插件是web开发框架的首要元素，同时插件也构成一个平台。插件平台能够让Grails框架拥有无限扩展的能力。

创建一个插件听起来好像很复杂，但是插件也分为很简单的插件和功能强大的复杂插件。只要知道如何创建一个Grails项目，也就能够创建用于共享数据模型和静态资源相关的插件了。

# 创建和安装插件

## 创建插件

用命令行的方式创建一个插件如下：

|  |
| --- |
| grails create-plugin [PLUGIN NAME] |

这个命令就会创建一个根据参数命名的插件项目。注意插件名不能有多个大些字符。

插件项目的目录结构基本同Grails常规项目的目录结构类似，唯一不同的地方是，在插件项目的根目录下有一个用于说明插件的groovy文件。

同样，测试插件也和普通项目一样运行run-app命令。

注意，插件项目没有index.gsp页面，如果需要可以自己在grails-app/views下面加一个。

用于描述插件的文件约定以GrailsPlugin结尾，如插件名为security，则该文件名为SecurityGrailsPlugin.groovy。

所有的插件项目都需要在项目的根目录下有一个插件描述文件。插件描述文件定义了插件的版本和其他元数据，还包括各个插件扩展点的hook。

通过下面的几个属性，还可以给插件添加额外信息。

* title - 插件的title
* version - 插件的版本. 例如 "0.1", "0.2-SNAPSHOT", "1.1.4" 等.
* grailsVersion - 可以运行该插件的版本. 例如. "1.2 > \*" (grails1.2或更高的版本)
* author - 插件的作者
* authorEmail - plugin author's contact e-mail
* description - 用于描述插件的详细信息
* documentation - 指向插件文档的URL

下面的例子是Quartz插件的描述文件内容：

|  |
| --- |
| class QuartzGrailsPlugin {  def version = "0.1"  def grailsVersion = "1.1 > \*"  def author = "Sergey Nebolsin"  def authorEmail = "nebolsin@gmail.com"  def title = "Quartz Plugin"  def description = '''\  The Quartz plugin allows your Grails application to schedule jobs\  to be executed using a specified interval or cron expression. The\  underlying system uses the Quartz Enterprise Job Scheduler configured\  via Spring, but is made simpler by the coding by convention paradigm.\  '''  def documentation = "http://grails.org/plugin/quartz"  …  } |

## 安装本地插件

为了确保新建的插件可用，首先确保执行了grails maven-install命令，该命令会将插件先安装到maven本地库中:

|  |
| --- |
| grails maven-install |

接下来就可以在常规项目的grails-app/conf/BuildConfig.groovy中安装插件了：

|  |
| --- |
| compile ":quartz:0.1" |

## 被排除的Artefacts

尽管插件项目和常规项目的目录结构基本类似，但是插件项目打包的时候并不会把所有的项目文件都打到包里头。下面的文件虽然是由create-plugin命令创建，但是不会被package-plugin命令打到插件包内。

* grails-app/conf/BootStrap.groovy
* grails-app/conf/BuildConfig.groovy (although it is used to generate dependencies.groovy)
* grails-app/conf/Config.groovy
* grails-app/conf/DataSource.groovy (and any other \*DataSource.groovy)
* grails-app/conf/UrlMappings.groovy
* grails-app/conf/spring/resources.groovy
* Everything within /web-app/WEB-INF
* Everything within /web-app/plugins/\*\*
* Everything within /test/\*\*
* SCM management files within \*\*/.svn/\*\* and \*\*/CVS/\*\*

除此之外，还可以在描述文件中增加pluginExcludes配置项来自定义还有哪些内容不需要打到插件包中

## 自定义插件内容

有两种方式可以自定义插件包中的内容。

一种是在scripts目录下增加一个\_Events.groovy文件并对CreatePluginArchiveStart事件做处理，该处理器在所有的插件文件都被复制到临时stagin目录之后被触发。例如：

|  |
| --- |
| eventCreatePluginArchiveStart = { stagingDir ->  // update staging directory contents here  } |

在BuildConfig.groovy中可以定义stagingDir：grails.project.plugin.staging.dir

另一种方式就是在项目的BuildConfig.groovy中给grails.plugin.resources一个closure。

|  |
| --- |
| grails.plugin.resources = { stagingDir ->  // update staging directory contents here  } |

