Глава 3

НАВИГАЦИЯ

В ЭТОЙ ГЛАВЕ...

- Статическая навигация
- Динамическая навигация
- Дополнительные вопросы, связанные с навигацией

В этой главе речь пойдет о настройке механизма навигации в Web-приложении. В частности, вы узнаете, как приложение может переходить с одной страницы на другую в зависимости от действий пользователя и выходных данных, получаемых в результате решений, принимаемых на уровне бизнес-логики.

Статическая навигация

Давайте сначала посмотрим, что происходит, когда пользователь приложения заполняет Web-страницу. Это может быть ввод данных в текстовых полях, выбор переключателей или выделение записей в списках.

Все эти операции редактирования осуществляются внутри браузера пользователя. Когда пользователь щелкает на кнопке, отвечающей за отправку данных формы, все внесенные им изменения передаются на сервер.

В это время Web-приложение анализирует входные данные пользователя и принимает решение о том, какую из JSF-страниц ему следует использовать для визуализации ответа. За выбор этой следующей JSF-страницы отвечает *обработчик навигации* (navigation handler).

В простом Web-приложении навигация является статической. То есть щелчок на определенной кнопке всегда приводит к выбору для визуализации ответа конкретной (фиксированной) JSF-страницы. В разделе "Простой пример" главы 1 уже показывалось обеспечение статической навигации между JSF-страницами в файле `faces-config.xml`.

К каждой кнопке добавляется атрибут `action`, например:

```
<h:commandButton label="Login" action="login"/>
```

На заметку! Как будет показано в главе 4, навигационные действия могут также присоединяться и к гиперссылкам.

Указываемое в этом атрибуте действие должно совпадать с *исходом* (outcome) в правиле навигации:

```
<navigation-rule>
 <from-view-id>/index.jsp</from-view-id>
 <navigation-case>
  <from-outcome>login</from-outcome>
  <to-view-id>/welcome.jsp</to-view-id>
 </navigation-case>
</navigation-rule>
```

Данное правило гласит, что действие `login` должно приводить к переходу на страницу `/welcome.jsp` только в том случае, если оно произошло внутри страницы `/index.jsp`.

Обратите внимание, что строки идентификаторов представления должны начинаться с символа `/`, а расширение — совпадать не с расширением URL-адреса, а с расширением файла (`.jsp`). Например, в приложении по всем страницам могут быть разбросаны кнопки с действием `logout`. С помощью одного единственного правила можно сделать так, чтобы щелчок на любой из них приводил к визуализации страницы `logout.jsp`:

```
<navigation-rule>
 <navigation-case>
```

```
    <from-outcome>logout</from-outcome>
    <to-view-id>/logout.jsp</to-view-id>
   </navigation-case>
  </navigation-rule>
```

Это правило будет действовать для всех страниц, потому что никакой элемент `from-view-id` не был указан.

Правила навигации с одинаковым `from-view-id` можно объединять, например:

```
<navigation-rule>
 <from-view-id>/index.jsp</from-view-id>
 <navigation-case>
  <from-outcome>login</from-outcome>
  <to-view-id>/welcome.jsp</to-view-id>
 </navigation-case>
 <navigation-case>
  <from-outcome>signup</from-outcome>
  <to-view-id>/newuser.jsp</to-view-id>
 </navigation-case>
</navigation-rule>
```

Такое объединение обязательным не является, но может оказаться полезным.

Внимание! Если данному действию не соответствует ни одно правило навигации, тогда просто снова отображается текущая страница.

Динамическая навигация

В большинстве Web-приложений навигация не является статической. Поток страниц зависит не только от того, на какой кнопке выполняется щелчок, но и от того, какие входные данные предоставляются. Например, отправка страницы регистрации может иметь два исхода: удачный и неудачный. Каким будет это исход — зависит от результата вычислений, а именно от того, действительными или недействительными окажутся предоставленные имя пользователя и пароль.

Для обеспечения динамической навигации у кнопки отправки должно быть *выражение метода* (method expression), например:

```
<h:commandButton label="Login" action="#{loginController.verifyUser}"/>
```

В этом примере `loginController` ссылается на бин некоего класса, у которого есть метод с именем `verifyUser`.

Выражение метода в атрибуте `action` не имеет никаких параметров. Возвращаемый им тип может быть любым. Возвращаемое значение преобразуется в строку путем вызова метода `toString`.

