
Tapestry 5.1

实例教程

Copyright 2009

傣许名 daixuming.t5@gmail.com

目录

前言.....	5
第一章 Hello Tapestry.....	6
安装 JDK	6
安装 Eclipse	6
安装 Tomcat	6
下载 Tapestry	6
新建 Web 工程.....	6
创建页面	9
把工程加入 Tomcat 服务器	10
运行和调试	10
第二章 基础.....	13
页面类.....	13
页面模板	13
Expansion.....	15
绑定表达式	15
使用组件	17
异常报告	18
类和模板的重新载入	19
定义组件的其它方法	20
Html 代码	21
组件事件请求.....	22
页面间的数据传递.....	25
页面池	28
禁止重定向	29
@Persist.....	30
Activation Context.....	31
SSO	32

子目录.....	37
第三章 创建自己的组件	38
创建组件	39
组件参数	40
参数的属性	43
创建布局	46
子目录	47
组件的解析	47
其它 Tapestry 元素	52
第四章 表单	54
Form、TextField、PasswordField 和 Label 组件	55
RadioGroup 和 Radio 组件	58
Select 组件	60
Checkbox 组件	63
Submit 组件	64
使用 Validator 校验表单	66
使用 Errors 组件显示错误信息	69
自定义错误信息	70
客户端校验	71
校验多个字段	72
第五章 本地化	74
支持特定的语言	74
消息目录	74
本地化整个模板	82
本地化 Asset	83
切换 Locale	85
第六章 Ajax	91
添加 Javascript	94
基础 Javascript 库	96
安装 Firebug	97

添加 Javascript 库	97
Autocomplete Mixin.....	99
Zone 组件	100
更新多个 Zone.....	107
第七章 集成 Spring.....	109
Tapestry IoC	113
依赖库	118
创建 Bean	118
装配 Bean	119
配置 Tapestry	119
在页面中注入 Bean.....	120

前言

Tapestry 是一个基于组件的 Web 开发框架。

本书的目的是以尽量短的篇幅介绍 Tapestry 5.1，让你用最少的时间了解 Tapestry 5.1 的大多数特性。

本书通过实例讲解 Tapestry 5.1。笔者为每章编写了一个小例子，也有两章共用一个例子的。这些例子都是经过调试能正确运行的。

本书使用的参考资料主要是 Tapestry 官方文档，其中有很多的名词术语，部分已经翻译成中文，但是有些名词术语，还是保留英文，希望不会给你造成太大影响。不妥之处，欢迎指教。

如果你有任何意见或者建议，可以通过 daixuming.t5@gmail.com 与笔者联系。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。

版权所有，侵权必究。

第一章 Hello Tapestry

本章主要内容是在 Windows 中搭建 Tapestry 5 开发环境，并开发一个简单的“Hello Tapestry”应用。

安装 JDK

Tapestry 5 使用了注释（Annotation），而注释是 Java 1.5（或者说 Java 5）才有的特性，因此我们必须选择版本高于 1.5 的 JDK。

从 <http://java.sun.com/javase/downloads/index.jsp> 可以下载到 JDK，我们选择 JDK 6，文件名类似 jdk-6u13-windows-i586-p.exe，下载后需要安装。

安装 Eclipse

从 <http://www.eclipse.org/downloads/> 可以下载到 Eclipse，注意要找“Eclipse IDE for Java EE Developers”，文件名类似 eclipse-jee-ganymede-SR2-win32.zip，下载后解压就可以用了。

安装 Tomcat

从 <http://tomcat.apache.org/> 可以下载到 Tomcat，文件名类似 apache-tomcat-6.0.18.zip，下载后解压成 D:\apache-tomcat-6.0.18，注意不要在 D:\apache-tomcat-6.0.18 目录下再嵌套一个 apache-tomcat-6.0.18 目录。

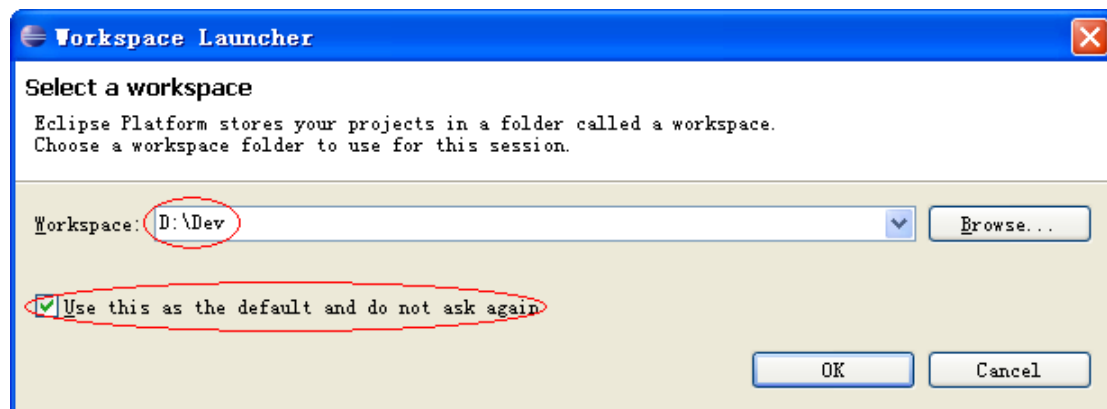
下载 Tapestry

从 <http://tapestry.apache.org/> 下载 tapestry-bin-5.1.0.5.zip 备用。

新建 Web 工程

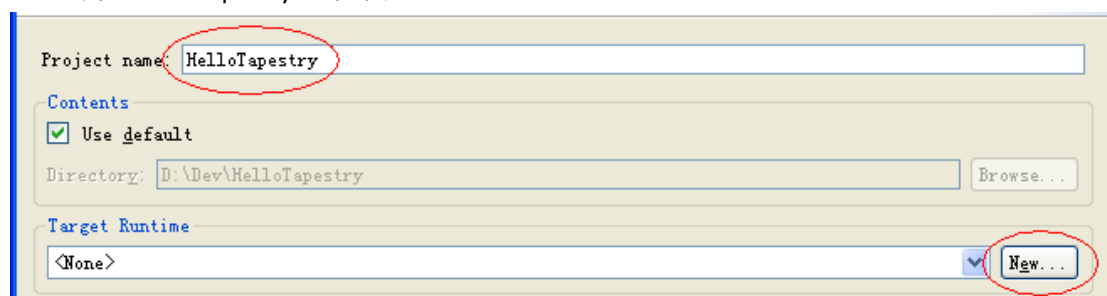
运行 Eclipse。

如果是第一次运行，Eclipse 会要求你选择一个工作空间，如图 1-1。选择一个目录（比如 D:\Dev）作为 Eclipse 的工作空间，以后我们创建的工程就保存在这个目录中。选中下方的复选框，下次运行 Eclipse 时就不会再弹出这个对话框了。在 Eclipse 启动后，关掉“Welcome”页面。



(图 1-1)

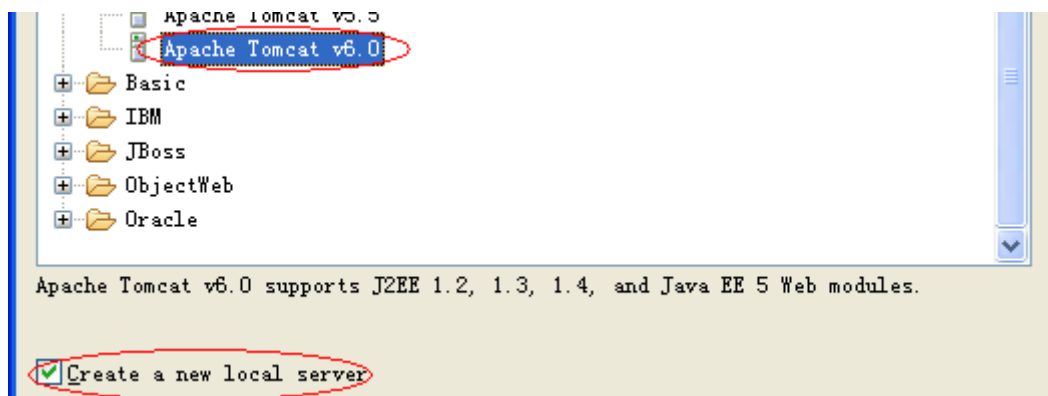
Eclipse 启动后，点击 Eclipse 菜单【File】【New】【Dynamic Web Project】。设置 Project Name 为 “HelloTapestry”，如图 1-2。



(图 1-2)

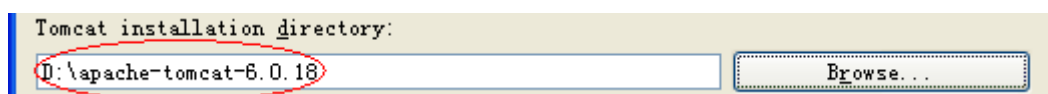
点击 Target Runtime 右边的 New 按钮。

选择 Apache Tomcat v6.0，并选中下方 “Create a new local server” 复选框，如图 1-3。



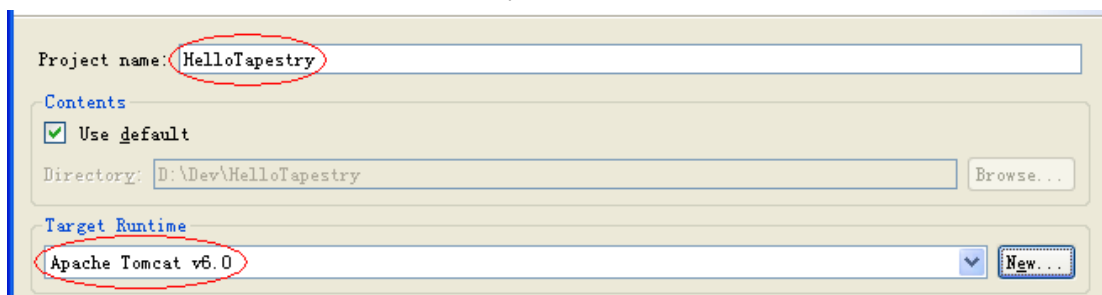
(图 1-3)

点击 Next 按钮。在 Tomcat Installation Directory 栏选择 Tomcat 安装目录（D:\apache-tomcat-6.0.18），如图 1-4。



(图 1-4)

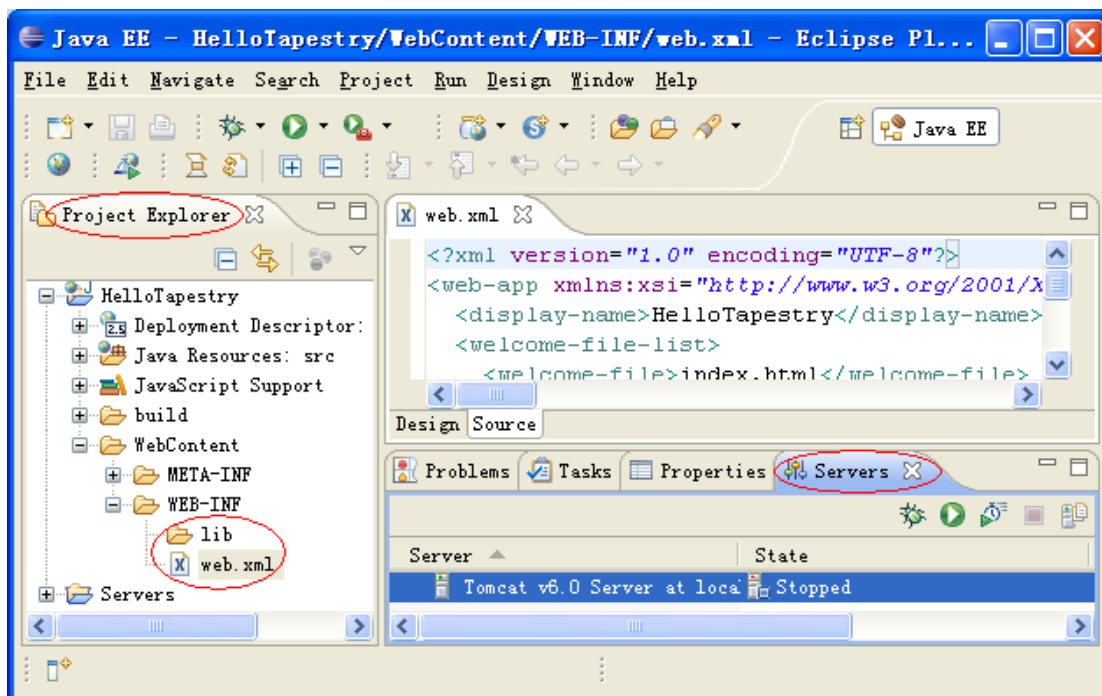
点击 Finish 按钮，完成 Tomcat Server 的创建。完成后的设置如图 1-5。如果下次再新建一个 Web 工程，我们就可以直接选择 “Apache Tomcat v6.0”，不用再新建一个了。



(图 1-5)

点击 Finish 按钮创建工程。

图 1-6 中，左边的 Project Explorer 面板显示工程的目录结构，右下有一个 Servers 面板，里面列出的是我们创建的 Tomcat Server。



(图 1-6)

接下来，打开 WebContent/WEB-INF/web.xml 文件，把其代码改为如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
    <display-name>Hello Tapestry</display-name>
    <context-param>
        <param-name>tapestry.app-package</param-name>
        <param-value>example.hellotapestry</param-value>
    </context-param>
```

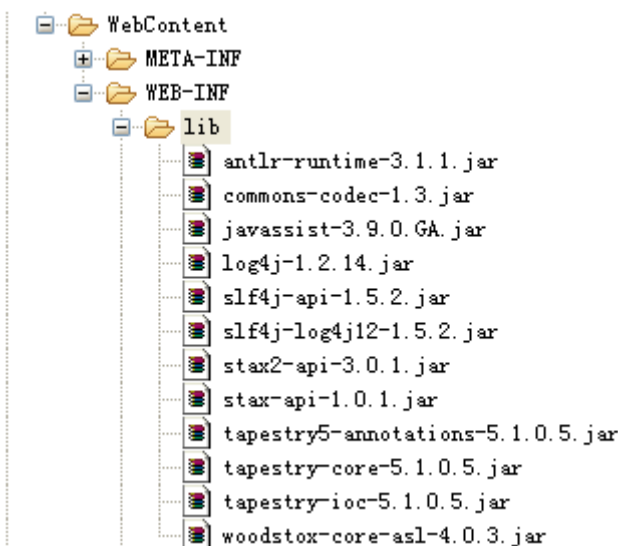


```

<filter>
    <filter-name>app</filter-name>
    <filter-class>org.apache.tapestry5.TapestryFilter</filter-clas
s>
</filter>
<filter-mapping>
    <filter-name>app</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
</web-app>

```

接着把前面下载的 `tapestry-bin-5.0.18.zip` 解压，进入 `lib` 目录，选中需要的 `jar` 文件（只有在图 1-7 列出的 `jar` 文件才是我们需要的），按 `Ctrl+C`，然后在 Eclipse 的 `Project Explorer` 面板中选中 `WebContent/WEB-INF/lib` 目录，按 `Ctrl+V`，就可以把我们需要的 `jar` 文件加入工程“HelloTapestry”了，见图 1-7。



(图 1-7)

至此，新建 `Web` 工程的工作全部完成，接下来我们就要着手创建页面了。

创建页面

新建一个 `Start` 类，其包名为“`example.hellotapestry.pages`”，代码如下：

```

public class Start {
    public String getGreeting() {
        return "Hello Tapestry!";
    }
}

```

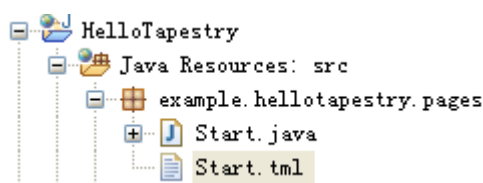
在 `Start` 类所在目录新建文件 `Start.html`，代码如下：

```

<html xmlns:t="http://tapestry.apache.org/schema/tapestry_5_1_0.xsd">
    ${prop:greeting}
</html>

```

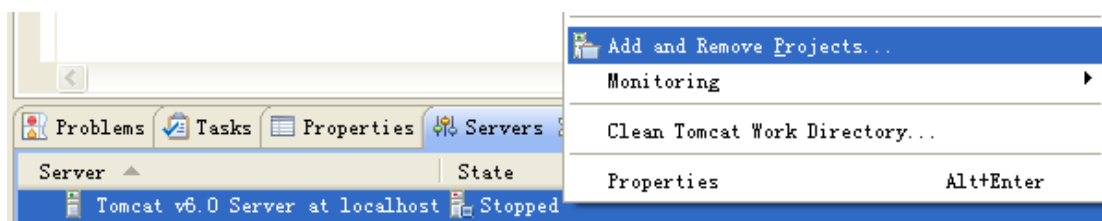
页面创建完成，完成后目录结构如图 1-8。



(图 1-8)

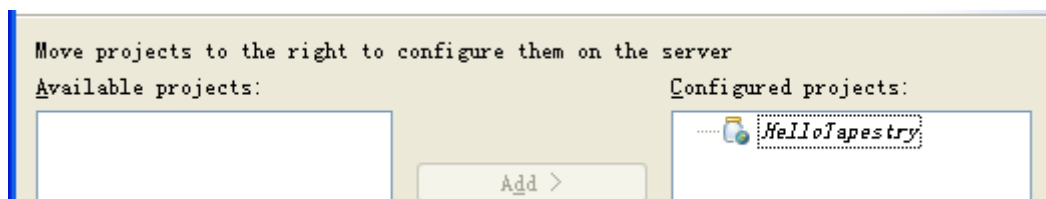
把工程加入 Tomcat 服务器

在 Servers 面板中，右键点击“Tomcat v6.0 Server at localhost”，在弹出菜单中点击“Add and Remove Projects”，如图 1-9。



(图 1-9)

把工程“HelloTapestry”加入右边，如图 1-10。



(图 1-10)

点击 Finish 按钮完成添加。

运行和调试

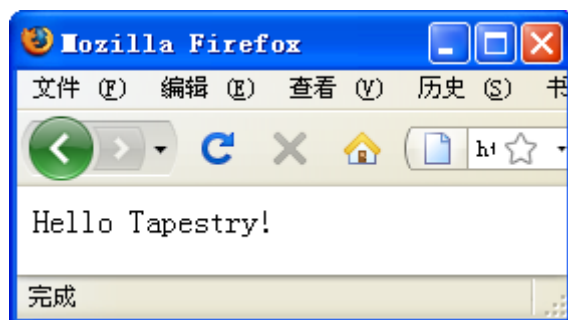
图 1-11 是 Servers 面板上的一组按钮，其中第一个是重启服务器进入调试模式，第二个是重启服务器，第四个是停止服务器。



(图 1-11)

重启服务器（图 1-11 中第二个按钮）。

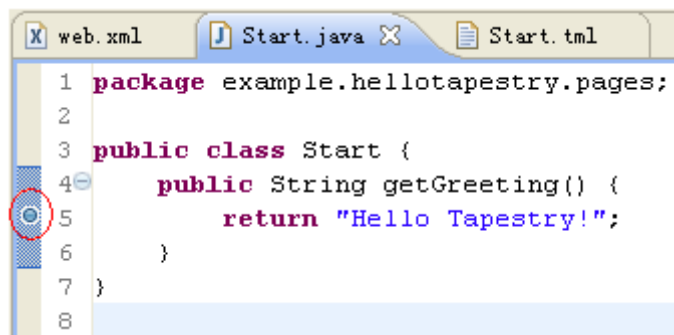
待服务器启动后，在浏览器中打开 <http://localhost:8080/HelloTapestry/Start>，页面效果如图 1-12。



(图 1-12)

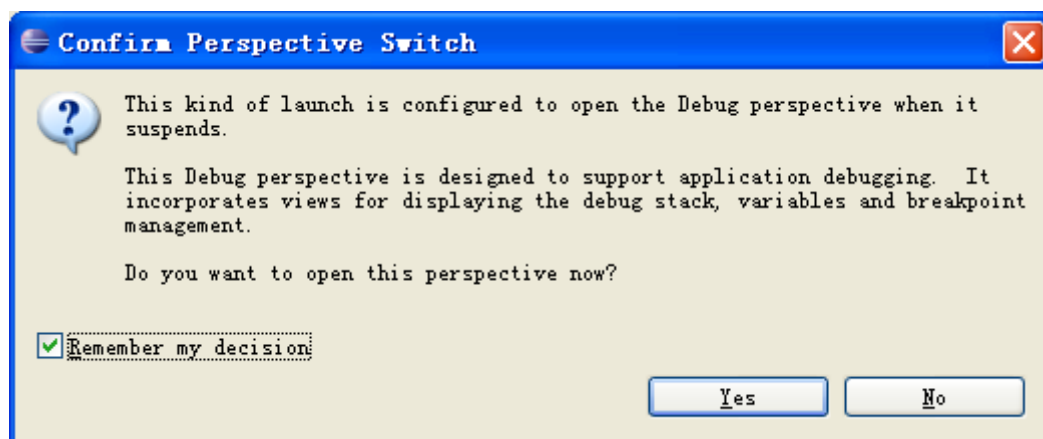
重启服务器进入调试模式 (图 1-11 中第一个按钮)。

在 `Start.java` 的第五行设置断点。如图 1-13, `Start.java` 的第五行左边有个小圆点, 表示在这行已经设置了断点。设置断点的方法就是双击小圆点所在位置, 出现小圆点就表示设置了断点, 再次双击则取消断点。



(图 1-13)

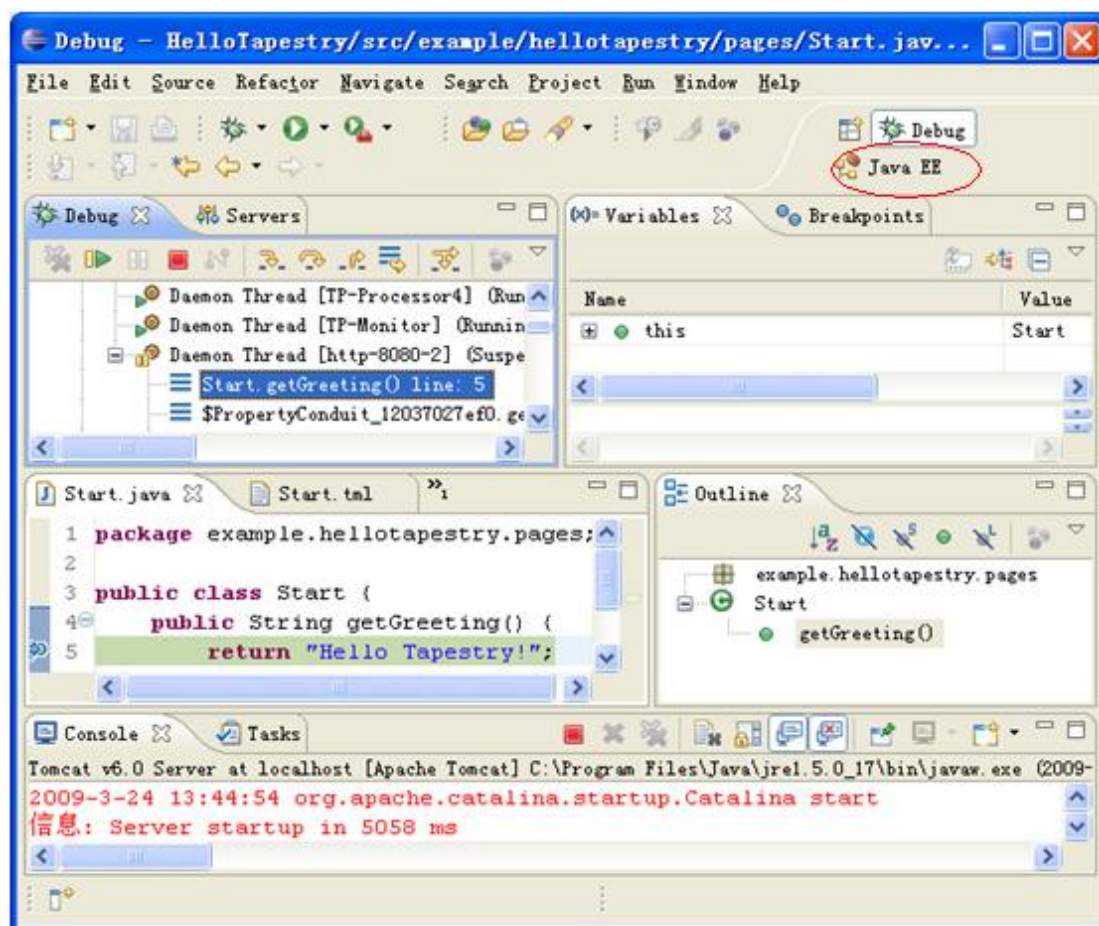
在浏览器中刷新 `Start` 页面, 当程序运行到断点位置, Eclipse 会弹出一个对话框, 问我们是否要切换到 `Debug perspective`, 如图 1-14。



(图 1-14)

选中 “Remember my decision” 复选框, 下次就不会再弹出这个对话框了。点击 `Yes` 按钮, Eclipse 会切换到 `Debug perspective`, 如图 1-15, 我们可以在这里查看变量, 进行单步调试等等。

如果要切换回 `Java EE perspective`, 可点击右上角的 “Java EE”。



(图 1-15)

第二章 基础

本章主要内容是介绍 Tapestry 5 的基础知识。

本章的例子仍然使用 “HelloTapestry”，在前面的基础上逐步增加内容。

一个 Tapestry 应用是由一系列页面组成的。

Tapestry 的缺省首页是 “Start”，我们也可以自己配置缺省首页，Tapestry 的配置后面会讲到。

一个页面一般包括一个页面类和一个页面模板。

页面类

Start 类就是 Start 页面的页面类。

在 Tapestry 5 中，页面类是纯粹的 POJO (Plain Old Java Objects)，无须继承特定的类，也无须实现特定的接口。但是页面类必须放在特定的包里。

我们先回过头看看 web.xml 文件，里面有如下一段代码：

