本章主要侧重于Grails是如何集成和构建在spring基础之上的。学习本章有助于深刻理解Grails也有助于更好的开发插件。

# Grails的基础

Grails应用在表象之下其实就是一个Spring MVC应用。Spring MVC属于Spring框架的一部分。Spring MVC简单易用非常适合Grails以它为基础构建自己的MVC框架。

Grails在下面几个方面借助了Spring MVC的强大特性：

* Basic controller logic - Grails继承了spring的 [DispatcherServlet](http://static.springsource.org/spring/docs/3.0.x/javadoc-api/org/springframework/web/servlet/DispatcherServlet.html) 并使用它把请求分发到各个 Grails [controllers](http://grails.org/doc/latest/guide/single.html#controllers)上。
* Data Binding and Validation - Grails的 [validation](http://grails.org/doc/latest/guide/single.html#validation) 和 [data binding](http://grails.org/doc/latest/guide/single.html#dataBinding) 本质上使用了Spring提供的实现。
* Runtime configuration - Grails的整个运行时预定都是建立在Spring的 [ApplicationContext](http://static.springsource.org/spring/docs/3.0.x/javadoc-api/org/springframework/context/ApplicationContext.html)之上。
* Transactions - Grails 的 [GORM](http://grails.org/doc/latest/guide/single.html#GORM)使用Spring的事务管理机制。

## Grails Application Context

Spring的开发人员往往需要Grails Application context是如何构造的。可以用以下两点间的说明：

* Grails有一个基于web-app/WEB-INF/applicationContext.xml 文件构造的ApplicationContext对象。  ApplicationContext构造了 [GrailsApplication](http://grails.org/doc/latest/api/org/codehaus/groovy/grails/commons/GrailsApplication.html) 实例和 [GrailsPluginManager](http://grails.org/doc/latest/api/org/codehaus/groovy/grails/plugins/GrailsPluginManager.html)实例
* 使用 ApplicationContext 作为父类 Grails可以分析GrailsApplication instance的约定并 构造了一个子类的ApplicationContext 用于整个 web application的root ApplicationContext。

## 已配置Spring bean

大多说的Grails配置都是在运行时发生的。每一个插件也可以配置注册在自己的ApplicationContext之中的bean。

# 配置Additional bean

## 使用Spring Bean DSL

可以在grails-app/conf/spring/resources.groovy文件中注册新的bean。resources.groovy使用Grails的Spring DSL注册bean。bean都以closure的形式注册在beans配置项中：

|  |
| --- |
| beans = {  // beans here  } |

可以如下注册一个bean：

|  |
| --- |
| import my.company.MyBeanImpl  beans = {  myBean(MyBeanImpl) {  someProperty = 42  otherProperty = "blue"  }  } |

一旦配置好，这个bean就可以被自动注入到controller/service等grails的artifact和其他支持依赖注入的class中（例如BootStrap.groovy）。自动注入只需要申明一个同bean name相同的public field就行了。例如：

|  |
| --- |
| class ExampleController {  def myBean  …  } |

使用DSL的一个好处就是可以把bean的声明和逻辑混合在一起，例如可以基于运行环境声明bean：

|  |
| --- |
| import grails.util.Environment  import my.company.mock.MockImpl  import my.company.MyBeanImpl  beans = {  switch(Environment.current) {  case Environment.PRODUCTION:  myBean(MyBeanImpl) {  someProperty = 42  otherProperty = "blue"  }  break  case Environment.DEVELOPMENT:  myBean(MockImpl) {  someProperty = 42  otherProperty = "blue"  }  break  }  } |

在resources.groovy中可以通过变量名application获取GraislApplication对象的实例：

|  |
| --- |
| import grails.util.Environment  import my.company.mock.MockImpl  import my.company.MyBeanImpl  beans = {  if (application.config.my.company.mockService) {  myBean(MockImpl) {  someProperty = 42  otherProperty = "blue"  }  } else {  myBean(MyBeanImpl) {  someProperty = 42  otherProperty = "blue"  }  }  } |