# 插件仓库

## grails中央插件仓库

插件都分布在grails的中央插件仓库中。可以使用list-plugins命令查看插件：

|  |
| --- |
| grails list-plugins |

# 理解插件的结构

插件的目录结构和常规项目类似：

|  |
| --- |
| + grails-app  + controllers  + domain  + taglib  etc.  + lib  + src  + java  + groovy  + web-app  + js  + css |

一旦插件被安装到某个项目中，插件的grails-app目录下的内容会被复制到常规项目的/plugin/插件名/grails-app目录下。这些内容并不会被复制到项目的源文件目录下，插件中的内容不会影响用到它的项目的源文件。

对于静态资源文件则有所不同。插件就像一个web应用一样，所有的静态资源文件都在web-app目录下面，可以像在web项目里头一样来引用一个资源文件：

|  |
| --- |
| <g:resource dir="js" file="mycode.js" /> |

当在开发插件的时候，资源文件的路径可能是/js/mycode.js，而插件被安装到某个项目之后，资源文件的路径可能就变成/plugin/myplugin-0.1/js/mycode.js。

不管在开发过程还是在安装到某个项目之后都可以使用pluginContextPath来获取插件当前的路径。

插件src/java和src/groovy目录下的源代码会编译后放到用到它的项目的web-app/WEB-INF/classes目录下面。

# 提供基本的artefacts

## 增加新的grails-app artifact（controller、tagLib、service等）

通过在grails-app目录下新增对应的artefact，插件可以给使用它的项目增加这些由它提供的artefact。

|  |
| --- |
| + ExamplePlugin.groovy  + scripts  + grails-app  + controllers <-- additional controllers here  + services <-- additional services here  + etc. <-- additional XXX here  + lib |

## 提供view、template

插件可以给使用它的项目带来新的view和template。grails会现在应用中查找某个view，如果没有找到这个view，就会到插件中去找。这就意味着，应用可以提供自己的view来覆盖掉插件的view。

例如：加入有一个有插件amazon提供的BookController。如果list action执行完毕，grails会先找grails-app/views/book/list.gsp，如果没有找到，就会去找plugins/amazon/grails-app/views/list.gsp。

## excluded artefact

默认情况下，插件打包的时候会排除掉下面的文件：

* grails-app/conf/BootStrap.groovy
* grails-app/conf/BuildConfig.groovy (although it is used to generate dependencies.groovy)
* grails-app/conf/Config.groovy
* grails-app/conf/DataSource.groovy (and any other \*DataSource.groovy)
* grails-app/conf/UrlMappings.groovy
* grails-app/conf/spring/resources.groovy
* Everything within /web-app/WEB-INF
* Everything within /web-app/plugins/\*\*
* Everything within /test/\*\*
* SCM management files within \*\*/.svn/\*\* and \*\*/CVS/\*\*

如果，想要web-app/WEB-INF目录下的某些文件打包到插件中，那最好修改插件的scripts/\_Install.groovy文件。

UrlMapping.groovy不会被打包到插件中，但是可以给mapping文件命名一个别的名字，这样这个mapping文件就会被打包到插件中了。例如：grails-app/conf/BlogUrlMapping.groovy。

# Evaluating conventions

在插件中需要知道当前项目有哪些可以用的controller、service等artefacts。这个过程就是evaluating Conventions。

插件有一个可用的隐藏对象application，application是类GrailsApplication的实例

artefact都实现GrailsClass接口。例如，获取当前应用的所有的GrailsClass实例：

|  |
| --- |
| for (grailsClass in application.allClasses) {  println grailsClass.name  } |

GrailsApplication还可以缩小查找的范围：

|  |
| --- |
| for (controllerClass in application.controllerClasses) {  println controllerClass.name  } |

可用的动态方法有：

* \*Classes - Retrieves all the classes for a particular artefact name. For example application.controllerClasses.
* get\*Class - Retrieves a named class for a particular artefact. For example application.getControllerClass("PersonController")
* is\*Class - Returns true if the given class is of the given artefact type. For example application.isControllerClass(PersonController)

The GrailsClass interface has a number of useful methods that let you further evaluate and work with the conventions. These include:

