" : "")

+ "" + serialize(args, "selected", "value") + ">");

out.println(JavaExtensions.toString(body));

out.print("</option>");

}

这些代码将输出一个标准的html option标签，并依据父标签的值设置为相应的option为selected，前三行用于设置用于输出的变量。最后三行输出tag的结果。

在[FastTags.java in github](https://github.com/playframework/play/blob/master/framework/src/play/templates/FastTags.java)里有不同难度的标签示例。

**标签命名空间**

请检查你的tag在项目或与play标签之间没有冲突，你可以使用类级别的注释@FastTags.Namespace来设置命名空间。

因此，对hello标签来说，可以如下设置(空间名称为my.tags)

@FastTags.Namespace("my.tags")

public class MyFastTag extends FastTags {

public static void \_hello (Map<?, ?> args, Closure body, PrintWriter out,

ExecutableTemplate template, int fromLine) {

...

}

}

之后在模板里用如下方式进行调用。

#{my.tags.hello/}

**在模板里的Java对象扩展**

当你在模板引擎里使用一个java对象时，play为其加入了一些新的方法，这些方法在原始的java类里并不存在，而是被模板引擎动态加入到模板里的,play对一些基本的java原始类进行了扩展。

例如，为了方便在模板里对数字进行格式化，play在java.lang.Number里增加了一个format方法：

之后，我们在模板里就很容易对一个数字进行格式化：

<ul>

#{list items:products, as:'product'}

<li>${product.name}. Price: ${product.price.format('## ###,00')} €</li>

#{/list}

</ul>

同样可用于java.util.Date

具体的内容见[Java extensions](http://www.playframework.org/documentation/1.2.4/javaextensions)，它列出了一些扩展了的方法，如下：

对集合的扩展：

join(‘/’) 用/进行连接后返回

last() 返回最后一个元素

pluralize() 是否大于1个，大于则返回一个字符串：集合名+s

pluralize(plural) 是否大于1个，大于则返回一个字符串：集合名+plural指定的字符串，如es

pluralize(singular,plural) 为1则返回集合名+singular指定的字符串，不为1则返回集合名+plural指定的字符串

对Date的扩展：

Format(foramt) 返回格式化后的日期，如new Date().format(‘dd MMMM yyyy))

Format(format,language) 带语言，如new Date().format(‘dd MM yy’,’en’)

Since()

Since(condition)

另外还对Long、Map、Number、对象、String、String数组等类型进行扩展。

**创建定制扩展**

只需要创建一个继承了play.templates.JavaExtensions的类即可。

例如，创建一个定制数字格式化的方法:

package ext;

import play.templates.JavaExtensions;

public class CurrencyExtensions extends **JavaExtensions** {

public static String ccyAmount(Number number, String currencySymbol) {

String format = "'"+currencySymbol + "'#####.##";

return new DecimalFormat(format).format(number);

}

}

每个扩展方法都是一个static方法，而且要返回一个java.lang.String结果，以便于在页面里显示。第一个参数将用于保存扩展后的对象。

使用方法如下：

<em>Price: ${123456.324234.ccyAmount()}</em>

模板扩展类会被play自动加载，你只需要重启你的应用程序即可。

**模板里可以使用的保留对象**

所有添加到renderArgs作用域的对象都将直接注入成模板变量。

例如，从controller向模板注入一个user bean：

renderArgs.put("user", user );

在一个action里渲染模板里时，框架会自动添加以下保留对象：

|  |  |  |  |
| --- | --- | --- | --- |
| **Variable** | **Description** | **API documentation** | **See also** |
| errors | Validation errors | [play.data.validation.Validation.errors()](http://www.playframework.org/@api/play/data/validation/Validation.html#errors%28%29) | [Validating HTTP data](http://www.playframework.org/documentation/1.2.4/validation) |
| flash | Flash scope | [play.mvc.Scope.Flash](http://www.playframework.org/@api/play/mvc/Scope.Flash.html) | [Controllers - Session & Flash Scope](http://www.playframework.org/documentation/1.2.4/controllers#session) |
| lang | The current language | [play.i18n.Lang](http://www.playframework.org/@api/play/i18n/Lang.html) | [Setting up I18N - Define languages](http://www.playframework.org/documentation/1.2.4/i18n#languages) |
| messages | The messages map | [play.i18n.Messages](http://www.playframework.org/@api/play/i18n/Messages.html) | [Setting up I18N - Externalize messages](http://www.playframework.org/documentation/1.2.4/i18n#messages) |
| out | The output stream writer | java.io.PrintWriter |  |
| params | Current parameters | [play.mvc.Scope.Params](http://www.playframework.org/@api/play/mvc/Scope.Params.html) | [Controllers - HTTP parameters](http://www.playframework.org/documentation/1.2.4/controllers#params) |
| play | Main framework class | [play.Play](http://www.playframework.org/@api/play/Play.html) |  |
| request | The current HTTP request | [play.mvc.Http.Request](http://www.playframework.org/@api/play/mvc/Http.Request.html) |  |
| session | Session scope | [play.mvc.Scope.Session](http://www.playframework.org/@api/play/mvc/Scope.Session.html) | [Controllers - Session & Flash Scope](http://www.playframework.org/documentation/1.2.4/controllers#session) |

以上列出的名称和owner/delegate/it在Groovy里是默认的保留字，在创建变量时要注意回避。

**05.用play验证http数据**

验证确保了某些指定的需求能够获得正确的值。一般用于在存入数据库前对数据进行验证或表单验证。

**在play里验证如何进行的？**

每个请求都有他自己的Validation(验证)对象和相应的错误集合。有以下三种方式来定义验证。

1. 在一个控制器方法，可以直接调用控制器的validation属性的方法。也可使用play.data.validation.Validation类的静态方法来访问API子集。
2. 在控制器的方法参数上使用注释声明来进行验证。
3. 为一个action方法的POJO参数添加@Valid注释来验证POJO属性。

验证对象负责维护play.data.validation.Error对象集合。每个error都有两个属性：

* key:用于确定是哪个元素导致的错误。当play发生错误时，key值可以任意设置，它遵循java变量默认命名约定。
* message:其内容包含了错误的文本描述。message可以是纯文本消息，也可以是key的消息绑定（特别是国际化支持）。

下面我们使用第一种方式来验证一个简单的http参数：

public static void hello(String name) {

validation.required(name);

…

}

此代码用于检测name变量是否正确设置。如果不正确，相应的错误消息将会增加到当前错误集合里。

如果需要，可以为每一个需要验证的变更重复这个操作：

public static void hello(String name, Integer age) {

validation.required(name);

validation.required(age);

validation.min(age, 0);

…

}

**验证的错误消息**

最后我们可以检索出错误信息并显示出来：

public static void hello(String name, Integer age) {

validation.required(name);

validation.required(age);

validation.min(age, 0);

if(validation.hasErrors()) {

for(Error error : validation.errors()) {

System.out.println(error.message());

}

}

}

假如name和age都为null，这时将显示：

Required

Required

这是因为在$PLAY\_HOME/resources/messages里设置的默认消息如下：

validation.required=Required

可通过以下三种方式定制验证消息：

1. 重写应用程序的messages文件，对其中的消息进行重新定义。
2. 提供一个定制消息作为附加的验证参数。
3. 为局部消息提供一个消息key作为附加的验证参数。

**Localised validation messages 局部验证消息**

最简单的方式就是重写messages文件内容：

validation.required = Please enter a value

也可提供其他语言的区位，详见[Internationalization](http://www.playframework.org/documentation/1.2.4/i18n)。

**验证消息参数**

在消息里为错误key使用占位符:

validation.required=%s is required

输出为:

name is required

age is required

**缺点**：当超过一个必须字体验证时（使用validation.requied(age)），play会发生不能确定当前的参数名称语法错误。在这种情况下，就必须直接指定域名称，比如：validation.required("age", age)。

错误key默认就是参数名称。比如在hello action里的name参数可以限定如下：

name = Customer name

输出为：

Customer name is required

age is required

也可使用error.message(String key)方法重载错误key:

Error error = validation.required(name).error;

if(error != null) {

System.out.println(error.message("Customer name"));

}

许多内建的验证定义附加消息参数都适用于验证参数。比如，match验证为指定的正则表达式定义了第2个字符串参数，这与%s占位符不同：

validation.match=Must match %2$s

与此相似，range验证定义了两个附加的数字参数2和3：

validation.range=Not in the range %2$d through %3$d

查看一下$PLAY\_HOME/resources/messages文件也了解更多的验证参数。

**定制局部验证消息**

$PLAY\_HOME/resources/messages使用默认的key为每个play内建的验证定义了验证信息。你可以指定不同的消息key,比如：

validation.required.em = You must enter the %s!

为消息使用新的消息key，就是为了手工在action方法里进行验证：

validation.required(manualKey).message("validation.required.em");

另外一种方式就是在注释的message参数里使用key:

public static void hello(@Required(message="validation.required.em") String name) {

…

}

使用同样的技术可以用于验证JavaBean的属性：

public static void hello(@Valid Person person) {

…

}

public class Person extends Model {

@Required(message = "validation.required.emphasis")

public String name;

…

}

**定制teral（非局部）验证消息**

如果没有为key定义错误消息，那么play只返回消息的key名称，也就是说只需使用文字消息代替消息key。上面的示例可修改为：

validation.required(manualKey).message("Give us a name!");

action方法参数注释：

public static void save(@Required(message = "Give us a name!") String name) {

…

}

JavaBean属性注释：

public static void save(@Valid Person person) {

…

}

public class Person extends Model {

@Required(message = "Give us a name!")

public String name;

…

}

**在模板里显示验证错误消息**

很多情况下都需要在视图模板里显示错误消息。在模板里使用errors对象可以访问这些错误消息，一些标签专用于显示错误消息：

示例:

public static void hello(String name, Integer age) {

validation.required(name);

validation.required(age);

validation.min(age, 0);

render(name, age);

}

模板代码:

#{ifErrors}

<h1>Oops…</h1>

#{errors}

<li>${error}</li>

#{/errors}

#{/ifErrors}

#{else}

Hello ${name}, you are ${age}.

#{/else}

在真实的应用程序里可能需要显示真实的窗体，因此就需要使用两个action:一个用于显示窗体，另一个用于处理POST。

当然，第二个action用于验证操作，当错误发生时就必须直接转向第一个Action。在这种情况下，在跳转期间就需要一个特定的技巧来保存错误消息。使用validation.keep()方法将为下一个action保存错误集合。

示例：

public class Application extends Controller {

public static void index() {

render();

}

public static void hello(String name, Integer age) {

validation.required(name);

validation.required(age);

validation.min(age, 0);

if(validation.hasErrors()) {

params.flash(); //把http参数添加到flash作用域

validation.keep(); // 为下一个请求保存错误消息

index();

}

render(name, age);

}

}

view/Application/index.html模板代码:

#{ifErrors}

<h1>Oops…</h1>

#{errors}

<li>${error}</li>

#{/errors}

#{/ifErrors}

#{form @Application.hello()}

<div>

Name: <input type="text" name="name" value="${flash.name}" />

</div>

<div>

Age: <input type="text" name="age" value="${flash.age}" />

</div>

<div>

<input type="submit" value="Say hello" />

</div>

#{/form}

当错误发生时，为每个域显示错误消息以获得更好的用户体验：

#{ifErrors}

<h1>Oops…</h1>

#{/ifErrors}

#{form @Application.hello()}

<div>

Name: <input type="text" name="name" value="${flash.name}" />

<span class="error">#{error 'name' /}</span>

</div>

<div>

Age: <input type="text" name="age" value="${flash.age}" />

<span class="error">#{error 'age' /}</span>

</div>

<div>

<input type="submit" value="Say hello" />

</div>

#{/form}

**验证注释**

在play.data.validation包里提供的注释语句来给每个Validation对象的方法进行注释进行强制验证，更为简洁。使用验证注释，只需要为控制器方法的参数进行注释即可：

public static void hello(@Required String name, @Required @Min(0) Integer age) {

if(validation.hasErrors()) {

params.flash(); //把http参数添加到flash作用域

validation.keep(); //为下一个请求保存错误消息

index();

}

render(name, age);

}

**验证复杂对象**

使用验证注释，可以应用于模型对象的属性，之后在控制器里所有的属性都必须是有效的。让我们重写一下User类。

对User类的属性使用验证注释：

package models;

public class User {

@Required

public String name;

@Required

@Min(0)

public Integer age;

}

修改hello方法，使用 @Valid注释来指定所有的User对象都必须是有效的：

public static void hello(@Valid User user) {

if(validation.hasErrors()) {

params.flash(); //把http参数添加到flash作用域

validation.keep(); //为下一个请求保存错误消息

index();

}

render(name, age);

}

修改后的窗体代码：

#{ifErrors}

<h1>Oops…</h1>

#{/ifErrors}

#{form @Application.hello()}

<div>

Name: <input type="text" name="user.name" value="${flash['user.name']}" />

<span class="error">#{error 'user.name' /}</span>

</div>

<div>

Age: <input type="text" name="user.age" value="${flash['user.age']}" />

<span class="error">#{error 'user.age' /}</span>

</div>

<div>

<input type="submit" value="Say hello" />

</div>

#{/form}

**内建验证**

play.data.validation 包包含了许多内建的验证[built-in validations](http://www.playframework.org/documentation/1.2.4/validation-builtin)，即可以使用Validation对象，也可使用注释。

**使用@CheckWith定制验证**

使用@CheckWith注释可以用来绑定自己Check验证实现。

示例：

public class User {

@Required

@CheckWith(MyPasswordCheck.class)

public String password;

static class MyPasswordCheck extends Check {

public boolean isSatisfied(Object user, Object password) {

return notMatchPreviousPasswords(password);

}

}

}

默认的验证错误消息key是validation.invalid，要使用不同的key，需要调用Check.setMessage方法（带一个消息key和一个消息参数）。

static class MyPasswordCheck extends Check {

public boolean isSatisfied(Object user, Object password) {

final Date lastUsed = dateLastUsed(password);

setMessage("validation.used", JavaExtensions.format(lastUsed));

return lastUsed == null;

}

}

消息总是查找第一个参数的名称作为field名称，把随后的消息参数作为子参数，因此，上面的示例可以定义如下：

validation.used = &{%1$s} already used on date %2$s

user.password = Password

当&{%1$s}使用第1个位置（field名称）作为消息key来进行消息查找，%2$s作为该key的值（格式化后的日期）。

消息语法- %s, %s2$s and &{…}详见[Retrieve localized messages](http://www.playframework.org/documentation/1.2.4/i18n#retrieve)。

**定制注释**

也可定制自己的验证注释，虽然要复杂一点，但可以使模型代码更简洁，而且可以引入验证参数。

比如，假定我们打算对URL进行验证（使用@URL），通过带参确定指定的URL允许通过。

首先，需要重写默认的消息，定制一个带参的验证注释：

import net.sf.oval.configuration.annotation.Constraint;

import java.lang.annotation.\*;

@Retention(RetentionPolicy.RUNTIME)

@Target({ElementType.FIELD, ElementType.PARAMETER})

@Constraint(checkWith = URICheck.class)

public @interface URI {

String message() default URICheck.message;

}

这个注释参考net.sf.oval.configuration.annotation.AbstractAnnotationCheck实现。

public class URICheck extends AbstractAnnotationCheck<URI> {

/\*\* Error message key. \*/

public final static String message = "validation.uri";

/\*\* URI schemes allowed by validation. \*/

private List<String> schemes;

@Override

public void configure(URI uri) {

setMessage(uri.message());

this.schemes = Arrays.asList(uri.schemes());

}

/\*\*

\* Add the URI schemes to the message variables so they can be included

\* in the error message.

\*/

@Override

public Map<String, String> createMessageVariables() {

final Map<String, String> variables = new TreeMap<String, String>();

variables.put("2", JavaExtensions.join(schemes, ", "));

return variables;

}

@Override

public boolean isSatisfied(Object validatedObject, Object value,

OValContext context, Validator validator) throws OValException {

requireMessageVariablesRecreation();

try {

final java.net.URI uri = new java.net.URI(value.toString());

final boolean schemeValid = schemes.contains(uri.getScheme());

return schemes.size() == 0 || schemeValid;

} catch (URISyntaxException e) {

return false;

}

}

}

在渲染消息之前，The isSatisfied方法调用requireMessageVariablesRecreation()来指示OVal 去调用createMessageVariables()。返回一个整齐的变量map来传递消息格式。map keys 并不被使用; the "2" in this example indicates the message parameter index. As before, the first parameter is the field name.