## 使用XML

Grails也支持使用grails-app/conf/spring/resources.xml类注册bean。例如可以如下注册一个bean：

|  |
| --- |
| <bean id="myBean" class="my.company.MyBeanImpl">  <property name="someProperty" value="42" />  <property name="otherProperty" value="blue" />  </bean> |

## 引用已经注册的beans

在resources.groovy、resources.xml和java代码中可以很方便的引用其他已经注册到容器中的bean。例如已经有一个名为bookService的bean。

DSL：

|  |
| --- |
| beans = {  myBean(MyBeanImpl) {  someProperty = 42  otherProperty = "blue"  bookService = ref("bookService")  }  } |

XML：

|  |
| --- |
| <bean id="myBean" class="my.company.MyBeanImpl">  <property name="someProperty" value="42" />  <property name="otherProperty" value="blue" />  <property name="bookService" ref="bookService" />  </bean> |

java代码：

|  |
| --- |
| package my.company;  class MyBeanImpl {  private BookService bookService;  private Integer someProperty;  private String otherProperty;  public void setBookService(BookService theBookService) {  this.bookService = theBookService;  }  public void setSomeProperty(Integer someProperty) {  this.someProperty = someProperty;  }  public void setOtherProperty(String otherProperty) {  this.otherProperty = otherProperty;  }  } |

# 使用Beans DSL让Spring支持运行时更改和新增bean依赖配置的能力

常规的spring配置是静态的，很难在运行的时候去更改配置。Grails提供了BeanBuilder可以通过编码在运行时根据系统属性或者环境变量动态的将bean配置到一起。

## The BeanBuilder class

Grails提供了grails.spring.BeanBuilder类使用Groovy动态语言的特性来配置spring bean。基本的使用示例如下：

|  |
| --- |
| import org.apache.commons.dbcp.BasicDataSource  import org.codehaus.groovy.grails.orm.hibernate.ConfigurableLocalSessionFactoryBean  import org.springframework.context.ApplicationContext  import grails.spring.BeanBuilder  def bb = new BeanBuilder()  bb.beans {  dataSource(BasicDataSource) {  driverClassName = "org.h2.Driver"  url = "jdbc:h2:mem:grailsDB"  username = "sa"  password = ""  }  sessionFactory(ConfigurableLocalSessionFactoryBean) {  dataSource = ref('dataSource')  hibernateProperties = ["hibernate.hbm2ddl.auto": "create-drop",  "hibernate.show\_sql": "true"]  }  }  ApplicationContext appContext = bb.createApplicationContext() |

上面的例子表示如何使用BeanBuilder给hibernate配置一个datasource

在插件和grails-app/conf/spring/resources.groovy文件中并不需要新建一个BeanBuilder的实例，grails的doWithSpring和beans块隐式的使用BeanBuilder。

bean的属性也可以通过BeanBuilder来设置：

|  |
| --- |
| sessionFactory(ConfigurableLocalSessionFactoryBean) { bean ->  // Autowiring behaviour. The other option is 'byType'. [autowire]  bean.autowire = 'byName'  // Sets the initialisation method to 'init'. [init-method]  bean.initMethod = 'init'  // Sets the destruction method to 'destroy'. [destroy-method]  bean.destroyMethod = 'destroy'  // Sets the scope of the bean. [scope]  bean.scope = 'request'  dataSource = ref('dataSource')  hibernateProperties = ["hibernate.hbm2ddl.auto": "create-drop",  "hibernate.show\_sql": "true"]  } |