На заметку! В версии JSF 1.1 в качестве возвращаемого типа для метода действия можно было указывать только тип `String`. В версии 1.2 разрешается использовать любой возвращаемый тип. В частности, перечисления являются довольно удобной альтернативой, поскольку компилятор уже умеет перехватывать опечатки в именах действий.

Ниже показан пример метода действия:

```
String verifyUser() {
  if (...)
   return "success";
```

```
    else
    return "failure";
  }
```

Этот метод возвращает строку исхода "success" или "failure". Обработчик навигации использует эту возвращенную строку для поиска подходящего правила навигации.

На заметку! Метод действия может также возвращать и значение null, указывающее, что снова должна быть отображена та же самая страница.

Ниже вкратце перечислены шаги, которые выполняются всякий раз, когда пользователь щелкает на командной кнопке, в атрибуте action которой указано выражение метода.

- Извлечение указанного бина.
- Вызов указанного метода.
- Передача результирующей строки обработчику навигации. (Как будет объясняться в разделе "Алгоритм навигации" далее в этой главе, помимо результирующей строки обработчик навигации также еще получает и строку выражения метода.)
- Поиск обработчиком навигации следующей страницы.

Таким образом, получается, что для реализации поведения перехода требуется всего лишь предоставить ссылку на метод в подходящем классе бина. Вариантов для размещения этого метода много. Но наилучший подход — отыскать класс, имеющий все данные, которые необходимы для принятия решений.

Далее давайте посмотрим, как этот процесс выглядит в реальном приложении. На рис. 3.1 показана программа, которая отображает пользователю ряд вопросов викторины.

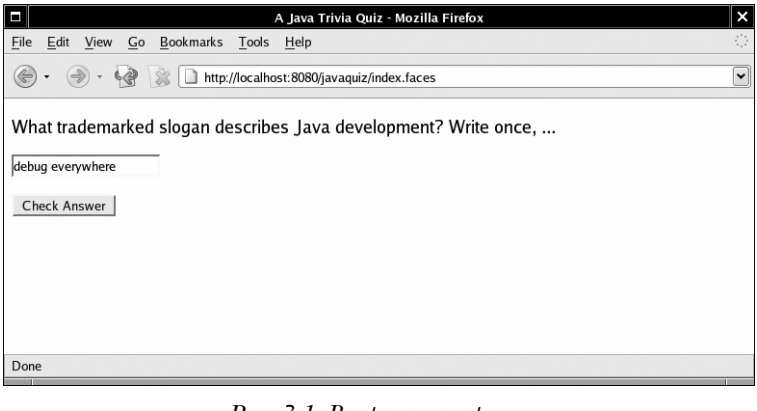

Рис. 3.1. Вопрос викторины

Когда пользователь щелкает на кнопке Check Answer (Проверить ответ), приложение проверяет, является ли предоставленный им ответ правильным. Если нет, ему дается еще один дополнительный шанс ответить на поставленный вопрос (рис. 3.2).

После двух неправильных ответов отображается следующий вопрос (рис. 3.3).

Рис. 3.2. Один неправильный ответ: предоставление второй попытки

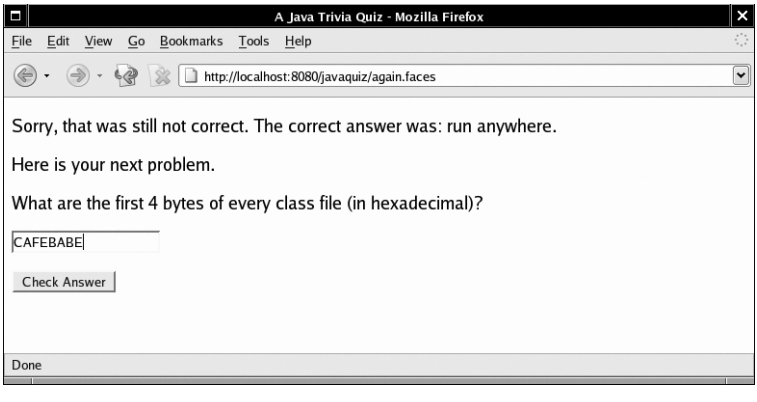

Рис. 3.3. Два неправильных ответа: переход к следующему вопросу

После правильного ответа, конечно, тоже отображается следующий вопрос. И, наконец, после последнего вопроса отображается страница с результатами и приглашением попробовать пройти викторину снова (рис. 3.4).