在模型里使用：

public class User {

@URI(message = "validation.uri.schemes", schemes = {"http", "https"})

public String profile;

}

消息定义如下：

validation.uri = Not a valid URI

validation.uri.schemes = &{%1$s} is not a valid URI - allowed schemes are %2$s

**06.域对象模型**

模型是play应用程序中核心。是应用程序操作的信息的特定领域呈现。

Martin Fowler将之定义为：

模型层主要负责表现商业内容、商业状态和商业规则的信息。在这里主要进行商业状态控制和作用，相应技术细节则委托给基础设施。这个层是商业软件的最重要部分。

通用的java反映模式用许多简单的java Bean来映射模型以保存数据，应用程序逻辑被放入“service”层，用于操作模型对象。

Martin fowler把这种反映模式命名为贫血对象模型 [Anemic object model](http://www.martinfowler.com/bliki/AnemicDomainModel.html):

贫血对象模型的基本特征就是外表看起来就是一个真实的事物，但在模型里几乎没有行为，只有getter和setter，不能在模型对象里放入逻辑。模型的行为通过许多包括有域逻辑的service对象来实现。  
  
这样的模型是和OO相反的，OO对象既有数据也有对象的方法。

**属性模仿**

在play里，类的变量都是public的。这引起java开发界的一些质疑。在java的标准教程里，为了对数据进行封闭，要求属性都是private的。这导致了一些批评。

java并没有真正的内建属性定义系统。只是一个Java Bean的约定：在java对象里的属性都要有getter和setter方法，如果属性是只读的，那么只能有getter。

虽然系统可以很好地工作，但编码十分乏味。每个属性都必须声明为私有变量并为此书写getter和setter。因此，许多时候，getter和setter实现都是相同的：

private String name;

public String getName() {

return name;

}

public void setName(String value) {

name = value;

}

在play里，play会为模型自动生成这些内容，以保证代码清晰。事实上，所有public变量都是实例属性。在play里约定为：类的任何**public,non-static,no-final**域都将以属性对待。

比如：

public class Product {

public String name;

public Integer price;

}

类在加载时，就变成了如下内容:

public class Product {

public String name;

public Integer price;

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public Integer getPrice() {

return price;

}

public void setPrice(Integer price) {

this.price = price;

}

}

要访问一个属性里，只需要以下代码：

product.name = "My product";

product.price = 58;

在加载时会自动翻译为：

product.setName("My product");

product.setPrice(58);

**注意！**  
  
在自动生成方式下，不能直接作用getter和setter方法来访问属性。这些方法仅存在于运行时状态，因此，如果在编写代码时调用这些方法，将导致不能找到方法的编译错误。

当然也可自行定义getter和setter方法。如果自行定义了这两个方法，play会自动使用这两个手工编写的方法。

为了保护Product类的price属性值，可以使用以下方法：

public class Product {

public String name;

public Integer price;

public void setPrice(Integer price) {

if (price < 0) {

throw new IllegalArgumentException("Price can’t be negative!");

}

this.price = price;

}

}

然后试着给price赋值一个负数时，就会抛出参数异常：

product.price = -10: // Oops! IllegalArgumentException

Play总是会使用已经手工定义好的getter或setter，如下：

@Entity

public class Data extends Model {

@Required

public String value;

public Integer anotherValue;

public Integer getAnotherValue() {

if(anotherValue == null) {

return 0;

}

return anotherValue;

}

public void setAnotherValue(Integer value) {

if(value == null) {

this.anotherValue = null;

} else {

this.anotherValue = value \* 2;

}

}

public String toString() {

return value + " - " + anotherValue;

}

}

在另外一个类里对上面的类进行断言:

Data data = new Data();

data.anotherValue = null;

assert data.anotherValue == 0;

data.anotherValue = 4

assert data.anotherValue == 8;

正常工作，这是因为增加了getter和setter的类遵循Java Bean约定。

**设置数据库来持久化模型对象**

很多时候都需要把模型对象数据永久保存，最自然的方式就是把数据存入数据库。

在开发期间，可以直接使用内建的数据库来临时保存数据到内存或一个子目录里。

play发布包里包含了H2和Mysql的JDBC驱动（$PLAY\_HOME/framework/lib/）。如果打算使用PostgreSQL 或Oracle数据库，那么就需要把对应的jdbc驱动放入库目录里，或放入应用程序的lib目录。

连接到任何JDBC规范的数据，只需要设置jdbc的[db.url](http://www.playframework.org/documentation/1.2.4/configuration#db.url), [db.driver](http://www.playframework.org/documentation/1.2.4/configuration#db.driver), [db.user](http://www.playframework.org/documentation/1.2.4/configuration#db.user) 和 [db.pass](http://www.playframework.org/documentation/1.2.4/configuration#db.pass)属性：

db.url=jdbc:mysql://localhost/test

db.driver=com.mysql.jdbc.Driver

db.user=root

db.pass=

使用[jpa.dialect](http://www.playframework.org/documentation/1.2.4/configuration#jpa.dialect)属性可以为jpa定义方言。

在代码里，就可以从play.db.DB获取一个java.sql.Connection。

Connection conn = DB.getConnection();

conn.createStatement().execute("select \* from products");

**用hibernate持久化对象模型**

使用hibernate来自动持久化java对象到数据库里。

在任何java对象增加 @javax.persistence.Entity注释就可以定义一个jpa实体。play会自动启动一个jpa实体管理器。

@Entity

public class Product {

public String name;

public Integer price;

}

**注意！**   
一个经常发生的错误是使用hibernate的@Entity注释来代替JPA注释。请记住，play是通过hibernate来使用的jpa的api。也就是说要引入：javax.persistence.Entity包，而不是hibernate的包。

然后就可以从play.db.jpa.JPA获取一个EntityManager：

EntityManager em = JPA.em();

em.persist(product);

em.createQuery("from Product where price > 50").getResultList();

play提供了一个非常漂亮的支持类来处理jpa，只需要实体类继承play.db.jpa.Model即可。

@Entity

public class Product extends Model {

public String name;

public Integer price;

}

之后使用Product实例的简单方法就可以操作Product对象：

Product.find("price > ?", 50).fetch();

Product product = Product.findById(2L);

product.save();

product.delete();

**保持模型stateless**

play被设计成“什么都不共享”的机制。这个机制用于保持应用是完全无状态的。这样就允许程序可以同时运行于多个服务器节点。

Play 框架的设计架构就是无状态的。它没有提供服务器端的机制用来维护跨多个请求的数据。如果确实需要保存这样的数据的话，可以考虑下面几种方案：

1. 如果数据很少很简单，可以存储在flash或session使用域里。但这些域只能存储小于4kb的数据，并且只能是字符串类型的数据。
2. 使用数据库，比如需要创建一个横跨多个请求的wizard对象：
   * 在第一次请求时初始化并持久化对象到数据库中。
   * 把新创建的对象的ID存储到flash或session域中。
   * 在接下来的请求中，使用对象id找回数据、更新数据、再次存储等。
3. 暂时存储到Cache中，比如需要创建一个横跨多个请求的wizard对象：
   * 在第一次请求时初始化并持久化对象到Cache中。
   * 把新创建的对象的key存储到flash或session域中。
   * 在接下来的请求中，使用正确的key找回数据、更新数据、再次存储等。
   * 在结束最后一次请求后，把对象永久存储到数据库中。

Cache不是一个可靠的存储位置，在系统未出现故障时应该可以正确操作数据。但Cache是比Java Servlet session更好的选择。

# 07.JPA持久化

play提供了一些非常有用的帮助类来简单管理jpa实体。

**注意：**如果需要，你仍旧可以继续使用原始的JPA API。

## 启动JPA实体管理器

当play找到至少一个注释了@javax.persistence.Entity标识的类时，play将自动启动hibernate实体管理器。前提是已经有一个正确的JDBC数据源配置，否则会导致失败。

## 获取JPA实体管理器

当JPA实体管理器启动后，就可以在应用程序代码中得到管理器，并使用JPA帮助类了，比如：

public static index() {

Query query = JPA.em().createQuery("select \* from Article");

List<Article> articles = query.getResultList();

render(articles);

}

## 事务管理

play会自动管理事务。当http请求到达，play就会为每个http请求启动一个事务。当http response发送的时候，就会把事务提交。如果代码抛出异常，事务将会自动回滚。

如果需要在代码中强制回滚事务，可以使用JPA.setRollbackOnly()方法，以告诉JPA不要提交当前事务。

也可使用注释明确哪些事务要进行处理。

如果在控制器里用@play.db.jpa.Transactional(readOnly=true)注释了控制器的某个方法，那么这个事务是只读的。

如果不想让play启动事务，可以使用以下注释@play.db.jpa.NoTransaction。

如果不想让类的所有方法执行事务，可以对控制器类进行注释： @play.db.jpa.NoTransaction.

当使用@play.db.jpa.NoTransaction注释时，Play不会从连接池中获取连接，以提高运行速度。

## play.db.jpa.Model支持类

在play中，这是最主要的帮助类，如果你的jpa实体继承了play.db.jpa.Model类，那么这个实体类将得到许多非常有用的方法来管理jpa访问。

比如下面的Post模型对象：

@Entity

public class Post extends Model {

public String title;

public String content;

public Date postDate;

@ManyToOne

public Author author;

@OneToMany

public List<comment> comments;

}

play.db.jpa.Model类自动提供了一个自增长的Long id域。采用自增长的Long id主键对jpa模型来说是个好主意。

注意，我们事实上已经使用了play中的一特性，也就是说play会自动把Post类中的public成员认作属性。因此，我们不需要为这些成员书写setter/getter。

## 为GenreicModel定制id映射

play并不强制使用play.db.jpa.Model。你的JPA实体也可以继承play.db.jpa.GenericModel，如果不打算使用Long类型的id作为主键，就必须这样做。

比如，下面是一个非常简单的User实体类。它的id是UUID， name和mail属性都是非空值，我们使用Play验证进行强制检测：

@Entity

public class User extends GenericModel {

@Id

@GeneratedValue(generator = "system-uuid")

@GenericGenerator(name = "system-uuid", strategy = "uuid")

public String id;

@Required public String name;

@Required

@MaxSize(value=255, message = “email.maxsize”)

@play.data.validation.Email

public String mail;

}

## Finding对象

play.db.jpa.Model提供了几种方式来查找数据，比如：

### Find by ID

这是查找对象最简单的方式：

Post aPost = Post.findById(5L);

### Find all

List<Post> posts = Post.findAll();

这是获取所有posts对象最简单的方式，类似的应用还有：

List<Post> posts = Post.all().fetch();

下面对结果进行分页：

// 最多100条

List<Post> posts = Post.all().fetch(100);

或

// 50至100条

List<Post> posts = Post.all().from(50).fetch(100);

### 使用简单查询进行查找

以下方式允许你创建一些非常有用的查询，但仅限于简单查询：

Post.find("byTitle", "My first post").fetch();

Post.find("byTitleLike", "%hello%").fetch();

Post.find("byAuthorIsNull").fetch();

Post.find("byTitleLikeAndAuthor", "%hello%", connectedUser).fetch();

简单查询遵循以下语法[属性][比较]And?，比较可取以下值：

* LessThan –小于给定值
* LessThanEquals – 小于等于给定值
* GreaterThan – 大于给定值
* GreaterThanEquals – 大于等于给定值
* Like –等价于SQL的like表达式，但属性要为小写。
* Ilike – 和Like相似，大写不敏感，也就是说参数要转换成小写。
* Elike -等价于SQL的like表达式，不进行转换。
* NotEqual – 不等于
* Between – 两个值之间（必须带2个参数）
* IsNotNull – 非空值（不需要任何参数）
* IsNull – 空值（不需要任何参数）

### 使用JPQL 查询进行查找

如：

Post.find(

"select p from Post p, Comment c " +

"where c.post = p and c.subject like ?", "%hop%"

);

或仅查询某部分:

Post.find("title", "My first post").fetch();

Post.find("title like ?", "%hello%").fetch();

Post.find("author is null").fetch();

Post.find("title like ? and author is null", "%hello%").fetch();

Post.find("title like ? and author is null order by postDate", "%hello%").fetch();

也可仅有order by语句：

Post.find("order by postDate desc").fetch();

## Counting统计对象

统计对象非常容易：

long postCount = Post.count();

或使用查询进行统计：

long userPostCount = Post.count("author = ?", connectedUser);

## 用play.db.jpa.Blob存储上传文件

使用play.db.jpa.Blob类型可以存储上传的文件到文件系统里（不是数据库）。在服务器端，Play以文件方式存储上传的图片到应用程序目录下的attachments文件夹下。文件名(一个[UUID](http://en.wikipedia.org/wiki/Uuid)是指在一台机器上生成的数字，通用唯一识别码，用来唯一标识不同的文件)和MIME类型将存储到数据库的属性中（SQL数据类型为VARCHAR）。

在play里上传、存储、下载文件非常容易。这是因为框架自动对html窗体到jpa模型进行了文件上传绑定,而且play还提供了便利的方法来操作二进制数据，就像操作普通文本一样简单。为了在模型里存储上传文件，需要增加一个play.db.jpa.Blob类型的属性

import play.db.jpa.Blob;

@Entity

public class User extends Model {

public String name;

public Blob photo;

}

为了上传文件，需要在视图模板里添加一个窗体，在窗体里使用文件上传组件，组件名称应为模型的Blob属性，如user.photo：

#{form @addUser(), enctype:'multipart/form-data'}

<input type="file" name="user.photo">

<input type="submit" name="submit" value="Upload">

#{/form}

之后，在控制器里增加一个方法用于存储上传的文件：

public static void addUser(User user) {

user.save();

index();

}

这些代码除了jpa实体的存储操作外，好像什么都没做，这是因为play自动对上传文件进行了处理。首先，在启动action方法之前，上传的文件已经存储到应用程序的tmp/uploads/文件夹下，接着，当实体存储完成后，上传的文件会被复制到应用程序的attachments/目录，文件的名称为UUID。最后，当action完成后，临时文件将被删除。

如果同一用户上传另外一个文件，服务器将把上传的文件当作新文件进行存储，并为新文件生成一个新的UUID文件名，也就是说之前上传的文件无效。要实现多文件上传，就必须自行去实现，比如采用异步job方式。

如果http请求没有指定文件的MIME类型，你可以使用文件名称扩展。

要想把文件存储到不同的目录，需要配置[attachments.path](file:///D:\framework\play-1.2.4\Desktop\configuration#attachments.path)。

要想下载存储的文件，需要给控制器的renderBinary()方法传递Blob.get()参数。

## 强制保存

Hibernate负责维护从数据库查询出来的对象缓存，这些对象将被当作持久化对象进行对待，其时限和EntityManager生命周期一样长。也就是说所有绑定了事务的对象的任何改变都会在事务提交时自动进行持久化。在标准的JPA里，更新操作属于事务范围，也就不需要强制调用任何方法来持久化值。

负面影响就是你必须手工管理所有的对象，而不是告诉EntityManager去更新对象（哪种更直观）。我们必须告诉EntityManager哪个对象不需要更新，这个操作是通过调用refresh()来实现的，本质上是回滚一个单实体。我们在提交事务之前调用refresh()方法的目的就是为了让某些对象不被更新。

下面是一个通用情况，在窗体已经提交后，对一个持久化对象进行编辑：

public static void save(Long id) {

User user = User.findById(id);

user.edit("user", params.all());

validation.valid(user);

if(validation.hasErrors()) {

//这里我们必须丢弃用户的编辑

user.refresh();

edit(id);

}

show(id);

}

这里我们看到，许多开发者并未意识到这个问题，总是忘记在错误的情况下丢弃对象现有状态。

因此，应该知道我们在play里修改了什么？所有继承自JPASupport/JPAModel的持久化对象在没有明白调用save()方法时都不会进行存储。因此，你可以重新书写上面的代码：

public static void save(Long id) {

User user = User.findById(id);

user.edit("user", params.all());

validation.valid(user);

if(validation.hasErrors()) {

edit(id);

} else{

user.save(); // 强制保存

show(id);

}

}

这样就更加直观。但是，如果在一个比较大的对象视图里每次都明确调用save()方法将变得乏味，这时可使用关系注释的cascade=CascadeType.ALL属性来自动调用save()方法。

## 更多公共类型generic typing问题

play.db.jpa.Model定义了许多公共方法。这些方法使用一种类型参数来指定方法的返回值类型。在使用这些方法的时候，返回值的具体类型由调用的上下文类型接口确定。

比如，findAll定义如下：

<T> List<T> findAll();

使用情况为：

List<Post> posts = Post.findAll();

在这里，java编译器使用你分配给结果方法List<Post>的类型作为T的实际类型。因此，T的结果类型为Post。

遗憾的是，如果通用方法的返回值直接作为另外一个方法调用的参数时，或用作循环时，这些方法将不能正常工作。因此，下面的代码将抛出编译错误“Type mismatch: cannot convert from element type Object to Post”:

for(Post p : Post.findAll()) {

p.delete();

}

当然可以使用临时局部变量来解决这个问题：

List<Post> posts = Post.findAll(); //类型引用在这里实现！

for(Post p : posts) {

p.delete();

}

请等一等，还有更好的方式，你可以使用已经实现的但不太广泛使用的java语言特性来解决该问题，这样可以使代码更短小易读：

for(Post p : Post.<Post>findAll()) {

p.delete();

}

很重要的一点就是play不支持XA(两阶段提交)。如果你在同一请求里使用多个不同的jpa配置，**play将试着提交更多的事务**。如果在第一个数据库成功提交，而在第二个数据库提交失败，那么第一个数据库提交的数据将不会回滚。当在同一个请求里使用多个jpa配置时一定要牢记这一点。

**08.Play.libs库包**

play.libs包包含了很多有用的库，以帮助实现通用的编程任务。

大多数据库都是简单的帮助类，而且非常易懂易用：

* [Codec](http://www.playframework.org/@api/play/libs/Codec.html):数据编码和解码工具箱
* [Crypto](http://www.playframework.org/@api/play/libs/Crypto.html):密码图形工具（验证码？）
* [Expression](http://www.playframework.org/@api/play/libs/Expression.html):动态评价表达式？
* [F](http://www.playframework.org/@api/play/libs/F.html): java实用编程工具
* [Files](http://www.playframework.org/@api/play/libs/Files.html):文件系统控制帮助类
* [I18N](http://www.playframework.org/@api/play/libs/I18N.html):国际化帮助类
* [IO](http://www.playframework.org/@api/play/libs/IO.html):流控制帮助类
* [Images](http://www.playframework.org/@api/play/libs/Images.html):图片控制工具箱
* [Mail](http://www.playframework.org/@api/play/libs/Mail.html): E-mail功能
* [MimeTypes](http://www.playframework.org/@api/play/libs/MimeTypes.html): MIME类型交换
* [OAuth](http://www.playframework.org/@api/play/libs/OAuth.html): OAuth客户端协议（安全认证）
* [OAuth2](http://www.playframework.org/@api/play/libs/OAuth2.html): OAuth2客户端协议（安全认证）
* [OpenID](http://www.playframework.org/@api/play/libs/OpenID.html): OpenID客户端协议（安全认证）
* [Time](http://www.playframework.org/@api/play/libs/Time.html): 时间和持续期间工具箱
* [WS](http://www.playframework.org/@api/play/libs/WS.html): 强大的Web services客户端
* [XML](http://www.playframework.org/@api/play/libs/XML.html): 加载XML结构
* [XPath](http://www.playframework.org/@api/play/libs/XPath.html): 用XPath解析XML

下面的内容着重介绍一些非常重要的库。

**用XPath解析XML**

[XPath](http://en.wikipedia.org/wiki/XPath) 或许是最容易解析XML文档的工具了，而且不需要使用代码生成工具。play.libs.XPath库为高效完成解析任务提供了所有需求。

XPath可以操作所有的org.w3.dom.Node类型:

org.w3.dom.Document xmlDoc = … // 找到 a Document somewhere

for(Node event: XPath.selectNodes("events//event", xmlDoc)) {

String name = XPath.selectText("name", event);

String data = XPath.selectText("@date", event);

for(Node place: XPath.selectNodes("//place", event)) {

String place = XPath.selectText("@city", place);

…

}

…

}

**Web Service client**

play.libs.WS提供了一个强大的http客户端，Under the hood it uses [Async HTTP client](https://github.com/AsyncHttpClient/async-http-client).

