Grails的web flow基于Spring的web flow项目。一个web flow就是跨越多个http请求的一个会话。web flow用flow scope来保存跨多个请求的状态信息。一个flow也定义开始和结束两个状态。

web flow不使用http session，它把http请求的状态信息保存在一个序列化的form里，然后再把这个序列化的form用一个key值保存起来。这个key值作为一个http请求的参数使得flow能够像session那样跨请求保存状态信息。flow的这种实现方式使它比http session更加灵活。

# 安装插件

对于grails2.2.X的版本,安装插件如下:

|  |
| --- |
| compile ':webflow:2.0.8.1', {  exclude 'grails-webflow'  } |

对于grails2.3以上的版本,安装插件如下:

|  |
| --- |
| compile ':webflow:2.0.8.1' |

# 创建一个flow

创建一个常规的controller，然后在controller中创建一个已Flow结尾的action方法，这样既创建了一个flow：

|  |
| --- |
| class BookController {  def index() {  redirect(action: "shoppingCart")  }  def shoppingCartFlow = {  …  }  } |

通常我们省略Flow而称这个web flow为shoppingCart。

# 开始和结束状态

web flow都有一个开始和结束状态。用户第一次初始化一个会话的时候既进入了开始状态。在代码中，开始状态就是第一个接受一个closure的方法调用：

|  |
| --- |
| class BookController {  …  def shoppingCartFlow ={  showCart {  on("checkout").to "enterPersonalDetails"  on("continueShopping").to "displayCatalogue"  }  …  displayCatalogue {  redirect(controller: "catalogue", action: "show")  }  displayInvoice()  }  } |

上面的例子中showCart就是flow的开始状态。因为showCart没有定义一个action或重定向到另一个action，所以这是一个view state。按照约定，显示grails-app/views/book/shoppingCart/showCart.gsp这个view。

注意，和常规的controller的action不同，flow的view都在和flow同名的子目录下面：

grails-app/views/book/shoppingCart。

shoppingCart flow有两个可选的结束状态。第一个是displayCatalogue，displayCatalogue将请求重定向到另一个controller的action。第二个结束状态是displayInvoice，这个结束状态因为是空的所以就是简单的显示grails-app/views/book/shoppingCart/displayInvoice.gsp这个view。

一旦一个flow结束了以后，只能从开始状态再重新开始。

# action states和view states

## view states

一个view state就是一个没有定义action和调用重定向redirect的节点。例如：

|  |
| --- |
| enterPersonalDetails {  on("submit").to "enterShipping"  on("return").to "showCart"  } |

默认情况下，这个state使用grails-app/views/book/shoppingCart/enterPersonalDetails.gsp作为view。state还定义了两个事件：submit和return。view用于触发这些事件。

使用render方法，可以给view state指定一个view:

|  |
| --- |
| enterPersonalDetails {  render(view: "enterDetailsView")  on("submit").to "enterShipping"  on("return").to "showCart"  } |

现在它使用grails-app/views/book/shoppingCart/enterDetailsView.gsp这个页面作为view了。

也使用'/'开头也可以指定shared view：

|  |
| --- |
| enterPersonalDetails {  render(view: "/shared/enterDetailsView")  on("submit").to "enterShipping"  on("return").to "showCart"  } |

现在它使用grails-app/views/shared/enterDetailsView.gsp了。

## action states

action state是指执行代码而不是用于显示view的状态。action state的执行结果用于flow的流转。创建一个action state就是调用一个action方法并传入一个closure：

|  |
| --- |
| listBooks {  action {  [bookList: Book.list()]  }  on("success").to "showCatalogue"  on(Exception).to "handleError"  } |

上面的例子就是创建了一个listBooks action。

flow的action和controller的action很像，甚至可以直接使用contrller的action。如果一个action执行没有错误，正常返回就会触发一个success事件。上面的例子中，action返回一个Map，这个Map就被看成一个model给view使用，并自动放到了flow scope中了。

另外，action如果执行发生异常，也可以使用异常处理器来处理异常：

|  |
| --- |
| on(Exception).to "handleError" |

这是flow流转到了handleError这个状态。

action也可以很复杂并和flow request context交互：

|  |
| --- |
| processPurchaseOrder {  action {  def a = flow.address  def p = flow.person  def pd = flow.paymentDetails  def cartItems = flow.cartItems  flow.clear()  def o = new Order(person: p, shippingAddress: a, paymentDetails: pd)  o.invoiceNumber = new Random().nextInt(9999999)  for (item in cartItems) { o.addToItems item }  o.save()  [order: o]  }  on("error").to "confirmPurchase"  on(Exception).to "confirmPurchase"  on("success").to "displayInvoice"  } |