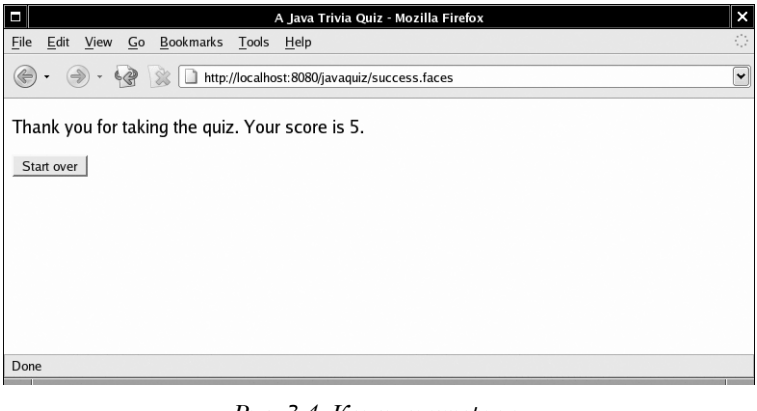

Рис. 3.4. Конец викторины

Наше приложение имеет два класса. Класс `Problem`, показанный в листинге 3.1, содержит описание одной задачи с вопросом, ответом и методом для проверки того, является ли предоставленный ответ правильным.

Класс `QuizBean` содержит описание викторины, которая состоит из ряда вопросов. Кроме того, экземпляр `QuizBean` еще также отслеживает текущую задачу и общий счет пользователя. Весь код полностью читатель сможет найти в листинге 3.2.

Листинг 3.1. javaquiz/src/java/com/corejsf/Problem.java

```
1.  package com.corejsf;
2.
3.  public class Problem {
4.     private String question;
5.     private String answer;
6.
7.     public Problem(String question, String answer) {
8.        this.question = question;
9.        this.answer = answer;
10.    }
11.
12.    public String getQuestion() { return question; }
13.
14.    public String getAnswer() { return answer; }
15.
16.    // Переопределить для выполнения более сложной проверки.
17.    public boolean isCorrect(String response) {
18.       return response.trim().equalsIgnoreCase(answer);
19.    }
20. }
```

В этом примере для размещения методов, связанных с навигацией, наиболее подходит класс `QuizBean`. Этому бину известно все о действиях пользователя и он может определять, какая страница должна отображаться следующей.

Листинг 3.2. javaquiz/src/java/com/corejsf/QuizBean.java

```
1.  package com.corejsf;
2.
3.  public class QuizBean {
4.     private int currentProblem;
5.     private int tries;
6.     private int score;
7.     private String response;
8.     private String correctAnswer;
9.
10.    // Здесь мы жестко кодируем задания. В реальном приложении
11.    // они бы извлекались из базы данных.
12.    private Problem[] problems = {
13.       new Problem(
14.          "What trademarked slogan describes Java development? Write once, ...",
15.          "run anywhere"),
16.       new Problem(
17.          "What are the first 4 bytes of every class file (in hexadecimal)?",
18.          "CAFEBABE"),
19.       new Problem(
20.          "What does this statement print? System.out.println(1+\"2\");",
```

```
21.         "12"),
22.       new Problem(
23.         "Which Java keyword is used to define a subclass?",
24.         "extends"),
25.       new Problem(
26.         "What was the original name of the Java programming language?",
27.         "Oak"),
28.       new Problem(
29.         "Which java.util class describes a point in time?",
30.         "Date")
31.     };
32.
33.     public QuizBean() { startOver(); }
34.
35.     // СВОЙСТВО: question
36.     public String getQuestion() {
37.       return problems[currentProblem].getQuestion();
38.     }
39.
40.     // СВОЙСТВО: answer
41.     public String getAnswer() { return correctAnswer; }
42.
43.     // СВОЙСТВО: score
44.     public int getScore() { return score; }
45.
46.     // СВОЙСТВО: response
47.     public String getResponse() { return response; }
48.     public void setResponse(String newValue) { response = newValue; }
49.
50.     public String answerAction() {
51.       tries++;
52.       if (problems[currentProblem].isCorrect(response)) {
53.         score++;
54.         nextProblem();
55.         if (currentProblem == problems.length) return "done";
56.         else return "success";
57.       }
58.       else if (tries == 1) {
59.         return "again";
60.       }
61.       else {
62.         nextProblem();
63.         if (currentProblem == problems.length) return "done";
64.         else return "failure";
65.       }
66.     }
67.
68.     public String startOverAction() {
69.       startOver();
70.       return "startOver";
71.     }
72.
73.     private void startOver() {
74.       currentProblem = 0;
75.       score = 0;
76.       tries = 0;
```

```
77.       response = "";
78.    }
79.
80.    private void nextProblem() {
81.       correctAnswer = problems[currentProblem].getAnswer();
82.       currentProblem++;
83.       tries = 0;
84.       response = "";
85.    }
86. }
```