创建一个请求非常容易：

HttpResponse res = WS.url("http://www.google.com").get();

一旦获得HttpResponse对象后，就可以访问所有的response属性了：

int status = res.getStatus();

String type = res.getContentType();

也可获得不同类型的内容体：

String content = res.getString();

Document xml = res.getXml();

JsonElement json = res.getJson();

InputStream is = res.getStream();

也可以非阻塞方式使用**async** API来创建一个http请求。然后就可以得到一个Promise<HttpResponse>，一旦兑现成功，就可以和平常一样使用HttpResponse：

Promise<HttpResponse> futureResponse = WS.url(

"http://www.google.com"

).getAsync();

**Functional programming with Java功能扩展？**

play.libs.F库从功能编程（functional programming）带来了许多非常有用的概念constructs。这些概念用于处理复杂的抽象环境。这些概念之前也有涉及：

* Option<T> (T值可设置也可不用设置)
* Either<A,B> (不是A就是B)
* Tuple<A,B> (同是包含了A和B)

**Option<T>, Some<T> and None<T>**

在某些情况下，函数可能不会返回结果（比如find方法），通常做法（非常糟糕）是返回null，这样做实际上非常危险，因为函数的返回类型不清晰。Option<T>就是一个优雅的解决方案。如果函数成功执行，就返回明确的类型Some<T>（封装了真实的结果），否则返回对象None<T>（Option<T>的子类型）。示例：

/\* 安全除法(绝不抛出ArithmeticException运行时错误) \*/

public Option<Double> div(double a, double b) {

if (b == 0)

return None();

else

return Some(a / b);

}

用法：

Option<Double> q = div(42, 5);

if (q.isDefined()) {

Logger.info("q = %s", q.get()); // "q = 8.4"

}

这儿还有更便利的语法，这是因为Option<T>实现了Iterable<T>：

for (double q : div(42, 5)) {

Logger.info("q = %s", q); // "q = 8.4"

}

仅在div成功的情况下，for循环才被执行一次。

**Tuple<A, B>**

Tuple<A, B>类包装了两种类型的对象A和B。使用\_1和\_2域可以分别找回A和B，如：

public Option<Tuple<String, String>> parseEmail(String email) {

final Matcher matcher = Pattern.compile("(\\w+)@(\\w+)").matcher(email);

if (matcher.matches()) {

return Some(Tuple(matcher.group(1), matcher.group(2)));

}

return None();

}

然后：

for (Tuple<String, String> email : parseEmail("foo@bar.com")) {

Logger.info("name = %s", email.\_1); // "name = foo"

Logger.info("server = %s", email.\_2); // "server = bar.com"

}

T2<A, B>类是Tuple<A, B>的别名。处理三个元素时使用T3<A, B, C>类，一直可到T5<A, B, C, D, E>。

**Pattern Matching模式匹配**

有些时候我们需要在java里进行模式匹配。遗憾的是java没有内建的模式匹配机制，java也缺乏功能性概念，所以很难增加到库里。在这里，我们采取的方案并不算太坏。

我们的主意是使用最后一次“for loop”语法来实现基本的模式匹配任务。模式匹配必须检测对象是否匹配必须的条件，还要能提取感兴趣的值。play使用的模式匹配库位于play.libs.F：

示例，假设已经有一个对象类型的引用，现在想检测这个对象是否是String类型，并且是以‘command:’字符串开始的。

标准方式为：

Object o = anything();

if(o instanceof String && ((String)o).startsWith("command:")) {

String s = (String)o;

System.out.println(s.toUpperCase());

}

使用Play模式匹配库，就可以这样写：

for(String s: String.and(StartsWith("command:")).match(o)) {

System.out.println(s.toUpperCase());

}

只有在条件符合的情况下，for循环才会被执行一次，并且会自动提取字符串值，而不需要进行转换。这是因为每个对象都是类型安全的，并且由编译器进行检测，不需要进行转换。

**Promises**

Promise是play定制的“将来Future”类型。事实上一个Promise<T>类型也是一个Future<T>类型，因此，可以用作标准的Future。但它拥有一个非常有趣的属性：使用onRedeem(…)方法能够获取注册返回的能力，该方法仅在允许的值可用时才能被调用。

Promise实例可以用在任何Future实例里（比如Jobs, WS.async，等待)。

Promises还可进行组合，比如:

Promise p = Promise.waitAll(p1, p2, p3)

Promise p = Promise.waitAny(p1, p2, p3)

Promise p = Promise.waitEither(p1, p2, p3)

**OAuth**

[OAuth](http://oauth.net/) 是一个开源协议的安全API验证， 应用地桌面和web应用程序。

这里有两种不同的定义: OAuth 1.0和OAuth 2.0。Play提供库来以消费者方式连接至可能的服务。

标准步骤为：

* 跳转用户到提供者的验证页
* 在用户授权验证结束后，将直接返回到原来的服务器
* 你的服务器将会为当前用户交换修改已经验证的信息令牌，这步操作是以服务器至服务器的方式完成的。

play会自动完成许多处理过程。

**OAuth 1.0**

OAuth 1.0功能库是通过play.libs.OAuth类提供的，这个类基于[oauth-signpost](http://code.google.com/p/oauth-signpost/)。主要用于[Twitter](http://apiwiki.twitter.com/) 和 [Google](http://code.google.com/apis/accounts/docs/OAuth.html)的验证服务。

要想连接到一个服务，你需要使用下面的信息创建一个OAuth.ServiceInfo实例，以获取service provider:

* Request token URL
* Access token URL
* Authorize URL
* Consumer key
* Consumer secret

access token可通过如下方式找到：

public static void authenticate() {

// TWITTER 是OAuth.ServiceInfo对象

// getUser() 用于返回当前用户

if (OAuth.isVerifierResponse()) {

// 得到verifier检验者

// 然后使用请求tokens得到access tokens

OAuth.Response resp = OAuth.service(TWITTER).retrieveAccessToken(

getUser().token, getUser().secret);

// 存储它们并返回index

getUser().token = resp.token; getUser().secret = resp.secret;

getUser().save()

index();

}

OAuth twitt = OAuth.service(TWITTER);

Response resp = twitt.retrieveRequestToken();

//得到未经授权的标志tokens

//在继续之前先要保存

getUser().token = resp.token; getUser().secret = resp.secret;

getUser().save()

// 跳转用户到验证页

redirect(twitt.redirectUrl(resp.token));

}

现在可以使用token对通过分配的请求执行下面的调用：

mentions = WS.url(url).oauth(TWITTER, getUser().token, getUser().secret).get().getString();

尽管这个事例没有进行错误检测，但在生产环境，还是应该进行检测。OAuth.Response对象拥有一个error域，错误发生的时候，其中含有错误的内容。很多情况下不能给用户授权的原因是提供者已经下线或错误太多。

完整的示例见samples-and-tests/twitter-oauth。

**OAuth 2.0**

OAuth 2.0比OAuth 1.0更简单，这是因为它不需要调用signing请求。主要用于[Facebook](http://developers.facebook.com/docs/authentication/) 和 [37signals](http://37signals.com)。

play.libs.OAuth2类提供了该支持。

为了连接到服务，你需要使用下面的信息创建一个OAuth2实例，从服务提供者获取：

* Access token URL
* Authorize URL
* Client ID
* Secret

public static void auth() {

// FACEBOOK 是一个OAuth2对象

if (OAuth2.isCodeResponse()) {

// authUrl必须和retrieveVerificationCode调用一致

OAuth2.Response response = FACEBOOK.retrieveAccessToken(authUrl);

//如果有错误发生则为null

String accessToken = response.accessToken;

//调用成功则为null

OAuth2.Error = response.error;

//存储accessToken，在请求服务时要用到

index();

}

// authUrl是一个包含了服务绝对URL的字符串

// 应该跳转回来

// 下面将触发一个跳转

FACEBOOK.requestVerificationCode(authUrl);

}

一旦获取当前用户的access token，就可以用它来查询服务：

WS.url(

"https://graph.facebook.com/me?access\_token=%s", access\_token

).get().getJson();

完整示例见samples-and-tests/facebook-oauth2。

**OpenID**

[OpenID](http://openid.net/) 是一个开源的分散式身份认证系统。在你的系统里不需要保存特定用户的信息就可以很容易接受新的用户。只需通过它们的OpenID跟踪验证用户即可。

本示例提供一个轻量级的演示，用于演示如果使用OpenID验证用户：

* 对每个请求，检测用户是否已经连接
* 如果没有，就显示用户可以提交OpenID的页面
* 跳转用户到OpenID提供者
* 当用户回来的时候，获取验证的OpenID并存储到HTTP session里

play.libs.OpenID类提供了OpenID功能：

@Before(unless={"login", "authenticate"})

static void checkAuthenticated() {

if(!session.contains("user")) {

login();

}

}

public static void index() {

render("Hello %s!", session.get("user"));

}

public static void login() {

render();

}

public static void authenticate(String user) {

if(OpenID.isAuthenticationResponse()) {

UserInfo verifiedUser = OpenID.getVerifiedID();

if(verifiedUser == null) {

flash.error("Oops. Authentication has failed");

login();

}

session.put("user", verifiedUser.id);

index();

} else {

if(!OpenID.id(user).verify()) { // will redirect the user

flash.error("Cannot verify your OpenID");

login();

}

}

}

login.html模板代码为：

#{if flash.error}

<h1>${flash.error}</h1>

#{/if}

<form action="@{Application.authenticate()}" method="POST">

<label for="user">What’s your OpenID?</label>

<input type="text" name="user" id="user" />

<input type="submit" value="login…" />

</form>

</code>

路由定义为：

GET / Application.index

GET /login Application.login

\* /authenticate Application.authenticate

**09.异步Jobs**

因为play是一个web应用程序，因此许多应用程序逻辑都是由控制器返回给http请求的。

但有些时候，我们需要在http请求外执行一些应用逻辑。比如非常有用的初始化任务，维护任务或运行不能被http请求池中断的长时运行的任务等等。

Jobs可以被框架全面进行管理。意思是play负责管理所有的数据库连接原料stuff，JPA实体管理器负责管理数据同步和事务管理。要想创建一个job，只需要继承play.jobs.Job即可。

package jobs;

import play.jobs.\*;

public class MyJob extends Job {

public void doJob() {

//在这儿执行某些逻辑

}

}

有些时候需要任务返回结果，这时就需要重载doJobWithResult()方法。

package jobs;

import play.jobs.\*;

public class MyJob extends Job<String> {

public String doJobWithResult() {

//在这儿执行某些逻辑

return result;

}

}

本示例仅使用了String作为返回类型，当然可以返回任何对象类型。

**引导程序任务Bootstrap jobs**

引导程序任务会在play应用启动时执行。要想实现该任务，只需要在你的任务上添加@OnApplicationStart注释：

import play.jobs.\*;

@OnApplicationStart

public class Bootstrap extends Job {

public void doJob() {

if(Page.count() == 0) {

new Page("root").save();

Logger.info("A root page has been created.");

}

}

}

**注意：**在这里不需要返回结果，即使这样做了，结果也会丢失。

默认情况下，所有标识为@OnApplicationStart的任务都将以队列方式执行。当所有的job执行结束后，应用程序才正式启动并开始处理请求。

如果你打算让你的任务在应用程序启动时执行，但你又想立即管理进行请求处理，那么可以使用@OnApplicationStart(async=true)注释。然后，你的job将在后台启动。

**警告！**

当运行于DEV模式时，应用程序将在第一个请求到达时才启动。此外，在DEV模式时，在需要的时候，应用程序会自动重启。

当运行于PROD模式时，应用程序将和服务器一起同步启动。

**预定义任务Scheduled jobs**

预定义任务由框架周期性执行。你可以使用@Every注释要求play在一个特定的周期内运行job。

import play.jobs.\*;

@Every("1h")

public class Bootstrap extends Job {

public void doJob() {

List<User> newUsers = User.find("newAccount = true").fetch();

for(User user : newUsers) {

Notifier.sayWelcome(user);

}

}

}

如果@Every注释还不足以满足需要，你可使用带有CRON表达式的@On注释来运行你的job。

import play.jobs.\*;

/\*\* Fire at 12pm (noon) every day \*\*/

@On("0 0 12 \* \* ?")

public class Bootstrap extends Job {

public void doJob() {

Logger.info("Maintenance job ...");

...

}

}

**小建议**  
  
我们是使用[Quartz library](http://www.quartz-scheduler.org/docs/tutorials/crontrigger.html)来解析CRON表达式的。

你不能返回结果，即使这样做了，结果也会被丢弃。

**触发任务job**

调用Job实例的now()方法可以在任何时候触发job来执行一段特定的任务。这个时候，job将以非阻塞方式立即执行。

public static void encodeVideo(Long videoId) {

new VideoEncoder(videoId).now();

renderText("Encoding started");

}

调用job的now()方法以返回一个Promise值。

**停止应用程序**

使用@OnApplicationStop注释可以在应用程序关闭时执行某些操作。

import play.jobs.\*;

@OnApplicationStop

public class Bootstrap extends Job {

public void doJob() {

Fixture.deleteAll();

}

}

**10.在HTTP下进行异步编程**

本节将介绍在play里如何进行异步处理以实现典型的长轮询（long-polling）、流以及其他[Comet-style](http://en.wikipedia.org/wiki/Comet_%28programming%29)类型的应用程序以支持上千个同时发生的连接。

**暂停http请求**

Play使用的是短小的请求，它使用固定的线程池来处理http连接者的查询请求。为了达到最佳效果，线程池应该尽可能小。我们通常使用的最适合的数字就是处理器数量+1来设置默认池大小。

也就是说如果请求的处理时间很长的话（比如等待一个长时的计算），这个请求就会耗尽线程池，使应用程序的响应变得很慢。当然可以扩大线程池，但这样会浪费资源，而且线程池不可能是无限的。

考虑一下聊天室应用，浏览器发送一个阻塞式的http请求用于等待显示新的信息。这个请求将会非常这长（比如几分钟），并且会打破线程池。如果计划允许100个用户同时连接聊天室程序，那么我们就需要提供至少100个线程，当然，这是可行的。但如果是1000个，或100000个呢？

在这种情况下，play允许你暂停一个请求。http请求将停留在连接状态，但请求执行将被从线程池中弹出，过会再试。你即可在固定等待的时间内告诉play去测试一下该请求，也可等待一个允许值变为可能情况。

**小提示：** 请看一下真实的示例samples-and-tests/chat

比如，下面这个动作将要加载一个非常耗时的job并且一直等到job完成返回结果：

public static void generatePDF(Long reportId) {

Promise<InputStream> pdf = new ReportAsPDFJob(report).now();

InputStream pdfStream = await(pdf);

renderBinary(pdfStream);

}

这里，我们使用await(…)来让Play暂停请求，直到Promise<InputStream> 值返回redeemed。

**Continuations**

因为框架需要收回线程以便为其他请求服务，因此play就必须暂停你的代码。在之前的play版本中await(…) 等价于waitFor(…)，用于暂停你的action，之后又重新调用。

为了易于约定我们介绍的异步代码，Continuations允许代码被暂停和被透明恢复，因此书写如下代码是非常必要的：

public static void computeSomething() {

Promise<String> delayedResult = veryLongComputation(…);

String result = await(delayedResult);

render(result);

}

在这里，事实上你的代码将分成两步用两个不同的线程来执行。但这些代码对你来说是完全透明的。

使用await(…)和continuations,你可能需要写一个循环：

public static void loopWithoutBlocking() {

for(int i=0; i<=10; i++) {

Logger.info(i);

await("1s");

}

renderText("Loop finished");

}

当使用一个线程处理这个请求时，在默认的开发模式下，Play能够同时为不同的请求运行这些循环。

更实际的示例是异步从远程URL获取内容。下面将并行运行三个远程http请求：每个都调用play.libs.WS.WSRequest.getAsync()方法来执行一个GET请求 ，异步返回一个play.libs.F.Promise。action方法通过调用三个Promise组合实例的await(…)方法来暂停进入的http请求。当三个远程调用返回结果后，线程将自动恢复并且渲染response。

public class AsyncTest extends Controller {

public static void remoteData() {

F.Promise<WS.HttpResponse> r1 = WS.url("http://example.org/1").getAsync();

