Действия, определяемые программистом (JSP Custom Actions).

Этот механизм позволяет расширить язык JSP новыми действиями, дополняющими стандартные действия JSP. Наиболее ярким примером набора таких действий являются действия, входящие в библиотеку действий JSTL (Java Standard Tag Library) – это набор custom actions, который поддерживается любым современным сервером приложений.

Custom actions – это элементы в составе JSP-страницы, которые обрабатываются на сервере в момент поступления запроса. По усмотрению разработчика они могут выполнять генерацию фрагмента ответа либо выполнять любое действие, не связанное с визуализацией интерфейса пользователя (например, обращаться к EJB-компонентам или БД для получения информации, которая будет отображена с помощью других действий).

Custom actions – это основной механизм, позволяющий исключить из JSP-страниц код на языке Java (скриптовые элементы).

Функциональность действия, определяемого программистом, реализуется специальным классом, который называется обработчиком действия (tag handler). Каждому действию соответствует один класс обработчика. Эти классы должны реализовывать определенные интерфейсы, которые предусмотрены в Tag Extension API.

Дескриптор библиотеки действий (Tag Library Descriptor)

Так как обращение к функциональности, реализованной обработчиком действия, происходит через элемент JSP-страницы, то JSP-транслятору необходимо знать соответствие между элементами и обработчиками. Кроме того, необходимо предоставить транслятору описание атрибутов, которые могут быть у элемента действия, и переменных, порождаемых действием. Также JSP-транслятор может на этапе трансляции контролировать правильность использования действий в JSP-страницах, для чего требуется дополнительная информация о действии.

Для формального описания элементов действий и связанной с ними дополнительной информации используются особые XML-дескрипторы. Один дескриптор может описывать несколько действий (библиотеку действий), поэтому он называется Tag Library Descriptor (TLD) и оформляется файлом с расширением `tld`.

Для правильной трансляции JSP-страниц, использующих custom actions, TLD необходим так же, как и обработчики действий. Фактически TLD во многом определяет то, какой код генерируется в PIC для взаимодействия с обработчиком действия.

TLD представляет собой XML-документ, корневым элементом которого является элемент `taglib`. Пространство имен XML-языка, используемого для написания TLD, имеет идентификатор `http://java.sun.com/JSP/TagLibraryDescriptor`.

При описании формата TLD будем использовать следующие обозначения после имени элемента: ? – элемент необязателен, если присутствует, то не может повторяться; + – элемент обязателен, может повторяться; * – элемент необязателен, если присутствует, то может повторяться. Если обозначение отсутствует, то элемент обязателен и не может повторяться. Порядок следования элементов в XML важен.

Библиотека действий

Обобщенная структура TLD выглядит так:

```
taglib
      tlib-version
      short-name
      uri?
```

```
validator?
      validator-class
      init-param*
            param-name
            param-value
tag* (см. отдельно)
tag-file* (см. отдельно)
function* (см. отдельно)
taglib-extension*
      extension-element+
```

Элемент `tlib-version` указывает версию данной библиотеки действий, например, `<tlib-version>1.0</tlib-version>`.

Элемент `short-name` – это краткое название библиотеки действие, которое может использоваться инструментарием разработчика JSP, например, в качестве префикса для действий из этой библиотеки.

Элемент `uri` – глобально-уникальный идентификатор (обычно в формате URL), назначенный данной библиотеке действий. Этот идентификатор следует использовать как значение атрибута `uri` директивы `taglib` при подключении данной библиотеки к JSP. Пример: `<uri>http://johnheadlong.nightmail.ru/tags/common</uri>`.

Элемент `validator` используется для описания класса, который будет задействован на этапе трансляции JSP-страниц, использующих действия из этой библиотеки, для проверки правильности этих действий с учетом правил, зафиксировать которые в TLD нет возможности. Полностью квалифицированное имя данного класса описывается с помощью вложенного элемента `validator-class`. Элементы `init-param` описывают параметры инициализации класса `validator-class`.

Действие

Для определения действия в TLD служит элемент `tag`:

```
tag*
      name
      tag-class
      tei-class?
      body-content
      variable*
            name-given | name-from-attribute
            variable-class?
            declare?
            scope?
      attribute*
            name
            required?
            rtexprvalue? type? | fragment?
      dynamic-attributes?
      tag-extension*
            extension-element+
```

Элемент `tag` описывает действие, входящее в состав библиотеки с помощью вложенных элементов:

- `name` – имя элемента действия в том виде, в каком оно будет использовано в JSP. Имя действия в пределах библиотеки должно быть уникальным;
- `tag-class` – полностью квалифицированное имя класса обработчика действия (tag handler);

- `tei-class` – полностью квалифицированное имя класса, который будет задействован на этапе трансляции использующих данное действие JSP-страниц для проверки правильности его использования с учетом правил, зафиксировать которые в TLD нет возможности;
- `body-content` – тип тела действия, предусмотрено четыре значения:
  - `JSP` – действие может иметь тело (возможно пустое), которое будет интерпретироваться транслятором как обычный JSP-фрагмент. Например, если тело содержит выражения, то они будут вычисляться, вложенные действия также будут выполняться;
  - `empty` – тело действия должно быть пустым;
  - `scriptless` – в теле действия запрещено использовать скриптовые элементы, разрешены только шаблон, EL-элементы и вложенные действия;
  - `tagdependent` – действие может иметь тело, возможно пустое, которое будет интерпретироваться на этапе выполнения обработчиком действия. Последнему тело действия будет доступно точно в том виде, в каком оно было указано в исходном файле JSP. Таким образом, в теле действия можно внедрять фрагменты на других языках;
- `variable` – описание переменной, порождаемой действием.
- `attribute` – описание атрибута действия.
- `dynamic-attributes` – указывает, предусмотрена ли для данного действия возможность указывать дополнительные (динамические) атрибуты, не только значения, но и имена которых вычисляются на этапе обработки запроса. По умолчанию такая возможность отключена (`false`). Если она включена (`true`), то обработчик действия помимо прочих обязательных интерфейсов должен реализовать интерфейс `javax.servlet.jsp.tagext.DynamicAttributes`. Динамические атрибуты могут принимать значения любого объектного типа, вычисляемые на этапе обработки запроса. Динамические атрибуты в JSP можно формировать с помощью стандартного действия `<jsp:attribute>`.