Взгляните на код внутри метода `answerAction` в классе `QuizBean`. Этот метод возвращает строку `"success"` или `"done"`, если пользователь ответил на вопрос правильно, строку `"again"`, если пользователь ответил на вопрос неправильно первый раз, и строку `"failure"` или `"done"`, если во второй раз пользователь тоже ответил неправильно.

```
public String answerAction() {
 tries++;
  if (problems[currentProblem].isCorrect(response)) {
   score++;
   if (currentProblem == problems.length - 1) {
    return "done";
   }
   else {
    nextProblem();
    return "success";
   }
  }
  else if (tries == 1) {
   return "again";
  }
  else {
   if (currentProblem == problems.length - 1) {
     return "done";
   }
  else {
   nextProblem();
   return "failure";
  }
 }
}
```

Выражение метода `answerAction` присоединяется к кнопкам на каждой странице. Например, страница `index.jsp` содержит следующий элемент:

```
<h:commandButton value="Check answer" action="#{quiz.answerAction}"/>
```

Здесь `quiz` — экземпляр класса `QuizBean`, определение которого находится в файле `faces-config.xml`.

На рис. 3.5 показана структура каталогов данного приложения, а в листинге 3.3 — главная страница викторины `index.jsp`. Код страниц `success.jsp` и `failure.jsp` мы решили не приводить, поскольку они отличаются от `index.jsp` только сообщением, отображаемым в верхней части.

В листинге 3.4 показан код страницы `done.jsp`, которая отображает конечный счет (т.е. общее количество очков) и приглашает пользователя попробовать пройти викторину снова. На этой странице стоит обратить внимание на командную кнопку. Выглядит так, будто бы можно было применить статическую навигацию, поскольку щелчок на кнопке Start Over (Начать сначала) всегда возвращает пользователя к странице `index.jsp`. Однако мы использовали выражение метода:

```
<h:commandButton value="Start over" action="#{quiz.startOverAction}"/>
```

Метод `startOverAction` выполняет полезную работу, необходимую для того, чтобы игру можно было начать сначала, а именно — сбрасывает значение количества очков и перемешивает элементы ответов:

```
public String startOverAction() {
  startOver();
  return "startOver";
}
```

В целом методы действия применяются для выполнения двух следующих вещей.

- Для внесения в модель обновлений, появляющихся в результате действий пользователя;
- Для указания обработчику навигации, какая страница должна отображаться далее.

На заметку! Как будет рассказываться в главе 7, к кнопкам также могут присоединяться и слушатели действий. Тогда при выполнении пользователем щелчка на кнопке выполняется код, находящийся в методе `processAction` присоединенного слушателя. Однако слушатели действий не могут взаимодействовать с обработчиком навигации.

На рис. 3.5 показана структура каталогов данного приложения.

Поскольку мы выделили строки исхода так, чтобы они уникальным образом указывали, какой должна быть следующая Web-страница, мы можем использовать одно единственное правило навигации:

```
<navigation-rule>
 <navigation-case>
  <from-outcome>success</from-outcome>
  <to-view-id>/success.jsp</to-view-id>
 </navigation-case>
 <navigation-case>
  <from-outcome>again</from-outcome>
  <to-view-id>/again.jsp</to-view-id>
 </navigation-case>
 ...
</navigation-rule>
```

На рис. 3.6 показана диаграмма переходов, а в листинге 3.6 — строки сообщений.