F.Promise<WS.HttpResponse> r2 = WS.url("http://example.org/2").getAsync();

F.Promise<WS.HttpResponse> r3 = WS.url("http://example.org/3").getAsync();

F.Promise<List<WS.HttpResponse>> promises = F.Promise.waitAll(r1, r2, r3);

//暂停处理，直到所有三个远程调用结束

List<WS.HttpResponse> httpResponses = await(promises);

render(httpResponses);

}

}

**回调Callbacks**

还可以使用回调实现上面的示例。这次，await(…)方法包含了play.libs.F.Action实现，当三个远程调用结束后就调用这个回调方法。

public class AsyncTest extends Controller {

public static void remoteData() {

F.Promise<WS.HttpResponse> r1 = WS.url("http://example.org/1").getAsync();

F.Promise<WS.HttpResponse> r2 = WS.url("http://example.org/2").getAsync();

F.Promise<WS.HttpResponse> r3 = WS.url("http://example.org/3").getAsync();

F.Promise<List<WS.HttpResponse>> promises = F.Promise.waitAll(r1, r2, r3);

//暂停处理，直到所有三个远程调用结束

await(promises, new F.Action<List<WS.HttpResponse>>() {

public void invoke(List<WS.HttpResponse> httpResponses) {

render(httpResponses);

}

});

}

}

**HTTP response流 streaming**

既然不用中心请求也可执行循环，你或许会希望向浏览器发送结果变量的部分数据（不是全部）。这就是Content-Type:Chunked HTTP response大量http response类型。它允许你多次使用多个块来发送http response。浏览器将实时接收这些块。

使用await(…)和continuations,就可以实现这个功能:

public static void generateLargeCSV() {

CSVGenerator generator = new CSVGenerator();

response.contentType = "text/csv";

while(generator.hasMoreData()) {

String someCsvData = await(generator.nextDataChunk());

response.writeChunk(someCsvData);

}

}

即使CSV生成需要1个小时，play也能同时使用单个线程处理多个请求，一旦为客户端的数据生成好后，play就会向客户端发送。

**使用WebSockets**

WebSockets是一种在浏览器和应用程序间实现双向通信的途径。在浏览器端使用 “ws://” url：

new Socket("ws://localhost:9000/helloSocket?name=Guillaume")

在play端需要声明一条WS路由:

WS /helloSocket MyWebSocket.hello

MyWebSocket是一个WebSocketController。一个 WebSocket控制器和一个标准的http控制器很相似，但处理的内容不同：

* 它有一个请求对象，但没有response对象
* 它有一个可访问的session，但是只读的
* 它没有renderArgs, routeArgs和flash域
* 它只能从路由模式和QueryString里读取params
* 它拥有两个通信通道：一进一出

当客户连接到ws://localhost:9000/helloSocket套接字时， Play将调用MyWebSocket.hello动作方法。一旦MyWebSocket.hello动作方法存在，套接字就会被关闭

因此一个非常基础的套接字示例应该是这个样子：

public class MyWebSocket extends WebSocketController {

public static void hello(String name) {

outbound.send("Hello %s!", name);

}

}

在这里，当客户端连接到socket时，它将接收到‘Hello Guillaume’消息，play随后将关闭这个socket。

当然，通常情况下你不需要立即关闭socket，用await(…)和continuations也能实现。

比如一个基础的Echo服务器：

public class MyWebSocket extends WebSocketController {

public static void echo() {

while(inbound.isOpen()) {

WebSocketEvent e = await(inbound.nextEvent());

if(e instanceof WebSocketFrame) {

WebSocketFrame frame = (WebSocketFrame)e;

if(!e.isBinary) {

if(frame.textData.equals("quit")) {

outbound.send("Bye!");

disconnect();

} else {

outbound.send("Echo: %s", frame.textData);

}

}

}

if(e instanceof WebSocketClose) {

Logger.info("Socket closed!");

}

}

}

}

在上面的示例里，嵌套的‘if’和‘cast’很乏味，而且容易出错。即使这个简单示例也不容易处理。更复杂的多个联合线程的情况下将会有更多的事件类型，这将是一个恶梦。

这就是为什么我们要向你介绍在[play.libs.F](http://www.playframework.org/documentation/1.2.4/libs#FunctionalprogrammingwithJava)库里的基础的模式匹配的原因。

现在我们重写一下echo示例：

public static void echo() {

while(inbound.isOpen()) {

WebSocketEvent e = await(inbound.nextEvent());

for(String quit: TextFrame.and(Equals("quit")).match(e)) {

outbound.send("Bye!");

disconnect();

}

for(String msg: TextFrame.match(e)) {

outbound.send("Echo: %s", frame.textData);

}

for(WebSocketClose closed: SocketClosed.match(e)) {

Logger.info("Socket closed!");

}

}

}

**11.在play框架里使用Ajax**

play框架允许你很容易执行ajax请求， 它默认使用[jQuery](http://jquery.com)作为实现。本节我们将描述如何在框架里有效使用[jQuery](http://jquery.com)。

play框架同时带来了方便的jsAction标签，用于透明地从控制器得到方法定义。

**通过jsAction标签使用jQuery**

#{jsAction /}标签返回一个JavaScript函数，它包含了基于服务器动作的URL和可用的变量。它不能执行AJAX请求，这些都必须使用返回的URL通过手工进行处理。

示例：

GET /hotels/list Hotels.list

现在就可以导入这些路由客户端：

<script type="text/javascript">

var listAction = #{jsAction @list(':search', ':size', ':page') /}

$('#result').load(

listAction({search: 'x', size: '10', page: '1'}),

function() {

$('#content').css('visibility', 'visible')

}

)

</script>

在这个示例里，我们从默认的Application控制器里请求方法列表。同时传递了三个参数： search, size和page。我们执行的请求操作之后被存入listAction变量。现在就可以使用jQuery和load函数了。

事实上，下面的请求会被发送：

GET /hotels/list?search=x&size=10&page=1

在这种情况下，这个请求将返回HTML数据。

然后，也可使用jQuery对数据进行整理后返回JSON或XML。在控制器里，使用适当的render方法(renderJSON, renderXML或XML模板)

参考[jQuery](http://docs.jquery.com/Main_Page) 文档可获取更多信息。

请注意，我们也可执行POST，对应的jQuery方法应该调整为：

$.post(listAction(), function(data) {

$('#result').html(data);

});

**12.** **Internationalization国际化支持**

国际化 (I18N)意思是允许你的应用程序适用于不同的语言环境。参数下列步骤实现国际化。

**仅使用 UTF-8!**

Play只支持一种编码：UTF-8。因为编码会导致一些古怪的问题，因此我们选择只支持一种编码。UTF-8允许显示所有的语言的所有字符。

请确定在你的应用程序里始终使用的是UTF-8:

* 源文件编辑用的是UTF-8
* 在HTTP里定义了合适的编码headers
* HTML meta标签设置的是UTF-8
* 数据库使用的也是UTF-8而且总是使用UTF-8进行连接

**注意！**   
  
UTF-8编码是导致许多play配置文件、java属性文件都不能命名为\*.properties的原因。Java强制要求属性文件必须是**ISO-8859-1编码，**而Play配置文件必须是UTF-8编码，还需要作更多解释吗？

**国际化你的信息**

为了支持I18N，你必须国际化你的所有信息。

在conf/目录创建一个名叫messages的文件，这个文件是一个真正的java属性文件。

hello=Hello!

back=Back

然后就可以为每种语言定义明确的message文件，只需要为文件名加上ISO语言代码文件后缀。

比如法语为conf/messages.fr，中文为zh，英语为en:

hello=Bonjour!

back=Retour

**通过应用程序定义支持的语言**

在[application.langs configuration](http://www.playframework.org/documentation/1.2.4/configuration#application.langs)里定义支持的语言列表。

新用户第一次进行请求时，play会猜测默认使用的语言。它主要是通过解析HTTP Accept-language header来实现的。然后把猜出的语言存储到PLAY\_LANG cookie.里，以方便用户的再次访问。

也可使用language/country进行区别，比如en\_US、en\_GB,或zh\_CN和zh\_TW。然而应该知道某些用户暴露的语言并不是他能接受的语言，因此，我们应该提供国际通用的语言（比如en）。

比如，如果许多用户来自US，但你又想支持British English，推荐使用 “en”来代表US English，使用 “en\_GB”来代表British English。

在应用程序代码里，可以为用户访问的play.i18n.Lang对象找回当前语言。

String lang = Lang.get();

如果你打算永久改变用户语言，请使用change()方法:

Lang.change("ja");

新值将回存给用户的语言cookie。

**依照你的区域定义日期格式**

配置[date.format](http://www.playframework.org/documentation/1.2.4/configuration#date.format)以指定默认使用的日期格式。

**找回区域信息**

**Message arguments**

在代码里，可以找回在message文件里定义的消息，对java来说，使用的是play.i18n.Messages对象。

public static void hello() {

renderText(Messages.get("hello"));

}

通过标准的java.util.Formatter ‘格式化字符串语法’可以支持消息格式化。也可在messages里定义动态的内容：

hello=Hello %s!

%s将输出一个字符串。Message参数是通过给Messages.get附加（变量）参数来提供的：

public static void hello(String user) {

renderText(Messages.get("hello", user));

}

**模板输出**

在模板里可以使用特定的&{…}语法来显示本地化messages:

<h1>&{'hello'}</h1>

或在消息参数里使用动态：

<h1>&{'hello', params.user}</h1>

**多参数**

可以定义多个消息参数，比如下面的消息引用了两个十进制参数：

guess=Please pick a number between %d and %d

在显示消息参数时依照正确的顺序进行显示：

<p>&{'guess', low, high}</p>

**立即数Argument indices**

为了用于不同的顺序，也可为消息指定明确的参数。比如，假定下面有两个参数：

guess.characteristic=Guess %s’s %s.

消息输出如下：

<p>&{'guess.characteristic', person.name, 'age'}</p>

法国区域的有两个相反顺序的消息，因此在法国区域里，我们指定了立即参数：

guess.characteristic=Devinez %2$s de %1$s.

%2$s输出**第二个**参数作为十进制数。

最后，我们想对‘age’进行限制，因此我们可以使用消息key person.age改变输出，定义如下：

guess.characteristic=Guess %s’s &{%s}.

person.age = age

和

guess.characteristic=Devinez &{%2$s} de %1$s.

person.age = l’age

这里，&{%s}就是它自身，使用消息key作为参数值。

**13.使用cache**

为了创建高效的系统，有里会需要缓存数据。play有一个cache库，在分布式环境中使用[Memcached](http://www.danga.com/memcached/)。

如果没有配置Memcached，play将使用标准的缓存来存储数据到JVM heap堆中。在JVM应用程序里缓存数据打破了play创造的“什么都不共享”的原则：你不能在多个服务器上运行同一应用程序，否则不能保证应用行为的一致性。每个应用实例将拥有不同的数据备份。

清晰理解缓存约定非常重要：当把数据放入缓存时，就不能保证数据永久存在。事实上不能这样做，缓存非常快，只在内存里存在（并没有进行持久化）。

因此使用缓存最好的用途就是数据不需要进行修改的情况

public static void allProducts() {

List<Product> products = Cache.get("products", List.class);

if(products == null) {

products = Product.findAll();

Cache.set("products", products, "30mn");

}

render(products);

}

**The cache API**

缓存API由play.cache.Cache类提供，这个类包含了许多用于从缓存设置、替换、获取数据的方法。参考Memcached文档以了解每个方法的用法。

示例：

public static void showProduct(String id) {

Product product = Cache.get("product\_" + id, Product.class);

if(product == null) {

product = Product.findById(id);

Cache.set("product\_" + id, product, "30mn");

}

render(product);

}

public static void addProduct(String name, int price) {

Product product = new Product(name, price);

product.save();

showProduct(product.id);

}

public static void editProduct(String id, String name, int price) {

Product product = Product.findById(id);

product.name = name;

product.price = price;

Cache.set("product\_" + id, product, "30mn");

showProduct(id);

}

public static void deleteProduct(String id) {

Product product = Product.findById(id);

product.delete();

Cache.delete("product\_" + id);

allProducts();

}

有一些方法是以safe前缀开头的方法，比如safeDelete, safeSet。而标准方法是非阻塞式的，比如：

Cache.delete("product\_" + id);

delete方法将立即返回，不会一直等到缓存对象是否真的删除。因此，如果发生错误时，比如IO错误，那么要删除的对象仍旧存在。

在继续之前如果需要确定是否真的把对象删除了，就可以使用safeDelete方法：

Cache.safeDelete("product\_" + id);

这个方法是阻塞式的，并且会返回一个boolean值来确定是否真的把对象删除了。因此，从缓存中删除对象的完整模式应该是这样的：

if(!Cache.safeDelete("product\_" + id)) {

throw new Exception("Oops, the product has not been removed from the cache");

}

...

注意：这些方法会导致调用中断，安全方法会慢慢的顺着执行下去,一般情况下，只要确实需要的时候才使用这些方法。

同时要注意，当expiration == "0s" (0秒)时， 真正的中止时间可能与缓存执行的时间不同。

**不要把Session当成缓存!**

如果使用框架带来的内存式的session来作缓存， 你就会发现play只允许很小的字符串数据可以存入HTTP Session，这里不应该是缓存应用程序数据的地方！

如果你已经习惯了这样做：

httpServletRequest.getSession().put("userProducts", products);

...

// 之后进行请求

products = (List<Product>)httpServletRequest.getSession().get("userProducts");

在Play里可以用其他方式实现相同效果：

Cache.set(session.getId(), products);

...

//接下来的请求为：

List<Product> products = Cache.get(session.getId(), List.class)

在这里，我们使用唯一的UUID来为每个用户在缓存里保存唯一信息。请记住，这和session对象不同，缓存并不绑定任何特定的用户！

**配置mcached**

如果需要允许真正的Memcached实现，就需在 [memcached configuration](http://www.playframework.org/documentation/1.2.4/configuration#memcached) 定义守护地址 [memcached.host configuration](http://www.playframework.org/documentation/1.2.4/configuration#memcached.host)。

**14.发送e-mail**

play使用[Apache Commons Email](http://commons.apache.org/email/userguide.html)库来实现邮件功能。使用play.libs.Mail工具箱来发送邮件非常容易。

简单邮件示例：

SimpleEmail email = new SimpleEmail();

email.setFrom("sender@zenexity.fr");

email.addTo("recipient@zenexity.fr");

email.setSubject("subject");

email.setMsg("Message");

Mail.send(email);

HTML e-mail示例:

HtmlEmail email = new HtmlEmail();

email.addTo("info@lunatech.com");

email.setFrom(sender@lunatech.com", "Nicolas");

email.setSubject("Test email with inline image");

// 获取内容ID，并嵌入图片

URL url = new URL("http://www.zenexity.fr/wp-content/themes/images/logo.png");

String cid = email.embed(url, "Zenexity logo");

//设置html消息

email.setHtmlMsg("<html>Zenexity logo - <img src=\"cid:"+cid+"\"></html>");

//设置可选消息

email.setTextMsg("Your email client does not support HTML, too bad :(");

更多信息见 [Commons Email documentation](http://commons.apache.org/email/userguide.html)。

**Mail 和MVC 集成**

使用标准的模板机制和语法，也可发送复杂的和动态的邮件。

首先在应用程序里定义一个**Mailer notifier**。mailer notifier必须是play.mvc.Mailer的子类，而且包名必须是notifiers或其子包。

其次，每个e-mail sender的方法都必须是public static的，这个MVC控制的动作相似：

package notifiers;

import play.\*;

import play.mvc.\*;

import java.util.\*;

public class Mails extends Mailer {

public static void welcome(User user) {

setSubject("Welcome %s", user.name);

addRecipient(user.email);

setFrom("Me <me@me.com>");

EmailAttachment attachment = new EmailAttachment();

attachment.setDescription("A pdf document");

attachment.setPath(Play.getFile("rules.pdf").getPath());

addAttachment(attachment);

send(user);

}

public static void lostPassword(User user) {

String newpassword = user.password;

setFrom("Robot <robot@thecompany.com>");

setSubject("Your password has been reset");

addRecipient(user.email);

send(user, newpassword);

}

}

**text/html e-mail**

调用send方法将会渲染app/views/Mails/welcome.html模板作为邮件消息体。

<html><body><p>Welcome <b>${user.name}</b>, </p>

...

</html>

lostPassword方法的模板代码：

app/views/Mails/lostPassword.html

<html>

<body><head>...</head><body>

<img src="mycompany.com/images"/>

<p>

Hello ${user.name}, Your new password is <b>${newpassword}</b>.

</p>

</body>

</html>

**text/plain e-mail**

如果没有定义HTML模板，那么text/plain邮件将使用text模板进行发送。

调用send方法将会渲染app/views/Mails/welcome.txt模板作为邮件消息体。

Welcome ${user.name},

...

lostPassword方法的模板应该是这个样子： app/views/Mails/lostPassword.txt

Hello ${user.name},

Your new password is ${newpassword}.