Атрибут действия

Атрибуты действия – это основной механизм передачи информации от JSP-страницы, использующей действие, к обработчику действия. Атрибут описывается следующими вложенными элементами:

- `name` – имя атрибута;
- `required` – указывает, является ли атрибут обязательным (`true`) или нет (`false`, по умолчанию);
- `rtexprvalue` – указывает тип значения атрибута: динамическое (`true`) или статическое (`false`, по умолчанию). Динамическое значение атрибута может формироваться на этапе обработки запроса выражением или EL-элементом;
- `type` – полностью квалифицированное имя типа атрибута, по умолчанию – `java.lang.String`. Все атрибуты со статическими значениями должны иметь тип `java.lang.String`;
- `fragment` – значение `true` указывает, что данный атрибут имеет тип `javax.servlet.jsp.tagext.JspFragment`, то есть его значение представляет собой фрагмент JSP-кода, который передается обработчику действия и может вычисляться по его усмотрению столько раз, сколько необходимо. Если указан элемент `fragment`, то ни `rtexprvalue`, ни `type` указывать нельзя. Если указан элемент `fragment` со значением `false` (оно предполагается по умолчанию), то тип атрибута – `java.lang.String`.

Переменная действия

Переменные, порождаемые действием – это основной механизм передачи информации от обработчика действия в вызывающую его JSP-страницу. При этом порожденные действием переменные можно использовать как в скриптовых выражениях и скриптлетам, так и в EL-выражениях, потому что значение переменной сохраняется в атрибуте контекста страницы и в локальной переменной метода `_jspService()` класса PIC. Переменная описывается следующими вложенными элементами:

- `name-given` – имя порождаемой переменной;
- `name-from-attribute` – имя атрибута действия, значение которого будет указывать имя порождаемой переменной. Можно указывать только один из элементов `name-given` и `name-from-attribute`. Значение атрибута, указывающего имя порождаемой переменной должно быть известно на этапе трансляции, то есть данный атрибут не может принимать значение, формируемое на этапе обработки запроса;
- `variable-class` – полностью квалифицированное имя типа порождаемой переменной, по умолчанию `java.lang.String`;
- `declare` – определяет, генерировать ли в PIC объявление локальной переменной для переменной, порождаемой действием. Если да (`true`, по умолчанию), то переменная будет объявлена с типом, указанным в элементе `variable-class`, если нет, то предполагается, что локальная переменная была объявлена ранее;
- `scope` – область видимости порождаемой переменной, которая определяет момент, начиная с которого переменная будет доступна выражениям, скриптовым и EL-элементам. Возможные значения:
  - `NESTED` – переменная будет доступна только в рамках тела действия (используется по умолчанию). За пределами тела действия соответствующей локальной переменной не существует, но может существовать доступный для EL-элементов атрибут с таким же именем в одной из обычных областей видимости (`page`, `request`, `session` или `application`);
  - `AT_BEGIN` – переменная будет доступна сразу после открывающего тега элемента действия и до конца JSP-страницы;
  - `AT_END` – переменная будет доступна сразу после закрывающего тега элемента действия и до конца JSP-страницы;

Тэг-файл

Для определения тэг-файла в TLD служит элемент `tag-file`:

```
tag-file*
      name
      path
      tag-extension*
            extension-element+
```

Элемент `tag-file`, как и элемент `tag`, описывает действие, входящее в состав библиотеки. Этот элемент используется для описания действий, которые реализованы не Java-классом, а с помощью так называемого тэг-файла на языке JSP. Возможность реализовать custom action в виде тэг-файла появилась в JSP 2.0 для тех ситуаций, когда основной функцией действия является генерация фрагмента ответа. Информация об атрибутах действия, порождаемых переменных и прочее в данном случае описывается не в TLD, а внедряется в тэг-файл с помощью специальных директив, которые запрещено

использовать в JSP. Вложенный элемент `name` здесь указывает имя действия. Элемент `path` указывает путь к тэг-файлу относительно корня веб-модуля. Путь должен начинаться с `/WEB-INF/tags`, если библиотека действий не упакована в jar-архив или с `/META-INF/tags`, если библиотека действий упакована в jar-архив, который в веб-модуле располагается в каталоге `/WEB-INF/lib`.

Функция

Для описания функции, которую можно использовать при конструировании EL-выражений используется элемент `function`:

```
function*
      name
      function-class
      function-signature
      function-extension*
            extension-element+
```

Подробнее функции рассмотрены в лекции по языку EL.

Tag Extension API.

Это набор классов и интерфейсов, которые используются для разработки обработчиков действий – классов на языке Java, реализующих функциональность действий. Этот API расположен в пакете `javax.servlet.jsp.tagext`. В JSP 2.0 определено три вида обработчиков действий – классические (classic), простые (simple) и в виде тэг-файлов (tag file).

Классические обработчики действий (Classic Tag Handlers)

Различают три вида действий, определяемых программистом:

1) Simple Actions (простые действия) – это действия, которые вычисляются однократно и не обрабатывают свое тело;
2) Iterative Actions (циклические действия) – это действия, которые вычисляются циклически и не обрабатывают свое тело;
3) Body Actions (действия, обрабатывающие тело) – это действия, которые могут вычисляться циклически и обрабатывать свое тело до того, как оно будет включено в состав ответа.

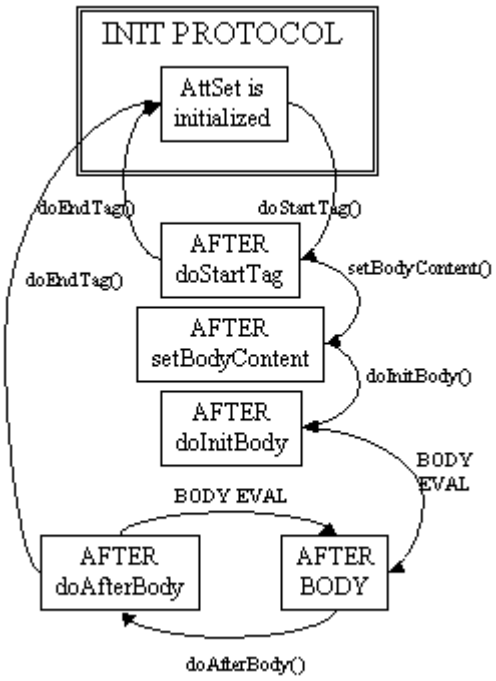
Данное разделение имеет наибольшее значение для классических обработчиков действий, так как каждому виду соответствует отдельный интерфейс. Простые обработчики действий и тэг-файлы также позволяют создавать действия любых из указанных видов, но не разделяют их с точки зрения программной реализации.