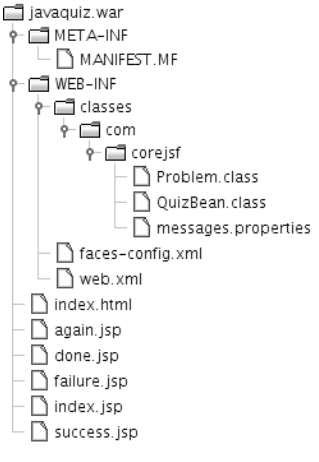

Рис. 3.5. Структура каталогов Java-приложения викторины

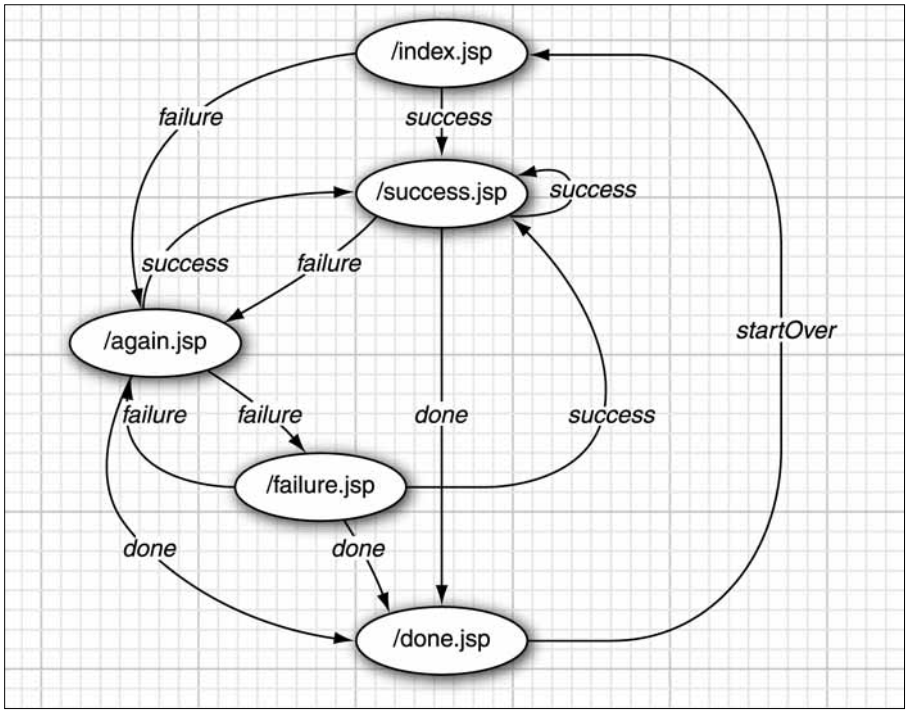

Рис. 3.6. Диаграмма переходов для приложения с викториной на Java

Листинг 3.3. `javaquiz/web/index.jsp`

```
<html>
   <%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
   <%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>

   <f:view>
      <head>
         <title><h:outputText value="#{msgs.title}"/></title>
      </head>
      <body>
         <h:form>
            <p>
               <h:outputText value="#{quiz.question}"/>
            </p>
            <p>
               <h:inputText value="#{quiz.response}"/>
            </p>
            <p>
               <h:commandButton value="#{msgs.answerButton}"
                  action="#{quiz.answerAction}"/>
            </p>
         </h:form>
      </body>
   </f:view>
</html>
```

Листинг 3.4. javaquiz/web/done.jsp

```
<html>
   <%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
   <%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
   <f:view>
      <head>
         <title><h:outputText value="#{msgs.title}"/></title>
      </head>
      <body>
         <h:form>
            <p>
               <h:outputText value="#{msgs.thankYou}"/>
               <h:outputFormat value="#{msgs.score}">
                  <f:param value="#{quiz.score}"/>
               </h:outputFormat>
            </p>
            <p>
               <h:commandButton value="#{msgs.startOverButton}"
                  action="#{quiz.startOverAction}"/>
            </p>
         </h:form>
      </body>
   </f:view>
</html>
```