**text/html e-mail with text/plain alternative**

如果HTML模板已经定义，同时又存在text模板，那么text模板将用于可选消息。在上一个示例里，如果app/views/Mails/lostPassword.html和app/views/Mails/lostPassword.txt都已定义，那么在发送邮件时默认将用lostPassword.html模板以text/html方式发送，同时可自由选择lostPassword.txt模板。因此你可以向好朋友用友好的HMTL模板，而对待讨厌的人可以选择使用text模板。

**在应用程序里链接到邮件**

使用如下语法可在应用程序里链接到邮件：

@@{application.index}

如果要从job里发送邮件，就必须为应用程序设置[application.baseUrl](http://www.playframework.org/documentation/1.2.4/configuration#application.baseUrl)为一个有效的基于URL的地址。

比如，从运行于play站点的一个Job发送邮件，其配置应该是这样的：

application.baseUrl=http://www.playframework.org/

**SMTP配置**

邮件功能可通过配置多个 [mail configuration](http://www.playframework.org/documentation/1.2.4/configuration#mail) 属性开启：

* SMTP server - [mail.smtp.host](http://www.playframework.org/documentation/1.2.4/configuration#mail.smtp.host)
* SMTP server 验证 - [mail.smtp.user](http://www.playframework.org/documentation/1.2.4/configuration#mail.smtp.user) 和 [mail.smtp.pass](http://www.playframework.org/documentation/1.2.4/configuration#mail.smtp.pass)
* 加密通道 - [mail.smtp.channel](http://www.playframework.org/documentation/1.2.4/configuration#mail.smtp.channel)
* JavaMail SMTP 登录事务 - [mail.debug](http://www.playframework.org/documentation/1.2.4/configuration#mail.debug).

下面两个配置用于让你重载默认的行为：

* [mail.smtp.socketFactory.class](http://www.playframework.org/documentation/1.2.4/configuration#mail.smtp.socketFactory.class)
* [mail.smtp.port](http://www.playframework.org/documentation/1.2.4/configuration#mail.smtp.port)

默认情况下，在DEV模式时，邮件将打印到控制台，在PROD模板下，将会真正发送到真实的SMTP server。在DEV模式下通过注释下面的配置也可改变默认的行为：

# Default is to use a mock 模拟Mailer

mail.smtp=mock

**使用 Gmail**

为了使用Gmail的服务器，你需要作如下配置：

mail.smtp.host=smtp.gmail.com

mail.smtp.user=yourGmailLogin

mail.smtp.pass=yourGmailPassword

mail.smtp.channel=ssl

**15.测试应用程序**

创建自动测试套件对应用程序来说是保持程序强健的一个好方式。

Play测试构建于[JUnit 4](http://www.junit.org/) 和 [Selenium](http://seleniumhq.org/) 。

**书写测试程序**

测试程序必须创建于test/目录。当play运行于测试模式时，这个文件夹是唯一被增加到源路径的文件夹。你可以创建三种类型的测试。

**单元测试**

单元测试是用于JUnit写的。在这种情况下，可用于测试应用程序的模型和一些工具箱测试。

示例：

import play.test.\*;

import org.junit.\*;

public class MyTest extends UnitTest {

@Test

public void aTest() {

assertEquals(2, 1 + 1); // 测试真正关心的地方

}

@Test

public void testUsers() {

assertEquals(3, Users.count());

}

}

**功能性测试**

功能性测试是用JUnit写的。在这种情况下，可以通过直接访问控制器对象进行测试。

示例：

import play.test.\*;

import play.mvc.\*;

import play.mvc.Http.\*;

import org.junit.\*;

public class ApplicationTest extends FunctionalTest {

@Test

public void testTheHomePage() {

Response response = GET("/");

assertStatus(200, response);

}

}

使用renderArgs()方法可以直接访问传递到视图层的参数，用于代替response自我断言，比如:

@Test

public void testUserIsFoundAndPassedToView() {

Response response = POST("/user/find?name=mark&dob=18011977")

assertThat(renderArgs("user"), is(notNullValue());

User user = (User) renderArgs("user");

assertThat(user.name, is("mark"));

}

**Selenium test用例测试**

Acceptance认同测试是用Selenium写的。在这里，你可以在一个自动化的浏览器对应用程序进行测试。

Selenium测试用HTML表书写，即可使用其自有语法，也可使用#{selenium /}标签。

示例：

#{selenium 'Test security'}

//试着登录到超级用户区域

clearSession()

open('/admin')

assertTextPresent('Login')

type('login', 'admin')

type('password', 'secret')

clickAndWait('signin')

//验证用户是否已经正常登录

assertText('success', 'Welcom admin!')

#{/selenium}

因为Selenium测试运行于浏览器内，要想访问模拟邮件和放入play缓存的字符串值，就必须使用Selenium扩展。

下面示例用于访问发送到指定帐号的最新邮件：

#{selenium 'Test email sending'}

// Open email form and send an email to boron@localhost

open('/sendEmail')

assertTextPresent('Email form')

type('To', 'boron@localhost')

type('Subject', 'Berillium Subject')

clickAndWait('send')

// Extract the last email sent to boron@localhost into a JavaScript

// variable called email

storeLastReceivedEmailBy('boron@localhost', 'email')

// Extract the subject line from the email variable into a variable

// called subject

store('javascript{/Subject:\s+(.\*)/.exec(storedVars["email"])[1]}', 'subject')

// Test the contents of the subject variable

assertEquals('Berillium Subject', '$[subject]')

#{/selenium}

下面的示例访问一个存储在play Cache的字符串：

#{selenium 'Get string from cache'}

open('/register')

assertTextPresent('Registration form')

type('Email', 'my@email.com')

type('Password', 'secretpass')

type('Password2', 'secretpass')

// .. Fill in the registration form ..

// Get the value of the magicKey variable from the cache

// (set to the CAPTCHA answer in the application)

storeCacheEntry('magicKey', 'captchaAnswer')

// Type it into the form

type('Answer', '$[captchaAnswer]')

clickAndWait('register')

#{/selenium}

**Fixtures固定值**

当运行测试时，需要为应用程序准备一些可用的数据。这里有一个非常简单的方式来实现每次测试时重新设置数据库。

play.test.Fixtures类用于帮助维护数据库并注入测试数据。在JUint测试方法里特定使用@Before注释。

@Before

public void setUp() {

Fixtures.deleteAll();

}

为了能导入数据，可以在YAML文件里进行定义，然后使用Fixtures帮助类自动导入。

# Test data

Company(google):

name: Google

Company(zen):

name: Zenexity

User(guillaume):

name: guillaume

company: zen

然后:

@Before

public void setUp() {

Fixtures.deleteAll();

Fixtures.loadModels("data.yml");

}

了解[YAML manual page](http://www.playframework.org/documentation/1.2.4/yaml)。

对Selenium测试来说，可以使用#{fixture /} 标签：

#{fixture delete:'all', load:'data.yml' /}

#{selenium}

// Write your test here

#{/selenium}

有时为了方便，可以把分割的数据放入不同的YAML文件中，并一次性用fixtures进行加载：

Fixtures.loadModels("users.yml", "roles.yml", "permissions.yml");

and for Selenium tests:

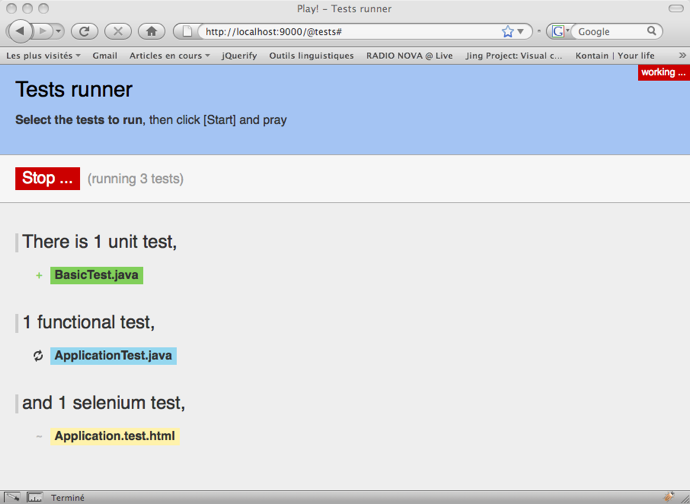
#{fixture delete:'all', load:['users.yml', 'roles.yml', 'permissions.yml'] /}

**运行测试**

使用play命令，如下：

# play test myApp

在这种模式下，Play将自动加载test-runner模块，此模块提供了一个基于web的测试运行，通过 <http://localhost:9000/@tests> URL进行访问。



当运行测试的时候，其结果将会存储在/test-result目录里。

在测试运行页面上，每个测试都是一个链接。右键》打开到新标签可以直接在外部运行测试。

当用这种方式运行测试时，play将启动特定的test framework ID，因此你可以在application.conf里定义特定的配置。

如果打算设置多个不同的测试配置，你可以使用framework IDs匹配test-?.\* (e.g: test-special)来实现。

如果使用了其他的framework ID作为默认的test框架，你必须确定所有在application.conf里的test配置对framework ID来说都是可用的。当加载特定的framework ID进行测试时，命令应该为：

play test --%test-your-special-id

比如：

%test.db=mem

%test.jpa.ddl=create-drop

**陆续集成,并自动运行测试**

auto-test命令和test命令做的是同一件事，只是auto-test会自动加载浏览器，运行所有的测试并停止。

如果你打算设置一个continuous integration system陆续集成系统的话，这是一个很有用的命令。

开始运行后，所有的结果都存储在/test-result目录下。此外，这个目录为测试组件的最终结果创建了标志文件 (either result.failed or result.passed) 。最后，这个目录下的application.log文件包含了所有的日志。

为了设置一个continuous integration system持续集成系统，其步骤如下：

* 检查最后一个版本的应用程序
* 运行play auto-test
* 等待处理结束
* 检查/test-result目录下的标志文件result.passed 或 result.failed

在CRON tab里运行这些步骤就完成了~！

通过配置[headlessBrowser](http://www.playframework.org/documentation/1.2.4/configuration#headlessBrowser)，可以改变浏览器兼容模式。

**16.安全指南**

play框架的安全设计只是存在于脑海中的一个概念——这就不可能阻止开发者在设计上出漏洞。本指南描述了在Web应用中常见的安全问题，并指导如何在play应用开发中去避免。

**Sessions**

一般情况下，你需要保存用户登录信息。如果不使用session，用户就需要在每次请求时传递证书。

这就是session要做的事：一系列cookies存储在用户的浏览器里，以用于标识不同的站点，并提供其他数据层之外的信息，比如语言。

**守住你的安全…安全**

session是一个key/values哈希表，有签名但没有加密。也就是说主要你的secret是安全的，那么第三方就不可能伪造session。

而secret存储在conf/application.conf里。保持这个文件的私有化非常重要：不要把它提交给公共仓库，为防止在安装某些人写的应用程序时对secret key进行修改，你可以运行命令play secret进行保护。

**不要存储关键性的数据**

然后，既然session没有加密，那么就不应该在session里存储关键性安全数据，通过本地网络连接或wifi连接进行嗅探，它将被看成是用户的cookie。

session被存储在cookie里, 而cookies被限制为4 KB大小。而且只能存储String。

**跨站点脚本攻击**

在web应用程序中，Cross-site scripting跨站点脚本是最常见的弱点。它包含了利用应用程序提供的窗体恶意注入到web页面的脚本代码。

假定这里有一个博客程序，任何人都可以发表评论，如果评论内容里包含了攻击脚本，那么你将打开你的站点进行攻击，可能步骤为：

* 为访问者弹出一个窗体
* 跳转访问者到被攻击者控制的站点
* 窃取当前用户的可见信息，并发回攻击者的站点

因此为防止类似攻击，进行保护非常重要。

play的模板引擎会自动转义字符串。如果需要在模板里插入未转义的HTML代码，那就需要使用java字符串扩展的 [raw()](http://www.playframework.org/documentation/1.2.4/javaextensions#raw) 方法。但如果这个字符串是用户输入的，那么你就需要首先确定它是否是无害的。

在对用户输入进行sanitizing消毒的时候，只允许白名单（只允许列表中的安全标签通过）通过，黑名单 (禁止一些不安全的标签列表)则禁止。

[详见cross-site scripting](http://en.wikipedia.org/wiki/Cross-site_scripting)。

**SQL注入**

SQL注入所谓SQL注入，就是通过把SQL命令插入到Web表单递交或输入域名或页面请求的查询字符串，最终达到欺骗服务器执行恶意的SQL命令，比如先前的很多影视网站泄露VIP会员密码大多就是通过WEB表单递交查询字符暴出的，这类表单特别容易受到SQL注入式攻击。这个漏洞可以摧毁数据，或访问不应该让当前用户看到的数据。



当使用高级的“find”方法时，就应该小心sql注入。当手工构建自己的查询时，尽量不要用+来作连接符，而是用?占位符。

比如:

createQuery("SELECT \* from Stuff WHERE type= ?1").setParameter(1, theType);

下面这个就不好了:

createQuery("SELECT \* from Stuff WHERE type=" + theType;

**跨站点请求伪造**

跨站点请求伪造CSRF攻击在web应用程序里非常危险：

这个攻击方法通过在用户信任的web应用程序的页面里加入恶意代码或链接进行攻击。如果web应用程序的session没有设置超时，那么攻击者就可以执行未经验证的命令。

为预防这类攻击，首先就是要正确全用GET和POST方法。意思是仅允许POST方法用于修改应用程序的状态。

对POST请求来说，只有真实的token才会触发动作。Play现在内建了一个帮助类用于处理这样的问题：

* checkAuthenticity() 检查真实性方法在控制器里可直接使用，用于检查请求参数里的token的真实有效，如果存在问题，就发送一个禁止response的命令。
* session.getAuthenticityToken()方法用于为当前session生成一个真实可信的token
* #{authenticityToken /}用于创建一个隐藏的输入域，可用于任何窗体

示例：

public static destroyMyAccount() {

checkAuthenticity();

…

}

上面的方法仅当窗体包含了真实有效的token时，才会被调用:

<form method="post" action="/account/destroy">

#{authenticityToken /}

<input type="submit" value="destroy my account">

</form>

对POST请求来说，Play提供form标签会自动生成一个有效的token:

#{form @destroyMyAccount()}

<input type="submit" value="destroy my account">

#{/form}

如果你想保护所有控制器的action动作，你也可以使用checkAuthenticity()方法作为[before filter](http://www.playframework.org/documentation/1.2.4/controllers#before)进行听说。

更多跨站点请求伪造见：[More on cross-site request forgery](http://en.wikipedia.org/wiki/Cross-site_request_forgery)。

**17.Play模块和模块仓库**

Play应用程序可以由多个应用模块装配而成。这就允许你重用应用组件到多个应用程序中去，或把一个大的应用程序分散到多个小的应用程序里。

**什么是模块?**

模块就是另一个play应用程序，一个模块在加载时会同时加载分散到各个地方的资源：

* 模板不能有conf/application.conf文件
* 模板可以有conf/routes文件，但这些路由不会被自动加载
* 在主程序路径下的所有文件都会被首先检索到，然后再检索所有要加载的模块
* 模板可以包含原始的java代码，前提是以jar文件形式打包到module/lib目录下
* 模块可以包含文档页面
* 每个模块都是可选的

使用play new-module命令可以创建自己的模块。

**如何从一个应用程序里加载模块**

位于应用程序/modules目录下的模块是被自动加载，也可使用[dependency management system](http://www.playframework.org/documentation/1.2.4/dependency)来自动管理你的应用程序模块。

**从模块加载默认的routes**

模块可以提供一个默认的routes文件。使用如下的声明，可以把它加载到主应用程序的routes文件里:

# Import the default CRUD routes

GET /admin module:crud

下面的声明将加载所有的可用的模块：

GET / module:\*

**为模块增加文档说明**

简单为模块添加documentation/manual/home.textile ，可以为模块增加一个文档页面。play的文档使用的就是相同的Textile语法，见： ${play.path}/documentation/manual/

如果你正在运行一个play应用程序，并且使用了多个带有文档的模块，那么本地的play文档（<http://localhost:9000/@documentation>）将包含这些模块的文档页面链接，具体位置是侧边栏的**Installed Modules** 。

**使用模块仓库**

[module repository](http://www.playframework.org/modules) 定义了所有的通用分布式模块。一个模块可以拥有多个版本。你必须检查模块的文档，以确定哪个版本适用于你的框架版本。

使用play list-modules 命令可以浏览模块仓库里的所有模块：

gbo-mac:~ guillaume$ play list-modules

~ \_ \_

~ \_ \_\_ | | \_\_ \_ \_ \_| |

~ | '\_ \| |/ \_' | || |\_|

~ | \_\_/|\_|\\_\_\_\_|\\_\_ (\_)

~ |\_| |\_\_/

~

~ play! 1.2, http://www.playframework.org

~

~ You can also browse this list online at http://www.playframework.org/modules

~

~ [bespin]

~ Bespin online editor

~ http://www.playframework.org/modules/bespin

~ Versions: 1.0, 1.0.1

~

~ [cobertura]

~ Cobertura

~ http://www.playframework.org/modules/cobertura

~ Versions: 1.0

...

在本地使用 play install {module}-{version}命令可以在本地安装模块。在本地安装模块后就可以模块用于不同的应用程序，而且不需要为每个应用程序复制备份。这对于大型模块来讲非常有用，这与框架扩展非常相似。

比如，在框架里安装Scala支持：

play install scala-head

我们约定head版本就模块的不稳定版本。你也可通过省略版本信息安装默认的模块版本，如：

play install scala

通过这种方式安装的模块将会直接下载模块到你的框架安装目录下的/modules文件夹里。

使用—path选项可以更改默认安装路径：

play install gwt --path=my-project

**贡献新模块到模块仓库里**

**先决条件**

要注册一个新模块，以下步骤是必须的：

1. 一个Google帐号，用于向Google Group传送内容
2. 一个开发者log-in OpenID (使用Google account也行)
3. 模块名称要符合 [a-zA-Z]+ 正则表达式要求
4. 模块必须包含有[documentation](http://www.playframework.org/documentation/1.2.4/modules#documentation)
5. 有模块源代码主页和报送bug的渠道，比如 [GitHub](http://github.com/), [Google Code](http://code.google.com/) 或 [Launchpad](http://launchpad.net/)项目。

**模块注册**

使用Google account作为OpenID，需要使用完整的URL，步骤为：

1. 在<http://www.playframework.org/modules>页面里，在*Developer login* 下输入https://www.google.com/accounts/o8/id ，单击*Login* 按钮
2. 登录到你的Google account
3. 在*Developer login*下创建一个完整的Google OpenID URL，如：https://www.google.com/accounts/o8/id?id=BItOawk7q69CFhRarQIo

在[play-framework Google Group](https://groups.google.com/forum/#%21forum/play-framework)页面里传送一个模块注册请求，比如：

Subject: Module registration request: {module name}

Module name:

jqueryui

Display name:

jQuery UI

Description:

Working examples of jQuery UI widgets, integrated with a Play application.