```
<context-param>
  <param-name>tapestry.app-package</param-name>
  <param-value>example.hellotapestry</param-value>
</context-param>
```

我们把 tapestry.app-package 参数设置为了 “example.hellotapestry”，那么页面类就要放在 “example.hellotapestry.pages” 包中，组件类要放在 “example.hellotapestry.components” 包中。如果一个类（通常是抽象类）是其它页面/组件类的基类，那么应该被放在 “example.hellotapestry.base” 包中，而不是 “example.hellotapestry.pages” 或者 “example.hellotapestry.components” 包中，这样它就不再是一个可用的页面/组件了，只用于被其它页面/组件继承。

页面模板

Start 页面的页面模板是 Start.tml 文件，其代码如下：

```
<html xmlns:t="http://tapestry.apache.org/schema/tapestry_5_1_0.xsd">
  ${prop:greeting}
</html>
```

页面模板文件扩展名为 tml (Tapestry Markup Language)。

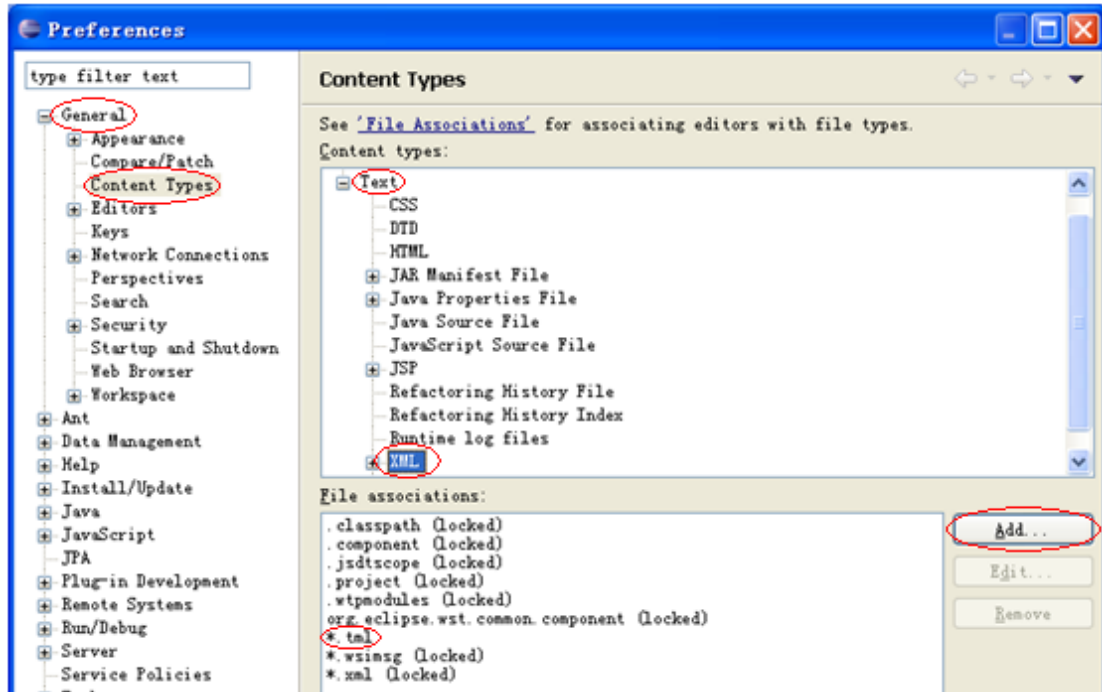
页面模板是一个 well formed XML 文档，这意味着每一个标签都必须有结束标签，每个属性都必须放在引号中，等等。

在页面模板的根元素包含有 Tapestry 5.1 命名空间声明：

```
xmlns:t=http://tapestry.apache.org/schema/tapestry_5_1_0.xsd
```

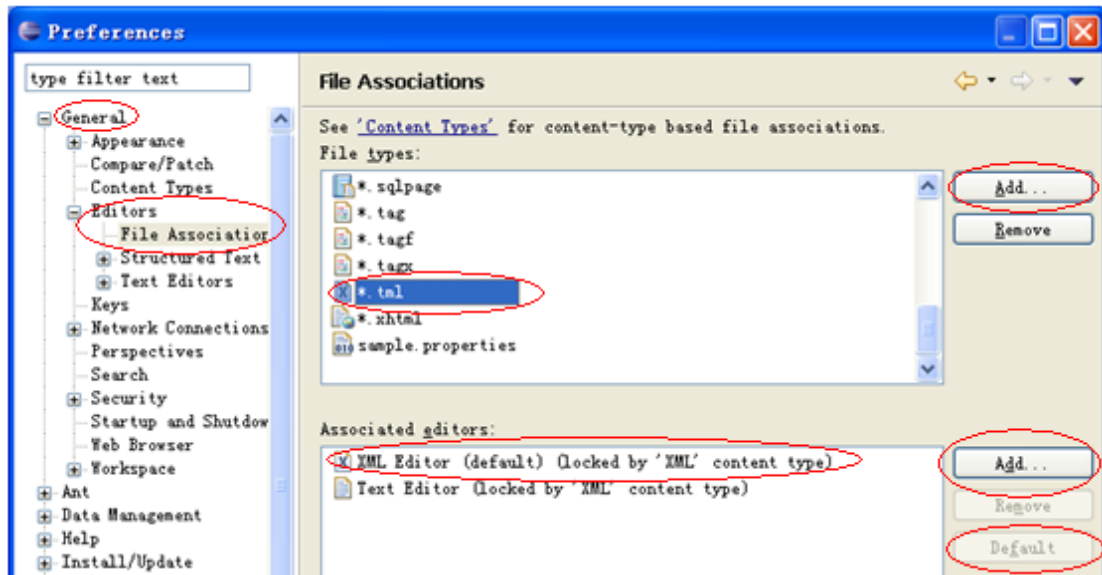
为了方便编辑 tml 文件，我们应该设置 Eclipse 用 XML 编辑器打开 tml 文件。

点击 Eclipse 菜单【Window】【Preferences】。点击 General 下的 Content Types，在右上列表框中选中 Text 下的 XML，然后在右下列表框中增加 “*.tml”，如图 2-1。



(图 2-1)

然后点击 General 下的 Editors 下的 File Associations，在右上列表框中增加 “*.tml”，选中 “*.tml” 然后在右下列表框增加 “XML Editor” 并设为默认编辑器，如图 2-2。



(图 2-2)

按 OK 按钮退出，以后 Eclipse 就会用 XML 编辑器打开 tml 文件了。

Tapestry 按如下顺序查找页面模板：

- 首先在页面类所在目录查找。

如果我们使用 Maven 的话，那么在一个典型的 Maven 目录结构中，Start 页面类会是：
src/main/java/example/hellotapestry/pages/Start.java，
而页面模板会是：

src/main/resources/example/hellotapestry/pages/Start.tml，

它们的源文件不在同一目录，但是在编译后，Start.class 和 Start.tml 会在同一目录中。

本书中我们没有使用 Maven，只需简单的把 Start.java 和 Start.tml 文件都放在 src/example/hellotapestry/pages 目录下，编译时，Start.tml 会被拷贝到 Start.class 所在目录。

- 如果页面类所在目录找不到相应的页面模板，则在 WebContent 目录下查找。

比如页面逻辑名为“Start”，则查找 WebContent/Start.tml，如果页面逻辑名“address/New”，则查找 WebContent/address/New.tml。

Expansion

在 Start.tml 中，有代码片段如下：

```
${prop:greeting}
```

类似这样的代码片段在 Tapestry 中被称作 Expansion。Expansion 以“\${”开头，“}”结尾，中间是一个绑定表达式（Binding Expression）。

Expansion 也可以被嵌入在元素属性中，例如：

```

```

绑定表达式

“prop:greeting”是一个绑定表达式（Binding Expression），其中“prop”是其前缀。除了“prop”，还有很多其它前缀，各前缀及其代表的意义如下：

asset	Asset 的路径。
block	Block 的 ID。
component	组件的 ID。
context	Context asset 的路径。
literal	文本字符串。
nullfieldstrategy	用于设置预定义 NullFieldStrategy。
message	用于获取本地化信息。
prop	属性名。
translate	Translator 名称。
validate	Validator 名称。
var	组件变量。

这些前缀的用法和例子，在后面的章节中陆陆续续会出现，而我们现在只需先了解“p

“prop”前缀的用法，这也是最常用的一个前缀。

“prop”表示类中的一个属性。

在一个典型的 `JavaBean` 中，一个名为 “greeting” 的属性的代码如下：

```
private String greeting;

public String getGreeting() {
    return greeting;
}

public void setGreeting(String greeting) {
    this.greeting = greeting;
}
```

其中包括私有的属性和公开的 `getter` 和 `setter` 方法。`getter` 方法的命名规则是把属性名第一个字母大写然后在前面加上 “get”（如果是布尔型，也可以加 “is”），`setter` 方法的命名规则是把属性名第一个字母大写然后在前面加上 “set”。如果属性只读则省略 `setter` 方法。

在使用 “prop” 时，并不要求类中真的存在一个叫做 “greeting” 的成员变量，只需要有相应的 `getter/setter` 方法就行了，正如我们在 `Start` 类中看到的一样。同时 “prop” 是大小写不敏感的，因此即使我们把页面模板中的表达式写成 “prop:gReetinG” 而且把页面类中的 `getter` 方法名写成 “getgreEtiNg” 也是可以的。

我们还可以获取属性的属性，例如：

```
prop:user.firstName
```

我们还可以在表达式中直接使用方法名而不是属性名，例如：

```
prop:getGreeting()
prop:getUser().getFirstName()
```

直接使用方法名的话，必须在方法名后加括号，同时方法必须公开、无参数且返回一个 `non-void` 值。使用方法名最大的好处是可以方便我们获取那些非 `getter` 方法的返回值，比如 `Collection.size()`。

我们也可以混合使用属性名和方法名，例如：

```
prop:user.getFirstName()
```

如果要避免空指针引发异常，可以用 “?.” 代替 “.”，例如：

```
prop:foo?.bar?.baz
```

这样，不管 `foo` 或者 `bar` 是 `null`，整个表达式都会返回 `null`，对其赋值则不会引发任何操作。

在 `Expansion` 中，“prop” 是缺省前缀，因此我们可以省略 `${prop:greeting}` 中的 “prop” 前缀，把 `Start.tml` 修改成：

```
<html xmlns:t="http://tapestry.apache.org/schema/tapestry_5_1_0.xsd">
    ${greeting}
</html>
```


使用组件

现在，我们要在 **Start** 页面增加一个超链接，指向一个叫做 “Another” 的页面。

修改 **Start.tml**：

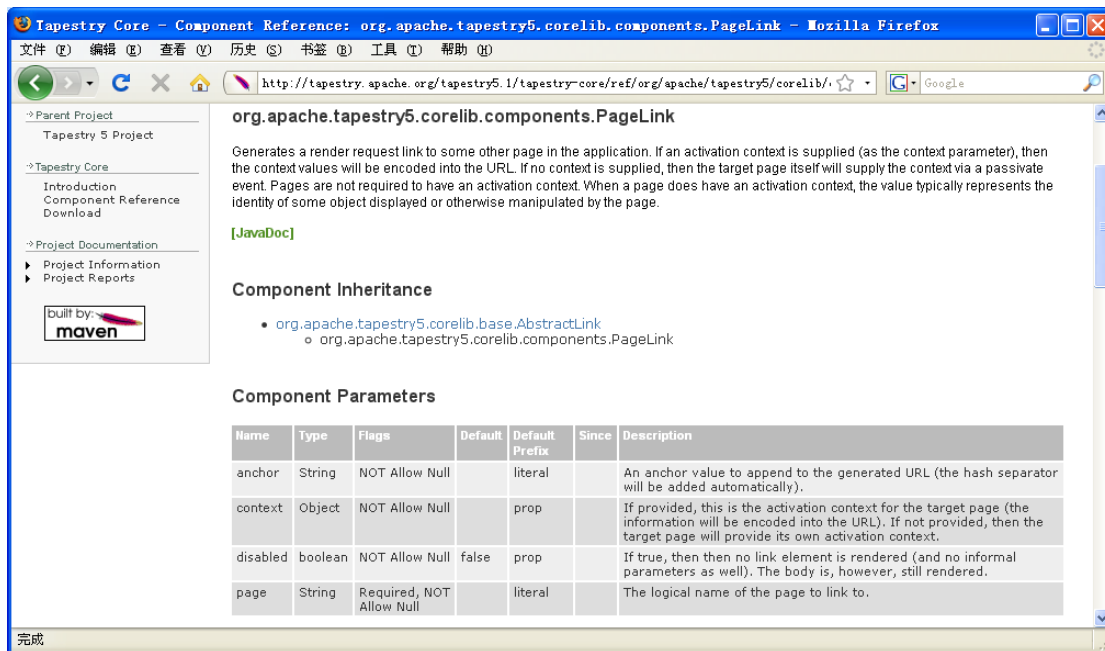
```
<html xmlns:t="http://tapestry.apache.org/schema/tapestry_5_1_0.xsd">
  ${greeting}
  <p>
    <t:pagelink page="literal:Another">Another</t:pagelink>
  </p>
</html>
```

这里，我们使用了一个 **PageLink** 组件。**PageLink** 组件有一个 “page” 参数，可能你已经意识到，它也是一个绑定表达式，其前缀是 “literal”。

在定义组件时要使用 “t:” 前缀，这是由根元素的命名空间声明决定的。

PageLink 是 Tapestry 内置的组件。<http://tapestry.apache.org/tapestry5.1/tapestry-core/ref/>上列出了所有的 Tapestry 内置组件。

打开 **PageLink** 组件的参考页面（<http://tapestry.apache.org/tapestry5.1/tapestry-core/ref/org/apache/tapestry5/corelib/components/PageLink.html>），如图 2-3，里面有 **PageLink** 组件的详细说明及例子。



(图 2-3)

其中有一个表格列出了 **PageLink** 组件的所有参数，“anchor”，“context”，“disabled” 和 “page”。其中 “page” 参数是 **String** 类型的，而且是必需的，不允许 **Null**，没有缺省值，“page” 参数的缺省前缀是 “literal”，由此我们也可以把前面例子中 “page” 参数的 “literal” 前缀省略：

```
<t:pagelink page="Another">Another</t:pagelink>
```

这些参数都是在组件中明确定义了的，称为正式参数（formal parameter）。有时，我们还要用到非正式参数（informal parameter），比如我们要把 Start 页面的超链接变成红色，可以使用下面代码：

```
<t:pagelink page="Another" style="color: red;">Another</t:pagelink>
```

这里的“style”就是非正式参数，Tapestry 把非正式参数直接输出为 Html 标签的属性。非正式参数的缺省前缀是“literal”。

有些组件是不支持支持非正式参数的。

异常报告

重启服务器，打开 <http://localhost:8080/HelloTapestry/>，页面上显示的是异常信息，告诉我们“Another”不是一个已知页面名，并且还列出了目前可用的页面名，如图 2-4。



（图 2-4）

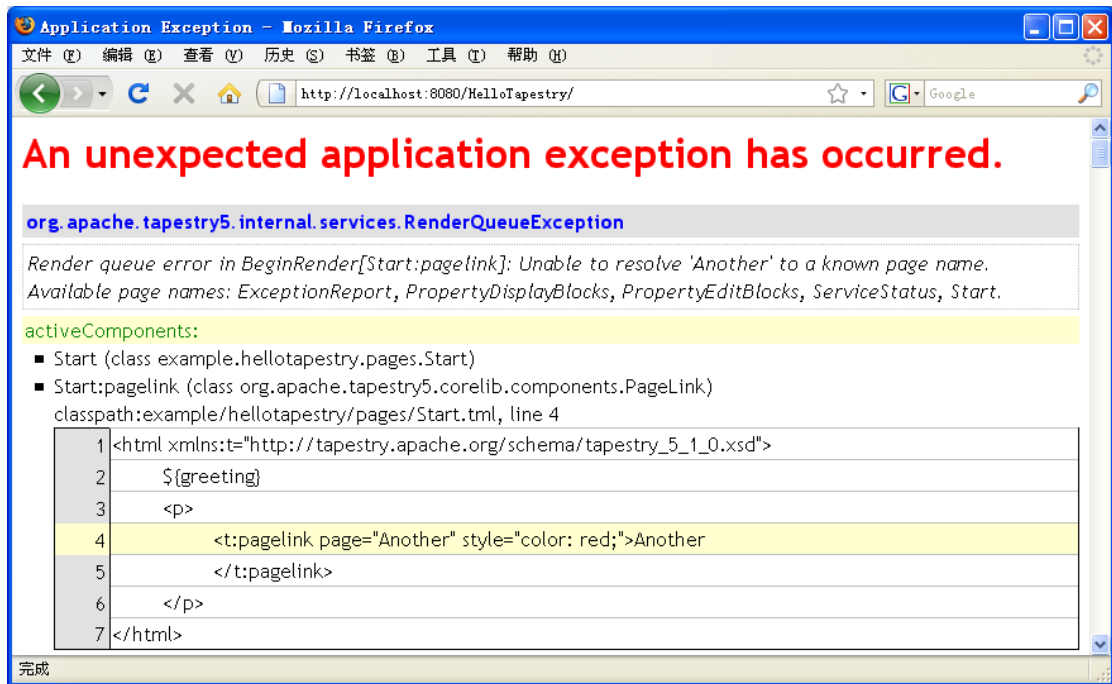
这些异常信息相当的简单，如果我们想看详细一点的信息，我们可以对 Tapestry 进行配置。

在“example.hellotapestry.services”包中新建一个类 AppModule。类名不能随便取，它是由 web.xml 中的 Filter 名称“app”，把第一个字母大写然后加上“Module”组成的。而且 AppModule 类必须放在“example.hellotapestry.services”包中，其中“example.hellotapestry”是 web.xml 中的 tapestry.app-package 参数。

AppModule 类的代码如下：

```
public class AppModule {
    public static void contributeApplicationDefaults(
        MappedConfiguration<String, String> configuration) {
        configuration.add(SymbolConstants.PRODUCTION_MODE, "false");
    }
}
```

重启服务器，刷新 Start 页面，我们可以看到 Start 页面非常详细的信息，Tapestry 甚至告诉我们错误发生在 Start.tml 的第 4 行，如图 2-5。



(图 2-5)

关于 Tapestry 的配置，请参考 <http://tapestry.apache.org/tapestry5.1/guide/conf.html>。

类和模板的重新载入

现在，我们来创建 Another 页面。

在 “example.hellotapestry.pages” 包中新建 Another 类：

```
public class Another {
}
```

在 “example.hellotapestry.pages” 包中新建 Another.tml：

```
<html xmlns:t="http://tapestry.apache.org/schema/tapestry_5_1_0.xsd">
  <t:pagelink page="Start">Start</t:pagelink>
</html>
```

如果我们现在刷新 Start 页面，页面仍然会显示异常。但是如果我们修改一下 Start 类或者 Start 模板，比如在空白处增加一个空格并保存（如果你不想要这个空格还可以再改回去），然后再刷新 Start 页面，Start 页面就能正确显示了。

当我们修改了 Start 页面类或者 Start 页面模板，Tapestry 会自动重新载入 Start 页面类或者 Start 页面模板，生成新的 Start 页面实例，因此，我们无需重启 Tamcat 服务器，就能看到新的页面了。如果刚才我们对 Start 页面类和页面模板都不做修改，那么 Tapestry 就不会重新生成 Start 页面实例，我们就只能重启 Tamcat 服务器才能看到新的 Start 页面。

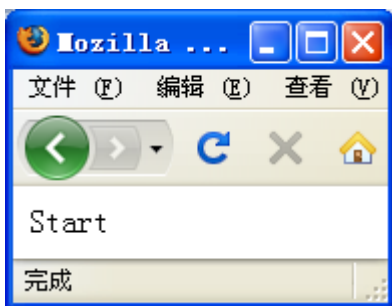
如果我们修改的不是一个页面，而是一个组件，那么所有使用这个组件的页面都会重新生成页面实例。

如果我们把这些文件打包进一个 jar 文件，那么即使这些文件被修改了，Tapestry 也不

会重新载入这些文件。对于开发来说，当然不存在任何问题，我们的文件都是实际存在于文件系统中的。如果打包成 jar 文件部署到 Servlet 容器，则要看 Servlet 容器的具体情况确定是否要重启服务器才能使修改生效。

定义组件的其它方法

现在，我们已经知道了定义组件的一种方法，这种方法使用了一个 XML 元素来定义组件。由于不是标准的 HTML 标签，普通的网页设计工具或者浏览器是不能正确预览组件的，我们也很难单独把页面模板交给专门的网页设计人员来处理。比如我们用浏览器直接打开 Another.html 文件，页面效果如图 2-6，其中的超链接没能正确显示。



(图 2-6)

为了解决这个问题，Tapestry 提供另一种定义组件的方法。这种方法使用普通的 HTML 标签，把组件类型作为标签的 type 属性。

使用这种方法定义组件，Another.html 的代码如下：

```
<html xmlns:t="http://tapestry.apache.org/schema/tapestry_5_1_0.xsd">
  <a href="#" t:type="pagelink" page="Start">Start</a>
</html>
```

现在用浏览器直接打开 Another.html 文件，超链接能正确显示了，如图 2-7。



(图 2-7)

虽然已经使用标准 HTML 标签来定义组件，但是在 HTML 标签中出现的 type 属性以及 page 属性对非程序人员来说还是很难理解。因此 Tapestry 提供更彻底的方法，把组件定义在页面类中。

修改 Another 类：

```
public class Another {
```

```
@Component(parameters = { "page=start" })
private PageLink goToStart;
}
```

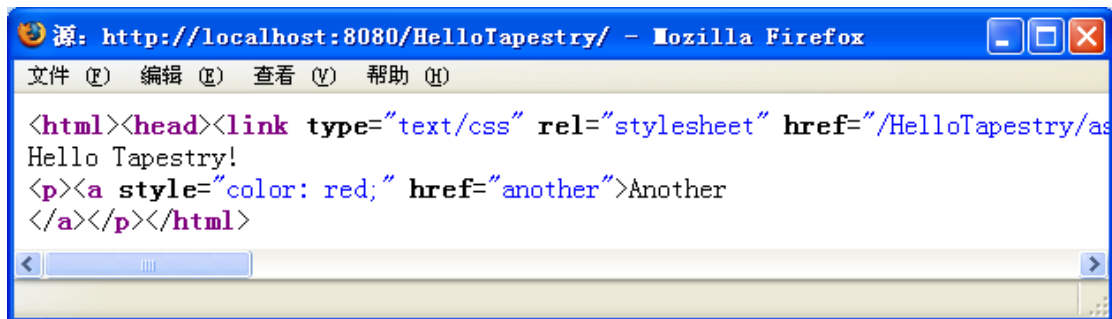
修改 Another.tml:

```
<html xmlns:t="http://tapestry.apache.org/schema/tapestry_5_1_0.xsd">
  <a href="#" t:id="goToStart">Start</a>
</html>
```

现在, Html 标签中增加的唯一属性是 id。

Html 代码

在浏览器中查看 Start 页面的 Html 源代码,如图 2-8。Tapestry 对 Html 代码进行了优化,标签之间的空格和换行符都被去除,而且在文本中连续的多个空格会被压缩成一个空格。



(图 2-8)

如果想在 Html 代码中保留某些空格和换行符,可以使用 xml:space 属性。

修改 Start.tml:

```
<html xmlns:t="http://tapestry.apache.org/schema/tapestry_5_1_0.xsd">
  ${greeting}
  <p xml:space="preserve">
    <t:pagelink page="Another" style="color: red;">Another
  </t:pagelink>
  </p>
</html>
```

刷新 Start 页面, Start 页面的 Html 源代码如图 2-9。



(图 2-9)

组件事件请求

我们往 Start 页面再增加一个超链接，同样指向 Another 页面，但是这次我们使用 `ActionLink` 组件而不是 `PageLink` 组件。

修改 Start.tml:

```
<html xmlns:t="http://tapestry.apache.org/schema/tapestry_5_1_0.xsd">
  ${greeting}
  <p xml:space="preserve">
    <t:pagelink page="Another" style="color: red;">Another
  </t:pagelink>
  </p>
  <p>
    <t:actionlink t:id="actionLink1">Another (ActionLink)
  </t:actionlink>
  </p>
</html>
```

修改 Start 类:

```
public class Start {
    @InjectPage
    private Another anotherPage;

    public String getGreeting() {
        return "Hello Tapestry!";
    }

    @OnEvent(value = "action", component = "actionLink1")
    Object toAnotherPage() {
        return anotherPage;
    }
}
```

运行程序，在 Start 页面点击两个超链接，都能跳转的 Another 页面。但是这两个超链接是不一样的，从 Html 源代码可以看到它们的 URL 分别是 <http://localhost:8080/e1/another> 和 <http://localhost:8080/e1/start.actionlink1>。

第一个 URL 表示一个页面（逻辑名为“another”），这个很容易理解。

第二个 URL 表示一个页面（逻辑名为“start”）中的一个组件（ID 为“actionlink1”）的一个事件（缺省为“action”），对这种类型 URL 的请求被称为组件事件请求（Component Event Request）。当 Tapestry 收到此请求，会触发 Start 页面类中的 `toAnotherPage()` 方法，根据此方法的返回结果，让客户端重定向到 Another 页面，最后在浏览器地址栏上看到的 URL 也是 <http://localhost:8080/e1/another>。显然，这种方式客户端实际上要发送两次请求。我们

可通过配置 Tapestry 以禁止重定向，那样就只需要一次请求，但是最后浏览器地址栏上的 URL 将会是 <http://localhost:8080/e1/start.actionlink1>。到底是否需要重定向，需要综合考虑性能和 URL 的友好性来决定。

Start 页面类中的 `toAnotherPage()` 方法被称为 Event Handler Method。Event Handler Method 由 `@OnEvent` 注释来标识，`@OnEvent` 注释有两个属性，`value` 表示事件类型，缺省为“action”，`component` 表示组件 ID，缺省表示任意组件，这两个属性都是大小写不敏感的。

一个组件事件可能会同时触发几个方法，这些方法执行顺序如下：

- 父类方法在子类方法之前执行。
- 同一个类中方法按字母顺序执行。
- 方法名相同，参数个数多的先执行。
- 如果子类方法覆盖了父类方法，那么只执行子类方法，但在确定执行顺序时还是把它看作父类方法。

如果不使用 `@OnEvent` 注释，可以使用特定的方法名来代替。方法命名原则是以“on”开头，后跟事件类型，如果还要指定组件，则接着跟一个“From”和组件 ID。如：