Листинг 3.5. javaquiz/web/WEB-INF/faces-config.xml

```
<?xml version="1.0"?>
<faces-config xmlns="http://java.sun.com/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
      http://java.sun.com/xml/ns/javaee/web-facesconfig_1_2.xsd"
   version="1.2">
   <navigation-rule>
      <navigation-case>
         <from-outcome>success</from-outcome>
         <to-view-id>/success.jsp</to-view-id>
         <redirect/>
      </navigation-case>
      <navigation-case>
         <from-outcome>again</from-outcome>
         <to-view-id>/again.jsp</to-view-id>
      </navigation-case>
      <navigation-case>
         <from-outcome>failure</from-outcome>
         <to-view-id>/failure.jsp</to-view-id>
      </navigation-case>
      <navigation-case>
         <from-outcome>done</from-outcome>
         <to-view-id>/done.jsp</to-view-id>
      </navigation-case>
      <navigation-case>
         <from-outcome>startOver</from-outcome>
         <to-view-id>/index.jsp</to-view-id>
```

```
28.       </navigation-case>
29.    </navigation-rule>
30.    <managed-bean>
31.       <managed-bean-name>quiz</managed-bean-name>
32.       <managed-bean-class>com.corejsf.QuizBean</managed-bean-class>
33.       <managed-bean-scope>session</managed-bean-scope>
34.    </managed-bean>
35.
36.    <application>
37.       <resource-bundle>
38.          <base-name>com.corejsf.messages</base-name>
39.          <var>msgs</var>
40.       </resource-bundle>
41.    </application>
42. </faces-config>
```

Листинг 3.6. javaquiz/src/java/com/corejsf/messages.properties

```
1.  title=A Java Trivia Quiz
2.  answerButton=Check Answer
3.  startOverButton=Start over
4.  correct=Congratulations, that is correct.
5.  notCorrect=Sorry, that was not correct. Please try again!
6.  stillNotCorrect=Sorry, that was still not correct.
7.  correctAnswer=The correct answer was: {0}.
8.  score=Your score is {0}.
9.  thankYou=Thank you for taking the quiz.
```

Дополнительные вопросы, связанные с навигацией

Приемов, описанных в предыдущих разделах, должно быть достаточно для удовлетворения большинства практических потребностей, связанных с навигацией. В этом разделе мы опишем оставшиеся правила для элементов навигации, которые могут присутствовать в файле `faces-config.xml`. На рис. 3.7 показана синтаксическая диаграмма для допустимых элементов.

На заметку! Как уже показывалось в разделе "Конфигурирование бинов" главы 2, информация о навигации также может размещаться и в других конфигурационных файлах, а не только в стандартном файле `faces-config.xml`.

В этой диаграмме видно, что каждый элемент `navigation-rule` и `navigationcase` может иметь назначаемые произвольно элементы `description`, `display-name` и `icon`. Эти элементы предназначены для использования в средствах-конструкторах и потому более подробно здесь рассматриваться не будут.

Переадресация

Если после `to-view-id` добавляется элемент `redirect`, тогда контейнер JSP завершает текущий запрос и отправляет клиенту HTTP-ответ, переадресовывающий его на другую страницу.