Project home page:

https://github.com/hilton/jqueryui-module

OpenID:

https://www.google.com/accounts/o8/id?id=BItOawk7q69CFhRarQIo

当模块成功注册后，就可以发布模块了。

**发布你的模块**

步骤：

1. 在conf/dependency.yml文件的第一行设置模块版本号, 如：self: play -> jqueryui 1.0
2. 用play build-module命令创建模块
3. 在*Developer login*下，登录到<http://www.playframework.org/modules>
4. 在*Your modules*下通过<http://www.playframework.org/modules/developers>浏览你的模块的页面
5. 使用*Manage module releases*下的窗体上传你的模块（已经把你的模块/dist目录打包成普通的zip文件）

也可通过使用官方的Google Group提供的帮助来共享你的工作信息。

**18.依赖管理**

Play的依赖管理系统允许你通过一个单独的dependencies.yml 文件来描述你的应用程序所采用的扩展依赖。

Play应用程序可以有三种类型的依赖：

* Play框架自身，这是因为play应用程序总是要依赖play框架
* 任何提供了jar文件并复制到应用程序lib/目录下的java库
* 安装在应用程序modules目录下的Play模块 module (实现是一些程序碎片)

一旦在conf/dependencies.yml文件里描述了应用程序的依赖，Play将会进行检索、下载并安装所有需要的依赖。

**依赖格式**

在dependencies.yml文件里用开发模拟的组织名称+所依赖的模块名称和对应的版本号就可以描述一个依赖关系，如：

organisation -> name revision

因此，1.0版本的[Play PDF module](http://www.playframework.org/modules/pdf)应该描述为：

play -> pdf 1.0

有些时候组织名称和依赖名称完全一致，比如[commons-lang](http://commons.apache.org/lang/)：

commons-lang -> commons-lang 2.5

在这种情况下，你可以省略组织名称：

commons-lang 2.5

**动态版本**

修订本可以是已经修订好的(应用于play1.2版本)或是动态的。一个动态的修订本的表示需要指定范围，比如：

* [1.0,2.0] 匹配1.0至2.0版本（含1.0和2.0）
* [1.0,2.0[匹配1.0至2.0版本（含1.0，不含2.0）
* ]1.0,2.0] 匹配1.0至2.0版本（不含1.0，但含2.0）
* ]1.0,2.0[匹配1.0至2.0版本（不含1.0和2.0）
* [1.0,) 匹配1.0及以上版本
* ]1.0,) 匹配1.0以上版本
* (,2.0] 匹配2.0及以下版本
* (,2.0[匹配2.0以下版本

**dependencies.yml**

当创建一个新play应用程序时，系统会自动在conf目录下创建一个dependencies.yml描述文件，其内容如下：

# Application dependencies

require:

- play 1.2

require节用于列出所有的依赖。在这里，新创建的play只依赖**Play version 1.2**，如果应用程序需要依赖[Google Guava](http://code.google.com/p/guava-libraries/)，那么应该修改为：

# Application dependencies

require:

- play 1.2

- com.google.guava -> guava r07

**‘play dependencies’命令**

为了让play检索、下载并安装上面描述的新依赖，请运行play dependencies：

$ play dependencies

~ \_ \_

~ \_ \_\_ | | \_\_ \_ \_ \_| |

~ | '\_ \| |/ \_' | || |\_|

~ | \_\_/|\_|\\_\_\_\_|\\_\_ (\_)

~ |\_| |\_\_/

~

~ play! 1.2, http://www.playframework.org

~ framework ID is gbo

~

~ Resolving dependencies using ~/Documents/coco/conf/dependencies.yml,

~

~ com.google.guava->guava r07 (from mavenCentral)

~ com.google.code.findbugs->jsr305 1.3.7 (from mavenCentral)

~

~ Downloading required dependencies,

~

~ downloaded http://repo1.maven.org/maven2/com/google/guava/guava/r07/guava-r07.jar

~ downloaded http://repo1.maven.org/maven2/com/google/code/findbugs/jsr305/1.3.7/jsr305-1.3.7.jar

~

~ Installing resolved dependencies,

~

~ lib/guava-r07.jar

~ lib/jsr305-1.3.7.jar

~

~ Done!

~

这时，play会从Maven仓库里为[Google Guava](http://code.google.com/p/guava-libraries/)下载两个jar文件（**guava-r07.jar**, **jsr305-1.3.7.jar**) ，并将它们安装到应用程序的lib目录下。

为什么只声明了一个依赖，但play却下载了两个jars文件？这是因为Google Guava还有一个透明依赖。事实上jsr305-1.3.7.jar我们并不需要，我们可以排除它。

**透明依赖**

默认情况下，任何透明依赖都会被play自动恢复和找回。如果需要，可以使用以下方式排除它们：

1. 在描述时使用particular:false设置来实现：

# Application dependencies

require:

- play 1.2

- com.google.guava -> guava r07:

transitive: false

2. 对整个项目禁用透明依赖：

# Application dependencies

transitiveDependencies: false

require:

- play 1.2

- com.google.guava -> guava r07

3.或排除特定依赖：

# Application dependencies

require:

- play 1.2

- com.google.guava -> guava r07:

exclude:

- com.google.code.findbugs -> \*

**保持lib/和modules/目录同步**

现在如果再次运行play dependencies命令时，jsr305-1.3.7.jar依赖将被忽略：

$ play deps

~ \_ \_

~ \_ \_\_ | | \_\_ \_ \_ \_| |

~ | '\_ \| |/ \_' | || |\_|

~ | \_\_/|\_|\\_\_\_\_|\\_\_ (\_)

~ |\_| |\_\_/

~

~ play! 1.2, http://www.playframework.org

~ framework ID is gbo

~

~ Resolving dependencies using ~/Documents/coco/conf/dependencies.yml,

~

~ com.google.guava->guava r07 (from mavenCentral)

~

~ Installing resolved dependencies,

~

~ lib/guava-r07.jar

~

~ \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

~ WARNING: Your lib/ and modules/ directories and not synced with

~ current dependencies (use --sync to automatically delete them)

~

~ Unknown: ~/Documents/coco/lib/jsr305-1.3.7.jar

~ \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

~

~ Done!

~

然而之前下载的**jsr305-1.3.7.jar**仍旧在lib/目录里。

play的依赖管理系统负责保持lib/和modules/目录同步，这里可以使用--sync作为play dependencies命令的参数：

play deps --sync

如果再次运行play deps命令，不想要的jar文件将会被删除。

当部署应用程序到生产环境时，可以通过删除模块源代码和模块文档来减少模块的大小，方法是使用--forProd参数运行如下命令：

play dependencies --forProd

这时，将删除每个模块下的documentation/, src/, tmp/, \*sample\*/ 和\*test\*/ 等目录。

**冲突判定Conflict resolution**

当两个组件需要不同修订版本的同一个依赖时，冲突管理只选择一个作为最终依赖。默认情况下是使用最新版本，放弃其他版本。

但这里有一个例外情况：当play框架自己的核心依赖发生冲突时，在$PLAY/framework/lib目录下的可用版本具有最高优先级。比如，Play依赖于commons-lang 2.5，如果你的应用程序需要commons-lang 3.0时：

# Application dependencies

require:

- play 1.2

- com.google.guava -> guava r07:

transitive: false

- commons-lang 3.0

运行play dependencies命令将不会采用commons-lang 3.0，虽然它是最新的：

play dependencies

~ \_ \_

~ \_ \_\_ | | \_\_ \_ \_ \_| |

~ | '\_ \| |/ \_' | || |\_|

~ | \_\_/|\_|\\_\_\_\_|\\_\_ (\_)

~ |\_| |\_\_/

~

~ play! 1.2, http://www.playframework.org

~ framework ID is gbo

~

~ Resolving dependencies using ~/Documents/coco/conf/dependencies.yml,

~

~ com.google.guava->guava r07 (from mavenCentral)

~

~ Some dependencies have been evicted驱逐,

~

~ commons-lang 3.0 is overriden by commons-lang 2.5

~

~ Installing resolved dependencies,

~

~ lib/guava-r07.jar

~

~ Done!

~

同样要注意，在$PLAY/framework/lib下可用的依赖将不会安装到你的应用程序的lib目录下。

有时你可能需要强制指定一个依赖的版本，要么是为了覆盖核心依赖，或是为了选择最新发布的修订版本。

这时就可以在某个依赖上指定force选项：

# Application dependencies

require:

- play 1.2

- com.google.guava -> guava r07:

transitive: false

- commons-lang 3.0:

force: true

**增加要的仓库**

默认情况下，play将搜索[central Maven repository](http://repo1.maven.org/maven2/)下的所有jar依赖，同时搜索[central Play modules repository](http://www.playframework.org/modules)下的所有**Play模块。**

在配置文件的repositories节里，你可以指定新的定制仓库：

# Application dependencies

require:

- play 1.2

- com.google.guava -> guava r07:

transitive: false

- commons-lang 3.0:

force: true

- com.zenexity -> sso 1.0

# My custom repositories

repositories:

- zenexity:

type: http

artifact: "http://www.zenexity.com/repo/[module]-[revision].[ext]"

contains:

- com.zenexity -> \*

使用这个配置后，所有com.zenexity organisation 下的依赖都会被检索，并且从远程http服务器进行下载。

**Maven仓库**

使用iBiblio（公共库和数字档案）类型，也可以增加maven2-compatible兼容仓库，比如：

# Application dependencies

require:

- play

- play -> scala 0.8

- org.jbpm -> jbpm-persistence-jpa 5.0.0:

exclude:

- javassist -> javassist \*

- org.hibernate -> hibernate-annotations \*

- javax.persistence -> persistence-api \*

repositories:

- jboss:

type: **iBiblio**

root: "http://repository.jboss.org/nexus/content/groups/public-jboss/"

contains:

- org.jbpm -> \*

- org.drools -> \*

**本地仓库**

最后，最重要的是你可能想要定义一个引用本地模块的仓库。就和在application.conf配置文件中配置模块一样（**play1.2开始不再赞成使用**），依赖可以很好工作。

首先创建以下目录结构：

myplayapp/

myfirstmodule/

mysecondmodule/

然后如下设置myplayapp/conf/depencencies.yml就可以实现了：

# Application dependencies

require:

- play

- myfirstmodule -> myfirstmodule

- mysecondmodule -> mysecondmodule

repositories:

- My modules:

type: local

artifact: ${application.path}/../[module]

contains:

- myfirstmodule

- mysecondmodule

**注意**：不要忘记了运行play dependencies myplayapp命令。

**定制ivy设置（Apache ivy:项目依赖管理工具）**

Play在hood下使用Ivy来管理项目依赖。如果你需要指定一个特定配置，比如调协一个代理proxy用于验证内部maven网络仓库,那么你可以修改ivysettings.xml文件，这个文件位于play主目录的.ivy2目录。

示例1，让ivy忽略checksums:

<!-- .ivy2/ivysettings.xml -->

<ivysettings>

<property name="ivy.checksums" value=""/>

</ivysettings>

示例2，使用基本验证：

<!-- .ivy2/ivysettings.xml -->

<ivysettings>

<credentials host="maven-repo.xxx" realm="Sonatype Nexus Repository Manager"

username="user" passwd="reallygreatpassword"/>

</ivysettings>

示例3，重用本地maven仓库，并进行仓库管理：

<!-- .ivy2/ivysettings.xml -->

<ivy-settings>

<!-- path to local maven repo and default maven layout -->

<property name="local-maven2-pattern"

value="${user.home}/.m2/repository/[organisation]/[module]/[revision]/[module]-[revision]"

override="false" />

<!-- set resolver chain as default -->

<settings defaultResolver="main" />

<!-- configure caches -->

<caches repositoryCacheDir="${user.home}/.ivy2/cache">

<!-- do not cache from local .m2-->

<cache name="nocache" useOrigin="true" />

<cache name="default" />

</caches>

<resolvers>

<chain name="main">

<!-- as this is not cached, even changing SNAPSHOT dependencies

are resolved correctly -->

<filesystem name="local-maven-2" m2compatible="true" local="true"

cache="nocache">

<ivy pattern="${local-maven2-pattern}.pom" />

<artifact pattern="${local-maven2-pattern}(-[classifier]).[ext]" />

</filesystem>

<!-- use repository manager as proxy to maven-central

(and all other repositories)-->

<ibiblio name="repomanager" m2compatible="true"

root="http://your.repomanager.intra/path/to/repo" cache="default"/>

</chain>

</resolvers>

</ivy-settings>

详见 [Ivy settings documentation](http://ant.apache.org/ivy/history/2.1.0/settings.html)。

**清除Ivy缓存**

Ivy缓存可能会损坏，特别是在conf/dependencies.yml文件里的repositories节使用http类型时很容易损坏。如果发生这样的事，并且依赖解决方案不能正常工作时，可以使用—clearcache选项清除ivy缓存。

$ play dependencies --clearcache

这个命令等效于rm -r ~/.ivy2/cache。

**19.管理数据库变化Evolution**

当使用关系数据库时，你需要去跟踪和安排数据库schema（结构）变化，特别是有多个存储位置的情况下，你就需要更多的经验来跟踪数据的schema变化：

* 当处理团队合作进行开发时，每个人都需要知道数据库结构的变化
* 当部署到生产服务器上时，就需要一个稳健的方式去更新数据库结构
* 如果在多台数据库服务器上工作时，就需要保持所有数据库结构同步

如果在JPA下工作，Hibernate会自动为你处理好这些数据库变化。如果你不打算使用JPA或打算手工对数据库结构进行更好的调整，那么Evolutions将非常有用。

**Evolutions脚本**

Play使用**evolutions 脚本**来跟踪你的数据库变化。这些脚本采用的是原始的sql语句来书写的，应该位于应用程序的db/evolutions目录。

第一个脚本名叫1.sql,第二为2.sql,以此类推…

每个脚本包含了两部分：

* **Ups** 部分用于描述必要的转换
* **Downs** 部分用于描述如何恢复他们

比如，查看第一个evolution脚本，这个脚本用于引导一个基本的应用：

# Users schema

# --- !Ups

CREATE TABLE User (

id bigint(20) NOT NULL AUTO\_INCREMENT,

email varchar(255) NOT NULL,

password varchar(255) NOT NULL,

fullname varchar(255) NOT NULL,

isAdmin boolean NOT NULL,

PRIMARY KEY (id)

);

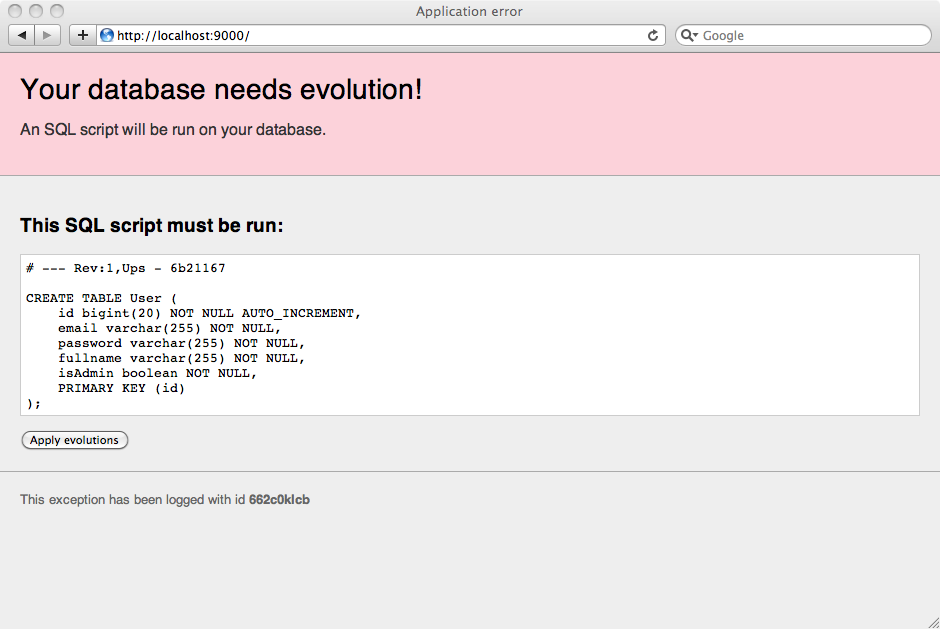
# --- !Downs

DROP TABLE User;

正如你看到的一样，必须在sql脚本里使用注释来界定**Ups**和**Downs** 节。

如果在applictaion.conf里配置了数据库，那么Evolutions将被自动激活，而且evolution scripts将被显示出来。通过设置[evolutions.enabled](http://www.playframework.org/documentation/1.2.4/configuration#evolutions.enabled) 为false可以禁此显示脚本。比如，当测试他们自己的数据库里，你可以为测试环境设置为禁用模式。

当**evolutions** 被激活后， 在DEV模式下，Play将为每个请求检查数据库结构状态，或在PROD模式下启动应用程序时进行数据库结构状态检查。在DEV模式下，如果数据库结构不是最新的，一个错误页面会显示出来，它会建议你运行合适的sql脚本同步你的数据结构。



如果你同意推荐的SQL脚本，你可以点击‘Apply evolutions’按钮执行该建议脚本。

如果使用的是内存数据库(**db=mem**),并且数据库是空的情况下，Play会自动运行所有的evolutions脚本。

**同步同时发生的改变**

现在让我们假定现在在同一项目中有两个开发者，A开发者因工作需要必须创建一个新的数据库表，因此他将2.sql evolution 脚本:

# Add Post

# --- !Ups

CREATE TABLE Post (

id bigint(20) NOT NULL AUTO\_INCREMENT,

title varchar(255) NOT NULL,

content text NOT NULL,

postedAt date NOT NULL,

author\_id bigint(20) NOT NULL,

FOREIGN KEY (author\_id) REFERENCES User(id),

PRIMARY KEY (id)

);

# --- !Downs

DROP TABLE Post;

Play将应用这个evolution脚本到A开发者的数据库。

另外一方面，B开发者因工作需要，必须删除User表。因此他也将创建2.sql evolution脚本：

# Update User

# --- !Ups

ALTER TABLE User ADD age INT;

# --- !Downs

ALTER TABLE User DROP age;

B开发者结束开发工作后进行了提交（称为Git)。现在，A开发者在继续开发前必须合并他的同事的工作，因此他运行git pull，但合并操作产生了一个冲突，如下：

Auto-merging db/evolutions/2.sql

CONFLICT (add/add): Merge conflict in db/evolutions/2.sql

Automatic merge failed; fix conflicts and then commit the result.