Любой классический обработчик действия должен реализовать интерфейс `Tag` и содержать публичный конструктор без параметров. Кроме того, для каждого атрибута действия (за исключением динамических) в классе обработчика должно быть определено закрытое поле и пара get- и set-методов. Эти методы используются на этапе обработки запроса для передачи значений атрибутов объекту обработчика.

Интерфейс `Tag` предусмотрен для реализации простых действий (simple actions). Для реализации действий других типов (циклических и обрабатывающих свое тело) используются интерфейсы `IterationTag` и `BodyTag`, соответственно, расширяющие интерфейс `Tag`.

Жизненный цикл экземпляра классического обработчика действия, обрабатывающего свое тело, может быть представлен так:

1) создание нового объекта обработчика действия (вызов конструктора без параметров) или получение ранее созданного объекта;
2) вызов метода `setPageContext()` для установки контекста JSP-страницы;
3) вызов метода `setParent()` для установки обработчика родительского действия, в теле которого используется данное действие;
4) вызов set-методов для установки атрибутов;
5) вызов метода `doStartTag()` для обработки открывающего тега элемента действия;
6) вызов методов `setBodyContent()` и `doInitBody()`, если тело действия непустое и `doStartTag()` вернул значение `EVAL_BODY_BUFFERED`;
7) вычисление тела, если `doStartTag()` вернул `EVAL_BODY_INCLUDE` или `EVAL_BODY_BUFFERED`;
8) вызов метода `doAfterBody()` и переход к п.7, если метод вернул `EVAL_BODY_AGAIN`;
9) вызов метода `doEndTag()` для обработки закрывающего тега элемента действия;
10) вызов метода `release()` перед уничтожением объекта обработчика действия.



Создание и уничтожение экземпляров обработчиков действий обычно оптимизируется, и экземпляр не уничтожается, как только закончилась обработка конкретного элемента действия, поскольку одно и то же действие используется многократно в пределах как одной, так и нескольких JSP-страниц. Таким образом, действия 2-9 могут многократно повторяться, поэтому при реализации методов `doEndTag()` и `doStartTag()` нужно выставлять начальные значения для полей класса обработчика и порождаемых действием переменных.

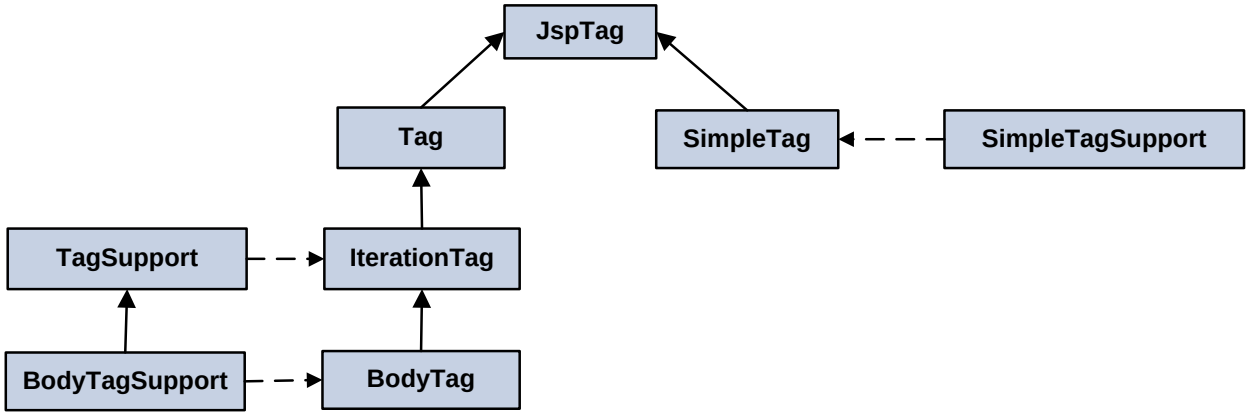
Жизненные циклы экземпляров классических обработчиков простых и циклических действий являются частными случаями жизненного цикла, приведенного выше.

В пакете `javax.servlet.jsp.tagext` определены классы `TagSupport` и `BodyTagSupport`, которые обеспечивают реализацию по умолчанию обязательных интерфейсов классических обработчиков действий. Эти классы используются как базовые, при их расширении достаточно переопределить нужные в конкретном обработчике методы. Класс `TagSupport` реализует интерфейс `IterationTag` и может использоваться как базовый для обработчиков простых и циклических действий. Класс `BodyTagSupport` расширяет класс `TagSupport` и дополнительно реализует интерфейс `BodyTag`, поэтому используется как базовый для обработчиков действий, обрабатывающих свое тело.

Простые обработчики действий (Simple Tag Handlers)

Данный способ реализации обработчиков действий предполагает наличие всего одного обязательного интерфейса `SimpleTag`. Этот интерфейс не расширяет интерфейс `Tag`, вследствие чего простые обработчики действий несовместимы по типу с классическими. Чтобы обеспечить возможность различным API работать как с простыми,

так и с классическими обработчиками, в качестве базового был введен маркерный интерфейс `JspTag`. В результате обобщенная схема классов и интерфейсов, составляющих Tag Extension API, выглядит следующим образом (рис. ?).



Простой обработчик действий обязан реализовать интерфейс `SimpleTag` и предоставить публичный конструктор без параметров. Также необходимы закрытые поля для атрибутов действия и get-/set-методы для работы с ними. Интерфейс `SimpleTag` позволяет реализовать обработчик действия любого вида: простого, циклического или обрабатывающего свое тело.

Рассмотрим методы интерфейса `SimpleTag`:

`setJspContext()` – метод вызывается на этапе инициализации обработчика действия, в качестве параметра передается контекст JSP-страницы (объект типа `JspContext`). Обработчик использует данный контекст для доступа к пользовательским объектам JSP с областью видимости `page` и как объект-контейнер для переменных, порождаемых действием. После приведения к типу `PageContext` из контекста страницы также можно получить доступ к инфраструктуре сервлетов (запрос, сессия, веб-приложение). При реализации метода `setJspContext()` переданный объект обычно сохраняется в закрытом поле обработчика действия и используется в других методах этого класса;

`setParent()` – метод вызывается на этапе инициализации обработчика действия, в качестве параметра передается объект типа `JspTag`, который соответствует объекту обработчика того действия, в теле которого используется данное действие. Например, если в JSP-странице действия вложены следующим образом:

```
<c:forEach ...>
    <c:out ... />
</c:forEach>
```

, то в качестве параметра метода `setParent()`, вызванного у обработчика действия `<c:out>` будет передан обработчик действия `<c:forEach>`.