```
@OnEvent(value = "action", component = "actionLink1")
Object toAnotherPage() {
    return anotherPage;
}
```

可以写成：

```
Object onActionFromActionLink1() {
    return anotherPage;
}
```

而

```
@OnEvent(value = "action")
Object toAnotherPage() {
    return anotherPage;
}
```

可以写成：

```
Object onAction() {
    return anotherPage;
}
```

事件方法的返回值类型有以下几种：

- **Null**
方法无返回值（void）或者返回 null。把当前页返回给客户端。
- **String**
页面逻辑名（如“Another”，大小写不敏感。把逻辑名所对应的页面返回给客户端。
- **Class**
页面类（如 Another.class）。把页面类所对应的页面返回给客户端。
- **Page**

页面类实例。使用@InjectPage 注释，参考前面例子。

- Link

org.apache.tapestry5.Link 接口的实现。能够转化为 URL，把 URL 所对应的页面返回给客户端。

- Stream

org.apache.tapestry5.StreamResponse 接口的实现。用于直接向客户端输出流，这在向客户端提供 PDF 或者图片时很有用。

- URL

把 URL 所对应的页面返回给客户端。

提交表单也会产生组件事件请求。

修改 Start.tml:

```
<html xmlns:t="http://tapestry.apache.org/schema/tapestry_5_1_0.xsd">
  ${greeting}
  <p xml:space="preserve">
    <t:pagelink page="Another" style="color: red;">Another
    </t:pagelink>
  </p>
  <p>
    <t:actionlink t:id="actionLink1">Another (ActionLink)
    </t:actionlink>
  </p>
  <t:form t:id="form1">
    <input type="submit" value="Submit" />
  </t:form>
</html>
```

修改 Start 类:

```
public class Start {
    @InjectPage
    private Another anotherPage;

    public String getGreeting() {
        return "Hello Tapestry!";
    }

    @OnEvent(value = "action", component = "actionLink1")
    Object toAnotherPage() {
        return anotherPage;
    }

    Object onSuccessFromForm1() {
        return anotherPage;
    }
}
```



```
}
}
```

运行程序，点击 **Submit** 按钮，也能跳转到 **Another** 页面。

页面间的数据传递

在 **Web** 应用中，我们经常需要在页面之间传递数据。

我们在 **Start** 页面中增加一个文本框，提交表单，则把文本框中的内容显示在 **Another** 页面。

修改 **Start.html**：

```
<html xmlns:t="http://tapestry.apache.org/schema/tapestry_5_1_0.xsd">
  ${greeting}
  <p xml:space="preserve">
    <t:pagelink page="Another" style="color: red;">Another
  </t:pagelink>
  </p>
  <p>
    <t:actionlink t:id="actionLink1">Another (ActionLink)
  </t:actionlink>
  </p>
  <t:form t:id="form1">
    <t:textfield value="theValue" />
    <input type="submit" value="Submit" />
  </t:form>
</html>
```

修改 **Start** 类：

```
public class Start {
    private String theValue = "Init Value";
    @InjectPage
    private Another anotherPage;

    public String getTheValue() {
        System.out.println("getTheValue : " + theValue + "");
        return theValue;
    }

    public void setTheValue(String theValue) {
        System.out.println("setTheValue : " + theValue + "");
        this.theValue = theValue;
    }

    public String getGreeting() {
```

```

        return "Hello Tapestry!";
    }

    @OnEvent(value = "action", component = "actionLink1")
    Object toAnotherPage() {
        return anotherPage;
    }

    Object onSuccessFromForm1() {
        System.out.println("onSuccessFromForm1");
        return anotherPage;
    }
}

```

在 `Start.html` 中我们增加了一个 `TextField` 组件，其属性 `value` 对应 `Start` 页面类中的 `theValue` 属性，`theValue` 是典型的 `JavaBean` 属性。为了更清楚了解组件的运作，我们在 `getTheValue()`、`setTheValue()` 以及 `onSuccessFromForm1()` 方法中额外加了一条打印语句。

打开 `Start` 页面，可在控制台看到如下输出：

```
getTheValue : 'Init Value'
```

此时 `Tapestry` 通过 `getTheValue()` 方法获取 `theValue` 的值，并把它作为文本框的初始值。

把文本框内容改为 “`New Value`”，提交表单，将在控制台看到如下输出：

```
setTheValue : 'New Value'
```

```
onSuccessFromForm1
```

此时 `Tapestry` 通过 `setTheValue()` 方法设置 `theValue` 属性，然后提交表单。

还有一种更简单的定义属性的方式，就是使用 `@Property` 注释。

修改 `Start` 类：

```

public class Start {
    @Property
    private String theValue = "Init Value";
    @InjectPage
    private Another anotherPage;

    // public String getTheValue() {
    //     System.out.println("getTheValue : " + theValue + "");
    //     return theValue;
    // }
    //
    // public void setTheValue(String theValue) {
    //     System.out.println("setTheValue : " + theValue + "");
    //     this.theValue = theValue;
    // }
}

```

```

public String getGreeting() {
    return "Hello Tapestry!";
}

@OnEvent(value = "action", component = "actionLink1")
Object toAnotherPage() {
    return anotherPage;
}

Object onSuccessFromForm1() {
    // System.out.println("onSuccessFromForm1");
    return anotherPage;
}
}

```

`@Property` 注释有两个参数，`read` 和 `write`，`read` 设置属性可读，`write` 设置属性可写，缺省都是 `true`。

刷新 **Start** 页面，一切正常。

为了把数据传递给 **Another** 页面，我们在 **Another** 页面类中增加一个 `setValue()` 方法，然后在 **Start** 页面类的 `onSuccessFromForm1()` 方法中调用此方法。

修改 **Another.html**：

```

<html xmlns:t="http://tapestry.apache.org/schema/tapestry_5_1_0.xsd">
  <a href="#" t:id="goToStart">Start</a>
  <br />
  The value is: ${value}
</html>

```

修改 **Another** 类：

```

public class Another {
    private String value;
    @Component(parameters = { "page=start" })
    private PageLink goToStart;

    public String getValue() {
        return value;
    }

    public void setValue(String value) {
        this.value = value;
    }
}

```

修改 **Start** 类：

```

public class Start {
    @Property

```

```

private String theValue = "Init Value";
@InjectPage
private Another anotherPage;

public String getGreeting() {
    return "Hello Tapestry!";
}

@OnEvent(value = "action", component = "actionLink1")
Object toAnotherPage() {
    return anotherPage;
}

Object onSuccessFromForm1() {
    anotherPage.setValue(theValue);
    return anotherPage;
}
}

```

然而，当我们提交表单，Another 页面并没有显示 Start 页面传过来的数据，如图 2-10。



(图 2-10)

这是为什么呢？

要知道造成这个问题的原因，需要了解 Tapestry 的页面池。

页面池

当 Tapestry 收到一个页面请求时，Tapestry 需要一个页面实例来生成 Html 代码。页面实例可看成是页面类和页面模板的混合体，用于生成最终的 Html 代码。

假设在一段时间内有许多用户请求 A 页面。如果对每个请求，都为其新建一个 A 页面实例并在使用后销毁，那么新建和销毁页面实例将是不小的开销。为了提高性能，Tapestry 使用页面池。当一个用户请求 A 页面，Tapestry 将从页面池中取出一个 A 页面实例并在使用后放回页面池供以后的请求使用，这样就减少了新建和销毁页面实例的开销。

当用户往页面中输入了某些数据，这些数据将保存在页面实例中，如果不加处理就把此页面实例放回页面池，那么这些数据将会被暴露给下一个使用此页面实例的用户。为了避免

出现这种情况，在一个页面实例被放回页面池之前 Tapestry 会对其进行清理，把其中的变量重置为初始值。

回过头来看前面的问题，当我们在 Start 页面中输入数据并提交表单，会依次发生如下事情：

第一步，客户端向 Tapestry 发出一个组件事件请求，触发 Start 页面实例的 `onSuccessFromForm1()` 方法。

第二步，Tapestry 从页面池取出一个 Another 页面实例并设置其 `value` 属性。

第三步，Tapestry 通知客户端重定向到此 Another 页面。

第四步，客户端请求 Another 页面。

第五步，Tapestry 从页面池取出一个 Another 页面实例用来生成 Html 代码发送给客户端。

很显然，在整个过程中，客户端向 Tapestry 发送了两次请求。对每次请求，Tapestry 都是从页面池中新取出一个 Another 页面实例来处理，请求处理完成后，Another 页面实例又会被放回页面池，保存在其中的数据也就被清除了。因此即使第五步中所用 Another 页面实例就是第二步中所用的那一个（也有可能不是同一个），其中的数据也已经被清除过了。

试想如果在第二步设置完 Another 页面实例的 `value` 属性后，就直接用此页面实例生成 Html 代码发送给客户端，而不需要经过重定向，那么前面数据丢失的问题就不存在了。

我们可以配置 Tapestry 以禁止重定向。

禁止重定向

修改 AppModule 类：

```
public class AppModule {
    public static void contributeApplicationDefaults(
        MappedConfiguration<String, String> configuration) {
        configuration.add(SymbolConstants.PRODUCTION_MODE, "false");
        configuration.add(
            SymbolConstants.SUPPRESS_REDIRECT_FROM_ACTION_REQUESTS,
            "true");
    }
}
```

刷新页面，现在 Start 页面输入的数据就能正确的被传递到 Another 页面了，如图 2-11。



(图 2-11)

禁止重定向可以让客户端少发一次请求，但是此时客户端浏览器显示的 URL 是 <http://localhost:8080/HelloTapestry/start.form1>，这个 URL 不是很友好。如果用户把这个 URL 加为书签，那么当我们修改了 Form 组件的 ID，那么这个 URL 也就失效了。

那么，在有重定向的情况下如何传递数据呢？

有两种方法。

@Persist

使用@Persist 注释是比较简单的一种方法。

修改 Another 类：

```
public class Another {
    @Persist
    private String value;
    @Component(parameters = { "page=start" })
    private PageLink goToStart;

    public String getValue() {
        return value;
    }

    public void setValue(String value) {
        this.value = value;
    }
}
```

把 AppModule 类改回去：

```
public class AppModule {
    public static void contributeApplicationDefaults(
        MappedConfiguration<String, String> configuration) {
        configuration.add(SymbolConstants.PRODUCTION_MODE, "false");
        // configuration.add(
        // SymbolConstants.SUPPRESS_REDIRECT_FROM_ACTION_REQUESTS, "true");
    }
}
```

运行程序，Another 页面能显示正确的结果。

这种方法的原理很简单，当 Tapestry 设置 Another 页面类的 value 属性时，如果看到“Persist”，就将其保存在 Session 中，以后再收到 Another 页面请求时，则从 Session 中取出这个值来作为 value 的值。

这种方法虽然很简单，但是缺点也很明显。由于使用 session 来保存数据，增加了服务器的负担，同时其 URL 也无法被做为书签。

Activation Context

另一种方法是使用 Activation Context，把数据附加在 URL 后面。比如要把“xxx”传给 Another 页面，可以让客户端重定向到 <http://localhost:8080/HelloTapestry/another/xxx>。

修改 Another 类：

```
public class Another {
    @Persist
    private String value;
    @Component(parameters = { "page=start" })
    private PageLink goToStart;

    public String getValue() {
        return value;
    }

    public void setValue(String value) {
        this.value = value;
    }

    String onPassivate() {
        return value;
    }
}
```

运行程序，在 Start 页面输入“xxx”并提交，查看浏览器中的地址栏，URL 已经变成了 <http://localhost:8080/HelloTapestry/another/xxx>。

当我们在 Start 页面中输入“xxx”并提交表单，Tapestry 把 Another 页面实例的 value 属性设置为“xxx”，然后查看 Another 页面是否有 Activation Context，这会触发 Another 页面的 passivate 事件，onPassivate()方法的返回值被作为页面的 Activation Context，附加在 URL 后面，最后让客户端重定向到此包含 Activation Context 的 URL。

现在数据已经附加在 URL 上，但是@Persist 注释仍然还在，Another 页面实际上仍然是从 session 获取数据而不是从 URL。我们可以通过 Activate 事件获取 URL 中的数据。

修改 Another.类：

```
public class Another {
    // @Persist
    private String value;
    @Component(parameters = { "page=start" })
    private PageLink goToStart;

    public String getValue() {
        return value;
    }
}
```

```

public void setValue(String value) {
    this.value = value;
}

String onPassivate() {
    return value;
}

void onActivate(String value) {
    this.value = value;
}
}

```

现在，`@Persist` 注释已经被去掉，程序仍然可以正常运行。

`Activate` 事件发生在生成 `Html` 代码之前，我们可以在其中做两件事：

- 获取 `URL` 中的附加数据。
- 初始化（比如根据 `Id` 获取内容）或者校验（比如检查用户是否登录）。

`Activate` 和 `passivate` 是互相对应的。如果有多个 `context`，则按顺序出现在 `onActivate()` 方法的参数中，而 `onPassivate()` 则需返回 `Object[]`、`List` 或者 `EventContext`，如：

```

Object[] onPassivate() {
    return new Object[] { "p01", 123 };
}

void onActivate(String cat, int id) {
    this.cat = cat;
    this.id = id;
}

```

SSO

有些数据可能被很多页面共享，比如登录状态，这时候就要用 `SSO`（`Session State Object`）。`SSO` 被保存在 `Session` 中，可以被同一用户的所有页面共享，但不会被其他用户共享。

下面我们来做一个登录程序，共两个页面。在 `Login` 页面输入用户名和密码，如果通过校验，则跳转到 `Welcome` 页面，`Welcome` 页面显示欢迎信息。

新建 `Login.tml`：

```

<html xmlns:t="http://tapestry.apache.org/schema/tapestry_5_1_0.xsd">
  <t:form t:id="loginForm">
    <div>
      用户名:
      <t:textfield value="userName" />
    </div>
    <div>

```



```

        密**码:
        <t:passwordField value="password" />
    </div>
    <t:submit value="登录" />
</t:form>
</html>

```

新建 Login 类:

```

public class Login {
    @Property
    private String userName;
    @Property
    private String password;
    @SessionState
    private String user;

    Class<?> onSuccess() {
        if ("abc".equals(userName) && "123".equals(password)) {
            user = userName;
            return Welcome.class;
        }
        return null;
    }
}

```

Tapestry 5.1 用 `@SessionState` 注释来标识 SSO。SSO 的使用非常简单，在第一次访问的时候会被自动创建，给 SSO 赋值即可保存这个值。

在 Login 页面，当输入的用户名是“abc”而且密码是“123”，Tapestry 会自动创建一个字符串类型的 SSO，并把用户名保存在其中。然后就可以在 Welcome 页面显示这个用户名了。

新建 Welcome.tml:

```

<html xmlns:t="http://tapestry.apache.org/schema/tapestry_5_1_0.xsd">
    欢迎${theUser}!
</html>

```

新建 Welcome 类:

```

public class Welcome {
    @SessionState
    private String theUser;

    public String getTheUser() {
        return theUser;
    }
}

```

在 `Welcome` 页面类中我们仅仅只需定义一个 `String` 类型的 `SSO`，提供 `getter` 方法是为了页面模板能访问这个 `SSO`。

我们在 `Start` 和 `Welcome` 页面类中定义 `SSO` 时使用了两个不同的变量名称，“`user`”和“`theUser`”，但它们是同一个 `SSO`，因为 `SSO` 只跟类型有关，而不管我们如何命名。如果我们要把两个同一类型的数据保存到 `SSO` 中，可以建一个包含这两个数据的类型。

重启服务器，`Login` 和 `Welcome` 页面效果分别如图 2-12 和 2-13。



(图 2-12)



(图 2-13)

在 `Welcome` 页面，我们要增加一个退出超链接，退出则删除 `SSO`。

修改 `Welcome.html`:

```
<html xmlns:t="http://tapestry.apache.org/schema/tapestry_5_1_0.xsd">
  欢迎${theUser}!
  <t:actionlink t:id="logout">退出</t:actionlink>
</html>
```

修改 `Welcome` 类:

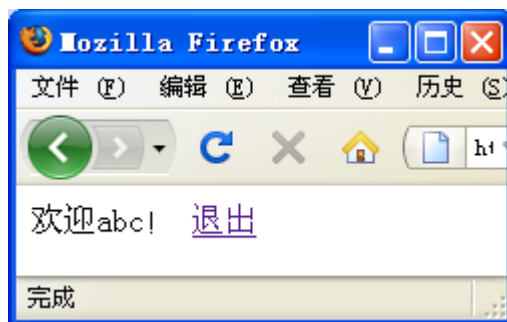
```
public class Welcome {
    @SessionState
    private String theUser;

    public String getTheUser() {
        return theUser;
    }

    void onActionFromLogout() {
        theUser = null;
    }
}
```

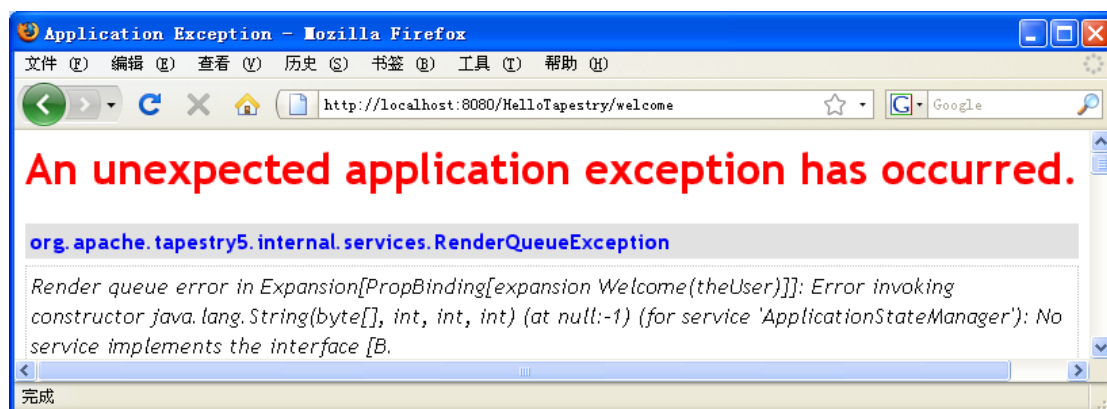
删除 `SSO` 只需把它赋值为 `null` 即可。

`Welcome` 页面效果如图 2-14。



(图 2-14)

在 Welcome 页面点击退出，页面出现异常，如图 2-15。



(图 2-15)

这是因为 SSO 已经被删除了，页面模板试图从一个不存在的 SSO 中获取用户名。因此，我们修改一下 Welcome 页面，如果 SSO 已经创建，说明用户已经登录，就显示用户名，否则显示登录超链接。

那么如何判断 SSO 是否已经创建呢？

不能简单的使用 “if (theUser == null)” 来判断，执行这样的判断语句也会导致 SSO 被自动创建并保存到 Session 中，而在真正用到之前，应该尽量避免创建 Session。

Tapestry 为我们提供了一种方法，能够判断 SSO 是否已经创建而又不会导致 SSO 被自动创建。就 Welcome 页面来说，我们在页面类里定义了一个 SSO 名为 “theUser”，那么我们只需再定义一个私有布尔属性，其名为 “theUser” 加 “Exists” 后缀。

修改 Welcome 类：

```
public class Welcome {
    @SessionState
    private String theUser;
    private boolean theUserExists;

    public String getTheUser() {
        return theUser;
    }

    public boolean isTheUserExists() {
```

```

        return theUserExists;
    }

    void onActionFromLogout() {
        theUser = null;
    }
}

```

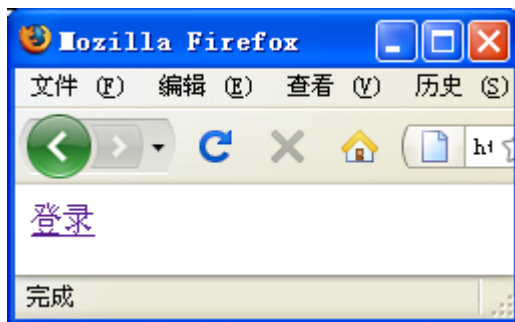
修改 Welcome.tml:

```

<html xmlns:t="http://tapestry.apache.org/schema/tapestry_5_1_0.xsd"
      xmlns:p="tapestry:parameter">
  <t:if test="theUserExists">
    欢迎${theUser}!
    <t:actionlink t:id="logout">退出</t:actionlink>
  <p:else>
    <t:pagelink page="Login">登录</t:pagelink>
  </p:else>
</t:if>
</html>

```

现在，退出后，Welcome 页面效果如图 2-16。



(图 2-16)

一般情况下，创建 SSO 需要公开无参数的构造方法，但是我们可以在 AppModule 中进行配置以使用带参数的构造方法。

我们还可以使用接口来表示 SSO，而不是具体实现类。

假设 MyState 实现接口 IState，我们可以在 AppModule 中配置 SSO 如下：

```

public void contributeApplicationStateManager(
    MappedConfiguration<Class<?>, ApplicationStateContribution> configuration) {
    ApplicationStateCreator<IState> creator = new ApplicationStateCreator<IState>() {
        public IState create() {
            return new MyState();
        }
    };
}

```

```
configuration.add(IState.class, new ApplicationStateContribution("session", creator));  
}
```

然后我们就可以在页面（或者组件）类中这样使用 `IState` 类型的 `SSO` 了：

```
@SessionState  
private IState iState;
```

如果以后想用 `IState` 的另一个实现类代替 `MyState` 类，只需修改配置即可。

子目录

在一个 `Web` 应用中可能会有很多页面，把它们全部放在根目录下显然不是一个好的做法，因此需要用子目录来组织它们。

假设现在有一个关于产品的模块，我们把这个模块的相关页面都放在 `product` 目录下，页面类和页面逻辑名之间关系如下：

页面类	页面逻辑名
<code>example.hellotapestry.pages.product.ShowAllProducts</code>	<code>product/ShowAllProducts</code>
<code>example.hellotapestry.pages.product.ProductDetail</code>	<code>product/Detail</code>
<code>example.hellotapestry.pages.product.NewProduct</code>	<code>product/New</code>

`Tapestry` 对页面逻辑名进行了优化，当包名（“`product`”）是类名的前缀/后缀时，前缀/后缀将被去除。

第三章 创建自己的组件

上一章我们在页面中使用了 Tapestry 内置的组件，如 PageLink、ActionLink、Form 等，这一章我们将要创建自己的组件。

仿照第一章的例子新建一个动态 Web 工程 “MyComponent”，加入需要的 jar 文件，修改 web.xml 成如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
    <display-name>My Component</display-name>
    <context-param>
        <param-name>tapestry.app-package</param-name>
        <param-value>example.mycomponent</param-value>
    </context-param>
    <filter>
        <filter-name>app</filter-name>
        <filter-class>org.apache.tapestry5.TapestryFilter</filter-class>
    </filter>
    <filter-mapping>
        <filter-name>app</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
</web-app>
```

首先我们要建两个页面，Start 和 Contact。

在 “example.mycomponent.pages” 包中新建以下四个文件：

Start.java:

```
public class Start {
}
```

Start.tml:

```
<html xmlns:t="http://tapestry.apache.org/schema/tapestry_5_1_0.xsd">
    <div>
        <t:pageLink t:page="Start">首页</t:pageLink>
        |
        <t:pageLink t:page="Contact">联系我们</t:pageLink>
    </div>
```

```
<div>首页</div>
</html>
```

Contact.java:

```
public class Contact {
}
```

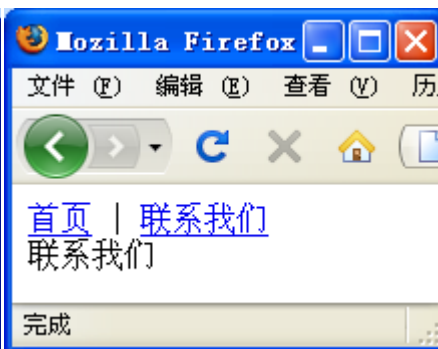
Contact.tml:

```
<html xmlns:t="http://tapestry.apache.org/schema/tapestry_5_1_0.xsd">
  <div>
    <t:pageLink t:page="Start">首页</t:pageLink>
    |
    <t:pageLink t:page="Contact">联系我们</t:pageLink>
  </div>
  <div>联系我们</div>
</html>
```

把“MyComponent”工程加入 Tomcat 服务器，重启服务器，页面效果如图 3-1 和 3-2。



(图 3-1)



(图 3-2)

创建组件

在 Start 和 Contact 页面，最上面的导航菜单是一样的，我们把它做成一个 NaviMenu 组件，这样我们就可以在这两个或者更多的页面上重用 NaviMenu 组件。

在 Tapestry 5 中，组件和页面的差别很小。组件也有组件类和组件模板，类似于前面的页面类和页面模板，同样的，组件里也可以包含其他组件。我们可以把页面看成是一棵组件树的根组件。

组件和页面的差别之一是它们的类所在的包不同。

在“example.mycomponent.components”包中新建 NaviMenu 类：

```
public class NaviMenu {
}
```

在“example.mycomponent.components”包中新建 NaviMenu.tml：

```
<t:container xmlns:t="http://tapestry.apache.org/schema/tapestry_5_1_0.xsd">
  <t:pageLink t:page="Start">首页</t:pageLink>
```

```
|
    <t:pageLink t:page="Contact">联系我们</t:pageLink>
</t:container>
```

注意模板的根元素`<container>`，`<container>`元素本身并不产生任何输出，其作用是为此个 XML 文档提供一个唯一的根元素。

现在，我们就可以在 **Start** 和 **Contact** 页面中使用 **NaviMenu** 组件了。

修改 **Start.tml**:

```
<html xmlns:t="http://tapestry.apache.org/schema/tapestry_5_1_0.xsd">
  <div>
    <t:navimenu />
  </div>
  <div>首页</div>
</html>
```