每个开发者都创建了一个2.sql evolution脚本。因此，A开发者需要合并这个文件：

<<<<<<< HEAD

# Add Post

# --- !Ups

CREATE TABLE Post (

id bigint(20) NOT NULL AUTO\_INCREMENT,

title varchar(255) NOT NULL,

content text NOT NULL,

postedAt date NOT NULL,

author\_id bigint(20) NOT NULL,

FOREIGN KEY (author\_id) REFERENCES User(id),

PRIMARY KEY (id)

);

# --- !Downs

DROP TABLE Post;

=======

# Update User

# --- !Ups

ALTER TABLE User ADD age INT;

# --- !Downs

ALTER TABLE User DROP age;

>>>>>>> devB

这个合并操作其实很容易做:

# Add Post and update User

# --- !Ups

ALTER TABLE User ADD age INT;

CREATE TABLE Post (

id bigint(20) NOT NULL AUTO\_INCREMENT,

title varchar(255) NOT NULL,

content text NOT NULL,

postedAt date NOT NULL,

author\_id bigint(20) NOT NULL,

FOREIGN KEY (author\_id) REFERENCES User(id),

PRIMARY KEY (id)

);

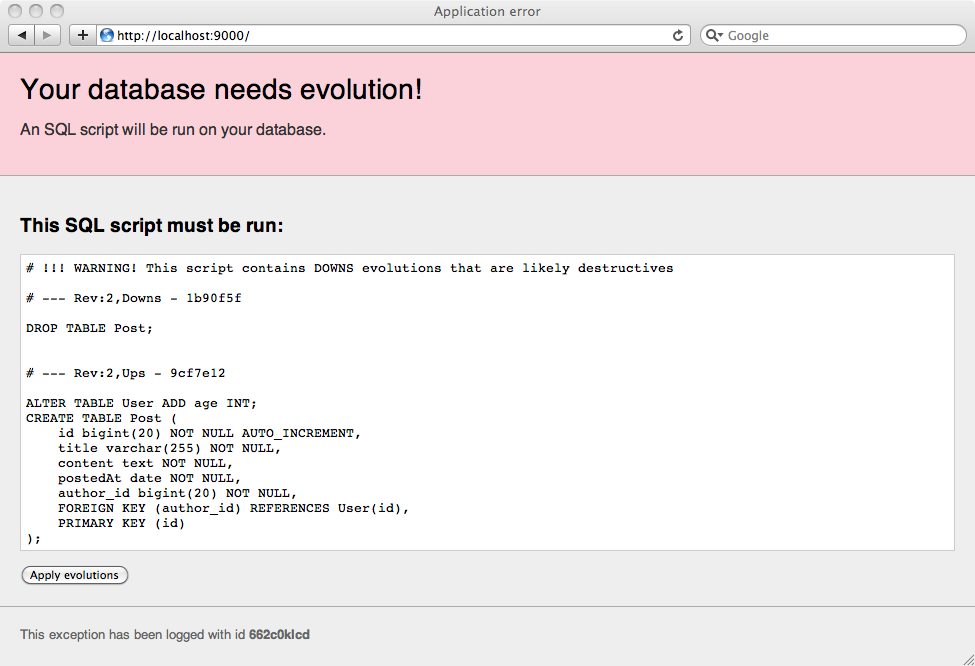
# --- !Downs

ALTER TABLE User DROP age;

DROP TABLE Post;

这个evolution脚本显示为新修订**revision 2** 的数据库，这个版本不同于A开发者之前已经应用的修订2版本。

因此Play会察觉到这个情况，并要求A开发者通过首先恢复已经应用的旧的修订2版本来同步他的数据库，并应用新的修订2版本脚本：



**数据不一致状态**

某些时候可能会在evolution脚本里犯错，那么这些脚本就会失效。在这种情况下，play将标记数据库结构为数据不一致状态，同时要求你在继续工作前手工解决这个问题。

比如，Evolution脚本的 **Ups** 存在一个错误：

# Add another column to User

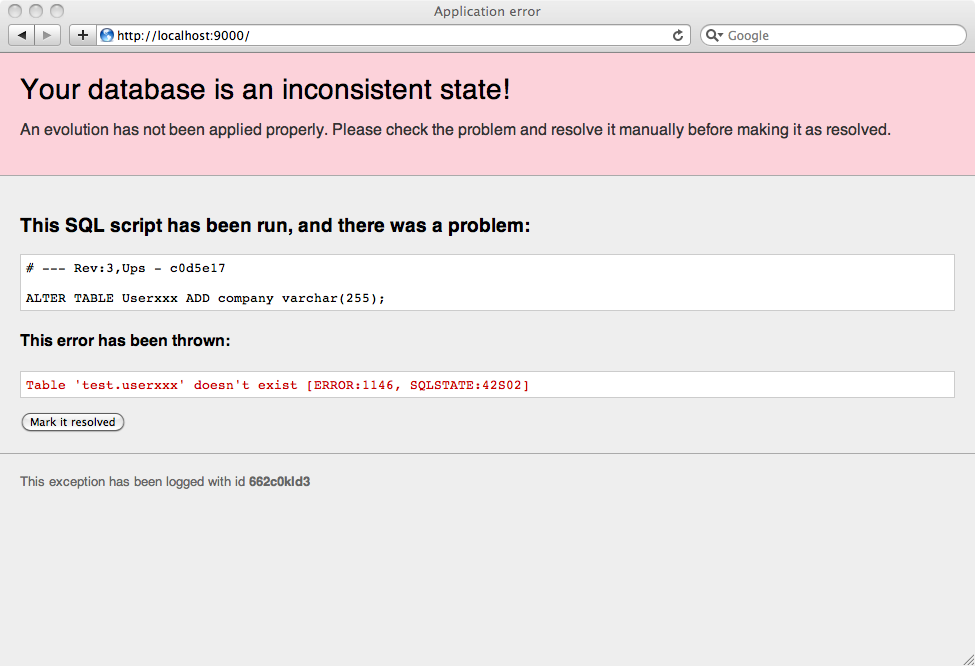
# --- !Ups

ALTER TABLE Userxxx ADD company varchar(255);

# --- !Downs

ALTER TABLE User DROP company;

因此，试着应用这个evolution将导致失败，play将把数据库结构标记为inconsistent:



现在要继续工作前，你必须解决这个不一致的问题，因此你就运行了fixed SQL 命令:

ALTER TABLE User ADD company varchar(255);

… 之后，通过单击按钮，手工解决了这个标记的问题。

但是由于你的evolution脚本存在错误，你或许希望修复这个错误。因此你修改了3.sql脚本：

# Add another column to User

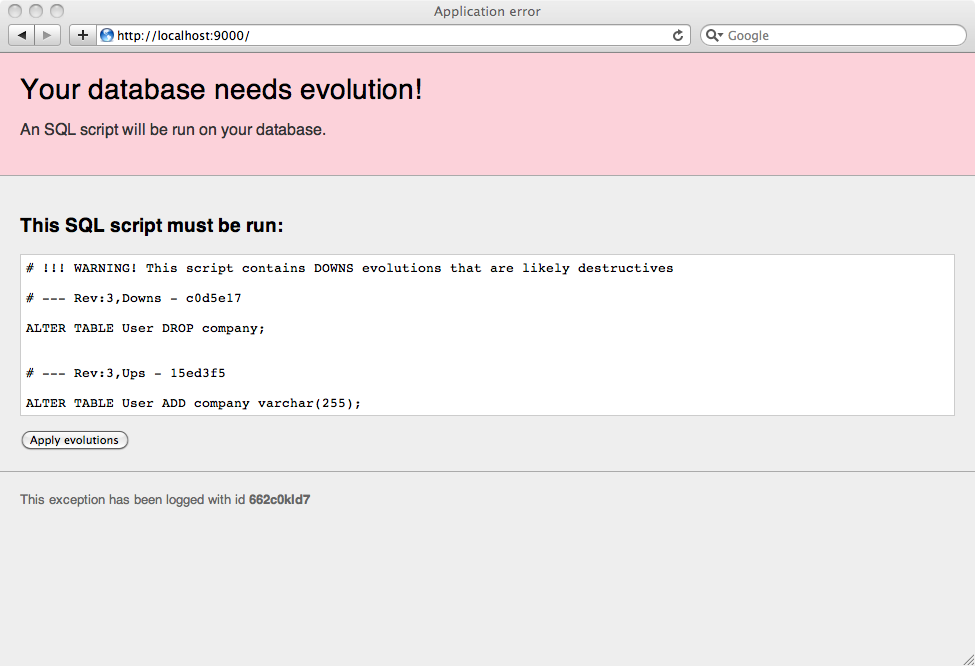
# --- !Ups

ALTER TABLE User ADD company varchar(255);

# --- !Downs

ALTER TABLE User DROP company;

Play检测到这个新的evolution，然后替换了之前的**3版，** 之后将运行下面的脚本：



现在所有的问题都已处理好，你又可以继续工作了。

在开发模式里，处理这样的问题非常简单，可以直接丢弃现在的开发数据库，然后再次重头申请所有的evolutions脚本即可。

**Evolutions 命令**

在DEV模式里，evolutions是交互式执行的。然而在PROD模式里，在运行应用程序前，你就必须使用evolutions命令来修订你的数据库结构。

在生产模式下如果你试着运行一个数据库不是最新版的应用程序时，应用程序将不能启动。

~ \_ \_

~ \_ \_\_ | | \_\_ \_ \_ \_| |

~ | '\_ \| |/ \_' | || |\_|

~ | \_\_/|\_|\\_\_\_\_|\\_\_ (\_)

~ |\_| |\_\_/

~

~ play! master-localbuild, http://www.playframework.org

~ framework ID is prod

~

~ Ctrl+C to stop

~

13:33:22 INFO ~ Starting ~/test

13:33:22 INFO ~ Precompiling ...

13:33:24 INFO ~ Connected to jdbc:mysql://localhost

13:33:24 WARN ~

13:33:24 WARN ~ Your database is not up to date.

13:33:24 WARN ~ Use `play evolutions` command to manage database evolutions.

13:33:24 ERROR ~

@662c6n234

Can't start in PROD mode with errors

Your database needs evolution!

An SQL script will be run on your database.

play.db.Evolutions$InvalidDatabaseRevision

at play.db.Evolutions.checkEvolutionsState(Evolutions.java:323)

at play.db.Evolutions.onApplicationStart(Evolutions.java:197)

at play.Play.start(Play.java:452)

at play.Play.init(Play.java:298)

at play.server.Server.main(Server.java:141)

Exception in thread "main" play.db.Evolutions$InvalidDatabaseRevision

at play.db.Evolutions.checkEvolutionsState(Evolutions.java:323)

at play.db.Evolutions.onApplicationStart(Evolutions.java:197)

at play.Play.start(Play.java:452)

at play.Play.init(Play.java:298)

at play.server.Server.main(Server.java:141)

错误消息要求你运行play evolutions命令：

$ play evolutions

~ \_ \_

~ \_ \_\_ | | \_\_ \_ \_ \_| |

~ | '\_ \| |/ \_' | || |\_|

~ | \_\_/|\_|\\_\_\_\_|\\_\_ (\_)

~ |\_| |\_\_/

~

~ play! master-localbuild, http://www.playframework.org

~ framework ID is gbo

~

~ Connected to jdbc:mysql://localhost

~ Application revision is 3 [15ed3f5] and Database revision is 0 [da39a3e]

~

~ Your database needs evolutions!

# ----------------------------------------------------------------------------

# --- Rev:1,Ups - 6b21167

CREATE TABLE User (

id bigint(20) NOT NULL AUTO\_INCREMENT,

email varchar(255) NOT NULL,

password varchar(255) NOT NULL,

fullname varchar(255) NOT NULL,

isAdmin boolean NOT NULL,

PRIMARY KEY (id)

);

# --- Rev:2,Ups - 9cf7e12

ALTER TABLE User ADD age INT;

CREATE TABLE Post (

id bigint(20) NOT NULL AUTO\_INCREMENT,

title varchar(255) NOT NULL,

content text NOT NULL,

postedAt date NOT NULL,

author\_id bigint(20) NOT NULL,

FOREIGN KEY (author\_id) REFERENCES User(id),

PRIMARY KEY (id)

);

# --- Rev:3,Ups - 15ed3f5

ALTER TABLE User ADD company varchar(255);

# ----------------------------------------------------------------------------

~ Run `play evolutions:apply` to automatically apply this script to the db

~ or apply it yourself and mark it done using `play evolutions:markApplied`

~

如果你打算让Play自动为你运行evolution，那么要使用如下命令：

$ play evolutions:apply

如果你喜欢在你的生产数据库上手工运行脚本，那么你需要告诉play你的数据库是最新的：

$ play evolutions:markApplied

如果在自动运行evolution脚本时存在错误，在DEV模式下，可以手工解决他们，之后标记数据库结构为已修正状态：

$ play evolutions:resolve

**20.日志配置**

play日志是基于[Log4j](http://logging.apache.org/log4j/1.2/index.html)的。既然许多java库都是使用Log4j或封装使用Log4j 的，那么就可以很容易在应用程序里进行日志配置。

**对应用程序进行日志**

Play使用play.Logger类提供了默认的日志功能，这个类使用Log4j来输出消息和异常到一个名叫play的日志文件。

对应用程序进行日志非常容易：

Logger.info("A log message");

Logger.error(ex, "Oops");

play.Logger类的方法支持使用java标准格式化语法进行格式化：

Logger.debug("The param was %s", param);

Logger.info("I want to log %s and %s and %s", a, b, c);

特定情况下，你仍旧可以使用Log4j直接创建日志器：

org.apache.log4j.Logger.getLogger("another.logger");

**配置日志级别**

通过配置[application.log](http://www.playframework.org/documentation/1.2.4/configuration#application.log)可以设置play的日志级别。修改值后不需要重新启动服务器即可立即应用。注意，这个级别只能显示应用程序的消息。

如果需要对Log4j进行完整配置，需要在conf目录下创建一个log4j.properties文件，因为这个目录是类路径，因此，该目录下的所有文件将被默认用于所有库。

默认的Log4j配置如下：

log4j.rootLogger=ERROR, Console

log4j.logger.play=INFO

# Console

log4j.appender.Console=org.apache.log4j.ConsoleAppender

log4j.appender.Console.layout=org.apache.log4j.PatternLayout

log4j.appender.Console.layout.ConversionPattern=%d{ABSOLUTE} %-5p ~ %m%n

复制并更新这个文件，如果有特别需要的话！

**生产配置**

详见： [configure logging for production](http://www.playframework.org/documentation/1.2.4/production#logging)

**21.管理多环境下的application.conf**

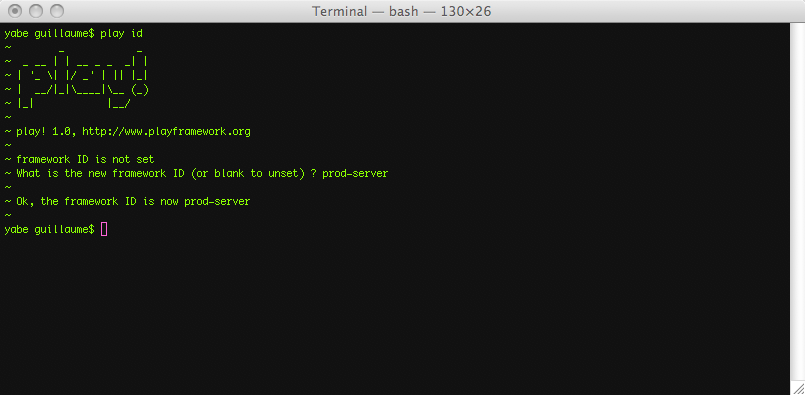
当进行团队合作开发时，不同开发者可能会在application.conf里使用不同的配置值，比如日志级别或数据库配置，当提交到VCS时就会导致冲突发生。