При реализации метода `setParent()` переданный объект сохраняется в закрытом поле класса обработчика действия и позже может быть использован для реализации взаимодействующих друг с другом действий (cooperating actions).

`getParent()` – метод должен возвращать тот объект, который был ранее передан методу `setParent()` в качестве параметра. Метод может быть использован для навигации по дереву вложенных друг в друга действий.

Так как в методах `setParent()` и `getParent()` используется тип `JspTag`, то действие, реализованное простым обработчиком, может быть вложено в тело действия, реализованного как классическим, так и простым обработчиком;

`setJspBody()` – метод, устанавливающий тело действия в виде объекта типа `JspFragment` (то есть в виде фрагмента JSP-кода), вычислением которого может управлять сам обработчик;

`doTag()` – предназначен для реализации функциональности действия, то есть в этом методе пишется весь тот код, который в классическом обработчике распределен по методам `doStartTag()`, `doAfterBody()` и `doEndTag()`.

Жизненный цикл простого обработчика действия может быть описан так:

1) для каждого использования элемента действия на JSP-странице создается новый экземпляр простого обработчика действия (с помощью конструктора без параметров). В отличие от классических, простые обработчики действий не кэшируются;
2) вызываются методы `setJspContext()` и `setParent()`, причем последний вызывается только тогда, когда данное действие использовано внутри другого действия;
3) вызываются set-методы для атрибутов действия;
4) если у элемента действия в JSP-странице есть тело, то вызывается метод `setJspBody()`;
5) вызывается метод `doTag()`.

Тело действия, реализованного простым обработчиком, может быть описано в TLD как `empty`, `tagdependent` или `scriptless`, значение `JSP` в этом случае использовать нельзя, то есть тело такого действия не может содержать скриптовые элементы.

Если часть JSP-страницы, следующую после элемента действия, следует проигнорировать, то простой обработчик действия может выбросить из метода `doTag()` исключение класса `SkipPageException`.

Поскольку тело действия, реализованного простым обработчиком, не может содержать скриптовых элементов, то теряют смысл порожденные действием переменные с областью видимости `NESTED`. Синхронизация переменных с областями видимости `AT_BEGIN` и `AT_END` выполняется после вызова метода `doTag()`. Под синхронизацией понимается локальным переменным, доступным в выражениях и скриптлетах, значений атрибутов контекста страницы с соответствующими именами. Что касается классических обработчиков действий, то после вызова метода `doStartTag()` синхронизируются переменные с областями видимости `AT_BEGIN` или `NESTED`, кроме случая, когда этот метод возвращает `EVAL_BODY_BUFFERED`. После вызова метода `doAfterBody()` переменные с областями видимости `AT_BEGIN` и `NESTED` снова синхронизируются. После вызова метода `doEndTag()` синхронизируются переменные с областями видимости `AT_BEGIN` и `AT_END`.

Когда действие, реализованное классическим обработчиком, используется в теле действия, реализованного простым обработчиком, возникает проблема передачи последнему ссылки на обработчик родительского действия. В каждом из таких случаев для объекта типа `SimpleTag` создается объект-оболочка типа `TagAdapter`, который передается в качестве параметра методу `setParent()` классического обработчика. Класс `TagAdapter` реализует интерфейс `Tag`, а его методы `setParent()`, `doStartTag()` и `doEndTag()` выбрасывают исключение `UnsupportedOperationException`. В классе `TagAdapter` есть конструктор с параметром типа `SimpleTag` (адаптируемый простой обработчик действия) и метод `getAdaptee()`, возвращающий адаптированный обработчик. В JSP 2.0 это всегда объект типа `SimpleTag`, хотя метод `getAdaptee()` возвращает значение типа `Tag`. Метод `TagAdapter.getParent()` всегда возвращает `getAdaptee().getParent()`, при этом результат будет соответствовать либо классическому обработчику, либо адаптеру простого обработчика.