修改 **Contact.tml**:

```
<html xmlns:t="http://tapestry.apache.org/schema/tapestry_5_1_0.xsd">
  <div>
    <t:navimenu />
  </div>
  <div>联系我们</div>
</html>
```

运行程序，一切正常。

组件参数

现在我们要改进导航菜单，在 **Start** 页面，让导航栏菜单的“首页”只显示文字而不是超链接，同样的在 **Contact** 页面“联系我们”也只显示文字。

我们来创建一个新的组件 **NavItem**，此组件对应导航菜单上的一个导航项，如果其目标页面就是当前页面的话，则显示文字，否则显示超链接。

NavItem 组件有两个参数，“**desc**”表示目标页面描述，“**page**”表示目标页面逻辑名。

Tapestry用`@Parameter`注释标识组件参数，参数名由`@Parameter`注释的`name`属性指定，定义参数的代码如下：

```
@Parameter(name = "desc")
private String pageDesc;

@Parameter(name = "page")
private String pageName;
```

如果不显式指定参数名，那么属性名就是参数名，因此上面的代码可以简写成如下：

```
@Parameter
private String desc;
```



```
@Parameter
private String page;
```

现在，我们来创建 `NavItem` 组件。

在 “`example.mycomponent.components`” 包中新建 `NavItem` 类：

```
public class NavItem {
    @Parameter
    private String desc;

    @Parameter
    private String page;

    @Inject
    private ComponentResources componentResources;

    public boolean isSelf() {
        return componentResources.getPageName().equalsIgnoreCase(page);
    }

    public String getPageDesc() {
        return desc;
    }

    public String getPageName() {
        return page;
    }
}
```

方法 `isSelf()` 用于判断当前页面是否就是目标页面，其中我们使用 `ComponentResources#getPageName()` 方法获取当前页面名称。

在 “`example.mycomponent.components`” 包中新建 `NavItem.tml`：

```
<t:container xmlns:t="http://tapestry.apache.org/schema/tapestry_5_1_0.xsd">
    <t:if test="self" else="block:elseBlock">${pageDesc}</t:if>
    <t:block id="elseBlock">
        <t:pageLink t:page="${pageName}">${pageDesc}</t:pageLink>
    </t:block>
</t:container>
```

这里出现了一个新的 Tapestry 元素 `<block>`。`<block>` 元素所包含的内容不会被直接显示，一般是在组件类中注入（`Inject`）它然后精确控制它的显示，这在本书后面的章节中还会有相关的例子。

If 组件的 `else` 参数表示当 `test` 参数（其绑定表达式为 “`self`”）返回 `false` 时显示的内容。

本例中，else 参数的表达式为“block:elseBlock”（表达式的前缀是“block”），表示 ID 为“elseBlock”的<block>元素。

有了 NavItem 组件，就可以把 NaviMenu.tml 改成：

```
<t:container xmlns:t="http://tapestry.apache.org/schema/tapestry_5_1_0.xsd">
  <t:naviitem desc="literal:首页" page="literal:Start" />
  |
  <t:naviitem desc="literal:联系我们" page="literal:Contact" />
</t:container>
```

运行程序，页面效果如图 3-3 和 3-4。



（图 3-3）



（图 3-4）

在 Tapestry 5.0 中，如果一个<block>元素被作为参数传递给一个组件，我们可以用<parameter>元素来代替它。

但是在 Tapestry 5.1 中，<parameter>元素已过时。Tapestry 5.1 用一种更简明的方式把 block 作为参数传递给组件，那就是使用参数命名空间（Parameter Namespace）。

要使用参数命名空间，首先必须在模板根元素包含参数命名空间声明，通常使用“p:”前缀。

```
xmlns:p="tapestry:parameter"
```

然后使用“p:”前缀和参数名把 block 传递给组件。

在 NavItem.tml 中使用参数命名空间的代码如下：

```
<t:container xmlns:t="http://tapestry.apache.org/schema/tapestry_5_1_0.xsd"
  xmlns:p="tapestry:parameter">
  <t:if test="self">
    ${pageDesc}
  <p:else>
    <t:pageLink t:page="${pageName}">${pageDesc}</t:pageLink>
  </p:else>
</t:if>
</t:container>
```

参数的属性

对NaviItem组件，page参数应该是必需的，这可以通过设置Parameter的required属性为true来实现：

```
@Parameter(required = true)
private String page;
```

而对desc参数，如果未设置，则把page参数的值作为其缺省值。参数缺省值由Parameter的value属性设置：

```
@Parameter(value = "prop:pageName")
private String desc;
```

value属性的值是“prop:pageName”，这也是一个绑定表达式，对应getPageName()方法，getPageName()方法返回的就是page参数的值。如果我们在这里未提供前缀，那么缺省使用参数的前缀，而参数的缺省前缀是“prop”，因此上面这段代码可以简化成：

```
@Parameter(value = "pageName")
private String desc;
```

除了可以用value属性来设置参数的缺省值外，还可以用缺省方法来设置，缺省方法的方法名为“default”后跟第一个字母大写的参数名。使用缺省方法设置缺省值的代码如下：

```
@Parameter
private String desc;

public String defaultDesc() {
    return getPageName();
}
```

现在，修改NaviItem类：

```
public class NaviItem {
    @Parameter(value = "pageName")
    private String desc;

    @Parameter(required = true)
    private String page;

    @Inject
    private ComponentResources componentResources;

    public boolean isSelf() {
        return componentResources.getPageName().equalsIgnoreCase(page);
    }

    public String getPageDesc() {
        return desc;
    }
}
```

```

    }

    public String getPageName() {
        return page;
    }
}

```

修改 NaviMenu.tml 文件，去掉第一个 Naviltem 组件的 “desc” 参数：

```

<t:container xmlns:t="http://tapestry.apache.org/schema/tapestry_5_1_0.xsd">
    <t:naviitem page="literal:Start" />
    |
    <t:naviitem desc="literal:联系我们" page="literal:Contact" />
</t:container>

```

页面效果如图 3-5 和 3-6。



(图 3-5)



(图 3-6)

在 NaviMenu.tml 中，Naviltem 组件的参数都有前缀 “literal”，如果我们将 “literal” 设置为这些参数的缺省前缀，那么我们就可以在这里省略 “literal” 前缀。

参数的缺省前缀由 @Parameter 的 defaultPrefix 属性设置。

修改 Naviltem 类：

```

public class NaviItem {
    @Parameter(value = "pageName", defaultPrefix = BindingConstants.LITERAL)
    private String desc;

    @Parameter(required = true, defaultPrefix = BindingConstants.LITERAL)
    private String page;

    @Inject
    private ComponentResources componentResources;

    public boolean isSelf() {
        return componentResources.getPageName().equalsIgnoreCase(pageName);
    }
}

```

```

e);
    }

    public String getPageDesc() {
        return desc;
    }

    public String getPageName() {
        return page;
    }
}

```

修改 NaviMenu.tml:

```

<t:container xmlns:t="http://tapestry.apache.org/schema/tapestry_5_1_
0.xsd">
    <t:naviitem page="Start" />
    |
    <t:naviitem desc="联系我们" page="Contact" />
</t:container>

```

页面效果如图 3-7 和 3-8



(图 3-7)



(图 3-8)

这里出现了问题，导航栏中的“Start”变成了“pageName”。

问题出在 NaviItem 组件的 desc 参数。

```

@Parameter(value = "pageName", defaultPrefix = BindingConstants.LITERA
L)
private String desc;

```

之前我们没有显式设置参数 desc 的缺省前缀，则其缺省前缀为“prop”，因此 value 属性的缺省前缀也就是“prop”，因此 value 的值相当于“prop:pageName”。现在我们把 desc 的缺省前缀设置为“literal”，则 value 属性的缺省前缀也就变成了“literal”，因此 value 的值相当于“literal:pageName”。要解决这个问题，只需明确给出“prop”前缀即可。

修改 NaviItem 类:

```

public class NaviItem {
    @Parameter(value = "prop:pageName", defaultPrefix = BindingConstan

```

```

ts.LITERAL)
    private String desc;

    @Parameter(required = true, defaultPrefix = BindingConstants.LITER
AL)
    private String page;

    @Inject
    private ComponentResources componentResources;

    public boolean isSelf() {
        return componentResources.getPageName().equalsIgnoreCase(pag
e);
    }

    public String getPageDesc() {
        return desc;
    }

    public String getPageName() {
        return page;
    }
}

```

现在再运行程序就正常了。

创建布局

Start 和 **Contact** 页面都由两部分组成，上面是导航菜单，下面是内容，它们的布局是一样的，不同的只是内容。

我们把它们的布局也做成一个组件 **MainLayout**，我们把 **MainLayout** 放在 **layout** 子目录下。

在 “**example.mycomponent.components.layout**” 包中新建 **MainLayout** 类：

```

public class MainLayout {
}

```

在 “**example.mycomponent.components.layout**” 包中新建 **MainLayout.tml**：

```

<html xmlns:t="http://tapestry.apache.org/schema/tapestry_5_1_0.xsd">
  <div>
    <t:navimenu />
  </div>
  <div>
    <t:body />
  </div>

```

```
</html>
```

这里我们用到了一个新元素<body>。

组件开始标签和结束标签之间的部分就是组件的body。

子目录

在使用子目录是，Tapestry 会对组件逻辑名进行优化，当包名是类名的前缀/后缀时，前缀/后缀将被去除。我们刚才创建的 MainLayout 组件是在 “example.mycomponent.components.layout” 包中的，其逻辑名为 “layout/Main”，而不是 “layout/MainLayout”。

在Start.tml中使用layout/Main组件：

```
<div t:type="layout/main" xmlns:t="http://tapestry.apache.org/schema/tapestry_5_1_0.xsd">
    首页
</div>
```

在Contact.tml中使用layout/Main组件：

```
<t:layout.main xmlns:t="http://tapestry.apache.org/schema/tapestry_5_1_0.xsd">
    联系我们
</t:layout.main>
```

由于<t:layout/main>不是一个合法的 XML 元素，因此在使用 XML 元素定义组件时，要把斜杠替换为点号。

刷新页面，一切正常。

如果我们在一个模板中多次使用来自同一子目录的组件，那么我们可以用库命名空间(Library Namespace)来简化对这些组件的使用。库命名空间的 URI 为 “tapestry-library:path”。

修改 Start.tml，在其中使用库命名空间：

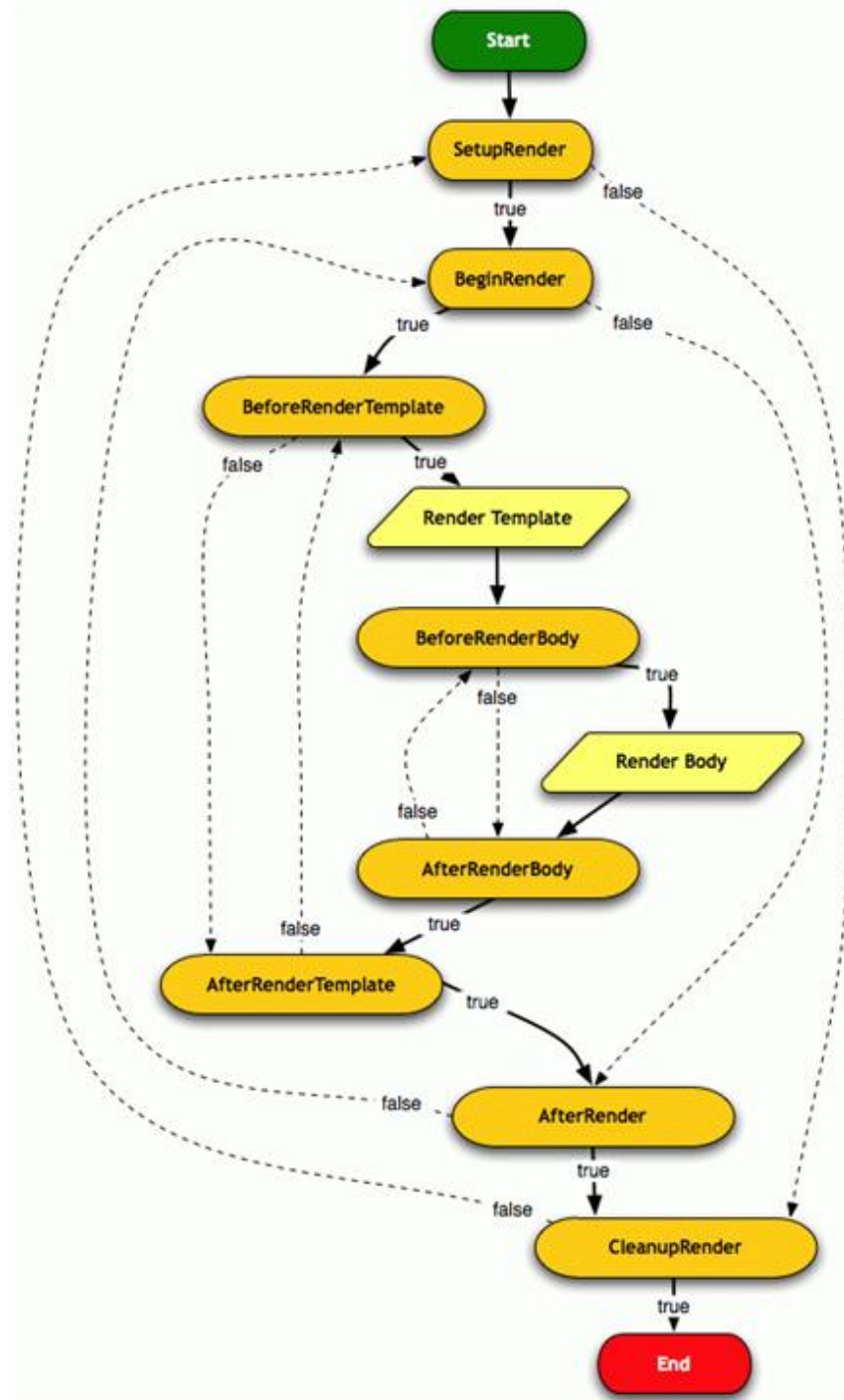
```
<l:main xmlns:l="tapestry-library:layout">
    首页
</l:main>
```

刷新页面，一切正常。

组件的解析

到目前为止，我们创建的组件（包括页面）都是根据模板来生成Html代码，然而模板不是必需的，组件的行为也可以更加复杂。

图 3-9 来自 Tapestry 官网，显示了组件的整个解析过程。



(图 3-9)

图中每个橘黄色阶段（SetupRender、BeginRender 等）对应组件类中的一些方法。Tapestry 用注释（Annotation）来标识方法所处的阶段，SetupRender 阶段的方法由 @SetupRender 注释标识，BeginRender 阶段方法由 @BeginRender 注释标识，以此类推。

比如下面的组件类：

```
public class Rendering {
    @SetupRender
    void function1() {
```



```

        ...
    }

    @SetupRender
    void function2(MarkupWriter writer) {
        ...
    }

    @BeginRender
    boolean function3() {
        ...
    }
}

```

方法 `function1()` 和 `function2()` 属于 `SetupRender` 阶段，而方法 `function3()` 属于 `BeginRender` 阶段。这些方法可以无返回值（`void`），也可以返回一个布尔值。图中实线表示方法无返回值或者返回布尔值 `true` 时的执行路径，虚线表示方法返回布尔值 `false` 时的执行路径。利用返回布尔值 `false`，可以跳过某些阶段，也可以再次访问某个阶段。比如如果 `SetupRender` 阶段的方法返回 `false`，那么将跳过中间很多阶段直接到达 `CleanupRender` 阶段，如果 `CleanupRender` 阶段的方法返回 `false`，将重新回到 `SetupRender` 阶段。这些方法可以不带参数，也可以带一个类型为 `MarkupWriter` 的参数。

当某个阶段有多个方法时，将按照如下步骤确定执行顺序：

- 多个方法分别来自子类和父类。那么如果是在 `SetupXXX`、`BeginXXX` 和 `BeforeXXX` 阶段，父类方法在子类方法之前执行。与之相匹配的是在 `AfterXXX` 和 `CleanupXXX` 阶段，子类方法在父类方法之前执行。
- 如果一个子类方法覆盖了父类方法，那么只会执行子类的方法，但是在执行顺序上仍然看成是父类方法。
- 单个类中的方法，按方法名称排序，名称相同的按参数个数排序。
- 当执行了一个返回布尔值的方法，不管返回 `true` 还是 `false`，同一阶段的其他方法都不再执行。

除了可以使用注释来标识方法所处的阶段，我们也可以使用特定的方法名来代替，方法命名规则是把注释名的第一个字母小写，比如：

```

@SetupRender
boolean function1() {
}

```

的替代写法是：

```

boolean setupRender() {
}

```

这两种写法可以混合使用。

一般来说，在 `SetupRender` 阶段做一些初始化工作，比如读取参数，设置临时变量等，与之相对应的是，在 `CleanupRender` 阶段做最后的清理工作。如果组件是用来产生一个标签

的，则在 **BeginRender** 阶段产生开始标签，在 **AfterRender** 阶段产生结束标签。**BeforeRenderTemplate** 和 **AfterRenderTemplate** 阶段用来装饰模板（把一些代码包裹在模板产生的代码外面）。当组件模板包含 **<body>** 元素，或者组件没有模板但有 **body** 时，**BeforeRenderBody** 和 **AfterRenderBody** 阶段的方法会触发，否则不会触发。

下面我们创建一个版权声明组件，这个组件将不使用模板。

在 “**example.mycomponent.components**” 包中新建 **Copyright** 类：

```
public class Copyright {
    void beginRender(MarkupWriter writer) {
        writer.write("Copyright 2009, Foo Corp.");
    }
}
```

把版权声明加入 **MainLayout.tml**：

```
<html xmlns:t="http://tapestry.apache.org/schema/tapestry_5_1_0.xsd">
  <div>
    <t:navimenu />
  </div>
  <div>
    <t:body />
  </div>
  <div>
    <t:copyright />
  </div>
</html>
```

下面是Tapestry官方文档里面的一个**Count**组件例子。

在 “**example.mycomponent.components**” 包中新建**Count**类：

```
public class Count {
    @Parameter
    private int start = 1;

    @Parameter(required = true)
    private int end;

    @Parameter
    private int value;

    private boolean increment;

    @SetupRender
    void initializeValue() {
        value = start;

        increment = start < end;
    }
}
```

```

    }

    @AfterRender
    boolean next() {
        if (increment) {
            int newValue = value + 1;

            if (newValue <= end) {
                value = newValue;
                return false;
            }
        } else {
            int newValue = value - 1;

            if (newValue >= end) {
                value = newValue;
                return false;
            }
        }

        return true;
    }
}

```

在 **Start** 页面，使用 **Count** 组件循环输出“首页”。

修改 **Start.tml**：

```

<l:main xmlns:t="http://tapestry.apache.org/schema/tapestry_5_1_0.xsd"
"
    xmlns:l="tapestry-library:layout">
    <t:count start="9" end="8" value="var:index">
        <div>${var:index}:首页</div>
    </t:count>
</l:main>

```

注意这里“**var**”前缀的用法。如果用“**prop**”前缀，需要在 **Start** 页面类中增加 **index** 属性。

Start 页面效果如图 3-10。



(图 3-10)

其它 Tapestry 元素

Tapestry 元素就是使用了 Tapestry 命名空间前缀的元素。前面我们已经接触过 3 个 Tapestry 元素，`<body>`、`<container>`和`<block>`。下面再简单介绍一下几个 Tapestry 5.1 新增的 Tapestry 元素。

`<content>`元素

`<content>`用来标记真正的模板内容，`<content>`元素之外的内容都将被忽略。比如下面的模板，只有红色部分才是真正的内容，其它部分可能只是为了方便 WYSIWYG（所见即所得）工具编辑或预览模板而加上的，在解析时会被忽略掉。

```
<html xmlns:t="http://tapestry.apache.org/schema/tapestry_5_1_0.xsd">
  <body>
    <t:content>
      <span>Copyright 2009, Foo Corp.</span>
    </t:content>
  </body>
</html>
```

`<remove>`元素

`<remove>`元素用来标记模板中需要被排除的内容，`<remove>`元素和它所包含的内容就像是根本不存在于模板中一样。`<remove>`元素主要用于包含服务器端注释，不会出现在 HTML 代码中，而一般的 HTML/XML 注释会被加入 HTML 代码中。

接下来的几个 Tapestry 元素是用来支持模板的继承的。

`<extension-point>`元素

`<extension-point>`元素用来标记可能会在子模板中被替换的内容，它有唯一的 ID（大小写不敏感），子模板根据这个 ID 替换父模板中的内容。

```
<t:extension-point id="title">
  <h1>${defaultTitle}</h1>
</t:extension-point>
```

<extend>元素

<extend>元素是子模板的根元素，只包含<replace>元素。

<replace>元素

<replace>元素根据 ID 替换父模板中的相应内容。

```
<t:extend xmlns:t="http://tapestry.apache.org/schema/tapestry_5_1_0.xsd">
  <t:replace id="title">
    <h1>
      Customer Service</h1>
  </t:replace>
</t:extend>
```

第四章 表单

本章主要介绍常用的表单组件和表单的校验。

下面我们要做一个注册新用户的程序，所谓注册其实仅仅是在控制台打印用户输入数据而已。

仿照第一章的例子新建一个动态 Web 工程“Register”，加入需要的 jar 文件，修改 web.xml 成如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
    <display-name>Register</display-name>
    <context-param>
        <param-name>tapestry.app-package</param-name>
        <param-value>example.register</param-value>
    </context-param>
    <filter>
        <filter-name>app</filter-name>
        <filter-class>org.apache.tapestry5.TapestryFilter</filter-clas
s>
    </filter>
    <filter-mapping>
        <filter-name>app</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
</web-app>
```

我们先实现简单一点的，让用户输入用户名、密码、电子邮件和年龄。

新建一个 User 类，用于表示用户信息：

```
public class User {
    private String name;
    private String password;
    private String email;
    private int age;

    public String getName() {
        return name;
    }
}
```

```
public void setName(String name) {
    this.name = name;
}

public String getPassword() {
    return password;
}

public void setPassword(String password) {
    this.password = password;
}

public String getEmail() {
    return email;
}

public void setEmail(String email) {
    this.email = email;
}

public int getAge() {
    return age;
}

public void setAge(int age) {
    this.age = age;
}

public String toString() {
    return "Name:" + name + "\n" + "Password:" + password + "\n" + "
Email:"
        + email + "\n" + "Age:" + age;
}
}
```

Form、TextField、PasswordField 和 Label 组件

在“example.register.pages”包中新建 Start 类：

```
public class Start {
    @Property
    private User user;
    @Property
    private String password2;
```

```

void onSuccess() {
    System.out.println(user);
    System.out.println("password2:" + password2);
}
}

```

在“example.register.pages”包中新建 Start.tml:

```

<html xmlns:t="http://tapestry.apache.org/schema/tapestry_5_1_0.xsd">
  <div>注册</div>
  <t:form t:id="regForm" clientValidation="false">
    <div>
      <t:label for="userName" />
      <t:textfield t:id="userName" value="user.name" label="用户名" />
    </div>
    <div>
      <t:label for="password" />
      <t:passwordfield t:id="password" value="user.password" label="密码" />
    </div>
    <div>
      <t:label for="password2" />
      <t:passwordfield t:id="password2" value="password2" label="重复密码" />
    </div>
    <div>
      <t:label for="email" />
      <t:textfield t:id="email" value="user.email" label="电子邮件" />
    </div>
    <div>
      <t:label for="age" />
      <t:textfield t:id="age" value="user.age" label="年龄" />
    </div>
    <input type="submit" value="确定" />
  </t:form>
</html>

```

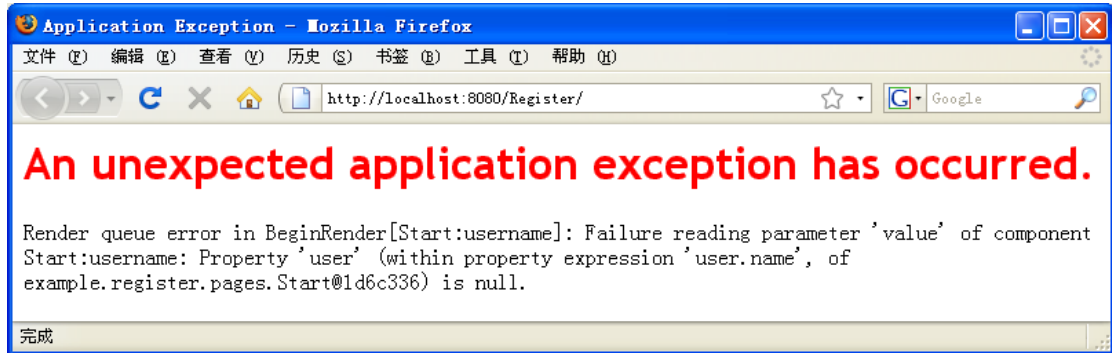
这里我们共使用了四个组件，Form、TextField、PasswordField 和 Label。

其中 Form 组件用来包含其他的表单组件，把 Form 组件的“clientValidation”属性设置为 false，就取消了表单的客户端校验功能，我们把表单校验放在本章后半部分讲解。当表单提交并校验成功，会触发 success 事件，在控制台打印用户输入数据。除了 success，For

m 组件的事件还有 prepareForRender、prepare、prepareForSubmit、validateForm、failure 和 submit。

关于这些组件的详细信息，请参考官方文档。

把 “Register” 加入服务器并重启服务器，打开 <http://localhost:8080/Register/>，页面报告异常，如图 4-1。



(图 4-1)

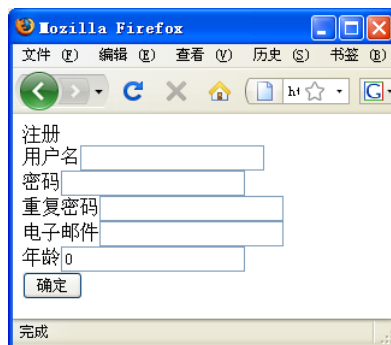
发生异常的原因是 Tapestry 要用表达式 “user.name” 的值作为 userName 文本框的初始值，但是由于 user 为 null，所以计算表达式时出现异常。

修改 Start 类：