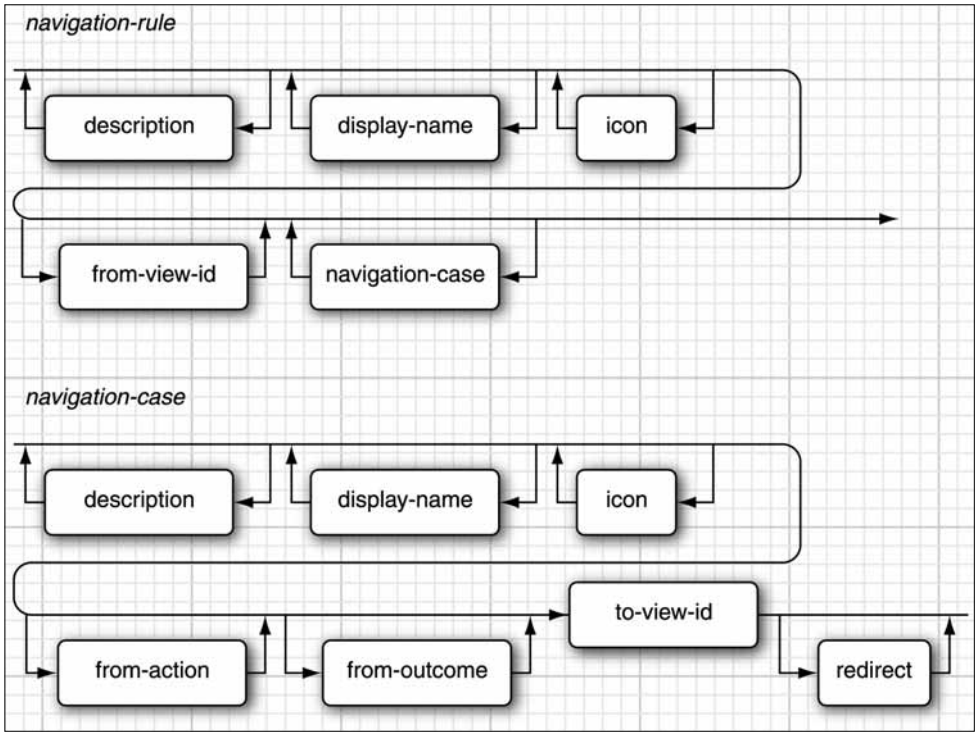

Рис. 3.7. Синтаксическая диаграмма для элементов, связанных с навигацией

Переадресация выполняется медленнее пересылки, поскольку подразумевает прохождение браузером полностью еще одного цикла. Однако переадресация дает браузеру шанс обновить поле адреса.

На рис. 3.8 показано, как изменяется поле адреса, когда добавляется такой элемент переадресации:

```
<navigation-case>
 <from-outcome>success</from-outcome>
 <to-view-id>/success.jsp</to-view-id>
 <redirect/>
</navigation-case>
```

Без переадресации, исходный URL-адрес (`localhost:8080/javaquiz/index.faces`) остается неизменным при переходе пользователя со страницы `/index.jsp` на страницу `/success.jsp`. С переадресацией браузер отображает новый URL-адрес (`localhost:8080/javaquiz/success.faces`).

Совет. Элемент `redirect` лучше всего применять для тех страниц, на которых пользователь может помещать закладку.

Внимание! Без элемента `redirect` обработчик навигации пересылает текущий запрос следующей странице, и все пары “имя-значение”, хранящиеся в области действия запроса, переносятся и на следующую страницу. Без элемента `redirect`, однако, данные области запроса утрачиваются.

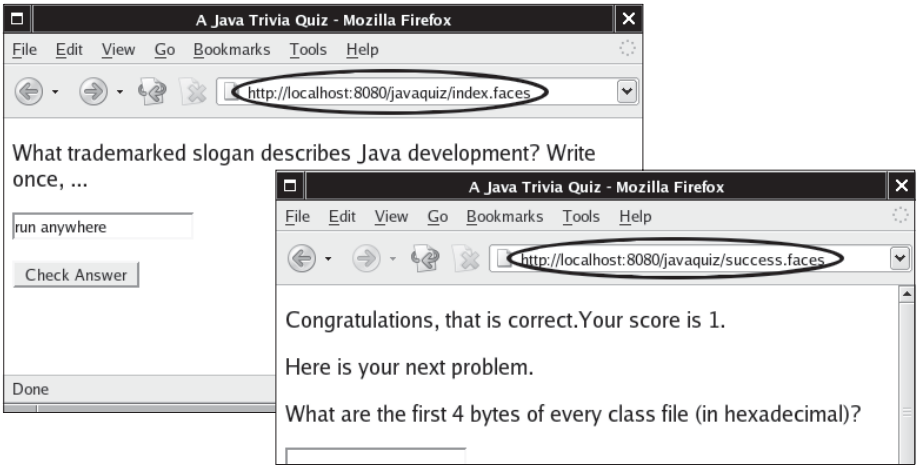

Рис. 3.8. Переадресация, приводящая к обновлению URL-адреса в окне браузера

Групповые символы

Внутри элемента `from-view-id` в правиле навигации также разрешается использовать и групповые символы, например:

```
<navigation-rule>
 <from-view-id>/secure/*</from-view-id>
 <navigation-case>
  . . .
 </navigation-case>
</navigation-rule>
```

Это правило будет действовать для всех страниц, начинающихся с префикса `/secure/`. Символ * может быть только один, и находиться он должен в конце строки идентификатора.

Если подходящими оказывается сразу несколько правил с групповыми символами, выбирается наиболее длинный вариант.