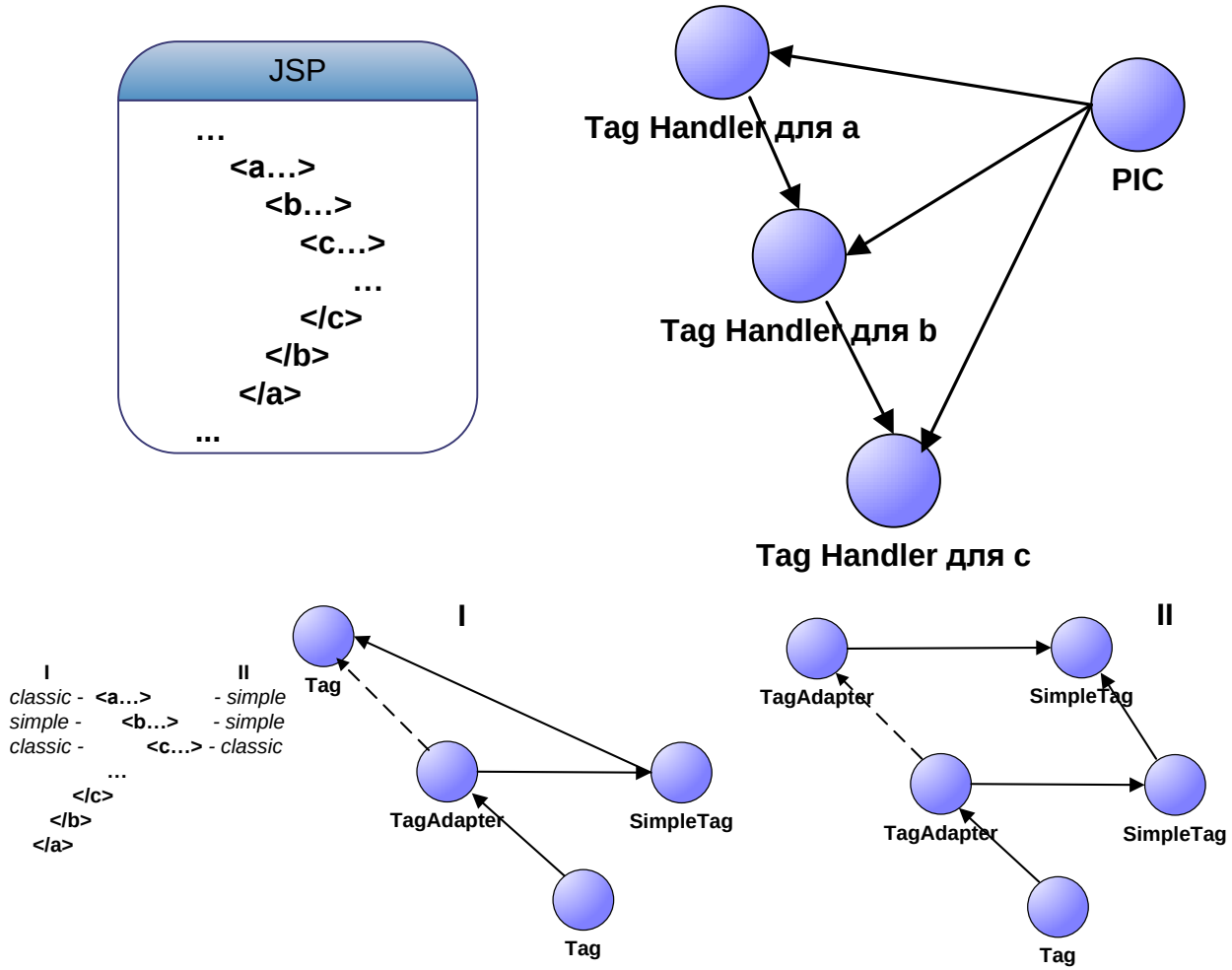


Рис. 2. Структура обработчиков действий на этапе выполнения

Класс SimpleTagSupport

Класс `SimpleTagSupport` представляет собой базовую реализацию интерфейса `SimpleTag`, упрощающую разработку простых обработчиков действий.

Данный класс при инициализации запоминает контекст страницы и обработчик родительского действия в закрытых полях, впоследствии данные объекты можно получить с помощью методов `getJspContext()` и `getParent()`.

Класс `SimpleTagSupport` содержит полезный метод `findAncestorWithClass()`. Этот метод обеспечивает поиск обработчика указанного типа для действия, в теле которого прямо или косвенно использовано данное действие. Поиск выполняется, начиная с указанного обработчика. Также методу передается тип искомого обработчика.

Метод `findAncestorWithClass()` использует метод `getParent()` для навигации по дереву обработчиков действий, структура которого соответствует структуре элементов действий в текущей JSP-странице. Навигация по родительским объектам выполняется до тех пор, пока тип очередного обработчика не будет совпадать с указанным, или очередной вызов `getParent()` не вернет `null`. Метод `findAncestorWithClass()` часто используют при реализации взаимодействующих действий (cooperating actions).

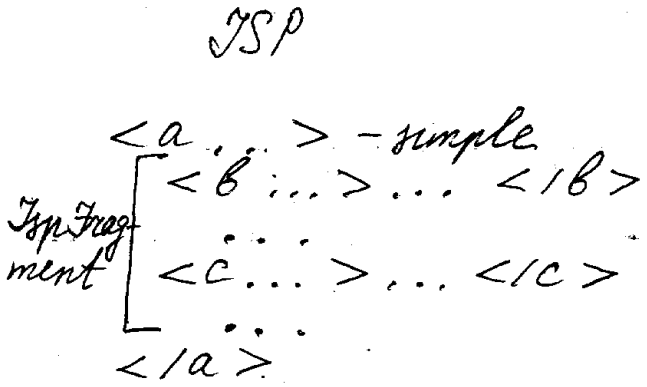
Класс JspFragment

Объекты этого класса используются для представления фрагментов JSP-кода в двух случаях:

1) когда атрибут действия описан в TLD как фрагмент (то есть элемент `fragment` указан для него со значением `true`, либо тип атрибута указан как `javax.servlet.jsp.tagext.JspFragment`), и его значение формируется с помощью стандартного действия `<jsp:attribute>`;
2) когда информация о теле действия передается простому обработчику действия.

Фрагмент JSP-кода, представленный объектом `JspFragment`, может вычисляться обработчиком явно с помощью метода `invoke()`. Результат вычисления записывается в объект типа `java.io.Writer`, передаваемый в качестве параметра метода.

Любой экземпляр `JspFragment` имеет доступ к контексту страницы, к которой он относится (объекту типа `JspContext`). Связывание контекста страницы с объектом `JspFragment` в спецификации не регламентируется. Получить связанный с фрагментом контекст страницы можно методом `getJspContext()`. Именно этот контекст будет использован для вычисления фрагмента, например, если в его составе используются действия, определенные программистом.



Кроме того, из контекста страницы можно получить объект `out`, который будет использован методом `JspFragment.invoke()`, если передать ему пустую ссылку на объект типа `Writer`.

Если тело действия описано в TLD как `tagdependent`, то при каждом вызове метода `JspFragment.invoke()` для соответствующего тела действия, оно будет выводиться в том самом виде, в каком оно записано в JSP. Если же тело действия описано как `scriptless`, то перед выводом оно будет вычисляться. Фрагменты, соответствующие атрибутам, вычисляются всегда.

Если при вычислении JSP-фрагмента вычислялись также использованные в его составе действия, и какой-либо из обработчиков действий вернул `SKIP_PAGE` из метода `doEndTag()` или выбросил исключение `SkipPageException` из метода `doTag()`, то метод `JspFragment.invoke()` в этом случае выбрасывает исключение `SkipPageException`. Обработчик действия, вызвавший метод `invoke()`, в свою очередь должен повторно выбросить исключение `SkipPageException`.

Интерфейс TryCatchFinally.

Это необязательный интерфейс, который может быть реализован обработчиком действия любого типа для обработки исключений, возникающих в ходе выполнения действия. В интерфейсе определено два метода:

`doCatch()` – вызывается в случае, если при выполнении действия (в том числе при вычислении тела) возникло исключение, и обработчик действия его не перехватил. В качестве параметра передается объект исключения типа `java.lang.Throwable`;

`doFinally()` – вызывается по окончании выполнения действия вне зависимости от того, возникло исключение или нет. Если исключение не возникло, метод вызывается после метода `doEndTag()`, иначе – после метода `doCatch()`.