```
public class Start {
    @Property
    private User user = new User();
    @Property
    private String password2;

    void onSuccess() {
        System.out.println(user);
        System.out.println("password2:" + password2);
    }
}
```

刷新 Start 页面，页面效果如图 4-2。



(图 4-2)

RadioGroup 和 Radio 组件

接下来我们想让用户输入性别。

我们创建一个枚举来表示性别，代码如下：

```
public enum Gender {  
    MALE, FEMALE;  
}
```

同时为 User 类增加一个 gender 属性：

```
public class User {  
    private String name;  
    private String password;  
    private String email;  
    private int age;  
    private Gender gender;  
  
    ...  
  
    public Gender getGender() {  
        return gender;  
    }  
  
    public void setGender(Gender gender) {  
        this.gender = gender;  
    }  
  
    public String toString() {  
        return "Name:" + name + "\n" + "Password:" + password + "\n" + "  
Email:"  
        + email + "\n" + "Age:" + age + "\n" + "Gender:" + gende  
r;  
    }  
}
```

修改 Start.html:

```
<html xmlns:t="http://tapestry.apache.org/schema/tapestry_5_1_0.xsd">  
    <div>注册</div>  
    <t:form t:id="regForm" clientValidation="false">  
        ...  
        <div>  
            <t:label for="gender" />  
            <t:radiogroup t:id="gender" value="user.gender" label="性别  
<div>  
                <t:radio t:id="male" value="male" label="男" />  
                <t:label for="male" />  
            </div>  
        </div>  
    </t:form>  
</html>
```

```

        <t:radio t:id="female" value="female" label="女" />
        <t:label for="female" />
    </t:radiogroup>
</div>
<input type="submit" value="确定" />
</t:form>
</html>

```

修改 Start 类:

```

public class Start {
    @Property
    private User user = new User();
    @Property
    private String password2;

    public Gender getMale() {
        return Gender.MALE;
    }

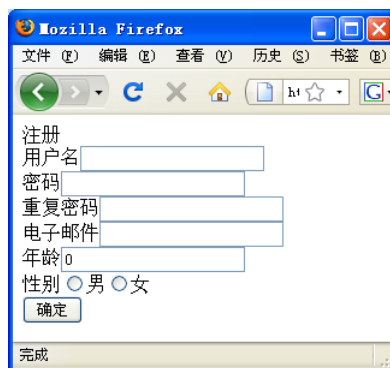
    public Gender getFemale() {
        return Gender.FEMALE;
    }

    void onSuccess() {
        System.out.println(user);
        System.out.println("password2:" + password2);
    }
}

```

我们在 Start.tml 中使用了一个 RadioGroup 组件,其 value 参数是表达式“user.gender”。在 RadioGroup 组件里面有两个 Radio 组件,对每个 Radio 组件,在 Start 类中都有一个 getter 方法为其提供值。比如第一个 Radio 组件,其 value 参数是表达式“male”,那么在 Start 类中就有一个 getMale()方法,此方法返回 Gender.MALE,如果这个 Radio 组件被选中,提交表单时“user.gender”就会被设置为 Gender.MALE。

Start 页面效果如图 4-3。



(图 4-3)

Select 组件

接下来我们用一个下拉框来让用户选择国家。

我们同样用一个枚举来表示国家，代码如下：

```
public enum Country {  
    CHINA, USA;  
}
```

为 `User` 类增加一个属性：

```
public class User {  
    private String name;  
    private String password;  
    private String email;  
    private int age;  
    private Gender gender;  
    private Country country;  
  
    ...  
  
    public Country getCountry() {  
        return country;  
    }  
  
    public void setCountry(Country country) {  
        this.country = country;  
    }  
  
    public String toString() {  
        return "Name:" + name + "\n" + "Password:" + password + "\n" + "  
Email:"  
                + email + "\n" + "Age:" + age + "\n" + "Gender:" + gender  
                + "\n" + "Country:" + country;  
    }  
}
```

修改 `Start.html`：

```
<html xmlns:t="http://tapestry.apache.org/schema/tapestry_5_1_0.xsd">  
    <div>注册</div>  
    <t:form t:id="regForm" clientValidation="false">  
        ...  
    </div>  
        <t:label for="country" />
```

```

        <t:select t:id="country" model="countries" value="user.country"
        label="国家" />
    </div>
    <input type="submit" value="确定" />
</t:form>
</html>

```

修改 **Start** 类:

```

public class Start {
    @Property
    private User user = new User();
    @Property
    private String password2;
    @Inject
    private Messages messages;

    public Gender getMale() {
        return Gender.MALE;
    }

    public Gender getFemale() {
        return Gender.FEMALE;
    }

    public SelectModel getCountries() {
        return new EnumSelectModel(Country.class, messages);
    }

    void onSuccess() {
        System.out.println(user);
        System.out.println("password2:" + password2);
    }
}

```

Select 组件的 `model` 参数的类型是 `org.apache.tapestry5.SelectModel`，用于提供下拉框中的选项。`SelectModel` 是一个接口，我们可以自己创建实现 `SelectModel` 接口的类。Tapestry 有一个专门用于枚举的实现类 `org.apache.tapestry5.util.EnumSelectModel`。`EnumSelectModel` 类的构造方法有两个参数，第一个是枚举的 `class`，第二个是 `org.apache.tapestry5.ioc.Messages`。关于 `Messages`，在本地化一章中有专门讨论，这里我们只要知道 `Messages` 会从 `properties` 文件获取本地化信息用于显示在下拉框中。

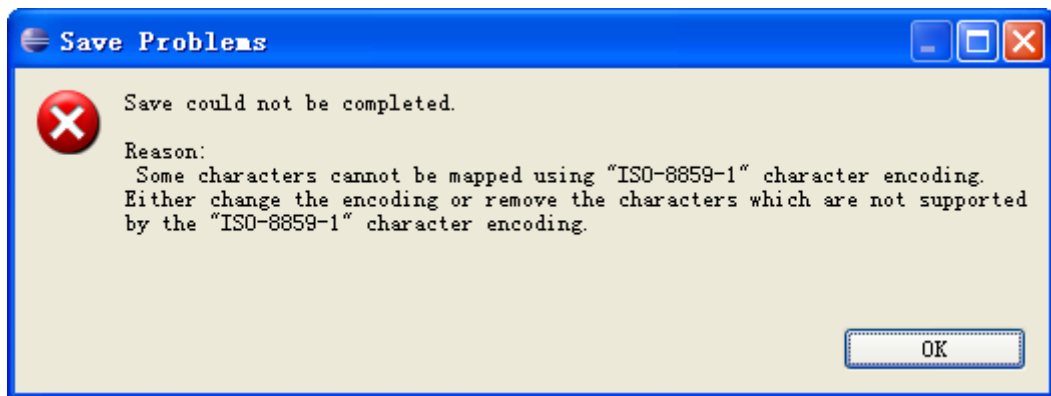
在 `WEB-INF` 目录创建 `app.properties` 文件，其内容为：

```

Country.CHINA=中国
Country.USA=美国

```

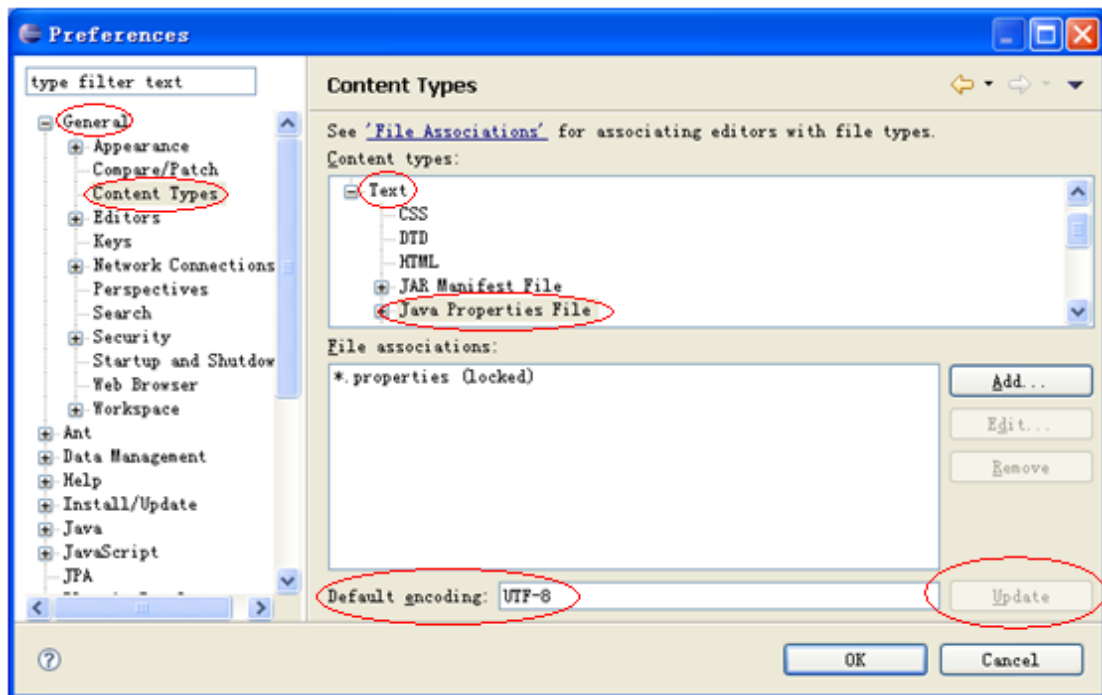
当我们保存 app.properties 文件时, Eclipse 提示错误, 如图 4-4。



(图 4-4)

在 Eclipse 中, properties 文件默认采用“ISO-8859-1”字符编码, 因此对某些中文字符不支持, 我们把它改成“UTF-8”就可以了。

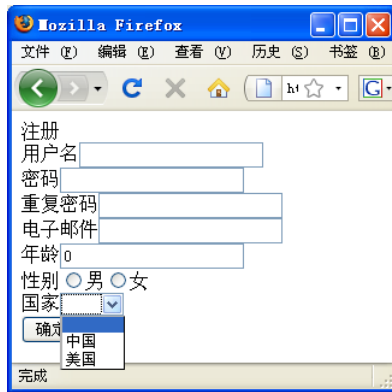
点击 Eclipse 菜单【Window】【Preferences】。选中 General 下的 Content Types, 在右上列表框中选中 Text 下的 Java Properties File, 然后把右下方的 Default encoding 从“ISO-885 9-1”改为“UTF-8”, 如图 4-5。点击 Default encoding 右边的 Update 按钮, 然后点击 OK 按钮退出就行了。



(图 4-5)

现在可以正常保存 app.properties 文件了。

Start 页面效果如图 4-6。



(图 4-6)

Checkbox 组件

最后，增加一个复选框，让用户选择是否要预订邮件。

修改 User 类，增加一个属性：

```
public class User {
    private String name;
    private String password;
    private String email;
    private int age;
    private Gender gender;
    private Country country;
    private boolean subscribe;

    ...

    public boolean isSubscribe() {
        return subscribe;
    }

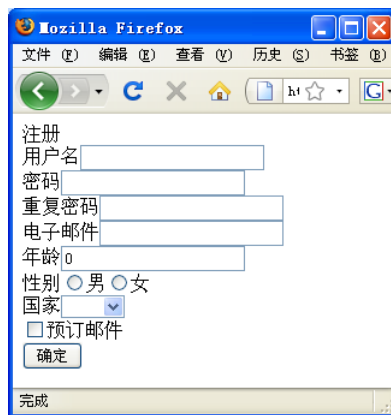
    public void setSubscribe(boolean subscribe) {
        this.subscribe = subscribe;
    }

    public String toString() {
        return "Name:" + name + "\n" + "Password:" + password + "\n" + "
Email:"
        + email + "\n" + "Age:" + age + "\n" + "Gender:" + gender
        + "\n" + "Country:" + country + "\n" + "Subscribe:" + sub
scribe;
    }
}
```

修改 Start.tml:

```
<html xmlns:t="http://tapestry.apache.org/schema/tapestry_5_1_0.xsd">
  <div>注册</div>
  <t:form t:id="regForm" clientValidation="false">
    ...
    <div>
      <t:checkbox t:id="subscribe" value="user.subscribe" label="
预订邮件" />
      <t:label for="subscribe" />
    </div>
    <input type="submit" value="确定" />
  </t:form>
</html>
```

Start 页面效果如图 4-7。



(图 4-7)

Submit 组件

在上面的表单中，我们并没有使用 Submit 组件，而是使用普通的 Html Input 标签。

如果我们要在表单中再增加一个“注册并登录”按钮，那么表单中就有两个提交按钮，这时使用 Submit 组件可以很容易针对每个按钮作不同的处理。

修改 Start.tml:

```
<html xmlns:t="http://tapestry.apache.org/schema/tapestry_5_1_0.xsd">
  <div>注册</div>
  <t:form t:id="regForm" clientValidation="false">
    ...
    <div>
      <t:checkbox t:id="subscribe" value="user.subscribe" label="
预订邮件" />
      <t:label for="subscribe" />
    </div>
    <t:submit t:id="register" value="注册" />
  </t:form>
</html>
```



```
<t:submit t:id="registerAndLogin" value="注册并登录" />
</t:form>
</html>
```

修改 **Start** 类:

```
public class Start {
    @Property
    private User user = new User();
    @Property
    private String password2;
    private boolean login;
    @Inject
    private Messages messages;

    public Gender getMale() {
        return Gender.MALE;
    }

    public Gender getFemale() {
        return Gender.FEMALE;
    }

    public SelectModel getCountries() {
        return new EnumSelectModel(Country.class, messages);
    }

    void onSelectedFromRegister() {
        login = false;
    }

    void onSelectedFromRegisterAndLogin() {
        login = true;
    }

    void onSuccess() {
        System.out.println(user);
        System.out.println("password2:" + password2);
        System.out.println("login:" + login);
    }
}
```

当我们点击 **Submit** 组件，会触发 **selected** 事件。如果点击注册按钮，**login** 被赋值为 **false**，如果点击注册并登录按钮，**login** 被赋值为 **true**。在注册成功后，如果 **login == true**，则执行登录操作。

Start 页面效果如图 4-8。



(图 4-8)

点击“注册并登陆”按钮，控制台输出如下：

Name:myname
Password:pwd123
Email:myname@foo.com
Age:11
Gender:MALE
Country:CHINA
Subscribe:true
password2:pwd123
login:true

使用 Validator 校验表单

为避免用户输入非法数据，必须对表单数据进行校验。
大部分的校验工作我们只需使用 Tapestry 内置的 Validator 就能完成。
比如，用户名是必须输入的，我们就可以用“Required” Validator 对用户名进行校验。

```
<t:textfield t:id="userName" value="user.name" label="用户名"
  validate="validate:required" />
```

在这里唯一要做的只是设置 TextField 组件的 validate 参数，其表达式前缀是“validate”，
“validate”是 validate 参数的缺省前缀，因此可以省略。

```
<t:textfield t:id="userName" value="user.name" label="用户名"
  validate="required" />
```

下表列出了所有 Tapestry 5 内置的 Validator。

名称	含义	用法示例
Required	必须输入。	required
MinLength	最少字符个数。	minlength=6
MaxLength	最多字符个数。	maxlength=20

Min	最小数字。	min=1
Max	最大数字。	max=99
Regexp	必须满足的正则表达式。	regexp=^[a-zA-Z0-9_]*\$
Email	必须是合法电子邮件地址。	email

如果需要使用多个 **Validator**，用 “,” 分隔。

我们对用户名、密码、重复密码、电子邮件和年龄进行如下校验：

- 用户名是必需的，长度为 6 至 20 个字符，而且只能由字母、数字和下划线组成。

```
<t:textfield t:id="userName" value="user.name" label="用户名"
  validate="required,minlength=6,maxlength=20,regexp=^[a-zA-Z0-9_]*$"/>
</t:textfield>
```

- 密码和重复密码是必需的，长度为 6 至 20 个字符。

```
<t:passwordfield t:id="password" value="user.password"
  label="密码" validate="required,minlength=6,maxlength=20"/>
</t:passwordfield>
<t:passwordfield t:id="password2" value="password2"
  label="重复密码" validate="required,minlength=6,maxlength=20"/>
</t:passwordfield>
```

- 电子邮件是必需的，而且必须符合 **Email** 地址格式。

```
<t:textfield t:id="email" value="user.email" label="电子邮件"
  validate="required,email"/>
</t:textfield>
```

- 年龄是必需的，大小在 1 至 99 之间。

```
<t:textfield t:id="age" value="user.age" label="年龄"
  validate="required,min=1,max=99"/>
</t:textfield>
```

由于年龄是 **int** 类型，因此 **Tapestry** 自动设置它是必须输入整数的。

最后的 **Start.tml** 代码如下：

```
<html xmlns:t="http://tapestry.apache.org/schema/tapestry_5_1_0.xsd">
  <div>注册</div>
  <t:form t:id="regForm" clientValidation="false">
    <div>
      <t:label for="userName" />
      <t:textfield t:id="userName" value="user.name" label="用户名"
        validate="required,minlength=6,maxlength=20,regexp=^[a-zA-Z0-9_]*$" />
    </div>
    <div>
      <t:label for="password" />
      <t:passwordfield t:id="password" value="user.password"
        label="密码" validate="required,minlength=6,maxlength=20" />
    </div>
    <div>
      <t:label for="password2" />
    </div>
  </t:form>
</html>
```

```

        <t:passwordfield t:id="password2" value="password2"
            label="重复密码" validate="required,minlength=6,maxlength=20" />
    </div>
    <div>
        <t:label for="email" />
        <t:textfield t:id="email" value="user.email" label="电子邮件"
            validate="required,email" />
    </div>
    <div>
        <t:label for="age" />
        <t:textfield t:id="age" value="user.age" label="年龄"
            validate="required,min=1,max=99" />
    </div>
    <div>
        <t:label for="gender" />
        <t:radiogroup t:id="gender" value="user.gender" label="性别"
            >
            <t:radio t:id="male" value="male" label="男" />
            <t:label for="male" />
            <t:radio t:id="female" value="female" label="女" />
            <t:label for="female" />
        </t:radiogroup>
    </div>
    <div>
        <t:label for="country" />
        <t:select t:id="country" model="countries" value="user.country"
            label="国家" />
    </div>
    <div>
        <t:checkbox t:id="subscribe" value="user.subscribe" label="
        预订邮件" />
        <t:label for="subscribe" />
    </div>
    <t:submit t:id="register" value="注册" />
    <t:submit t:id="registerAndLogin" value="注册并登录" />
</t:form>
</html>

```

打开 **Start** 页面，清空所有输入，然后提交表单，页面效果如图 4-9。

(图 4-9)

所有输入错误的字段都变成了红色，但是没有任何错误信息。

使用 **Errors** 组件显示错误信息

Errors 组件是专门用来显示错误信息的，它必须被包含在 **Form** 组件里面。

我们把错误信息显示在表单上方。

修改 **Start.html**:

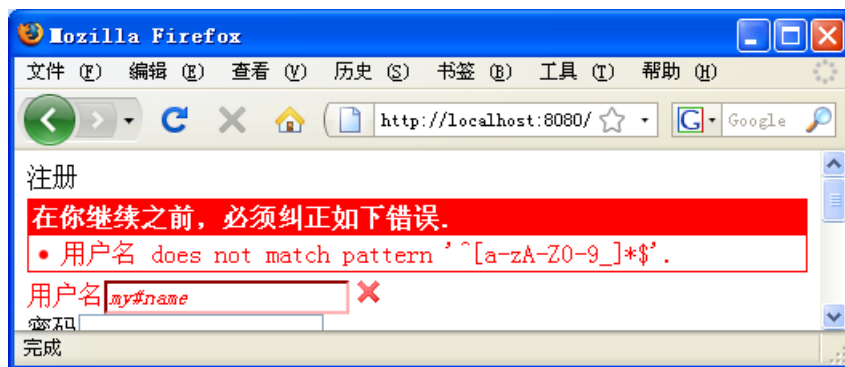
```
<html xmlns:t="http://tapestry.apache.org/schema/tapestry_5_1_0.xsd">
  <div>注册</div>
  <t:form t:id="regForm" clientValidation="false">
    <t:errors />
    ...
  </t:form>
</html>
```

再次提交 **Start** 页面，页面效果如图 4-10。

(图 4-10)

自定义错误信息

如果输入的用户名是“my#name”，页面效果如图 4-11。



(图 4-11)

这里我们看到的是英文的错误信息，这是由于缺少“Regexp” Validator 对应的中文错误信息引起的，还有其它一些 Validator 也都缺少中文错误信息。我们可以在 Tapestry 源代码中找到“ValidationMessages_zh_CN.properties”文件，把中文信息补齐，重新编译打包再拿来用。

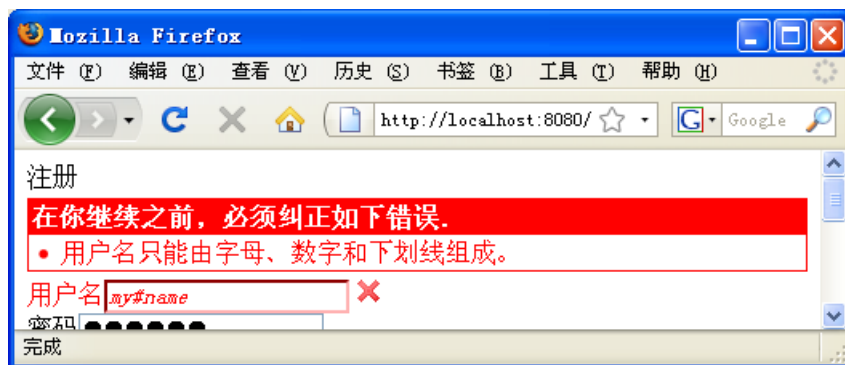
但是即使这里显示的是中文信息，也不够友好，用户很难理解正则表达式。

Tapestry 允许我们自己提供错误信息。

在“example.register.pages”包中新建 Start.properties 文件，内容如下：

```
userName-regexp-message=%2$s只能由字母、数字和下划线组成。
```

“userName-regexp-message”是由组件 ID、Validator 名称和“message”三部分组成的。重启服务器，页面效果如图 4-12。



(图 4-12)

有时正则表达式会很复杂，很难放在模板中。这时我们可以把正则表达式放到 properties 文件中。

把 Start.tml 中的正则表达式去掉：

```
<html xmlns:t="http://tapestry.apache.org/schema/tapestry_5_1_0.xsd">
  <div>注册</div>
  <t:form t:id="regForm" clientValidation="false">
    <t:errors />
    <div>
```

```

        <t:label for="userName" />
        <t:textfield t:id="userName" value="user.name" label="用户名"
"
            validate="required,minlength=6,maxlength=20,regexp" />
    </div>
    ...
</t:form>
</html>

```

然后在 **Start.properties** 中增加一行：

```

userName-regexp-message=%2$s只能由字母、数字和下划线组成。
userName-regexp=^[a-zA-Z0-9_]*$

```

运行程序，效果还是一样的。

客户端校验

校验包括客户端校验和服务器端校验两种。客户端校验是直接在浏览器中进行的，通常使用 **javascript**。服务器端校验则是把数据提交到服务器，在服务器端进行校验。客户端校验不需要来回传输数据，而且不消耗服务器计算资源。但是用户可能会有意无意的绕过客户端校验，因此服务器端的校验是必需的。

之前我们取消了表单的客户端校验功能，所看到的是服务器端校验。要恢复客户端校验，只需把 **Form** 组件的 **clientValidation** 参数设置为 **true**，或者干脆省略，因为其缺省为 **true**。

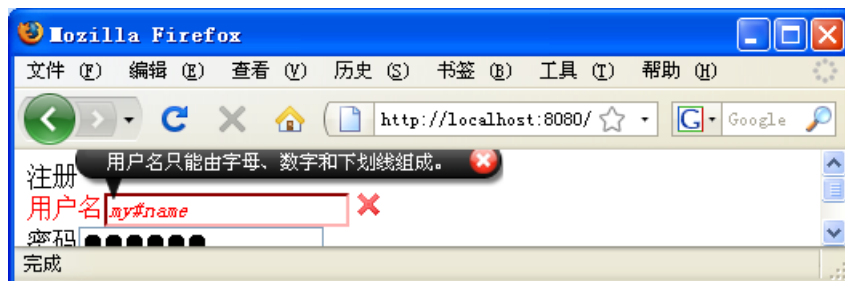
修改 **Start.tml**：

```

<html xmlns:t="http://tapestry.apache.org/schema/tapestry_5_1_0.xsd">
    <div>注册</div>
    <t:form t:id="regForm">
        ...
    </t:form>
</html>

```

现在，每当改变焦点或者点击提交按钮，就可以看如图 4-13 所示错误提示。



(图 4-13)

Email Validator 并没有实现客户端校验功能。

校验多个字段

现在我们要校验密码和重复密码是不是一样。

Tapestry 提供的 `Validator` 都是对单个字段进行校验的，象这种同时对多个字段进行的校验，我们可以在 `Form` 组件的 `validateForm` 事件中进行。

修改 `Start` 类：

```
public class Start {
    @Property
    private User user = new User();
    @Property
    private String password2;
    private boolean login;
    @Inject
    private Messages messages;
    @Component
    private Form regForm;
    @Component(id = "password2")
    private PasswordField password2Field;

    public Gender getMale() {
        return Gender.MALE;
    }

    public Gender getFemale() {
        return Gender.FEMALE;
    }

    public SelectModel getCountries() {
        return new EnumSelectModel(Country.class, messages);
    }

    void onSelectedFromRegister() {
        login = false;
    }

    void onSelectedFromRegisterAndLogin() {
        login = true;
    }

    void onValidateForm() {
        if (password2 != null && !password2.equals(user.getPassword()))
        {
            regForm.recordError(password2Field, "两个密码不一样。");
        }
    }
}
```