* getPropertyValue - Gets the initial value of the given property on the class
* hasProperty - Returns true if the class has the specified property
* newInstance - Creates a new instance of this class.
* getName - Returns the logical name of the class in the application without the trailing convention part if applicable
* getShortName - Returns the short name of the class without package prefix
* getFullName - Returns the full name of the class in the application with the trailing convention part and with the package name
* getPropertyName - Returns the name of the class as a property name
* getLogicalPropertyName - Returns the logical property name of the class in the application without the trailing convention part if applicable
* getNaturalName - Returns the name of the property in natural terms (eg. 'lastName' becomes 'Last Name')
* getPackageName - Returns the package name

# 挂接（hooking）到build事件

## Post-install配置和参与更新

Grails插件能够做到post-install配置并且会参与到应用的升级过程中。这主要很两个脚本有关，scripts目录下的\_Install.groovy和\_Upgrade.groovy。\_Install.groovy脚本在插件安装到应用之后执行，\_Upgrade.groovy在每次应用执行Upgrade命令的时候执行。脚本使用Gant脚本命令编写。脚本除了可以使用Gant的标准变量之外，还可以使用一个pluginBasedir变量。pluginBasedir指向插件的安装路径。

下面的例子在grails-app目录下创建一个新目录并载入一个配置模板文件：

|  |
| --- |
| ant.mkdir(dir: "${basedir}/grails-app/jobs")  ant.copy(file: "${pluginBasedir}/src/samples/SamplePluginConfig.groovy",  todir: "${basedir}/grails-app/conf") |

在自定义的脚本文件中不能使用pluginBasedir，但是可以使用xxxPluginDir，xxx是插件名。

## 脚本事件

hook到脚本事件主要是指修改\_Events.groovy中的内容。

# 挂接（hooking）到运行时配置

## 挂机到grails Spring配置

在插件的描述文件中可以hook到spring的配置，只要在文件中增加一个doWithSpring属性，并赋值一个closure代码块。例如i18n插件：

|  |
| --- |
| import org.springframework.web.servlet.i18n.CookieLocaleResolver  import org.springframework.web.servlet.i18n.LocaleChangeInterceptor  import org.springframework.context.support.ReloadableResourceBundleMessageSource  class I18nGrailsPlugin {  def version = "0.1"  def doWithSpring = {  messageSource(ReloadableResourceBundleMessageSource) {  basename = "WEB-INF/grails-app/i18n/messages"  }  localeChangeInterceptor(LocaleChangeInterceptor) {  paramName = "lang"  }  localeResolver(CookieLocaleResolver)  }  } |

上面的例子使用Spring Bean Builder向spring容器注册了一个messageSource等几个bean。

## 参与web.xml文件的产生过程

grails应用是在载入时产生web.xml文件的，grails插件不能修改web.xml文件的内容，但是可以参与到web.xml文件的产生过程中。可以在插件描述文件中增加一个doWithWebDescriptor属性，并赋予一个closure。例如：

|  |
| --- |
| def doWithWebDescriptor = { webXml ->  def mappingElement = webXml.'servlet-mapping'  def lastMapping = mappingElement[mappingElement.size() - 1]  lastMapping + {  'servlet-mapping' {  'servlet-name'("grails")  'url-pattern'("\*.dispatch")  }  }  } |

上面的例子就是在web.xml文件中最后一个servlet-mapping标签后面增加一个用于grails servlet的servlet-mapping标签。

可以在某个filter标签后插入一个filter标签：

|  |
| --- |
| def doWithWebDescriptor = { webXml ->  ...  // Insert the Spring Security filter after the Spring  // character encoding filter.  def filter = webXml.'filter-mapping'.find {  it.'filter-name'.text() == "charEncodingFilter"  }  filter + {  'filter-mapping'{  'filter-name'('springSecurityFilterChain')  'url-pattern'('/\*')  }  }  } |

## Doing Post Initialisation Configuration

有时希望在Spring的Application context构建完成之后再做一些工作，这时就可以定义一个名为doWithApplicationContext的closure。例如：

|  |
| --- |
| class SimplePlugin {  def name = "simple"  def version = "1.1"  def doWithApplicationContext = { appCtx ->  def sessionFactory = appCtx.sessionFactory  // do something here with session factory  }  } |

# 在运行时增加动态方法

## 基本

Grails插件可以向Grails class注册新的动态方法，只需定义一个名为doWithDynamicMethods的closure：

|  |
| --- |
| class ExamplePlugin {  def doWithDynamicMethods = { applicationContext ->  for (controllerClass in application.controllerClasses) {  controllerClass.metaClass.myNewMethod = {-> println "hello world" }  }  }  } |