## 在Spring MVC中使用BeanBuilder

在普通的Spring MVC项目中也可以使用BeanBuilder。

首先将grails-spring-<version>.jar引入到项目的classpath中。

接着，在项目的web.xml中设置两个context-param：

|  |
| --- |
| <context-param>  <param-name>contextConfigLocation</param-name>  <param-value>/WEB-INF/applicationContext.groovy</param-value>  </context-param>  <context-param>  <param-name>contextClass</param-name>  <param-value>  org.codehaus.groovy.grails.commons.spring.GrailsWebApplicationContext  </param-value>  </context-param> |

最后就是在/WEB-INF/目录下新建一个bean配置文件applicationContext.groovy，将bean的配置写在这个文件中。

|  |
| --- |
| import org.apache.commons.dbcp.BasicDataSource  beans {  dataSource(BasicDataSource) {  driverClassName = "org.h2.Driver"  url = "jdbc:h2:mem:grailsDB"  username = "sa"  password = ""  }  } |

## 从文件系统载入bean的配置信息

可以使用BeanBuilde直接从文件系统中载入配置信息：

|  |
| --- |
| def bb = new BeanBuilder()  bb.loadBeans("classpath:\*SpringBeans.groovy")  def applicationContext = bb.createApplicationContext() |

上面的例子表示BeanBuilder从classpath中将所有由SpringBeans.groovy结尾的配置文件中的bean信息载入到spring容器中。配置文件的内容可以是如下脚本：

|  |
| --- |
| import org.apache.commons.dbcp.BasicDataSource  import org.codehaus.groovy.grails.orm.hibernate.ConfigurableLocalSessionFactoryBean  beans {  dataSource(BasicDataSource) {  driverClassName = "org.h2.Driver"  url = "jdbc:h2:mem:grailsDB"  username = "sa"  password = ""  }  sessionFactory(ConfigurableLocalSessionFactoryBean) {  dataSource = dataSource  hibernateProperties = ["hibernate.hbm2ddl.auto": "create-drop",  "hibernate.show\_sql": "true"]  }  } |

## 使用Binding给配置文件传递参数

使用Grails的Binding类可以给配置文件传入参数，例如：

|  |
| --- |
| def binding = new Binding()  binding.maxSize = 10000  binding.productGroup = 'finance'  def bb = new BeanBuilder()  bb.binding = binding  bb.loadBeans("classpath:\*SpringBeans.groovy")  def ctx = bb.createApplicationContext() |

这样在\*SpringBeans.groovy中就可以直接使用maxSize和productGroup这两个变量了。

# Grails BeanBuilder DSL

## 设置构造函数参数

bean的构造函数的参数可以作为参数放在bean-defining方法的参数列表中（放在第一个参数的后面）：

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

上面的例子数码MyExampleBean有两个构造函数参数：

|  |
| --- |
| MyExampleBean(String foo, int bar) {  …  } |

## 配置BeanDefinition（使用工厂方法）

配置块closure的第一个参数就是bean实例本身，这样就可以给bean配置工厂方法。

|  |
| --- |
| bb.beans {  exampleBean(MyExampleBean) { bean ->  bean.factoryMethod = "getInstance"  bean.singleton = false  someProperty = [1, 2, 3]  }  } |

## 配置工厂类的bean

由工厂类提供的bean配置块没有Class参数，传入的参数就是用于产生bean实例的工厂bean。

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

另外，还可以提供工厂bean中用于创建bean实例的方法名：

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

上面的例子中getInstance方法用于创建myBean实例。

## 在运行时创建bean实例

有时，在配置的时候并不知道需要创建的bean的具体名字，可能只有在运行时在某个条件下才知道创建的bean具体该用什么名字。这种情况下BeanBuilder提供了使用占位符的能力：

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

上面的beanName可以是任何运行时变量或参数。

同样，适用占位符的bean也可以被别的bean引用：

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

## 使用匿名内部bean

使用closure可以创建一个bean的匿名内部bean，closue用匿名内部bean的类型作为参数：

|  |
| --- |
| bb.beans {  marge(Person) {  name = "Marge"  husband = { Person p ->  name = "Homer"  age = 45  props = [overweight: true, height: "1.8m"]  }  children = [bart, lisa]  }  bart(Person) {  name = "Bart"  age = 11  }  lisa(Person) {  name = "Lisa"  age = 9  }  } |