```
    }  
}  
  
void onSuccess() {  
    System.out.println(user);  
    System.out.println("password2:" + password2);  
    System.out.println("login:" + login);  
}  
}
```

当密码和重复密码不一样时，页面效果如图 4-14。

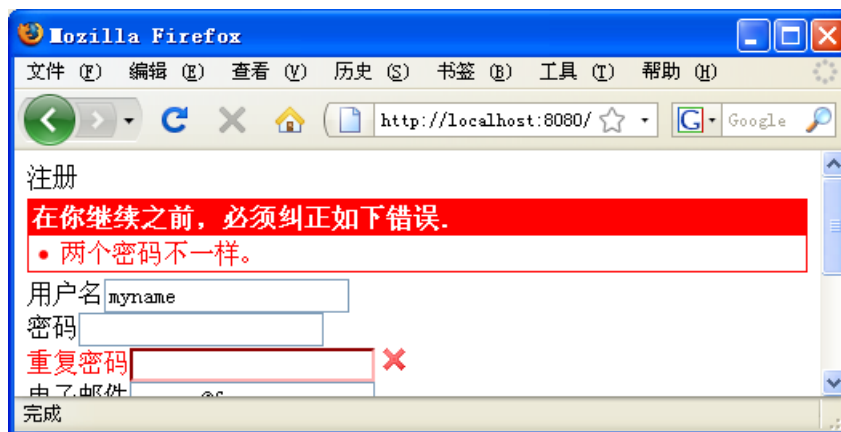


图 (4-14)

`Form#recordError(Field field, String errorMessage)`方法用于报告某个字段的错误。而 `Form#recordError(String errorMessage)`方法只报告错误，不跟任何特定的字段相关。

第五章 本地化

本地化（Localization）就是为来自不同国家/地区、语言的用户提供不同的用户界面。

本章的例子仍然使用上一章的“Register”。当用户来自中国内地，界面显示中文简体，对其他国家/地区的用户显示英文。

支持特定的语言

首先我们要让应用程序只支持英文和中文简体。

在“example.register.services”包中新建 AppModule 类：

```
public class AppModule {
    public static void contributeApplicationDefaults(
        MappedConfiguration<String, String> configuration) {
        configuration.add(SymbolConstants.SUPPORTED_LOCALES, "en, zh_CN");
    }
}
```

在前面的配置中，“en”和“zh”是语言代码，是由 ISO-639 定义的小写两字母代码，“CN”是国家/地区代码，是由 ISO-3166 定义的大写两字母代码。更详细的信息请参考 `java.util.Locale`。

“en”放在最前面，这样对于那些应用程序不支持的语言，都显示英文。

消息目录

一个消息目录（Message Catalog）就是一组 properties 文件。Properties 文件的格式就是 `java.util.ResourceBundle` 所用的文件格式。

每个页面/组件都可以有自己的消息目录。对于 Start 页面，它可以有一个主 properties 文件 `Start.properties`。还可以有特定于某种语言的 properties 文件，如 `Start_en.properties` 是特定于英语的，`Start_zh.properties` 是特定于中文的。更进一步的，Start 页面还可以有特定于讲某种语言的某个国家/地区的 properties 文件，如 `Start_zh_CN.properties` 是特定于讲中文的中国（大陆）的。这些 properties 文件和 Start 页面类在同一个目录中。

还有一个特殊的消息目录，其作用域是整个应用。它的主 properties 文件是 `app.properties`，“app”就是 `web.xml` 中的 Filter 名称。类似的，它还可以有 `app_zh.properties`、`app_zh_CN.properties` 等 properties 文件。这些文件放在 `WEB-INF` 目录下。

现在来看 Start 页面。首先在 `Start.tml` 中有如下代码：

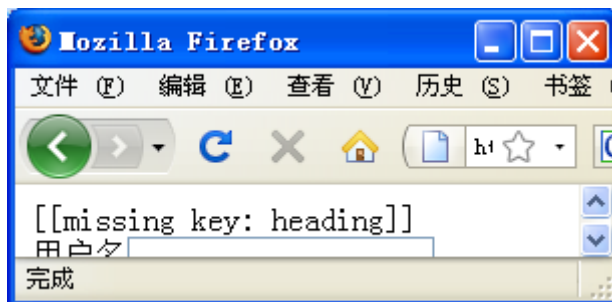
```
<div>注册</div>
```

把“注册”这两个字改成一个 Expansion，其表达式前缀是“message”。

修改 Start.tml:

```
<html xmlns:t="http://tapestry.apache.org/schema/tapestry_5_1_0.xsd">
  <div>${message:heading}</div>
  <t:form t:id="regForm">
    ...
  </t:form>
</html>
```

启动服务器。Start 页面效果如图 5-1。



(图 5-1)

当一个 Start 页面请求来自“zh_CN”，Tapestry 首先搜索 Start_zh_CN.properties 文件，用其中的“heading”来替换页面模板中的\${message:heading}，如果失败（Start_zh_CN.properties 文件不存在或者 Start_zh_CN.properties 文件中没有“heading”），则继续搜索 Start_zh.properties，如果还失败，则继续搜索 Start.properties。如果 Start 页面消息目录中没有“heading”，则搜索应用消息目录（Application Message Catalog），搜索顺序为 app_zh_CN.properties、app_zh.properties、app.properties。如果都没找到“heading”，则用“[[missing key: heading]]”代替。

现在，在“example.register.pages”包中新建如下两个文件：

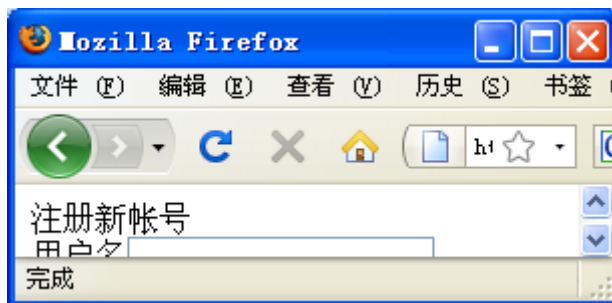
Start_en.properties:

```
heading=Register
```

Start_zh_CN.properties:

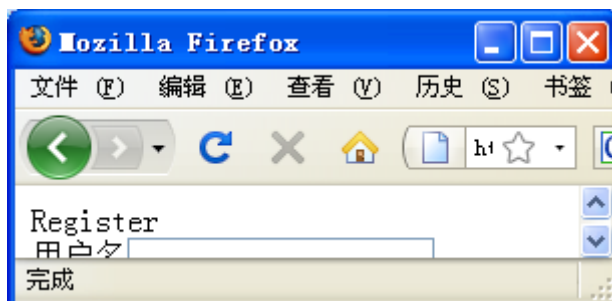
```
heading=注册新帐号
```

重启服务器，打开 <http://localhost:8080/Register/>，Start 页面效果如图 5-2。



(图 5-2)

把 Locale 名作为 URL 的第一个虚拟目录，就能得到与 Locale 相对应的页面。比如 http://localhost:8080/Register/zh_CN/Start 对应刚才的中文简体页面，而 <http://localhost:8080/Register/en/Start> 对应英文页面，如图 5-3。要注意 Locale 名必须是我们的应用程序支持的（在 `tapestry.supported-locales` 中设置，见本章开头部分）。



(图 5-3)

接着看 `Start.tml` 中的组件。

```
<t:textfield t:id="userName" value="user.name" label="用户名" validate="required,minlength=6,maxlength=20,regexp" />
```

我们可以仿照刚才的做法，把 `label` 参数设为 “`message:userName`”，然后在 `properties` 文件中增加 “`userName=xxx`”。更简单一点的，我们不提供 `label` 属性，那么 `TextField` 组件默认其值为 “`message:userName-label`”（如果找不到，直接以 “`User Name`” 字符串代替）。

修改所有组件后，`Start.tml` 代码如下：

```
<html xmlns:t="http://tapestry.apache.org/schema/tapestry_5_1_0.xsd">
  <div>${message:heading}</div>
  <t:form t:id="regForm">
    <t:errors />
    <div>
      <t:label for="userName" />
      <t:textfield t:id="userName" value="user.name"
        validate="required,minlength=6,maxlength=20,regexp" />
    </div>
    <div>
      <t:label for="password" />
      <t:passwordfield t:id="password" value="user.password"
        validate="required,minlength=6,maxlength=20" />
    </div>
    <div>
      <t:label for="password2" />
      <t:passwordfield t:id="password2" value="password2"
        validate="required,minlength=6,maxlength=20" />
    </div>
    <div>
      <t:label for="email" />
      <t:textfield t:id="email" value="user.email" validate="requ
```

```

ired,email" />
    </div>
    <div>
        <t:label for="age" />
        <t:textfield t:id="age" value="user.age" validate="required,min=1,max=99" />
    </div>
    <div>
        <t:label for="gender" />
        <t:radiogroup t:id="gender" value="user.gender">
            <t:radio t:id="male" value="male" />
            <t:label for="male" />
            <t:radio t:id="female" value="female" />
            <t:label for="female" />
        </t:radiogroup>
    </div>
    <div>
        <t:label for="country" />
        <t:select t:id="country" model="countries" value="user.country" />
    </div>
    <div>
        <t:checkbox t:id="subscribe" value="user.subscribe" />
        <t:label for="subscribe" />
    </div>
    <t:submit t:id="register" value="{message:register}" />
    <t:submit t:id="registerAndLogin" value="{message:registerAndLogin}" />
</t:form>
</html>

```

修改 Start_en.properties:

```

heading=Register
userName-label=User Name
password-label=Password
password2-label=Repeat Password
email-label=Email
age-label=Age
gender-label=Gender
male-label=Male
female-label=Female
country-label=Country
subscribe-label=Subscribe newsletter
register=Register

```

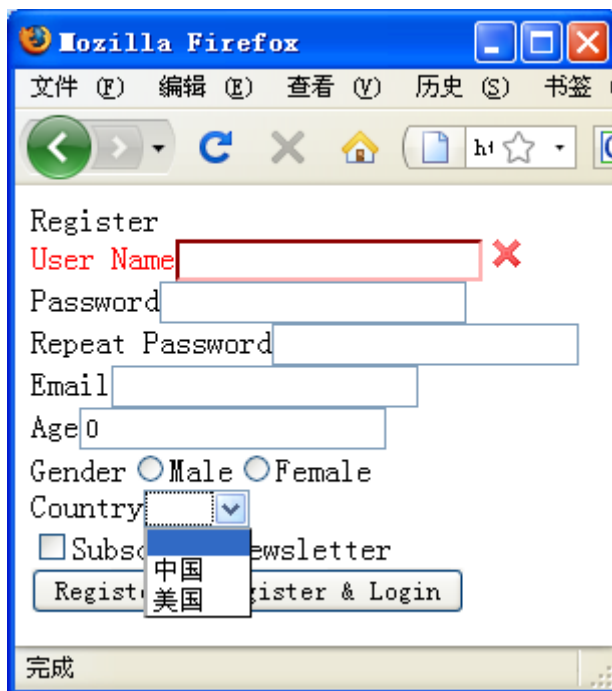
```
registerAndLogin=Register & Login
```

修改 Start_zh_CN.properties:

```
heading=注册新帐号
userName-label=用户名
password-label=密码
password2-label=重复密码
email-label=电子邮件
age-label=年龄
gender-label=性别
male-label=男
female-label=女
country-label=国家
subscribe-label=预订邮件
register=注册
registerAndLogin=注册并登录
```

当我们修改了 `properties` 文件，我们可以马上看到页面的变化，就像修改了类和模板一样。

刷新 <http://localhost:8080/Register/en/Start>，页面中的大部分文字都是英文了，但是下拉框中的国家名还是中文，如图 5-4。

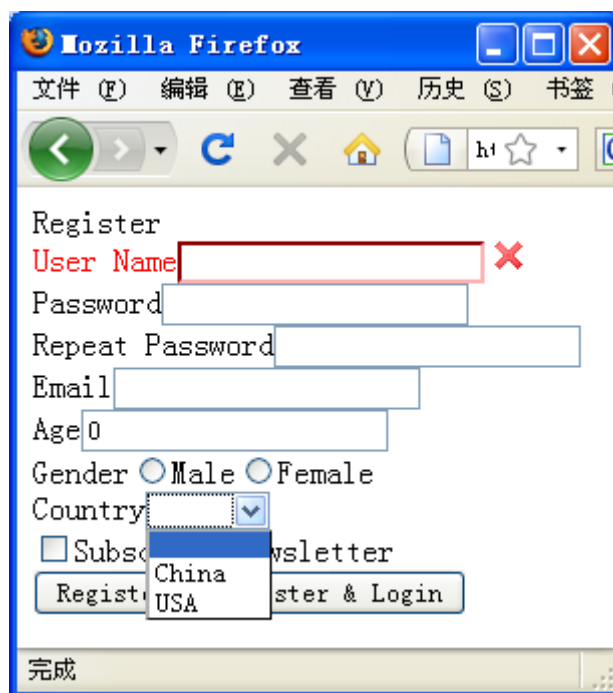


(图 5-4)

在 `WEB-INF` 目录下新建 `app_en.properties`:

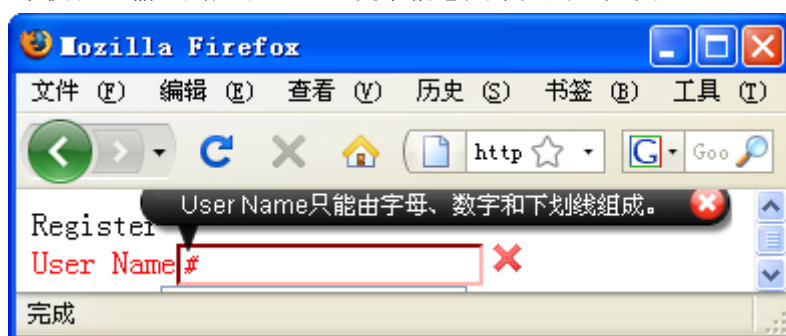
```
Country.CHINA=China
Country.USA=USA
```

重启服务器，页面效果如图 5-5。



(图 5-5)

再来看表单校验，输入用户名“#”，提示信息是中文的，如图 5-6。



(图 5-6)

修改 Start_en.properties:

```
heading=Register
userName-label=User Name
password-label=Password
password2-label=Repeat Password
email-label=Email
age-label=Age
gender-label=Gender
male-label=Male
female-label=Female
country-label=Country
subscribe-label=Subscribe newsletter
register=Register
registerAndLogin=Register & Login
userName-regexp-message=%2$s may only contain letters, numbers and und
```

```
erscores.
```

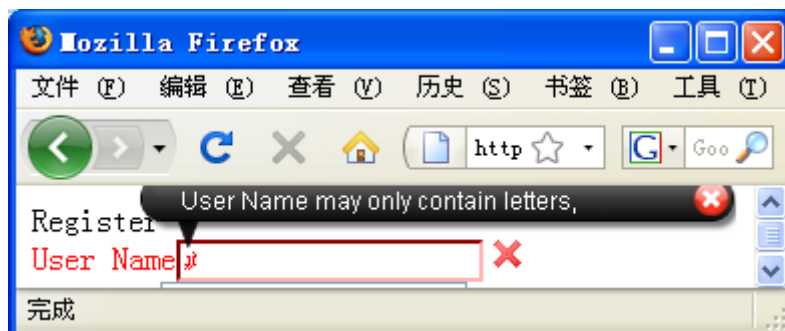
修改 Start_zh_CN.properties:

```
heading=注册新帐号
userName-label=用户名
password-label=密码
password2-label=重复密码
email-label=电子邮件
age-label=年龄
gender-label=性别
male-label=男
female-label=女
country-label=国家
subscribe-label=预订邮件
register=注册
registerAndLogin=注册并登录
userName-regexp-message=%2$s只能由字母、数字和下划线组成。
```

然后把 Start.properties 中的 “userName-regexp-message” 去掉:

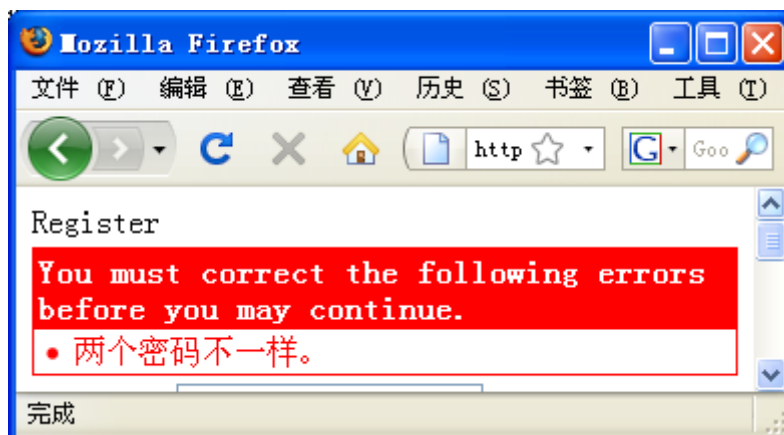
```
#userName-regexp-message=%2$s只能由字母、数字和下划线组成。
userName-regexp=[a-zA-Z0-9_]*$
```

刷新页面，现在能正确显示英文提示信息了，如图 5-7。



(图 5-7)

当两个密码不一样，页面效果如图 5-8。



(图 5-8)

对两个密码的一致性校验是在 **Start** 页面类的 `onValidateForm()`方法中进行的。

修改 **Start** 类:

```
public class Start {
    ...

    void onValidateForm() {
        if (password2 != null && !password2.equals(user.getPassword()))
        {
            regForm.recordError(password2Field, messages
                .get("differ-passwords"));
        }
    }

    void onSuccess() {
        System.out.println(user);
        System.out.println("password2:" + password2);
        System.out.println("login:" + login);
    }
}
```

我们用 **Messages** 从 **properties** 文件获取本地化信息的。这里我们用 `Messages#get()`方法。如果信息带参数，可用 `Messages#format()`方法，`Messages#format()`方法的使用可参考 `java.util.Formatter`。

修改 **Start_en.properties**:

```
heading=Register
userName-label=User Name
password-label=Password
password2-label=Repeat Password
email-label=Email
age-label=Age
gender-label=Gender
male-label=Male
female-label=Female
country-label=Country
subscribe-label=Subscribe newsletter
register=Register
registerAndLogin=Register & Login
userName-regexp-message=%2$s may only contain letters, numbers and underscores.
differ-passwords=Two versions of password do not match.
```

修改 **Start_zh_CN.properties**:

```
heading=注册新帐号
userName-label=用户名
```

```

password-label=密码
password2-label=重复密码
email-label=电子邮件
age-label=年龄
gender-label=性别
male-label=男
female-label=女
country-label=国家
subscribe-label=预订邮件
register=注册
registerAndLogin=注册并登录
userName-regexp-message=%2$s只能由字母、数字和下划线组成。
different-passwords=两个密码不一样。

```

刷新页面，页面效果如图 5-9。



(图 5-9)

本地化整个模板

有些页面或者组件的模板包含很多文字而较少其它东西，把文字全部放在 `properties` 文件中可能显得太麻烦，对此我们可以本地化整个模板。

举个简单的例子，一个 `Localization` 页面包括下面三个文件：

`Localization` 类：

```

public class Localization {
}

```

`Localization_en.tml`：

```

<html xmlns:t="http://tapestry.apache.org/schema/tapestry_5_0_0.xsd">
  <div>Localize template.</div>
</html>

```

`Localization_zh_CN.tml`：

```

<html xmlns:t="http://tapestry.apache.org/schema/tapestry_5_0_0.xsd">
  <div>本地化模板.</div>

```

```
</html>
```

这样我们就为不同语言的用户提供了不同的模板（Localization_en.tml 和 Localization_zh_CN.tml），而不需要把本地化信息都放到 properties 文件中了。

本地化 Asset

Asset 就是页面中需要浏览器额外下载的文件，如图片、css 文件和 javascript 文件等。这些也可以进行本地化。

接下来，我们要在 Start 页面的最上方放一张图片，这张图片也是需要本地化的。

首先，我们新建一个包“example.register.img”，在其中放两张图片，register_en.gif 如图 5-10 所示，register_zh_CN.gif 如图 5-11 所示。

Register

（图 5-10）

注册

（图 5-11）

然后修改 Start.tml:

```
<html xmlns:t="http://tapestry.apache.org/schema/tapestry_5_1_0.xsd">
  <div>
    
  </div>
  <div>${message:heading}</div>
  ...
</html>
```

这里的绑定表达式的前缀是“asset”，后面可以用相对路径。英文页面效果如图 5-12。



（图 5-12）

“asset”前缀默认从 classpath 获得 asset，我们还可以把 asset 放在 web context 中，然后通过“context”前缀获得 asset，例如“\${asset:context:img/register.gif}”或者“\${context:img/register.gif}”。

在 WebContent 目录下建一个子目录 css，在 css 目录新建以下两个 css 文件：

main_en.css 代码如下：

```
@CHARSET "UTF-8";
```

```
div label:first-child {  
    width: 140px;  
    float: left;  
    text-align: right;  
}  
  
div label:first-child:after {  
    content: ":";  
}
```

main_zh_CN.css 代码如下:

```
@CHARSET "UTF-8";  
  
div label:first-child {  
    width: 140px;  
    float: left;  
    text-align: right;  
}  
  
div label:first-child:after {  
    content: "：";  
}
```

注意这两个 css 文件中的红色部分，一个是英文冒号，一个是中文冒号。

修改 Start 类，为 Start 页面加上 css。

```
@IncludeStylesheet("context:css/main.css")  
public class Start {  
    ...  
}
```

英文页面效果如图 5-13。



(图 5-13)

切换 Locale

最后，我们要在页面上方显示超链接，点击超链接可以切换 Locale。我们把它做成一个 LocaleBar 组件。

本章开头我们在 AppModule 类里设置了 `tapestry.supported-locales` (“en,zh_CN”), 现在我们要在 LocaleBar 组件类中读取这个参数，根据这个参数生成相应数量的超链接。

在 “example.register.components” 包中新建 LocaleBar 类：

```
public class LocaleBar {
    @Inject
    @Service("ApplicationDefaults")
    private SymbolProvider symbolProvider;

    public String[] getLocales() {
        String symbol = symbolProvider
            .valueForSymbol(SymbolConstants.SUPPORTED_LOCALES);
        return symbol.split(",");
    }
}
```

在 “example.register.components” 包中新建 LocaleBar.tml：

```
<t:container xmlns:t="http://tapestry.apache.org/schema/tapestry_5_1_0.xsd">
    <t:loop source="locales" value="var:locale">
        >
            <t:actionlink t:id="switch" context="var:locale">${var:locale}</t:actionlink>
        </t:loop>
    </t:container>
```

```

</t:loop>
</t:container>

```

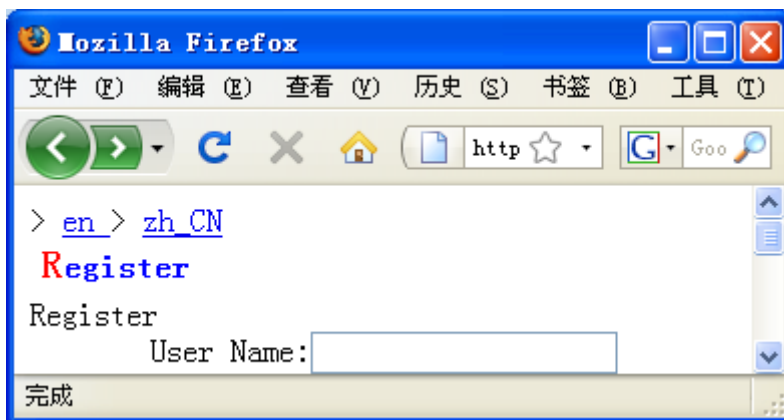
然后在 **Start.tml** 中使用 **LocaleBar** 组件:

```

<html xmlns:t="http://tapestry.apache.org/schema/tapestry_5_1_0.xsd">
  <div>
    <t:localebar />
  </div>
  <div>
    
  </div>
  ...
</html>

```

重启服务器，**Start** 页面效果如图 5-14。



(图 5-14)

超链接的文字应该改成适合向用户显示的语言环境名，而且与自身 **Locale** 相对应。

修改 **LocaleBar** 类:

```

public class LocaleBar {
    @Property
    private String locale;
    @Inject
    @Service("ApplicationDefaults")
    private SymbolProvider symbolProvider;

    public String[] getLocales() {
        String symbol = symbolProvider
            .valueForSymbol(SymbolConstants.SUPPORTED_LOCALES);
        return symbol.split(",");
    }

    public String getLocaleName() {
        Locale l = toLocale(locale);
    }
}

```

```

        return l.getDisplayName(l);
    }

    private Locale toLocale(String shortName) {
        String[] result = shortName.split("_");
        if (result.length == 1) {
            return new Locale(result[0]);
        } else {
            return new Locale(result[0], result[1]);
        }
    }
}

```

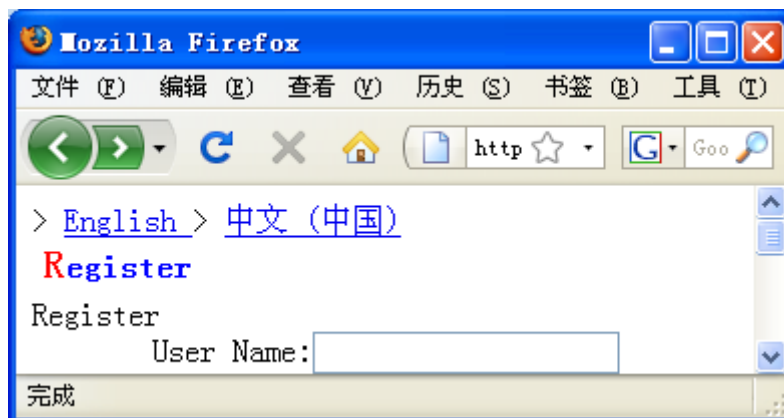
修改 LocaleBar.tml:

```

<t:container xmlns:t="http://tapestry.apache.org/schema/tapestry_5_1_
0.xsd">
    <t:loop source="locales" value="locale">
        >
            <t:actionlink t:id="switch" context="locale">${localeName}
            </t:actionlink>
        </t:loop>
    </t:container>

```

刷新页面，页面效果如图 5-15。



(图 5-15)

当前 Locale 对应的超链接不应该显示。

修改 LocaleBar 类:

```

public class LocaleBar {
    @Property
    private String locale;
    @Inject
    private Locale currentLocale;
    @Inject

```

```

@Service("ApplicationDefaults")
private SymbolProvider symbolProvider;

public String[] getLocales() {
    String symbol = symbolProvider
        .valueForSymbol(SymbolConstants.SUPPORTED_LOCALES);
    return symbol.split(",");
}

public String getLocaleName() {
    Locale l = toLocale(locale);
    return l.getDisplayName(l);
}

public boolean isCurrentLocale() {
    return toLocale(locale).equals(currentLocale);
}

private Locale toLocale(String shortName) {
    String[] result = shortName.split("_");
    if (result.length == 1) {
        return new Locale(result[0]);
    } else {
        return new Locale(result[0], result[1]);
    }
}
}

```

要获取当前 **Locale**，只需注入一个 **java.util.Locale** 就行了。

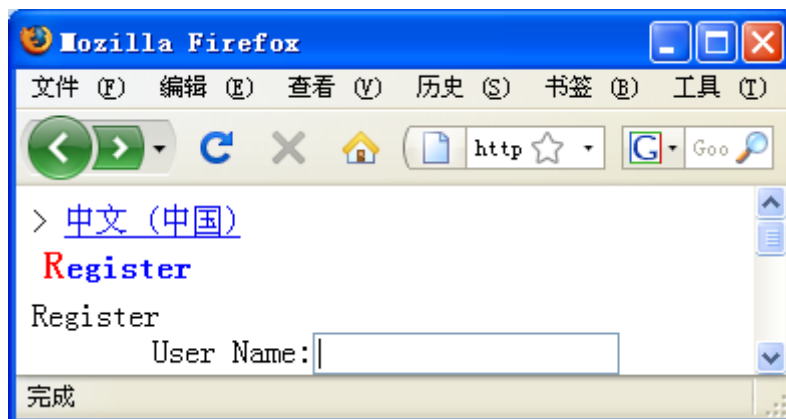
修改 **LocaleBar.tml**：

```

<t:container xmlns:t="http://tapestry.apache.org/schema/tapestry_5_1_
0.xsd">
    <t:loop source="locales" value="locale">
        <t:unless test="currentLocale">
            >
                <t:actionlink t:id="switch" context="locale">${localeName}
                </t:actionlink>
            </t:unless>
        </t:loop>
    </t:container>

```

页面效果如图 5-16。



(图 5-16)

最后，当点击超链接，切换 Locale。

修改 LocaleBar 类：

```
public class LocaleBar {
    @Property
    private String locale;
    @Inject
    private Locale currentLocale;
    @Inject
    @Service("ApplicationDefaults")
    private SymbolProvider symbolProvider;
    @Inject
    private PersistentLocale persistentLocale;

    public String[] getLocales() {
        String symbol = symbolProvider
            .valueForSymbol(SymbolConstants.SUPPORTED_LOCALES);
        return symbol.split(",");
    }

    public String getLocaleName() {
        Locale l = toLocale(locale);
        return l.getDisplayName(l);
    }

    public boolean isCurrentLocale() {
        return toLocale(locale).equals(currentLocale);
    }

    void onActionFromSwitch(String shortName) {
        persistentLocale.set(toLocale(shortName));
    }
}
```

```
private Locale toLocale(String shortName) {  
    String[] result = shortName.split("_");  
    if (result.length == 1) {  
        return new Locale(result[0]);  
    } else {  
        return new Locale(result[0], result[1]);  
    }  
}
```

现在，点击超链接就能切换 **Locale** 了。

第六章 Ajax

本章主要讲解如何在 Tapestry 应用中使用 Javascript 以及 Tapestry 对 Ajax 的支持。

新建一个动态 Web 工程“Ajax”，加入 jar 文件，修改 web.xml 成如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
    <display-name>Ajax</display-name>
    <context-param>
        <param-name>tapestry.app-package</param-name>
        <param-value>example.ajax</param-value>
    </context-param>
    <filter>
        <filter-name>app</filter-name>
        <filter-class>org.apache.tapestry5.TapestryFilter</filter-class>
    </filter>
    <filter-mapping>
        <filter-name>app</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
</web-app>
```

首先我们要创建两个辅助类。

Entry 类表示电话本中的一条记录，代码如下：

```
public class Entry {
    private String name;
    private String number;

    public Entry() {
    }

    public Entry(String name, String number) {
        this.name = name;
        this.number = number;
    }

    public String getName() {
```

```

        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getNumber() {
        return number;
    }

    public void setNumber(String number) {
        this.number = number;
    }
}

```

PhoneBook 类表示电话本，代码如下：

```

public class PhoneBook {
    public List<Entry> entries;

    public PhoneBook() {
        entries = new ArrayList<Entry>();
        entries.add(new Entry("XueBaoChai", "11111111"));
        entries.add(new Entry("LinDaiYu", "22222222"));
        entries.add(new Entry("JiaYingChun", "33333333"));
        entries.add(new Entry("JiaTanChun", "44444444"));
        entries.add(new Entry("JiaXiChun", "55555555"));
        entries.add(new Entry("ShiXiangYun", "66666666"));
        entries.add(new Entry("LiWan", "77777777"));
    }

    public List<String> selectNamesByKey(String key) {
        List<String> matches = new ArrayList<String>();
        for (Iterator<Entry> iter = entries.iterator(); iter.hasNext
        ());) {
            Entry entry = iter.next();
            if (entry.getName().toLowerCase().startsWith(key.toLowerCase
            se())) {
                matches.add(entry.getName());
            }
        }
        return matches;
    }
}

```

```

public Entry selectByName(String name) {
    for (Iterator<Entry> iter = entries.iterator(); iter.hasNext
());) {
        Entry entry = iter.next();
        if (entry.getName().equals(name)) {
            return entry;
        }
    }
    return null;
}

public void update(String key, Entry newEntry) {
    for (Iterator<Entry> iter = entries.iterator(); iter.hasNext
());) {
        Entry e = iter.next();
        if (e.getName().equals(key)) {
            e.setName(newEntry.getName());
            e.setNumber(newEntry.getNumber());
        }
    }
}
}

```

接下来我们做一个 **Start** 页面，页面上有一个文本输入框和一个超链接，在文本输入框输入一个名字，点击超链接查看电话本中有没有这个名字。

在 “example.ajax.pages” 包中新建 **Start.tml**：

```

<html xmlns:t="http://tapestry.apache.org/schema/tapestry_5_1_0.xsd">
  <div>
    <input type="text" id="name" />
    <a t:type="any" t:id="checker" href="#">查询</a>
  </div>
</html>

```

在 **Start.tml** 中我们用一个 **Any** 组件来生成超链接。

要注意这里用 **Html** 标签 **<a>** 来定义组件，在生成 **Html** 代码时，标签 **<a>** 和里面的非正式参数都会被输出。

在 “example.ajax.pages” 包中新建 **Start** 类：

```

public class Start {
    @SessionState
    private PhoneBook phoneBook;

    StreamResponse onActionFromChecker(String name) {
        String result;
        if (phoneBook.selectByName(name) != null) {

```

```

        result = "<div style='color:green;'>电话本中有此联系人。</div>";
    } else {
        result = "<div style='color:red;'>电话本中没有此联系人。</div>";
    }
    return new TextStreamResponse("text/html", result);
}
}

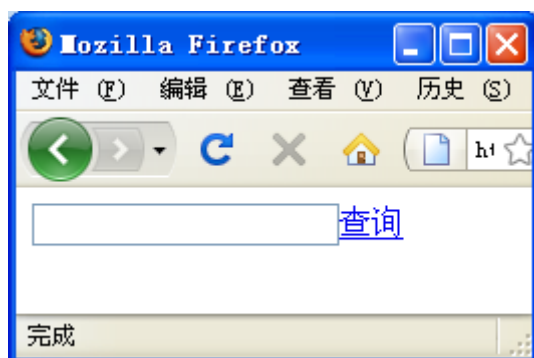
```

Any 组件的 id 为“checker”，在 Start 类中与之对应的事件方法是 onActionFromChecker(String name)，我们将在此方法中检查电话本中是否有某个联系人。此方法有一个参数 name，表示姓名，我们将通过请求类似 <http://localhost:8080/Ajax/Start.checker/name> 的 URL 来触发 onActionFromChecker(String name) 方法。

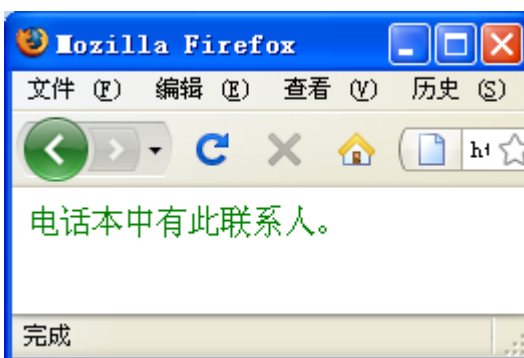
onActionFromChecker(String name) 方法将返回 org.apache.tapestry5.StreamResponse，StreamResponse 用于直接向客户端输出流，不会有重定向发生。

把“Ajax”加入 Tomcat 服务器，重启服务器。

在浏览器中打开 <http://localhost:8080/Ajax/>，页面效果如图 6-1，打开 <http://localhost:8080/Ajax/Start.checker/XueBaoChai>，页面效果如图 6-2。



(图 6-1)



(图 6-2)

添加 Javascript

现在我们来实现 javascript 部分。当我们点击 Start 页面中的查询超链接，发送 Ajax 请求到 <http://localhost:8080/Ajax/Start.checker/name>，把返回的内容显示在 Start 页面的下方。

修改 Start 类：

```

public class Start {
    @SessionState
    private PhoneBook phoneBook;
    @Environmental
    private RenderSupport renderSupport;

    void setupRender() {

```

```

        renderSupport.addScript("check = function(source, result) {"
            + "    new Ajax.Request('Start.checker/' + $F(source), {"
            + "        method: 'get',"
            + "        onSuccess: function(transport) {"
            + "            $(result).update(transport.responseText);"
            + "        }"
            + "    });"
            + "});"
    );

    StreamResponse onActionFromChecker(String name) {
        String result;
        if (phoneBook.selectByName(name) != null) {
            result = "<div style='color:green;'>电话本中有此联系人。</div>";
        } else {
            result = "<div style='color:red;'>电话本中没有此联系人。</div>";
        }
        return new TextStreamResponse("text/html", result);
    }
}

```

`RenderSupport#addScript(String format, Object... arguments)`方法用于往页面中添加 `javascript` 代码，此方法调用 `String.format(String format, Object... arguments)`方法生成格式化字符串，其中的 `arguments` 用于替换 `format` 中的参数（例如“%s”）。

我们把此类操作放在 `SetupRender` 阶段执行。

当点击查询超链接，调用 `javascript` 函数 `check(source, result)`。

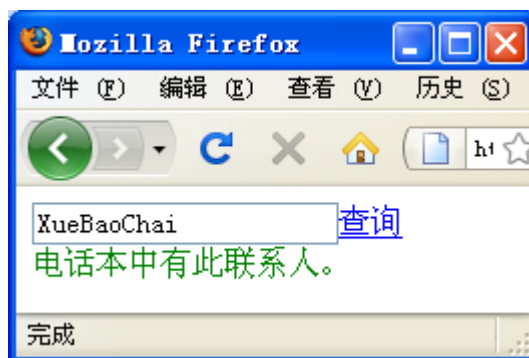
修改 `Start.tml`：

```

<html xmlns:t="http://tapestry.apache.org/schema/tapestry_5_1_0.xsd">
    <div>
        <input type="text" id="name" />
        <a t:type="any" t:id="checker" href="javascript:check('name','
result');">查询</a>
        <div id="result" />
    </div>
</html>

```

刷新 `Start` 页面，在文本框输入“`XueBaoChai`”，点击查询，页面效果如图 6-3。

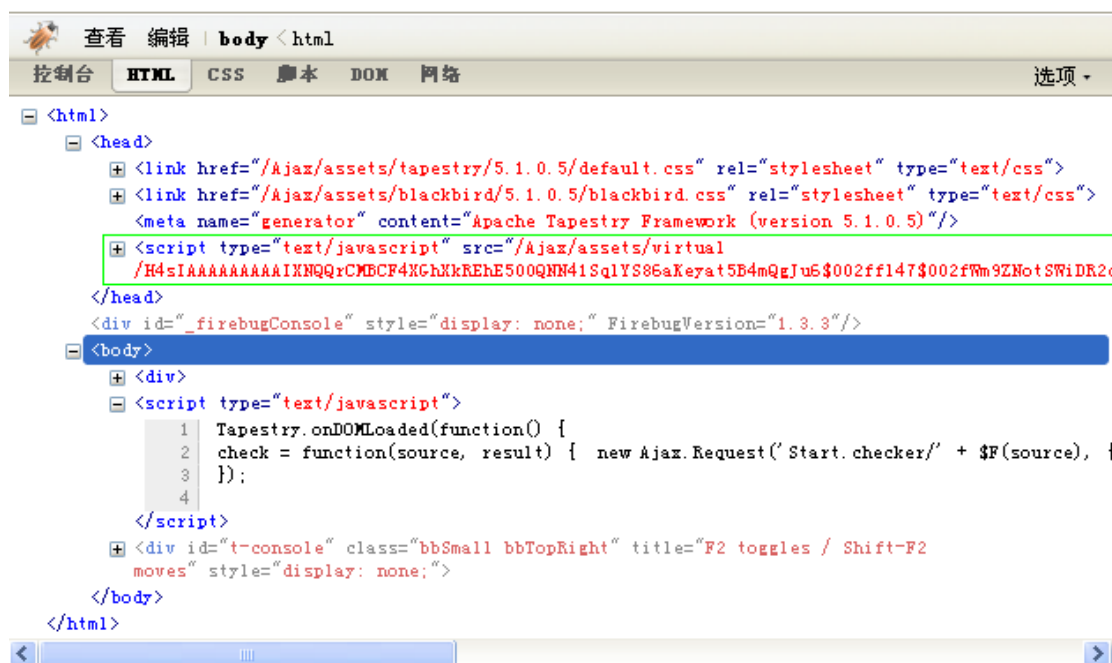


(图 6-3)

基础 Javascript 库

回头看 javascript 函数 `check(source, result)`，它并非是原始的 javascript，实际上它是基于 Prototype 的，也就是说，只有在页面引入了 Prototype 库的情况下才能正确执行这个函数。但是我们无需自己为页面引入 Prototype 库，因为 Tapestry 已经自动为我们完成了这个工作。

图 6-4 是使用 Firebug 看到的 Start 页面的 Html 源代码。




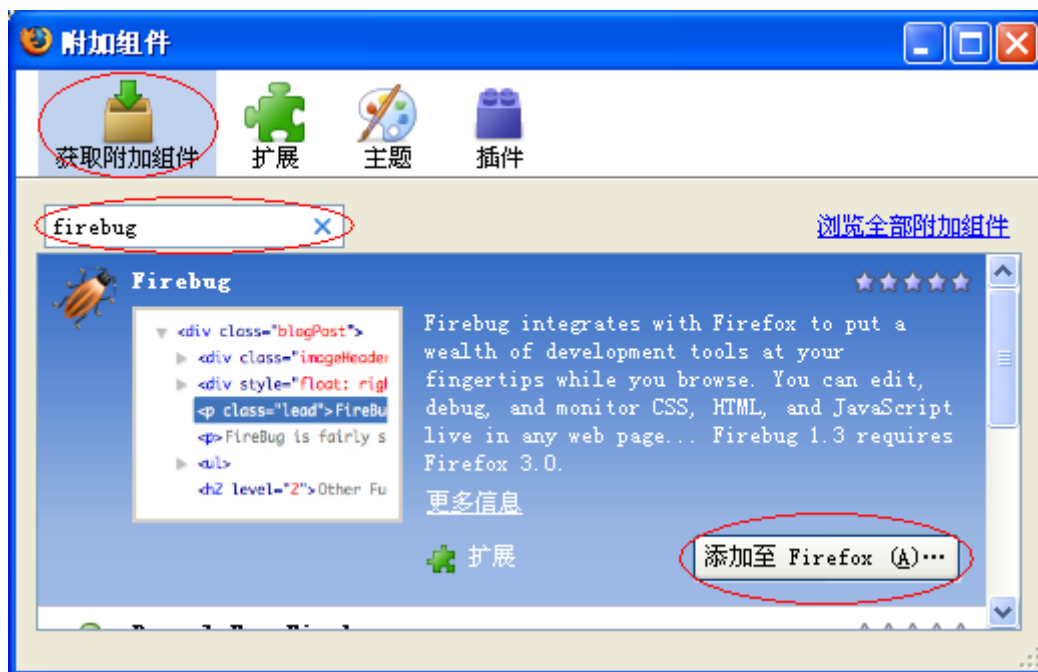
(图 6-4)

图 6-4 中绿色方框部分就是 Tapestry 自动引入的 javascript 库，它实际上是多个 javascript 库的联合体，包括 Tapestry 自己的 javascript 库以及 [Prototype](#)、[Scriptaculous](#) 等 Tapestry 所依赖的第三方 javascript 库，把它们放到一个库中，可以减少请求次数。另外，如果客户端浏览器支持 gzip 压缩，这个库也将会被压缩。

安装 Firebug

Firebug 是 Firefox 浏览器的附加组件，可以让我们更加方便的查看 Html 源代码、调试 javascript 等等，是一个很有用的工具。

要安装 Firebug，点击 Firefox 菜单【工具】【附加组件】，在“获取附加组件”面板中查找“firebug”，如图 6-5。点击“添加至 Firefox”按钮，安装完成后重启 Firefox，在 Firefox 状态栏的右边就能找到 Firebug 图标，点击此图标就可以打开 Firebug。



(图 6-5)

添加 Javascript 库

有时候，我们要把 javascript 放在外部文件中，这样更便于管理和维护，也便于代码重用。

我们把 Start 页面中的 javascript 代码放到外部文件中。

在 WebContent 目录下新建子目录 js，在 js 下新建文件 main.js，其内容如下：

```
check = function(source, result) {
    new Ajax.Request('Start.checker/' + $F(source), {
        method : 'get',
        onSuccess : function(transport) {
            $(result).update(transport.responseText);
        }
    });
};
```

修改 Start 类：

```

public class Start {
    @SessionState
    private PhoneBook phoneBook;
    @Environmental
    private RenderSupport renderSupport;
    @Inject
    @Path("context:js/main.js")
    private Asset mainJs;

    void setupRender() {
        renderSupport.addScriptLink(mainJs);
        // renderSupport.addScript("check = function(source, result) {"
        // + "    new Ajax.Request('Start.checker/' + $F(source), {"
        // + "        method: 'get',"
        // + "        onSuccess: function(transport) {"
        // + "            $(result).update(transport.responseText);"
        // + "        }"
        // + "    });"
        // + "});");
    }

    StreamResponse onActionFromChecker(String name) {
        String result;
        if (phoneBook.selectByName(name) != null) {
            result = "<div style='color:green;'>电话本中有此联系人。</div>";
        } else {
            result = "<div style='color:red;'>电话本中没有此联系人。</div>";
        }
        return new TextStreamResponse("text/html", result);
    }
}

```

`RenderSupport#addScriptLink(Asset... scriptAssets)`方法用于往页面中添加 javascript 库。

刷新页面，页面运行效果和刚才还是一样的。查看 **Html 源代码**，你会看到 `main.js` 已经和 **Tapestry** 的那些基础 javascript 库一起被打包成一个库文件了。

还有一种更简单的添加 javascript 库的方法，就是使用 `@IncludeJavaScriptLibrary` 注释。

修改 **Start** 类：

```

@IncludeJavaScriptLibrary("context:js/main.js")
public class Start {
    @SessionState
    private PhoneBook phoneBook;

```

```

// @Environmental
// private RenderSupport renderSupport;
// @Inject
// @Path("context:js/main.js")
// private Asset mainJs;

// void setupRender() {
// renderSupport.addScriptLink(mainJs);
// }

StreamResponse onActionFromChecker(String name) {
    String result;
    if (phoneBook.selectByName(name) != null) {
        result = "<div style='color:green;'>电话本中有此联系人。</div>";
    } else {
        result = "<div style='color:red;'>电话本中没有此联系人。</div>";
    }
    return new TextStreamResponse("text/html", result);
}
}

```

多次添加同一个 javascript 库不会造成重复，后来者会被简单忽略掉，因此每个页面/组件都可以添加它自己所需的 javascript 库而不必考虑其它页面/组件是否已经添加过同一个 javascript 库。

Autocomplete Mixin

我们经常会在一些网页上看到这样的功能：在文本框中输入一些字符，文本框下方就会出现一系列候选项供我们选择。

Tapestry 5 内置了一个 Mixin 叫做 Autocomplete，可以让我们方便的实现这样的功能。

我们来新建一个 Edit 页面，页面里有一个表单，表单里有一个文本框，我们在这个文本框上使用 Autocomplete。

在“example.ajax.pages”包中新建 Edit.tml:

```

<html xmlns:t="http://tapestry.apache.org/schema/tapestry_5_1_0.xsd">
  <t:form t:id="loadForm">
    <t:textfield t:id="name" t:mixins="autocomplete" />
    <t:submit value="确定" />
  </t:form>
</html>

```

AutoComplete 的事件名称是“providecompletions”，其事件方法返回数组、List 或者单

个对象，如果对象不是 `String` 类型，则用 `toString()` 方法转换成 `String`。

在 “`example.ajax.pages`” 包中新建 `Edit` 类：

```
public class Edit {
    @Property
    private String name;
    @SessionState
    private PhoneBook phoneBook;

    List<String> onProvideCompletionsFromName(String key) {
        return phoneBook.selectNamesByKey(key);
    }
}
```

重启服务器。打开 <http://localhost:8080/Ajax/Edit>，在文本框中输入“l”，文本框下方会显示所有以“l”开头的姓名，如图 6-6。



(图 6-6)

Zone 组件

Zone 组件可以让我们方便的通过 Ajax 动态更新客户端。

下面我们要实现的功能是：输入一个姓名，点击确定按钮，则显示此人的详细信息。

修改 `Edit.tml`：

```
<html xmlns:t="http://tapestry.apache.org/schema/tapestry_5_1_0.xsd">
  <t:form t:id="loadForm" zone="theZone">
    <t:textfield t:id="name" t:mixins="autocomplete" />
    <t:submit value="确定" />
  </t:form>
  <t:zone t:id="theZone" />
  <t:block t:id="viewBlock">
    <div>${entry.name}</div>
    <div>${entry.number}</div>
  </t:block>
```

</html>

修改 Edit 类:

```

public class Edit {
    @Property
    private String name;
    @Property(write = false)
    private Entry entry;
    @Inject
    private Block viewBlock;
    @SessionState
    private PhoneBook phoneBook;

    List<String> onProvideCompletionsFromName(String key) {
        return phoneBook.selectNamesByKey(key);
    }

    Block onSuccessFromLoadForm() {
        entry = phoneBook.selectByName(name);
        return viewBlock;
    }
}

```

Form 组件有一个 zone 参数，我们把它设置为“theZone”，当我们提交表单，就会有一个 Ajax 请求被发送到服务器，触发 Edit 页面实例的 onSuccessFromLoadForm()方法，此方法返回一个 Block（其 ID 为“viewBlock”），这个 Block 的内容直接被发送给客户端用来更新“theZone”。

在这个过程中是没有重定向发生的。类似的，如果 onSuccessFromLoadForm()方法的返回值是一个组件，也不会有重定向发生，但是如果返回值是页面名、页面 class、页面实例等，则仍然会有重定向发生。

刷新页面，输入“LinDaiYu”，按确定按钮，页面效果如图 6-7。



(图 6-7)

接下来我们在详细信息后面增加一个超链接，点击超链接，进入编辑状态。

修改 Edit.tml:

```
<html xmlns:t="http://tapestry.apache.org/schema/tapestry_5_1_0.xsd">
  <t:form t:id="loadForm" zone="theZone">
    <t:textfield t:id="name" t:mixins="autocomplete" />
    <t:submit value="确定" />
  </t:form>
  <t:zone t:id="theZone" />
  <t:block t:id="viewBlock">
    <div>${entry.name}</div>
    <div>${entry.number}</div>
    <div>
      <t:actionlink t:id="edit" context="entry.name" zone="theZone">修改</t:actionlink>
    </div>
  </t:block>
  <t:block>
    <t:form t:id="editForm">
      <t:textfield t:value="entry.name" />
      <t:textfield t:value="entry.number" />
      <t:submit value="保存" />
    </t:form>
  </t:block>
</html>
```

与 Form 组件类似，ActionLink 组件也有 zone 参数。

修改 Edit 类:

```
public class Edit {
    @Property
    private String name;
    @Property(write = false)
    private Entry entry;
    @Inject
    private Block viewBlock;
    @Component
    private Form editForm;
    @SessionState
    private PhoneBook phoneBook;

    List<String> onProvideCompletionsFromName(String key) {
        return phoneBook.selectNamesByKey(key);
    }
}
```

```

Block onSuccessFromLoadForm() {
    entry = phoneBook.selectByName(name);
    return viewBlock;
}

Form onActionFromEdit(String aName) {
    entry = phoneBook.selectByName(aName);
    return editForm;
}
}

```

刷新 Edit 页面，输入“LinDaiYu”，点击确定，页面效果如图 6-8，点击修改，页面效果如图 6-9



(图 6-8)



(图 6-9)

如果在图 6-9 修改姓名和电话，点击保存，页面将再次变成如图 6-8。

修改 Edit.tml:

```

<html xmlns:t="http://tapestry.apache.org/schema/tapestry_5_1_0.xsd">
  <t:form t:id="loadForm" zone="theZone">
    <t:textfield t:id="name" t:mixins="autocomplete" />
    <t:submit value="确定" />
  </t:form>
  <t:zone t:id="theZone" />
  <t:block t:id="viewBlock">
    <div>${entry.name}</div>
    <div>${entry.number}</div>
    <div>
      <t:actionlink t:id="edit" context="entry.name" zone="theZone">修改
    </t:actionlink>
    </div>
  </t:block>
  <t:block>
    <t:form t:id="editForm" zone="theZone">