Методы интерфейса `TryCatchFinally` не вызываются, если исключение возникло в set-методе для атрибута.

В результате трансляции действия, обработчик которого реализует интерфейс `TryCatchFinally`, получается примерно такой код:

```
Tag th = …;
th.setPageContext(pageContext);
th.setParent(…);
th.set…(…);
…
try {
      th.doStartTag();
      …
      th.doEndTag();
} catch (Throwable t) {
      th.doCatch(t);
} finally {
      th.doFinally();
}
```

Тэг-файлы

Тэг-файлы позволяют создавать действия, определяемые программистом, с использованием синтаксиса JSP. Это значит, что авторы JSP-страниц, не владеющие языком Java, могут создавать библиотеки действий. Кроме того, тэг-файлы упрощают разработку действий, основной функцией которых является генерация фрагмента ответа.

Тэг-файл должен иметь расширение `.tag` (в стандартном синтаксисе) или `.tagx` (в XML-синтаксисе). Тэг-файл может включать сегменты, для которых рекомендуется расширение `.tagf`.

Синтаксис тэг-файлов идентичен синтаксису JSP-страниц за следующими исключениями:

- Некоторые директивы недоступны или их использование ограничено, а также есть особые директивы для тэг-файлов.
- Стандартные действия `<jsp:invoke>` и `<jsp:doBody>` могут использоваться только в тэг-файлах.

Каждому тэг-файлу, определенному в веб-приложении, соответствует обработчик действия, доступный в JSP-страницах и других тэг-файлах. Веб-сервер может скомпилировать тэг-файл в Java-класс обработчика действия, либо интерпретировать его – спецификация отдает решение этого вопроса на откуп разработчикам веб-сервера, но задает семантику выполнения обработчика действия, определенного как тэг-файл.

Действия, определенные программистом в виде тэг-файлов, могут размещаться в одном из следующих мест:

1) В каталоге `/META-INF/tags/` (или его подкаталоге) JAR-файла, помещенного в каталог `/WEB-INF/lib/` веб-приложения. В этом случае определяемые тэг-файлами действия должны быть описаны в TLD.

2) В каталоге `/WEB-INF/tags/` веб-приложения (или его подкаталоге). В этом случае не нужно описывать соответствующие действия в TLD, что значительно упрощает разработку действий для конкретного веб-приложения. Веб-сервер неявно создает отдельную библиотеку действий для каталога `/WEB-INF/tags/` и каждого его подкаталога. Такая библиотека действий может быть импортирована с помощью атрибута `tagdir` директивы `taglib`. Название действия в рамках данной библиотеки совпадает с именем тэг-файла.

Одним из вариантов использования тэг-файлов является замена ими включения файлов с помощью директивы `include` или стандартного действия `<jsp:include>`. Данное решение обладает следующими преимуществами:

- Вызов действия, определенного в виде тэг-файла, выглядит компактнее комбинации действий `<jsp:include>` и `<jsp:param>`.
- Действия, определяемые программистом, можно вкладывать друг в друга.
- В теле действия, реализованного тэг-файлом, а также в фрагментах нельзя использовать скриптовые элементы, что облегчает сопровождение JSP-страниц.
- Параметры тэг-файла (атрибуты) всегда декларируются, в отличие от параметров включаемых JSP-страниц. Кроме того, для действия можно установить ограничения на значения атрибутов (тип, обязательность) и содержимое тела (отсутствует или не допускает использования скриптовых элементов). Веб-сервер осуществляет необходимые преобразования типов для атрибутов действия, тогда как при включении страницы все параметры представляют собой строки и преобразование типов необходимо выполнять явно.
- Переменные действия, объявленные в тэг-файле, автоматически экспортируются и синхронизируются с переменными JSP-страницы, что обеспечивает взаимодействие между вызываемым тэг-файлом и вызывающей страницей. При включении JSP-страниц межстраничное взаимодействие реализуется вручную.

Таким образом, если с включаемой JSP-страницей нужно взаимодействовать (передавать параметры и/или получать результаты), то ее преобразование в тэг-файл улучшит архитектуру веб-приложения. В противном случае это только замедлит обработку запроса.

Директивы для тэг-файлов

В тэг-файлах нельзя использовать директиву `page`, так как тэг-файл не является JSP-страницей; вместо нее используется директива `tag`. Директивы `include` и `taglib` действуют так же, как в JSP-страницах. Для описания атрибутов и переменных действий в тэг-файлах введены директивы `attribute` и `variable`, соответственно.

Директива `tag`. используется для определения характеристик действия, поэтому ее атрибуты аналогичны вложенным элементам элемента `<tag>` в TLD. Название действия совпадает с именем тэг-файла и не может быть изменено директивой `tag`. Наиболее важные атрибуты директивы `tag`:

- `body-content` – устанавливает тип тела действия, допустимые значения: `empty`, `tagdependent` или `scriptless` (по умолчанию).
- `dynamic-attributes` – указывает, что действие поддерживает динамические атрибуты. Значение данного атрибута указывает атрибут контекста страницы, в котором сохраняется карта (`Map`), содержащая названия и значения динамических атрибутов.
- `import` – аналогичен атрибуту import директивы page, импортирует указанные классы в класс, реализующий обработчик действия для тэг-файла.

Пример директивы `tag`, определяющей действие с типом тела scriptless и поддержкой динамических атрибутов:

```
<%@ tag body-content="scriptless" dynamic-attributes="dyn" %>
```

Директива `attribute` аналогична элементу `<attribute>` в TLD и определяет атрибут действия. В данной директиве используются следующие атрибуты:

- `name` – обязателен, указывает название атрибута действия, уникальное в рамках действия. Название атрибута действия не должно совпадать с именем переменной действия, задаваемым атрибутом `name-given` директивы `variable`.

- `required`, `fragment`, `rtexprvalue`, `type` – аналогичны одноименным элементам, вложенным в элемент `<attribute>` TLD (см. описание выше). Единственное отличие состоит в том, что по умолчанию атрибуты действия, заданного тэг-файлом, допускают динамические значения (атрибут `rtexprvalue` по умолчанию равен `true`).