这样所有的controller都有了一个名为myNewMethod的动态方法。

如果想给某个class增加动态方法，则直接给这个class的metaClass赋值新方法就行了：

|  |
| --- |
| class ExamplePlugin {  def doWithDynamicMethods = { applicationContext ->  String.metaClass.swapCase = {->  def sb = new StringBuilder()  delegate.each {  sb << (Character.isUpperCase(it as char) ?  Character.toLowerCase(it as char) :  Character.toUpperCase(it as char))  }  sb.toString()  }  assert "UpAndDown" == "uPaNDdOWN".swapCase()  }  } |

上面的例子给String类增加了一个swapCase的动态方法。

## Interacting with the ApplicationContext

doWithDynamicMethod closure传入了一个ApplicationContext类型的实例作为参数，这个参数有很大的作用。例如可以通过它获取sessionFactory：

|  |
| --- |
| import org.springframework.orm.hibernate3.HibernateTemplate  class ExampleHibernatePlugin {  def doWithDynamicMethods = { applicationContext ->  for (domainClass in application.domainClasses) {  domainClass.metaClass.static.load = { Long id->  def sf = applicationContext.sessionFactory  def template = new HibernateTemplate(sf)  template.load(delegate, id)  }  }  }  } |

上面的例子给每个domain class增加了一个load的static dynamic方法。

# 参与auto reloading事件

## 监控资源是否发生改变

有的时候能够监控到某些资源文件发生了改变并能随之作出某些动作的能力是很有用的。例如，如下的Grails service插件：

|  |
| --- |
| class ServicesGrailsPlugin {  …  def watchedResources = "file:./grails-app/services/\*Service.groovy"  …  def onChange = { event ->  if (event.source) {  def serviceClass = application.addServiceClass(event.source)  def serviceName = "${serviceClass.propertyName}"  def beans = beans {  "$serviceName"(serviceClass.getClazz()) { bean ->  bean.autowire = true  }  }  if (event.ctx) {  event.ctx.registerBeanDefinition(  serviceName,  beans.getBeanDefinition(serviceName))  }  }  }  } |

event对象有以下这些有用的属性：

* event.source - The source of the event, either the reloaded Class or a Spring Resource
* event.ctx - The Spring ApplicationContext instance
* event.plugin - The plugin object that manages the resource (usually this)
* event.application - The GrailsApplication instance
* event.manager - The GrailsPluginManager instance

## 影响其他插件

插件之间也可以互相影响，例如service插件reload了一个service之后，需要controller插件也reload对应的contoller，否则将新的service绑定到老的controller上会出现错误。

通过定义influence属性来指定需要受影响的插件：

|  |
| --- |
| def influences = ['controllers'] |

## 观察其它插件

可以通过定义observer属性来指定要观察的插件：

|  |
| --- |
| def observe = ["controllers"] |

上面的例子，当有controller发生变动时，你的插件也能收到event。

也可以定义观察所有的插件：

|  |
| --- |
| def observe = ["\*"] |

Log插件就是使用这样的配置，这样当所有类型的archetype发生变动时它都可以给新装载的对象加上一个log属性。

# 理解插件装载的顺序

## 控制插件的依赖关系

插件可以定义其需要依赖的其它插件。可以使用dependsOn属性：例如hibernate插件：

|  |
| --- |
| class HibernateGrailsPlugin {  def version = "1.0"  def dependsOn = [dataSource: "1.0",  domainClass: "1.0",  i18n: "1.0",  core: "1.0"]  } |

上列说明hibernate插件需要依赖dataSource、domainClass、i18n和core四个插件。

被依赖的插件会比依赖它们的插件更早被载入。

dependsOn属性还可以定义依赖插件的版本范围，例如：

|  |
| --- |
| def dependsOn = [foo: "\* > 1.0"]  def dependsOn = [foo: "1.0 > 1.1"]  def dependsOn = [foo: "1.0 > \*"] |

## 控制插件装载顺序

使用dependsOn属性是一种硬依赖，如果被依赖的插件转载不成功，那么依赖它们的插件也会装载失败。所以也可以使用loadAfter和loadBefore这两个属性来定义软依赖。例如：

|  |
| --- |
| def loadAfter = ['controllers'] |

这样插件就可以依赖比它先装载的插件了。例如：

|  |
| --- |
| if (manager?.hasGrailsPlugin("controllers")) {  openSessionInViewInterceptor(OpenSessionInViewInterceptor) {  flushMode = HibernateAccessor.FLUSH\_MANUAL  sessionFactory = sessionFactory  }  grailsUrlHandlerMapping.interceptors << openSessionInViewInterceptor  } |