На заметку! Вместо того чтобы пропускать элемент `from-view-id`, можно еще также использовать один из его следующих вариантов для указания правила, распространяющегося на все страницы:

```
<from-view-id>/*</from-view-id>
```

или

```
<from-view-id>*</from-view-id>
```

Использование `from-action`

Структура элемента `navigation-case` является более сложной. Помимо элемента `from-outcome` здесь еще также есть элемент `from-action`. Такая гибкость может быть полезна при наличии либо двух отдельных действий с одинаковой строкой действия, либо двух выражений с методами действия, тоже возвращающими одинаковую строку действия.

Например, предположим, что в нашем приложении с викториной `startOverAction` возвращает не строку `"startOver"`, а строку `"again"`. Такую строку может возвращать и `answerAction`. Для различения этих двух случаев навигации как раз и можно использовать элемент `from-action`. Содержимое этого элемента обязательно должно полностью соответствовать строке выражения метода в атрибуте `action`.

```
<navigation-case>
 <from-action>#{quiz.answerAction}</from-action>
 <from-outcome>again</from-outcome>
 <to-view-id>/again.jsp</to-view-id>
</navigation-case>
<navigation-case>
 <from-action>#{quiz.startOverAction}</from-action>
 <from-outcome>again</from-outcome>
 <to-view-id>/index.jsp</to-view-id>
</navigation-case>
```

На заметку! Обработчик навигации *не* вызывает метод, указанный внутри разделителей `#{...}`. Этот метод вызывается до того, как обработчик навигации вступает в игру. Обработчик навигации просто использует строку `from-action` в качестве ключа для поиска подходящего случая навигации.

Алгоритм навигации

В завершении этой главы мы приведем описание конкретного алгоритма, которым обработчик навигации пользуется для разрешения неоднозначных ситуаций. Если хотите, может пока пропустить этот раздел и вернуться к нему тогда, когда возникнет необходимость разобраться в правилах, созданных другими программистами или автоматизированными инструментальными средствами.

Итак, данный алгоритм имеет три входных элемента.

1. *Исход* (outcome), то есть значение атрибута `action` или строки, получаемой в результате вычисления выражения метода.
2. *Идентификатор текущего представления* (view ID).
3. *Действие* (action), то есть буквенное значение атрибута `action` в компоненте, инициировавшем навигацию.

Первый этап — отыскать подходящее правило навигации (`navigation-rule`), выполнив следующие шаги.

1. Если исходом является `null`, немедленно вернуться и снова отобразить текущую страницу.
2. Объединить все правила навигации с одинаковым значением `from-view-id`.
3. Попытаться отыскать правило навигации со значением `from-view-id`, в точности соответствующим идентификатору представления. Если такое правило существует, использовать его.
4. Проанализировать все правила навигации, значения `from-view-id` в которых заканчиваются специальным суффиксом наподобие `secure`. Для каждого такого правила проверить, совпадает ли префикс (без символа `*`) с соответствующим префиксом идентификатора представления. В случае обнаружения нескольких

подходящих правил использовать то, которое имеет наиболее длинный соответствующий префикс.

5. Если есть правило без элемента `from-view-id`, использовать его.
6. Если не удается обнаружить вообще никаких совпадений, отобразить снова текущую страницу.

Второй этап — проанализировать все элементы `navigation-case` в подошедшем правиле навигации (которое может состоять из нескольких объединенных элементов `navigation-rule` с подходящими значениями `from-view-id`).

Для того чтобы отыскать подходящий случай (`navigation-case`), необходимо выполнить перечисленные ниже шаги.

1. Если в элементе `navigation-case` присутствует и подходящий элемент `from-outcome`, и подходящий элемент `from-action`, использовать его.
2. В противном случае, т.е. если в элементе `navigation-case` присутствует подходящий элемент `from-outcome`, но нет подходящего элемента `from-action`, использовать его.
3. Или, если в элементе `navigation-case` присутствует подходящий элемент `from-action`, но нет подходящего элемента `from-outcome`, тогда использовать его.
4. Или же, если в элементе `navigation-case` нет ни подходящего элемента `from-outcome`, ни подходящего элемента `from-action`, использовать его.
5. Если не удается найти вообще никаких совпадений, снова отобразить текущую страницу.

Конечно, мы рекомендуем не создавать в своих программах сложные правила навигации. Если не использовать специальные символы и элементы `from-action`, тогда можно и не изучать никакие мудреные детали алгоритма навигации.