### Transition Action

还有一中action被称为Transition action，用于事件被触发后的状态流转。例如：

|  |
| --- |
| enterPersonalDetails {  on("submit") {  log.trace "Going to enter shipping"  }.to "enterShipping"  on("return").to "showCart"  } |

这个action没有执行什么代码而是根据submit和return事件进行状态流转。注意当submit事件触发后，还执行了一个记录日志的操作。

# flow事件

为了在flow中不同状态流转，需要触发对应的事件。事件可以从view state或者action state中触发。

## 从view state中触发事件

就如上面的例子中的start state，有两个用于触发的事件checkout和continueShopping：

|  |
| --- |
| def shoppingCartFlow = {  showCart {  on("checkout").to "enterPersonalDetails"  on("continueShopping").to "displayCatalogue"  }  …  } |

因为showCart状态是一个view state，所以会显示grails-app/book/shoppingCart/showCart.gsp这个view。在这个view中必须有触发事件的组件。在form中，可以使用submitButton标签：

|  |
| --- |
| <g:form>  <g:submitButton name="continueShopping" value="Continue Shopping" />  <g:submitButton name="checkout" value="Checkout" />  </g:form> |

这个form提交的时候自动调用shoppingCartFlow，对应的submitButton触发对应的事件。

如果，不使用form，也可通过grails的link标签：

|  |
| --- |
| <g:link event="checkout" /> |

## 从action中触发事件

在action中通过方法调用触发事件，可以使用error()和success()方法触发error和success事件。例如：

|  |
| --- |
| enterPersonalDetails {  on("submit") {  def p = new Person(params)  flow.person = p  if (!p.validate()) return error()  }.to "enterShipping"  on("return").to "showCart"  } |

上面的transition action调用error()触发了error事件会使flow再次流转到enterPersionalDetails的flow状态。

也可以使用任何方法触发事件，例如：

|  |
| --- |
| shippingNeeded {  action {  if (params.shippingRequired) yes()  else no()  }  on("yes").to "enterShipping"  on("no").to "enterPayment"  } |

# flow scopes

前面提到了flow把http请求的状态保存在一个叫flow的scope里。这里的flow scope是和session scope一样的类似概念。grails总共有如下的各种类型的scope，每种scope的作用域也不一样:

* request - 作用域为单个http请求
* flash - 作用域为当前http请求和下一个http请求
* flow - 作用域为单个flow，从flow开始到flow结束
* conversation - 当前flow还包括子flow
* session - 作用域为用户会话

flow action返回的model map对象会被自动保存到flow scope中。例如下面这个transition action：

|  |
| --- |
| enterPersonalDetails {  on("submit") {  [person: new Person(params)]  }.to "enterShipping"  on("return").to "showCart"  } |

需要意识到每进入一个flow的状态都是一个新的http请求，所以把数据保存在http request scope中，那么在进入下一个flow的状态时就没有了。另外web flow在状态流转时会：

* 自动将flash scope中的对象拿出来放到http request scope中
* 会将flow和conversion scope中的对象合并，并自动作为model转给view。所以在gsp中可以直接引用这些对象。

## Flow Scope和Serialization

放入flash、flow和conversation scope中的对象都必须实现serialization接口，否则会抛出异常。特别是对于domain对象，因为domain对象需要保存到某个scope中才能在view中被现实，所以domain对象需要实现serialization接口。例如，有下面一个domain类：

|  |
| --- |
| class Book {  String title  } |

如果要将Book对象的实例放到flow scope中，则Book domain class需要实现serialization接口：

|  |
| --- |
| class Book implements Serializable {  String title  } |

同样如果domain对象的某个关联属性没有实现serialization接口，则也会发生错误，例如:

|  |
| --- |
| class Book implements Serializable {  String title  Author author  } |

如果Author类没有实现serialization则Book类的实例放到flow scope时就会发生错误。

domain对象中的一些监听某些事件的方法也会引起错误，例如onLoad closure：

|  |
| --- |
| class Book implements Serializable {  String title  def onLoad = {  println "I'm loading"  }  } |

上面的例子，Book对象的实例发到flow scope时会抛出异常。改成如下则没有问题了：

|  |
| --- |
| class Book implements Serializable {  String title  transient onLoad = {  println "I'm loading"  }  } |

或者不使用闭包，而改成方法也没有问题：

|  |
| --- |
| class Book implements Serializable {  String title  def onLoad() {  println "I'm loading"  }  } |

# 数据绑定和校验

flow的数据绑定和校验基本和controller类似，例如前面例子中用到的flow的start状态：

|  |
| --- |
| enterPersonalDetails {  on("submit").to "enterShipping"  on("return").to "showCart"  } |