## 作用域和环境

还可以定义插件在哪个环节载入和插件的作用域。可以如下声明：

|  |
| --- |
| def environments = ['development', 'test', 'myCustomEnv']  def scopes = [excludes:'war'] |

以上说明插件在开发、测试和myCustomEnv这三个环境下载入，同时插件使用除了war之外的作用域。

插件可用的构建作用域如下：

* test - when running tests
* functional-test - when running functional tests
* run - for run-app and run-war
* war - when packaging the application as a WAR file
* all - plugin applies to all scopes (default)

# Artefact API

artefact就是特殊的groovy类，grails会给这些类加上动态方法，例如domain和controllr类等。grails 插件能够获取当前可用的artefact，并且还可以扩展atefact的功能。grails也允许创建自己的artefact。

## 获取可用的artefs

插件开发人员往往需要知道当前应用可用的artefact，可以通过grailsApplication对象获取当前有哪些domain、controller、service等artefact对象。grailsApplication对象已经自动注入到了controller和GSP，并且可以通过Grails的依赖注入机制注入到任何class中。

grailsApplication对象有几个重要的方法用来查询artefact，最常用的属性：

|  |
| --- |
| for (cls in grailsApplication.<artefactType>Classes) {  …  } |

可用的artefactType有：

* domain
* controller
* tagLib
* service
* codec
* bootstrap
* urlMappings

所以，如果想要获取所有的domain对象，可以使用如下代码：

|  |
| --- |
| for (cls in grailsApplication.domainClasses) {  …  } |

查找URLMapping对象：

|  |
| --- |
| for (cls in grailsApplication.urlMappingsClasses) {  …  } |

需要知道的是返回的对象类型并不是Class，而是GrailsClass的实例。GrailsClass对象有几个重要的属性：

* shortName - the class name of the artefact without the package (equivalent of Class.simpleName).
* logicalPropertyName - the artefact name in property form without the 'type' suffix. So MyGreatController becomes 'myGreat'.
* isAbstract() - a boolean indicating whether the artefact class is abstract or not.
* getPropertyValue(name) - returns the value of the given property, whether it's a static or an instance one. This works best if the property is initialised on declaration, e.g. static transactional = true.

artefact API还可以通过名字获取artefact或者判断class是否是一个artefact:

* get<type>Class(String name)
* is<type>Class(Class clazz)

## 增加自定义artefact

插件可以很容易的增加自定义artefact。只需要实现一个ArtefactHandler接口并在插件中注册就行了：

|  |
| --- |
| class MyGrailsPlugin {  def artefacts = [ org.somewhere.MyArtefactHandler ]  …  } |

赋给arteFacts属性的列表可以是handler class或者是handler class的实例。

[ArtefactHandler](http://grails.org/doc/latest/api/org/codehaus/groovy/grails/commons/ArtefactHandler.html)接口有一个简单的实现类[ArtefactHandlerAdapter](http://grails.org/doc/latest/api/org/codehaus/groovy/grails/commons/ArtefactHandlerAdapter.html)，使用ArtefactHandlerAdapter可以简化部分开发工作。

除了artefactHandler以外，每一个先进的artefact都还需要一个实现GrailsClass的wrapper class。GrailsClass也有一个基本的实现类：[AbstractInjectableGrailsClass](http://grails.org/doc/latest/api/org/codehaus/groovy/grails/commons/AbstractInjectableGrailsClass.html)。

理解自定义artefact的最好方式就是看例子，可以看看quartz插件是如何实现自己的自定义artefact的:

* [GrailsJobClass](http://github.com/nebolsin/grails-quartz/blob/master/src/java/grails/plugins/quartz/GrailsJobClass.java)
* [DefaultGrailsJobClass](http://github.com/nebolsin/grails-quartz/blob/master/src/java/grails/plugins/quartz/DefaultGrailsJobClass.java)
* [JobArtefactHandler](http://github.com/nebolsin/grails-quartz/blob/master/src/java/grails/plugins/quartz/JobArtefactHandler.java)