匿名内部bean的配置也可以使用工厂bean：

|  |
| --- |
| bb.beans {  personFactory(PersonFactory)  marge(Person) {  name = "Marge"  husband = { bean ->  bean.factoryBean = "personFactory"  bean.factoryMethod = "newInstance"  name = "Homer"  age = 45  props = [overweight: true, height: "1.8m"]  }  children = [bart, lisa]  }  } |

## 抽象bean和父类bean的配置

抽象bean定义不需要Class参数。假如，有如下两个Class：

|  |
| --- |
| class HolyGrailQuest {  def start() { println "lets begin" }  } |
| class KnightOfTheRoundTable {  String name  String leader  HolyGrailQuest quest  KnightOfTheRoundTable(String name) {  this.name = name  }  def embarkOnQuest() {  quest.start()  }  } |

定义一个抽象bean：

|  |
| --- |
| import grails.spring.BeanBuilder  def bb = new BeanBuilder()  bb.beans {  abstractBean {  leader = "Lancelot"  }  …  } |

抽象bean配置了一个leader属性，抽象bean的定义就可以被其他bean的定义使用parent属性继承了。

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

注意当bean.parent需要放在其它bean属性设置的前面。

## 使用Spring的Namespaces

可以在BeanBuilder中可以给bean设置namespace：

|  |
| --- |
| xmlns context:"http://www.springframework.org/schema/context" |

然后就可以像方法调用一样的使用Spring Namespaces提供的标签了，例如标签component-scan：

|  |
| --- |
| context.'component-scan'('base-package': "my.company.domain") |

获取JNDI资源：

|  |
| --- |
| xmlns jee:"http://www.springframework.org/schema/jee"  jee.'jndi-lookup'(id: "dataSource", 'jndi-name': "java:comp/env/myDataSource") |

使用Sping namespace可以在BeanBuilder中充分利用强大的Sping AOP特性。例如有如下两个class：

|  |
| --- |
| class Person {  int age  String name  void birthday() {  ++age;  }  } |
| class BirthdayCardSender {  List peopleSentCards = []  void onBirthday(Person person) {  peopleSentCards << person  }  } |

可以使用Spring AOP namespace拦截birthday方法的调用：

|  |
| --- |
| xmlns aop:"http://www.springframework.org/schema/aop"  fred(Person) {  name = "Fred"  age = 45  }  birthdayCardSenderAspect(BirthdayCardSender)  aop {  config("proxy-target-class": true) {  aspect(id: "sendBirthdayCard", ref: "birthdayCardSenderAspect") {  after method: "onBirthday",  pointcut: "execution(void ..Person.birthday()) and this(person)"  }  }  } |

## 属性占位符配置

在BeanBuilder中也可以使用属性占位符。例如，在Config.groovy中有如下两个属性：

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

那么在resources.xml中可以这样使用：

|  |
| --- |
| <bean id="dataSource"  class="org.springframework.jdbc.datasource.DriverManagerDataSource">  <property name="driverClassName">  <value>${database.driver}</value>  </property>  <property name="url">  <value>jdbc:${database.dbname}</value>  </property>  </bean> |

在resources.groovy中可以如下使用，注意占位符使用单引号：

|  |
| --- |
| dataSource(org.springframework.jdbc.datasource.DriverManagerDataSource) {  driverClassName = '${database.driver}'  url = 'jdbc:${database.dbname}'  } |

在resources.groovy中更好的方式是使用grailsApplication来获取配置信息：

|  |
| --- |
| dataSource(org.springframework.jdbc.datasource.DriverManagerDataSource) {  driverClassName = grailsApplication.config.database.driver  url = "jdbc:${grailsApplication.config.database.dbname}"  } |