Пример объявления обязательного атрибута целочисленного типа, допускающего только статические значения:

```
<%@ attribute name="x" required="true" rtexprvalue="false"
        type="java.lang.Integer" description="The first operand" %>
```

Пример объявления необязательного атрибута, принимающего в качестве значения фрагмент JSP-кода:

```
<%@ attribute name="prompt" fragment="true" %>
```

Директива `variable` аналогична элементу `<variable>` в TLD и определяет переменную, порождаемую действием. В данной директиве используются следующие атрибуты:

- `name-given`, `name-from-attribute`, `variable-class`, `declare`, `scope` – аналогичны одноименным элементам, вложенным в элемент `<variable>` TLD (см. описание выше).
- `alias` – определяет атрибут контекста тэг-файла для хранения значения переменной. Веб-сервер синхронизирует значение данного атрибута с переменной, имя которой задано атрибутом `name-from-attribute` этой же директивы. С точки зрения разработчика атрибут `alias` определяет псевдоним для переменной действия, название которой неизвестно в момент создания тэг-файла и определяется только на этапе трансляции JSP-страницы, использующей данное действие.

Пример объявления переменной целочисленного типа с названием `sum` и областью видимости `NESTED`:

```
<%@ variable name-given="sum" variable-class="java.lang.Integer"
        scope="NESTED" description="The sum of the two operands" %>
```

Пример объявления переменной строкового типа с псевдонимом `result`; название переменной в вызывающей JSP-странице берется из атрибута действия с названием `var`, который также необходимо определить в тэг-файле:

```
<%@ attribute name="var" required="true" rtexprvalue="false"%>
<%@ variable name-from-attribute="var" alias="result" %>
```

Встроенные объекты тэг-файлов

Разработчики тэг-файлов имеют доступ к встроенным объектам, которые можно использовать в выражениях и скриптлетах. Встроенные объекты `request`, `response`, `session`, `application`, `out` и `config` аналогичны одноименным встроенным объектам JSP-страниц. Помимо них определен встроенный объект `jspContext` типа `javax.servlet.jsp.JspContext` и контекста `page`, который представляет собой контекст данного тэг-файла.

Стандартные действия для тэг-файлов

Для тэг-файлов определены стандартные действия `<jsp:invoke>` и `<jsp:doBody>`.

Действие `<jsp:invoke>` предназначено для вызова фрагмента JSP-кода, результат обработки которого выводится в ответ или сохраняется в контекстной переменной (атрибуте некоторого контекста). Атрибуты действия:

- `fragment` – обязателен, указывает название атрибута действия, в котором передается фрагмент JSP-кода (у этого атрибута действия свойство `fragment` должно быть равно `true`).
- `var` – название контекстной переменной, в которой результат вызова фрагмента сохраняется в виде строки.
- `varReader` – название контекстной переменной, в которой результат вызова фрагмента сохраняется в виде объекта `java.io.Reader`.
- `scope` – контекст, в котором сохраняется переменная.

Если при вызове действия не указан ни один из атрибутов `var` и `varReader`, то результат вызова фрагмента выводится в ответ, формируемый тэг-файлом.

Действие `<jsp:doBody>` аналогично действию `<jsp:invoke>`, но выполняет вызов JSP-фрагмента, представляющего тело действия. Поэтому атрибуты данного действия те же, что у действия `<jsp:invoke>`, за исключением атрибута `fragment`.

Возврат результатов из тэг-файла в JSP-страницу

В тэг-файле доступны те же атрибуты контекстов запроса, сессии и веб-приложения, что и в вызывающей его JSP-странице. Но в тэг-файле используется собственный контекст страницы, отличный от контекста страницы, вызывающей действие. Поэтому атрибуты контекстов запроса, сессии и веб-приложения можно считать «глобальными переменными», а атрибуты контекста страницы – «локальными переменными».

Поскольку у каждого тэг-файла имеется собственный контекст страницы, локальные переменные страницы, вызывающей действие, недоступны в вызываемом тэг-файле, и наоборот. Для передачи параметров вызываемому действию JSP-страница может использовать атрибуты действия и глобальные переменные. Для каждого атрибута действия веб-сервер создает в тэг-файле локальную переменную (атрибут контекста страницы), доступную в EL-выражениях. Таким образом, атрибуты действия действуют как параметры, передаваемые по значению, а глобальные переменные – как параметры, передаваемые по ссылке. Если тэг-файл изменяет значение глобальной переменной, то это изменение доступно и в вызывающей JSP-странице. Недостаток использования глобальных переменных заключается в отсутствии стандартных средств документирования таких зависимостей тэг-файла от своего окружения.

Тэг-файл может «экспортировать» свои локальные переменные с помощью директивы `variable`. Установить значение переменной действия можно с помощью действия `<c:set>`. Веб-сервер создает другую переменную с указанным именем в вызывающей JSP-странице, которая инициализируется значением соответствующей ей переменной в тэг-файле. Изменение значения переменной в JSP-странице не влияет на переменную в тэг-файле. Но обратное неверно – веб-сервер синхронизирует переменную в тэг-файле с переменной в JSP-странице в соответствии с правилами, зависящими от области видимости переменной действия.

Если у переменной действия область видимости `AT_END`, то переменная в JSP-странице инициализируется значением переменной в тэг-файле после выполнения действия. Если область видимости – `AT_BEGIN`, то переменная в JSP-странице обновляется перед вызовом в тэг-файле действий `<jsp:doBody>` и `<jsp:invoke>`, а также после выполнения тэг-файла. Если же область видимости – `NESTED`, то переменная в JSP-странице обновляется перед вызовом в тэг-файле действий `<jsp:doBody>` и `<jsp:invoke>`, а после выполнения тэг-файла восстанавливается значение этой переменной, которое она имела перед вызовом тэг-файла.

Необходимость синхронизации значений переменных перед вызовом в тэг-файле действий `<jsp:doBody>` и `<jsp:invoke>` объясняется тем, что JSP-фрагмент имеет доступ к переменным контекста той страницы (или тэг-файла), в которой он определен.

Обратите внимание на то, что переменные, порождаемые обработчиком действия в виде тэг-файла, нельзя использовать в скриптовых выражениях и скриптлетах, так как для них не создаются локальные переменные в методе `_jspService()` класса PIC.

Пример действия, определенного программистом

Действие `<sample:userInRole>` позволяет определить, находится ли пользователь в одной из указанных ролей. Результат проверки сохраняется в переменной `isInRole` контекста страницы. Если проверка успешна, то тело действия вычисляется и вставляется в ответ. Атрибуты действия:

- `role` – обязательный, содержит список названий ролей, перечисленных через запятую.