上面的start状态展现一个view，并等到view中的form触发submit或者return事件：

|  |
| --- |
| <g:form>  <!-- Other fields -->  <g:submitButton name="submit" value="Continue"></g:submitButton>  <g:submitButton name="return" value="Back"></g:submitButton>  </g:form> |

对于这个form提交的数据，可以通过如下方法绑定和校验：

|  |
| --- |
| enterPersonalDetails {  on("submit") {  flow.person = new Person(params)  !flow.person.validate() ? error() : success()  }.to "enterShipping"  on("return").to "showCart"  } |

上面的例子讲了params是如何绑定到Persion对象的，同样调用person.validate()方法进行数据校验，当校验出错时调用error()触发error事件，error事件会时flow再次流转到enterPersonalDetails状态，这样用户就可以重新输入正确的数据。

同样，flow中的数据绑定也支持command对象，例如：

|  |
| --- |
| enterPersonalDetails {  on("submit") { PersonDetailsCommand cmd ->  flow.personDetails = cmd  !flow.personDetails.validate() ? error() : success()  }.to "enterShipping"  on("return").to "showCart"  } |

上面就使用了PersonDetailsCommand对象做数据绑定。

# subflow和conversation

## 调用subflow

web flow中还可以调用子flow，例如下面的例子：

|  |
| --- |
| def searchFlow = {  displaySearchForm {  on("submit").to "executeSearch"  }  executeSearch {  action {  [results:searchService.executeSearch(params.q)]  }  on("success").to "displayResults"  on("error").to "displaySearchForm"  }  displayResults {  on("searchDeeper").to "extendedSearch"  on("searchAgain").to "displaySearchForm"  }  extendedSearch {  // Extended search subflow  subflow(controller: "searchExtensions", action: "extendedSearch")  on("moreResults").to "displayMoreResults"  on("noResults").to "displayNoMoreResults"  }  displayMoreResults()  displayNoMoreResults()  } |

使用了subflow放到调用了一个子flow。如果sub flow和调用它的flow在同一个controller中，那么subflow方法的controller参数并不是必须的。

子flow其实就是另一个flow，例如extendedSearche：

|  |
| --- |
| def extendedSearchFlow = {  startExtendedSearch {  on("findMore").to "searchMore"  on("searchAgain").to "noResults"  }  searchMore {  action {  def results = searchService.deepSearch(ctx.conversation.query)  if (!results) return error()  conversation.extendedResults = results  }  on("success").to "moreResults"  on("error").to "noResults"  }  moreResults()  noResults()  } |

注意到results对象被放到了conversation scope中，conversation scope可以使得状态可以在子 flow和调用它的flow之间共享。并且子flow中的moreResults()和noResults()方法会触发父flow中的moreResults和noResults事件：

|  |
| --- |
| extendedSearch {  // Extended search subflow  subflow(controller: "searchExtensions", action: "extendedSearch")  on("moreResults").to "displayMoreResults"  on("noResults").to "displayNoMoreResults"  } |

## subflow的输入和输出

subflow还可以定义输入和输出用于从父flow中接收数据和返回结果给父flow。假设有一个根据特定的专业技能查找个人信息的flow：

|  |
| --- |
| def searchFlow = {  input {  expertise(required: true)  title("Search person")  }  search {  onEntry {  [personInstanceList: Person.findAllByExpertise(flow.expertise)]  }  on("select") {  flow.person = Person.get(params.id)  }.to("selected")  on("cancel").to("cancel")  }  selected {  output {  person {flow.person}  }  }  cancel()  }  } |

这个flow接受两个数据参数，一个必须的参数expertise，一个不是必须的有默认值的参数title。所有的输入参数都放在flow scope中，所以和本地变量一样只在当前flow中可见。

注意flow的结束状态还可定义output参数，例如persion {flow.persion}。如果output参数是一个closue，则每一次flow调用的时候都会被执行一次，如果output参数不是一个closure，则其值就会在flow定义的时候就固定了而不会每一次执行都重新计算。

调用有input和output的subflow的例子如下：

|  |
| --- |
| def newProjectWizardFlow = {  ...  managerSearch {  subflow(controller: "person", action: "search",  input: [expertise : "management", title: "Search project manager"])  on("selected") {  flow.projectInstance.manager = currentEvent.attributes.person  }.to "techleadSearch"  }  techleadSearch {  subflow(controller: "person", action: "search",  input: [expertise : { flow.technology }, title: "Search technical lead"])  on("selected") {  flow.projectInstance.techlead = currentEvent.attributes.person  }.to "projectDetails"  }  ...  } |

用Map类型传入intput参数，sublfow的output参数通过currentEvent.attributes获得。