另外，不同的部署环境，需要不同的配置，比如：dev, test, staging 和 production。

**框架id(framework ID)**

为了解决这个问题，play允许你得到每个已经安装play框架的id，使用如下命令可以定义框架ID:

play id



然后就可以在配置keys上加上带有框架id的前缀：

application.name=Cool app

application.mode=dev

application.log=INFO

# Configuration for gbo

%gbo.application.log=DEBUG

%gbo.db=mem

# Configuration for src

%scr.http.port=9500

# Production configuration

%prod.http.port=80

%prod.application.log=INFO

%prod.application.mode=prod

**从命令行设置框架id**

通过另外一个命令可以直接给框架指定一个id，比如下面这个用于生产模式的应用程序：

play run --%production

这里，在application.conf文件里增加以下两行：

application.mode=dev

%production.application.mode=prod

这个命令兼容所有使用框架id的命令，默认的id定义仍旧使用play id命令。

以此类推， play test等价于：

play run --%test

**22.生产部署**

为了优化生产环境的应用程序，这里有一些小建议。

**application.conf**

首先，最好的方式就是给你的应用程序框架指定的一个id。让我们以production为例。参考 [manage application.conf in several environments](http://www.playframework.org/documentation/1.2.4/ids)节以了解如何实现。

**设置框架为prod模式：**

%production.application.mode=prod

在这种模式下，框架将预编译所有的java源代码和模板。如果在这步发生错误，应用程序将不能启动。对源代码的修改将不能进行热加载。

**定义一个真实的数据库：**

如果你使用的是开发数据库（如db=mem 或 db=fs),你应用配置真正可长期使用的数据库，如mysql:

%production.db.url=jdbc:mysql://localhost/prod

%production.db.driver=com.mysql.jdbc.Driver

%production.db.user=root

%production.db.pass=1515312

**禁止JPA的自动结构更新：**

如果使用了hibernate提供的数据库结构自动更新特性，那么你应该在生产模式下禁止该特性。

对生产服务器来说，让hibernate自动修改生产环境下的数据库结构和数据是个坏主意！

进行初始化部署是另一个潜在的问题。在这种情况下只需要进行如下定义：

%production.jpa.ddl=create

**定义一个安全的secret key:**

Play secret key将用于安全密码功能，比如session签名。你的应用程序必须保证这个key十分安全可靠。

%production.application.secret=c12d1c59af499d20f4955d07255ed8ea333

可以使用play secret命令来生成一个新的安全的随机key (至少在某个OS上是安全的)。如果计划把应用程序发布到多个服务器上，请记住一定要在所有的应用程序实例上使用同一个key！

**日志配置**

在生产模式下，使用循环日志文件是个好主意。一定不要发送日志信息到控制台，这是因为这此日志信息将会写入logs/system.out文件，而且大小不受限制！

在conf/目录创建一个定制的log4j.properties文件：

log4j.rootLogger=ERROR, Rolling

log4j.logger.play=INFO

# Rolling files

log4j.appender.Rolling=org.apache.log4j.RollingFileAppender

log4j.appender.Rolling.File=application.log

log4j.appender.Rolling.MaxFileSize=1MB

log4j.appender.Rolling.MaxBackupIndex=100

log4j.appender.Rolling.layout=org.apache.log4j.PatternLayout

log4j.appender.Rolling.layout.ConversionPattern=%d{ABSOLUTE} %-5p ~ %m%n

**前端http服务器（Front-end HTTP server）**

把应用程序作为独立服务器部署到80端口非常容易：

%production.http.port=80

但是，如果你计划在同一台服务器部署多个应用程序，或者为应用程序的多个实例实现负载平衡（以实现可伸缩性部署或故障容错），可以使用前端http服务器实现这个功能。

**注意，如果使用前端服务器和和直接使用play服务器相比，前者将不能提供更好的展现效果！**

**部署到lighttpd服务器的设置**

本示例将向你展现如何配置[lighttpd](http://www.lighttpd.net/)作为前端web服务器。注意，本配置同样适用于Apache，但如果你只需要虚拟化或负载平衡功能， lighttpd是最好的选择，而且非常容易配置！

/etc/lighttpd/lighttpd.conf文件的配置示例如下（linux）:

server.modules = (

"mod\_access",

"mod\_proxy",

"mod\_accesslog"

)

…

$HTTP["host"] =~ "www.myapp.com" {

proxy.balance = "round-robin" proxy.server = ( "/" =>

( ( "host" => "127.0.0.1", "port" => 9000 ) ) )

}

$HTTP["host"] =~ "www.loadbalancedapp.com" {

proxy.balance = "round-robin" proxy.server = ( "/" => (

( "host" => "127.0.0.1", "port" => 9000 ),

( "host" => "127.0.0.1", "port" => 9001 ) )

)

}

**部署到****Apache服务器的设置**

下面的示例显示了如何在Apache httpd server下 进行配置：

LoadModule proxy\_module modules/mod\_proxy.so

…

<VirtualHost \*:80>

ProxyPreserveHost On

ServerName www.loadbalancedapp.com

ProxyPass / http://127.0.0.1:9000/

ProxyPassReverse / http://127.0.0.1:9000/

</VirtualHost>

**Apache作为前端代理服务器，可以允许透明更新你的应用程序**

一个基本的例子，就是让前端代理服务器负载平衡两个play应用程序，如果其中一个失效，那么另外一个将自动承担全部任务。

让我们在9999和9998两个端口分别启动相同的应用程序。

把应用程序复制一件，修改其application.conf文件中的端口号。

在每个应用程序目录下分别运行如下命令：

play start mysuperwebapp

现在，让我们对Apache服务器进行配置，让其实现负载平衡。

在Apache里，我们使用如下配置：

<VirtualHost mysuperwebapp.com:80>

ServerName mysuperwebapp.com

<Location /balancer-manager>

SetHandler balancer-manager

Order Deny,Allow

Deny from all

Allow from .mysuperwebapp.com

</Location>

<Proxy balancer://mycluster>

BalancerMember http://localhost:9999

BalancerMember http://localhost:9998 status=+H

</Proxy>

<Proxy \*>

Order Allow,Deny

Allow From All

</Proxy>

ProxyPreserveHost On

ProxyPass /balancer-manager !

ProxyPass / balancer://mycluster/

ProxyPassReverse / http://localhost:9999/

ProxyPassReverse / http://localhost:9998/

</VirtualHost>

其中最重要的部分为balancer://mycluster，这里声明加载负载平衡。+H选项的意思是让第二个Play应用程序处于待命状态。当然，这里也可指示第二个play应用程序为load-balance负载平衡状态。

当需要更新mysuperwebapp时，需要使用如下命令：

play stop mysuperwebapp1

这里，所有的工作都会被load-balancer转发到mysuperwebapp2,与此同时， mysuperwebapp1进行更新操作，一定操作完成，就可执行以下命令：

play start mysuperwebapp1

这时，你就可以安全更新mysuperwebapp2了。

Apache同时提供了显示服务器集群cluster的方法。方法是在浏览器里访问应用程序的/balancer-manager地址，就可以查看服务器集群的当前状态。

因为Play是完全无状态的，因此，你不需要在2个集群(服务器)间管理session，事实上，你可以很容易扩展到2个以上的play实例。

**高级代理设置**

当使用http前端服务器，需要一个地址作为http服务器的统一对外地址，用于接收所有的请求。通常情况下，同一台服务器即作为play应用服务器，又作为代理服务器，这时，play将把所有的请求看成是来自127.0.0.1的请求。

Proxy服务器可以为请求添加一个特定的header来告诉被代理的应用程序，这个请求来自哪里。大多数web服务器都会远程客户端ip地址添加一个X-Forwarded-For header作为第一个参数。如果你在XForwardedSupport配置里允许forward转发支持, Play会从代理的ip至客户端的ip更改request.remoteAddress，为了能够正常工作，你就必须列出你的代理服务器的ip地址。

然而，这样并不涉及主机头，它将仍旧被代理服务器发布。如果你使用的是Apache 2.x, 你可添加如下指令：

ProxyPreserveHost on

host: header仍旧是客户端核发的原始host请求header。通过这两种技术的组合，你的应用程序看起来好像是直接暴露出来的单一主机。

**HTTPS配置**

内建服务器支持HTTPS协议，你可以在生产环境中直接使用。它支持证书管理，即可以是传统的java keystore，也可以是简单的cert key文件。为了为你的应用程序启动HTTPS连接，只需要在applictaion.conf里声明https.port 配置属性就行。

http.port=9000

https.port=9443

这里需要把你的证书放到conf目录。Play支持X509证书和keystore证书。X509 证书必须采用如下命名方式：  
**host.cert** 作为证书命名，**host.key**作为key命名。如果使用的是keystore，那么默认命令应该为**certificate.jks**。

如果使用的是X509证书，那么可以在application.conf里配置如下参数：

# X509 certificates

certificate.key.file=conf/host.key

certificate.file=conf/host.cert

# In case your key file is password protected

certificate.password=secret

trustmanager.algorithm=JKS

使用keystore时，配置如下:

keystore.algorithm=JKS

keystore.password=secret

keystore.file=conf/certificate.jks

注意，上面采用的是默认值。

你可以使用**openssl**生成自签名self-signed证书：

openssl genrsa 1024 > host.key

openssl req -new -x509 -nodes -sha1 -days 365 -key host.key > host.cert

如果使用的是Java keystore机制，那么可以在application.conf里配置如下属性：

# Keystore

ssl.KeyManagerFactory.algorithm=SunX509

trustmanager.algorithm=JKS

keystore.password=secret

keystore.file=certificate.jks

上面的值采用的是默认值。

**不依赖Python进行部署**

默认情况下，大多数Unix机器都安装了Python，在windows下，play已经自动植入了Python。但可能存在你需要部署的服务器并没有安装Python。

为解决这个问题，play应用程序里的build.xml文件提供了有限的功能。

在应用程序目录下，使用以下命令启动服务器:

ant start -Dplay.path=/path/to/playdirectory

警告: 使用play命令时，输出信息将直接转发到System.out，使用Ant时，标准输出将不可访问，因此，提供Log4j属性文件是必要的。

停止服务器时，使用如下命令:

ant stop -Dplay.path=/path/to/playdirectory

注意，你可在系统环境变量里为设置play框架路径，也可在应用程序的build.xml文件里设定。

**23.部署选择**

Play应用程序实际上可以部署到任何环境下：如部署到Servlet容器里、作为独立服务器、或部署到Heroku, Google Application Engine, Stack, a Cloud, 等等

**独立Play应用程序**

最简单、最稳定的方式就是不需要任何容器简单运行play应用程序。如果需要更高级的特性，比如虚拟主机，你可使用前端http服务器，比如Lighttpd或Apache。

内存的HTTP服务可以支持每秒上千的HTTP请求，而且不存在瓶颈。此外，它还使用了效率更高的线程模型（Servlet容器采取的是每个请求分配一个线程），而且允许不同的模块使用不的服务器 (Grizzly, Netty,等等)

这些服务器都支持在不中断正在执行线程的情况下支持long polling（长轮询），允许管理超长请求（等待一个长时间任务完成），允许把文件对象直接转换成流(包括任何指定了Content-Length内容长度的InputStream)。

当使用同开发时相同的环境运行应用程序时会存在几个小问题。当部署到jee应用程序服务器时，会产生许多小bug（不同的程序主目录, 类加载问题, 库冲突等等)。

参考['Put your application in production' page](http://www.playframework.org/documentation/1.2.4/production) 。

**Java EE应用服务器**

Play应用程序也可运行于多个流行的应用服务器。大多数应用服务器都支持out of the box方式。

**支持应用服务器**

下面的应用服务器已测试可正常运行play应用：

* JBoss 4.2.x
* JBoss 5.x
* JBoss 6M2
* Glassfish v3
* IBM Websphere 6.1
* IBM Websphere 7
* Geronimo 2.x
* Tomcat 6.x
* Jetty 7.x
* Resin 4.0.5

**部署**

想要部署到上述服务器，你需要把应用程序打包成一个war文件，命令为：

play war myapp -o myapp.war

请注意你的应用服务器必须支持war文件部署。

在打包的时候，play会让你‘独立(isolate)’你的play应用程序，不要与其他应用程序混合部署，以避免应用程序库之间的版本匹配问题。JEE/Servlet并没有对此进行规范，属地方规范（vendor-specific）。

为了‘独立(isolate)’你的WAR文件，我们推荐你参考你的应用服务器手册。下面演示了如果在JBoss应用服务器里独立一个war文件。注意，下面这些步骤是可选的：

在你的应用程序war目录下的myapp.war/WEB-INF/jboss-web.xml文件里插入如下内容：

<jboss-web>

<class-loading java2classloadingcompliance="false">

<loader-repository>

com.example:archive=unique-archive-name

<loader-repository-config>java2ParentDelegation=false</loader-repository-config>

</loader-repository>

</class-loading>

</jboss-web>

用你想要的唯一值替换com.example:archive=unique-archive-name。

**数据源**

Play也支持数据源和资源查找。为了使用JNDI数据源，需要设置[database configuration](http://www.playframework.org/documentation/1.2.4/configuration#dbconf)：

db=java:comp/env/jdbc/mydb

jpa.dialect=org.hibernate.dialect.Oracle10gDialect

jpa.ddl=verify

数据库插件一旦检测到db=java:，就会冻结默认的JDBC系统。

**定制web.xml**

在某些应用服务器里，比如IBM Websphere,需要你在web.xml里的resource-ref元素里声明数据源。默认情况下，在执行play war命令时，play会生成web.xml文件。为了定制生成的web.xml,首先要生成WAR未打包版本，然后复制web.xml文件到war/WEB-INF文件夹下。最后再次执行play war命令，命令将会从生成文件夹复制你定制的web.xml。

比如，为了给IBM Websphere 7声明一个数据源，我们可以在我们的war/WEB-INF/web.xml里声明一个resource-ref：

<?xml version="1.0" ?>

<web-app xmlns="http://java.sun.com/xml/ns/j2ee"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee http://java.sun.com/xml/ns/j2ee/web-app\_2\_4.xsd"

version="2.4">

<display-name>Play! (%APPLICATION\_NAME%)</display-name>

<context-param>

<param-name>play.id</param-name>

<param-value>%PLAY\_ID%</param-value>

</context-param>

<listener>

<listener-class>play.server.ServletWrapper</listener-class>

</listener>

<servlet>

<servlet-name>play</servlet-name>

<servlet-class>play.server.ServletWrapper</servlet-class>

</servlet>

<servlet-mapping>

<servlet-name>play</servlet-name>

<url-pattern>/</url-pattern>

</servlet-mapping>

<resource-ref>

<description>Play Datasource for testDatasource</description>

<res-ref-name>jdbc/mydb</res-ref-name>

<res-type>javax.sql.DataSource</res-type>

<res-auth>Container</res-auth>

</resource-ref>

</web-app>

**基于云的主机Cloud-based hosting**

**AWS Elastic Beanstalk**

[AWS Elastic Beanstalk](http://aws.amazon.com/en/elasticbeanstalk/) is Amazon’s(亚马逊)的java主机平台，基于亚马逊Web Services基础设施。更多信息详见： [Java development 2.0: Play-ing with Amazon RDS](http://www.ibm.com/developerworks/java/library/j-javadev2-19/).

**CloudBees**

[CloudBees](http://www.cloudbees.com/) 也是一个java主机平台，详见[CloudBees module](http://www.playframework.org/modules/cloudbees)。

**Cloud Foundry**

[Cloud Foundry](http://www.cloudfoundry.com/) VMware的云提供者。详见[Running Play Framework Application on CloudFoundry](http://iambivas.blogspot.com/2011/08/running-play-framework-application-on.html) and the [CloudFoundry module](http://www.playframework.org/modules/cloudfoundry).

**Google App Engine (GAE)**

[Google App Engine](http://code.google.com/appengine/) 是Google提供的较好的云主机平台，它和通用的主机平台相比，具有不同的pros 和 cons。特别要注意的是，GAE不支持JPA持久化，详见[GAE module](http://www.playframework.org/modules/gae)。

**Heroku**

[Heroku cloud application platform](http://www.heroku.com/) 是一个特别为play提供支持的主机平台。通过以下步骤进行部署：

1. 在[Linux](http://toolbelt.herokuapp.com/linux/readme), [Mac](http://toolbelt.herokuapp.com/osx/download), or [Windows](http://toolbelt.herokuapp.com/windows/download)下安装Heroku命令行客户端  
2. 安装[git](http://git-scm.com/) 并设置你的ssh key  
3.在 [Heroku.com](http://heroku.com/signup)上创建一个帐号  
4. 从命令行登录到Heroku

heroku auth:login

5. 创建一个git仓库：

git init

6. 创建一个.gitignore 文件，配置如下，以忽略play生成的文件：

/tmp

/modules

/lib

/test-result

/logs

7. 增加文件到git仓库并提交：

git add .

git commit -m init

8.在Heroku上创建一个新的应用程序:

heroku create -s cedar

9. 发送应用程序到Heroku:

git push heroku master

10. 在你的浏览器里打开应用程序：

heroku open

以下命令用于浏览运行日志：

heroku logs

为了scale（缩放）应用，可以多个‘dynos’方式运行:

heroku scale web=2

为了在产品里使用 [Heroku Shared Database](http://devcenter.heroku.com/articles/database) ，需要在conf/application.conf里增加以下配置：

%prod.db=${DATABASE\_URL}

%prod.jpa.dialect=org.hibernate.dialect.PostgreSQLDialect

%prod.jpa.ddl=update

更多信息见[Heroku Dev Center](http://devcenter.heroku.com)。

**playapps.net**

[playapps.net](http://www.playapps.net/) 是Zenexity提供的play特定的主机平台，其创建者是play。详见[playapps.net module](http://www.playframework.org/modules/playapps)。