```

```

        <t:hidden value="oldName" />
        <t:textfield t:value="entry.name" />
        <t:textfield t:value="entry.number" />
        <t:submit value="保存" />
    </t:form>
</t:block>
</html>

```

在 `editForm` 表单中，我们增加了一个 `Hidden` 组件用于保存修改之前的姓名。

修改 `Edit` 类：

```

public class Edit {
    @Property
    private String name;
    @Property
    private String oldName;
    @Property(write = false)
    private Entry entry;
    @Inject
    private Block viewBlock;
    @Component
    private Form editForm;
    @SessionState
    private PhoneBook phoneBook;

    List<String> onProvideCompletionsFromName(String key) {
        return phoneBook.selectNamesByKey(key);
    }

    Block onSuccessFromLoadForm() {
        entry = phoneBook.selectByName(name);
        return viewBlock;
    }

    Form onActionFromEdit(String aName) {
        oldName = aName;
        entry = phoneBook.selectByName(aName);
        return editForm;
    }

    Block onSuccessFromEditForm() {
        phoneBook.update(oldName, entry);
        return viewBlock;
    }
}

```


刷新页面，修改姓名和电话，如图 6-11，点击保存，页面没发生任何变化。



(图 6-11)

当我们点击保存，会向服务器发送一个 Ajax 请求，现在我们想知道的是到底这个请求返回了什么内容，导致我们的页面不能正确更新。

打开 Firebug，选择控制台面板。

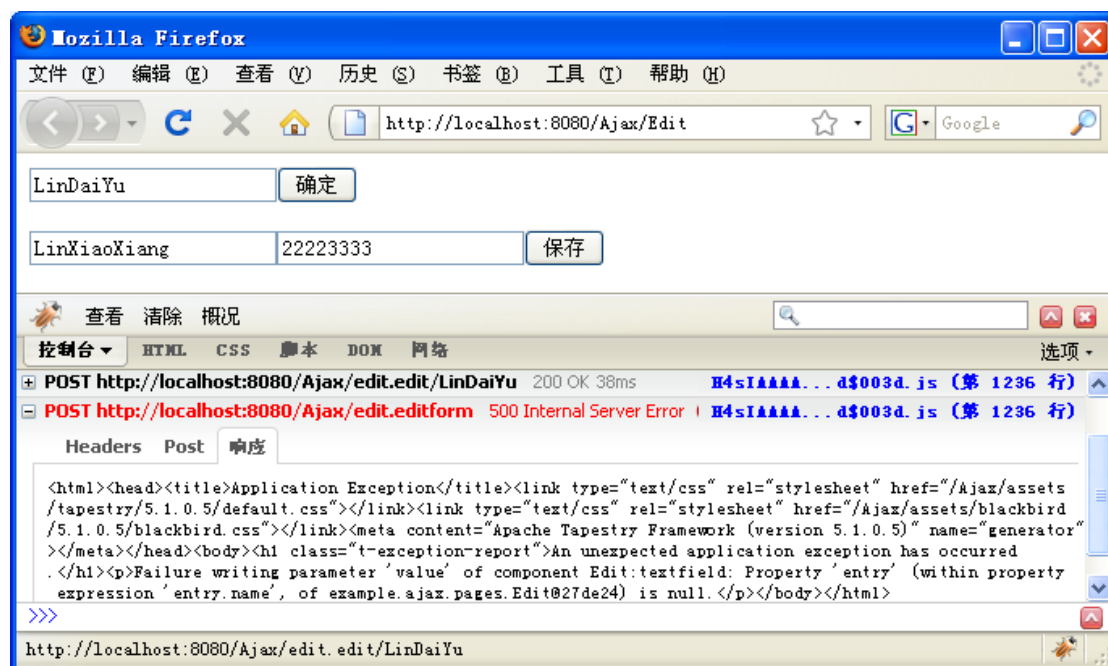
如果控制台被禁用，如图 6-12，选中所有复选框，点击下方的按钮启用控制台。



(图 6-12)

控制台启用后，就能显示每个 Ajax 请求的详细信息了。

现在，再做一次保存操作，在控制台可以看到异常信息，如图 6-13。



(图 6-13)

引起异常的原因是 entry 属性为 null。

修改 Edit 类：

```
public class Edit {
    @Property
    private String name;
    @Property
    private String oldName;
    @Property(write = false)
    private Entry entry = new Entry();
    @Inject
    private Block viewBlock;
    ...
}
```

刷新页面，修改姓名和电话，如图 6-14，点击保存，页面效果如图 6-15。



(图 6-14)



(图 6-15)

更新多个 Zone

最后，我们再对 Edit 页面做一点改进：当点击图 6-14 的保存按钮，图 6-15 上方的文本框中的内容也变成“LinXiaoXiang”。

修改 Edit.tml:

```
<html xmlns:t="http://tapestry.apache.org/schema/tapestry_5_1_0.xsd">
  <t:zone t:id="loadZone">
    <t:form t:id="loadForm" zone="theZone">
      <t:textfield t:id="name" t:mixins="autocomplete" />
      <t:submit value="确定" />
    </t:form>
  </t:zone>
  <t:zone t:id="theZone" />
  ...
</html>
```

修改 Edit 类:

```
public class Edit {
    @Property
    private String name;
    @Property
    private String oldName;
    @Property(write = false)
    private Entry entry = new Entry();
    @Inject
    private Block viewBlock;
    @Component
    private Form editForm;
    @Component
    private Zone loadZone;
    @SessionState
    private PhoneBook phoneBook;

    List<String> onProvideCompletionsFromName(String key) {
        return phoneBook.selectNamesByKey(key);
    }

    Block onSuccessFromLoadForm() {
        entry = phoneBook.selectByName(name);
        return viewBlock;
    }

    Form onActionFromEdit(String aName) {
        oldName = aName;
    }
}
```

```
entry = phoneBook.selectByName(aName);  
return editForm;  
}  
  
Object onSuccessFromEditForm() {  
    phoneBook.update(oldName, entry);  
    // return viewBlock;  
    name = entry.getName();  
    return new MultiZoneUpdate("theZone", viewBlock).add("loadZone",  
        loadZone.getBody());  
}
```

再做一次前面的操作，修改姓名和电话，如图 6-16，点击保存，页面效果如图 6-17。



(图 6-16)



(图 6-17)

第七章 集成 Spring

本章简单介绍 Tapestry 如何集成 Spring。

Tapestry 集成 Spring 后，就可以在页面/组件或者 Service 中注入 Spring Bean。

本章的例子是一个电话本程序。在这个例子中，我们首先使用 Tapestry 的 IoC 功能，让你对 Tapestry 的 IoC 以及 Service 的概念有基本的了解，然后讲解 Tapestry 如何集成 Spring。

本章的例子需要用到数据库。

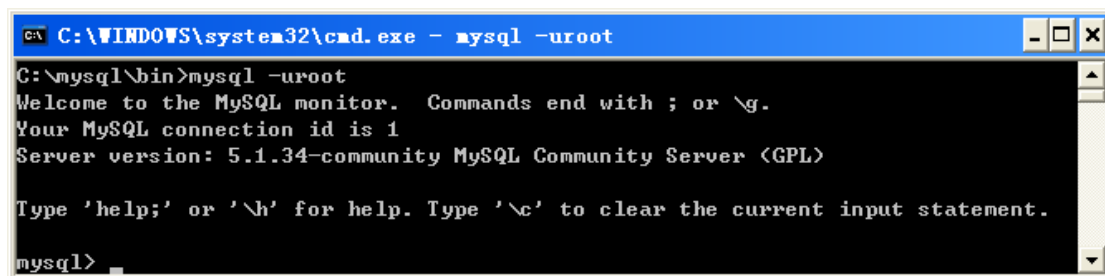
从 <http://dev.mysql.com/downloads/mysql/5.1.html> 下载 mysql-noinstall-5.1.34-win32.zip，解压到 C:\mysql。

打开一个命令窗口，进入 C:\mysql\bin 目录，执行命令 “mysqld”，如图 7-1，就可以启动 MySQL 了。要停止 MySQL，执行命令 “mysqladmin shutdown -uroot”。



(图 7-1)

打开另一个命令窗口，进入 C:\mysql\bin 目录，执行命令 mysql -uroot，进入 MySQL 命令提示符，如图 7-2。



(图 7-2)

在 mysql 提示符下执行下面 sql 语句：

```
DROP DATABASE IF EXISTS `myexample`;  
CREATE DATABASE `myexample`;  
USE `myexample`;  
CREATE TABLE `phonebook` (  
  `name` VARCHAR(20) NOT NULL,  
  `number` VARCHAR(20) NOT NULL  
);
```

现在，我们已经有了一个数据库 “myexample”，在 “myexample” 中有一张表 “phone book”。

仿照第一章新建动态 Web 工程 “Spring”，加入 jar 文件，把 web.xml 修改成如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
    <display-name>Spring</display-name>
    <context-param>
        <param-name>tapestry.app-package</param-name>
        <param-value>example.spring</param-value>
    </context-param>
    <filter>
        <filter-name>app</filter-name>
        <filter-class>org.apache.tapestry5.TapestryFilter</filter-class>
    </filter>
    <filter-mapping>
        <filter-name>app</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
</web-app>
```

新建 Entry 类：

```
public class Entry {
    private String name;
    private String number;

    public Entry() {
    }

    public Entry(String name, String number) {
        this.name = name;
        this.number = number;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getNumber() {
```

```

        return number;
    }

    public void setNumber(String number) {
        this.number = number;
    }
}

```

在 “example.spring.pages” 包中新建 Start.tml:

```

<html xmlns:t="http://tapestry.apache.org/schema/tapestry_5_1_0.xsd">
  <t:form>
    <div>
      姓名:
      <t:textfield t:value="entry.name" />
    </div>
    <div>
      号码:
      <t:textfield t:value="entry.number" />
    </div>
    <div>
      <t:submit value="添加" />
    </div>
  </t:form>
</html>

```

在 “example.spring.pages” 包中新建 Start 类:

```

public class Start {
    @Property
    private Entry entry = new Entry();

    void onSuccess() throws ClassNotFoundException, SQLException {
        Class.forName("org.gjt.mm.mysql.Driver");
        Connection conn = null;
        PreparedStatement st = null;
        try {
            conn = DriverManager.getConnection(
                "jdbc:mysql://localhost:3306/myexample", "root", "
");
            st = conn
                .prepareStatement("insert into phonebook(name, number) values (?, ?)");
            st.setString(1, entry.getName());
            st.setString(2, entry.getNumber());
            st.execute();
        } finally {

```

```

        st.close();
        conn.close();
    }
}
}

```

注意onSucess()方法直接抛出了ClassNotFouedException和SQLException。事件方法被允许抛出任何异常，Tapestry会捕捉这些异常并显示异常报告页面。因此没必要像下面这样处理异常：

```

void onSuccess() {
    try {
        ...
    } catch (ClassNotFouedException e) {
        throw new RuntimeException(e);
    }
    try {
        ...
    } catch (SQLException e) {
        throw new RuntimeException(e);
    }
}

```

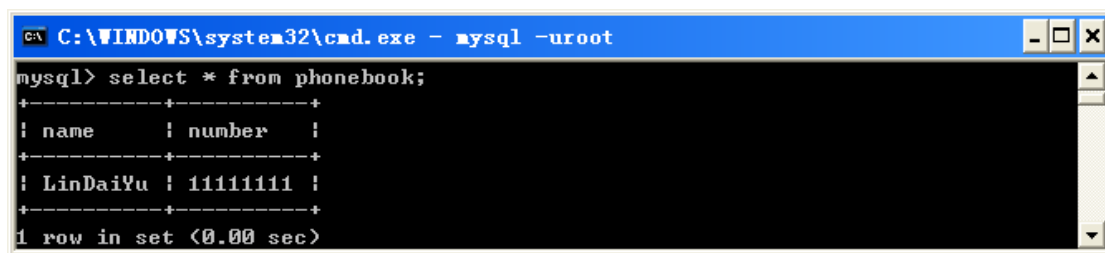
由于要访问数据库，因此在运行程序之前，我们还需要先加入MySQL的驱动。从<http://dev.mysql.com/downloads/connector/j/5.1.html>下载mysql-connector-java-5.1.7.zip，解压后，把mysql-connector-java-5.1.7-bin.jar加入WEB-INF/lib。

把工程“Spring”加入服务器，重启服务器。

打开<http://localhost:8080/Spring/>，输入数据，页面效果如图7-3，点击“添加”，可以从数据库中查询到新增的记录，如图7-4。



(图 7-3)



(图 7-4)

接下来我们要重构前面的代码，把数据访问逻辑放到一个 Service 中。

Tapestry IoC

Tapestry 5 有一套自己的 IoC 实现，其基本单位是 Service。

一个 Service 由一个接口和一个实现类组成。通常一个接口只有一个 Service，但有时也可以有多个 Service（一个接口有多个实现类）。

下面我们来创建一个 Service，用于向数据库中插入数据。

新建 PhoneBookService 接口：

```
public interface PhoneBookService {
    void save(Entry entry);
}
```

新建 PhoneBookServiceImpl 类：

```
public class PhoneBookServiceImpl implements PhoneBookService {
    public void save(Entry entry) {
        try {
            Class.forName("org.gjt.mm.mysql.Driver");
        } catch (ClassNotFoundException e) {
            throw new RuntimeException(e);
        }
        Connection conn = null;
        PreparedStatement st = null;
        try {
            conn = DriverManager.getConnection(
                "jdbc:mysql://localhost:3306/myexample", "root", "
");
            st = conn
                .prepareStatement("insert into phonebook(name, number) values (?, ?)");
            st.setString(1, entry.getName());
            st.setString(2, entry.getNumber());
            st.execute();
        } catch (SQLException e) {
            throw new RuntimeException(e);
        } finally {

```

```

        try {
            st.close();
        } catch (Exception e) {
        }
        try {
            conn.close();
        } catch (Exception e) {
        }
    }
}
}

```

接口和实现类都有了，现在我们只需在 `AppModule.java` 中配置一下，就可以使用这个 `Service` 了。

在 “`example.spring.services`” 包中新建 `AppModule` 类：

```

public class AppModule {
    public static void bind(ServiceBinder binder) {
        binder.bind(PHONEBOOKSERVICE.class, PHONEBOOKSERVICEIMPL.class);
    }
}

```

我们通过 `ServiceBinder#bind()` 方法注册我们的 `Service`，`bind()` 方法有两个参数，第一个是接口，第二个是实现类。如果省略第二个参数，`Tapestry` 默认实现类与接口在同一个包内，其名字为接口名后跟 “`Impl`”，因此上面的代码中第二个参数是可以省略的。

每一个 `Service` 都有一个唯一的 ID，缺省为接口名（比如我们刚才创建的 `Service`，其 ID 就是 “`PhoneBookService`”），我们也可以通过如下代码显式设置 `Service` 的 ID：

```
binder.bind(PHONEBOOKSERVICE.class).withId("MyID");
```

如果我们要在页面/组件类中使用 `PhoneBookService`，可以使用如下代码：

```

@Inject
@Service("PhoneBookService")
private PhoneBookService phoneBookService;

```

其中 `@Service` 注释用来标识 `Service` 的 ID。但是通常我们不用 `@Service` 注释，因为如果我们重构程序，改变了接口名，那么上面这段代码就出问题了。

更简单的用法如下：

```

@Inject
private PhoneBookService phoneBookService;

```

使用这种方法，`Tapestry` 会试图查找一个实现 `PhoneBookService` 接口的 `Service`，如果没有或者有多个 `Service` 实现这个接口，则会失败。

现在，我们修改 `Start` 类，在 `Start` 类中注入 `PhoneBookService`：

```

public class Start {
    @Property

```

```

private Entry entry = new Entry();
@Inject
private PhoneBookService phoneBookService;

void onSuccess() {
    // Class.forName("org.gjt.mm.mysql.Driver");
    // Connection conn = null;
    // PreparedStatement st = null;
    // try {
    // conn = DriverManager.getConnection(
    // "jdbc:mysql://localhost:3306/myexample", "root", "");
    // st = conn
    // .prepareStatement("insert into phonebook(name, number) value
s (?, ?)");
    // st.setString(1, entry.getName());
    // st.setString(2, entry.getNumber());
    // st.execute();
    // } finally {
    // st.close();
    // conn.close();
    // }
    phoneBookService.save(entry);
}
}

```

重启服务器，运行程序，一切都正常。

我们再来创建一个 **Service**，用于获取一个数据库连接。

新建 **ConnectionPool** 接口：

```

public interface ConnectionPool {
    Connection getConnection();
}

```

新建 **ConnectionPoolImpl** 类：

```

public class ConnectionPoolImpl implements ConnectionPool {
    public Connection getConnection() {
        try {
            Class.forName("org.gjt.mm.mysql.Driver");
        } catch (ClassNotFoundException e) {
            throw new RuntimeException(e);
        }
        Connection conn = null;
        try {
            conn = DriverManager.getConnection(
                "jdbc:mysql://localhost:3306/myexample", "root", "
");

```

```

        } catch (SQLException e) {
            throw new RuntimeException(e);
        }
        return conn;
    }
}

```

在 AppModule 类中注册 ConnectionPool:

```

public class AppModule {
    public static void bind(ServiceBinder binder) {
        binder.bind(PhoneBookService.class, PhoneBookServiceImpl.class);
        binder.bind(ConnectionPool.class);
    }
}

```

然后，我们就可以在 PhoneBookService 中注入 ConnectionPool 了。要在 Service 中注入另一个 Service，只需把它作为构造方法的参数传入即可。

修改 PhoneBookServiceImpl 类:

```

public class PhoneBookServiceImpl implements PhoneBookService {
    private final ConnectionPool connectionPool;

    public PhoneBookServiceImpl(ConnectionPool connectionPool) {
        this.connectionPool = connectionPool;
    }

    public void save(Entry entry) {
        // try {
        //     Class.forName("org.gjt.mm.mysql.Driver");
        // } catch (ClassNotFoundException e) {
        //     throw new RuntimeException(e);
        // }

        Connection conn = null;
        PreparedStatement st = null;
        try {
            conn = connectionPool.getConnection();
            st = conn
                .prepareStatement("insert into phonebook(name, number) values (?, ?)");
            st.setString(1, entry.getName());
            st.setString(2, entry.getNumber());
            st.execute();
        } catch (SQLException e) {
            throw new RuntimeException(e);
        } finally {

```

```

        ...
    }
}
}

```

运行程序，一切正常。

在开发模式下，可以查看所有的 **Service** 及其状态。

修改 **AppModule** 类：

```

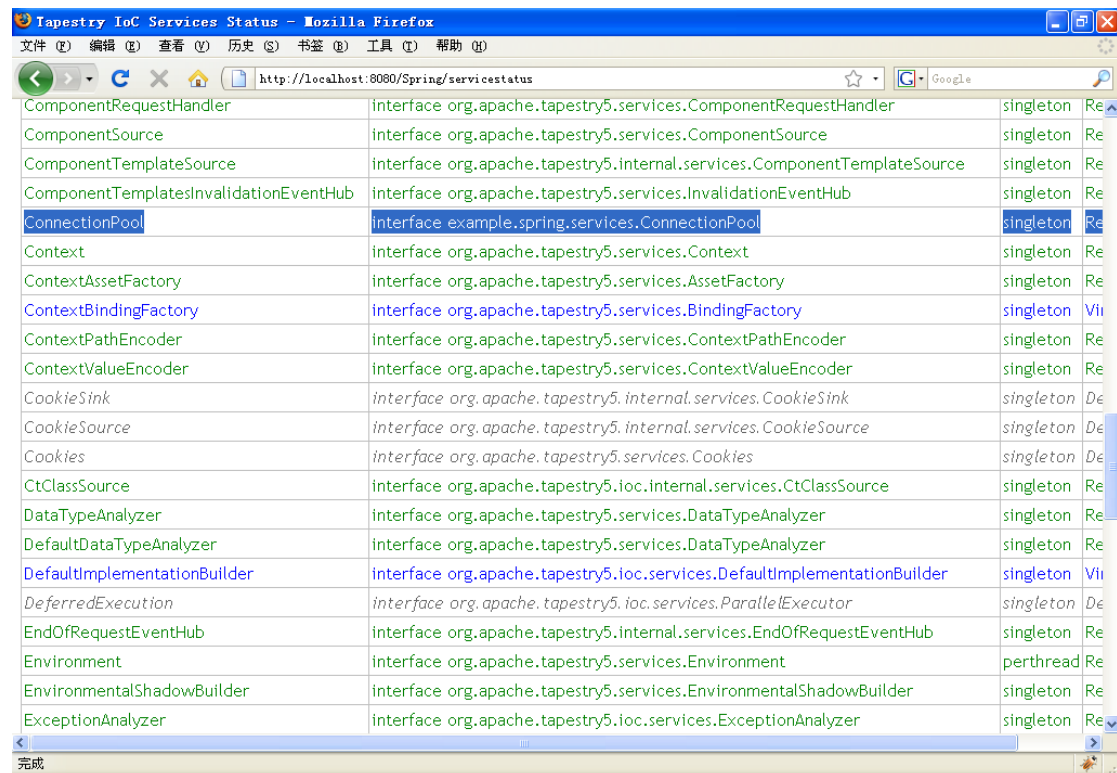
public class AppModule {
    public static void bind(ServiceBinder binder) {
        binder.bind(PhoneBookService.class, PhoneBookServiceImpl.class);

        binder.bind(ConnectionPool.class);
    }

    public static void contributeApplicationDefaults(
        MappedConfiguration<String, String> configuration) {
        configuration.add(SymbolConstants.PRODUCTION_MODE, "false");
    }
}

```

打开 <http://localhost:8080/Spring/servicestatus>，可以查看所有的 **Service**，包括系统内置的和我们自己创建的，如图 7-5。



(图 7-5)

通过前面的例子，我们对 **Tapestry IoC** 已经有了基本的了解。

Tapestry 同时也支持 Spring，我们完全可以用 Spring IoC 来代替 Tapestry IoC。

接下来，我们要做一个 ShowAll 页面，显示电话本中的所有记录，我们将在 ShowAll 页面中注入 Spring Bean。

依赖库

首先，Tapestry 5 提供一个专门用于集成 Spring 的 `tapestry-spring-5.1.0.5.jar` 库，把它加入 `WEB-INF/lib`。

其次，我们还需要 Spring 的二进制代码。Tapestry5.1 支持 Spring2.5.6。从 <http://www.springframework.org/download> 可以下载到 Spring，选择“带有依赖项”的发布版，文件名类似 `spring-framework-2.5.6.SEC01-with-dependencies.zip`，里面包含了很多第三方类库，可以省去再从不同站点额外下载第三方类库。解压下载的文件，把 `spring.jar` 和 `commons-logging.jar` 加入 `WEB-INF/lib`。

创建 Bean

新建 PhoneBookDao 接口：

```
public interface PhoneBookDao {  
    List<Entry> findAll();  
}
```

新建 PhoneBookDaoImpl 类：

```
public class PhoneBookDaoImpl extends JdbcDaoSupport implements PhoneBookDao {  
    public List<Entry> findAll() {  
        final List<Entry> entries = new ArrayList<Entry>();  
        getJdbcTemplate().query("select name, number from phonebook",  
            new RowCallbackHandler() {  
                public void processRow(ResultSet rs) throws SQLException {  
                    do {  
                        Entry entry = new Entry(rs.getString("name"),  
rs                            .getString("number"));  
                        entries.add(entry);  
                    } while (rs.next());  
                }  
            });  
        return entries;  
    }  
}
```

在这段访问数据库的代码中，我们仅提供了一条 SQL，而不需要打开/关闭 Connection、打开/关闭 Statement、异常处理等操作，是因为这些操作都被封装在 JdbcTemplate 中了。

装配 Bean

在 WEB-INF 目录下新建 spring-service.xml，代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
    <bean id="phoneBookDao" class="example.spring.PhoneBookDaoImpl">
        <property name="jdbcTemplate" ref="jdbcTemplate" />
    </bean>
    <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
        <property name="dataSource" ref="dataSource" />
    </bean>
    <bean id="dataSource"
          class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name="driverClassName" value="org.gjt.mm.mysql.Driver" />
        <property name="url" value="jdbc:mysql://localhost:3306/myexample" />
        <property name="username" value="root" />
        <property name="password" value="" />
    </bean>
</beans>
```

Spring 根据这个配置文件装配 Bean，建立 Bean 之间的关系。

配置 Tapestry

修改 web.xml：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
    <display-name>Spring</display-name>
    <context-param>
        <param-name>tapestry.app-package</param-name>
```

```

        <param-value>example.spring</param-value>
    </context-param>
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/spring-service.xml</param-value>
    </context-param>
    <filter>
        <filter-name>app</filter-name>
        <filter-class>org.apache.tapestry5.spring.TapestrySpringFilter
    </filter-class>
    </filter>
    <filter-mapping>
        <filter-name>app</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
</web-app>

```

为了集成 Spring, 必须用 `org.apache.tapestry5.spring.TapestrySpringFilter` 代替 `org.apache.tapestry5.TapestryFilter`。

在页面中注入 Bean

在 “example.spring.pages” 包中新建 ShowAll.html:

```

<html xmlns:t="http://tapestry.apache.org/schema/tapestry_5_1_0.xsd">
    <t:grid source="entries"></t:grid>
</html>

```

在 “example.spring.pages” 包中新建 ShowAll 类:

```

public class ShowAll {
    @Inject
    private PhoneBookDao phoneBookDao;

    public List<Entry> getEntries() {
        return phoneBookDao.findAll();
    }
}

```

从上面的代码可以看出, 在页面中注入 Spring Bean 和注入 Tapestry Service 的方法是一样的。类似的, 我们也可以在 Tapestry Service 中注入 Spring Bean。

重启服务器。打开 <http://localhost:8080/Spring/ShowAll>, 页面效果如图 7-6。



(图 7-6)