Пример использования действия в JSP-странице:

```
<sample:userInRole role="admin,manager"/>
<c:if test="${isInRole}">
    <th>Email</th>
</c:if>
```

Заголовок столбца таблицы будет выведен только в том случае, если пользователь находится в роли `admin` или `manager`. Такого же результата можно было достичь с использованием тела действия:

```
<sample:userInRole role="admin,manager">
    <th>Email</th>
</sample:userInRole>
```

1. Классический обработчик действия

Определение действия в TLD при реализации классическим обработчиком:

```
<tag>
    <name>userInRole</name>
    <tag-class>tagext.UserInRoleClassicHandler</tag-class>
    <body-content>JSP</body-content>
    <attribute>
      <name>role</name>
      <required>true</required>
      <rtexprvalue>true</rtexprvalue>
      <type>java.lang.String</type>
    </attribute>
    <variable>
      <name-given>var</name-given>
      <variable-class>java.lang.Boolean</variable-class>
      <scope>AT_END</scope>
    </variable>
</tag>
```

Исходный код классического обработчика действия:

```
package tagext;

import java.io.IOException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.JspWriter;
import javax.servlet.jsp.PageContext;
import javax.servlet.jsp.tagext.BodyContent;
import javax.servlet.jsp.tagext.BodyTagSupport;

public class UserInRoleClassicHandler extends BodyTagSupport {

    // атрибуты действия
    private java.lang.String role;
    private java.lang.String var;
```

```
    private java.lang.String scope;
    // промежуточные данные - соответствие пользователи требуемой роли
    private boolean isInRole;

    public UserInRoleClassicHandler() {
        super();
    }

    /** обработка начального тэга */
    public int doStartTag() throws JspException, JspException {
        isInRole = false;
        String[] roles = role.split(",");
        for (String aRole : roles)
            if (((HttpServletRequest) pageContext.getRequest())
                    .isUserInRole(aRole)) {
                isInRole = true;
                break;
            }

        if (isInRole) {
            return EVAL_BODY_INCLUDE;
        } else {
            return SKIP_BODY;
        }
    }

    /** обработка завершающего тэга */
    public int doEndTag() throws JspException, JspException {
        pageContext.setAttribute("isInRole", isInRole);
        return EVAL_PAGE;
    }

    // методы для установки значений атрибутов действия
    public void setRole(java.lang.String value) {
        this.role = value;
    }

    public void setVar(java.lang.String value) {
        this.var = value;
    }

    public void setScope(java.lang.String value) {
        this.scope = value;
    }

}
```

2. Простой обработчик действия

Определение действия в TLD при реализации простым обработчиком незначительно отличается от приведенного выше для реализации классическим обработчиком – в теле действия, обрабатываемого простым обработчиком, нельзя использовать скриптовые элементы, поэтому изменяется тип тела действия:

```
<tag>
    <name>userInRole</name>
    <tag-class>tagext.UserInRoleHandler</tag-class>
    <body-content>scriptless</body-content>
    <!—атрибут и переменная действия, см. выше -->
    ...
</tag>
```

Исходный код простого обработчика действия:

```
package tagext;
```

```
import javax.servlet.http.HttpServletRequest;
import javax.servlet.jsp.PageContext;
import javax.servlet.jsp.tagext.*;
import javax.servlet.jsp.JspWriter;
import javax.servlet.jsp.JspException;

public class UserInRoleHandler extends SimpleTagSupport {

    // атрибуты действия
    private java.lang.String role;
    private java.lang.String var;
    private java.lang.String scope;

    /** обработка действия */
    public void doTag() throws JspException {
        JspWriter out=getJspContext().getOut();
        try {
            if (!(getJspContext() instanceof PageContext))
                return;
            PageContext ctx = (PageContext) getJspContext();
            boolean isInRole = false;
            String[] roles = role.split(",");
            for (String aRole : roles)
                if (((HttpServletRequest) ctx.getRequest())
                        .isUserInRole(aRole)) {
                    isInRole = true;
                    break;
                }

            if (isInRole) {
                JspFragment f=getJspBody();
                if (f != null) f.invoke(out);
            }

            ctx.setAttribute("isInRole", isInRole, PageContext.PAGE_SCOPE);

        } catch (java.io.IOException ex) {
            throw new JspException(ex.getMessage());
        }

    }

    // методы для установки значений атрибутов действия
    // см. выше
    ...
}
```

3. Обработчик действия в виде тэг-файла

Для реализации действия `<sample:userInRole>` в виде тэг-файла немного изменим спецификацию действия:

- Атрибут `var` является обязательным и представляет собой название переменной контекста страницы, в которую помещается результат проверки.
- Если проверка неуспешна, то вычисляется и вставляется в ответ фрагмент JSP-кода, передаваемый в атрибуте `failMessage`.

Пример использования модифицированного действия `<sample:userInRole>` в JSP-странице:

```
<sample:userInRole role="admin,manager" var="isAdmin">
    <jsp:attribute name="failMessage">
        Output is limited due to authorization constraints
    </jsp:attribute>
```

```
    <jsp:body>
        <p>Admin or Manager</p>
    </jsp:body>
</sample:userInRole>
```

Исходный код обработчика действия в виде тэг-файла:

```
<%@ tag pageEncoding="UTF-8"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<%@ attribute name="role" required="true"%>
<%@ attribute name="var" required="true" rtexprvalue="false"%>
<%@ attribute name="failMessage" fragment="true" %>
<%@ variable name-from-attribute="var" alias="result"
        variable-class="java.lang.Boolean" scope="AT_END"%>

<c:set var="result" value="${false}"/>
<c:forTokens var="aRole" items="${role}" delims=",">
    <c:if test='<%= request.isUserInRole(
              (String) jspContext.findAttribute("aRole"))%>'>
        <c:set var="result" value="${true}"/>
    </c:if>
</c:forTokens>
<c:choose>
    <c:when test="${result}">
        <jsp:doBody/>
    </c:when>
    <c:otherwise>
        <jsp:invoke fragment="failMessage"/>
    </c:otherwise>
</c:choose>
```

Данный пример тэг-файла демонстрирует использование переменной, порождаемой действием, а также обработку JSP-фрагментов, представленных телом и аргументом действия.